Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)

Jurriaan Hage (editor)

Technical Report UU-CS-2010-020
August 2010

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Preface

The 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010) takes place at Avifauna in Alphen aan den Rijn, the Netherlands from September 1 to 3, 2010. It signifies the return of IFL to Europe after a first successful visit to the United States where it was hosted by Seton Hall University in South Orange, New Jersey. IFL 2010 is hosted by the Software Technology group of Utrecht University's Department of Information and Computing Sciences. At the time of writing, the symposium had 58 registered participants from Belgium, Brasil, Denmark, Estonia, France, Germany, Greece, Hungary, the Netherlands, Poland, Spain, Sweden, the United Kingdom and the United States of America.

The goal of the IFL symposia is to bring together researchers actively engaged in the implementation and application of functional and function-based programming languages. It is a venue for researchers to present and discuss new ideas and concepts, works in progress, and publication-ripe results.

Following the IFL tradition, there is a post-symposium review process to produce formal proceedings which will be published by Springer Verlag in the Lecture Notes in Computer Science series. All participants in IFL 2010 were invited to submit either a draft paper or an extended abstract describing work to be presented at the symposium. The submissions were screened by the program committee chair to make sure they are within the scope of IFL. Submissions appearing in the draft proceedings are not peer-reviewed publications. After the symposium, authors have the opportunity to incorporate the feedback from discussions at the symposium into their paper and may submit a revised full article for the formal review process. These revised submissions will be reviewed by the program committee using prevailing academic standards.

The IFL 2010 program consists of 39 presentations and one invited talk. The contributions in this volume are ordered according to the intended schedule of presentation (see overleaf). In order to make IFL 2010 as accessible as possible, we have not insisted on any particular style or length for the submissions. Such rules only apply to the version submitted for post-symposium reviewing.

It is clear this year that concurrency and parallelism are very popular topics: there are three full sessions devoted to this topic, with the focus ranging from Software Transactional Memory to FPGAa and GPGPUs. Other topics are static analysis and type systems (two sessions), libraries and DSLs (one session), semantics (one session), compilers (one session) and event and GUI programming (one session). Functional programming languages that are represented at the symposium include Clean, Erlang, F#, Haskell, Hume, JavaScript, SAC, Scheme/Racket, and Timber, with the majority of papers dealing with Haskell.

As is usual for IFL, the program last three days with a social event and an invited talk. The invited talk will be given by Johan Nordlander of Luleøa University, who discusses the craft of building applications with Timber, a progamming language that claims to be purely functional, classically object-oriented and inherently concurrent at the same time, while also fostering a purely reactive model of interaction that naturally lets its real-time behavior be controlled by declaration.

The social event takes place on September 2 and consists of two parts: a cultural part in the city of Utrecht and, in the evening, a buffet dinner during a boat trip through the waters of Zuid-Holland.

We are grateful to many people for their help in preparating for IFL 2010. We were fortunate to use the portal of the UU Summer Schools; we thank in particular Marc Gebuis and Roos Nieuwenhuizen for their continuing support. Avifauna and in particular Marit van der Louw, Suzanne Oldenburg and Victor Galjé helped us with the local planning.

From the Department of Information and Computing Sciences, Doaitse Swierstra, Wilke Schram, Marinus Veldhorst, Edith Stap, Frans Wiering, Geraldine Leebeek, Martijn Dekker and Corine Jolles have contributed to the organisation of IFL 2010. I want to thank in particular the PhD students who helped with all the preparations for IFL: Sean Leather, José Pedro Magalhães and Jan Rochel.

I want to thank the Programme Committee for accepting my invitation and helping me make some of the more important decisions. However, most of their work still awaits them. Special

thanks are due to Marco Morazán, last year's chair, for letting us build on his work, wisdom and experience.

We gratefully ackowledge the financial support of the Department of Information and Computing Sciences. We were extremely happy to find that Microsoft Research was willing to sponsor IFL 2010, allowing us to decrease the student registration fees substantially. Finally, the grant we obtained from the Koninklijke Nederlandse Academie van Wetenschappen (KNAW) gave us further financial room to make IFL 2010 even more accessible to participants.

Jurriaan Hage
Symposium chair of IFL 2010

Sponsoring institutions

The symposium was supported by Microsoft Research, the Koninklijke Nederlandse Academie van Wetenschappen (KNAW) and the Department of Information and Computing Sciences of Utrecht University, the Netherlands.

KONINKLIJKE NEDERLANDSE
AKADEMIE VAN WETENSCHAPPEN

Universiteit Utrecht

Organisation

IFL 2010 was organised by members of the Software Technology group of the Department of Information and Computing Sciences of Utrecht University in the Netherlands.

Executive committee

Program and organisation chair:	Jurriaan Hage
Organising committee:	Sean Leather
	José Pedro Magalhães
	Jan Rochel

Programme committee

Jost Berthold	University of Copenhagen	Denmark
Olaf Chitil	University of Kent	UK
John Clements	California Polytechnic State University	USA
Matthew Fluet	Rochester Institute of Technology	USA
Andy Gill	Kansas University	USA
Jurriaan Hage	University of Utrecht	Netherlands
Bastiaan Heeren	Open University of the Netherlands	Netherlands
Ralf Hinze	University of Oxford	UK
John Hughes	Chalmers University of Technology	Sweden
Yukiyoshi Kameyama	University of Tsukuba	Japan
Gabriele Keller	University of New South Wales	Australia
Pieter Koopman	Radboud University Nijmegen	Netherlands
Luc Maranget	INRIA	France
Simon Marlow	Microsoft Research	UK
Marco T. Morazán	Seton Hall University	USA
Rex Page	University of Oklahoma	USA
Ricardo Peña	Universidad Complutense de Madrid	Spain
Sven-Bodo Scholz	University of Hertfordshire	UK
Tom Schrijvers	Catholic University of Leuven	Belgium
Don Stewart	Galois	USA
Wouter Swierstra	Vector Fabrics	Netherlands
Don Syme	Microsoft	UK
Peter Thiemann	University of Freiburg	Germany
Phil Trinder	Heriott-Watt University	Scotland
Janis Voigtländer	University of Bonn	Germany
Viktória Zsók	Eötvös Loránd University	Hungary

Programme

September 1, 2010

8.00 - 9.00	**Registration**
9.00 - 9.20	**Welcome**
9.20 - 10.40	*Static Analysis I*
	Session Chair: Tom Schrijvers
	1. *Size Invariants and Ranking Functions Synthesis in a Functional Language*
	Ricardo Peña and Agustin D. Delgado
	2. *Data-Driven Detection of Catamorphisms Towards Problem Specific Use of Program Schemes for Inductive Program Synthesis*
	Martin Hofmann
	3. *Strictness Optimization for Higher-Order Functions in a Typed Intermediate Language*
	Tom Lokhorst, Atze Dijkstra and Doaitse Swierstra
	4. *Untyped General Polymorphic Functions*
	Martin Pettai
10.40 - 11.20	Break (adjacent room)
11.20 - 12.40	*Events and Workflow*
	Session Chair: Rex Page
	1. *The Usual Tasks: A Library for Ad-Hoc Work in iTasks*
	Bas Lijnse, Rinus Plasmeijer and Erik Crombag
	2. *iTask as a new paradigm to building GUI applications*
	Peter Achten, Rinus Plasmeijer and Steffen Michels
	3. *Multiple-Occurrence I/O*
	Gergely Patai
	4. *Gin: Graphical iTask Notation*
	Peter Achten, Jeroen Henrix and Rinus Plasmeijer
12.40 - 14.00	Lunch (adjacent room)
14.00 - 15.20	*Compilers and Interpreters*
	Session Chair: Bastiaan Heeren
	1. *Towards Dependently-Typed Attribute Grammars*
	Arie Middelkoop, Atze Dijkstra and Doaitse Swierstra
	2. *Mapping Interpreters onto Runtime Support*
	Stijn Timbermont
	3. *Hiding State in ClaSH Hardware Descriptions*
	Marco Gerards, Christiaan Baaij, Jan Kuper and Matthijs Kooiman
	4. *Implementing a Non-Strict Purely Functional Language in JavaScript?*
	Eddy Bruel and Jan Martin Jansen
15.20 - 16.00	Break (adjacent room)
16.00 - 17.40	*Concurrency I*
	Session Chair: Viktoria Zsók
	1. *Concurrent Non-Deferred Reference Counting on the Microgrid: First Experiences*
	Stephan Herhut and Sven-Bodo Scholz
	2. *First Results from Auto-Parallelising SAC for GPGPUs*
	Jing Guo, Sven-Bodo Scholz, Jeyarajan Thiyagalingam
	3. *Improving your CASH flow: the Computer Algebra SHell*
	Christopher Brown, Kevin Hammond, Jost Berthold, Hans-Wolfgang Loidl
	4. *mHume for parallel FPGA*
	Abdallah Al Zain, Wim Vanderbauwhede and Greg Michaelson
	5. *The Essence of Synchronisation in Asynchronous Data Flow Programming*
	Clemens Grelck

September 2, 2010

9.00 - 10.20	*Concurrency II* Session Chair: 1. *An Executable Semantics for D-Clean* Viktória Zsók, Rinus Plasmeijer and Pieter Koopman 2. *Dependency Graphs for Parallelizing Erlang Programs* Melinda Tóth, István Bózó, Zoltán Horváth and Atilla Erdödi 3. *Counter Automata for Parameterised Timing Analysis of Box-Based Systems* Christoph Herrmann and Kevin Hammond 4. *Introducing the PilGRIM: a Pipelined Processor for Executing Lazy Functional Languages* Arjan Boeijink, Jan Kuper and Philip Hölzenspies
10.20 - 11.00	Break (adjacent room)
11.00 - 12.00	**Invited Talk** by Johan Nordlander: The craft of building with Timber Session Chair: Jurriaan Hage
12.00 - 13.30	Lunch (adjacent room)
14.00 - 23.30	*Social Event*

September 3, 2010

9.00 - 10.40 *Libraries and DSLs*
Session Chair: tba
1. *A Database Coprocessor for Haskell*
Jeroen Weijers, Torsten Grust, George Giorgidze and Tom Schreiber
2. *The Design and Implementation of Feldspar: an Embedded Language for Digital Signal Processing*
Mary Sheeran, Anders Persson, David Engdal, Josef Svenningsson and Koen Claessen
3. *Combinators for Local Search In Haskell*
David Senington and David Duke
4. *Modular Components with Monadic Effects*
Tom Schrijvers and Bruno Oliveira
5. *Experiences using F# for developing analysis scripts and tools over search engine query log data*
Stefan Savev and Peter Bailey

10.40 - 11.20 Break (adjacent room)

11.20 - 12.40 *Concurrency III*
Session Chair: tba
1. *A Comparison of Lock-based and Lock-free Taskpool Implementations in Haskell*
Michael Lesniak
2. *A high-level implementation of STM Haskell using the Transactional Locking II algorithm*
Andre Rauber Du Bois
3. *Twilight in Haskell - Software Transactional Memory with Safe I/O and Typed Conflict Management*
Annette Bieniusa, Arie Middelkoop and Peter Thiemann
4. *Towards Orthogonal Haskell Data Serialisation*
Jost Berthold

12.40 - 14.00 Lunch (adjacent room)

14.00 - 15.20 *Semantics*
Session Chair: tba
1. *From Bayesian Notation to Pure Racket, via Discrete Measure-Theoretic Probability in Lambda-ZFC*
Neil Toronto and Jay McCarthy
2. *On the relation of call-by-need and call-by-name in a natural semantics*
Lidia Sánchez-Gil, Mercedes Hidalgo Herrero and Yolanda Ortega Mallén
3. *Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq*
Dariusz Biernacki, Filip Sieczkowski and Malgorzata Biernacka
4. *Towards Strategies for Dataflow Programming*
Joaquin Aguado and Michael Mendler

15.20 - 16.00 Break (adjacent room)

16.00 - 17.40 *Static Analysis II*
Session Chair: Janis Voigtländer
1. *Extensible Pattern Matching in an Extensible Language*
Sam Tobin-Hochstadt
2. *Where are you going with those types?*
Vincent St-Amour, Matthias Felleisen, Matthew Flatt, Sam Tobin-Hochstadt
3. *Purity in Erlang*
Mihalis Pitidis and Konstantinos Sagonas
4. *Theory, Practice and Pragmatics of Fusion*
Thomas Harper, Daniel James, and Ralf Hinze
5. *Composing Reactive GUIs in F# with WebSharper*
Joel Björnson, Anton Tayanovskyy and Adam Granicz

Size Invariants and Ranking Functions Synthesis in a Functional Language *

Ricardo Peña
ricardo@sip.ucm.es

Agustin D. Delgado
elsmda@gmail.com

Universidad Complutense de Madrid, Spain

Abstract. The paper presents preliminary results in automatic inference of size invariants, and of ranking functions proving termination of functional programs, by adapting linear techniques developed for other languages. The results are promising and allow to solve some problems left open in previous works on automatic inference of safe memory bounds.
Keywords: functional languages, linear techniques, abstract interpretation, size analysis, ranking functions.

1 Introduction

In a previous work [17] we presented static analysis-based algorithms for inferring upper bounds to memory consumption of programs written in a first-order functional language. The technique used was abstract interpretation with the abstract domain being the complete lattice of monotonic functions $f : \mathbb{R}^{+^n} \to \mathbb{R}^+$ ordered by point-wise $\leq$. We showed some examples of applications and the bounds obtained were rather good in simple programs. For instance, we got precise linear heap and stack bounds for functions such as *merge*, *append*, and *split*, and quadratic over-approximations for the heap consumption of functions such as *mergesort* and *quicksort*. The algorithms were even able to infer a constant stack space for tail recursive functions.

A remarkable feature of the algorithms was that in some circumstances the abstract interpretation was *reductive* in the lattice, meaning that iterating the interpretation by introducing as hypothesis the previously inferred bound, obtained a tighter bound, and still a correct one.

Unfortunately, the work was incomplete because it needed some information to be introduced by hand for every particular program, namely the size of some local variables and an upper bound to the length of the longest call chain of recursive functions. These two problems deserve independent and complex analyses by themselves, and we decided to defer their solution. The algorithms were proved correct provided correct functions for this figures were given.

In [14] we approached one of these problems —inferring the longest recursive call chain— by translating our functional programs into a term rewriting

* Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), and S2009/TIC-1465 (PROMETIDOS).

system (TRS) and then using termination proofs of TRSs, based on dependency pairs and polynomial synthesis, for computing this bound. The first results were encouraging but the approach could not prove any bound for simple algorithms such as *mergesort* and *quicksort*. We felt that having a previous size analysis could probably improve the termination proofs.

In this paper we approach both size analysis and termination proofs by using linear techniques, which have been proved successful in analysing other kind of languages such as imperative, and even bytecode programs, and also logic ones. The main contribution of the paper is showing that these techniques have the same power in a first-order functional language than in any other paradigm, and that the only adaptations needed are applying some transformations to the source programs, and having some care when applying the techniques in a context different to that they were conceived in.

The plan of the paper is as follows: In Sec. 2 we do a brief survey of linear techniques applied to both size analysis and ranking function synthesis. Then, Sec. 3 presents the aspects of our functional language *Safe* relevant to this paper. Sec. 4 is devoted to obtaining size invariants of *Safe* programs, while Sec. 5 applies the well-known ranking function synthesis method by Podelski and Rybalchenko [18] to computing our bound to the longest call chain. Finally, Sec. 6 provides a short conclusion.

2 Linear Constraints Techniques

Abstract interpretation [10] is a very powerful static analysis technique able to infer a variety of properties in programs written in virtually any programming language. In functional languages it has been successfully applied to strictness and update avoidance analyses of lazy languages, to sharing and binding time analyses, and many others. The abstract domains are usually finite small ones when abstracting non-functional values, but they tend to grow exponentially when they are extended to higher-order values.

Polyhedral abstract domains have been extensively used in logic and imperative languages, but not so frequently in functional ones. These domains are useful when quantitative rather than qualitative properties are sought for, as it is the case of size or cost relations between variables. Since the seminal work of Cousot and Halbwachs [11], polyhedra have been used to analyse arithmetical linear relations between program variables. A convex polyhedron is a subset of $\mathbb{R}^n$ limited by hyperplanes. There exist at least two finite representations of convex polyhedra:

- By three sets, namely of vertexes, rays, and lines in a n-dimensional space.
- By a conjunction of linear constraints between n variables.

There are algorithms for translating these representations into each other, although their cost is exponential. A frequent (and also costly operation) is to compute the convex hull of two polyhedra, which is the minimum convex polyhedron containing both. It is associative and commutative. The advantage of

using linear constraints is that most of the interesting problems involving them are decidable. For instance, to know whether a set of constraints is satisfiable, or whether a constraint is implied by a set of other ones, to project a set of constraints over a subset of their variables, to compute the convex hull of two sets of constraints, to maximise or minimise a linear function with respect to a set of constraints, and some others.

Invariant synthesis In this context, an invariant is a linear relation between the variables involved in a loop, holding at the beginning of each iteration. An abstract interpretation for synthesising loop invariants starts by computing a polyhedron with the relations known at the beginning of the loop, and iterates calculating the convex hull between the polyhedron coming from the previous iteration and the one obtained by the transition relation of the loop body. After a few iterations, some relations will stabilise while some others will not. The first ones constitute the invariant. Several tools have been developed for obtaining these invariants (for instance ASPIC, see [12]), or giving the necessary infrastructure to compute them (as e.g. [4]).

In the logic programming field, Benoy and King [7] applied a similar technique to the inference of size relations between the arguments of a logic predicate. In a first step, the logic program is transformed into an abstract one on arithmetic variables, by replacing the original predicate arguments by their corresponding sizes. An abstract interpretation of the transformed program infers the invariant size relations between the arguments of the original program. The ascending chain (in the sense of set inclusion) of polyhedra obtained by the fixpoint algorithm may in principle be infinite, because some relations do not stabilise. A *widening* technique is used to eliminate these variant relations while the invariant ones are retained. Of course, if the invariant relations are not linear the algorithm does not obtain anything meaningful.

Ranking functions synthesis Detecting termination statically has attracted the attention of much research work. Given that this is an undecidable problem in general, the algorithms try to cover as many particular decidable cases as possible. One successful approach has been the work by Ben-Amram and his group, starting in the seminal paper [13], where in a first phase the program being analysed is transformed into a so-called *size-change graph*. This is the program call flow graph enriched with information about the arguments that strictly decrease at a call and those that may decrease or remain equal. This part of the analysis is outside of the proposed termination algorithms, and may be done by hand or by a previous size analysis. What is nice in the approach is that termination of size-change graphs is decidable, although the algorithm is exponential in the worst case (these programs are called *size-change terminating*, or SCT). However, by using benchmarks the authors convincingly show that this case is very unusual and that most of the time a polynomial algorithm suffices [6]. Moreover, in these cases they can synthesise a global ranking function ensuring that the program terminates, which can be checked (i.e. proved that it decreases

at each transition) in polynomial time. Synthesising such a function is however a NP-complete problem which they decide by using a SAT-solver [5].

Another successful line of research has been the synthesis of linear ranking functions. In [18], Podelski and Rybalchenko give a complete method to synthesise this kind of functions for simple while-loops in which the loop guard is a conjunction on linear expressions, and the loop body is a multiple assignment of linear expressions to variables. The kernel of the method is solving a set of linear constraints. This small piece can be the basis for inferring termination of more complex programs. In [19] they show that the union of simple well-founded relations (which in general is not well-founded) can prove termination of a program with nested loops provided this union is an invariant of the transition relation (in this case, the union is called *disjunctively well-founded*). Another successful method for analysing complex loops and synthesising linear ranking functions is [9]. In a recent work [2] the authors present a complete method for synthesising a global ranking function for a complex nested control flow structure, provided the ranking function has the form of a lexicographically ordered tuple containing linear expressions.

In [1], the authors claim to have used —although not many details are given— the Podelski and Rybalchenko's method as one of the steps for solving recurrence relations obtained by analysing Java bytecode programs. The idea is to use the ranking function as an upper bound to the recurrence depth (i.e. to the number of unfoldings required in the worst case to reach a non-recursive case). We will pursue this idea here for inferring an upper bound to the depth of the call tree of a recursive *Safe* function.

3 The *Safe* Functional Language

Safe is a first-order eager polymorphic functional language with a syntax similar to that of (first-order) Haskell, and with an unusual memory management system. Its main aim is to facilitate the compile-time inference of safe upper bounds to memory consumption. Its memory model is based on disjoint heap regions where data structures are built. The compiler infers the optimal way to assign data to regions, so that their lifetimes are as short as possible, compatible with allocating and deallocating regions by following a stack-based strategy. The region-based model has two benefits: (1) a garbage collector is not needed; (2) the compiler may compute an upper bound to the size of each region and to the whole heap. More information about *Safe*, its type system, and its memory inference algorithms can be found at [15–17].

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and follows with Hindley-Milner type inference, desugaring pattern matching into **case** expressions, **where** clauses into **let** expressions, collapsing several function-defining equations into a single one, and some other transformations.

As regions are not relevant to this paper, in Fig. 1 we show a simplified *Core-Safe*'s syntax where regions have been deleted. A program *prog* is a sequence of

$$
\begin{array}{lll}
prog \rightarrow \overline{data_i}; \overline{dec_j}; e & & \{\textit{Core-Safe} \text{ program}\} \\
dec \rightarrow f\ \overline{x_i}\ = e & & \{\text{recursive, polymorphic function definition}\} \\
e \rightarrow a & & \{\text{atom } a\text{: either a literal } c \text{ or a variable } x\} \\
\quad \mid a_1 \oplus a_2 & & \{\text{primitive operator application}\} \\
\quad \mid f\ \overline{a_i} & & \{\text{function application}\} \\
\quad \mid C\ \overline{a_i} & & \{\text{constructor application}\} \\
\quad \mid \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 & & \{\text{non-recursive, monomorphic } \textbf{let}\} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt_i} & & \{\text{case expression}\} \\
alt \rightarrow C\ \overline{x_i} \rightarrow e & & \{\text{case alternative}\}
\end{array}
$$

Fig. 1. Simplified *Core-Safe* syntax

```
split 0 ys      = ([], ys)
split n []      = ([], [])
split n (y:ys)  = (y:ys1,ys2)    where (ys1, ys2) = split (n-1) ys

merge [] ys  = ys
merge xs []  = xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys

msort []  = []
msort [x] = [x]
msort xs  = merge (msort xs1) (msort xs2)
            where (xs1,xs2) = split (length n / 2) xs
```

Fig. 2. *mergesort* algorithm in *Full-Safe*

possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. The abbreviation $\overline{x_i}$ stands for $x_1 \cdots x_n$, for some n.

In Fig. 2 we show a *Full-Safe* version of the *mergesort* algorithm, which we will use as running example throughout the paper. In Fig. 3 we show the translation to *Core-Safe* of two of its functions.

Our purpose is to analyse *Core-Safe* programs in order to infer invariant size relations between the arguments and results of each function, and upper bounds to the runtime size of the call tree unfolded when invoking a recursive function. As part of this process, it will be important to discover which sub-expressions contribute to the non-recursive (or base) cases of a recursive function, and which ones contribute to the recursive ones. Moreover, we will distinguish between mutually exclusive recursive calls, i.e. those which will never execute together at runtime (e.g. those occurring in different alternatives of a **case** expression), and sequential recursive calls (those with may execute one after the other).

An approximation to this property can be obtained by an algorithm separating textual calls occurring in different branches of a **case**, and putting together textual calls occurring in the sub-expressions e_1 and e_2 of a **let**. In Fig. 4 we show the algorithm $seqs_f$, written in a Haskell-like language, analysing this property for the *Core-Safe* expression of function-f's body. It returns a list of lists of qualified expressions, meaning this that a tag B (for base cases), or R (for recursive

```
merge x y = case x of
             []    -> y
             ex:x' -> case y of
                        []    -> x
                        ey:y' -> case ex <= ey of
                                   True  -> let z1 = merge x' y in ex:z1
                                   False -> let z2 = merge x y' in ey:z2
msort x = case x of
           []    -> []
           ex:x' -> case x' of
                     []  -> ex:[]
                     _:_ -> let n  = length x        in
                            let n2 = n/2             in
                            let (x1,x2) = split n2 x in
                            let z1 = msort x1        in
                            let z2 = msort x2        in
                            merge z1 z2
```

Fig. 3. functions *merge* and *msort* in *Core-Safe*

$$\textbf{data}\ \mathit{QExp} = B\ \mathit{Exp} \mid R\ \mathit{Exp} \mid \mathit{Var} := [\mathit{QExp}]$$

$$\begin{aligned}
&\mathit{seqs}_f :: \mathit{Exp} \rightarrow [[\mathit{QExp}]] \\
&\mathit{seqs}_f\ e = [[B\ e]] && \text{-- if } e \in \{c, x, a_1 \oplus a_2, g\ \overline{a_i}, C\ \overline{a_i}\} \\
&\mathit{seqs}_f\ (f\ \overline{a_i}) = [[R\ (f\ \overline{a_i})]] \\
&\mathit{seqs}_f\ (\textbf{case of}\ \mathit{alts}) = \mathit{concat}\ [\mathit{seqs}_f\ e \mid (C\ \overline{x_i} \rightarrow e) \in \mathit{alts}] \\
&\mathit{seqs}_f\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2) = [(x_1 :=\ s_1) : s_2 \mid s_1 \in \mathit{seqs}_f\ e_1,\ s_2 \in \mathit{seqs}_f\ e_2]
\end{aligned}$$

Fig. 4. Algorithm for extracting the base and recursive cases of a *Core-Safe* expression

ones) is added to each individual sub-expression. Each internal list represents a mutually exclusive way of executing the function, while the sub-expressions inside a list indicate a possible sequential execution of them.

In Fig. 5 we show the application of the algorithm to *merge* and *msort*. For *merge* x y we obtain four internal lists illustrating that there are two mutually exclusive base cases, and that the two recursive calls exclude each other. When applied to $msort(x)$ we obtain three internal lists illustrating that there are two base cases, and that the two recursive calls may be sequentially executed.

4 Size Invariants Inference

Following [7], in order to reason about sizes, the original program must first be translated into an abstract program in which data structures have been replaced by their corresponding sizes, and previously to that, a notion of *size* must be defined. For logic programs, the more frequently used ones are the list length for list arguments, the value for integer arguments, zero for any other atom, and the term size for the rest of variables. In our memory model we have a precise notion of size in terms of memory cells occupied by a data structure: each constructor application fits exactly in one cell. So, the size of a list is its length plus one because the empty list constructor needs an additional cell. Moreover, we have a

$$
\begin{aligned}
seqs_{merge} = {} & [[B\ y], [B\ x], \\
& [z_1 := [R\ (merge\ x'\ y)], B\ (e_x : z_1)], [z_2 := [R\ (merge\ x\ y'))], B\ (e_y : z_2)]]
\end{aligned}
$$

$$
\begin{aligned}
seqs_{msort} = {} & [[B\ [\,]], [B\ (e_x : [\,])], \\
& [n := [B\ (length\ x)], n_2 := [B\ (n/2)], (x_1, x_2) := [B\ (split\ n_2\ x)], \\
& z_1 := [R\ (msort\ x_1)], z_2 := [R\ (msort\ x_2)], B\ (merge\ z_1\ z_2)]]
\end{aligned}
$$

Fig. 5. Decomposition of *merge* and *msort* into base and recursive cases

$$
\begin{array}{l}
merge^S\ x\ y = \textbf{case}\ x\ \textbf{of} \\
\quad x = 1 \rightarrow\ y \\
\quad x \geq 2 \rightarrow\ \textbf{case}\ y\ \textbf{of} \\
\qquad y = 1 \rightarrow x \\
\qquad y \geq 2 \rightarrow \textbf{case}\ ?\ \textbf{of} \\
\qquad\quad T \rightarrow \textbf{let}\ z_1 = merge^S\ (x-1)\ y\ \textbf{in}\ z_1 + 1 \\
\qquad\quad F \rightarrow \textbf{let}\ z_2 = merge^S\ x\ (y-1)\ \textbf{in}\ z_2 + 1 \\
msort^S\ x = \quad \textbf{case}\ x\ \textbf{of} \\
\quad x = 1 \rightarrow\ 1 \\
\quad x \geq 2 \rightarrow\ \textbf{case}\ x\ \textbf{of} \\
\qquad x = 2 \rightarrow 2 \\
\qquad x \geq 3 \rightarrow\ \textbf{let}\ n = length^S\ x\ \textbf{in} \\
\qquad\quad \textbf{let}\ n_2 = n/2\ \textbf{in} \\
\qquad\quad \textbf{let}\ (x_1, x_2) = split^S\ n_2\ x\ \textbf{in} \\
\qquad\quad \textbf{let}\ z_1 = msort^S\ x_1\ \textbf{in} \\
\qquad\quad \textbf{let}\ z_2 = msort^S\ x_2\ \textbf{in} \\
\qquad\quad merge^S\ z_1\ z_2
\end{array}
$$

Fig. 6. Abstract size functions *merge* and *msort*

precise notion of *data structure*: it comprises the set of cells corresponding to its recursive spine. In this way, a list of n lists constitutes $n + 1$ independent data structures. Additionally we define:

- The size of an integer constant or variable is its value n.
- The size of a Boolean constant or variable is zero.

We will call *size programs*, or size functions, to the abstract programs or functions resulting from the size translation. If f is the original function, f^S will denote its size version. The size functions resulting from the translation of *merge* and *msort* of Fig. 3 are shown in Fig 6.

The next step consists of performing an abstract interpretation of the size functions. The abstract domain will be that of convex polyhedra ordered by set inclusion, represented in this paper by conjunctions of linear constraints. The *meet* operation, or greatest lower bound $\sqcap$, is the intersection of two polyhedra, and consists of just putting together the two sets of constraints. The *join* operation, or least upper bound $\sqcup$, is the convex hull of the two polyhedra. We have used the algorithm developed by Andy King et al [8] to compute convex hulls.

The algorithm we propose has been inspired by Benoy and King's algorithm [7] for analysing logic programs. It consists of the following steps:

1. The size functions obtained by translating a *Core-Safe* program are analysed in the order they occur in the file: from low-level ones not using any other function, to higher-level ones using those previously defined in the file.
2. For each size function, a set of invariant size relations between input arguments and output results are inferred.
3. These relations are kept in a global environment. When analysing the current function, say f^S, these relations are instantiated with the sizes of the actual arguments at each site where an already analysed function is called. These relations, together with the rest of relations inferred for f^S, are used to infer f^S's invariant relations.

Analysing the current function f^S consists of the following steps. In order to fix ideas, we will call $\overline{x_i}$ to f^S's formal arguments and z to its result (or $\overline{z_j}$ if it is a tuple).

1. Function $seqs_f$ of Fig. 4 is applied to the size function f^S in a similar way as it was applied to *Core-Safe* expressions.
2. As a result, a separate set of constraints for each of the code sequences is inferred. The base sequences are then separated from the recursive ones (recursive sequences can be identified by the presence of at least a recursive call to f^S).
3. The constraints of each base sequence are expressed in terms of $\overline{x_i}$ and $\overline{z_j}$. This can always be done by projecting a set of constraints with more variables to variables $\overline{x_i}$ and $\overline{z_j}$.
4. The constraints of each recursive sequence are expressed in terms of $\overline{x_i}$, $\overline{z_j}$, and of two sets of variables $\overline{x_i}^k$ and $\overline{z_j}^k$ for each recursive call k in the sequence. The variables $\overline{x_i}^k$ represent the input arguments sizes of that call, while the $\overline{z_j}^k$ represent its output sizes.

Then, a fixpoint algorithm for f^S is launched having the following steps:

1. The initial polyhedron is $P_{next} = B_1 \sqcup \ldots \sqcup B_n$ where $B_l(\overline{x_i}, \overline{z_j})$, $1 \leq l \leq n$, are the polyhedra of the base cases.
2. At each iteration, the variables of polyhedron $P_{next}(\overline{x_i}, \overline{z_j})$ are renamed, obtaining $Q_{prev} = P_{next}[\overline{x'_i}/\overline{x_i}, \overline{z'_j}/\overline{z_j}]$. The idea is that $Q_{prev}(\overline{x'_i}, \overline{z'_j})$ represents the constraints coming from the previous iterations.
3. Now, for each recursive sequence l, its constraints are enriched by adding the constraints coming from Q_{prev}, as many times n_l as recursive calls are in the sequence, by previously substituting the $\overline{x'_i}$ for the $\overline{x_i}^k$ and the $\overline{z'_j}$ for the $\overline{z_j}^k$, $1 \leq k \leq n_k$. Let us call $R_l(\overline{x_i}, \overline{z_j}, \overline{x_i}^1, \overline{z_j}^1, \ldots, \overline{x_i}^{n_l}, \overline{z_j}^{n_l})$ to the polyhedron resulting from the l-th sequence.
4. Each R_l is projected over the variables $\overline{x_i}, \overline{z_j}$ obtaining $RP_l(\overline{x_i}, \overline{z_j})$. If there are m recursive sequences, then the following polyhedron is computed:

$$P_{next}(\overline{x_i}, \overline{z_j}) = RP_1 \sqcup \ldots \sqcup RP_m \sqcup B_1 \sqcup \ldots \sqcup B_n$$

5. If $P_{next}[\overline{x'_i}/\overline{x_i}, \overline{z'_j}/\overline{z_j}] = Q_{prev}$ then stop; else go to (2).

$$
\begin{array}{l}
B_1^{merge} = \{x = 1,\ z = y\} \\
B_2^{merge} = \{x \geq 2,\ y = 1,\ z = x\} \\
R_1^{merge} = \{x \geq 2,\ y \geq 2,\ x' = x - 1,\ y' = y,\ z = 1 + z'\} \\
R_2^{merge} = \{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y - 1,\ z = 1 + z'\}
\end{array}
$$

$$
\begin{array}{l}
B_1^{msort} = \{x = 1,\ z = 1\} \\
B_2^{msort} = \{x = 2,\ z = 2\} \\
R_1^{msort} = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 = z'_1 + z'_2\}
\end{array}
$$

Fig. 7. Restrictions corresponding to the base and recursive cases of *merge* and *msort*

Iter.	$\mathbf{P}_{next}$	$\mathbf{R}_i$
1_{merge}	$\{x \geq 1,\ z + 1 = x + y\}$	$R_1 = \{x \geq 2,\ y \geq 2,\ x' = x - 1,\ y' = y,$ $z = 1 + z',\ z' + 1 = x' + y'\}$
		$R_2 = \{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y - 1,$ $z = 1 + z',\ z' + 1 = x' + y'\}$
2_{merge}	$\{x \geq 1,\ z + 1 = x + y\}$	
1_{msort}	$\{x \geq 1,\ x \leq 2,\ z = x\}$	$R_1 = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 =$ $z'_1 + z'_2,\ x'_1 \geq 1,\ x'_1 \leq 2,\ z'_1 = x'_1,$ $x'_2 \geq 1,\ x'_2 \leq 2,\ z'_2 = x'_2\}$
2_{msort}	$\{x \geq 1,\ x \leq 3,\ z = x\}$	$R_1 = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 =$ $z'_1 + z'_2,\ x'_1 \geq 1,\ x'_1 \leq 3,\ z'_1 = x'_1,$ $x'_2 \geq 1,\ x'_2 \leq 3,\ z'_2 = x'_2\}$
3_{msort}	$\{x \geq 1,\ x \leq 4,\ z = x\}$	

Fig. 8. Fixpoint iterations for *merge* an *msort*

In Fig. 7 we show the base and recursive sets of constraints inferred for *merge* and *msort* before launching the fixpoint algorithm. In the *merge* case, the constraints are easily obtained from the sequences resulting from $seqs_{merge}\ (merge^S)$. In the *msort* case, the following relations obtained from the *length, split* and *merge* invariants, are additionally needed:

$n + 1 = x$	$n := length\ x$
$n_2 = 0.5x$	$n_2 := n/2$
$x + 1 = x_1 + x_2,\ x \geq x_2,\ x \leq n_2 + x_2$	$(x_1, x_2) := split\ n_2\ x$
$z + 1 = z_1 + z_2$	$z := merge\ z_1\ z_2$

In Fig. 8 we show the polyhedra obtained for *merge* and *msort* after a few iterations of the fixpoint algorithm. For *merge*, the fixpoint is reached after the first iteration. For *msort*, the ascendant chain is infinite because of the restrictions $x \leq 2,\ x \leq 3,\ x \leq 4, \ldots$ The widening technique used in [7], and original from [11], eliminates this just by keeping as P_{next} the restrictions of iteration i implied by the ones of iteration $i + 1$. This leads to $\{x \geq 1,\ z = x\}$ as the size invariant of *msort*.

5 Ranking Functions Synthesis

In [17] we presented several inference algorithms to statically obtain upper bounds to the memory consumption of a *Core-Safe* program. One of them was for inferring resident heap memory, a second one for *peak* heap memory, and a last one for peak stack memory. There, we used several functions assumed correct for the following data:

- $nr_f(\overline{x})$ and $nb_f(\overline{x})$ respectively gave upper bounds to the number of non-base and base calls of the runtime call tree unfolded by function f when it is called with arguments sizes $\overline{x}$. A non-base call is one recursively calling f again, and a base call is one ending a chain of recursive calls to f.
- $len_f(\overline{x})$ gave an upper bound to the length of the longest chain of recursive calls to f.
- $|\,y\,|_f\ (\overline{x})$ gave an upper bound to the size of variable y (assumed to belong to f's body) as a function of fs' argument sizes $\overline{x}$.

The algorithms were proved correct assuming that we have correct functions for the above figures, but we left open how to infer them. A first step for inferring size functions $|\,y\,|_f$ has been given with the inference of size invariants presented in Sec. 4. By following a similar idea to that of [1], nr_f and nb_f can be approximated should we have a correct function for $len_f(\overline{x})$. In effect, having $len_f(\overline{x})$ and a bound n_f to the maximum number of calls issued from an invocation to f, we can compute the above functions as follows:

$$nb_f(\overline{x}) = \begin{cases} 1 & \text{if } n_f = 1 \\ n_f^{\,len_f(\overline{x})-1} & \text{if } n_f > 1 \end{cases} \qquad nr_f(\overline{x}) = \begin{cases} len_f(\overline{x}) - 1 & \text{if } n_f = 1 \\ \frac{n_f^{\,len_f(\overline{x})-1}-1}{n_f-1} & \text{if } n_f > 1 \end{cases}$$

This figures correspond to the internal and leaf nodes of a complete tree of branching factor n_f and height $len_f(\overline{x})$. The branching factor n_f is a static quantity easily computed by taking the maximum number of consecutive calls in the sequences returned by function $seqs_f$ of Fig. 4. For instance, $n_{merge} = 1$ and $n_{msort} = 2$.

So, it suffices to approximate the function $len_f(\overline{x})$. To this aim we will use Podelski and Rybalchenko's method [18]. It is complete for linear ranking functions of loops of the form **while** B **do** S, where $B(\overline{x})$ is a conjunction of linear constraints over the variables $\overline{x}$ involved in the loop, and $S(\overline{x}, \overline{x}')$ is a transition relation expressed as a conjunction of linear constraints over the variable values respectively before and after executing the loop body. Using these constraints over $\overline{x}$ and $\overline{x}'$, the method creates another set of constraints over $\overline{\lambda_1}$ and $\overline{\lambda_2}$, two lists m of non-negative variables, being m the number of restrictions contained in the conjunction of $B(\overline{x})$ and $S(\overline{x}, \overline{x}')$. This set is satisfiable if and only if a linear ranking function exists for the **while**, and it can be synthesised from the values of $\overline{\lambda_1}$ and $\overline{\lambda_2}$. More precisely, the method synthesises a vector $\overline{r}$ of real

numbers and two constants $\delta > 0$ and δ_0 such that:

$$\begin{cases} \overline{r}.\overline{x} \geq \delta_0 & \forall \overline{x} \,.\, B(\overline{x}) \\ \overline{r}.\overline{x}' \leq \overline{r}.\overline{x} - \delta & \forall \overline{x},\, \overline{x}' \,.\, B(\overline{x}) \wedge S(\overline{x}, \overline{x}') \end{cases}$$

These conditions guarantee the termination of the loop. The aim of [18] is proving termination and to exhibit a certificate of the proof. In this respect, *any* ranking function is a valid certificate. Our aim is slightly different: we seek for the *least* upper bound to the length of the worst case call chain to a recursive *Core-Safe* function f^S. Then, we introduce two variations to [18]:

- We replace the restriction $\delta > 0$ by $\delta \geq 1$. In this way, each transition counts at least as an internal call to f^S and so we will get an upper bound to the number of internal calls in the chain (this would not be true if $0 < \delta < 1$).
- We reformulate the problem as a *minimisation* one. We ask for the solution giving the minimum value of the following objective function:

$$Obj \stackrel{\text{def}}{=} \sum \overline{\lambda_1} + \sum \overline{\lambda_2} - \delta_0$$

Minimising $-\delta_0$ is equivalent to maximising δ_0, expressing that we look for the minimum value of $\overline{r}.\overline{x}$ that is still an upper bound. Minimising the values of $\overline{\lambda_1}$ and $\overline{\lambda_2}$ is needed because we have seen that requiring only the first condition frequently leads to unbounded linear problems with an infinite number of solutions and the minimum one is in the infinite for some λ_i.

The only remaining task is to codify our abstract size functions as **while** loops. In this respect, the only meaningful information is the size change between the arguments of an external call to f^S and the arguments of its internal calls. The result sizes are not relevant for termination of the call chains. But we must decide what to do when there are more than one internal call, either excluding each other (as in $merge^S$), or executed in sequence (as in $msort^S$). Our approach has been to compute the convex hull of the restrictions coming from all the internal calls, and to use this polyhedron both as the guard $B(\overline{x})$ —by collecting all restrictions depending only on $\overline{x}$—, and as the transition relation $S(\overline{x}, \overline{x}')$ —by collecting all restrictions depending both on $\overline{x}$ and $\overline{x}'$. The justification of this decision in the case of excluding calls is clear: at each 'iteration' the function may decide to take a possible different branch, so the convex hull amounts to compute the logical 'or' of the restrictions coming for all the branches. In the case of consecutive calls, the reasoning is different: at each internal node of the call tree, the function will take all the children branches and we seek for a bound to the worst case path. The convex hull collects in this case the minimum set of restrictions applicable to all the branches. It is like having a loop that non-deterministically decides at each iteration which branch will follow in the tree. A bound to the iterations of this 'loop' is then a bound to the longest path in the call tree.

In Fig. 9 we show the restrictions of each internal call for $split^S$, $merge^S$, and $msort^S$, and their respective convex hulls. When introducing this data to the

Function	Internal call 1	Internal call 2	Convex hull
split $n\ x$	$\{n \geq 1,\ x \geq 2,\ n' = n-1,\ x' = x-1\}$		$\{n \geq 1,\ x \geq 2,\ n' = n-1,\ x' = x-1\}$
merge $x\ y$	$\{x \geq 2,\ y \geq 2,\ x' = x-1,\ y' = y,\ z = 1+z'\}$	$\{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y-1,\ z = 1+z'\}$	$\{x \geq 2,\ y \geq 2,\ x+y = x'+y'+1\}$
msort x	$\{x \geq 3,\ x' = \frac{1}{2}x\}$	$\{x \geq 3,\ x' = \frac{1}{2}x+1\}$	$\{x \geq 3,\ x' \geq \frac{1}{2}x,\ x' \leq \frac{1}{2}x+1\}$

Fig. 9. Termination restrictions of *split*, *merge* and *msort*

above formulation of Podelski and Rybalchenkos's method, we got the following ranking functions:

Function	$\overline{r}$	δ_0	$len_f(\overline{x})$	$B(\overline{x})$
split $n\ x$	$[0,1]$	2	x	$n \geq 1 \wedge x \geq 2$
merge $x\ y$	$[1,1]$	4	$x+y-2$	$x \geq 2 \wedge y \geq 2$
msort x	$[2]$	2	$2x$	$x \geq 3$

We take as $len_f(\overline{x})$ the expression $\overline{r}.\overline{x} - \delta_0 + 2$, because $\overline{r}.\overline{x} - \delta_0 + 1$ is a bound to the number of 'iterations', each one corresponding to an internal call to f^S, and we add 1 for taking into account the code before the loop representing the initial call of the chain. Of course, this length is valid when $B(\overline{x})$ holds at the beginning. Otherwise, the length is just 1. Notice that the bounds for *split* and *merge* are tight, while the one for *msort* is not (a tight bound would be $\log_2 x + 1$). An obvious limitation of the method is that it only can give a linear function as a result.

6 Conclusions

We have shown that linear techniques can be successfully applied to a first-order functional language in order to infer size invariants between the arguments and results of functions, and upper bounds to the longest call chain of recursive functions. To this respect, some previous transformations of the program may be needed in order to distinguish between internal recursive calls related by 'or' (i.e. excluding each other), from those related by 'and' (i.e. executed in a sequence). This distinction comes for free in Prolog programs, but not in functional ones.

Linear techniques —namely abstract interpretation on polyhedral domains and linear ranking function synthesis— have been extensively used in logic and imperative languages (see [3] for a broad bibliography), but apparently there have been no much interest in applying them to functional languages. An exception regarding termination analysis (not necessarily a linear one) is Sereni and Jones work [20] applying the SCT criterion to the termination of ML programs.

This paper should be considered as a proof-of-concept one in the sense that we still have neither a complete implementation, nor a full-size benchmark of test cases. The algorithms presented here have been implemented in SWI-Prolog[1], by using its CLP(Q) and simplex libraries. We have adapted to this Prolog system Andy King's algorithm [8] for computing convex hulls. We are grateful to the implementers of these tools and algorithms, and also to our colleague Samir Genaim for putting us on the track of Andy King's works.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis Symposium, SAS'08*, pages 221–237. LNCS 5079, Springer, 2008.
2. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium, SAS'10*, LNCS (to appear), pages 1–16. Springer, 2010.
3. R. Bagnara, P. M. Hill, E. Ricci 0002, and E. Zaffanella. Precise widening operators for convex polyhedra. *Sci. Computer Programming*, 58(1-2):28–56, 2005.
4. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. The Parma Polyhedra Library, User's Manual. Dept. of Mathematics, Univ. of Parma, Italy, Available at: `http://www.cs.unipr.it/ppl/`, July 2002.
5. Amir M. Ben-Amram and Michael Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 218–232. Springer, 2008.
6. Amir M. Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.
7. Florence Benoy and Andy King. Inferring Argument Size Relationships with CLP(R). In John P. Gallagher, editor, *LOPSTR*, volume 1207 of *LNCS*, pages 204–223. Springer, 1996.
8. Florence Benoy, Andy King, and Frédéric Mesnard. Computing convex hulls with a linear solver. *TPLP*, 5(1-2):259–271, 2005.
9. Michael Colón and Henny Sipma. Practical Methods for Proving Program Termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
11. Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
12. Laure Gonnord and Nicolas Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006.
13. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.
14. S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Proc. Logic-Based Program Synthesis and Transformation, LOPSTR'08, Valencia, Spain*, pages 43–57, July 2008.

[1] Available at `http://www.swi-prolog.org/`.

15. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
16. M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-order Functional Language. In S. Escobar, editor, *Work. on Functional and Logic Programming, WFLP 2009, Brasilia*, pages 145–161. LNCS 5979, 2009.
17. M. Montenegro, R. Peña, and C. Segura. A space consumption analysis by abstract interpretation. In *Selected papers of Foundational and Practical Aspects of Resource Analysis, FOPARA'09, Eindhoven*, pages 34–50. LNCS 6324, Nov. 2009.
18. Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
19. Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.
20. Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.

Data-Driven Detection of Catamorphisms

Towards Problem Specific Use of Program Schemes for Inductive Program Synthesis

Martin Hofmann

Faculty Information Systems and Applied Computer Science, University of Bamberg,
email: martin.hofmann@uni-bamberg.de

Abstract. Inductive Program Synthesis or Inductive Programming (IP) is the task of generating (recursive) programs from an incomplete specification, such as input/output (I/O) examples. All known IP algorithms can be viewed as search in the space of all candidate programs, with consequently exponential complexity. To constrain the search space and guide the search traditionally program schemes are used, usually given a priori by an expert user. Consequently, all further given data is now interpreted w.r.t this schema which now almost exclusively decides on success and failure, depending on whether it fits the data or not. *Instead of trying to fit the data to a given schema* indiscriminately, we propose to utilise knowledge about data types to *choose and fit a suitable schema to the data*! Recursion operators associated with data type definitions are well known in functional programming, but less in IP. We propose to exploit universal properties of type morphisms which may be detected in the given I/O examples. This technique allows us to introduce catamorphisms as generic recursion schemes on arbitrary inductive data types in our analytical inductive functional programming system IGOR II.

1 Introduction

Inductive Program Synthesis or Inductive Programming (IP) researches the task of automatically synthesising probably recursive programs. Contrary to deductive program synthesis, which relies on a complete and formalised specification, IP uses incomplete specifications such as input/output (I/O) examples or execution traces of the desired program's behaviour. Usually, it focuses on the synthesis of *declarative* (logic, functional, or functional logic) programs.

The aims of IP are manifold. On the one hand, research in IP provides better insights in the cognitive skills of human programmers. On the other hand, powerful and efficient IP systems can enhance software systems in a variety of domains—such as automated theorem proving and planning—and offer novel approaches to knowledge based software engineering such as model driven software development or test driven development, as well as end user programming support in the XSL domain [1]. So it should be clear that to "automagically" create a whole system is far too daring, but generating parts of software such as modules or functions is a quite realistic task.

Beginnings of IP research first addressed inductive synthesis of functional programs from small sets of positive I/O examples only [2]. One of the most influential classical systems was THESYS [3] synthesising linear recursive LISP programs by rewriting I/O pairs into traces and folding of traces based on recurrence detection. Currently, functional IP is covered by the analytical approaches IGOR I [4], and IGOR II [5, 6] and by the evolutionary/generate-and-test based approaches ADATE [7] and MAGICHASKELLER [8]. G∀ST is an example of using a theorem prover generating a program along a proof by contradiction [9].

Analytical approaches work example-driven, so the structure of the given I/O pairs guides the construction of generalised programs. They are typically very fast and can guarantee certain characteristics of the generated programs such as minimality of the generalisation w.r.t. to the given examples and termination. However they are restricted to programs describable by a small set of I/O pairs. See [5] for more information on analytical approaches.

Essentially, generate-and-test approaches successively enumerate all possible programs and test them against the given I/Os. In the case of evolutionary systems, first one or more hypothetical programs are constructed, which are then evaluate them against the I/Os and the most promising hypotheses are developed further. Other systems enumerate exhaustively w.r.t. a library of primitives or an algebraic data type describing the class of programs. Although they are all very powerful and usually quite unrestricted, they are extremely time consuming.

Two decades ago, inductive logic programming (ILP) systems were presented with focus on learning recursive logic *programs* in contrast to learning classifiers: FFOIL [10], GOLEM [11], PROGOL [12], and the interactive system DIALOGS [13].

Thus, IP can be viewed as a special branch of machine learning because programs are constructed by inductive generalisation from examples. Therefore, as for classification learning, each approach can be characterised by its restriction and preference bias [15]. However, IP cannot be evaluated with respect to some covering measure or generalisation error since (recursive) programs must treat *all* I/O examples correctly to be an acceptable hypothesis for the given examples.

Similar to deductive program synthesis, where a program is derived from a complete specification, all known IP algorithms struggle with the exponential explosion of the search space which makes search intractable from some point. To reduce complexity usually additional information preferably as program schemes is used (c.f. divide-and-conquer in DIALOGS-II). However, they are fixed a priori, unable to adapt to the given data. Consequently, either a schema fits more or less by chance or due to an expert user, or it misleads the search.

We propose to *exploit universal properties of type morphisms* which may be detected in the I/O examples. Thus, we introduce catamorphisms as generic recursion schemes on arbitrary inductive data types *only if they are suitable*, generalising the detection of list catamorphisms as presented in [6]. We start with an introductory example in Section 2, introduce the necessary theoretical concepts on term rewriting (§3), give an overview of the IP system IGOR II (§4), and describe the extension of our algorithm to detect recursion schemes (§5). We finish with some empirical results (§7) and conclude in Section 8.

2 A Motivating Example

Consider the problem of computing the length of a list. The data type definition of a list with elements of type α is standard. Either the list is empty or an element of type α is inserted at the front of an α-list (`[α]`). Natural numbers are defined as Peano's integers. HASKELL [16] is our functional language of choice where type constructors are capitalised, variables and function names are in lower case.

```
data [α] = [] | (α : [α]) --quasi Haskell
data Nat =  Z | (S Nat)
```

Four simple I/O equations together with the type of the function specify the problem of computing the length of a list.

```
length :: [α] → Nat
length         []  =         Z
length     (a:[]) =     (S Z)
length   (a:b:[]) =   (S(S Z))
length (a:b:c:[]) = (S(S(S Z)))
```

A typical recursive solution generated by an IP system would look as follows. The pattern wildcard (`_`) denotes a variable not occurring on the right-hand side.

```
length []     = Z
length (_:xs) = S (length xs)
```

A functional programmer however, used to think in terms of recursion schemes and higher-order functions, maybe would come to an alternative solution. It is common to define `length` in terms of the higher-order function `fold` which takes an anonymous λ-abstraction taking two arguments and incrementing the second by one, the default value `Z`, and the original list `xs` as input.

```
length xs = fold (λ _ n → S n) Z xs
```

The well-known higher-order function `fold` is a program scheme for structural recursion over lists, taking a binary function `f` of type `(α → β → β)`, a default value `v` of type β and an α-list as input, returning a value of type β.

```
fold :: (α → β → β) → β →[α] → β
fold _ v []     =  v
fold f v (x:xs) =  x `f` (fold f v xs)
```

Intuitively, it can be seen as a function replacing each *cons*-constructor `(:)` of the input list by a call to `f`[1] and the empty list `[]` by the default value `v`. So the expression `fold f v (a:(b:[]))` would lazily expand to `(a `f` b `f` v)`.

The higher-order function `fold` can be generalised to a *catamorphism*, describing structural recursion on arbitrary inductively defined data types. We will show that the applicability of a catamorphism can be checked w.r.t. given I/O examples and used for IP to exploit catamorphisms as a general program scheme for structural recursion.

[1] `` `f` `` denotes in HASKELL infix application of `f`.

3 Terms, Equations, and Rewriting

Before we come to the main contribution of this paper, we need to fix some concepts. Let for our purpose be a functional program a set of equations consisting of pairs of terms over a many-sorted signature Σ which form a term rewriting system. We follow the common definitions as described in, e.g., Terese [17].

The set of symbols of Σ consists of two disjoint sets of *defined functions symbols* $\mathcal{D}$ and data type *constructors* $\mathcal{C}$. Terms over a set of variables $\mathcal{X}$ and symbols from Σ or $\mathcal{C}$ are denoted by $\mathcal{T}_\Sigma(\mathcal{X})$ or $\mathcal{T}_\mathcal{C}(\mathcal{X})$, respectively. Terms in $\mathcal{T}_\mathcal{C}(\mathcal{X})$ are called *constructor terms*. $\mathcal{V}ar(t)$ are all variables of a Term t. Since Σ is many-sorted, our terms are typed. An equation, or rule, defining (amongst others) a function $F \in \mathcal{D}$ consists of a left-hand side (lhs) of the form $F(p_1, \ldots, p_n)$, with $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$ for $i = 1 \ldots n$ and a right-hand side (rhs) $r \in \mathcal{T}_\Sigma(\mathcal{X})$. The constructor terms p_i on the lhs of an equation may contain variables and are called *pattern*. All variables on the rhs of an equation are required to occur in the pattern on the lhs. We say that such variables are *bound* (or *unbound* otherwise).

A *substitution* is a function $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$. For our purpose, we write it in postfix and extend it to terms replacing all contained variables simultaneously. So $t\sigma$ is the result of applying the substitution σ to term t, i.e., applying σ to each $v \in \mathcal{V}ar(t)$. If $s = t\sigma$, then t is called a *generalisation* of s and we say that t *subsumes* s and s matches t by σ, respectively. Given two terms t_1 and t_2 and a substitution σ such that $t_1\sigma = t_2\sigma$, we say that t_1 and t_2 *unify*. Given a set of terms $S = \{s_0, \ldots, s_n\}$, then there exists a term t which subsumes all terms in S and which itself is subsumed by any other term also subsuming all terms in S. The term t is called the *least general generalisation* (lgg) [18] of the terms in S. To generalise lggs to a set of equations, we tacitly treat the equal sign as a constructor symbol with the lhs and the rhs as arguments.

The operational semantics of a set of equations in the above mentioned form are best described in terms of a *term rewriting system* (TRS). An equation can be read as a *simplification* (or *rewrite*) *rule* replacing a term matching the lhs by the rhs. TRS whose equations have the above described form are called *constructor* (*term rewriting*) *systems* (CS). From now on, we will use the terms equation and rule as well as equation set and CS.

Let $\boldsymbol{i}$ be the vector $i_1, \ldots, i_n$. Evaluating an n-ary function F for an input $\boldsymbol{i}$ consists of repeatedly rewriting the term $F(\boldsymbol{i})$ w.r.t. the rewrite relation R implied by the CS until the term is in *normal form*, i.e., cannot be rewritten further. A sequence of (in)finitely many rewrite steps $t_0 \to_R t_1 \to_R \ldots$ is called *derivation*. If a derivation starts with term t and results in a normal form s this is written $t \xrightarrow{!}_R s$. We say that t normalises to s and call s the normal form of t. To define a function on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this a *confluent* CS, where no two lhss of a CS unify. A CS is *terminating* if each possible derivation terminates. A sufficient condition for termination is that the inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

4 The IP System Igor II

IGOR II is our analytical, functional inductive programming system. The specification of a problem given to IGOR II comprises the definitions of all used data types which fixes the set of constructors $\mathcal{C}$, a set of example equations E for the *target function*, i.e., the function to be induced, and a set of *background knowledge* equations B, i.e., user provided and predefined functions usable during synthesis. Both E and B have constructor terms on their rhss and describe the I/O behaviour of these defined function symbols $\mathcal{D}$ on a suitable subset of their domain. IGOR II returns a set of equations P which form together with B a confluent CS which is correct in the following sense:

$$\forall(F(\boldsymbol{i}) = o) \in E\,. \quad F(\boldsymbol{i}) \xrightarrow{!}_{P \cup B} o \tag{1}$$

So P together with B is a CS which rewrites each lhs of an example from E to its rhs. We call $\boldsymbol{i}$ *example input* and o *example output* even if the example equations may contain variables.

Let Σ be the signature of $E \cup B$. Then $\Sigma \cup \mathcal{D}_A$ is the signature of the induced program P, where $\mathcal{D}_A$ is a set of arbitrary but fixed defined function symbols not contained in $E \cup B$. These defined symbols are names of auxiliary, possibly recursive functions which are introduced dynamically during the synthesis process. Variables are also dynamically introduced during the induction process and not a priori fixed. For detailed information on IP and CS see [19].

Auxiliary functions are restricted in two aspects: First, the input type of an auxiliary function is identical with the input type of the function calling it. Thus, IGOR II cannot automatically infer auxiliary parameters as, e.g., the accumulator parameter in the efficient implementation of `reverse`. Second, auxiliary function symbols cannot occur at the root of the rhs of another function calling it. Such restrictions are called *language bias*.

Obviously, there are infinitely many solutions P which are correct in the sense of property (1), namely the example equations E themselves for example. As almost all inductive inference methods IGOR II has a so called *preference bias* to choose the 'best' among many possible solutions. IGOR II prefers those solutions which patterns partition the example inputs in fewer subsets. Since the search is complete w.r.t to the preference bias on the hypotheses, those solutions with the least number of partitioning patterns are found.

Contrary to generate-and-test approaches, where the search space is usually traversed randomly or depth-first, IGOR II organises the induction of a terminating, confluent, and correct CS as a uniform-cost search. During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with unbound variables in the rhss*. We call such equations and hypotheses containing them *unfinished equations* and *unfinished hypotheses*.

The initial hypothesis is a CS with one rule per target function such that its pattern subsumes all its example inputs. Usually, one equation is not enough to explain all its examples and the rhs remains unfinished. Now successively

the best hypothesis, w.r.t. the preference bias, is selected, an unfinished rule is chosen and replaced by its successor rules. Since IGOR II has multiple functions computing successor rules and each may have multiple results, the currently best hypothesis may be developed to multiple successors.

IGOR II applies three methods to replace unfinished rules by successor rules. The first method *splits rules by pattern refinement* and replaces an unfinished rule with pattern $\boldsymbol{p}$ by at least two new rules with more specific patterns in order to establish a case distinction. The second method *introduces auxiliary functions* and generates new induction problems (new example equations) for those subterms of an unfinished rhs containing unbound variables. The third method *introduces function calls* to the target function itself (a recursive call) or to any other defined function, i.e., other target functions, background functions, or previously introduced auxiliary functions. Finding the arguments of such a call is considered as new induction problem.

A goal state is reached, if at least one of the best—according to the preference bias—hypotheses is finished, i.e., does not contain unfinished equations. Such a finished hypothesis is terminating and confluent by construction and since its equations entail the example equations, it is also correct.

Although IGOR II in the current implementation synthesis HASKELL-programs, it is not limited to a specific language. Older version for example synthesised in the term-rewriting language MAUDE, and IGOR II was applied to the XML domain and synthesised XSL transformations [1]. However, it heavily relies on pattern matching and functional I/Os, and thus using declarative programming languages is kinf of natural. Nevertheless, experiments showed that using a "functional core" of a procedural or object-oriented language is possible [20], even though a bit awkward.

4.1 Initial Rules

As a set of example equations for a target function, IGOR II requires the first n (positive) I/O examples w.r.t. the structure of the underlying data type. Traditionally, IP systems usually required also negative examples, to prune the search space , and correctly generalise over the the positive examples. IGOR II does not need negative examples but still generalises correctly, because given that the target program is functional and really the first n examples are given, domain and codomain of the target functions are fully describes up to the n^{th} example.

The initial rule is constructed by antiunifying [18] all I/O examples, i.e., computing their lgg. In particular, the pattern of the initial rule is the most specific term which subsumes all example inputs. Considering only lggs of example inputs as patterns narrows the search space, but does not constrain completeness of hypothesis regarding the example equations. Consider an arbitrary CS C with non-unifying but possibly not most specific patterns which is correct regarding a set of example equations. There exists a CS C', s.t. for any rule whose lhs is not a lgg of its subsumed example inputs, one can construct a rule whose lhs is the lgg of the same example inputs and computes the same normal form for each example input [5].

4.2 Splitting Rules by Pattern Refinement

Let the example equations whose inputs match the lhs $F(\boldsymbol{p})$ of the unfinished rule be denoted by $E_{F,\boldsymbol{p}}$. The first method for generating successors of an unfinished rule is to replace its pattern $\boldsymbol{p}$ by a set of more specific patterns, s.t. the new patterns induce a partition of the example inputs in $E_{F,\boldsymbol{p}}$. This results in a set of new rules replacing the original rule and induce a case distinction. It has to be assured that no two of the new patterns unify.

This is done as follows. Let u be a position in the pattern $\boldsymbol{p}$ of our rule at hand, then $\boldsymbol{p}|_u$ denotes the subterm of $\boldsymbol{p}$ at position u. Now choose such a u, such that $\boldsymbol{p}|_u$ is a variable. Since $\boldsymbol{p}$ is the lgg of the example inputs, at least two inputs have different constructor symbols at position u. Then all example inputs with the same constructor at position u are taken into the same subset. This leads to a partition of the example equations. Finally, for each subset of equations the lgg is computed leading to a set of initial rules with refined patterns.

If $\boldsymbol{p}$ contains multiple variable positions all partitions are generated. Since specialised patterns subsume fewer inputs, the number of unbound variables in the initial rhss (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer case distinctions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

4.3 Introducing Auxiliary Functions

The second method to generate successors is applicable, if all example equations $F(\boldsymbol{i}) = o \in E_{F,\boldsymbol{p}}$ have the same constructor symbol c at the root of their rhs. Let c be of arity m then the rhs of the unfinished rule is replaced by the term $c(S_1(\boldsymbol{p}), \ldots, S_m(\boldsymbol{p}))$ where each $S_i \in \mathcal{D}_A$ denotes a new defined (auxiliary) function. The rule is finished, because all variables are bound.

Examples for the new auxiliary functions are abduced from the examples of the current function as follows: Let again be $o|_j, j = 1, \ldots, m$ be the j^{th} subterm of the example rhss o, then the equations $S_j(\boldsymbol{i}) = o|_j$ are the example equations of the new subfunction S_j. Thus, correct rules for S_j compute the j^{th} subterms of the rhss o such that the term $c(S_1(\boldsymbol{p}), \ldots, S_m(\boldsymbol{p}))$ normalises to the rhss o.

4.4 Introducing Function Calls

A third method to generate successor sets for an unfinished rule with pattern $\boldsymbol{p}$ for a target function F is to replace its rhs by a call to a defined function F', i.e., by a term $F'(S_1(\boldsymbol{p}), \ldots, S_k(\boldsymbol{p}))$. Each $S_j, j = 1, \ldots, k$, denotes a new introduced defined (auxiliary) function. This finishes the rule, since now the rhs does not longer contain variables not contained in the lhs. In order to get a rule leading to a correct hypothesis, for each example equation $F(\boldsymbol{i}) = o$ of function F whose input $\boldsymbol{i}$ matches $\boldsymbol{p}$ with substitution σ it must hold that $F'(S_1(\boldsymbol{p}), \ldots, S_k(\boldsymbol{p}))\sigma \xrightarrow{!} o$, i.e., the call to F' computes the same output as F for each input $\boldsymbol{i}$.

This holds if for each example rhs o an example equation $F'(\boldsymbol{i'}) = o'$ of function F' exists such that $o = o'\tau$ for a substitution τ and $S_j(\boldsymbol{p})\sigma \stackrel{!}{\rightarrow} \boldsymbol{i'}[j]\tau$ for each S_j and $\boldsymbol{i'}[j]$, where $\boldsymbol{i'}[j]$ is the j^{th} element in the vector $\boldsymbol{i'}$. Thus, if we find such example equations of F', then we abduce them as $S_j(\boldsymbol{i}) = \boldsymbol{i'}[j]\tau$ for the new subfunctions S_j and induce those from these examples. Provided, the final hypothesis is correct for F' and all S_j then it is also correct for F.

In order to assure termination of the final hypothesis it must hold $i' < i$ according to some reduction order $<$, usually the syntactic term lengths, if the function call is recursive.

5 Detecting Catamorphisms

Reasoning about functional programs with means of category theory is well known as *Constructive Algorithmics* or *Bird-Meertens-Formalism* [21], and many concepts as e.g. *type functors* and *catamorphisms* have arrived in everyday functional programming.

For our purpose it is relevant to recall that inductive data types are generated by constructors and come together with a scheme for structural recursion induced by their constructors. Categorially, given a *distributive* base category, they correspond to initial (functor-) algebras in the category of F-algebras.

Given an endofunctor $\mathsf{F} : \mathcal{C} \rightarrow \mathcal{C}$, by definition, an F-algebra $\mathbf{A} = (A, \varphi)$ is a tuple consisting of an object A (carrier) of $\mathcal{C}$ and a morphism $\varphi : \mathsf{F}A \rightarrow A$ (algebra structure). A homomorphism between F-algebras $\mathbf{A} = (A, \varphi)$ and $\mathbf{B} = (B, \psi)$ is a morphism $f : A \rightarrow B$ s.t. $f \circ \varphi = \psi \circ \mathsf{F}f$. All F-algebras and F-algebra morphisms together with identities and composition form the category of F-algebras $\mathbf{Alg}_\mathsf{F}$. The category $\mathbf{Alg}_\mathsf{F}$ may or may not contain initial objects, i.e. *initial* F-*algebras*, depending on F, but if one exists it is uniquely defined up to isomorphism. In a distributive category initial F-algebras are guaranteed to exists for polynomial functors. Such an initial F-algebra is said to be *the* initial F-algebras and denoted $\mu\mathbf{F} = (\mu\mathsf{F}, \mathrm{in}_\mathsf{F})$. Given an endofunctor $\mathsf{F} : \mathcal{C} \rightarrow \mathcal{C}$ and the initial algebra, for any F-algebra $\mathbf{A} = (A, \varphi)$ there exists a *unique* morphism $(\!|\, \varphi \,|\!)_\mathsf{F} : \mu\mathsf{F} \rightarrow A$, the F-*catamorphisms*, s.t.

$$\begin{array}{ccc} \mathsf{F}\mu\mathsf{F} & \xrightarrow{in_\mathsf{F}} & \mu\mathsf{F} \\ {\scriptstyle \mathsf{F}(\!|\, \varphi \,|\!)_\mathsf{F}}\downarrow & & \downarrow{\scriptstyle (\!|\, \varphi \,|\!)_\mathsf{F}} \\ \mathsf{F}A & \xrightarrow{\varphi} & A \end{array} \qquad (\!|\, \varphi \,|\!)_\mathsf{F} \circ in_\mathsf{F} = \varphi \circ \mathsf{F}(\!|\, \varphi \,|\!)_\mathsf{F}. \tag{2}$$

Thus we know that given an inductively defined data type we get a unique morphism to any other type for free. Put differently, from an inductive programming point of view we can say that, when searching for a function $f : \mu\mathsf{F} \rightarrow A$, we can express it using a catamorphism $f = (\!|\, \varphi \,|\!)_\mathsf{F}$ if we can find an appropriate mediating function φ. For our system IGOR II this is just another induction problem given the accordant example equations. Thus we can exploit data type

specific knowledge from functional programming for inductive synthesis. In the next subsection (§5.1) we describe the algorithm from the categorial semantic perspective. Later (§5.2) we again take on a syntactic point of view and dive into the nitty-gritty of rewriting and constructing terms.

5.1 The Categorial Perspective

Recall that our category of choice is distributive. Therefore are our types polynomial, i.e. only built from primitive types by products and coproducts. The functors are polynomial too, i.e. only built from products, coproducts, the identity functor, and the constant functor. Thus, referring to diagram (2), we can say that for the functor F of our F-algebra it holds that $\mathsf{F} = \mathsf{F}_1 + \ldots + \mathsf{F}_n$, i.e. F is the coproduct of n functors $\mathsf{F}_i, i = 1 \ldots n$. Hence, the mediating function of our catamorphism $\varphi : \mathsf{F}A \to A$ is in fact a case distinction $\varphi = [\varphi_1, \ldots, \varphi_n]$, so for each element of our coproduct of type $\mathsf{F}A$ one function $\varphi_i : \mathsf{F}_i A \to A$.

To obtain the appropriate example equations, figuratively speaking, starting from $\mu\mathsf{F}$, we just need to follow the arrows from diagram (2) counterclockwise applying the inverse of in_F to decompose our inductive type $\mu\mathsf{F}$ into a product type. Then we apply $f = (\!|\, \varphi \,|\!)_\mathsf{F}$ to all components of the product which are of type $\mu\mathsf{F}$ recursively, all other product components remain unchanged. These are the inputs for φ which composes the results.

We proceed as follows. Since $f = (\!|\, \varphi \,|\!)_\mathsf{F}$ we only need to abduce the correct inputs for φ, the outputs stay the same. If for φ_i the functor F_i is a constant functor K_B, it holds that $\varphi_i : \mathsf{F}A \to A$ and we can take the accordant constant part of type $\mathsf{F}A$ from the examples of f as new inputs. If the functor is the identity functor $\mathsf{Id}_{\mu\mathsf{F}}$ we replace this part in the input terms by the result of a recursive call of f which is the rhss of an example equation of f with the accordant input. If F_i is a product functor the input for φ_i is a nested tuple. Its arguments can be abduced by applying this procedure recursively.

5.2 The Term Rewriting Perspective

With category theory we won't get any further, so lets put on the term rewriting goggles. Consider a set of example equations $F(\boldsymbol{i}) = o \in E_{F,}\boldsymbol{p}$ and assume for the sake of simplicity $\boldsymbol{i}$ to be a vector with only one field. Assume further, $\boldsymbol{i}$ is of type τ, an inductive data type. Since all terms subsumed by $\boldsymbol{p}$ are of the same type, its type constructors induce a natural partitioning into n disjoint subsets $E_{F,}\boldsymbol{p} = \{E_{F,}\boldsymbol{p}_1, \ldots, E_{F,}\boldsymbol{p}_n\}$, i.e. for each constructor symbol c_i one. Each pattern $\boldsymbol{p}_i$ is of the form $c_i(\boldsymbol{p'})$.

Each set of example equations $E_{F,}\boldsymbol{p}_i$ gives now rise to a new subfunction S_i. The example equations can be abduced using the examples in $E_{F,}\boldsymbol{p}_i$ which are of the form $F(c_i(\boldsymbol{p'})) = o$. We create an new set of examples E_{S_i} such that for each equation in $E_{F,}\boldsymbol{p}_i$ we create a an equation $S_i(\boldsymbol{q}) = o$, where $\boldsymbol{q} := (q_1, \ldots, q_n)$ is (in our case for now) a vector containing a single n-ary tuple. If $\boldsymbol{p'}[i]$ is of type

τ and there is an equation s.t. $F(\boldsymbol{p'}[i]) = o_i$, then it holds that q_i is o_i. If $\boldsymbol{p}[i]$ is not of type τ then q_i is $\boldsymbol{p}[i]$.[2]

In plain words, an n-ary constructor term $c_i(\boldsymbol{p'})$ which is input to F is transformed to an n-ary tuple and given as input to S_i. Each direct subterm t of type τ of c_i is replaced by the result of a recursive call to F, i.e. by the rhss of the equation of F that subsumes t. All other direct subterms are kept unchanged.

Thus, for each constructor symbol of the inductive type τ we get one function. The coproduct of those functions, i.e. a case distinction on the constructor symbol, is exactly the mediating function needed for the catamorphism. If there is no support in the example equations for one constructor symbol, or for one function not all examples can be abduced, no examples are generated!

5.3 Example

Consider the recursive problem of mirroring a binary tree, where either the tree is empty or it contains a node holding an element of type α and two subtrees.

```
data Tree α = E | N (Tree α) (Tree α)
```

Four simple equations specify the problem of mirroring a tree.

```
mirror  :: (Tree α) →(Tree α)
mirror  E                              = E
mirror (N a E E)                       = (N a E E)
mirror (N b (N a E E) (N c E E))       = (N b (N c E E) (N a E E))
mirror (N d (N b (N a E E) (N c E E))
            (N f (N e E E) (N g E E))) =
       (N d (N f (N g E E) (N e E E))
            (N b (N c E E) (N a E E)))
```

The next diagram depicts the problem. The constructors `E` and `N` correspond to the function $empty_E$ end $node_E$ and `mirror` is the function $f = (\!|\, [g, h] \,|\!)$.

$$\begin{array}{ccccc}
\mathbf{1} & \xrightarrow{empty_E} & Tree_E & \xleftarrow{node_E} & E \times Tree_E \times Tree_E \\
\| & & \downarrow f & & \downarrow id_E \times f \times f \\
\mathbf{1} & \xrightarrow[g]{} & Tree_E & \xleftarrow[h]{} & E \times Tree_E \times Tree_E
\end{array}$$

Following our described algorithm we partition the examples w.r.t. the constructor symbol on the root position on the lhs. The first example trivially becomes the example for g, all others those for h. The function g is becomes a constant function `g _ = E` , solved per definition, and constituting the base case. For h, the outputs remain the same, but the inputs are turned into nested tuples where *all tree-subterms were replaced by there mirror image*, i.e. the result of a recursive call.

[2] Recall that $\boldsymbol{p}[i]$ is the i^{th} element in vector $\boldsymbol{p}$, but q_i the i^{th} element of the tuple q!

```
h (a, (E,E))                          = (N a E E)
h (b, ((N a E E), (N c E E)))         = (N b (N c E E) (N a E E))
h (d, ((N b (N c E E) (N a E E))
      ,(N f (N g E E) (N e E E)))) =
  (N d (N f (N g E E) (N e E E))
       (N b (N c E E) (N a E E)))
```

Computing the *lgg* of all examples of h reveals that it is solved too, because all variables are already bound. The function `mirror` can now be written using a generic implementation of catamorphisms on inductive data types.

```
mirror t          = cata (⊥ ::(Tree α)) (g ⊕ h) t
g       _         = E
h (x, (t1, t2)) = N x t2 t1
```

The function `cata`, borrowed from the Pointless HASKELL library for point-free programming with recursion patterns [23], takes a typed expression to resolve the polymorphic type, the sum of the functions `g` and `h` ($\oplus$), and the tree.

6 Benefits gained from Schemes

On the first glance, one might say that adding a new operator for primitive recursion does not improve the expressiveness of IGOR II, since the operators for partitioning and recursive call can already model primitive recursion. This is true, but with a higher-order function in general, and `fold` or catamorphisms in particular, IGOR II can achieve what was not possible before: de facto introducing an invented auxiliary function at the root position of a rhs.

How is this possible, because the auxiliary function to be invented will always occur inside the rhs? Reconsider the generalisation of `reverse` with higher-order schemes. This is the solution, slightly simplified, where IGOR II invented `snoc` [3].

```
reverse x          = fold snoc [] x
snoc x0 []         = [x0]
snoc x0 (x1 : x2)  = x1 : snoc x0 (x1 : x2)
```

Compare it to a definition of `reverse` using `snoc`, but without higher-order:

```
reverse    []  = []
reverse (x:xs) = snoc x (reverse xs)
```

Well, it is a little bit shorter, but the function `snoc` occurs at the root position of the rhs. IGOR II can never invent such a function, because there is no way to abduce accordant I/Os. Even more, `snoc` has a different type as `reverse`. IGOR II requires the input type of an auxiliary function to be the same as the input type of its caller. There is no cue how to add an additional argument to an auxiliary function, because there is no way to provide I/Os for it.

There is another benefit of higher-order schemes. Consider the I/O examples of a function `lengths`, returning the length of each list in the input list[4].

[3] As common, let `fold` be the catamorphism and `map` the type functor on lists.

[4] For the sake of brevity we write in HASKELL's common form with syntactic sugar.

```
lengths                :: [[α]] →[Peano]
lengths []             = []
lengths [[]]           = [Z]
lengths [[a]]          = [S Z]
lengths [[b,a]]        = [S(S Z)]
lengths [[],[]]        = [Z, Z]
lengths [[],[a]]       = [Z,S Z]
lengths [[],[b,a]]     = [Z,S(S Z)]
lengths [[a],[]]       = [S Z, Z]
lengths [[b],[a]]      = [S Z,S Z]
lengths [[c],[b,a]]    = [S Z,S(S Z)]
lengths [[c,a],[]]     = [S(S Z), Z]
lengths [[b,a],[c]]    = [S(S Z),S Z]
lengths [[c,d],[b,a]] = [S(S Z),S(S Z)]
```

IGOR II's solution generalising these examples is shown in the code below.

```
lengths x0 = map fun1 x0
fun1 x0    = fold fun2 Z x0
fun2 x0 x1 = S x1
```

It is apparent, that with each application of a higher-order schema the input type is simplified. From `[[α]] → [Peano]` of `lengths` we can it reduce it to `[α] → Peano` for $\texttt{fun}_1$ and finally to $\alpha \rightarrow$ `Peano` for $\texttt{fun}_2$ such that the constructor application on the rhs is immediately solved by computing the lgg.

We can now tell more about the class of programs synthesisable by IGOR II, too. Sound statements about the kind of programs learnable are not trivial. Now however, we can state that the class of primitive structural recursive functions is a subset of the class of synthesisable functions. A proof needs to follow, though.

7 Empirical Analysis

To our knowledge is IGOR II the only system using recursion schemes in an analytical inductive way, while the indiscriminate use of generate-and-test systems is uninteresting for our purpose. Previous empirical analysis showed that it is faster or at least as fast as any other comparable analytical IP system [19]. With all generate-and-test approaches it can compete w.r.t to time efficiency and also with most of them w.r.t. expressiveness. Therefore, we tested our system only against its old implementation on a set of various example problems. The old version without schemes always applied all three operators (§4) in parallel. The new version prefers hypotheses which use recursion schemes and only applies the "ordinary" operators when the new higher-order operator is not applicable.

Most of the example problems are typically solved with catamorphism, only `fib`, `hanoi`, and mutual recursive `odd`/`even` do not require it. The others more or less suggest for them. A description of all problems is given in Figure 1.

All tests have been conducted under Ubuntu 7.10 on an Intel Dual Core 2.33 GHz with 4GB memory with the HASKELL-implementation version 0.7.1.3 of

`addN`	adds n to all integers in a list
`allodd`	is `True` if all number in the list are `odd`, otherwise `False`
`and`	returns the conjunction of a list of Booleans
`drop`	removes the first n elements of a list
`evens`	filters all even numbers
`fib , +`	computes the n^{th} Fibonacci number, uses addition
`preorder , ++`	traverses a binary tree in preorder, uses list concatenation
`hanoi`	computes a recursive solution of the Hanoi puzzle
`lasts`	returns all last elements in a list of lists
`lengths`	returns the lengths of all lists in a list
`mirror`	mirrors a binary tree
`odd/even`	a mutual recursive definition of `odd` and `even`
`powset , ++`	the power set, uses list concatenation
`reverse`	reverses a list
`reverse , last`	reverses a list with `last` as background knowledge
`reverse , ++`	reverses a list, uses list concatenation
`sum`	sums up a list of integers
`zeros`	filters all zeros in a list of integers

Fig. 1. Description of example problems

IGOR II. The code, the specification and batch file used can be obtained from `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`.

Table 7 shows the number of loops taken by IGOR II to find the correct solution for both, with high-order (with HO) templates and without (no HO). The speedup is rounded to natural numbers. The runtimes are not included, because the difference between both settings are not significant and a deviation within milliseconds is beyond an acceptable accuracy of measurement. However, Table 7 includes the minimal, maximal, average, and the median, of the runtime to convey an intuition of the time taken. Note that the maximal runtimes were taken by `lengths` (11.11s) and by `addN` (26.68s), but without higher-order.

Table 1. Algorithm loops needed for solution

	`addN`	`allodd`	`and`	`drop`	`evens`	`fib, add`	`preorder`	`hanoi`	`lasts`	`lengths`	`mirror`	`odd/even`	`powset, ++`	`reverse`	`rev, last`	`rev, +`	`sum`	`zeros`
no HO	13	↻	↻	↻	23	173	5	5	5	$1581^{\perp}$	4	2/2	$76^{\perp}$	10	11	159	6	6
with HO	2	3	2	2	3	173	3	5	3	2	1	2/2	5	2	2	2	2	2
speedup	7	–	–	–	8	1	2	1	2	791	4	1/1	15	5	6	80	3	3

$\perp$ wrong solution, ↻ timeout

Runtime statistics in seconds
0.001 min, 139.761 max, 4.509 avg., 0.008 median, 24.295 std.dev.

In general we can conclude from the test setting that using catamorphisms as higher-order program schemes reduce the complexity of the search. Examples where IGOR II has been lost in search space (`allodd`, `evens`, `lengths`) are now solvable, because each catamorphism application allows to remove one layer of data constructors. To solve problems where the use of catamorphisms might inappropriate (`switch`), so the preference bias mislead the search, the deterioration of performance is marginal. The performance of other problems where a catamorphism is not applicable remains unchanged.

8 Conclusion

Contrary to previous approaches to incorporate program schemes, where either an (often very well) informed expert user has to provide a template in advance, or templates are used simply on suspicion, regardless whether they are target-aiming or not, we presented an approach to detect the applicability of a program scheme in the provided I/O examples. We utilise the universal property of catamorphisms on lists to introduce it as higher-order function where appropriate. We implemented these findings as a further operator for our IP system IGOR II and an empirical analysis underpins the benefits. As a side effect this allows us to make statements about the class of functions synthesisable by IGOR II.

The approach of exploiting universal properties of morphisms seems to be even more promising. Other morphisms with universal properties suggest for additional, not only structural recursion schemes, as e.g., anamorphisms as the dual to catamorphisms, as well as hylo- or paramorphisms as presented by [24].

In general, the theory about functional programming has already formalised many morphisms with certain properties, which are worth to be looked at and tested for applicability. An idea could be to organise program schemes in a decision tree, where schemes in the same subtree share certain properties. By descending this tree, appropriate schemes can be selected.

References

1. Hofmann, M.: Automatic Construction of XSL Templates – An Inductive Programming Approach. VDM Verlag, Saarbrücken (2007)
2. Biermann, A.W., Kodratoff, Y., Guiho, G.: Automatic Program Construction Techniques. The Free Press, NY, USA (1984)
3. Summers, P.D.: A methodology for LISP program construction from examples. Journal ACM **24** (1977) 162–175
4. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454
5. Kitzelmann, E.: Analytical inductive functional programming. In Hanus, M., ed.: Proceedings of th 18th International Symposium on Logic-Based Program Synthesis and Transformation. Volume 5438 of LNCS., Springer (2008) 87–102

6. Hofmann, M., Kitzelmann, E.: I/O guided detection of list catamorphisms: towards problem specific use of program templates in IP. In: PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation, New York, NY, USA, ACM (2010) 93–100
7. Olsson, R.J.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–83
8. Katayama, S.: Systematic search for lambda expressions. In van Eekelen, M.C.J.D., ed.: Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005. Volume 6., Intellect (2007) 111–126
9. Koopman, P.W.M., Plasmeijer, R.: Systematic synthesis of functions. In Nilsson, H., ed.: Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006. Volume 7., Intellect (2007) 35–54
10. Quinlan, J.R.: Learning first-order definitions of functions. Journal of Artificial Intelligence Research **5** (1996) 139–161
11. Muggleton, S., Feng, C.: Efficient induction of logic programs. In: Proceedings of the 1st Conference on Algorithmic Learning Theory, Ohmsma, Tokyo, Japan (1990) 368–381
12. Muggleton, S.: Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming **13**(3-4) (1995) 245–286
13. Flener, P.: Inductive logic program synthesis with Dialogs. In Muggleton, S., ed.: Proceedings of the 6th International Workshop on Inductive Logic Programming, Stockholm University, Royal Institute of Technology (1996) 28–51
14. Hernández-Orallo, J., Ramírez-Quintana, M.J.: Inverse narrowing for the induction of functional logic programs. In Freire-Nistal, J.L., Falaschi, M., Ferro, M.V., eds.: Joint Conference on Declarative Programming. (1998) 379–392
15. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education (1997)
16. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming **13**(1) (Jan 2003) 0–255.
17. Terese: Term Rewriting Systems. Volume 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)
18. Plotkin, G.: A note on inductive generalisation. In Meltzer, B., Michie, D., eds.: Machine Intelligence 5. Edinburgh University Press, Edinburgh (1969) 153–163
19. Hofmann, M., Kitzelmann, E., Schmid, U.: Analysis and evaluation of inductive programming systems in a higher-order framework. In Dengel, A., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R., eds.: German Conference on Artificial Intelligence (KI'08). Volume 5243 of LNAI., Springer (2008) 78–86
20. Hieber, T., Hofmann, M.: Automated method induction: Functional goes object oriented. In: Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers. Volume Volume 5812 of Lecture Notes in Computer Science., Springer (2010) 159–173
21. Bird, R.S., De Moor, O.: Algebra of Programming. Volume 100 of International Series in Computing Science. Prentice Hall (1997)
22. Lambek, J.: A fixpoint theorem for complete categories. Mathematische Zeitschrift **103**(2) (April 1968) 151–161 Springer Berlin / Heidelberg.
23. Cunha, A.: Point-free programming with hylomorphisms. In: Workshop on Datatype-Generic Programming. 2004
24. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Proceedings of th 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA, USA, August 26-30, 1991, Springer-Verlag (1991) 124–144

Strictness Optimization for Higher-Order Functions in a Typed Intermediate Language
(extended abstract)

Tom Lokhorst, Atze Dijkstra, and S. Doaitse Swierstra

Department of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
tom@lokhorst.eu, atze@cs.uu.nl, doaitse@cs.uu.nl

Abstract. Strictness optimizers in compilers for lazy functional languages are traditionally good at optimizing functions by using unboxed types. In this paper we explore how we can use the information generated by a type-based strictness analyzer to generate efficient code for both first-order, and higher-order functions. We extend the kind system of our intermediate language to encode 'strictness properties'. Since this new kind cannot be compiled to lower level languages, we then use a partial evaluator to remove any reference to the new kind.

1 Introduction

Lazy functional languages like Haskell and Clean allow programmers to write modular, composable programs. Users of these languages can, for example, write clear and concise algorithms, and define combinatorial languages.

While this is a great benefit to programmers, it can be difficult for an implementation of the language. Many compilers implement some form of strictness analysis to make a program less lazy, to potentially conserve heap-space.

For years, compilers have implemented strictness optimizations for first-order functions. Some have even allowed programmers to explicitly specify unboxed types to aid the strictness analyzer. However, propagating and optimizing strictness information for higher-order functions has always been difficult.

In this paper we explore a method for efficiently compiling higher-order functions using explicitly provided strictness information. We look at UHC's TyCore intermediate language, and extend it to allow us to propagate strictness information throughout the compilation pipeline.

2 Background

2.1 Type-Based Strictness Analysis

In this paper we are only concerned with the actual strictness optimization, not the analysis. We therefor assume that the output of a strictness analyzer is already available.

The strictness information we take as input is in the form of type annotations. The information is generated by, for example, the type-based strictness analysis described in the paper "Making 'Stricterness' More Relevant" by Stefan Holdermans and Juriaan Hage. Below is an example of its output:

$$const :: \alpha^{\,S} \to \beta^{\,L} \to \alpha$$
$$const\ x\ y = x$$

The annotations S and L on the types convey the strictness properties of the *const* function. In this case, the function is strict in its first argument, and non-strict in its second.

However, next to the two concrete annotations, the analyzer can also output annotation variables. E.g.:

$$app :: (\alpha^{\,\varphi} \to \beta) \to \alpha^{\,\varphi} \to \beta$$
$$app\ f\ x = f\ x$$

This example shows that the strictness of x, the second argument to *app*, is the same as the strictness of the argument of f. In other words, if *app* is given a function that is strict in its argument, the second argument to *app* ideally should also be passed strictly. Conversely, if the function given to *app* is non-strict in its argument, the second argument should not be passed strictly, as this is not a safe optimization.

2.2 TyCore

The target language used in this paper is an extended version of TyCore. TyCore is a new typed intermediate language developed for the Utrecht Haskell Compiler (UHC) as an alternative for the current untyped Core. TyCore is itself based on Strict Core, a language developed by Max Bolingbroke and Simon Peyton-Jones in their paper "Types are Calling Conventions".

The language is strict by default, supports multiple arguments and return values and has first class type values. For example, the lazy polymorphic identity function is implemented as such:

$$id : \langle \alpha : \star \rangle \to \langle \{\alpha\} \rangle \to \langle \alpha \rangle$$
$$\quad = \lambda \langle \alpha : \star \rangle\ \langle x : \{\alpha\} \rangle \to \langle |x| \rangle$$

The *id* functions takes an explicit type argument α and a lazy value x, the lazy value is evaluated and returned.

3 First-Order Strictness Optimization

Strictness optimization for first-order functions is relatively straightforward. Lets look at the *const* function, annotated with strictness information provided by the analyzer:

$$const : \langle \{Int\}^{S} \rangle \to \langle \{Char\}^{L} \rangle \to \langle Int \rangle$$
$$= \lambda \langle x : \{Int\}^{S} \rangle \ \langle c : \{Char\}^{L} \rangle \to \langle |x| \rangle$$

Note that, for simplicity, we've made the *const* function monomorphic. To optimize this function, the optimizer can simply remove the laziness around strictly annotated types:

$$const : \langle Int \rangle \to \langle \{Char\} \rangle \to \langle Int \rangle$$
$$= \lambda \langle x : Int \rangle \ \langle c : \{Char\} \rangle \to \langle x \rangle$$

4 Higher-Order Strictness Optimization

As discussed before, higher-order functions can also be successfully annotated by the strictness analyzer. It will annotate the types with annotation variables, in a similar way that type inferencers come up with type variables.

We have extended TyCore to make these annotation variables first class, by adding a new *kind* to the language. That is, in the same way that we can supply types of kind $\star$, we can now also supply strictness properties of kind φ.

The following example demonstrates the 'function application'-function:

$$app : \langle \beta : \varphi \rangle \to \langle \langle Int^{\beta} \rangle \to \langle Int \rangle \rangle \to \langle Int^{\beta} \rangle \to \langle Int \rangle$$
$$= \lambda \langle \beta : \varphi \rangle \ \langle f : \langle Int^{\beta} \rangle \to \langle Int \rangle \rangle \ \langle x : Int^{\beta} \rangle \to f \ \langle x \rangle$$

We see that the strictness this function is now an explicit parameter β.

This also makes TyCore somewhat dependently typed, as the type of the *app* function depends on the value of β. However, we only allow this dependency for types and strictness properties, types may never depend on a value.

The *app* function is now polyvariant in the strictness of its second argument. However, we cannot compile such a function. In lower level languages, particularly in machine languages, strict function application is drastically different from lazy function application.

Strict function application means that an argument has to be fully evaluated, whereas lazy function application involves creating a thunk on the heap. Since these two types of code are different, when generating code, we have to pick one; No one piece of code can encompass both at the same time.

Our solution is to do a two-step process; First, we introduce a **case** on the strictness to handle both alternatives:

$$app : \langle \beta : \varphi \rangle \to \langle \langle Int^{\beta} \rangle \to \langle Int \rangle \rangle \to \langle Int^{\beta} \rangle \to \langle Int \rangle$$
$$= \lambda \langle \beta : \varphi \rangle \ \langle f : \langle Int^{\beta} \rangle \to \langle Int \rangle \rangle \ \langle x : Int^{\beta} \rangle \to$$
$$\textbf{case } \beta \textbf{ of}$$
$$S \to f \ \langle x \rangle$$
$$L \to f \ \langle \{|x|\} \rangle$$

Secondly, we use an inliner, and φ-partial evaluator at the call site to remove any reference to the type annotations. After the inliner and partial evaluator complete, the code generator will no longer have to deal with functions that are polyvariant in their strictness properties.

Untyped General Polymorphic Functions *

Martin Pettai

University of Tartu, Estonia
martinp@ut.ee

Abstract. In statically typed functional languages, functions are required to have a type. This also applies to polymorphic functions. Type soundness is achieved by proving for each function that it will not produce run-time type errors for any possible argument type. These proofs cannot always be deduced automatically by the compiler, essentially requiring the programmer to write the proof. This limits the amount of polymorphism that can be used in practice. We propose a solution to this by allowing untyped functions that can only branch and recurse on static (type-level) information. This is enough to define very general polymorphic functions, including higher-order polymorphic functions, but it allows all applications of untyped functions to be reduced during the compile time. Thus all types can still be checked statically and no run-time type errors can occur.

Keywords: Ad-hoc polymorphism, higher-order polymorphism, static type checking, type-level recursion, typecase

1 Introduction

1.1 Background and Motivation

In statically typed functional languages, types and values are usually strictly separated. Functions are defined on values. When a polymorphic function is defined on values, this also implies a function on types because the type of the value returned by a (polymorphic) function is uniquely defined by the type of the value given as its argument. The set of such functions on types that can be implicitly defined is usually quite restricted in statically typed languages, e.g. if only parametric polymorphism [6] is allowed.

Some functional languages have features that allow more general polymorphism, e.g. type classes [12]. These allow more functions on types to be implicitly defined. This additional polymorphism makes type checking and type inference more difficult for the compiler. Type inference often becomes undecidable. Even type checking (if the programmer annotates the types of functions) can become very hard, because the compiler has to prove for each function that application of the function does not cause a type error for any possible argument type (the set of possible arguments is defined by the type annotation).

Here is an example for Haskell type classes (with GHC extensions [11]):

* This work is partially supported by Estonian Science Foundation, grant no. 7543.

```
class C1 a ; instance C1 Int ; instance C1 a => C1 [[[[[[a]]]]]]
class C2 a ; instance C2 Int ; instance C2 a => C2 [[a]]
class C3 a ; instance C3 Int ; instance C3 a => C3 [[[a]]]
f :: (C2 a, C3 a) => a -> Int
f x = 2 * g x
g :: (C1 a) => a -> Int
g = undefined
```

To check the type-correctness of `f`, the compiler has to prove that the argument `x` (which has a type that belongs to classes `C2` and `C3`) can be used as an argument of `g` (which requires a type that belongs to class `C1`). The compiler cannot prove the required statement `(C2 a, C3 a) => C1 a` automatically. The programmer can help the compiler by adding an instance for `(C2 a, C3 a) => C1 a`. This requires expressing the methods of `C1` using only the methods of `C2` and `C3`. Essentially, the programmer has to write a proof of the statement `(C2 a, C3 a) => C1 a`, which can be considered to be a proof of type-correctness of `f`.

These proofs can be hard to write. This limits the amount of polymorphism that can be used in practice. Sometimes it is possible to statically determine all argument types (a finite number of them) with which the polymorphic function may be called. This is the case, for example, when the function and all its applications occur in the same module (i.e. they are not compiled separately). In this case it is not necessary to prove type-correctness for all possible argument types, it suffices to check it only for those argument types that may actually be used. If the programmer does not have to write the proofs then it would be easy to use more polymorphism than is practical in traditional statically typed languages. In this paper, we propose such an approach that obviates the need for proving type-correctness of polymorphic functions, allowing the programmer to easily write more general polymorphic functions.

The goal of this paper is to create a language where polymorphic functions can be defined almost as easily as monomorphic functions. The class of polymorphic functions that can be defined in this language should be large and include higher-order polymorphic functions and all first-order polymorphic functions whose implied function on types can be defined using only primitive recursion on types.

1.2 Our Solution—Untyped Functions

One way of using polymorphic functions without needing to prove their type-correctness for all possible argument types, is to use a dynamically typed language that has a typecase expression in addition to the ordinary case expression. In such a language, it would be possible to branch on types, and thus define polymorphic functions, as easily as defining monomorphic functions. For example, in Common Lisp [8], we can have the definition

```
(defun f (x)
  (etypecase x
```

```
(integer (code-char (+ 3 x)))
(number (* 10 x))
(character (* 2 (char-code x)))
(list (cdr x))))
```

after which the expression `(list (f 64) (f #\A) (f '(0 1 2)) (f 1.2))` is evaluated to `(#\C 130 (1 2) 12.0)`. Here, the implied function on types maps `integer` to `character`, other number types to themselves, `character` to `integer`, `list` to `list`, and all other types (e.g. `float`) to `string`. If the function is applied to a value of any other type (e.g. `string`), a type error will result.

The function `f` is an *untyped* function. It does not have an explicit type that can be used to check whether it can be applied to an argument of a given type and to determine the result type corresponding to the given argument type. Instead, these checks and computations on types will be performed by evaluating (starting with the beta-reduction) the function on the given argument. If during the evaluation a type error results (as in the case of the application `(f "str")`), the function was not applicable to the given argument and type checking fails. If the evaluation succeeds, it will result in a value whose type is the return type of the function for the given argument.

In dynamically typed languages, run-time computations are performed also on types, thus type errors can occur at run time. To detect type errors before run time, we propose to perform computations (reductions) in two phases.

In the first phase (this corresponds to type-check time), we perform reductions (e.g. typecases, type-level beta-reductions) that depend only on types, not on values. In the second phase (this corresponds to run time), we perform those reductions (e.g. branching on values, recursion on values) that depend on values. These reductions do not depend on types, thus type errors cannot occur in this phase. By separating the reductions into two phases, we have turned the dynamically typed language into a statically typed language.

Untyped functions can also be recursive and higher-order. To avoid type-level non-termination, we require some annotations from the programmer. These annotations are much simpler than type annotations because they are only used for verifying termination, not for type-correctness.

In Sect. 2, we will describe such a language that has higher-order untyped functions. While our goal is to create a statically typed language, we initially give (in Sect. 2.1) an informal description of our language as a dynamically typed language, i.e. without distinguishing the static and dynamic semantics. This is to make it easier to understand, because in our language the static and dynamic semantics are more similarly defined than in traditional statically typed functional languages. In our language, branching (reduction of case expressions) and beta-reductions are used in both static and dynamic semantics, making it natural to use an operational semantics (big-step reduction rules) for both.

In Sects. 2.2 and 2.3, we will make our language statically typed, by giving separate reduction rules for static and dynamic semantics. These correspond to the static and dynamic phases of the reduction process. In Sect. 2.4, we will prove type soundness of our language. In Sect. 2.5, we will describe an example

of a polymorphic function that is easy to define in our language but would be difficult to define using type classes.

In Sect. 3, we will give a denotational proof that type checking of our language always terminates.

1.3 Related Work

General Polymorphic Functions. Parametric polymorphism is included in many statically typed functional languages. One approach of implementing more general polymorphism has been type classes [12]. They were included already in Haskell 98 [4, 7] but have subsequently been extended [5]. Type classes use a relational approach. To define a polymorphic function, first a relation (type class) on types must be defined (or an existing one reused) and then the function on values can be defined, using the type class to restrict a parametrically polymorphic type. In contrast, we use a functional approach, where only a function on values must be defined, the function on types is implied, not separately defined.

In the relational approach, the return type of a polymorphic function cannot always be uniquely determined from the argument type. Functional dependencies [5] can be used to solve this problem in some cases. The relational approach also makes the syntax used for defining polymorphic functions (relational-style top-level declarations) very different from the syntax used for monomorphic functions (functional-style lambda-, let- and case-expressions). Higher-order polymorphic functions (except curried multi-argument functions) cannot be defined, because a type class cannot be given as argument to another type class.

A more functional-style alternative to Haskell type classes with functional dependencies is open type functions [9]. These allow functions on pure types to be defined using equations similar to those used for defining monomorphic functions on values. Higher-order functions cannot be defined.

Both open type functions and polymorphic functions defined using type classes are *open*, which means that after the initial definition, the domain of such a function can later (e.g. in other modules) be extended to other types (e.g. new types introduced in those modules) for which it is not yet defined. In contrast, an untyped function in our language is *closed*, it is defined once and cannot be extended without defining a new function with the same name.

Open functions are useful when separate compilation is used, i.e. different parts of the function definition can be compiled separately if they are in different modules. In our paper we consider only the case where separate compilation is not used, i.e. the whole function definition and all applications of the function are in the same module. This restriction allows us to support much more polymorphism easily.

Extensional polymorphism [3] is more similar to our approach. To define a polymorphic function, it uses a functional-style let-expression combined with a typecase-expression. Here the relation on types is implied. The typecase-expression can only be used to branch on the inferred monomorphic type of the polymorphic function, not on any type expression. A monomorphic type must be inferred or annotated for every application of a polymorphic function.

Higher-order polymorphic functions cannot be defined, because the functions must have a type but the type system is predicative.

Untyped Functions. Functions where the argument and return types are not explicitly specified (i.e. untyped functions) are often allowed in dynamically typed languages. Some of these languages have a typecase operator to branch on types. We saw an example of using a polymorphic untyped function in one such language, Common Lisp [8], in Sect. 1.2.

In dynamically typed languages, run-time type errors can occur. One way of reducing them is to use a static type inference algorithm to infer the types that can be statically determined, allowing to perform some type checks statically. Such an approach for Common Lisp is described in [2].

Another possibility is to start from a statically typed language and allow some type checks to be performed dynamically, e.g. by introducing a special type `Dynamic` whose values are pairs of a value and its type [1]. Values of any other type can then be converted to the `Dynamic` type. Functions of type `Dynamic` $\rightarrow$ `Dynamic` do not have argument and return types specified, i.e. they are untyped functions. These functions can branch on the type of their argument using a typecase construct. Unlike in our language, here the typecase constructs are evaluated during run time, making run-time type errors possible. Unlike in dynamically typed languages, run-time type errors can only occur in those parts of the program that use the `Dynamic` type.

This approach is especially useful when the type of the argument of a polymorphic function is not known before run time (e.g. it will be read from a file or from network), whereas our approach is most useful when the type of the argument is statically known for each application of the polymorphic function.

Staged Computations. The idea of performing computations in several stages has been implemented in MetaML [10]. The number of stages and the distinction between them (i.e. which computations are performed in each stage) is defined by the programmer, using explicit annotations, unlike in our language, where the phases are fixed by the reduction rules. The variables bound in earlier stages can be used in later stages but not vice versa. The types of expressions evaluated in later stages are known in earlier stages (the type system distinguishes between the types of expressions evaluated in different stages). Unlike our language, MetaML does not have a typecase operator that can be used in earlier stages to branch on the types of expressions of later stages.

2 Static Typing for a Dynamic Language

2.1 Syntax and Informal Semantics

The syntax of our language is given in Fig. 1. We start by considering it as a dynamic language. We give this language a call-by-value semantics. The language has a simple monomorphic type system containing only a unit type `Unit` (with a single value `unit`), a list type constructor `List` (with data constructors `nil`

```
TYPE ::= Unit | List TYPE | TYPE -> TYPE
BOUND ::= TYPE | Type TYPE | *
EXPR ::= VAR | VAR : TYPE | unit | nil EXPR | cons EXPR EXPR
       | typeof EXPR | EXPR EXPR
       | tlam VAR{NAT} ( VAR :< BOUND ) . EXPR
       | vlam VAR ( VAR : EXPR ) : EXPR . EXPR
       | iffun EXPR then EXPR else EXPR | iftype EXPR then EXPR else EXPR
       | tcase EXPR of Unit -> EXPR; List VAR -> EXPR; (VAR -> VAR) -> EXPR
       | vcase EXPR of nil -> EXPR; cons VAR VAR -> EXPR
       | Unit | List EXPR | EXPR -> EXPR
CLASS ::= TYPE | Type TYPE | Fun NAT BOUND
NAT ::= 0 | 1 | 2 | ...
```

Fig. 1. Syntax of the language

and `cons`), and a functional type constructor `->`. This is powerful enough to define functions on natural numbers, because `List Unit` can be used as a unary natural number type. Because the type system is monomorphic, the constructor `nil` has a type argument (an expression that reduces to a type) to construct an empty list of the specified element type, e.g. `nil Unit` is of type `List Unit`.

In addition to the ordinary expressions that have a type in `TYPE` and a value, the set of expressions `EXPR` also contains pure types (from `TYPE`) without a value and untyped functions constructed with `tlam`. To statically classify all type-correct expressions, we introduce the set of classes `CLASS` that in addition to the types of ordinary values, contains the classes of simple types used as expressions (e.g. the class of the expression `List Unit` is `Type (List Unit)`) and the classes of untyped functions (of the form `Fun` n d).

This language allows computing with both types (from the set `TYPE`) and values. Both can be passed as arguments of functions and also returned from functions. Branching can be done on types using the `tcase` construct, and on values using the `vcase` construct. The construct `typeof` can be used to compute the type of a value whose class is in `TYPE`. Constructs `iffun` and `iftype` can be used to branch on the class of expressions whose class can be outside `TYPE`.

The language has two constructs for defining functions, both of which allow recursion. The construct `tlam` allows recursion on types. For example,

```
(tlam f{0} (t :< *) .
           tcase t of Unit -> Unit;
                      List u -> (List Unit -> f u);
                      (u -> v) -> (f u -> f v))
        (List (List Unit))
==> List Unit -> List Unit -> Unit
```

The variable `f` is bound to the function that can be recursively called but with the type bound (after `:<`) decreased to the current argument. A function can only be applied to an argument smaller than the current bound. This guarantees

termination of type-level recursion. The set of bounds `BOUND` contains in addition to the ordinary types and the classes of pure types, a special bound `*`, the largest element of the set. The set is ordered by the partial order defined in Fig. 5.

The construct `tlam` also allows to define functions that can take as argument other untyped functions of a smaller kind. We use kinds to classify untyped functions. Kinds have also been used in other functional languages, e.g. in Haskell [7] they are used to classify type constructors. The kind (which is a natural number in our language) is determined by a kind annotation (e.g. `{0}` in the previous example). We note that kind and bound annotations are only included in the language to make type checking decidable. If decidability of type-checking is not required then kind and bound annotations can be dropped from the language. This would allow general recursion at the type level and thus would increase the set of functions that can be defined in the language.

The construct `vlam` allows general recursion on values of the specified type. This is equivalent to two lambda abstractions combined with a fixpoint operator, i.e. `vlam` f `(`x `:` t_1`)` `:` t_2 `.` e is equivalent to **fix** $(\lambda f : t_1 \to t_2.\ \lambda x : t_1.\ e)$ in typed lambda calculus with a fixpoint operator **fix**. Binding both the function and the argument has the advantage of making the language simpler because we do not need a separate fixpoint operator or let-expression in the core language.

The type annotations of the argument and return type can be expressions reducing to types not just constant types. This is useful for defining polymorphic functions that also use recursion over values but where the type of values that are recursed over depends on the argument type of the polymorphic function. For example, we can express the parametrically polymorphic function `foldr` without having polymorphic types:

```
tlam _{2} (f :< *) . tlam _{1} (c :< *) . tlam _{0} (xs :< *) .
   (vlam recurse (zs : typeof xs) : typeof c .
       vcase zs of
          nil       -> c;
          cons y ys -> f y (recurse ys)
   ) xs
```

We note that `_` is a variable identifier (belonging to the set `VAR`) in our language. We use this identifier when we do not care about the value bound to it, as is conventionally done in other functional languages (e.g. Haskell [7], although there it is a special construct, not an identifier).

Our language also has a typed variable (`VAR : TYPE`) as a separate construct. This can be considered to be an abstract constant of the specified type. It is mainly used during the reductions to facilitate type checking. and is not intended to be used directly by the programmer.

2.2 Static Semantics

In Sect. 2.1, we described our language as a dynamic language where computations with types and values are performed together in a single phase. We will now

describe how to perform the reductions in two phases, so that the reductions in the first phase (described in the current section) depend only on type-level information and the reductions in the second phase (described in Sect. 2.3) depend only on value-level information.

The reduction rules that can be applied in the static (type-level) phase are given in Fig. 2. These rules define the relation $\longrightarrow_t$, and they depend on the typing relation : defined in Fig. 4 and on the ordering relation $<$ defined in Fig. 5. Here and later, the values of variables are assumed to belong to the following sets:

$$x, x_i \in \texttt{VAR} \qquad e, e_i \in \texttt{EXPR} \qquad b, b_i \in \texttt{BOX} \qquad n, n_i \in \texttt{NAT}$$
$$t, t_i \in \texttt{TYPE} \qquad d, d_i \in \texttt{BOUND} \qquad c, c_i \in \texttt{CLASS}$$

Thus, in Figs. 2, 4, 5, and 7, these restrictions can be considered as additional premises to the rules.

We now define in Fig. 3 the type-level final expressions, i.e. the expressions that cannot be reduced further in the type level. We call these expressions box expressions (they can be viewed as boxes that have a value inside but this value cannot be seen at the type level and will only be revealed later at the value level). Box expressions b reduce to themselves (i.e. $b \longrightarrow_t b$) if they are type-correct. Every type-correct expression reduces to some box expression. An expression e is not type-correct if (and only if) it does not reduce to any expression at all (i.e. $\neg\exists e'(e \longrightarrow_t e')$).

The subset `BOX_v` contains those box expressions that may be reduced in the value level if they were type-correct. Other expressions will not be reduced any further in the value level (they either are already fully reduced or gave a type error during type-level reductions).

Because expressions are reduced only to box expressions in the type level, we give in Fig. 4 the type rules to determine the type of box expressions.

The body of `vlam` is reduced (in rule (VLAM)) to a box expression using only the type of the argument, not its value. To achieve this, the occurrences of the argument variable x in the body are replaced with an abstract constant $x : t$ of the same type. The same technique is used for the variables bound by the `cons`-branch of the `vcase` expression (in rule (VCASE)). For the `vcase` expression, we have the restriction that both branches must reduce to a value of the same type. The restriction is necessary to achieve type soundness. It is checked by the premises $b_2 : t_2$ and $b_3 : t_2$ of the (VCASE) rule (if this check fails, a type error is returned because no rule can be applied).

The type-level beta-reduction is performed using two different rules. The rule ($\mathsf{APP_B}$) is used when the argument of the application is an ordinary value or pure type. The rule ($\mathsf{APP_F}$) is used when the argument is an untyped function. These rules also check two restrictions on type-level functions that make type-checking decidable.

The first restriction is that the argument of an untyped function must be smaller than the function that is applied. To compare them, we define in Fig. 5 a well-founded and computable partial order. It must be computable to make the

$$\frac{}{x\ :\ t \longrightarrow_t x\ :\ t}\ (\mathsf{VAR}) \qquad \frac{e \longrightarrow_t b \qquad b : t}{\texttt{typeof}\ e \longrightarrow_t t}\ (\mathsf{TYPEOF}) \qquad \frac{}{\texttt{unit} \longrightarrow_t \texttt{unit}}\ (\mathsf{UNIT})$$

$$\frac{}{\texttt{Unit} \longrightarrow_t \texttt{Unit}}\ (\mathsf{TYPE_U}) \qquad \frac{e \longrightarrow_t t}{\texttt{List}\ e \longrightarrow_t \texttt{List}\ t}\ (\mathsf{TYPE_L}) \qquad \frac{e_1 \longrightarrow_t t_1 \qquad e_2 \longrightarrow_t t_2}{e_1\ \texttt{->}\ e_2 \longrightarrow_t t_1\ \texttt{->}\ t_2}\ (\mathsf{TYPE_F})$$

$$\frac{e \longrightarrow_t t}{\texttt{nil}\ e \longrightarrow_t \texttt{nil}\ t}\ (\mathsf{NIL}) \qquad \frac{e_1 \longrightarrow_t b_1 \qquad b_1 : t \qquad e_2 \longrightarrow_t b_2 \qquad b_2 : \texttt{List}\ t}{\texttt{cons}\ e_1\ e_2 \longrightarrow_t \texttt{cons}\ b_1\ b_2}\ (\mathsf{CONS})$$

$$\frac{}{\texttt{tlam}\ x_1\{n\}\ (\ x_2\ \texttt{:<}\ d\ )\ .\ e \longrightarrow_t \texttt{tlam}\ x_1\{n\}\ (\ x_2\ \texttt{:<}\ d\ )\ .\ e}\ (\mathsf{TLAM})$$

$$\frac{e_1 \longrightarrow_t t_1 \qquad e_2 \longrightarrow_t t_2 \qquad e_3[(x_1\ :\ (t_1\ \texttt{->}\ t_2))/x_1, (x_2\ :\ t_1)/x_2] \longrightarrow_t b_3 \qquad b_3 : t_2}{\texttt{vlam}\ x_1\ (\ x_2\ :\ e_1\ )\ :\ e_2\ .\ e_3 \longrightarrow_t \texttt{vlam}\ x_1\ (\ x_2\ :\ t_1\ )\ :\ t_2\ .\ b_3}\ (\mathsf{VLAM})$$

$$\frac{e_1 \longrightarrow_t b_1 \qquad b_1 : \texttt{Fun}\ n\ d \qquad e_2 \longrightarrow_t b_2}{\texttt{iffun}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow_t b_2}\ (\mathsf{IFFUN_+})$$

$$\frac{e_1 \longrightarrow_t b_1 \qquad b_1 : c \qquad \neg\exists n \exists d (c = \texttt{Fun}\ n\ d) \qquad e_3 \longrightarrow_t b_3}{\texttt{iffun}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow_t b_3}\ (\mathsf{IFFUN_-})$$

$$\frac{e_1 \longrightarrow_t b_1 \qquad b_1 : \texttt{Type}\ t \qquad e_2 \longrightarrow_t b_2}{\texttt{iftype}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow_t b_2}\ (\mathsf{IFTYPE_+})$$

$$\frac{e_1 \longrightarrow_t b_1 \qquad b_1 : c \qquad \neg\exists t (c = \texttt{Type}\ t) \qquad e_3 \longrightarrow_t b_3}{\texttt{iftype}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow_t b_3}\ (\mathsf{IFTYPE_-})$$

$$\frac{e_1 \longrightarrow_t \texttt{Unit} \qquad e_2 \longrightarrow_t b_2}{\texttt{tcase}\ e_1\ \texttt{of Unit ->}\ e_2\texttt{; List}\ x_1\ \texttt{->}\ e_3\texttt{; (}x_2\ \texttt{->}\ x_3\texttt{) ->}\ e_4 \longrightarrow_t b_2}\ (\mathsf{TCASE_U})$$

$$\frac{e_1 \longrightarrow_t \texttt{List}\ t_1 \qquad e_3[t_1/x_1] \longrightarrow_t b_3}{\texttt{tcase}\ e_1\ \texttt{of Unit ->}\ e_2\texttt{; List}\ x_1\ \texttt{->}\ e_3\texttt{; (}x_2\ \texttt{->}\ x_3\texttt{) ->}\ e_4 \longrightarrow_t b_3}\ (\mathsf{TCASE_L})$$

$$\frac{e_1 \longrightarrow_t t_2\ \texttt{->}\ t_3 \qquad e_4[t_2/x_2, t_3/x_3] \longrightarrow_t b_4}{\texttt{tcase}\ e_1\ \texttt{of Unit ->}\ e_2\texttt{; List}\ x_1\ \texttt{->}\ e_3\texttt{; (}x_2\ \texttt{->}\ x_3\texttt{) ->}\ e_4 \longrightarrow_t b_4}\ (\mathsf{TCASE_F})$$

$$\frac{\begin{array}{c} e_1 \longrightarrow_t b_1 \qquad b_1 : \texttt{List}\ t_1 \qquad e_2 \longrightarrow_t b_2 \qquad b_2 : t_2 \\ e_3[(x_1\ :\ t_1)/x_1, (x_2\ :\ \texttt{List}\ t_1)/x_2] \longrightarrow_t b_3 \qquad b_3 : t_2 \end{array}}{\texttt{vcase}\ e_1\ \texttt{of nil ->}\ e_2\texttt{; cons}\ x_1\ x_2\ \texttt{->}\ e_3 \longrightarrow_t \texttt{vcase}\ b_1\ \texttt{of nil ->}\ b_2\texttt{; cons}\ x_1\ x_2\ \texttt{->}\ b_3}\ (\mathsf{VCASE})$$

$$\frac{\begin{array}{c} e_1 \longrightarrow_t \texttt{tlam}\ x_1\{n_1\}\ (\ x_2\ \texttt{:<}\ d_1\ )\ .\ e_3 \qquad e_2 \longrightarrow_t b_2 \qquad b_2 : d_2 \qquad d_2 < d_1 \\ e_3[b_2/x_2, (\texttt{tlam}\ x_1\{n_1\}\ (\ x_2\ \texttt{:<}\ d_2\ )\ .\ e_3)/x_1] \longrightarrow_t b_3 \qquad b_3 : c_3 \qquad c_3 < \texttt{Fun}\ n_1\ d_1 \end{array}}{e_1\ e_2 \longrightarrow_t b_3}\ (\mathsf{APP_B})$$

$$\frac{\begin{array}{c} e_1 \longrightarrow_t \texttt{tlam}\ x_1\{n_1\}\ (\ x_2\ \texttt{:<}\ d_1\ )\ .\ e_3 \qquad e_2 \longrightarrow_t b_2 \qquad b_2 : \texttt{Fun}\ n_2\ d_2 \qquad \texttt{Fun}\ n_2\ d_2 < \texttt{Fun}\ n_1\ d_1 \\ e_3[b_2/x_2, (\texttt{tlam}\ x_1\{n_2\}\ (\ x_2\ \texttt{:<}\ d_2\ )\ .\ e_3)/x_1] \longrightarrow_t b_3 \qquad b_3 : c_3 \qquad c_3 < \texttt{Fun}\ n_1\ d_1 \end{array}}{e_1\ e_2 \longrightarrow_t b_3}\ (\mathsf{APP_F})$$

$$\frac{e_1 \longrightarrow_t b_1 \qquad b_1 : (t_1\ \texttt{->}\ t_2) \qquad e_2 \longrightarrow_t b_2 \qquad b_2 : t_1}{e_1\ e_2 \longrightarrow_t b_1\ b_2}\ (\mathsf{APP_V})$$

Fig. 2. Type-level reduction rules

```
BOX ::= BOX_v | TYPE | tlam VAR{NAT} ( VAR :< BOUND ) . EXPR
BOX_v ::= VAR : TYPE | unit | nil TYPE | cons BOX_v BOX_v | BOX_v BOX_v
        | vlam VAR ( VAR : TYPE ) : TYPE . BOX_v
        | vcase BOX_v of nil -> BOX_v; cons VAR VAR -> BOX_v
```

Fig. 3. Box expressions

$$\frac{}{(x\ :\ t) : t}\ (\mathsf{Var}) \qquad \frac{b_1 : (t_1\ \texttt{->}\ t_2) \qquad b_2 : t_1}{b_1\ b_2 : t_2}\ (\mathsf{App}) \qquad \frac{}{t : \texttt{Type}\ t}\ (\mathsf{Type})$$

$$\frac{}{\texttt{unit} : \texttt{Unit}}\ (\mathsf{Unit}) \qquad \frac{}{\texttt{nil}\ t : \texttt{List}\ t}\ (\mathsf{Nil}) \qquad \frac{b_1 : t \qquad b_2 : \texttt{List}\ t}{\texttt{cons}\ b_1\ b_2 : \texttt{List}\ t}\ (\mathsf{Cons})$$

$$\frac{}{\texttt{tlam}\ x_1\{n\}\ \texttt{(}\ x_2\ \texttt{:<}\ d\ \texttt{)}\ \texttt{.}\ e : \texttt{Fun}\ n\ d}\ (\mathsf{Tlam})$$

$$\frac{b : t_2}{(\texttt{vlam}\ x_1\ \texttt{(}\ x_2\ \texttt{:}\ t_1\ \texttt{)}\ \texttt{:}\ t_2\ \texttt{.}\ b) : (t_1\ \texttt{->}\ t_2)}\ (\mathsf{Vlam})$$

$$\frac{b_1 : \texttt{List}\ t_1 \qquad b_2 : t_2 \qquad b_3 : t_2}{(\texttt{vcase}\ b_1\ \texttt{of nil ->}\ b_2\texttt{;}\ \texttt{cons}\ x_1\ x_2\ \texttt{->}\ b_3) : t_2}\ (\mathsf{Vcase})$$

Fig. 4. Type rules for box expressions

relation $\longrightarrow_t$ computable. It must be well-founded to ensure that there cannot be an infinite recursion if the type of the recursive function is decreased during each recursive call. The restriction is checked by the premise $d_2 < d_2$ of the ($\mathsf{APP_B}$) rule and the premise $\texttt{Fun}\ n_2\ d_2 < \texttt{Fun}\ n_1\ d_1$ of the ($\mathsf{APP_F}$) rule.

$$\frac{t_1 < t_2}{\texttt{Type}\ t_1 < \texttt{Type}\ t_2} \qquad \frac{n_1 <_{\texttt{NAT}} n_2}{\texttt{Fun}\ n_1\ d_1 < \texttt{Fun}\ n_2\ d_2} \qquad \frac{d_1 < d_2}{\texttt{Fun}\ n\ d_1 < \texttt{Fun}\ n\ d_2}$$

$$\frac{}{t < \texttt{List}\ t} \qquad \frac{}{t_1 < (t_1\ \texttt{->}\ t_2)} \qquad \frac{}{t_2 < (t_1\ \texttt{->}\ t_2)} \qquad \frac{}{t < \texttt{*}}$$

$$\frac{t_1 < t_2 \qquad t_2 < t_3}{t_1 < t_3} \qquad \frac{}{t_1 < \texttt{Type}\ t_2} \qquad \frac{}{t < \texttt{Fun}\ n\ d} \qquad \frac{}{\texttt{Type}\ t < \texttt{Fun}\ n\ d}$$

Fig. 5. A well-founded and computable order on $\texttt{CLASS} \cup \{\texttt{*}\}$

The second restriction is that the value returned from the application must also be smaller than the function that was applied. This is checked by the premise $c_3 < \texttt{Fun}\ n_1\ d_1$ of the rules ($\mathsf{APP_B}$) and ($\mathsf{APP_F}$). If the returned value could be larger than the applied function then we could give a function as argument to itself by wrapping it inside a smaller function that returns this larger function.

When an untyped function `f` is applied to a typed argument or pure type, only the bound is decreased for the recursive copies of `f`, the kind is left unchanged. This is handled by the ($\mathsf{APP_B}$) rule. This allows a recursive untyped function to take an untyped function as a second (curried) argument after being recursively called on a typed argument. This is used in the function `proc` in Sect. 2.5.

2.3 Dynamic Semantics

We first define in Fig. 6 the final expressions, i.e. the expressions that reduce to themselves. Here and later variables a, a_i denote final expressions. Any box expression reduces in the value level to either a final expression or not to any expression at all. In the latter case it is a run-time error.

```
FINAL ::= TYPE | VAR : TYPE | unit | nil TYPE | cons FINAL FINAL
        | tlam VAR{NAT} ( VAR :< BOUND ) . EXPR
        | vlam VAR ( VAR : TYPE ) : TYPE . BOX_v
```

Fig. 6. Final expressions

In Fig. 7, we now give the reduction rules that can be applied in the dynamic (value-level) phase. These will only be given for type-correct box expressions in the subset `BOX_v`. Other type-correct box expressions (in the set `BOX` $\setminus$ `BOX_v`) will reduce to themselves. Because the rules do not actually use type information, we

$$\frac{}{\texttt{unit} \longrightarrow_v \texttt{unit}}\ (\mathsf{unit})$$

$$\frac{}{\texttt{nil}\ t \longrightarrow_v \texttt{nil}\ t}\ (\mathsf{nil}) \qquad \frac{b_1 \longrightarrow_v a_1 \qquad b_2 \longrightarrow_v a_2}{\texttt{cons}\ b_1\ b_2 \longrightarrow_v \texttt{cons}\ a_1\ a_2}\ (\mathsf{cons})$$

$$\frac{}{\texttt{vlam}\ x_1\ \texttt{(}\ x_2\ \texttt{:}\ t_1\ \texttt{)}\ \texttt{:}\ t_2\ \texttt{.}\ b_3 \longrightarrow_v \texttt{vlam}\ x_1\ \texttt{(}\ x_2\ \texttt{:}\ t_1\ \texttt{)}\ \texttt{:}\ t_2\ \texttt{.}\ b_3}\ (\mathsf{vlam})$$

$$\frac{b_1 \longrightarrow_v \texttt{nil}\ t \qquad b_2 \longrightarrow_v a_2}{\texttt{vcase}\ b_1\ \texttt{of nil ->}\ b_2\texttt{;}\ \texttt{cons}\ x_1\ x_2\ \texttt{->}\ b_3 \longrightarrow_v a_2}\ (\mathsf{vcase_n})$$

$$\frac{b_1 \longrightarrow_v \texttt{cons}\ a_1\ a_2 \qquad b_3[a_1/(x_1\ \texttt{:}\ t), a_2/(x_2\ \texttt{:}\ \texttt{List}\ t)] \longrightarrow_v a_3}{\texttt{vcase}\ b_1\ \texttt{of nil ->}\ b_2\texttt{;}\ \texttt{cons}\ x_1\ x_2\ \texttt{->}\ b_3 \longrightarrow_v a_3}\ (\mathsf{vcase_c})$$

$$\frac{b_1 \longrightarrow_v \texttt{vlam}\ x_1\ \texttt{(}\ x_2\ \texttt{:}\ t_1\ \texttt{)}\ \texttt{:}\ t_2\ \texttt{.}\ b_3 \qquad b_2 \longrightarrow_v a_2 \qquad b_3[a_2/(x_2\ \texttt{:}\ t_1), (\texttt{vlam}\ x_1\ \texttt{(}\ x_2\ \texttt{:}\ t_1\ \texttt{)}\ \texttt{:}\ t_2\ \texttt{.}\ b_3)/(x_1\ \texttt{:}\ t_1\ \texttt{->}\ t_2)] \longrightarrow_v a_3}{b_1\ b_2 \longrightarrow_v a_3}\ (\mathsf{app})$$

Fig. 7. Value-level reduction rules

can drop the type annotations before performing the dynamic reduction phase. This requires dropping types from the value-level language (currently the type-level and value-level languages have the same syntax, defined in Fig. 1) and value-level reduction rules (defined in Fig. 7).

2.4 Type Soundness

The following theorem states that every expression that is a result of type-level reductions is well-typed and its further reduction at the value level will not cause type errors and will preserve types. We note that `FINAL` is a subset of `BOX` and thus the type rules in Fig. 4 can also be applied to final expressions.

Theorem 1. *If $e \longrightarrow_t b$ where e does not contain free variables or abstract constants then $b : t$ for some t and $b \longrightarrow_v a$ such that $a : t$.*

We first prove the next theorem, from which this theorem easily follows. We note that if e does not contain free variables (typed or untyped) then b has the same

property, because the rules in Fig. 2 do not introduce new free variables, they only change some bound variables from untyped to typed.

Theorem 2. *If $e \longrightarrow_t b$ then $b : t$ for some t and for all b' that can be obtained from b by replacing all typed free variables $(x_0 \ : \ t_0)$ by final expressions of the same type (t_0), $b' \longrightarrow_v a$ such that $a : t$.*

We prove this theorem by induction on the structure of the derivation tree of $e \longrightarrow_t b$. We consider separate cases according to the rule that is applied at the bottom of the tree. The statement $b : t$ is easy to check for all cases because type assertions in the rules in Fig. 2 mirror those in the corresponding rules in Fig. 4.

Now we consider the statement $b' \longrightarrow_v a$ such that $a : t$. For the rules (TYPEOF), ($\mathsf{TYPE_U}$), ($\mathsf{TYPE_L}$), ($\mathsf{TYPE_F}$), and (TLAM) it is obvious because the box expressions resulting from these rules (even after replacing free variables; we note that replacing free variables by final expressions of the same type does not change the type of the whole expression because the typing rules in Fig. 4 are compositional) do not belong to the set `BOX_v` and are thus implicitly reduced to themselves in the value level. For the rules (UNIT), (NIL), and (VLAM) it is also obvious because the box expressions resulting from these rules (even after replacing free variables) are explicitly reduced to themselves by the corresponding value-level rules. For the rules ($\mathsf{IFFUN_+}$), ($\mathsf{IFFUN_-}$), ($\mathsf{IFTYPE_+}$), ($\mathsf{IFTYPE_-}$), ($\mathsf{TCASE_U}$), ($\mathsf{TCASE_L}$), ($\mathsf{TCASE_F}$), ($\mathsf{APP_B}$), and ($\mathsf{APP_F}$) it is obvious because those rules contain in their premises a statement $e' \longrightarrow_t b$ for some e' and thus these cases can be directly reduced to the induction hypothesis. For the (VAR) rule it is also obvious, because if the typed variable is replaced by any final expression of the same type, this final expression reduces to itself in the value level. For the (CONS) rule the induction step is also easy, we use the induction hypothesis for the premises $e_1 \longrightarrow_t b_1$ and $e_2 \longrightarrow_t b_2$.

Now we consider the rule (VCASE). We use the induction hypothesis for the premise $e_1 \longrightarrow_t b_1$ to get $b'_1 \longrightarrow_v a_1$ and a_1 `:` `List` t_1 (here b'_1 is the expression obtained from b_1 by replacing its free variables in the same way as in b'). Because a_1 has list type and is a result of value-level reductions, it must have a `nil` or `cons` head (expressions with a `unit` or `vlam` head would not have list type), thus the first premise of either ($\mathsf{vcase_n}$) or ($\mathsf{vcase_c}$) is satisfied. To get the second premise, we use the induction hypothesis for $e_2 \longrightarrow_t b_2$ and $e_3[(x_1 \ : \ t_1)/x_1, (x_2 \ :$ `List` $t_1)/x_2] \longrightarrow_t b_3$ to get $b'_2 \longrightarrow_v a_2$ and $b'_3[a_1/(x_1 \ : \ t_1), a_2/(x_2 \ :$ `List` $t_1)] \longrightarrow_v a_3$ (here b'_2 and b'_3 use the same free variable replacements as b', and in b'_3 additional variables $x_1 \ : \ t_1$ and $x_2 \ :$ `List` t_1 are replaced, which are not free in b but are free in b_3). Thus we can use one of the rules ($\mathsf{vcase_n}$) and ($\mathsf{vcase_c}$) to get $b' \longrightarrow_v a$ with $a : t$.

Now we consider the rule ($\mathsf{APP_V}$). This case is similar to the case of (VCASE). Here we use the induction hypothesis for the premise $e_1 \longrightarrow_v b_1$ to get $b'_1 \longrightarrow_v a$ where $a : t_1$ `->` t_2 and thus a must be a `vlam` expression, satisfying the first premise of (app). The second premise is obviously satisfied. For the third premise, the variables $x_2 : t_1$ and $x_1 : t_1$ `->` t_2 (which are free in b_3 but not in b) are replaced in addition to the replacements in b'.

This concludes the proof.

2.5 An Example

Here is an example of a polymorphic function in our language. It takes as an argument any value of a functional type (t_1 `->` ... `->` t_n `->` t where t is non-functional) and returns the function with its (curried) arguments reversed (i.e. the return value is of type t_n `->` ... `->` t_1 `->` t).

```
tlet reverseargs f =
   tlet revtypes {2} t {1} cont =
      tcase t of
         Unit       -> cont t;
         List _     -> cont t;
         (t1 -> t2) -> revtypes t2 (tlam u . t1 -> cont u)
   in tlet proc {2} ts {1} cont =
      tcase ts of
         Unit       -> cont f;
         List _     -> cont f;
         (t -> ts') ->
            vlam (x : t) : ts' . proc ts' (tlam g . cont (g x))
   in
      proc (revtypes (typeof f) (tlam t . t)) (tlam g . g)
```

Here we use the following syntactic sugar:

$$\texttt{vlam } (x_1 \texttt{ : } t_1) \texttt{ : } t_2 \texttt{ . } e \equiv \texttt{vlam \_ } (x_1 \texttt{ : } t_1) \texttt{ : } t_2 \texttt{ . } e$$

$$\texttt{tlam } x\{n\} \texttt{ } (x_1) \texttt{ . } e \equiv \texttt{tlam } x\{n\} \texttt{ } (x_1 \texttt{ :< *}) \texttt{ . } e$$

$$\texttt{tlam } x_1 \texttt{ . } e \equiv \texttt{tlam \_\{0\} } (x_1 \texttt{ :< *}) \texttt{ . } e$$

$$\texttt{tlet } x \texttt{ } \{n_1\} \texttt{ } x_1 \texttt{ } \{n_2\} \texttt{ } x_2 \texttt{ ... } \{n_k\} \texttt{ } x_k \texttt{ = } e_1 \texttt{ in } e_2 \equiv$$
$$\equiv \texttt{(tlam \_\{}(n_1 + 1)\texttt{\} (}x\texttt{) . } e_2\texttt{) (tlam } x\{n_1\} \texttt{ } (x_1) \texttt{ .}$$
$$\texttt{tlam \_\{}n_2\texttt{\} (}x_2\texttt{) . ... tlam \_\{}n_k\texttt{\} (}x_k\texttt{) . } e_1\texttt{)}$$

As we see from the example, the functions `reverseargs`, `revtypes`, and `proc` do not need a type annotation (neither given by the programmer nor inferred by the compiler) that would specify the possible pairs of argument and return types. We only need some kind annotations for higher-order untyped functions.

The function `reverseargs` is an example of a polymorphic function that would be difficult to define using type classes (I have not been able to define it). Even if it can be defined, it would require a different algorithm, because our algorithm uses an accumulator `cont` to build type-level functions, but type classes do not support higher-order type-level functions (except constructors).

3 Denotations of Higher-Order Polymorphic Functions and Termination of Type Checking

In this section we prove that the type-level phase of the reduction process of our language always terminates. We first define the denotations of expressions.

Then we define the set of *allowed* values. These are the values that cannot contain type-level non-termination. Finally we prove that the denotations of all expressions are *allowed* values.

Let T be the set of all monomorphic types (the set $\texttt{TYPE}$) and V_t the set of values of the type $t \in T$. We use final expressions as the values, i.e. $V_t = \{a \in \texttt{FINAL} \mid a : t\}$. Thus final expressions denote themselves. Let $\bot_t$ be the special value corresponding to value-level non-termination (or other value-level error) in type t and let $\bot$ and $\bot'$ be the special values corresponding to a type error and type-level non-termination, respectively (these are different).

Then the set of all values that are not untyped functions, is $S'_0 = \{\bot', \bot\} \cup T \cup \bigcup\{V_t \cup \{\bot_t\} \mid t \in T\}$. Let $\texttt{FUNCLASS} = \{\texttt{Fun}\ \ n\ \ d \mid n \in \texttt{NAT} \wedge d \in \texttt{BOUND}\}$ be the subset of $\texttt{CLASS}$ that is the set of classes that untyped functions can have. Then S'_0 is the set of values of all non-functional classes (i.e. those in the set $\texttt{CLASS} \setminus \texttt{FUNCLASS}$, and in addition $\bot$ and $\bot'$).

The denotation of a non-functional expression e is a value in S'_0. If $e \longrightarrow_t b \longrightarrow_v a$ then a is used as the denotation of e. If $e \longrightarrow_t b$ but the value-level reduction b of leads to non-termination then the denotation of e is $\bot_t$ where $b : t$. If the type-level reduction of e leads to a type error then the denotation of e is $\bot$. If it does not terminate then the denotation of e is $\bot'$.

The denotation of an expression of class $c \in \texttt{FUNCLASS}$ is a mathematical function with domain Z_c (the set Z_c is defined later in this section). The codomain is initially left unspecified. This function is total because those arguments in Z_c for which the untyped function is not defined, are mapped to $\bot$, $\bot_t$, or $\bot'$.

We also need to clarify the denotation of type-level application. If the denotation of a functional-class expression e_1 is f and the denotation of e_2 is v then the denotation of $e_1\ \ e_2$ is $f(v)$ if v is in the domain of f and $\bot$ otherwise.

The set of *allowed* values of all non-functional classes is $S_0 = S'_0 \setminus \{\bot'\}$, i.e. all non-functional values, except type-level non-termination, are allowed. The set of *allowed* values for class $c \in \texttt{FUNCLASS}$ is $S_c = Z_c \to Z_c$ where the set of values that are allowed to be given as arguments to and to be returned from an untyped function of class c is

$$Z_c = S_0 \cup \bigcup\{S_{c'} \mid c' \in \texttt{FUNCLASS} \wedge c' < c\}$$

which includes all *allowed* non-functional values and *allowed* untyped functions of classes smaller than c. This definition uses transfinite recursion to define S_c for all $c \in \texttt{FUNCLASS}$, which is possible because the order $<$ here is the well-founded order from Fig. 5 (restricted to the set $\texttt{FUNCLASS}$). The set of all *allowed* values can now be defined as

$$S = S_0 \cup \bigcup\{S_c \mid c \in \texttt{FUNCLASS}\}$$

We note that the denotation of a type-level application of an *allowed* value to an *allowed* value is always an *allowed* value, even if the argument is not in the domain of the applied function, because in that case the result is $\bot$, which is an *allowed* value.

Now we use structural induction on the set of `tlam`-expressions (i.e. we can use the induction hypothesis for the inner `tlam`-expressions contained as subexpressions of the body of the current `tlam`-expression, even if there are non-`tlam` constructs in between) to prove that the denotation of every `tlam`-expression is an *allowed* value if all free variables in it are bound to *allowed* values.

By the induction hypothesis, we can also assume that all inner `tlam`-expressions inside the current `tlam`-expression f denote *allowed* values.

To handle type-level recursion, we use an inner induction over the recursive copies of f (i.e. a transfinite induction over classes (the well-ordered set `CLASS`), because class is the only thing that changes in the recursive copies).

Thus we can also assume by the hypothesis of the inner induction that occurrences of the recursive variable inside f are bound to *allowed* values.

Let the class of f be c. Then the denotation of f is a function with domain Z_c. To prove that it is a function from Z_c to Z_c (i.e. Z_c can be its codomain), we now consider what happens if f is applied to a value in Z_c (i.e. an *allowed* value of a class smaller than c). After performing the beta-reduction, we have an expression e that does not contain as subexpressions any `tlam`-expression that does not denote an *allowed* value (here we use the hypothesis of the outer induction for the `tlam`-expressions that are subexpressions of e; the free variables introduced by the beta-reduction (the recursive variable and the argument variable) have been bound to allowed values).

The expression e contains only a finite number of type-level function applications outside `tlam`-subexpressions. Each of those will be reduced only once, because expressions that are not inside `tlam`-expressions cannot be duplicated (only beta-reduction can duplicate subexpressions). Type-level reduction of each application terminates because the applied `tlam`-expression is an *allowed* value. Besides type-level applications, e contains a finite number of other constructs, each of which adds a finite number of steps to the type-level reduction of e.

Thus the evaluation of e takes a finite number of reduction steps, i.e. it terminates. Thus e is reduced to a box expression b. If b is not a `tlam`-expression then b denotes a value in S_0. If b is a `tlam`-expression then it is either one of the `tlam`-expressions that are subexpressions of e (those denote *allowed* values) or is obtained from those `tlam`-expressions by applying one of them to one or more arguments (these returned values are also *allowed* because a functional *allowed* value can only return *allowed* values). Thus b denotes an *allowed* value. If the class of b is smaller than c then the value of b is returned from f, otherwise a type error (an *allowed* value $\perp \in Z_c$) is returned.

Thus the denotation of f is a function that returns a value in Z_c for each argument in Z_c. Thus f is an *allowed* value of class c.

Thus for every program (every expression without free variables) all `tlam`-expressions contained in it denote *allowed* values. This implies (using the same argument as in the inductive step for proving that e denotes an *allowed* value) that the whole program is an *allowed* value. Thus the evaluation of the program cannot produce type-level non-termination, i.e. the type-level phase of the reduction process always terminates.

4 Conclusion

We have created a language that combines the advantages of static and dynamic languages. We showed how to use untyped functions that usually belong to dynamically typed languages, in a statically typed language, to define very general polymorphic functions, including higher-order polymorphic functions. Untyped functions allow defining computations (e.g. branching, recursion) on types almost as easily as on values, making those polymorphic functions easy to define.

We eliminated the possibility of run-time type errors by performing computations in two phases—computations on types at type-check time and computations on values at run time. To maintain decidability of type checking, recursion on types was restricted to a variant of primitive recursion combined with higher-order functions. On values, general recursion was still allowed.

Untyped functions allowed using polymorphic functions in a monomorphic type system. While we have not yet implemented the language, a monomorphic type system should make it easier to implement than a polymorphic type system.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.
2. H. G. Baker. The Nimble type inferencer for Common Lisp-84. Technical report, Nimble Comp., 1990.
3. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 118–129, New York, NY, USA, 1995. ACM.
4. C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18:241–256, 1996.
5. M. P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.
6. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
7. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.
8. K. M. Pitman. *Common Lisp HyperSpec*. 1996. `http://www.lispworks.com/documentation/HyperSpec/Front/`.
9. T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM.
10. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.
11. The GHC Team. The Glasgow Haskell Compiler home page. `http://haskell.org/ghc/`, 2010.
12. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

The Usual Tasks: A Library for Ad-Hoc Work in iTasks

[Extended Abstract]

Bas Lijnse[1,2] and Erik Crombag[1] and Rinus Plasmeijer[1]

[1] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
[2] Faculty of Military Sciences,
Netherlands Defence Academy, Den Helder, the Netherlands
{b.lijnse,e.crombag,rinus}@cs.ru.nl

For many day-to-day tasks of a white-collar worker, generic applications like e-mail, word processors or spreadsheets, suffice to get the job done. Moreover, the cost of having specialized tools for every task does not outweigh the overhead of using generic applications. If an organization implements a workflow management system (WFMS) to support their business processes, it will also only support those tasks for which the benefits outweigh the cost of implementation in the WFMS. Hence, the tasks of a worker are divided into a set of tasks under control of the WFMS, and a remainder ad-hoc set using generic tools. This division can be undesirable for both workers and managers. For workers, it means working from two task lists and having information in multiple systems. This can be inconvenient especially when ad-hoc tasks supplement tasks in the WFMS. For managers, it reduces the accuracy of the information that a WFMS can provide for resource planning and control. Since not all tasks are visible to the WFMS, a worker who appears to have no ongoing work, may have an overflowing e-mail inbox and a large to-do list. This paper presents a library for the iTasks WFMS that facilitates common ad-hoc tasks *within* an iTasks based system. The library provides tasks for creating groups of users, e-mail like messaging, creating and sharing lists and simple group decision support. This selection of ad-hoc tasks has been derived through analysis and clustering of the ad-hoc work involved in organizing a small conference, collected through a workshop with a group of faculty members experienced in this area.

iTask as a new paradigm to building GUI applications – extended abstract –

Steffen Michels, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
S.Michels@student.ru.nl, {rinus, p.achten}@cs.ru.nl

1 Introduction

The *iTask* system [21, 20] is a *workflow management system* (WFMS) which uses a workflow description language (WDL) embedded in the functional language *Clean* as a combinator library. There are many advantages to this approach. First, the embedding in a *general purpose programming language* adds recursion and arbitrary computations to the WDL. Second, the use of a *functional* language adds higher-order tasks: workflows that can accept and / or compute workflows during execution. Third, the use of *Clean* adds generic programming facilities [4] which is used intensively to abstract away from a lot of boilerplate programming [17]. As a result, the *iTask* WDL is a rich and high level formalism to create workflow applications.

An additional advantage of the *iTask* system is that it is currently implemented using web browser technology. This makes it readily available to any platform that provides a standard web browser. The programmer who uses the *iTask* WDL to create multi-user web-enabled workflows does not have to concern herself about the technological challenges that come with programming web applications: the generic foundations of *iTask* abstract from these concerns.

The key contribution of this paper is that we generalize the application domain of the *iTask* WDL, and show how to adapt the *iTask* combinator library in such a way that it is a new paradigm for programming distributed GUI applications. There are several reasons to substantiate this claim. First, traditional widget-callback GUI paradigms force the programmer to break the program logic in terms of callback functions. In such approaches, the programmer also has to carefully arrange the handling and identification of state to handle the proper 'communication' between subsequent callbacks. In *iTask*, the program is a single, typically recursive, expression that expresses the behavior of the program in terms of user input. Second, in traditional widget-based GUI paradigms, the programmer is responsible for the entire life-cycle of the GUI elements: creation, management, event handling, and destruction all need to be programmed explicitly and carefully. In *iTask*, most of this is handled by means of the generic interface, and the programmer is only bothered with pure functional data types that model the GUI, rather than implement it. Third, past experience with using

iTask for a non-workflow interactive application [13] turned out to be successful: it was both possible to add a rich new GUI component to the framework and the program structure was a fairly straightforward recursive function.

However, the *iTask* WDL has not been designed and implemented as a general GUI programming language. In this paper we show what is needed to extend it to make it suited for this purpose. The contributions of this paper are:

- We extend the *iTask* system and WDL with a number of missing features to support office-like GUI applications. This concerns infrastructure for menus, model-view-controller abstraction, support for multiple windows and dialogs, rich GUI elements and MDI infrastructure.
- The extensions are orthogonal to the basic *iTask* paradigm. 'Standard' workflow applications are not modified due to the extensions. Vice versa, interactive GUI applications can use the workflow facilities.
- We show how to create a multi-document editing application and also create a first prototype of an entire IDE for the Clean programming language. An additional advantage of using *iTask* is that this IDE is platform independent.

This high level of abstraction comes with the cost that the programmer cannot arbitrarily influence the layout of the user interface. There will also be no general canvas or the possibility to realise highly interactive applications like games. The paradigm is restricted to applications based on entering data using standard controls, like a multi-document text editor or an integrated development environment.

First we shortly summarise our extensions to the *iTask* system and our first experiences with realising complex applications in Section 2. In Section 3 we compare our approach to existing solutions. We draw conclusions in Section 4.

2 Results

One important extension added to the language are menus. They are defined in a similar way as in Clean Event I/O [3]. They are neatly integrated into the existing language since they generate actions the same way buttons do. The structure is defined once for a process, the actual menu is dynamically generated based on the context. Also whether items are enabled can declaratively be defined using predicates.

A very powerful extension for solving the classical *model-view problem* [15] is the concept of *views* on *shared data*. Here *shared variables* which behave like global data accessible by a reference are used. A view on such a variable is actually a realisation of lenses [5]. The programmer only has to define one function for converting the value from the model to the editor domain and another function for changing the model value based on a changed editor value. Actually this concept is the only way parallel tasks can interact, which forces the programmer to realise applications by defining an abstract state which is modified by tasks. This abstract state only models data but **not** the application's control flow. The tasks working on it therefore have a high level of independence.

Multiple windows and dialogs can be expressed by *grouped tasks* running in parallel. One can influence the behaviour (fixed, floating window or modal dialog) of each task. Also new tasks can dynamically be added without interfering with existing ones based on the result of other tasks or by menu commands.

Those concepts are powerful enough to make it possible to define a high-level combinator for realising multiple-document interface applications. Handling the entire application state and providing an encapsulated state for each document is done automatically. It can be used for implementing for instance a multi-file text editor. Also an application like an IDE can be realised, by using specialised types for providing source code input with syntax highlighting and using special tasks for accessing operating system functionality on the server, like writing to the file system and calling processes.

3 Related Work

In contrast to existing functional libraries for generating user interfaces the *iTask* paradigm does not deal with composing widgets to build a user interface directly, but works on a higher level.

Object I/O [1, 2] and *wxHaskell* [16] are examples with a more imperative taste in the sense that the user explicitly has to create user interface elements and update them if the state of the application changes. *Object I/O* has a global application state and also local states for realising encapsulation. For communication between processes sophisticated message passing mechanisms are used. In *wxHaskell* there is the concept of *mutable values* similar to *iTask*'s shared variables. One can wait for such a variable to change and can update widgets accordingly. In *iTasks* sharing data among a subset of tasks inside a process, the entire process or between multiple processes can be done using the same concept of shared data and views.

Haggis [11] also lets the user create widgets explicitly but gives the user a more compositional view on the user interface. Each component is treated as virtual I/O device. Components are repeatedly combined together to build up the entire application. Also a separation between the user interface and the application, which means between the representation and the actual value or interaction with the user, is made. This ensures a higher level of abstraction for the implementation of the program logic. No callback functions are used to handle events but one can wait for a message to be generated by a component. One can for example wait for a button to be pressed or a variable to change. Concurrency is used to make it possible to compose parts of the user interface waiting for messages at the same time.

A more functional approach of defining user interfaces is *Fudgets* [6, 7]. Here *fudgets*, which are *stream processors* and pass messages, are hierarchically combined to build up the application. There are no mutable variables. Sharing data has to be realised by routing messages between components.

Fruit [9, 10] emphasises being built on a formal model even more. The main building blocks here are *signals* which are continuous time-varying values and

signals transformers using pure functions for mapping signals to other signals. A GUI application is modelled as a signal transformer from a signal including all user inputs to a signal representing a picture. The cost of the formal model used by *Fruit* is that it is very cumbersome to define I/O other than turning the user input into a picture. All I/O operations explicitly have to be added to the input and output signal.

An example where generic programming techniques are used for generating forms is the *iData toolkit* [19]. It supports the creation of interactive web applications consisting of interconnected forms. A mechanism for providing views similar to the approach discussed in this paper is used. It has been shown that a complex application like a *Conference Management System* which also uses destructively updated shared data can be realised using this approach [18]. However the *iData toolkit* is not based on a workflow semantics. In contrast to *iTasks* there the system automatically keeps track of the control flow, with *iData* the programmer has to keep track of the application state.

Two more recent approaches that are also based on functional languages are *Links* [8] and *Hop* [22]. Both languages aim to deal with web programming within a single framework, just as the iData and iTask approach do. iTask has similar capabilities as Links and Hop in terms of client-side processing and thread-creation due to the use of client-side interpreter technology [12]. Furthermore, iTask provides a higher degree of automation due to the intensive use of generic programming techniques.

4 Conclusions

The *iTask* system has been extended to deal with essential features needed for implementing modern user interfaces. It is possible to structure commands using menus, to organise work in different windows and to modify a shared data model by different parallel tasks. Also special types can be added to allow for abstract representations of more complex user interface components. Further it is possible to build complex applications using the extended workflow language, as long as they are mainly based on standard controls filled in by the user and are not too interactive. The extended system is still based on the semantics of workflows and stays abstract and declarative.

The proposed paradigm for implementing graphical user interfaces has a higher level of abstraction than existing solutions with the price that it gives the programmer less control over how the user interface looks like. We think that the paradigm is highly suited for using rapid prototyping techniques and has a low learning curve, because it is based only on few core concepts. Still it is very powerful because all the power of functional programming can be used. Being embedded in a workflow system provides features like dealing with multiple users for free. Finally the program logic is based on a language for which abstract formal semantics can be defined [14], which helps reasoning about applications.

References

1. Peter Achten and Marinus J. Plasmeijer. Interactive functional objects in Clean. In *Implementation of Functional Languages*, pages 304–321, 1997.
2. Peter Achten and Martin Wierich. A tutorial to the Clean Object I/O Library - version 1.2. ftp://ftp.cs.ru.nl/pub/Clean/supported/ObjectIO.1.2/doc/tutorial.pdf, February 2000.
3. P.M. Achten and M.J. Plasmeijer. The Ins and Outs of Concurrent CLEAN I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.
4. Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005.
5. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.
6. M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.
7. Magnus Carlsson and Thomas Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden, March 1998.
8. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th'06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, November 2006. Springer-Verlag.
9. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.
10. Antony Alexander Courtney. *Modeling user interfaces in a functional language*. PhD thesis, Yale University, New Haven, CT, USA, 2004. Director-Hudak, Paul.
11. Sigbjörn Finne and Simon Peyton Jones. Composing the user interface with Haggis. In *Advanced Functional Programming: Second Interational School, LNCS #1129*, pages 26–30. Springer-Verlag, 1996.
12. Jan Martin Jansen. *Functional Web Applications – Implementation and Use of Client Side Interpreters*. PhD thesis, 8, July 2010. ISBN 978-90-9025436-4.
13. Pieter Koopman, Peter Achten, and Rinus Plasmeijer. Validating specifications for model-based testing. In Hamid Arabnia and Hassan Reza, editors, *Proceedings of the'08*, pages 231–237, Las Vegas, NV, USA, 14-17, July 2008. CSREA Press.
14. Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In *Scholz, S.-B. (ed.), IFL'08 : Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, pages 53–64. Hertfordshire, UK : University of Hertfordshire, September 2008.
15. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August-September 1988.
16. Daan Leijen. wxHaskell: a portable and concise gui library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM.
17. Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for end-users. In *Proceedings 21st Symposium on Implementation and Application of Functional Languages*, September 2009.

18. Rinus Plasmeijer and Peter Achten. A conference management system based on the iData toolkit. In *Implementation and application of functional languages : 18th international symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006 ; revised selected papers*, pages 108–125. Springer Verlag, 2006.
19. Rinus Plasmeijer and Peter Achten. iData for the world wide web - programming interconnected web forms. In *In Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), volume 3945 of LNCS*, pages 24–26. Springer Verlag, 2006.
20. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An Introduction to iTasks: Defining Interactive Work Flows for the Web. In *Central European Functional Programming School, Revised Selected Lectures, CEFP 2007*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, June 23-30 2007. Springer.
21. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th'07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.
22. Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the 11th'06*, pages 975–985, Portland, Oregon, USA, 22-26, October 2006.

Multiple-Occurrence I/O

Gergely Patai

Budapest University of Technology and Economics, Budapest, Hungary
patai@emt.bme.hu

Abstract. Event-driven systems have to deal with several different kinds of input and output simultaneously, and are generally programmed in an imperative style. However, imperative event-driven descriptions are difficult to compose in general. Due to the increased need for event-driven programming, better abstractions are sorely needed by the programming community. This paper presents an approach that is fully compositional and keeps side effects explicit where they are needed.

1 Introduction

Event-driven programming is becoming more and more important in all application areas. It is commonplace to cite examples like the web, distributed systems, cloud computing, heterogenous sensor networks and the like, but even single-host software systems have increasing need for parallelism. As a complex example for a heterogenous system, we might think of popular sandbox-style AAA games, which have to deal with several different kinds of interaction, for instance:

- to be able to provide a vast play area, terrain and models with different levels of detail (LOD) must be constantly streamed from the disk, cached, and moved between the CPU and the GPU;
- sound playback can involve both streaming (which might use a callback in the background to request the subsequent sections) and immediate triggering (for sound effects);
- a multiplayer game has to communicate over the network and synchronise world states among the players;
- at some points, the user must be provided a traditional GUI with widgets.

In practice, a complex application has to work with blocking I/O, interrupts, callbacks, and possibly even explicit message queues at the same time.

Since the applications in question are fundamentally parallel in nature, it is all the more important to provide referentially transparent building blocks. This paper presents a system where effectful event sources and processors are both first class citizens and entirely composable. We will walk through

- a simplified model for event sources and processors (Section 2);
- an implementation of the constructs in Haskell (Section 3);
- a toy example application that involves dealing with a GUI and networking (Section 4);

– and a review of related work (Section 5).

The codename of the library implementing the ideas discussed in the paper is Moio, short for 'multiple-occurrence I/O', and the code – including the model and the example application – is available for download[1].

2 Modelling Event Structures

An event-driven system can be thought of as a cloud of actors floating around in some suitable environment and communicating with asynchronous messages, which we will uniformly refer to as events from now on.

An actor is an entity that consumes a stream of events and produces another one as its output, optionally performing side effects that we shall ignore during the modelling phase, save for the temporal dimension. There are two special classes of actors that might be immediately interesting: producers and consumers. By our definition, these are actors with either the input or the output end unused.

Intuition tells us that actors are arrows in some category. We can choose the objects of this category to be either producers or consumers, and conclude that actors are in essence transformers of producers/consumers depending on our choice.

There is a fundamental difference between the two classes: if we look at them as functors, it turns out that producers are covariant, while consumers are contravariant. This suggests that producers are a more practical choice, since we cannot define the concept of natural transformation between a covariant functor (in our case, the identity) and a contravariant one, hence there is a lot less room to form interesting compositions. Also, producers provide the initial triggers for all the events in the system; without them, nothing could happen. Therefore, we shall ignore consumers from now on, and only concentrate on producers and actors in general.

2.1 Producers

A producer is an entity that emits a stream of events while optionally performing other side effects. Since side effects can also be considered an exchange of messages, we can simplify our model by identifying producers with their output.

One of the most basic forms of composition is interleaving. If we allow simultaneous occurrences on the conceptual level, producers form a monoid with interleaving as the binary operation and the empty producer as the identity.

We define events as values with a timestamp, hence an event stream is essentially a multiset of time-value pairs. We know that IO actions form a monad, and that they are the special case of producers: those that output only a single value. Therefore, it is a valid question to ask whether producers form a monad in general.

[1] `http://github.com/cobbpg/moio`

At first approximation, we can try treating producers as lists:

$$\textbf{type } \mathit{Producer}\ \alpha = [(\mathit{Time}, \alpha)]$$

The *Time* type needs to be totally ordered with a least element, and it should also support addition. In practice, we could use any numerical type and limit ourselves to positive numbers. We shall keep the elements in the order of increasing times in order to be able to work with infinite streams.

A natural choice for the semantics of *bind* is the following: $p \gg\!= k$ is a producer obtained by interleaving all the producers we get by mapping k over p and throwing away the outputs before the timestamp of the respective value passed to k. The pruning step is needed due to causality: we start observing $k\ x$ at the time of x, therefore we cannot see its earlier outputs.

Unfortunately, this model fails to be a monad instance, since there is no way to give a valid definition for *return*. If we consider the definition $\mathit{fmap}\ f\ e = e \gg\!= \mathit{return} \circ f$, it is clear that we have no way to preserve the timestamps, thereby violating the identity law.

We can fix our model by adding information about the time of creation (or start of observation) of a producer:

$$\textbf{type } \mathit{Producer}\ \alpha = \mathit{Time} \to [(\mathit{Time}, \alpha)]$$

We keep the above interpretation of *bind*, and define *return* x to be a producer that emits x as soon as it is synthesised. Assuming the existence of the *merge* function that properly interleaves two lists of timestamped values, we can easily define our fundamental compositions:

```
instance Monoid (Producer α) where
  mempty = const []
  e1 `mappend` e2 = λt → e1 t `merge` e2 t
instance Monad Producer where
  return x = λt → [(t, x)]
  e >>= k = λt → mconcat ∘ map spawn ∘ prune t $ e t
    where spawn (t, x) = prune t (k x t)
          prune t = dropWhile ((<t) ∘ fst)
```

The monad and monoid operations allow us to build complex producers from simple ones. However, they provide no means to connect two values from the same stream, since binding in this monad creates an independent branch for each occurrence. If we want to observe a whole stream, we need to define general actors.

2.2 Actors

Actors are stream processors capable of maintaining state between incoming events. Before defining their model, we need to think about the set of operations necessary to construct and compose them.

First of all, since actors have both an input and an output end, it is natural to define their sequential composition. When two actors are composed, we need to put a queue between them in order to avoid blocking on the writing side, which would defeat the purpose of event-driven programming. Intuitively, the identity of composition is an actor that forwards all its input immediately.

As for constructing actors, there are four basic building blocks to define:

$$\begin{aligned}
&\mathit{done} :: \mathit{Actor}\ \alpha\ \beta \\
&\mathit{put} :: \beta \to \mathit{Actor}\ \alpha\ \beta \to \mathit{Actor}\ \alpha\ \beta \\
&\mathit{get} :: (\alpha \to \mathit{Actor}\ \alpha\ \beta) \to \mathit{Actor}\ \alpha\ \beta \\
&\mathit{delay} :: \mathit{Time} \to \mathit{Actor}\ \alpha\ \beta \to \mathit{Actor}\ \alpha\ \beta
\end{aligned}$$

The empty actor is called *done*; $\mathit{put}\ x\ a$ is an actor that outputs x and continues as a; $\mathit{get}\ f$ is an actor that waits for an input, applies f to it and continues as the result; finally, $\mathit{delay}\ t\ a$ is our placeholder for computations in general, which waits for t time and continues as a.

Also, we might want to define a conversion function that observes the fact that every producer is an actor, which happens to ignore its input (from now on, we will use the letter χ to denote the type of an unused value):

$$\mathit{pToA} :: \mathit{Producer}\ \alpha \to \mathit{Actor}\ \chi\ \alpha$$

On the conceptual level, actors are simply functions mapping producers to producers:

$$\textbf{type}\ \mathit{Actor}\ \alpha\ \beta = \mathit{Producer}\ \alpha \to \mathit{Producer}\ \beta$$

Unfortunately, this simple model fails for the same reason as the first attempt at modelling producers: we cannot define *put*, since we are missing the necessary time information. We can try fixing it using the same trick, by supplying some kind of start time or current time:

$$\textbf{type}\ \mathit{Actor}\ \alpha\ \beta = \mathit{Time} \to \mathit{Producer}\ \alpha \to \mathit{Producer}\ \beta$$

This is slightly better, but we quickly find that we cannot deal with input events arriving during a delay. It would be possible to simply forget them, but that is not the semantics we are looking for. To solve the problem, we need to attach an input queue (represented by a list for simplicity) to the actor itself:

$$\textbf{type}\ \mathit{Actor}\ \alpha\ \beta = \mathit{Time} \to [\alpha] \to \mathit{Producer}\ \alpha \to \mathit{Producer}\ \beta$$

The queue contains no timestamps, just the values received that are yet to be processed by *get*. Now we can define all the four constructors, but we lost the ability to compose actors sequentially in a sane way. The problem is that the second member of the composition must be passed an empty queue for the lack of a better option. However, this means that we cannot define an identity actor, because composition itself has a side effect of losing queued elements. In order to cure this issue, we need to extend the model once more:

$$\textbf{type}\ \mathit{Actor}\ \alpha\ \beta = \mathit{Time} \to [\alpha] \to \mathit{Producer}\ \alpha \to ([\beta], \mathit{Producer}\ \beta)$$

The extension can be thought of as supplying an extra list of values that must have been output in the past. This allows us to thread input queues through series of actors and recover the laws of sequential composition.

At last, we can complete our category and use actors to transform producers:

$$(\leftarrowtail) :: \mathit{Actor}\ \alpha\ \beta \to \mathit{Producer}\ \alpha \to \mathit{Producer}\ \beta$$
$$a \leftarrowtail p = \lambda t \to \mathit{snd}\ (a\ t\ [\,]\ p)$$

The introduction of queues gives us an additional operation that might be useful in practice: a way to flush the contents of the queue, e.g. to be able to cope with high-frequency input.

$$\mathit{flush} :: \mathit{Actor}\ \alpha\ \beta \to \mathit{Actor}\ \alpha\ \beta$$

The model-level implementation of the operations carries no further enlightenment beyond the discovery of all the above issues, therefore it is left out of this paper and only distributed along with the library.

3 Implementing Composable Events

In a real program we have to work within an IO environment. Actors must execute in parallel, therefore it is straightforward to represent them with a thread that keeps references to its input and output channel. In contrast, producers only need to know about their consumer to be able to function properly.

A consumer needs to be able to receive values and also notification about the end of their input stream. Therefore, we can define them as callbacks accepting an optional value, where *Nothing* is used as a stop signal:

$$\textbf{type}\ \mathit{Sink}\ \alpha = \mathit{Maybe}\ \alpha \to \mathit{IO}\ ()$$

3.1 Producers

As we said, a producer is a computation that requires a callback. This makes it basically a continuation monad with an IO result type:

$$\textbf{newtype}\ \mathit{Producer}\ \alpha = P\ \{\mathit{unP} :: \mathit{Sink}\ \alpha \to \mathit{IO}\ ()\}$$

We can basically reuse the implementation of the continuation monad. The only thing we need to pay attention to is the treatment of stop signals. The single-shot producers constructed by *return* must immediately follow their output with a stop signal, and *bind* must propagate them to the sink. Unlike IO, we can also define a notion of failure: the empty producer.

$$\textbf{instance}\ \mathit{Monad}\ \mathit{Producer}\ \textbf{where}$$
$$\quad \mathit{return}\ x = P\ \$\ \lambda s \to s\ (\mathit{Just}\ x) \gg s\ \mathit{Nothing}$$

```
  P p >>= k = P $ λs → p (maybe (s Nothing) (($s) ∘ unP ∘ k))
  fail _ = P ($Nothing)
```

It might be surprising that interleaving is not mentioned in any way in the monad instance, unlike our pure model. The reason is that threading makes it implicit. On the other hand, the monoid instance does have to introduce concurrency explicitly:

```
instance Monoid (Producer α) where
  mempty = P ($Nothing)
  P p1 `mappend` P p2 = P $ λs → do
    v ← newEmptyMVar
    forkIO $ p1 (putMVar v)
    forkIO $ p2 (putMVar v)
    flip fix 2 $ λloop n → when (n > 0) $ do
      mx ← takeMVar v
      when (isJust mx ∨ n ≡ 1) (s mx)
      loop $! if isNothing mx then n − 1 else n
```

When we merge two streams, the resulting stream ends only when both constituents end, so it has to absorb all the stop signals but the last one. The *mconcat* operation can be optimised to create only one *MVar*.

Just like the *main* program, the top-level producer does not create any interesting output, so embedding it into the IO environment means simply passing it a dummy consumer:

```
nop :: IO ()
nop = return ()

embed :: Producer χ → IO ()
embed (P p) = p (const nop)
```

But how do we get side effects in a producer? The simplest option is to lift IO computations as single shot producers:

```
ioP :: IO α → Producer α
ioP act = P $ λs → act >>= λx → s (Just x) ≫ s Nothing
```

Of course the interesting case is when we have an IO computation that can potentially produce several results in sequence. In order to embed such a computation, first we need a primitive to define 'pure' producers that have no side effects of their own, but can be made to emit a value through a sink. The sink returned by *mkSink* self-destructs after receiving a stop signal:

```
mkSink :: IO (Producer α, Sink α)
mkSink = do
  v ← newEmptyMVar
  a ← newIORef (putMVar v)
```

```
return (P $ λs → fix $ λloop → do
          mx ← takeMVar v
          forkIO $ s mx
          if isJust mx then loop else writeIORef a (const nop)
        , λx → readIORef a >>= ($x)
        )
```

This operation can be used to define the desired embedding:

```
mkP :: (Sink α → IO ()) → Producer α
mkP f = P $ λs → do
  (P p, s′) ← mkSink
  forkIO $ f s′
  p s
```

Since now our producers are effectful, we have to be careful about sharing them. Just as a value of type $IO\ \alpha$ is a recipe for a computation, not the actual effectful operation, $Producer\ \alpha$ is a recipe to create a stream of events and produce side effects. However, while we can simply pass around the result of an IO computation, we cannot do the same with event streams, which do not exist as first-class values.

In order to get the same effect, we can make a copy of a producer that has the same output as the original, but does not perform any side effect on its own. The *subscribe* function constructs a single-shot producer that starts up the producer passed to it immediately and emits a shareable producer that has the same output as the original:

```
subscribe :: Producer α → Producer (Producer α)
subscribe (P p) = mkP $ λs → do
  c ← newChan
  forkIO $ p (writeChan c)
  let p′ = mkP $ λs → do
    c′ ← dupChan c
    fix $ λloop → do
      mx ← readChan c′
      s mx
      when (isJust mx) loop
  s (Just p′)
  s Nothing
```

There is another important pattern that the combinator base can technically describe, but only with less possibility of garbage collection: shutting down a producer when an event fires.

A naive implementation of such a combinator might be the following:

```
until :: Producer χ → Producer α → Producer α
until c p = (Just <$> p) `mappend` (Nothing <$ c) ↣ cutoff
  where cutoff = get $ maybe done (flip put cutoff)
```

The producer constructed by *until s p* emits whatever *p* emits until the first occurrence on *s*. Unfortunately, it still has to cling to both *s* and *p* afterwards as long as they are alive. To prevent this, the library offers *until* as a primitive:

```
until :: Producer χ → Producer α → Producer α
until (P c) (P p) = P $ λs → do
  v ← newEmptyMVar
  tid ← forkIO $ p s
  c $ λ_ → do
    killThread tid
    s Nothing
    killThread =≪ myThreadId
```

As soon as there is stop signal, both producers are killed. This should cause no problem, because if these producers are shared with *subscribe*, we only kill the particular specimen that forwarded the output of the original producer.

3.2 Actors

While actors are more complicated in the model than producer, the opposite is true in the implementation. As said already, an actor is a computation that knows about its input and its consumer. The input end is a FIFO channel that can accumulate incoming values while the actor is preoccupied:

```
newtype Actor α β = A {unA :: Chan (Maybe α) → Sink β → IO ()}
```

The three basic constructors are straightforward to implement:

```
done :: Actor α β
done = A $ λ_ o → o Nothing

put :: β → Actor α β → Actor α β
put x (A a) = A $ λi o → o (Just x) ≫ a i o

get :: (α → Actor α β) → Actor α β
get f = A $ λi o → do
  mx ← readChan i
  case mx of
    Nothing → o Nothing
    Just x → unA (f x) i o
```

However, since *delay* was only used as a placeholder for computations, we replace it with an actor that performs arbitrary IO and continues as the result of the computation:

```
ioA :: IO (Actor α β) → Actor α β
ioA act = A $ λi o → act ≫= λ(A a) → a i o
```

We can also construct effectful producers that are equipped with an input channel for a stop signal. The actor created with *mkA* takes a function that must

construct two actions from the sink: the producer itself and the clean-up action to be executed when the termination request arrives:

$$mkA :: (Sink\ \alpha \to IO\ (IO\ (), IO\ ())) \to Actor\ \chi\ \alpha$$

The only thing left is the composition of actors and producers. This is where we can fire up the thread for the actor:

```
(↢) :: Actor α β → Producer α → Producer β
A a ↢ P p = P $ λs → do
  c ← newChan
  forkIO $ a c s
  p (writeChan c)
```

It is also possible to define a category (and even an arrow) instance for actors, but in practice it seems to be more convenient to stay in the producer monad and use $\leftarrowtail$ to introduce stateful stream transformers. There are two reasons in favour of sticking to the monad interface: it is more flexible, and it does not force us to pair up actors that have nothing to do with each other.

3.3 Monadic Actors

We can also construct actors using an alternative, monadic interface, which allows for an imperative style of programming, which might be more convenient in certain cases. The data structure behind these constructions is a cross between those of actors and producers:

$$\begin{aligned}\textbf{newtype}\ ActorM\ \iota\ o\ \alpha = AM\ \{\, unAM\ ::\ & Chan\ (Maybe\ \iota) \to Sink\ o \\ & \to (\alpha \to IO\ ()) \to IO\ ()\,\}\end{aligned}$$

The type parameters are the following: ι is the input of the actor, o is the output, and α is the value produced by the monad. The structure above can support the following interface:

```
halt :: ActorM ι o ()              ‡ done
emit :: o → ActorM ι o ()          ‡ put
fetch :: ActorM ι o ι              ‡ get
ioAM :: IO α → ActorM ι o α        ‡ ioA
```

The implementation is almost the same as for the non-monadic interface. We can also define conversion functions that let us freely combine the two styles:

```
amToA :: ActorM α β χ → Actor α β
aToAM :: Actor α β → ActorM α β χ
```

It is up to the programmer which style they prefer. Generally, smaller actors are easier to construct with the non-monadic interface. When it comes to more complicated control structures, the monadic interface might be more fit for the purpose.

3.4 Dynamic Composition

We have already seen that actors subsume producers. However, albeit less directly, the situation can be reversed. The library defines a dynamic composition operator (not a primitive):

```
(↞) :: Producer (Actor α β) → Producer α → Producer β
pf ↞ px = do
  pf′ ← subscribe pf
  px′ ← subscribe px
  f ← pf
  until pf′ (px′ ↣ f)
```

The expression $pf \twoheadleftarrow px$ describes a producer obtained by feeding px into an actor that changes its behaviour whenever pf produces a value by replacing itself with the latest occurrence on pf. Obviously, this is a generalisation of $\hookleftarrow$, since we can pass an producer constructed with *return* as the first argument to get back the simple composition.

Dynamic composition is particularly interesting, because it puts the objects and the arrows of the actor category in the same bin.

4 Example: a Bulletin Board Server with a GUI

To show the approach in action, let us implement a simple bulletin board server that allows several clients to post messages simultaneously, and displays these messages on a common interface. The interface can be seen on Figure 1.

Despite its simplicity, this is an interesting exercise, because it involves two different types of interaction with the world: callback based GUI and blocking IO for the network. First we will introduce some small building blocks, then walk through the main program logic in one go.

In order to convert the clicks on a GTK button into a producer, we can use the *mkSink* primitive:

```
mkButtonP :: Button → IO (Producer ())
mkButtonP button = do
  (event, sink) ← mkSink
  button `on` buttonActivated $ sink (Just ())
  return event
```

In our example, the button either starts or stops the server. It is easier to deal with this task if we convert button clicks into a stream of alternating values. This can be generalised into a cyclic behaviour, so we can define an actor that cycles through a given list element by element whenever it receives an event.

```
cycleA :: [a] → Actor x a
cycleA vals = rep (cycle vals)
  where rep (x : xs) = get ∘ const ∘ put x $ rep xs
```

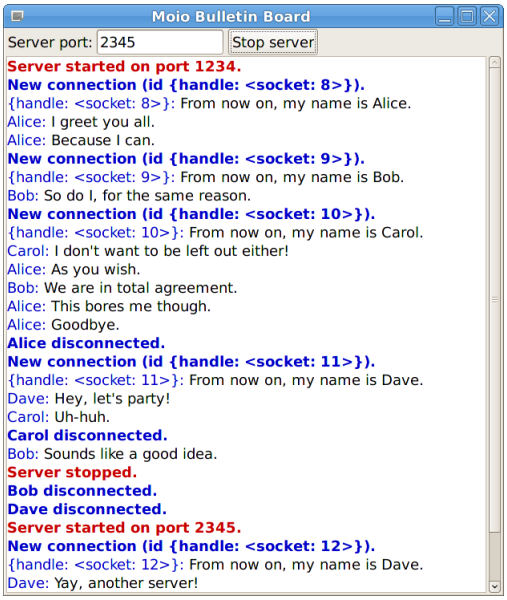


Fig. 1. The bulletin board interface

When we start up a server, we essentially create a source of client connections. This source can be turned into an actor that keeps producing these connections until it receives a stop request. We can use the *mkA* primitive to define such actors.

```
listenA :: PortNumber → Actor x Handle
listenA port = mkA $ λsink → do
  sock ← listenOn (PortNumber port)
  let go = forever $ do
    (hdl, _, _) ← accept sock
    hSetBuffering hdl LineBuffering
    sink (Just hdl)
    stop = sClose sock
  return (go, stop)
```

If we have a client connection, we have a source of client inputs. Using the *mkP* primitive we can turn this into a producer that keeps running while the handle is alive, and emits an extra line before stopping.

```
fetchLines :: Handle → Producer String
fetchLines hdl = mkP $ λsink → fix $ λloop → do
  mline ← catch (Just <$> hGetLine hdl) (\_ → return Nothing)
  case mline of
    Just line → sink mline ≫ loop
    Nothing → sink (Just "#END") ≫ sink Nothing
```

Our bulletin board allows the clients to change their screen names with a special command. This means that the names have to be kept track of for each connection. We can achieve this by transforming the stream of inputs with a simple actor.

```
addName :: String → Actor String (String, String)
addName name = get $ λline →
  case words line of
    "#NAME" : name′: _ →
      put (name, printf "From now on, my name is %s." name′)
          (addName (escapeMarkup name′))
    _ → put (name, line) (addName name)
```

Given the building blocks above and skipping some GUI setup, the logic of the main program can be described in a big top-level producer. The full source of the producer is given on Figure 2. This definition is within the do block describing the *main* function, because it refers to some widgets extracted from the Glade description during the setup phase.

The name *toggleServerP* refers to the producer connected to the button on the top, created with *mkButtonP*. Since we will need its output in two places, we use *subscribe* to derive a copy of the event stream devoid of side effects.

We immediately connect to *toggle* and check whether the incoming value is *True* or *False*. In the former case, we change the label of the button and fire up the server at the port specified in the top text entry field using *listenA*. We connect the *listenA* actor to *toggle*, so it also receives the next click of the button that will cause it to shut down. Conversely, if *isStart* is false, i.e. we are stopping the server, we just reset the button label and do not emit anything, since the whole conditional construct describes a producer of client connection handles.

When a new connection is made, we get a handle. First we display the identifier of the handle, and use it as the initial name for the client in question by passing it to *addName*. Afterwards, we start observing the input coming from this client, and keep track of its name. Whenever we get a line of input, we display it. Due to the way the producer monad works, the inputs of the clients are properly interleaved.

5 Related Work

The classical incarnation of functional stream processors can be found in the Fudgets framework [1], including both the simple and the monadic interfaces as described in Section 3. The actor model presented in this paper differs from the stream processor concept of Fudgets in a few ways. On the conceptual side, the actor model deals with temporal behaviour explicitly, using time information to define the semantics of some constructs. Also, Fudgets implements the continuation style interface directly with ADTs, and requires serially composed stream processors to synchronise at every message exchange by blocking the producer

```
let mainProducer = do
  ‡ Create a shared stream for the button clicks.
  toggle ← subscribe $ cycleA [True, False] ↢ toggleServerP
  ‡ Start/stop the server and produce the client connection handles.
  isStart ← toggle
  hdl ← if isStart
    then do
      port ← ioP $ do
        let readPort = fromMaybe 1234 ∘ fmap fst ∘ listToMaybe ∘ reads
        p ← readPort <$> entryGetText portNumber
        entrySetText portNumber $ show p
        buttonSetLabel toggleServer "Stop server"
        appendText red True $ printf "Server started on port %d.\n" (p :: Int)
        return (fromIntegral p)
      listenA port ↢ toggle
    else do
      ioP $ do
        buttonSetLabel toggleServer "Start server"
        appendText red True "Server stopped.\n"
      mempty
  ‡ Display the arrival of the new client.
  let idstr = escapeMarkup (show hdl)
  ioP $ appendText blue True $ printf "New connection (id %s).\n" idstr
  ‡ Display the input coming from all the clients
  (name, line) ← addName idstr ↢ fetchLines hdl
  ioP $ if line ≢ "#END"
    then do
      appendText blue False $ printf "%s: " name
      appendText black False $ printf "%s\n" (escapeMarkup line)
    else do
      appendText blue True $ printf "%s disconnected.\n" name
```

Fig. 2. The top-level logic of the bulletin board application

end until the consumer reaches the *get* state. The fundamental difference on the implementation side is that Moio actors can perform arbitrary IO.

Enno Scholz put forward the idea of imperative streams called the St monad [8, 9], which is also a flavour of multiple-occurrence IO. The semantics of *bind* for Scholz's streams requires the continuation to be shut down and replaced by the next one whenever a new value arrives, i.e. the continuations arising from input events run sequentially and exclusively, not in parallel. In order to define stream processors, the programmer has to use explicit mutable variables within the St monad. Also, due to the way it is implemented (it is a function that takes the input and returns the resulting action; $St\ \alpha \sim IO\ (Devices \rightarrow IO\ (Maybe\ \alpha, Bool))$), even the absence of input events has to be propagated through the program.

Out of the functional GUI frameworks, FranTk [7] is also relevant, as it treats producers and consumers as first-class abstractions. It makes note of their duality and presents a family of corresponding operations on them. Producers and consumers can be connected through wires, and the library also provides mutable variables that play nice with the framework. However, unlike Moio, FranTk assigns an identity to effectful entities, and makes the equivalent of *subscribe* implicit when composing producers/consumers with pure transformations (e.g. map or filter). This decision blurs the borderline between effectful and pure entities, and relies on *unsafePerformIO* to work as expected.

Reactive [3] is the latest products of Conal Elliott's experiments in FRP. The event abstraction of the library corresponds to our producers. However, the difference is that the model of Reactive events does not include an initial time. The monad instance has a similar interpretation, the difference being that instead of pruning the events arising from the continuation, the times are related through a monoid operation (addition for relative time and maximum for absolute time). Unfortunately, the Reactive event model contains a semantic bug: the monad laws do not hold. The extension of the model presented in Section 2 is one way to fix the problem.

The Rx framework [4] is a very similar approach to compositional events. It introduces the concept of *observables*, which is in essence identical to our producer concept: continuation monad with side effects. Erik Meijer points out that this structure is a categorical dual to iterators, and can be mechanically derived by flipping the arrows and writing out the resulting interfaces:

$$\textbf{type}\ \alpha \rightsquigarrow \beta = \alpha \rightsquigarrow IO\ \beta$$
$$\textbf{type}\ Iterator\ \alpha = () \rightsquigarrow (() \rightsquigarrow \alpha)$$
$$\textbf{type}\ Observable\ \alpha = (\alpha \rightsquigarrow ()) \rightsquigarrow ()$$

While Rx provides the monadic interface, it does not have stream processor combinators. Also, since it lives in a non-pure environment, one must be careful about unwanted duplication of side effects. To prevent them, the library provides the *Let* construct, which achieves the same effect as *subscribe*, but with a slightly different interface.

F# presents .NET events as first-class values [10]. This allows the definition of combinators that imitate familiar list functions and other useful patterns. In

a related publication [6], Petricek and Syme discuss the difficulties of garbage collection in an event-driven system, and point out the importance of inspecting connections both ways. In contrast, Moio has no way to let consumers signal to producers about their termination.

The Timber language [5] is a different approach to provide composable abstractions for event-driven programming. At its heart, various actors (synchronous and asynchronous event handlers, objects) are built on top of a monadic structure [2] that takes care of communication and timing. Compile-time scheduling is a major selling point, since Timber is aimed at programming real-time systems. However, the compositions are less explicit than in Moio, since they are absorbed by the global monadic structure.

References

1. Magnus Carlsson and Thomas Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1993, Copenhagen, Denmark, pages 321–330, ACM Press, 1993.
2. Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The Semantic Layers of Timber. In *First Asian Symposium on Programming Languages and Systems (APLAS)*, Beijing, China, 2003.
3. Conal Elliott. Push-pull functional reactive programming In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Edinburgh, Scotland, 2009, pages 25–36, ACM, New York, NY, USA, 2009.
4. Erik Meijer et al. Reactive Extensions for .NET (Rx). Microsoft DevLabs. `http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx`
5. Johan Nordlander, Mark P. Jones, Magnus Carlsson, and Jan Jonsson. Programming with Time-Constrained Reactions. Technical report, Luleå University of Technology, 2005.
6. Tomas Petricek and Don Syme. Collecting Hollywood's Garbage: Avoiding Space-Leaks in Composite Events. In *Proceedings of the 2010 international symposium on Memory management, ISMM 2010*, 2010, pages 53–62, Toronto, Ontario, Canada, ACM, New York, NY, USA, 2010.
7. Meurig Sage. FranTk - A declarative GUI language for Haskell In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000, pages 106–117, Montreal, Quebec, ACM, New York, NY, USA, 2000.
8. Enno Scholz. A Monad of Imperative Streams. In *Proceedings of the 1996 Glasgow Workshop on Functional Programming*.
9. Enno Scholz. Imperative streams—a monadic combinator library for synchronous programming. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 1998, pages 261–272, Baltimore, Maryland, United States, ACM, New York, NY, USA, 1998.
10. Don Syme. Simplicity and Compositionality in Asynchronous Programming through First Class Events. 2006–2010. `http://tinyurl.com/composingevents`

Gin: Graphical iTask Notation
– extended abstract –

Jeroen Henrix, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
jmhenrix@student.ru.nl, {rinus, p.achten}@cs.ru.nl

1 Introduction

In this paper we present Gin. Gin is both a graphical notation for the iTask system as well as a tool to construct iTask applications in an interactive and graphical way. The iTask system [3] is a workflow management system (WFMS) that is embedded in the pure and lazy functional programming language Clean. WFMSs are software applications that coordinate business processes. This coordination is based on a workflow model: a formal description in a workflow definition language (WDL) of the tasks that comprise a business process, their ordering and data dependencies.

Traditionally, WDLs are based on coloured Petri Nets [1]. This has the immediate advantage that they come with an intuitive graphical notation that can be used in the development process by workflow engineers and domain experts. The iTask WDL, on the other hand, is founded on a set of combinator functions, which is a common approach in the functional language community. Hence, to understand and appreciate an iTask workflow, one needs to be trained in functional programming. Using Gin, this is less of a requirement, because Gin offers a structured and graphical way of building iTask models, and confronts the user much less with functional programming activities. Hence, building a simple model should also be simple to the user, whereas for the development of a complex model one is likely to be better off using the full expressive power of Clean.

The contributions of our work are:

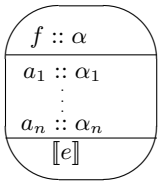
- We identify graphical workflow patterns that correspond with iTask combinators. We adopt graphical conventions from the standard workflow community if possible. We integrate combinator language features (scoping of variables, recursion, block structure) within Gin.
- We implement the Gin tool as a proof of concept. It is integrated in the iTask system and uses the Clean compiler for error and type checking. Errors are reported back to the user in terms of the Gin model.

In the remainder of this extended abstract we briefly discuss the Gin language (Section 2) and tool (Section 3). We end with conclusions (Section 4). *Related work will be discussed in the full paper.*

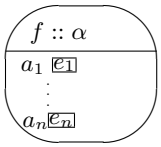
2 The Gin language

The Gin language is a mostly graphical WDL to which Clean text fragments can be added. It defines a graphical notation for a subset of the iTask WDL. iTask expressions of type *Task* α are represented in Gin by directed graphs. As conventional in graphical WDLs, nodes indicate tasks, and edges indicate control flow relations. Because the iTask combinators are block structured, unstructured control flow like arbitrary jumps cannot be expressed in iTask. Therefore, Gin only allows well-structured flows, consisting of atomic tasks and nested, non-overlapping control blocks. We informally introduce the Gin language by means of a map $[\![\cdot]\!]$ from iTask expressions to Gin diagrams. Gin supports the following constructs:

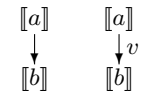
- *Task definition*: Functions $f :: \alpha_1 \to \alpha_2 \to \ldots \alpha_n \to$ *Task* α $(n \geq 0)$ defined as $f\ a_1\ a_2 \ldots a_n\ =\ e$ are depicted as shown on the right. The arguments a_i are simple variable names, patterns are not supported. The variables a_i are in scope of $[\![e]\!]$.



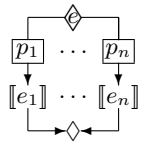
- *Task application*: functions $f :: \alpha_1 \to \alpha_2 \to \ldots \alpha_n \to$ *Task* α $(n \geq 0)$ defined as $f\ a_1\ a_2 \ldots a_n\ =\ e$ applied to arguments $e_1 \ldots e_n$ are depicted as nodes in the workflow graph, shown on the right. Arguments $e_1 \ldots e_n$ can be any Clean expression. Higher order tasks are supported, the higher-order argument is then represented as a directed graph.



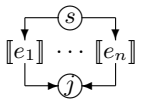
- *Sequential composition*: $[\![a \texttt{ >>| } b]\!]$ and $[\![a \texttt{ >>= } \lambda v \to b]\!]$ are represented as shown on the right. Variable v is in scope of $[\![b]\!]$.



- *Case distinction*: $[\![\textbf{case}\ e\ \textbf{of}\ p_1 \to e_1\,;\ \ldots\ ;\ p_n \to e_n\,;]\!]$ $(n \geq 1)$. Here, e and e_i can be any Clean expression, and p_i can be any pattern. Each pattern variable introduced in p_i is in scope of $[\![e_i]\!]$.



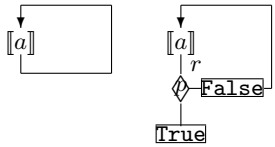
– *Parallel composition*: parallel composition of tasks is depicted by a *split connector* s, one or more branches $e_1 \ldots e_n$ and a *join connector* j, as shown on the right. Pairs of split and join connectors are mapped to iTask combinators. Different types of join connectors are used for different types of joins, see the table below.



Split	Join	iTask combinator	Result type
$\wedge$	$\vee_{first}$	`anyTask` $[e_1, \ldots, e_n]$	`Task` a
$\wedge$	$\vee_{left}$	e_1 `-||` e_2	`Task` a
$\wedge$	$\vee_{right}$	e_1 `||-` e_2	`Task` a
$\wedge$	$\wedge_{(,)}$	e_1 `-&&-` e_2	`Task` (a, b)
$\wedge$	$\wedge_{[]}$	`allTasks` $[e_1, \ldots, e_n]$	`Task` $[a]$

The full paper will explain these combinators.

– *Iteration*: ⟦`forever` a⟧ and ⟦a `<!` $(\lambda r \rightarrow p)$⟧ (repeat ... until) are represented as shown to the right. The variable r is in scope of p, but no further.



– *Multiple instances*: Many WDLs allow multiple instances of the same task to be started with different parameters. In iTask, this is accomplished by applying list combinators (such as `sequence` and `allTasks`) to lists of tasks created with different parameters. Gin defines tasks for these list combinators, and has a graphical representation for static lists expressions and list comprehensions.
Due to their length, these diagrams are omitted from this extended abstract, but will appear in the full paper.

Workflow models expressed in Gin are a hybrid form of graphical constructs and textual Clean expressions. Composition of iTask combinators is expressed graphically, while patterns and first-order task arguments are denoted as text. Ordinary Clean functions can be embedded in Gin models. These do not have a grapical representation.

3 The Gin tool

The Gin tool is a proof-of-concept implementation. In this section we discuss the following major parts of the tool: the front-end (Section 3.1), the implementation of the front-end (Section 3.2), the compilation of a constructed iTask workflow by Gin (Section 3.3), and the handling and reporting of errors (Section 3.4).

3.1 The front-end

The Gin front-end (Figure 1) is a web-based editor; it is implemented as a Java applet consisting mainly of a drawing canvas and a repository. A new workflow starts with a blank canvas. Nodes can be added to the workflow by dragging

them from the repository to the canvas. The editor supports common editing operations one would expect from a workflow editor like moving nodes, adding connectors etc.

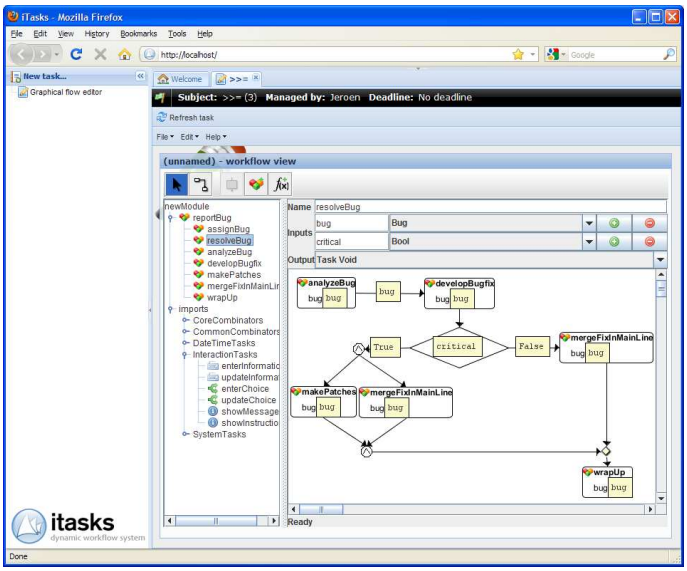


Fig. 1. Gin front-end showing a bug reporting workflow

3.2 Integration of Gin in iTask

Gin workflows graphs are stored as values of an algebraic data type named `GraphicalWorkflow`. These values contain all information to visually reconstruct their workflow diagrams, including layout.

In iTask, the function `updateInformation` can generate a default editor for any data type. This function creates an interactive element using a couple of generic functions: `gVisualize` for generating editor GUIs and `gUpdate` for updating edited values. With the use of specialization, we have created a specialized editor for `GraphicalWorkflow` values (as shown in Figure 1). Hence, editing Gin workflows boils down to editing values of type `GraphicalWorkflow`. The entire process around the manipulation of `GraphicalWorkflow` values can be regarded as a workflow itself - a meta-workflow. This meta-workflow is modeled in iTask; its main part consists of the task `updateInformation wf`, where `wf :: GraphicalWorkflow`.

3.3 Compiling Gin workflows to iTask workflows

The Gin tool compiles a `GrapicalWorkflow` value to an executable iTask in five steps. First, a block detection algorithm parses sequential blocks, parallel blocks

and structured loops in the graph and maps these to nodes in a block tree structure. Second, the block tree is transformed to an abstract syntax tree (AST) containing an iTask expression of type $f :: \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \alpha_n \rightarrow \mathit{Task}\ \alpha$. This transformation uses a set of bindings from graph node types to abstract syntax. Third, the AST is printed to concrete Clean source code, which is written to a file. The source file additionally contains a `Start` function which wraps the iTask expression in a **dynamic** [2] and writes it to a file. Fourth, the Gin tool calls the Clean compiler to compile the source file and runs the resulting executable, so the **dynamic** is written to disk. Finally, the **dynamic** file is loaded by the Gin tool, ready to be executed.

3.4 Error handling

During editing, the user gets immediate feedback about the incorrect parts of the model. This is implemented by means of a compiler feedback loop. After each editing operation, the iTask server starts the compilation process in the background. Compilation may fail for several reasons. Either parsing fails because the graph structure is not well formed, or the generated source code is erroneous, so the Clean compiler outputs error messages. The Gin tool keeps track of a map from the line numbers in the compiled source code back to graphical nodes. If errors are found, the incorrect graph nodes are highlighted. When the mouse cursor is moved over these nodes, a text box with the error message is shown.

4 Conclusions

The Gin language is a graphical notation for a subset of the iTask WDL. Gin makes iTask models more accessible for domain experts who may be unfamiliar with functional programming. Gin captures the iTask control flow and workflow decomposition in a grapical notation resemblant to conventional WDLs. However, task parameters and operations on data structures are still expressed as Clean expressions.

Our proof-of-concept implementation shows that the iTask system provides a suitable environment for embedding custom editors for complex data types like workflow graphs, including feedback for error handling.

In our experience, graphical front-ends like Gin make good use cases for accessing the compiler via an API. If an API had been available for the Clean compiler, we could pass generated ASTs directly to the compiler, instead of having to print the ASTs, write source files which are read and parsed again by the compiler. Besides, compiler errors can be passed in type-safe way.

References

1. W.M.P. van der Aalst. Chapter 10: Three good reasons for using a Petri-net-based Workflow Management System. volume 428 of *The Kluwer International Series in Engineering and Computer Science*, pages 161–182, Boston, Massachusetts, 1998. Kluwer Academic Publishers.

2. Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages, IFL '98, London, UK*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.
3. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.

Towards Dependently-Typed Attribute Grammars

Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra

Universiteit Utrecht,
The Netherlands

Abstract. Complex traversals over Abstract Syntax Trees (ASTs) can be programmed conveniently with Attribute Grammars (AGs). In order to specify partial correctness of such an AG, we investigate in this paper AGs with attributes that have a dependent type. We identify extensions needed to the AG formalism to effectively deal with dependently-typed attributes.

1 Introduction

The static semantics of context-free languages can be described with Attribute Grammars (AGs) [7]. AGs extend a context-free grammar with attributes associated with a nonterminal, and equations (rules) between these attributes associated with a production. An Abstract Syntax Tree (AST) represents a proof of the recognition of a sentence in the context-free language. Attributes are convenient: these represent individual aspects or properties of the AST, can be combined at each node of the AST to define more complex attributes, and it is easy to add new attributes to the grammar. From a practical point of view, via a preprocessor, an AG can be compiled to an efficient algorithm in some host language that, given an AST, computes values for these attributes. The translated code in the host language then serves as an implementation for a compiler.

In the last years, we applied AGs in several large projects, including the Utrecht Haskell Compiler [3], the Helium [5] compiler for learning Haskell, and the editor Proxima [13]. The use of AGs in these projects is invaluable, for two reasons: rules for attributes can be written in isolation and combined automatically, and trivial rules can be inferred, saving the programmer from writing boilerplate code.

Given a compiler such as the UHC, the question arises to what extend the compiler generates correct code (for some definition of correctness). In UHC, similar to any project that many people contribute to, we regularly encounter subtle bugs related to broken hidden invariants. Through the use of regression testing, we get a fair idea of how well the implementation is, however we never get sure about the number of erroneous cases that lurk in a dark corner.

The holy grail in compiler implementation is to formally verify that a compiler exhibits certain properties of interest. For example, that a type inferencer is a sound and complete implementation of a type system, and that the meaning of a well typed source program is preserved. In practice, this formal verification is (still) infeasible for large projects with limited contributors. Given the complexity of compilers involved, it is already a struggle to implement and maintain a compiler that gives output that seems correct, let alone to attempt to prove this.

When we implement our compilers in languages with advanced type systems, we can ensure that compilers have certain properties by construction. In mainstream strongly typed languages, it is possible to guarantee that (even for non-trivial inputs) the compiler produces output that conforms to a context-free grammar. It is enforceable that a compiler produces, for example, syntactically-correct HTML or C code. These are properties we get for free when we define our datastructures properly in, for example, the strongly-typed programming language Haskell.

Unfortunately, syntactic guarantees do not save us from hidden invariants. For that, we desire semantic guarantees. This is possible by using more complex types, but at a severe cost. It actually requires us to encode proofs by using the type system as a logic. In a dependently-typed language such as Agda, we can impose arbitrary complex constraints on values via types, but as a consequence, proving that the program is well typed becomes considerably harder.

To move from a compiler implementation in Haskell to Agda, for example, the least extra effort required—syntax and library issues aside—is to prove termination of the program (ensures that the program is a proof in a consistent logic). Compilers typically consist of primitive recursive functions, and those parts in a compiler that rely on iteration either compute fixpoints, or have iteration limits. Unification-based type inferencers may need to ensure that types in the substitution are finite, imposed via the 'occurs check', which is a relatively easy property. The extra effort is substantial, but theoretically speaking, not a creative challenge.

From there on, the programmer can make the types more specific, and prove stronger invariants. For a project the size of UHC, a verified compiler is still too much to ask for, as it makes implementation and maintenance more difficult and time consuming. Recent developments in dependently-typed programming, however, make such approaches more and more feasible. Several success stories are known, which is promising for the future. Since we implement our compilers with Attribute Grammars, we present in this paper how these can be combined with dependently-typed programming, using Agda in particular.

AGs can be compiled to (a sequence of) functions that take a record of inherited attributes, have cases for each type of AST node, recursively visit the children of such a node, and produce a record of synthesized attributes. Such functions can be written in Agda. There are thus no theoretic limitations that prevent AGs to be embedded in a dependently-typed programming language.

However, dependently-typed programming enforces structure to a program that is inconvenient when using plain Attribute Grammars. In short, to profit from programming with AGs, a fine granularity of attributes is beneficial, but may require partially defined attributes. In a dependently-typed setting, all attributes must have a total definition, and to accomplish that, a coarser granularity of attributes may be required. The coarser the granularity of attributes, the less benefits an AG has over a conventional fold. We demonstrate this with an example in Section 2, and we show that we can keep the fine granularity of attributes when we extend attribute grammars with context-dependent attribution.

In previous work, we implemented an AG preprocessor for Haskell, and some other languages (cite, cite). The language of this preprocessor is called RULER-FRONT. In this

paper, we introduce a variation called RULER-FRONT$_{\text{AGDA}}$, which has additional syntax to support the more complex types for attributes, and to deal with dependently-typed pattern matching in the left-hand sides of rules. In particular, dependent pattern matching against attributes may impose additional constraints on other attributes. For example, in the scope of a match against an equality proof, we get the additional information that two attributes are the same and Agda requires them to get a common name. The example in Section 2 again demonstrates.

As contributions, we extensively discuss an example in Section 2, then describe the AG-language RULER-FRONT in Section 3, and its translation to Agda in Section 4.

2 Example

In this section, we demonstrate a mini-compiler implemented as dependently-typed attribute grammar. We take as example scope checking of a simple language. The actual concrete syntax of the source language that the compiler takes and the target language that the compiler produces is irrelevant for this paper. The abstract syntax we represent with Agda data types given below. Scope checking of a simple language is only a minor task in a compiler. However, it still shows many aspects that a more realistic compiler has.

We assume that the reader has experience with a functional programming language such as Haskell, and has some knowledge about dependently-typed programming as well as Attribute Grammars.

The code of this compiler consists of blocks of Agda code, and blocks of RULER-FRONT. To easy the distinction between the two, we underline keywords of Agda, and format keywords of RULER-FRONT bold.

2.1 Dependently-typed program in Agda

The main distinctions between Agda and a conventional programming language such as Haskell are universes, dependent type signatures and dependent pattern matching. We briefly explain these distinctions through some example code needed later.

A universe contains types: the type of such a type must be a type in the successor of the universe. Furthermore, the syntax of types in a universe is the same as the syntax of values, which means that lambda abstraction and pattern matching is available for types. In this example, we only use the universe *Set* of types.

As an example, below is a block of Agda code. We introduce types to represent identifiers, environments, and scoping errors. We simply take identifiers as strings, and an environment is a cons-list of them.

```
Ident : Set
Ident = String

Env : Set
Env = List Ident

data Err : Set where
  scope : (nm : Ident) → Err
```

```
Errs : Set
Errs = List Err
```

We use one type of error, a *scope*-error, consisting of an identifier *nm*. The constructor *scope* takes a value of type *Ident* as parameter. In a dependent type, we may give a name to such a value in the type signature. In this example, we give it the name *nm*. This name may be used in the remainder of the signature to specify predicates over *nm* (types that depend on *nm*). We do not require this functionality yet; later, we add an additional field to the *scope* constructor to represent a proof that *nm* is not in some environment.

A type $nm \in \Gamma$ (in *Set*) is an example of a predicate as mentioned above, where *nm* and Γ are value-level identifiers. A value of the above predicate is a proof that the predicate holds over the value-level identifiers.

In the example, we manipulate environments. We reason about identifiers in the environment. Two relations, $\_ \sqsubseteq \_$ and $\_ \in \_$, play an important role. A predicate $\Gamma_1 \sqsubseteq \Gamma_2$ specifies that Γ_1 is a subsequence of Γ_2, and $nm \in \Gamma$ that specifies that *nm* is an element of Γ. Such relations are representable in Agda as data types, where the clauses of the relation are represented as constructors. A value of such a type is a proof: a derivation tree with constructor-values as nodes.

We use the subsequence relation between environments in the example to reason about environments that we extend by preprending or appending identifiers of other environments. A subset relation would be another possibility, but former suffices and is slightly easier in use. The curly braces in the code below introduce implicit parameters that may be omitted if their contents can be fully determined by Agda through unification.

```
data _⊑_ : (Γ₁ : Env) → (Γ₂ : Env) → Set where
  trans   : {Γ₁ : Env} {Γ₂ : Env} {Γ₃ : Env} → Γ₁ ⊑ Γ₂ → Γ₂ ⊑ Γ₃ → Γ₁ ⊑ Γ₃
  subLeft  : {Γ₁ : Env} {Γ₂ : Env} → Γ₁ ⊑ (Γ₁ ++ Γ₂)
  subRight : {Γ₁ : Env} {Γ₂ : Env} → Γ₂ ⊑ (Γ₁ ++ Γ₂)
```

With *subLeft* and *subRight*, we can add arbitrary environments in front or after the environment on the right-hand side. With *trans* we repeat this process.

For membership in the environment, either the element has to be on the front of the list, or there is a proof that the element is in the tail.

```
data _∈_ : (nm : Ident) → (Γ : Env) → Set where
  here : {nm : Ident} {Γ' : Env} → nm ∈ (nm :: Γ')
  next : {nm : Ident} {nm' : Ident} {Γ : Env} → nm ∈ Γ → nm ∈ (nm' :: Γ)
```

With the above definitions, we prove lemma *inSubset* that, when an environment Γ' is a subsequence of Γ, and *nm* is an element in Γ', states that *nm* is also in Γ.

```
append : {nm : Ident} {Γ : Env} → (nm ∈ Γ) → (Γ' : Env) → (nm ∈ (Γ ++ Γ'))
append {nm} {.nm :: Γ} (here)      Γ' = here {nm} {Γ ++ Γ'}
append {nm} {nm' :: Γ} (next inΓ) Γ' = next (append {nm} {Γ} inΓ Γ')
append {_}  {[]}       ()          _
prefix : {nm : Ident} {Γ' : Env} → (nm ∈ Γ') → (Γ : Env) → (nm ∈ (Γ ++ Γ'))
```

```
prefix inΓ′ [ ]       = inΓ′
prefix inΓ′ (x :: Γ) = next (prefix inΓ′ Γ)

inSubset : {nm : Ident} {Γ : Env} {Γ′ : Env} → (Γ′ ⊑ Γ) → nm ∈ Γ′ → nm ∈ Γ
inSubset (subLeft {_} {Γ′}) inΓ′ = append inΓ′ Γ′
inSubset (subRight {Γ})     inΓ′ = prefix inΓ′ Γ
inSubset (trans subL subR) inΓ′ = inSubset subR (inSubset subL inΓ′)
```

The proof of *inSubset* consists of straightforward structural induction. The lemmas *append* and *prefix* take care of the base-cases. With *append*, the proof for Γ and $\Gamma +\!\!+ \Gamma'$ is the same until we find the element. We then produce a *here* with the appropriate environment. When we have a proof that an identifier is in the environment, we can never run into an empty environment without finding the identifier first. Agda's totally checker requires us to add a clause for when the environment is empty. The match against the proof can then not succeed, hence the match is called absurd (indicated with the empty parentheses), and the right-hand side may be omitted.

Similarly, we prove that given an identifier *nm* and an environment Γ, that we can either determine that *nm* is in Γ or that it is not in Γ. The sum-type $\_ \uplus \_$ (named *Either* in Haskell) provides constructors inj_1 (*Left* in Haskell) and inj_2 (*Right* in Haskell) for this purpose.

In this prove, we use *notFirst* to prove by contradiction that an identifier is not in the environment if it is not equal to the head of the environment, and neither in the tail of the environment. If it would be in the head, it conflicts with the former assumption. If it would be in the tail, it conflicts with latter assumption.

```
notFirst : {nm : Ident} {nm′ : Ident} {Γ : Env} → ¬(nm ≡ nm′) → ¬(nm′ ∈ Γ)
                                                → (nm′ ∈ (nm :: Γ)) → ⊥
notFirst prv₁ _    here          = prv₂ refl
notFirst _    prv₁ (next prv₂) = prv₁ prv₂
```

To find an element *nm′* in environment, we walk over the cons-list. If the environment is empty, the proof that the identifier is not in the list is trivial: if it would be, neither *here* nor *next* can match, because they expect a non-empty environment, hence we can only match with an absurd pattern, which provides the contradiction.

If the environment is not empty, then it has an identifier *nm* as the head. We use the string equality test '*mbEqual*′ to give us via constructor *yes* a proof of equality between *nm* and *nm′* or a proof of its negation via *no*. We wish to express the pattern $nm' \in_? (nm' :: \Gamma)$, with the intention of matching the case that *nm′* matches the head of the environment. In Agda, however, a pattern match must be strictly linear: an identifier may only be matched against once. To express that a value is the same as an already introduced value, we use a dot-pattern. This requires a proof that these are indeed the same, which is provided by a match against *refl* (the only constructor of an equality proof).

```
_ ∈? _ : (nm : Ident) → (Γ : Env) → ¬(nm ∈ Γ) ⊎ (nm ∈ Γ)
nm′ ∈? [ ] = inj₁ (λ())
nm′ ∈? (nm :: Γ)  with nm ≡? nm′
```

```
nm′ ∈? (.nm′ :: Γ) | yes refl = inj₂ here
nm′ ∈? (nm :: Γ)   | no prv′ with nm′ ∈? Γ
nm′ ∈? (nm :: Γ)   | no prv′  | inj₂ prv = inj₂ (next {nm′} {nm} prv)
nm′ ∈? (nm :: Γ)   | no prv′  | inj₁ prv = inj₁ (notFirst prv′ prv)
```

A with-expression is a variation on case-expressions in Haskell. It allows case distinction on a computed value, but also a refinement of previous pattern matches. Semantically, it computes the value of the scrutinee, then calls the function again with this value as additional parameter. Horizontal bars control to what clauses belong to what with-expression.

2.2 Attribute Grammar in Agda

In AG blocks, we define the grammar, attributes, and functions between attributes. The preprocessor takes these blocks and translates them to Agda functions. The compiler is an Agda function implemented as AG that takes a source-language term of the type *Source* and translates it to a target-language term of the type *Target*, provided that there are no scoping errors. Initially, we take *Source* and *Target* as the same language, without using any dependent types. Later, in Section 2.3, we specify that a term of type *Target* is free of scoping errors, and have to prove that our compiler indeed produces such terms.

To use Attribute Grammars to analyze *Source*-terms, we define a context-free grammar for it below. Later, we declare attributes on the nonterminals in the grammar, and give functions between these attributes. The *Source* language consists of a sequence (_ ◇ _) of *use* and *def* terms. Its operational semantics is simply: it evaluates to () if for every *use*, there is an equally named *def* in the sequence, otherwise evaluation diverges.

```
grammar Root : Set   -- start symbol of grammar and root of the AST
  prod root nonterm root : Source
grammar Source : Set
  prod use  term    nm    : Ident
  prod def  term    nm    : Ident
  prod _◇_ nonterm left   : Source
           nonterm right  : Source
```

The grammar actually represents an Agda-data type, where the productions are constructors, and terminals and nonterminals are the fields. We optionally generate this data-type from such a grammar declaration. The type for the *Target* language (for now structurally equal to *Source*) demonstrates.

```
data Target : Set where
  use  : (nm : Ident) → Target
  def  : (nm : Ident) → Target
  _◇_ : (left : Target) → (right : Target) → Target
```

With an Attribute Grammar, we translate an AST to its semantics, which is conceptually an AST node decorated with attributes. We represent the semantics of an AST

node as an Agda function that takes values for inherited attributes as inputs, and produces values for synthesized attributes as outputs. The conceptual decorated tree is the closure of this function.

For the example, we compute bottomup a synthesized attribute *gathEnv* that contains the identifiers defined by the term. Topdown we pass an inherited attribute *finEnv* that contains all the gathered identifiers of the term combined with the initial environment. Furthermore, we compute bottom up an attribute *errs*, containing errors for each undefined identifier, and an attribute *target* containing the compiled version of the source term.

From the above informal description of the attributes, we can already indicate a problem. The inherited attribute *finEnv* depends on the synthesized attribute *gathEnv*. This requires us to provide a result of the function as part of the argument of the same function. Such a circular programs are not allowed by Agda's termination checker. However, there is no circular dependency when we compute the attributes in two passes or more passes over the AST. In the first pass, we compute attribute *gathEnv*. In the second pass, we take *gathEnv* as resulting from the first pass and use it to compute the value for the inherited attribute *finEnv* for the second pass. For the class of Ordered Attribute Grammars [6], a multi-pass algorithm is derivable. Earlier work showed that this class is sufficiently strong for type checking Haskell. Several extensions may be needed when fixpoint iteration is required [10].

In Agda, we can express a multi-pass algorithm as a coroutine. A coroutine is a function that can be invoked (visited) multiple times, each time it may take additional parameters and yield results. As semantics of a AST node, we take a coroutine that which each invocation takes some of the remaining inherited attributes as a record, and computes a record with some of the remaining synthesized attributes. With an interface declaration, we express statically what attributes are computed in what visit.

```
itf Root
  visit compile  inh initEnv  : Env
                 syn errs     : Errs
                 syn target   : Target
itf Source
  visit analyze  syn gathEnv : Env
  visit translate inh finEnv  : Env
                 syn errs     : Errs
                 syn target   : Target
```

An attribute may only be declared once. However, an inherited attribute with the name x is considered to be different from a synthesized attribute x. The visits are ordered totally by the order of appearance. For an AST node of nonterminal Root, there is only one visit, named *compile*, which takes the initial environment and produces the outcome. It does so by invoking the two visits on its *Source* subtree. First the visit *analyze* to gather the identifier, and then the visit *translate* to produce the outcome.

The implementation of the coroutine is derived from functions that define attributes. These functions are called rules, and we specify them per production with a datasem-

block. The left-hand side consists of a pattern that refers to one or more attributes, and the right-hand side is an Agda expression that may refer to attributes.

```
datasem Root
  prod root
    root.finEnv = root.gathEnv ⫩ lhs.initEnv
    lhs.errs    = root.errs
    lhs.target  = root.target
```

An attribute reference is denoted *childname.attrname*. The children names *loc* and *lhs* are special. With *lhs* we refer to the inherited and synthesized attributes of the current AST node. With *loc* we refer to local attributes of the node. A terminal field *x* of a production is in scope as attribute *loc.x*. A nonterminal field *c* of a production is in scope as child named *c*, with attributes as defined by the interface of the corresponding nonterminal. An attribute *c.x* on the left-hand side refers to a synthesized attribute if *c* = *lhs*, and an inherited attribute otherwise. On the right-hand side, it is exactly the other way around. To refer on the right-hand side to synthesized attributes of *lhs*, or to inherited attributes of a child[1], the attribute reference has to be additionally prefixed with *inh.* or *syn.*.

For the *use*-production of *Source*, we check if the element is in the environment. If not, we produce a singleton error-list. The *Target* term is a copy of the *Source* term. For *def*, no errors arise, and again the *Target* term is a copy of the *Source* term. Finally, for _ ⋄ _, we pass the *finEnv* down to both children, synthesize the *gathEnv* and *errs* of the children, and synthesize the target term.

```
datasem Source
  prod use
    lhs.errs    with loc.nm ∈? lhs.finEnv
                | inj₁ _ = [ scope nm ]
                | inj₂ _ = [ ]
    lhs.target  = use loc.nm
  prod def
    lhs.errs    = [ ]
    lhs.target  = def loc.nm
  prod _ ⋄ _
    left.finEnv   = lhs.finEnv
    right.finEnv  = lhs.finEnv
    lhs.gathEnv   = left.gathEnv ⫩ right.gathEnv
    lhs.errs      = left.errs ⫩ right.errs
    lhs.target    = left.target ⋄ right.target
```

Some advantages of Attribute Grammars show up in this example. Firstly, the order of appearance of rules is irrelevant. This allows us to write the rules that belong to

[1] This is rarely useful with a conventional AG and can be simulated with local attributes. However, in a dependently-typed AG, we may wish to refer to such values in a predicate, as we see in a later example. Then it is convenient to be able to directly refer to such a value.

each other in separate datasem-block of the same nonterminal, and use the preprocessor to merge these separate blocks into a single block. For large grammars with many attributes, such as UHC, this is an essential asset to keep the code maintainable.

Furthermore, the rules for many attributes exhibit a standard pattern, especially when the grammar has many productions. For example, values for many attributes are simply passed down (*finEnv*, for example), or are a combination of some of the attributes of the children (*errs* and *gathEnv*, for example). We provide default rules [10] to abstract over these patterns. Children are ordered per production (for example, alphabetic order). When for a child c an attribute x is not explicitly defined, but there is default-rule for x, then we take as value for x the right-hand side of the default rule applied to a list of all the x attributes of the children smaller than c. In particular, first element of this list is the nearest smaller child. The last element in this list is *lhs* (if it has an attribute x).

```
datasem Source
  default errs = concat    -- i.e. left.errs ++ right.errs
  default finEnv = last    -- i.e. lhs.finEnv
```

In the AG code for UHC about a third of the attributes have an explicit definition. All others are provided by such default-rules.

To effectively be able to use Attribute Grammars, the above two features are a necessity. In the next section, we construct a dependently-typed variation on the above Attribute Grammar, and need extensions to the AG formalism to retain these two features.

2.3 Dependently-Typed Attribute Grammar in Agda

We change the definition of the *Target* language. When constructing a term in this language, we demand a proof that all identifiers exist in the environment. For that, we give *Target* a type depending on a value: the final environment. Furthermore, we demand for each scoping error a proof that the identifier that caused the error is not in the environment. In this section, we show how a compiler that has these properties by construction. These are just a small number of properties. We can define many more, such as demanding that identifiers are not defined duplicately, or that error messages correspond to a a used identifier in the source language. As we see in this section, an Attribute Grammar is suitable for exactly this purpose: the additional predicates and proofs just become additional attributes and rules.

```
data Target : (Γ : Env) → Set where
  def   : ∀ {Γ} (nm : Ident) → (nm ∈ Γ) → Target Γ
  use   : ∀ {Γ} (nm : Ident) → (nm ∈ Γ) → Target Γ
  _◇_ : ∀ {Γ} → (left : Target Γ) → (right : Target Γ) → Target Γ

data Err : Env → Set where
  scope : {Γ : Env} (nm : Ident) → ¬(nm ∈ Γ) → Err Γ

Errs : Env → Set
Errs env = List (Err env)
```

In the previous section, we returned both a list of errors and a target term as result, with the hidden invariant that the target term is invalid when there are errors. The advantage is that we could refer to both values as separate attributes. A fine granularity of attributes is important in order to use default rules, and to be able to refer to values without having to unpack it first. This also has a disadvantage: the hidden invariant is not enforced. Patterns like the above occur often in the code of UHC.

With the dependently-typed *Target* type, we are unable to construct such a term in the presence of errors. The hidden invariant is made explicit. However, we cannot construct both error messages and a target type anymore. Instead of two attributes, we now return a single attribute of type (*Errs inh.finEnv*)⊎(*Target inh.finEnv*), that contains either a list of error messages or a valid target term.

Furthermore, both the error messages and the target term take both the final environment as parameter. To keep knowledge about this environment hidden inside the compiler, we wrap the result in a data type *Outcome*, which hides the environment via an existential. We produce this type at the root of the AST.

```
data Outcome : Set where
  outcome : ∀ {Γ} → (Errs Γ) ⊎ (Target Γ) → Outcome
```

The interface now contains dependent-types that may depend on values of attributes. For example, we require a proof that the gathered environment is a subsequence of the final environment. The type of this proof refers to the two respective attributes.

```
itf Root
  visit compile  inh initEnv    : Env
                 syn outcome    : Outcome
itf Source
  visit analyze  syn gathEnv    : Env
  visit translate inh finEnv    : Env
                 inh gathInFin  : syn.gathEnv ⊆ inh.finEnv
                 syn outcome    : (Errs inh.finEnv) ⊎ (Target inh.finEnv)
```

The dependencies between attributes in the interface must be acyclic. Furthermore, there may not be references to attributes in later visits. That ensures that we can generate a type signature for a coroutine with this interface.

As additional work, we need to construct the proofs for the attribute *gathInFin*, and membership proofs for constructors *use* and *def*. At the root, we prefix the initial environment with the gathered environment. We get the required proof via *subRight*.

```
datasem Root
  prod root
    root.finEnv    = lhs.initEnv ⧺ root.gathEnv
    root.gathInFin = subRight
    lhs.outcome    = outcome root.outcome
```

For production *use*, we get both of the proofs we desire from the membership test of the name in the environment. For *def*, we have to do a bit more work. The gathered environment for this case is a singleton environment, so we get the proof for this

environment via *here*. Since we also get a proof as *lhs.gathInFin* that the gathered environment is a subsequence of the final environment, we get the desired result via lemma *inSubset*. Finally, for the '*diam*' production, we only have to consider the possible outcomes of the two children.

```
datasem Source
  default finEnv    = last
  default gathEnv = concat
  prod use
    lhs.outcome     with loc.nm ∈? lhs.finEnv
                    | inj1 notIn = inj1 [scope loc.nm notIn]
                    | inj2 isIn  = inj2 (use loc.nm isIn)
  prod def
    loc.inFin       = inSubset lhs.gathInFin here
    lhs.outcome     = inj2 (def loc.nm loc.inFin)
  prod _ ◇ _
    left.gathInFin  = trans subLeft lhs.gathInFin
    right.gathInFin = trans (subRight {syn.lhs.gathEnv} {lhs.finEnv})
                            lhs.gathInFin
    lhs.outcome     with left.outcome
                    | inj1 es with right.outcome
                    | inj1 es1 | inj1 es2 = inj1 (es1 ++ es2)
                    | inj1 es1 | inj2 _   = inj1 es1
                    | inj2 t1  with left.outcome
                    | inj2 t1  | inj1 es1 = inj1 es1
                    | inj2 t1  | inj2 t2  = inj2 (t1 ◇ t2)
```

The above code shows again an advantage of using Attribute Grammars: we can easily add additional predicates and write separate rules for them. However, this example also shows a problem: we merged what used to be two attributes into a single attribute *outcome*. This causes several problems that become more urgent when the AG gets larger and has more attributes.

Firstly, two attributes that we could before give separate rules for, now have to be written in a large rule. This affects the ability to describe and arrange rules for various aspects separately. For example, we cannot reason about the computations for errors separately from the computations related to target terms. A fine granularity of attributes is the advantage that an Attribute Grammar has compared to a conventional catamorphism.

Secondly, when an attribute has a type with more than one inhabitant, we can only match against the contents in the right-hand side of a rule. When multiple rules refer to this attribute, they duplicate the match in their right-hand sides. When the match also opens an existential, it is also opened twice, resulting in incomparable values that are actually the same. To remedy this situation, we would like to be able to write rules in the context of having matched against another attribute.

Finally, the merged rules in the example above prevented us from using a default rule for the errors. If we had an error attribute available for all children that returned an inj_1 then the default-rule produces exactly the expressions as we now wrote manually.

2.4 Dependent Attribute Grammars with Context

To improve attribute granularity, we distinguish context in the interface. A visit may result in one or more different outcomes, and we can match against these outcomes.

```
itf Root
  visit compile
    inh initEnv    : Env
    syn outcome    : Outcome
itf Source
  visit analyze
    syn gathEnv    : Env
  visit translate
    inh finEnv     : Env
    inh gathInFin  : syn.gathEnv ⊆ inh.finEnv
    context failure
      syn errs     : Errs inh.finEnv
    context success
      syn target   : Target inh.finEnv
```

Furthermore, we provide a more fine-grained mechanism to specify rules. We organize a production as a sequence of visits, with for each visit a number of clauses. In a clause, the next visit is defined. This structure forms a tree, with a structure largely defined by the interface: rules may be specified at each level of the tree, thus in a production, visit or clause. The lifetime of a rule is limited to the subtree it appears in, and we may schedule the evaluation of a rule anywhere in this lifetime (not earlier).

The subtree also introduces a scope. The common attributes of clauses are also accessible from the parent-scope. This way, we can define rules in an as high-as-possible scope and let the rules be ordered automatically based on their attribute dependencies. When a rule depends on a clause or particular visit, we can restrict its lifetime to such a visit.

To prevent notational overhead, in the code for a production, we only have to introduce the visits that matter for the lifetime and contexts of rules and clauses. Those that are not explicitly introduced are done so implicitly. If a visit has only one clause, then that clause may be omitted, and we introduce an anonymous clause for it.

During the invocation of a visit, its clauses are tried in the order of appearance. The execution of rules may fail under certain conditions (explained below) and cause a backtrack to the next clause. The result context of the visit can be specified per clause, or for all clauses at the same time per visit.

Moreover, we introduce an internal visit, which is not visible in the interface, but allows a choice of clause to be made during the execution of the visit.

```
datasem Root
  prod root
    root.finEnv     = root.gathEnv ⫩ lhs.initEnv
    root.gathInFin = subLeft
    internal
      clause rootOk : success
        context translate of root : failure
        lhs.outcome = outcome (inj1 root.errs)
      clause rootFail : success
        context translate of root : success
        lhs.outcome = outcome (inj2 root.target)
```

For the semantics of *Source*, we distinguish separate attributes and rules in different contexts. For example, in the *use* production, we match against the outcome of the membership test, and have two clauses: one for each outcome.

```
datasem Source
  default finEnv   = last
  default gathEnv = concat
  prod use
    loc.checkIn = loc.nm ∈? lhs.finEnv
    visit translate
      clause notInEnv : failure
        match inj1 notIn = loc.checkIn
        lhs.errs         = [scope loc.nm notIn]
      clause isInEnv : success
        match inj2 isIn  = loc.checkIn
        lhs.target       = use loc.nm isIn
  prod def
    loc.inFin  = inSubset lhs.gathInFin here
    lhs.target = def loc.nm loc.inFin
    visit translate : success
  prod _ ⋄ _
    left.gathInFin   = trans subLeft lhs.gathInFin
    right.gathInFin  = trans (subRight {syn.lhs.gathEnv} {lhs.finEnv})
                             lhs.gathInFin
    visit translate
      clause both : success
        context translate of left  : success
        context translate of right : success
        lhs.target       = left.target ⋄ right.target
      clause contexts : failure
        default errs = concat
```

If a visit results in more than one context and a match against this context is omitted, then we are not allowed to use any of the synthesized attributes of that visit. As a short-

cut, we may introduce a pseudo clause **contexts**, which stands for multiple clauses: one for each combination of remaining contexts. The default-rules may refer to attributes of children available due to matches against contexts.

3 RULER-FRONT and RULER-CORE

i ::= **itf** $I\ v$ -- interface decl, with first visit v
v ::= **visit** x **inh** $\overline{a}\ \overline{k}$ -- one visit
| **tail** τ -- last continuation
k ::= **context** x **syn** $\overline{a}\ v$ -- one context
a ::= $x :: \tau$ -- attribute decl, with Haskell type *hty*
s ::= **sem** $x :: I\ t$ -- semantics expr, defines nonterm x
t ::= **visit** $x\ \overline{r}\ \overline{c}$ -- calleable visit
| **internal** $\overline{r}\ \overline{c}$ -- internal visit
| **tail** e -- semantics for the remainder
c ::= **clause** $n\ \overline{r}\ t$ -- clause definition, with next visit t
n ::= **contexts** | x
r ::= $p = e$ -- assert-rule, evaluates monadic e
| **match** p **and** $\overline{q} = e$ -- match-rule, backtracking variant
| **context** x **of** $c : x$ -- context match
| **invoke** x **of** c -- invoke-rule, invokes x on c, while e
| **attach** x **of** $c : I = e$ -- attach-rule, attaches a part. eval. child
| $p =$ **detach** x **of** c -- detach-rule, stores a child in an attr
p ::= o -- attribute def
| $.e$ -- dot pattern
| $x\ \overline{p}$ -- constructor match
q ::= $o \equiv p$ -- pattern refinement of attr o
o ::= $x.x$ -- expression, attribute occurrence
x, I, e -- identifiers, interface names, expressions respectively

4 Translation to Agda

5 Related Work

Dependent types originate in Martin-Löf's Type Theory. Various dependently-typed programming languages are gaining popularity, including Agda [12], Epigram [9], and Coq [1]. We present the ideas in this paper with Agda as host language. In principle, the presented ideas are applicable to all these languages, as long as the language has a concept of a dependent pattern match.

In Coq and Epigram, a program is written via interactive theorem proving with tactics or commands. The preprocessor-based approach of this paper suits a declarative approach more, hence we choose for Agda as programming language.

Attribute grammars [7, 8] were considered to be a promising implementation for compiler construction. Recently, many Attribute Grammar systems arose for mainstream languages, such as Silver [16] and JastAdd [4] for Java, and UUAG [14] for Haskell. To our knowledge, Attribute Grammars with more sexy types have not been investigated yet.

In certain languages it is possible to implement AGs via meta-programming facilities, which obliviates the need of a preprocessor. Viera, et al. [15] show how to implement AGs into Haskell through type level programming. The ideas presented in this paper are orthogonal to such approaches. In Haskell, the type-level programming is difficult to express. In Agda, this would be easier. However, to make conveniently use of AG-functionality without a preprocessor, Agda would require programmer-controlled proof-search functionality.

Several attribute grammar techniques are important to our work. Kastens [6] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, which allows us to generate coroutines such that the circularity checker accepts the program as terminating.

Boyland [2] introduces conditional attribute grammars. In such a grammar, semantic rules may be guarded. A rule may be evaluated if its guard holds. Evaluation of guards may influence the evaluation order, which makes the evaluation less predictable. In comparison, in our clauses-in-visits model, we have to explicitly indicate in what visits guards are evaluated (the match-statements of a clause), which makes evaluation clear. Our approach has the additional benefit that children may be conditionally introduced and visited.

The work for this paper is carried out for a research project to support the implementation of the type inference component of the Utrecht Haskell compiler. In the workshop version of this paper [11], we presented an earlier version of RULER-FRONT's clauses-per-visit model to allow attribute grammars to implement functions that perform case distinction on more than a single AST. In a later paper [10], we improved on this model to allow iteration of visits, and dynamic growing of trees, to model fixpoint construction of proof trees. That work was carried out using Haskell as target language.

6 Conclusion

We investigated Attribute Grammars with dependently-typed attributes. We designed a language for AGs, RULER-FRONT, that has several extensions to specifically support AGs, and implemented this language as a preprocessor for Agda. RULER-FRONT is a variation on Ordered Attribute Grammars, which are needed to ensure that the generated code terminates if all the attribute rules terminate. Furthermore, in the types of attributes we may refer to other attributes defined for the nonterminal. Finally, rules can be given not only per production, but also per visit. A visit we can split up in clauses to deal with context-information resulting from visits to children as well as pattern matching in rules.

As future work, the question is to what extend we can exploit patterns in Attribute Grammar descriptions to generate proof boilerplate. From a practical perspective, for example, to deal with fixpoint iteration in AGs and its termination, or to deal with

lookups and insertions in the environment. From a theoretic perspective, for example, to generate templates related to type soundness and subject reduction.

Acknowledgments. This work was supported by Microsoft Research through its European PhD Scholarship Programme.

References

1. Bertot, Y.: A short presentation of coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 5170, pp. 12–16. Springer (2008)
2. Boyland, J.: Conditional Attribute Grammars. ACM TPLS 18(1), 73–108 (1996)
3. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. pp. 93–104. ACM, New York, NY, USA (2009)
4. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA. pp. 1–18. ACM (2007)
5. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for Learning Haskell. In: ACM SIGPLAN Haskell Workshop (HW'03). pp. 62 – 71. ACM Press, New York (Sep 2003)
6. Kastens, U.: Ordered Attributed Grammars. Acta Inf. 13, 229–256 (1980)
7. Knuth, D.E.: Semantics of Context-Free Languages. Math. Sys. Theory 2(2), 127–145 (1968)
8. Knuth, D.E.: The Genesis of Attribute Grammars. In: WAGA. pp. 1–12 (1990)
9. McBride, C.: Epigram: Practical programming with dependent types. In: Vene, V., Uustalu, T. (eds.) Advanced Functional Programming. Lecture Notes in Computer Science, vol. 3622, pp. 130–170. Springer (2004)
10. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative Type Inference with Attribute Grammars. In: Proceedings of the International Conference on Generative Programming and Component Engineering (2010)
11. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Visit Functions for the Semantics of Programming Languages. http://people.cs.uu.nl/ariem/wgt10-visit.pdf (2010)
12. Norell, U.: Dependently typed programming in agda. In: Kennedy, A., Ahmed, A. (eds.) TLDI. pp. 1–2. ACM (2009)
13. Schrage, M.M., Jeuring, J.T.: Proxima - A presentation-oriented editor for structured documents (2004)
14. Universiteit Utrecht: Universiteit Utrecht Attribute Grammar System. http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem
15. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In: Hutton, G., Tolmach, A.P. (eds.) ICFP. pp. 245–256. ACM (2009)
16. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electr. Notes Theor. Comput. Sci. 203(2), 103–116 (2008)

Mapping Interpreters onto Runtime Support

Stijn Timbermont*

Software Languages Lab – Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
stimberm@vub.ac.be

1 Problem Description

The job of an interpreter is to generate a computational process that establishes a mapping between the language constructs and features of the defined language and those available in the defining languages. It is often possible to map a feature in the defined language onto a similar feature in the defining language. For example, an *if* construct in the defined language can often be conveniently implemented using a similar *if* construct in the defining language. There is a significant distinction between lifting the language constructs and features of the defining language directly, and manually implementing the necessary runtime support to realize the mapping. The latter is of course necessary if the defined language requires a feature that is lacking in the defining language. A good example of this distinction is automatic memory management: it is usually much easier to implement a garbage collected language in another garbage collected language, than to implement it in one that is not. A similar example is the implementation of tail-call elimination. An extreme case of this kind of language implementation reuse can be found in meta-circular interpreters. For example, a Scheme interpreter written in Scheme has the opportunity to not only reuse the automatic memory management support, but also the implementation for closures, first-class continuations, tail-call elimination, etc. However, most interpreters are not meta-circular and therefore suffer from the mismatch between the defined and the defining language.

The problem we address here is that the structure of an interpreter is not only dictated by the set of features requires by the defined language, but also by the interface to the runtime support provided by the defining language that is necessary to implement the defined language. We illustrate this problem in section 3. This influence on the interpreter structure is problematic in the following two ways.

- The structure of the interpreter can be problematic if it is needlessly convoluted and therefore difficult to write, read and maintain. This can be the consequence of an ill-suited or low-level interface to necessary runtime support.

* Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

– Evolution of the runtime support implementation is hampered when it requires dramatic changes to the interpreter. By consequence, evolving the defined language (e.g. adding a language construct) can be problematic if it requires additional runtime support which in turn requires drastic changes to the original interpreter.

2 Mapping Interpreters onto Runtime Support

We address this problem by providing a method to write an interpreter first and then build the necessary runtime support around it, rather than building the interpreter directly on top of the runtime support. This means that the interpreter abstracts over the concrete runtime support, and therefore allows for variation. The two main challenges in this approach are as follows.

– We need to be able to define an interpreter without assuming too much about how the runtime support is provided, and to automatically adapt it to match the target runtime support. We draw inspiration from research on defunctionalized interpreters [6,1,4] that bridges the gap between functional interpreters and abstract machines. The former are an elegant specification of the semantics of the defined programming language, whereas the latter is much better suited to specify the mapping onto the concrete runtime support.
– After adapting the interpreter into a form that is well-suited for a particular runtime support implementation, we need to be able to integrate the two parts. The challenge here is to do so without having to manually go through the transformed interpreter code. To solve this we use state-of-the-art techniques for generic programming [8,2]. The idea is that the transformed interpreter is generic w.r.t. the runtime support. Instantiating the generated interpreter with a concrete runtime support implementation then results in a complete and working system.

3 Impact of Runtime Support on Interpreter Structure

To illustrate the impact of runtime support on interpreter structure, we consider a piece of interpreter code and show how it has to be adapted to a number of variations w.r.t. the interface to required runtime support. Suppose the example is part of an interpreter for a language that has a construct for array lookup. Such an expression might look as follows.

```
arr[i+1]
```

The expression has two subexpressions: one for the array and one for the index. Evaluation proceeds by evaluating these two subexpressions and performing the actual lookup operation. Such an evaluation function can be written in Scheme as follows.

```
(define (eval-array-lookup al)
  (let ((arr (eval (al.array al)))
        (idx (eval (al.index al))))
    (array-lookup arr idx)))
```

We assume the `aa.*` functions to be getter function that extract the subexpressions from an array lookup expression, and `array-lookup` to be the function of the memory model that actually performs the lookup.

This implementation clearly captures the semantics of an array lookup expression, without unnecessary clutter. The fact that such expressions may contain arbitrary subexpressions — a feature of the defined language — is mapped onto recursive invocations of `eval` and the introduction of local variables — both features of the defining language (reusing the underlying runtime support for a call stack).

Now suppose that instead of using Scheme as a defining language, we use C. This means that automatic memory management cannot be simply reused anymore. There is the option of using a conservative garbage collection strategy, but if that is undesirable we have to provide our own garbage collection algorithm. The latter requires an invasive modification to the interpreter: all root pointers must be known to the algorithm. This affects our code example, even though there is no allocation going on. The adapted code might look as follows (we continue to use a Scheme like syntax for consistency).

```
(define (eval-array-lookup al)
  (let ((arr (eval (aa.array al))))
    (register arr)
    (let ((idx (eval (aa.index al))))
      (unregister)
      (array-lookup arr idx))))
```

The idea is that, since every invocation of `eval` may trigger garbage collection, local variables that need to be preserved across evaluations must be registered with the memory management system. Note that the result of the second evaluation does not need to be preserved since there is no evaluation in the rest of the function. Furthermore, when all recursive evaluations have taken place, the roots must be explicitly unregistered again.

4 Selectively Defunctionalized Interpreters

We draw inspiration from research on defunctionalized interpreters [6,1,4] that relates functional interpreters and abstract machines by means of standard transformations. The core idea is that a CPS transformation [3,7] makes the control flow explicit and that defunctionalization [6] turns the continuations into an explicit call stack. While the related work provides the insight that these transformations relate functional interpreters and abstract machines, it does not provide an algorithm that can perform this transformation automatically. This is

because the transformations are applied selectively, based on the concrete interpreter at hand. More concretely, the CPS transformation should only be applied on expressions that actually perform an evaluation of a subexpression. Detecting this can be achieved via selective CPS transformation [5]. It employs a simple effect annotation mechanism to distinguish computations that do and do not invoke an evaluation. We integrate this approach with a polymorphic and more expressive type and effect system based on [9].

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19, New York, NY, USA, 2003. ACM.
2. Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM.
3. Oliver Danvy and Andrzex Filinski. Representing control: a study of the cps transformation. *Mathematical Structures in Computer Science*, 2(04):361–391, 1992.
4. Olivier Danvy. Defunctionalized interpreters for programming languages. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 131–142, New York, NY, USA, 2008. ACM.
5. Lasse R. Nielsen and BRICS. A selective cps transformation. *Electronic Notes in Theoretical Computer Science*, 45:311 – 331, 2001. MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.
6. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM.
7. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 288–298, New York, NY, USA, 1992. ACM.
8. Jeremy G. Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, 2005. ACM.
9. Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect discipline. *Information and Computation*, 111(2):245–296, 1994.

Hiding State in CλaSH Hardware Descriptions

Marco Gerards, Christiaan Baaij, Jan Kuper, and Matthijs Kooijman

University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
{m.e.t.gerards,c.p.r.baaij,j.kuper}@ewi.utwente.nl

Abstract. Synchronous hardware can be modelled as a mapping from input and state to output and a new state. Functions in this form are referred to as transition functions. It is natural to use a functional language to implement transition functions. The CλaSH compiler is capable of translating Haskell code written in this form to VHDL. Modelling hardware using multiple components is convenient. Components can be considered as instantiations of functions. To avoid packing and unpacking state when composing components, functions are lifted to arrows. By using arrows the chance of making errors will decrease as it is not required to manually (un)pack the state. Furthermore, the Haskell do-syntax for arrows is a pleasant notation for describing hardware designs.

1 Introduction

In synchronous digital hardware, a clock signal is used as the heartbeat of the digital circuit. Synchronous digital hardware can be modelled using a Mealy machine. In a Mealy machine, current inputs (i) and the current state (s) which is stored inside registers are mapped, using a transition function, to a new state (s') and output of the circuit (o), see figure 1.

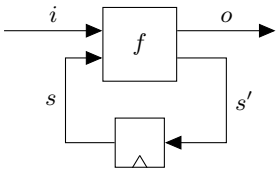


Fig. 1. Mealy machine

This transition function can be seen as a mathematical function, which is applied to the inputs and state at every clock cycle. However, when designing digital hardware, viewing the transition function as a pure mathematical function is not enough. For instance, $(a + b) \cdot c$ and $a \cdot c + b \cdot c$ are mathematically equivalent since $\cdot$ is distributive over $+$. However, in a hardware implementation the

latter expression would yield a bigger circuit. When designing hardware, some of the design steps result in different hardware with exactly the same functionality, but with different non-functional properties such as speed or area. Mostly, digital logic is implemented as an Application-Specific Integrated Circuits (ASICs) or instantiated in a Field Programmable Gate Arrays (FPGAs). FPGAs are reconfigurable logic devices which are mainly used for prototyping and decreasing the time to market. Since manual translation of a transition function to descriptions that can be used to produce the physical hardware is very cumbersome, this is often automated using software. The hardware is described using a Hardware Description Language (HDL), for which synthesis tools exists to translate the source code to a low level hardware description. This description specifies which components are used and how they are connected. The physical place of components and their interconnect inside the ASIC or FPGA can be assigned using a Place And Route (PAR) tool. A particularly nice feature of many HDLs is, that they allow modularization, which enable engineers to separately design relatively small pieces of hardware. The HDLs have syntax to compose components and instantiate components multiple times.

Two popular HDLs are VHDL and Verilog. Although it is possible to use these languages and the respective tools to design hardware and synthesise it, the source code descriptions are far from the mathematical function we started with. Furthermore, it is hard to prove that functionality remains the same after some design steps. In our experience, the hardware descriptions written in a (modified) Haskell subset are very compact and well readable when compared to their equivalent VHDL descriptions. Because of these reasons, it is natural to use functional programming languages to design synchronous digital hardware. In [2, 9] we have introduced a modified subset of Haskell, together with a compiler called CλaSH, which is based on the Glassgow Haskell Compiler (GHC).

In [9] the streaming reduction circuit, introduced in [6], was used as an example of how to describe hardware in a functional language. When using CλaSH, the user has to pass the state and current input of the circuit to the transition function. Since most circuits consist of many components, which in turn can consist of subcomponents, the user has to thread the state through the functions by packing and unpacking it. This might seem unnatural when viewing the circuit as a connection of components. The states are local to their corresponding instance of a component and only the inputs and outputs of components are connected. Furthermore, distributing and collecting the state manually increases the chance of making errors.

In this paper, we describe how to hide the state from the user by using an *automata arrow* [11]. Each arrow describes a component, which can be combined with other arrows (components). It is possible to combine multiple arrows to a single arrow, which is similar to combining many subcomponents to a single component. The main contribution of this article is showing how to deal with state when designing synchronous hardware using Haskell and presenting this using a nontrivial example.

Section 2 introduces related work and compares this to our own work. Arrows will be shortly discussed in section 3. Section 4 explains how to deal with the hardware state, when designing synchronous hardware in Haskell. In section 5, the streaming reduction circuit[2] is introduced and implementations with and without arrows are compared.

2 Related work

Where the CλaSH compiler takes Haskell code as input, Lava[3] and ForSyDe[13] are domain specific embedded languages defined in Haskell. Both languages are stream processing languages. They operate on infinite streams. The state of synchronous hardware can be modelled using a delay function. By using this function, the state of the hardware is introduced. Instead of defining mappings from streams to streams, CλaSH defines a mapping from current input and current state to the next state and output, which corresponds to a Mealy machine. Since the input of CλaSH is not a domain specific language, all choice constructs in Haskell (if, guards, pattern matching, etc) are available. Lava has only the "mux" primitive, ForSyDe supports the if-then-else and case-expressions. Like Kansas Lava [7] and ForSyDe, CλaSH has support for integer types and primitive operations; Chalmers Lava has only support for the bit type and related primitives. CλaSH, Lava and ForSyDe support polymorphic, higher-order functions. ForSyDe requires explicit wrapping of functions and processes and also explicit component instantiations, making dscriptions in ForSyDe more cluttered than those in CλaSH.

In VHDL [1], all components are created using component declarations and connected using port maps. From variable and signal declarations in VHDL it is not clear if these variables and signals will become state. This depends on the actual code, not on the declarations. When using CλaSH, this is more transparant, as the current and next states are explicitly defined. Higher-level abstractions are cumbersome in VHDL, functional languages are better suited when high-level abstractions are desired.

In [11], arrows are introduced and circuits using delay functions are taken as an example. In section 5, it is shown that arrows can also be used for functional hardware modelled with Mealy machines whereas examples in [11] do not make the state explicit in the function arguments and use a delay function instead. In the examples in [11], only relatively small hardware designs were explored. We will show it is possible, using Haskell and CλaSH, to create relatively big hardware designs. In our approach we will use the automata arrow as introduced in [11].

3 Arrows

This section briefly discusses arrows in Haskell, enough to understand the remainder of this article. For an elaborate discussion we refer to [8] or [11] which both contain an excellent introduction to arrows in Haskell.

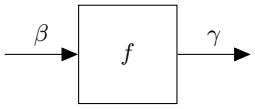


Fig. 2. An arrow

Arrows can be used to describe components, which have several inputs and several outputs, and their compositions. An arrow can be depicted graphically as in figure 2, where β and γ denote the types of the input and output data, respectively, and where f denotes the associated function from β to γ. Since arrows provide a general structure, Haskell libraries [8, 11] have been defined with functions common to many arrows. Examples of arrows are functions, stream processors, state transformers, etc.

Arrows give a uniform interface for composition. Every arrow in Haskell is an instance of the type class *Arrow* and of the type class *Category*. However, in this article we will use the notation as used in [11] and introduce arrows as a single type class *Arrow*.

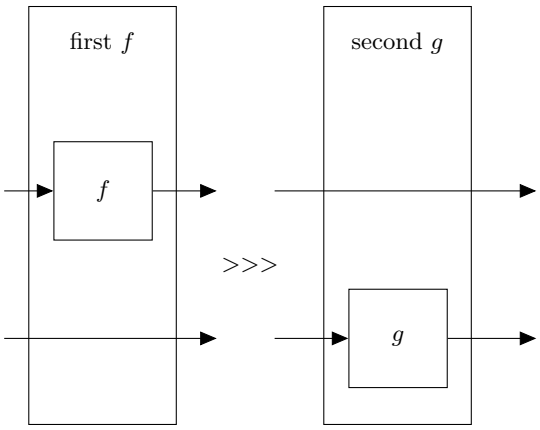


Fig. 3. Composition of arrows using first, second and >>>

For every arrow, sequential and parallel composition can be defined. To create an arrow from a regular function, the function *pure* is used. The function >>> takes two arrows and composes them such that the output of the first arrow is connected to the input of the second arrow. For parallel composition, the function *first* is used. The function *first* takes an arrow with input type β and output type γ and creates a new arrow with input and output types respectively (β, δ) and (γ, δ). The arrow that is its argument is only applied to the first element of the tuple. The second element in the tuple will not be modified. The

function *second* is similar to first, except that it applies the arrow to the second element of the tuple. This function can be defined by using *first*. In figure 3, the expression *(first f) >>> (second g)* is shown graphically. The type class Arrow is defined as in listing 1.

```
class Arrow α where
  pure  :: (β → γ) → α β γ
  (>>>) :: α β γ → α γ δ → α β δ
  first :: α β γ → α (β,δ) (γ,δ)
```

Listing 1. The Arrow type class

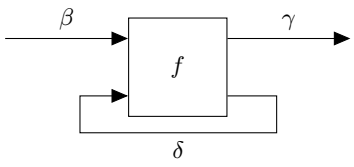


Fig. 4. An arrow loop

Using these operators all parallel and sequential structures can be created. To create feedback loops, another type class called *ArrowLoop* is required, see figure 4. This type class is defined as in listing 2.

```
class Arrow α ⇒ ArrowLoop α where
    loop :: α (β, δ) (γ, δ) → α β γ
```

Listing 2. The ArrowLoop type class

To model hardware, we use one specific arrow, namely the automata arrow. The automata arrow, described in [11], takes an input and produces an output together with a new automata arrow. We will use this to store the state inside the new arrow. In the next section a function is defined to lift a transition function to an automata arrow. The reason why we use the automata arrow together with the lifting function, instead of the circuit arrow from [11] is that the new arrow will have strong correspondence to the transition function. When using the form we propose, the arrow (which contains the state) receives an input and produces a new arrow (which contains state) together with an output. The automata arrow is defined in listing 3.

```
newtype Aut i o = A {
      apply :: i -> (o, Aut i o)
}

instance Arrow Aut where
    pure f = A (\b -> (f b, pure f))
    (A f) >>> (A g) = A (\b -> let (c,f') = f b
                                  (d,g') = g c
                              in (d, f' >>> g'))
    first (A f) = A (\(b,d) -> let (c,f') = f b
                               in ((c,d), first f'))

instance ArrowLoop Aut where
    loop (A f) = A (\i -> let ((c,d), f') = f (i, d)
                          in (c, loop f'))
```

Listing 3. Definition of the Automata Arrow

4 State

To model a Mealy machine, a transition function maps the input and the state to output and a new state, as was explained in section 1. In CλaSH, this state is a function argument to the transition function. All transition functions in CλaSH are defined as

```
transition :: state → input → (state, output)
```

The input state and output state have the same type (state), as both correspond to the register contents. The types input, output and state can be freely constructed using a predefined set of types, under the restriction that their sizes are finite and fixed at compile time. Both restrictions are natural since using these types, hardware registers and buses are created.

The automata arrow will be used to hide state inside the arrow. Instead of using the transition function, a new function of type *Aut* is defined which maps input to an output and a new function of type *Aut*. The function of type *Aut* is an automata arrow. The function itself contains the state. The type of the state cannot be observed from the type *Aut*. Because of this, the state is not required as an argument to this function and is effectively hidden. Writing transition functions is natural when designing hardware, writing the transition functions directly as automata arrows is not. However, the correspondence between the two is clear and a mapping from a transition function to a automata arrow is defined using the lifting function `^^^` in listing 4.

```
(^^^) :: (s -> i -> (s,o)) -> s -> Aut i o
(^^^) f init = A applyS
      where applyS = \i -> let (s,o) = f init i
                           in (o, f ^^^ s)
```

Listing 4. Function to lift a transition function to an automata arrow

This function requires the transition function and an initial state as function arguments. The initial state is used when the system is reset, which for instance occurs after power on. Since the creation of a new arrow can not be implemented in actual hardware, the CλaSH compiler recognises the arrow, extracts the state and creates registers that represent the state.

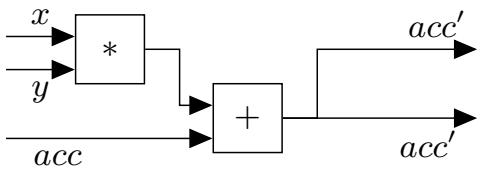


Fig. 5. Multiply Accumulate transition function

One of the simplest synchronous circuits is the multiply accumulate (MAC). The accumulator adds the product of the inputs to its state and uses the result as new state and sends it to the output. The corresponding transition function is visualised in figure 5. It is defined in listing 5.

```
mac acc (x,y) = (acc', acc')
 where
   acc' = acc + x * y
```

Listing 5. A multiply accumulate

When arrows are not used, the initial state is passed as an extra argument to CλaSH when the circuit is simulated or when the circuit is synthesised. When the circuit is lifted to an arrow, the initial state is an argument to the lifting function ^^^, which hides the state inside the function. To lift the function *mac* to the arrow macA using the initial state 0, the following definition is used:

```
macA = mac ^^^ 0
```

In CλaSH the do-notation to compose arrows is available, which was introduced in [12]. Using this do-notation, the arrows are automatically composed using first, >>> and pure. If the keyword *rec* is specified, *loop* is automatically used to compose arrows which require feedback. Section 5 shows an example which uses rec. In listing 6 it is shown how to define a circuit which contains two MACs, of which the results are added to produce an output.

```
macsum = proc (a,b,c,d) -> do                          1
           r1 <- mac ^^^ 0 <- (a,b)                    2
           r2 <- mac ^^^ 0 <- (c,d)                    3
           returnA         <- (r1+r2)                  4
```

Listing 6. Composing MAC components

In this example, the two components appear at lines 2-3. At the right, the inputs of the components are specified. When a component has multiple inputs, tuples are used. Between the two arrows, the transition function *mac* is shown, lifted to an arrow using the initial state 0. The output appears at the left side of the lines describing the components. The arrow *macsum* receives the inputs (a, b, c, d) at line 1 and returns $r1 + r2$ as output at line 4.

5 Reduction Circuit

The small example in the previous section does not yet show the full strength of CλaSH, nor why arrows are useful. A more elaborate example of a circuit is the streaming reduction circuit [6], which is introduced below.

When solving the matrix equation $Ax = b$ for a big sparse positive definite matrix A, often the conjugate gradient algorithm is used. The conjugate gradient algorithm is time consuming, while for some applications a fast response is required. One method to enable a fast execution of this algorithm is by implementing this algorithm in hardware, for instance using an FPGA. A kernel operation of the conjugate gradient algorithm is the sparse matrix vector multiplication (SM×V). When calculating a matrix vector multiplication, dot products can be used to calculate the elements of the result vector. For an SM×V, the number of multiplications and additions required for an element in the result vector depends on the number of non-zeros in the respective row of the matrix. In most FPGA implementations, a pipelined floating point adder is used to calculate the additions. Every clock cycle an addition can be scheduled, however it will take several clock cycles before the result is available because the adder is pipelined.

This causes some difficulties when summing a row of numbers, as is required for an SM×V. Take for instance a row of three values summed using a pipelined adder of 14 stages. An ideal situation would be using a single pipelined floating point adder, which receives the rows of floating point values at its input. Some logic is used to identify the rows and to make sure the correct values are added using minimal buffering, as buffers are relatively expensive. It is trivial to add the first two values. However, it will take 14 clock cycles before the result is available and can be added to the third value. Meanwhile, values of other rows are available for reduction. This illustrates the pipeline can be scheduled to reduce values of multiple rows simultaniously.

Various circuits which can sum rows of floating point values exists. These are called reduction circuits. Since these reduction circuits use pipelining and because of varying row lengths, it is hard to design a reduction circuit. Reduction circuits are an active area of research. Many other reduction circuits with

different properties are available [4–6, 10, 14]. Several designs rely on either a minimal or a maximal row length, where some require multiple adders, while others schedule a single floating point adder.

In [6] the streaming reduction circuit is introduced, together with an algorithm to schedule the pipeline and a proof to show that the defined buffer sizes are sufficient. An implementation of the streaming reduction circuit in both VHDL and CλaSH was compared in [2]. In the streaming reduction circuit, values appear sequentially at the input, one value at every clock cycle. The streaming reduction circuit uses a single floating point adder with α pipeline stages. This pipeline is denoted by $\mathcal{P}$. If two values of the same row are available at the input, they can be summed by inserting them into the pipeline. Since intermediate results which appear at the output of the pipeline have to be further reduced, they have to be temporarily stored. For the streaming reduction circuit, this is done in the partial result buffer (denoted by $\mathcal{R}$). This partial result buffer has an additional task: it will reorder the final results, such that the results are sent to the output of the reduction circuit in the order of their arrival. When two intermediate results are reduced, it is not possible to simultaneously reduce values which appear at the input. Therefore, the values at the input must be buffered and their order of arrival must be preserved. To this end, we use a FIFO input buffer (denoted by $\mathcal{I}$). To determine if either values from the input buffer, from the end of the pipeline and/or from the partial result buffer are used, five rules are checked. The rules can determine which values to use, i.e. the top two values from $\mathcal{I}$ (denoted as $\mathcal{I}_1$ and $\mathcal{I}_2$), the output of the the adder pipeline (denoted as $\mathcal{P}_\alpha$) or values from $\mathcal{R}$.

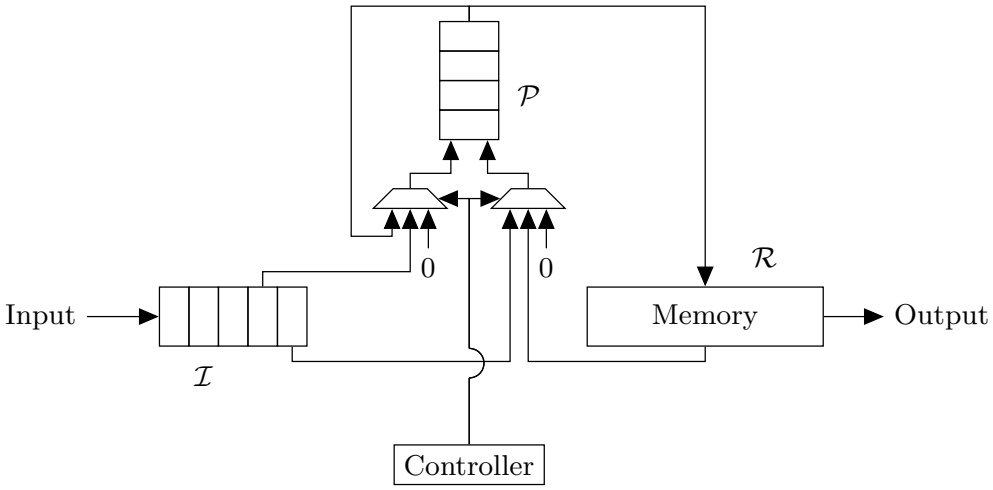
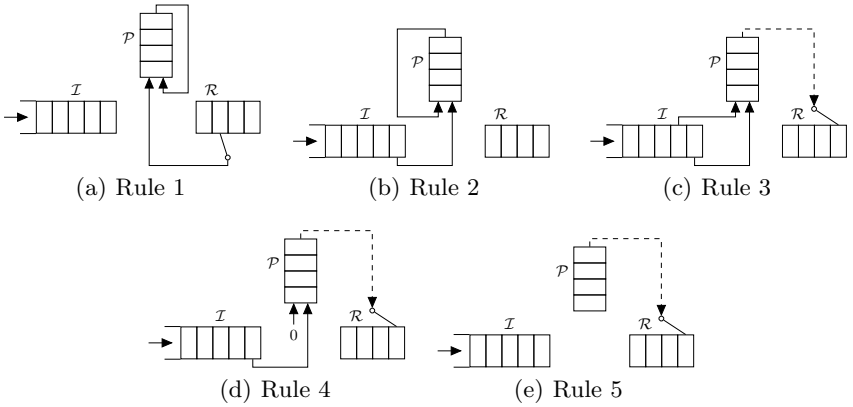


Fig. 6. Reduction circuit



(a) Rule 1 (b) Rule 2 (c) Rule 3

(d) Rule 4 (e) Rule 5

Fig. 7. Rules.

The five rules, in decending order of priority, are:

1. If there is a value available in $\mathcal{R}$ with the same row index as $\mathcal{P}_\alpha$, then this value from $\mathcal{R}$ enters the pipeline together with $\mathcal{P}_\alpha$.
2. If $\mathcal{I}_1$ has the same index as $\mathcal{P}_\alpha$, then $\mathcal{I}_1$ and $\mathcal{P}_\alpha$ enter the pipeline.
3. If there are at least two elements in $\mathcal{I}$, and $\mathcal{I}_1$ and $\mathcal{I}_2$ have the same index, then they enter the pipeline.
4. If there are at least two elements in $\mathcal{I}$, but $\mathcal{I}_1$ and $\mathcal{I}_2$ have different indexes, then $\mathcal{I}_1$ enters the pipeline together with the unit element of the operation dealt with by the pipeline (thus for example, 0 in case of addition, 1 in case of multiplication).
5. In case there are less than two elements available in $\mathcal{I}$, no elements enter the pipeline.

The rules are schematically shown in figure 7. The datapath of the reduction circuit is shown in figure 6. The components $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{P}$, together with the controller ($\mathcal{C}$) are shown in this figure. To identify rows within the reduction circuit, discriminators are used as identification. They are assigned to new rows which enter the reduction circuit and are released when a row is fully reduced and leaves the reduction circuit, after which the discriminator is reused. Discriminators require fewer bits than the row index.

Although figure 6 makes clear how data flows through the reduction circuit, it neglects the control signals. Figure 8 shows the entire circuit including control signals, as it was implemented in Haskell. The controller, denoted by $\mathcal{C}$, checks which rule has to be executed and routes all signals. The discriminators are assigned by $\mathcal{D}$.

To connect the components in the case the automata arrow is not used, the code from listing 7 was used.

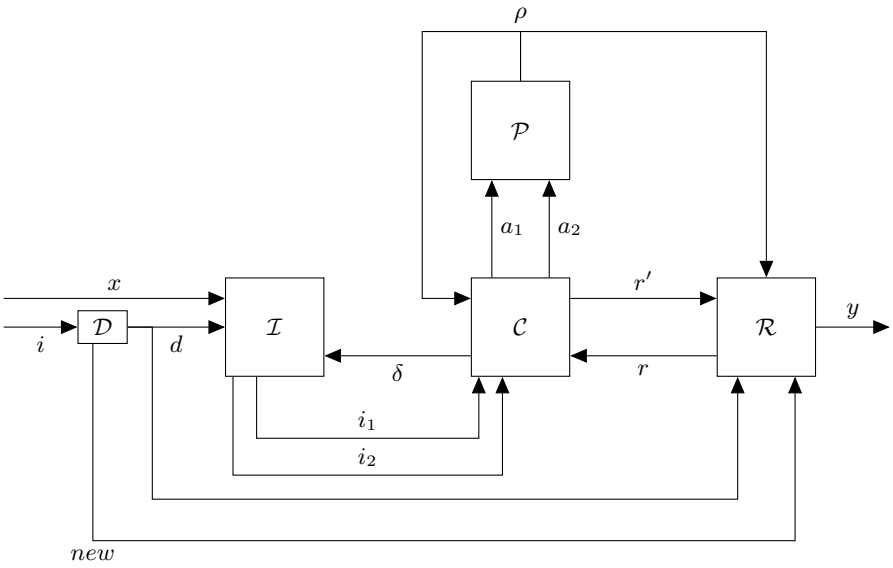


Fig. 8. Reduction circuit signals

```
reduction σ_red (x,i) = (σ'_red, y)                                  1
where                                                                2
  State Reduction {...} = σ_red                                      3
  (σ'_D, (new, d))      = D σ_D i                                    4
  (σ'_I, (i_1, i_2))    = I σ_I (x,d,δ)                              5
  (σ'_P, ρ)             = P σ_P (a_1, a_2)                           6
  (σ'_R, (r,y))         = R σ_R (new, d, ρ, r)                       7
  (σ'_C, (a_1, a_2, δ, r')) = C σ_C (i_1, i_2, ρ, r)                 8
  σ'_red                = State Reduction {σ_D = σ'_D, σ_I = σ'_I,   9
                                           σ_P = σ'_P, σ_R = σ'_R,   10
                                           σ_C = σ'_C}               11
```

Listing 7. Reduction circuit without arrows

As described before, the user has to manually extract the substates from the reduction circuit state, apply the transition functions and then repack it. In line 3, all states (denoted by σ_{red}) are unpacked. Lines 4-8 show how the components are instantiated and how the state is distributed to all components. The new states (denoted by σ'_x) are collected and packed at the lines 9-11. Besides risking possible errors and making the code hard to read, this requires work from the user which can be automated by CλaSH. When arrows are used, the function which connects the components looks as shown in listing 8.

```
proc (x,i) → do                                                          1
rec                                                                      2
  (new,d)          ← 𝒟^^^𝒟₀ ← i                                         3
  (i₁, i₂)         ← ℐ^^^ℐ₀  ← (x, d, δ)                                 4
  ρ                ← 𝒫^^^𝒫₀  ← (a₁, a₂)                                  5
  (r,y)            ← ℛ^^^ℛ₀ ← (new, d, ρ, r')                           6
  (a₁, a₂, δ, r')  ← 𝒞^^^𝒞₀  ← (i₁, i₂, ρ, r)                            7
returnA ← y                                                              8
```

Listing 8. Reduction circuit with arrows

Listing 8 shows that manually unpacking and packing the state is not required anymore when using arrows. The transition functions are now lifted using an initial state (denoted by the calligraphic letters with subscript zero) to arrows (lines 3-7). Only the composition of the components is shown, the state is only visible through the initial state. Since it is natural to define the initial state where the component is instantiated, this is a desired notation.

When arrows are used to implement the reduction circuit, it is clear an *ArrowLoop* is required. In line 2 of listing 8 this is enabled using *rec*. The component (or function) $\mathcal{I}$ requires a result from the controller, while the controller requires a result from $\mathcal{I}$, i.e. the functions depend on each other's results. In figure 8, this is shown using the signals δ, i_1 and i_2. These same signals are shown in listing 8. Because the result produced by the input buffer does not depend on the signal sent by the controller, Haskell's lazy evaluation will make sure this functional dependency will not be a problem.

6 Conclusions and Future Work

Functional languages are well suited for hardware design. The well known Mealy machine can be described using a function from input and state to output and a new state. This can be modelled in a functional language using a single function, called the transition function. Dealing with state when not using arrows is very cumbersome. The notation of arrows yields both a pleasant notation and a method to hide the state inside the arrow.

Our approach was tested by modelling the streaming reduction circuit, a nontrivial circuit, in Haskell and compile it using CλaSH. From this example, it is clear that it is possible to design nontrivial hardware using Haskell. Using arrows, packing and unpacking the substates can be avoided, which makes the code easier to read. ArrowLoop is used since loops are often required for digital hardware design. Since for such hardware, not all outputs depend directly on the input values. Because of this, lazy functional languages are desirable for hardware design.

Although using this extension it becomes easy to design and implement synchronous hardware using CλaSH, it is not yet possible to design hardware with multiple clock domains. Only synchronous hardware is supported by CλaSH. In the future, support for asynchronous hardware will be considered. Further research is required in these directions.

References

1. IEEE Standard 1076-2008 VHDL Language Reference Manual (2009)
2. Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., Gerards, M.: CλaSH: Structural descriptions of synchronous hardware using Haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools, Nice, France. IEEE Computer Society Press, Los Alamitos (September 2010), to be published
3. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98. pp. 174–184 (1998)
4. Bodnar, M.R., Durbano, J.P., Humphrey, J.R., Curt, P.F., Prather, D.W.: FPGA-based, floating-point reduction operations. In: MATH'06: Proceedings of the 10th WSEAS International Conference on APPLIED MATHEMATICS. pp. 5–9. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA (2006)
5. de Dinechin, F., Pasca, B., Cret, O., Tudoran, R.: An FPGA-specific approach to floating-point accumulation and sum-of-products. In: 2008 International Conference on Field-Programmable Technology. p. 33 (2008)
6. Gerards, M.E.T., Kuper, J., Kokkeler, A.B.J., Molenkamp, E.: Streaming reduction circuit. In: 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools. pp. 287–292. IEEE Computer Society Press, Los Alamitos (August 2009), http://eprints.eemcs.utwente.nl/17041/
7. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing kansas lava (11/2009 2009), http://www.ittc.ku.edu/csdl/fpg/sites/default/files/kansas-lava-ifl09.pdf, submitted to IFL'09
8. Hughes, J.: Programming with arrows. Advanced functional programming: 5th international school, AFP 2004, Tartu, Estonia, August 14-21, 2004: revised lectures p. 73 (2005)
9. Kuper, J., Baaij, C., Kooijman, M., Gerards, M.: Exercises in architecture specification using CλaSH. In: Forum on Specification, Verification and Design Languages, 2010. FDL 2010 (September 2010), to be published
10. Nagar, K.K., Zhang, Y., Bakos, J.D.: An integrated reduction technique for a double precision accumulator. In: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications - HPRCTA '09. pp. 11–18 (2009)
11. Paterson, R.: Arrows and computation. The Fun of Programming pp. 201–222 (2003)
12. Paterson, R.: A new notation for arrows. In: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming - ICFP '01. p. 229 (2001)
13. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23(1), 17 (2004)
14. Zhuo, L., Morris, G.R., Prasanna, V.K.: High-performance reduction circuits using deeply pipelined operators on FPGAs. IEEE Transactions on Parallel and Distributed Systems 18(10), 1377 (2007)

Implementing a non-strict purely Functional Language in JavaScript

Extended Abstract

Eddy Bruël[1], Jan Martin Jansen[2]

[1] Vrije Universiteit Amsterdam
[2] Faculty of Military Sciences,
Netherlands Defence Academy, Den Helder, the Netherlands
ejpbruel@gmail.com,jm.jansen.04@nlda.nl

Abstract. This paper describes an implementation of a non-strict purely functional language in JavaScript. This particular implementation is based on the translation of a high-level functional language such as Haskell or Clean into JavaScript via the intermediate functional language SAPL. The resulting code relies on the use of an evaluator function to emulate the non-strict semantics of these languages.

1 Introduction

Non-strict purely functional languages such as Haskell (see [7]) and Clean (see [8]) have many interesting properties. Nevertheless, their use on the client side of real world web-applications has so far been limited. That this is so it seems to be at least partly due to the lack of browser support for these languages: a significant part of software development today takes place in the browser, using JavaScript. Therefore, the availability of an implementation in the browser has the potential to significantly improve the applicability of non-strict purely functional languages in this area. Several implementations of non-strict purely functional languages in the browser already exist. However, these implementations are either based on the use of a Java applet for the implementation of the SAPL interpreter (see [3, 9, 2]) or the use of a dedicated plug-in for a Haskell like functional language (see [6]). Both require the loading of a plug-in, which is often infeasible in environments where the user has no control over the configuration of his/her system. As an alternative to this, one might consider the use of JavaScript. A JavaScript virtual machine is included with every major browser, so that installing a plug-in would no longer be required. Although traditionally perceived as being slower than Java, the introduction of JIT compilers for JavaScript has changed this picture significantly. Modern implementations of JavaScript, such as the V8 engine that ships with Googles Chrome browser, offer performance that rivals and sometimes even surpasses that of Java. JavaScript has been used as a target platform for the client-side implementation of other functional languages like HOP and LINKS (see [10, 5, 1]). But these are strict functional languages, which simplifies the translation to JavaScript considerably.

It is possible to translate programs written in Java into JavaScript using the Google Webtoolkit. In this way one can port the Java implementation of the SAPL interpreter to a JavaScript implementation. However, for this particular case, this is a naive solution. JavaScript offers many features not available in Java that provide opportunities for a more efficient implementation. One of these is the fact that JavaScript, unlike Java, is a dynamic language, allowing for solutions that are based on compilation rather than interpretation. This paper describes an implementation of the non-strict purely functional language SAPL in JavaScript, that is written with these features in mind. SAPL can be used as an intermediate language for high-level functional languages, like Haskell or Clean, into JavaScript. The representations that are the result of this translation rely on the use of an evaluator function *eval* to implement non-strict semantics. The eval function is used to reduce thunks (closures) on the moment they are needed. Using the dynamic compilation features of JavaScript closures are turned into JavaScript expressions that can be further executed.

2 The SAPL Language

The SAPL language has been designed with interpretation in mind: it features a minimal set of language constructs, required to encode higher-level constructs such as pattern matching and list comprehensions. This is akin to the use of a minimal kernel language for the compilation of high-level languages such as Haskell or Clean, with one important distinction: unlike these languages, SAPL provides no language constructs for algebraic data types and case analysis. Most intermediate formalisms provide these constructs in order to guaranteed an efficient implementation. In contrast, SAPL relies on the encoding of data types and patterns into ordinary functions, as described by Jansen (see [3]). Unlike previous encodings, the resulting representations have a complexity of $O(1)$, making them amenable to efficient interpretation. As a consequence of this, an implementation of SAPL can be kept extremely simple.

3 Translating SAPL into JavaScript

Because Java does not support runtime compilation, the Java implementation of SAPL is necessarily based on interpretation (using graph reduction). In contrast, JavaScript allows generated code to be evaluated at run-time through the use of the *eval* function. This suggests that a JavaScript implementation of SAPL may be based on compilation rather than interpretation, which is usually more efficient. The set of language constructs supported by JavaScript is a superset of that of SAPL. The only problem is caused by SAPL's non-strict evaluation semantics. JavaScript only has strict evaluation semantics, implying that non-strict semantics must be emulated somehow. The approach taken in this paper is based on the explicit representation of unevaluated expressions (or thunks) by means of arrays, combined with the use of an evaluator function which, for lack of a better name, we have named *eval* as well. Contrasting this with graph

reduction, lazy evaluation is achieved not by having the *eval* function pick the next redex to reduce, but by generating calls to *eval* at the appropriate places in the code.

The emulation of lazy evaluation can be kept efficient by making smart use of JavaScript's reference semantics for arrays. These allow thunks to be shared between expressions, a fact that can be exploited by updating thunks with their value the first time they are evaluated, so that the same expression never has to be evaluated twice. This optimization is based on ideas taken from the STG machine (see [4]), and requires the use of indirection nodes to avoid having to make a full scan of the program to replace all references to a thunk with its value. Future work includes the use of strictness annotations, to be added by the compiler during the generation of SAPL code to allow unnecessary calls to *eval* to be ommited from the code. These seem to be the main cause of overhead for the current implementation. Our hope is that the addition of this optimization will make the implementation efficient enough for use in a production system.

References

1. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.
2. J. Jansen, P. Koopman, and R. Plasmeijer. iEditors: extending iTask with interactive plug-ins. In S.-B. Scholz, editor, *Proceedings of the 20th'08*, pages 170–186, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.
3. J. M. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April 2006, The University of Nottingham*, volume 7 of *Trends in Functional Programming*. Intellect Publisher, 2006.
4. S. L. P. Jones and J. Salkild. The spineless tagless g-machine. 1989.
5. F. Loitsch and M. Serrano. Hop client-side compilation. In *Trends in Functional Programming, TFP 2007, New York*, pages 141–158. Interact, 2008.
6. E. Meijer, D. Leijen, and J. Hook. Client-Side Web Scripting with HaskellScript. In *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, Lecture Notes in Computer Science, pages 196–210. Springer, 1999.
7. S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.
8. R. Plasmeijer and M. v. Eekelen. *Concurrent Clean language report (version 2.0)*, Dec. 2001. http://clean.cs.ru.nl.
9. R. Plasmeijer, J. M. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Principles and Practice of Declarative Programming, Valencia, Spain, July 2008*, volume PPDP 08, 2008.
10. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, October 22-26 2006*, pages 975–985, 2006.

Concurrent Non-Deferred Reference Counting on the Microgrid: First Experiences*

Extended Abstract

Stephan Herhut[1] and Sven-Bodo Scholz[1]

University of Hertfordshire, U.K.
{s.a.herhut,s.scholz}@herts.ac.uk

1 Introduction

Functional programming languages are considered particularly suitable for concurrent execution due to their call-by-value semantics and side-effect free nature. However, when evaluated on a von Neumann architecture, side effects can ultimately not be avoided. Conceptual values from the purely functional world need to be manifested in memory, thus requiring some form of heap. Managing this heap brings back some of the challenges that imperative programming approaches face when it comes to concurrent execution. Fortunately, in the context of functional languages the added complexity of side-effects remains confined to the programming language's runtime system.

Commonly, heap management is implemented in the runtime system by means of deferred garbage collection. Under deferred garbage collection, the heap usage is monitored and, whenever a high-water mark is reached, program execution is stopped and no-longer referenced objects are removed from the heap. Following Amdahl's law [1], performing garbage collection sequentially would seriously impact scalability for a larger number of cores. Parallel garbage collectors [2, 3] alleviate this problem to a certain extent by performing the garbage collection itself in parallel. However, their scaling is limited by the inevitable locking of heap objects during collection.

Even though SaC [4] uses a different approach to heap management, the concurrent performance of heap management operations in SaC suffers from similar problems. The SaC runtime system employs non-deferred garbage collection using reference counting [5] for heap management. Here, a count of all live references to an object is stored alongside the object in the heap. Whenever a reference goes out of scope or an object is passed into a new scope, the counter is updated. Once the counter reaches zero, the object is freed.

Similar to the deferred case, updating the reference counter of a heap object requires exclusive access. Using a lock-based approach would incur prohibitive overheads and degrade the scalability of programs. In a data-parallel setting with only a limited number of worker threads, the use of locks can be avoided by creating local copies of the reference counter for each thread [6]. However, the involved administration costs and memory overhead increase linearly with the number of threads.

* This research is supported by EU research grant FP7/2007/215216 Apple-CORE.

On the Microgrid architecture [7] with its large number of cores and thousands of hardware threads, one reference counter per object per thread is not viable. Apart from the involved memory overheads, the administration costs quickly outgrow the actual workload. In this paper, we present an alternative approach to lock-free concurrent reference counting that can be efficiently implemented on the Microgrid using exclusive places and delegation.

An exclusive place on a Microgrid is a dedicated hardware resource that is guaranteed to execute only a single thread at a time. Thus, if a resource is only accessed by code running on an exclusive place, no locking is required. Delegation allows any thread running on any core to delegate the execution of code to an exclusive place. Such delegation requests can be performed synchronous, *i.e.*, the delegating thread waits for completion of the delegated task, or asynchronous, *i.e.*, the delegating thread directly continues executing.

The use of exclusive places and delegation for reference counting foots on two observations. The first insight is that, in SAC, reference counting operations of different threads can be arbitrarily interleaved, as long as each thread's reference counting operations remain in order. The intuition here is that for each thread in a data-parallel operation the reference count of global objects is invariant, *i.e.*, the reference counter is modified during execution but ultimately returns to its initial value. The second observation is that reference counting operations do not need to be executed before a thread can continue. For heap management purposes, it suffices if they are executed eventually such that no longer needed objects can be freed from the heap.

Thus, to perform reference counting operations concurrently, it suffices to delegate all reference counting operations to a single exclusive place. As delegation on the Microgrid guarantees that delegation requests from the same core are handled in order by a receiving exclusive place, in particular requests by the same thread will be handled in order. Thus, the resulting interleaving is safe.

Yet, using just delegation has the same drawbacks as using locks would have: If multiple threads issue a reference counting request, one thread will be blocked until the other thread was serviced. In contrast to locks, however, delegation allows to alleviate this effect: As the effect of the reference counting operation is not required for either thread to continue, reference counting requests can be executed asynchronously. To achieve this, we make use of the asynchronous delegation operation of the Microgrid.

Asynchronous delegation allows us to hide the latency of reference counting operations. However, the pressure on the exclusive place still remains. In particular, the number of simultaneously queued reference counting requests might eventually exceed the buffers available in hardware. To ease the pressure on queues and distribute the workload, we employ multiple exclusive places. If reference counting request were naïvely distributed, the resulting sequence of reference counting operations could be invalid. To prevent such an invalid interleaving of reference counting operations, we use a static mapping between heap regions and exclusive places. This ensures that all reference counting requests for a given heap-allocated object are serviced by the same exclusive place.

2 Outlook

We have implemented a first prototype of our concurrent non-deferred reference-counting scheme for the data-parallel runtime system of SaC. Currently, we are running first experiments to characterise the resource usage and dynamic behaviour of our approach. In particular, we are interested in the number of exclusive places that are required to efficiently service the reference counting operations of different classes of workloads. In the full version of this paper, we aim to provide a detailed description of our approach and hope to present first results that show the viability of our approach on the Microgrid.

References

1. Amdahl, G.M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Proceedings of AFIPS. Volume 30., Washington, DC, USA (1967) 483
2. Marlow, S., Harris, T., James, R.P., Peyton Jones, S.: Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In: ISMM '08: Proceedings of the 7th International Symposium on Memory Management, New York, NY, USA, ACM (2008) 11–20
3. Doligez, D., Leroy, X.: A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM (1993) 113–123
4. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13**(6) (2003) 1005–1059
5. Grelck, C., Trojahner, K.: Implicit Memory Management for SaC. In Grelck, C., Huch, F., eds.: Implementation and Application of Functional Languages, 16th International Workshop, IFL'04, University of Kiel, Institute of Computer Science and Applied Mathematics (2004) 335–348 Technical Report 0408.
6. Grelck, C., Scholz, S.B.: SAC: A Functional Array Language for Efficient Multithreaded Execution. International Journal of Parallel Programming **34**(4) (2006) 383–427
7. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. Journal of Systems Architecture **55**(3) (2009) 149–161

First Results from Auto-Parallelising SAC for GPGPUs (Extended Abstract)

Jing Guo[1], Jeyarajan Thiyagalingam[2], Sven-Bodo Scholz[1]

[1] Department of Computer Science
University of Hertfordshire, Hatfield, UK
{j.guo,s.scholz}@herts.ac.uk
[2] Oxford e-Research Centre
University of Oxford, Oxford, UK.
jeyarajan.thiyagalingam@oerc.ox.ac.uk

1 Introduction

The use of graphics processing units (GPUs) for general-purpose computing has become an indispensable route for obtaining high performance. The main driving forces behind this development are the favourable performance/price ratios as well as the favourable performance/power ratios of these architectures.

However, this comes for the price of programming difficulties. Although existing frameworks like CUDA or OpenCL [4] serve as excellent vehicles for programming this kind of hardware at low-level, typically rather substantial program modifications are required to effectively utilise these machines. Annotation-based tools such as *hi*CUDA [3] or OPENMP-variants [5] or compiler-assisted methods [7] offer some improvements but still require programmers to have some understanding of the underlying hardware as well as the willingness to rewrite parts of a given application to be effective.

In this extended abstract, we discuss the results of a different approach. We automatically generate CUDA code from a high-level, functional program specifications in SAC. Despite Matlab-like program specifications, we show that for a range of benchmarks, speedups between a factor of $5\times$ and $50\times$ can be achieved by means of a few GPU-specific extensions to an auto parallelising compiler: a small set of dedicated optimisations on top of a rather naïve compilation scheme suffice. We also demonstrate that further substantial improvements can be expected from more sophisticated program transformations.

This work builds on our earlier work [2] where we proposed a compilation scheme for mapping high-level SAC programs to CUDA-based devices and demonstrated that we can obtain substantial performance benefits without losing high-level abstractions. In this abstract, we discuss our findings when extending this to provide an auto-parallelising compiler framework for translating SAC programs to CUDA-based GPU executables. We assume that the readers are familiar with the CUDA programming model and the SAC programming language. Interested readers can refer to [1, 2, 6] for detailed information.

2 Compilation of SAC into CUDA

The compilation of SAC to CUDA focuses on translating individual data-parallel WITH-loop to equivalent kernel function. The transformation consists of the following three main steps:

- **Identifying Eligible WITH-loops**. Due to the inherent limitations in the CUDA architecture and programming model, namely the absence of stack and limitations towards creating nested threading mechanisms, only WITH-loops without function invocations will be parallelised at the outermost level.
- **Inserting Data Transfers**. Free array variables found inside CUDA-WITH-loops are host variables and should be mapped to the device memory. We extended the type system to distinguish between host and device type variables. With this notion, we introduced two dedicated instructions for performing mapping between two types of variables. In effect these are host-to-device and device-to-host memory transfers.
- **Creating CUDA Kernels**. After proper data transfer instructions are inserted, each WITH-loop partition is outlined as a kernel function and replaced by the corresponding invocation to it. This creates a thread hierarchy whose shape is determined by the shape of the generated array. Each thread computes a linear memory offset within the array from where the elements are fetched from or to where the computed results are stored.

3 Key Optimisations

In Section 2, we outlined the basic steps of translating SAC programs to CUDA. However, the baseline compilation is not sufficient to obtain noticeable speedups. In our framework we perform two key optimisations to improve the baseline performance:

- Reducing memory transfer (*memopt*);
- Expanding parallel region (*expar*).

As we outlined in [2] and in the literature, it can be found that it is performance critical for CUDA applications to have minimum data transfers between host and device. Since data in CUDA global memory is persistent across kernel invocations, a large portion of the data exchanges can actually be eliminated. We have identified two most common cases:

- **Retention of Invariant Arrays**. If several data transfers concerning the same host array, such as when the result of a CUDA-WITH-loop is consumed by a subsequent CUDA-WITH-loop without being modified in between or a number of CUDA-WITH-loops repeatedly accesses one or more read-only arrays, we perform only the first transfer and subsequent transfers can be eliminated.
- **Hoisting Data Transfers from *for* Loops**. If host arrays involved in data transfers nested inside *for* loops are not referenced in any instructions of the loop and they are either loop function arguments or return values (assuming *for* loops have been converted into tail-end recursive functions), the associated data transfers can be hoisted from the loops.

The promise of single assignment and the support from type system to differentiate host- and device-type arrays make this optimisation readily achievable. Implemented as two different sub-cycle optimisations in our compiler framework, they are repeatedly applied to the program until no more transfers can be eliminated. We have formalised a full compilation scheme to perform these two optimisation passes and it will be presented in the full version of the paper.

We have also discovered that the above optimisation is not capable of eliminating certain data transfer when programs contain interleaved parallel and sequential code. Based on this observation, we implemented a prototype optimisation, *Expand Parallel Region*, which contains two main transformations:

- Moving sequential instructions into CUDA-WITH-loops so that they are executed redundantly by all thread.
- Grouping sequential instructions into single-threaded CUDA kernels.

This optimisation enables *memopt* to remove the remaining data transfers. We show the effectiveness of this optimisation on two benchmarks in Section 4.

4 Performance Evaluation

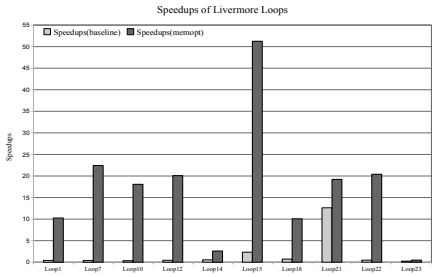


Fig. 1. Speedups of Livermore Loops.

We evaluated the effectiveness of our compiler framework by measuring the speedups of auto-parallelised SAC programs against their sequential implementations. The benchmark suite includes a subset of Livermore loops and a set of full-scale applications written in SAC. All benchmarks are executed on a system with an Nvidia Tesla C1060 graphic card and an Intel Xeon 5110 dual core processor. We use CUDA version 3.0 and enable *-O3* option for all compilations. The SAC version of the compiler is v1.00-beta.

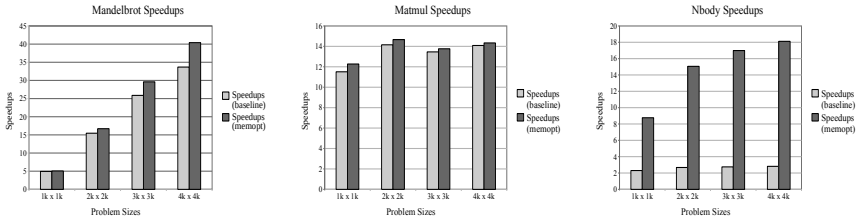


Fig. 2. Speedups of Mandelbrot, Matrix Multiplication and NBody Simulation.

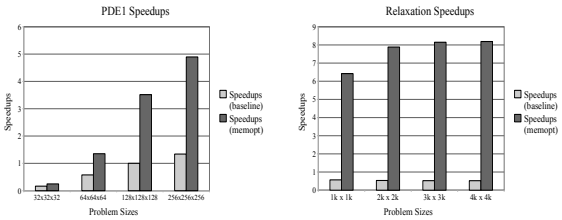


Fig. 3. Speedups of PDE1 and Relaxation applications.

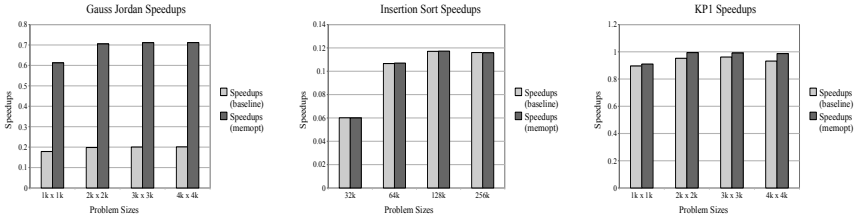


Fig. 4. Slowdowns of KP1, Gauss-Jordan and Insertion Sort.

As can be observed in the results, almost all Livermore loops show significant speedups (between $2.5\times$ and $50\times$) when memory transfer optimisation is enabled. The only exception is Loop23 where a slowdown is observed. Further investigation reveals that high level of sequential code coverage (*for* loop) prevents effective parallelisation. Loop14 achieves relatively low speedups ($2.5\times$) due to its indirect memory access pattern which hinders effective hardware memory access coalescing.

Most of the complete applications benefit from memory transfer optimisation and demonstrate noticeable speedups ranging from $5\times$ to $40\times$. Different performance improvements depend on the exploitable parallelism as well as the memory access patterns. However, three benchmarks, namely Gauss-Jordan elimination, KP1 and Insertion sort, show slowdowns instead of speedups.

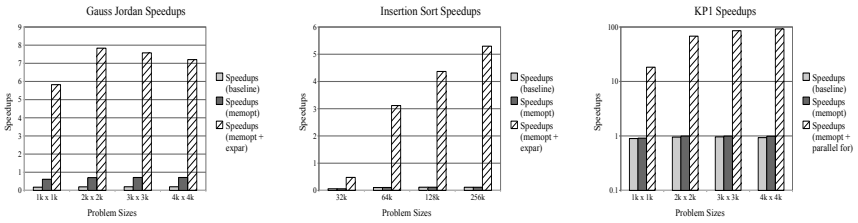


Fig. 5. Speedups of Gauss-Jordan, Insertion Sort and KP1 after expanding parallel region and parallelising *for* loops.

After applying the expand parallel region optimisation to Gauss-Jordan elimination and Insertion sort, speedups of $7.8\times$ and $5.3\times$ respectively are achieved (See Figure5). For KP1, we manually applied transformations to *for* loop nests, such as loop interchange and fusion, to parallelise the outermost loops. Performance is improved by two orders of magnitude.

5 Conclusions

In this abstract, we described an auto-parallelising compiler framework for translating high level SAC programs to equivalent CUDA based GPGPU programs. Our results show significant performance improvement with our optimisation passes combined and we yet to exploit the full potential of the SAC compiler to secure additional performance benefits. The work we presented here is part of a larger auto-parallelising compiler framework which requires our focus on improving the optimising capability of the framework. Among a number of issues to be addressed, some are: Formalisation of the compilation schemes, performing careful data access analysis and to re-target the code generation to accommodate other models such as *hi*CUDA.

References

1. David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.
2. Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In *Trends in Functional Programming (To Appear)*, volume 10. Intellect, 2010.
3. Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A High-Level Directive-Based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on GPGPUs*, pages 52–61, New York, USA, 2009. ACM.
4. Khronos Group. OpenCL 1.1, Last accessed July 16, 2010. `http://www.khronos.org/opencl/`.
5. Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110, New York, USA, 2009. ACM.
6. Sven-Bodo Scholz. Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.
7. Wolfe, Michael. Implementing the PGI Accelerator model. In *GPGPU '10: Proceedings of the 3rd Workshop on GPGPUs*, pages 43–50, New York, USA, 2010. ACM.

Improving your CASH flow: The *C*omputer *A*lgebra *SH*ell (Extended Abstract)

Christopher Brown[1], Hans-Wolfgang Loidl[2], Jost Berthold[3], and Kevin Hammond[1]

[1] School of Computer Science, University of St. Andrews, UK.
{chrisb,kh}@cs.st-andrews.ac.uk
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, UK.
hwloidl@macs.hw.ac.uk
[3] DIKU Department of Computer Science, University of Copenhagen, Denmark.
berthold@diku.dk

Abstract. Some important challenges in the field of symbolic computation —and functional programming— are the transparent access to complex, mathematical software, the exchange of data between independent systems with specialised tasks and the exploitation of modern parallel hardware. One attempt to solve this problem is **SymGrid-Par**, a system for exploiting parallel hardware in the context of computer algebra. Specifically, **SymGrid-Par** provides an easy-to-use platform for parallel computation that connects several underlying computer algebra systems, communicating through a standardised protocol for symbolic computation.
In this paper we describe a new component of **SymGrid-Par** known as **CASH**: the *C*omputer *A*lgebra *SH*ell. **CASH** is a system that allows direct access to **SymGrid-Par** via GHCi. **CASH** thus allows **Haskell** programmers to exploit high-performance parallel computations using a system designated for solving problems in computer algebra; whilst still maintaining the purity and express-ability offered by the **Haskell** environment. We demonstrate access to both sequential and parallel services of **SymGrid-Par**. For the latter we use parallel skeletons, implemented in the **Haskell** dialect of **Eden**; these skeletons are called from **CASH** but exploit a computational algebra system known as GAP to offload the mathematical complexity.

1 Introduction

In this paper we describe a system for orchestrating parallel computation, namely **SymGrid-Par**. In particular, however, we describe a new component of the **SymGrid-Par** system: **CASH**, the *C*omputer *A*lgebra *SH*ell. **CASH** is a system that allows the user to express symbolic computation in a full **Haskell** [1] environment, directly calling computer algebra systems via **SymGrid-Par** and also exploiting high-level parallel skeletons, allowing complex —and time consuming— algorithms in computer algebra to be solved efficiently.

The advantages of using **Haskell** as a client to a system like **SymGrid-Par** are twofold:

- Firstly, we can exploit environments already built for expressing complex computer algebra algorithms, such as the GAP system [2]. GAP has many bespoke data-types for expressing computation algebra, and also has a rich library of algorithms. It makes sense to exploit this environment, rather than re-implementing such a library and representation in **Haskell**. With **CASH** it is possible to marshall data-types between **Haskell** and GAP in a completely transparent way. The **CASH** user simply calls a service provided by a computer algebra system or by the **SymGrid-Par** middleware.
- Secondly, **SymGrid-Par** has a middleware layer exposing a collection of parallel skeletons, callable from the **CASH** shell. These parallel skeletons are higher-order, and are called just like higher-order functions in **Haskell**. Using these skeletons, it is possible to use **SymGrid-Par** to exploit high parallel performance both in general algorithms implemented in **Haskell**, and also in computational algebra. Some examples of the higher-order skeletons available to **CASH** are: `parMap`, `parZipWith`, `parFold` and `parMapFold`. There is also a number of domain-specific skeletons for solving computational algebra problems, such as `Orbit` and `SumEuler`. These skeletons are currently also implemented in **Haskell**, and are installed in **SymGrid-Par** as available services.

The main contributions of this paper are therefore as follows:

- we design, implement and describe **CASH**, a **Haskell** shell for calling (potentially parallel) computer algebra algorithms;
- we demonstrate some uses of **CASH**, and show how it is possible to define bespoke parallel skeletons in **Haskell**, made available as **SymGrid-Par** services, directly callable from the **CASH** shell;
- we show how it is possible to use a combination of **CASH** and **SymGrid-Par** to exploit systems that manipulate computational algebra (namely GAP) in order to define domain-specific skeletons.

2 SymGrid-Par

In this section we aim to give a brief overview of **SymGrid-Par** [3]. **SymGrid-Par** orchestrates symbolic components into a Grid-enabled application. Each component executes within an instance of a Grid-enabled engine, which can be geographically distributed to form a wide-area computational grid, built as a loosely-coupled collection of Grid-enabled clusters. Components communicate via the standard OpenMath protocol [5].

SymGrid-Par has been designed to achieve a high degree of flexibility in constructing a platform for high-performance, distributed symbolic computation. The most visible aspect of this flexibility is the possibility to connect different computer algebra systems (CAS) to co-operate in the execution of a program.

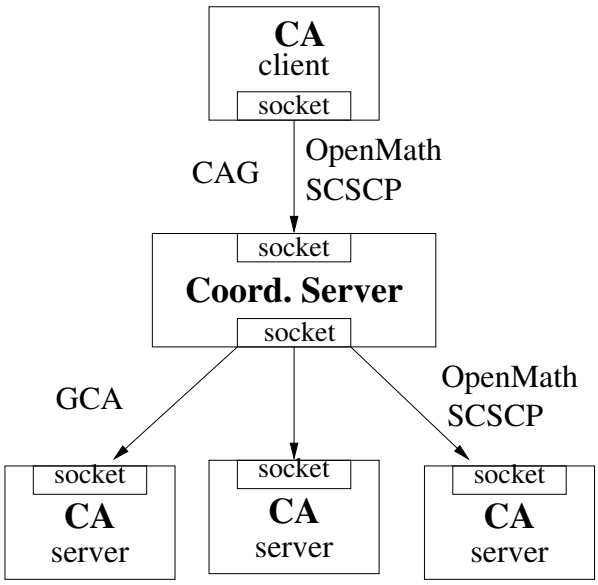


Fig. 1. Current **SymGrid-Par** Architecture

This requires a common data and communication protocol, which has been defined earlier in the SCIEnce project in the form of the SCSCP protocol [6].

The system structure of **SymGrid-Par** is shown in Figure 1, and is divided into three main components:

- **The Client**. The end user works in his/her own familiar programming environment, avoiding having to relearn a new computer algebra system, or a new language to exploit parallelism. In this paper, we concentrate on a new client component known as **CASH**, that allows ***Haskell*** *users* to exploit powerful parallel computer algebra in a simple and transparent way.
- **The Coordination Server**. This middleware provides parallelised services and parallel skeletons to the client. The client may invoke these skeletons as standard higher-order functions. The Coordination Server then delegates work (usually calls to expensive computational algebra routines) to the Computation Server. Currently the Coordination Server is implemented in **Haskell**, allowing the user to exploit polymorphism, purity and higher-order functions for effective implementation of high-performance parallelism.
- **The Computation Server**. This component is typically a system dedicated to computational algebra. Currently, we use GAP [2] as the symbolic computation engine.

One important aspect in the design of **SymGrid-Par** is the use of standards when describing the interaction between components of the above software ar-

	Bespoke interface	***SCSCP interface***
Access Layer:	Grid	Grid
Service Layer:	Grid Service	Grid Service
Application Layer:	Skeletons/Strategies	Skeletons
Coordination Layer:	parallel **Haskell** (GpH)	parallel **Haskell** (Eden)
Communication Layer:	Bespoke	SCSCP
Data Layer:	Strings	OpenMath
Connection Layer:	Pipes	Sockets

Fig. 2. Layers (left) in the **SymGrid-Par** Design: bespoke (middle) and SCSCP-based (right)

chitecture. Additionally to building on established standards, we have defined a protocol for the interaction between computer algebra systems, the SCSCP protocol [6] discussed below.

Figure 2 depicts the **SymGrid-Par** design as a stack of layers (left) of increasing levels of abstraction. The middle stack presents an early version of **SymGrid-Par** [7], based on a bespoke interface between **Haskell** and the underlying CAS. The right stack describes the latest version of **SymGrid-Par** [4], based on established standards and supporting a distributed collection of servers. The realisation of the levels is specific to the application domain of parallel, symbolic computation and provides at the highest level a Grid-enabled interface to access it in a location transparent way.

Connection Layer: The connection layer defines the software level of connecting physically distributed machines. As a flexible, standardised way of establishing and managing connections between the **SymGrid-Par** middleware and the computer algebra systems, *sockets* have been used.

Data Layer: The data layer defines the data format for items to be exchanged. Here we build on the preceding standardisation process in the community and use the OpenMath standard. This is an XML [8] based data format designed specifically for representing symbolic and mathematical objects. More specifically, we use a specialised content dictionary within this general framework, which is defined together with the SCSCP protocol.

Communication Layer: The communication layer defines a protocol of messages that are exchanged in realising communication between two components of the system. The standard developed earlier in this project, specifying this layer, is the *Symbolic Computation Software Composability Protocol, SCSCP* [9].

Coordination Layer: The coordination layer specifies in which form parallelism is specified and managed. Several approaches, with varying levels of abstraction, are possible to realise this layer. We are using parallel implementations of the functional programming language **Haskell** as a high-level parallel programming model.

Application Layer: The application layer presents a high-level API to the application programmer, for realising parallel symbolic applications. It should abstract over the low level details of the orchestration on the level below and should be powerful enough to specify the parallel execution. As a domain specific instance of this layer, tailored to the characteristics of symbolic computation, we have defined an interface that is based on the concept of *algorithmic skeletons* [10]. The concrete interfaces, namely CGA and GCA between parallel system and computer algebra system, are defined in [3].

Service Layer: The service layer defines in which way an application, or service, is made available on the web, in the form of a Grid-enabled application. This level is part of the SymGrid-Services component of the overall infrastructure.

Access Layer: Finally, the access layer defines in what form the end-user accesses a concrete service. Again, this level is part of the SymGrid-Services component.

3 Basic SCSCP Interface

In essence, the **CASH** is GHCi, the interactive version of the Glasgow Haskell Compiler [11], with a library of Haskell functions implementing the SCSCP standard. We assume familiarity with GHCi and focus on the implementation of the interface in this section.

The interface between a client and a server is defined by the SCSCP communication protocol [6], which itself builds on the existing OpenMath standard for data representation. In this section we outline the **Haskell**-side implementation of the SCSCP interface, which is used by the **CASH** client to drive the computation and by the Coordination Server to receive calls to possibly parallel services.

The OpenMath data format is an XML-based representation of mathematical objects such as polynomials or permutations. To realise the conversion between **Haskell** data structures and OpenMath objects, the HaXml package [12] is used. While this package handles all low-level aspects of, the visible interface to user is simply a class `OMData` with conversion routines in both directions:

```
class OMData a where
    toOM   :: a -> OMObj
    fromOM :: OMObj -> a
```

The SCSCP communication protocol defines the message exchange between client and server-processes. Its main functionality is the (remote) call of a service provided by the SCSCP-server and the delivery of the result to the SCSCP-client. Additionally, attributes may be provided, controlling the maximum amount of time or heap allowed for the execution of the service. In order to enable service detection, the protocol also specifies how to retrieve a list of all available services and their types. Here we focus on the main functionality of calling a remote service. The user-level interface is provided by the function `callSCSCP`, which

takes a service name and a list of OpenMath objects as arguments and issues an SCSCP call to the server, which must be initialised separately. To simplify this interface, a family of functions `call1`, `call2` etc is defined, which hide the (un)marshalling of the data-structures.

```
-- basic interface to SCSCP service: needs service-name and arg. list
callSCSCP :: CAName -> [OMObj] -> OMObj

-- specialised calls, hiding (un-)marshalling of data
call1 :: (OMData a, OMData b) =>
         CAName -> a -> b
call2 :: (OMData a, OMData b, OMData c) =>
         CAName -> a -> b -> c
```

4 An Example of CASH Usage

As a simple example we implement a generic greatest common divisor (GCD) function by calling existing factorisation functions on the computer algebra side (in this case GAP) and then combining the results on the **Haskell** side. Although this operation is in itself fairly trivial, it highlights several important design features of **SymGrid-Par**. First of all we make use of **Haskell**'s overloading mechanism to develop a generic GCD implementation. All the compute intensive parts of the code are delegated to SCSCP services that are provided by a computer algebra system. Notably, we don't restrict the program to one particular system, and it is possible to combine different systems, exploiting their strengths in implementing particular application domains. Finally, by expressing the top-level algorithm in **Haskell**, we can make use of its support for high-level parallel programming.

The structure of the `myGcd` algorithm contains two (potential) SCSCP calls to perform factorisations of the inputs `x` and `y`. Then, a bag-intersection is performed on the **Haskell** side, to identify all common factors. Finally, the GCD is computed by folding the generic multiplication operation, potentially implemented as an SCSCP-call (if it is non-empty) over the list of common factors.

```
-- intersection on multi-sets (bags)
bagInter :: (Eq a) => [a] -> [a] -> [a]
bagInter [] _ = []
bagInter (x:xs) ys | elem x ys = x:(bagInter xs (delete x ys))
                   | otherwise = bagInter xs ys

-- generic GCD computation
myGcd :: (Num a, Factorisable a) => a -> a -> a
myGcd x y = let
              xs = factors x
              ys = factors y
              zs = xs `bagInter` ys
            in
            if null zs then fromInteger 1 else foldl1 (*) zs
```

We now demonstrate this algorithm on polynomials. The interface to the relevant operations on polynomials is very simple. Since all operations on polynomials are done on the computer algebra side, its data type is just a wrapper around the `OMObj` data type. In general it would be possible to extract structural information about the polynomial by parsing the XML data structure. This might be useful in general: for example to make decisions on the algorithm to be used based on the size of polynomial. For our example, however, this is not necessary. The basic operations over the polynomials are implemented via SCSCP calls to the corresponding SCSCP services, using the `call2` wrapper function, which implicitly applies `toOM` and `fromOM` to perform the marshalling.

```
newtype Polynomial a = P OMObj

instance Show a => Show (Polynomial a) where
  show (P pOM) = fromOM pOM

instance Eq a => Eq (Polynomial a) where
  (P p1) == (P p2) = p1 == p2

instance (Eq a, Show a) => Num (Polynomial a) where
  (*) = call2 scscp_WS_ProdPoly
  (+) = call2 scscp_WS_SumPoly
  fromInteger n = fromOM $ toOM ("0*x_1+"++(show n))

instance OMData (Polynomial a) where
  toOM (P pOM) = pOM
  fromOM pOM = P pOM
```

The `myGcd` algorithm builds on a class `Factorisable`, containing one function `factors` that returns all factors to a given argument. We create instantiations for (fixed precision) integers and for polynomials, invoking the corresponding service on the computer algebra side.

```
class Factorisable a where
  factors :: a -> [a]

instance Factorisable Int where
  factors = call1 scscp_WS_FactorsInt

instance Factorisable (Polynomial a) where
  factors = call1 scscp_WS_Factors
```

Below we give a trace of using **CASH** on this GCD implementation. We first have to initialise the SCSCP server, specifying the host and the port number to which to connect. The first example uses the GCD algorithm on integers, performing an SCSCP for factorising inputs, but combining the results entirely on **Haskell** side. We can check the result against the `gcd` function of the **Haskell** prelude. In general, we can use existing **Haskell**-side tools, such as

QuickCheck, to confirm results either against existing **Haskell**-side implementations, or against the result produced by another computer algebra system. In the next section we define two simple input polynomials. The first and second polynomial contains three factors; both of which also occur in the first polynomial. In this example the result of `myGcd p1 p2` is `p2`, calculated as the product of its factors. We can easily check this on **Haskell** side. Finally, we generate some random polynomials and test basic arithmetic and our GCD algorithm.

```
*Cash> -- initialisation
*Cash> initServer (server "localhost" (Just 26133))
*Cash> -- integer example
*Cash> myGcd (12::Int) 18
6
*Cash> gcd 12 18
6

*Cash> -- define input polynomials
*Cash> let p1 = polyFromString "x_1^3-x_1"
*Cash> factors p1
[x_1-1,x_1,x_1+1]
*Cash> let p2 = polyFromString "x_1^2+x_1"
*Cash> p2
x_1^2+x_1
*Cash> factors p2
[x_1,x_1+1]
*Cash> myGcd p1 p2
x_1^2+x_1
*Cash> p2 == myGcd p1 p2
True

*Cash> -- generate random polynomials
*Cash> let p4 = mkRandPoly 4
*Cash> p4
-2*x_1^3+x_1^2+x_1-2
*Cash> let p5 = mkRandPoly 4
*Cash> p5
-x_1^3+3*x_1^2-x_1-2

*Cash> -- test basic arithmetic
*Cash> p4*p5
2*x_1^6-7*x_1^5+4*x_1^4+8*x_1^3-9*x_1^2+4

*Cash> -- again, factorisation
*Cash> factors p4
[-2*x_1-2,x_1^2-3/2*x_1+1]
*Cash> factors p5
[-x_1+2,x_1^2-x_1-1]
*Cash> myGcd p4 p5
1
```

This example has shown how the **CASH** client provides easy access to a (sequential) computer algebra server to perform complex operations. In the infrastructure discussed in Section 2 the client interacts with a Coordination Server, which is itself also implemented in **Haskell**. This Coordination Server provides several parallel services, as well as a range of parallel skeletons, such as `parMap`, `parZipWith`, `parFold` and `parMapFold`. By using these skeletons with concrete services the end user can easily create parallel applications without having to be an expert in parallel programming.

In the following example, the user first calls a parallelised computer algebra algorithm computing the sum of the Euler totient function over a range of integers, using a given blocksize to improve parallel performance. This service is provided by the Coordination Server and internally uses Eden primitives to achieve parallel execution. To the end user, however, this is just another service that can be called directly from the **CASH**. The structure of this computation is in essence a fold, of integer addition (`WS_Plus`), over a map, of the Euler totient function (`WS_Phi`). The **SymGrid-Par** middleware provides a range of general purpose skeletons, including `parMapFold` that suites this application. Therefore, the end user can easily construct a parallel application by calling this skeleton with the appropriate services. Of course, on the **CASH** level it is easy to compose purely functional operations provided by the underlying compute engine. Thus, the final example performs only the map in parallel on the Coordination Server, and computes the final sum directly in the **CASH**.

```
*Cash> -- this is a session communicating with the Coordination Server
*Cash> initServer (server "localhost" (Just 12321))

*Cash> -- parallel computation of sumEuler function; args: range, blocksize
*Cash> :{
*Cash| let { z :: Int
*Cash|     ; z = fromOM $ callSCSCP scscp_CS_SumEuler (map toOM [100::Int, 10::Int]) }
*Cash| :}
*Cash> z
3044

*Cash> -- same computation, in parallel for all list elements, using a skeleton
*Cash> let { xs = map toOM [(1::Int)..100] }
*Cash> :{
*Cash| let { z :: Int
*Cash|     ; z = parMapFold "WS_Phi" "WS_Plus" (0::Int) xs }
*Cash| :}
*Cash> z
3044

*Cash> -- as above, but performing sum sequentially on Haskell side
*Cash> let { zs :: [Int]; zs = parMap "WS_Phi" xs }
*Cash> sum zs
3044
```

This is only a very simple example of how to use and construct parallel computer algebra algorithms by calling the appropriate service of the **SymGrid-Par** middleware. However, it shows the power of the system, being able to access any computer algebra operation provided as a service by the underlying compute engine, its ease of expressing parallelism and its flexibility, by mixing operations performed on both sides. Additionally to the general purpose skeletons mentioned above, several domain-specific skeletons, for example a parallel Orbit skeleton [13] are also available. A discussion of implementing parallel computer algebra services in the Coordination Server of the **SymGrid-Par** architecture is given in [14].

5 Conclusions

SymGrid-Par with the **CASH** client as a front-end combines the best of two worlds: symbolic computation and functional programming. A rich repertoire of complex library functions over mathematical data structures becomes available to the **Haskell** programmer by providing a standard interface to calling computer algebra systems. The high-level language features of a functional language can be used on top level when composing calls to the library functions. In particular, overloading and higher-order functions enable the implementation of generic algorithms and promote code-reuse. Parallelism can be easily expressed on the **Haskell** level, and this is used in the Coordination Server component of **SymGrid-Par**. As an added benefit for the user of **CASH**, all parallelism constructs become directly available on the client side, when using this front end. Therefore, the end user has the choice between using predefined parallel skeletons that are provided by the Coordination Server, or directly using the parallelism constructs provided by GpH [15] or Eden [16].

There are many avenues of future work, both in extending **SymGrid-Par** and **CASH** itself. To simplify the interface for the **CASH** user we want to develop **Haskell** class hierarchies reflecting important computer algebra datatypes with their operations. Here we plan to build on the concepts for overloading developed for computer algebra systems, capturing the challenging constraints imposed by this application domain, and adapt them to a language with static typing and an advanced type system. We plan to look at concrete computer algebra algorithms and explore suitable granularities of calls to computer algebra operations. We expect these to be coarse grained in general to compensate for the overhead imposed by the system. However, moving code to the **Haskell** level might enable advanced optimisations usually not supported by computer algebra systems. On a related issue, we would like to combine several computer algebra systems, exploiting their individual strengths on certain algorithms, to develop a complex overall algorithm. The **CASH** would be a very useful tool in combining different systems in such a way. To enhance usability of **SymGrid-Par**, we would like to extend the library of parallel skeletons provided to the **CASH** user by implementing more domain-specific skeletons for symbolic computation. Our longer term vision is a system that combines multi-core machines with wide-

area, Grid-style architectures, which are the focus here. To achieve that, we plan to support, in future releases of **SymGrid-Par**, GpH's model of parallelism on multi-cores and combine it with Eden's model on networks, exploiting the strengths of the systems on different architectures.

Acknowledgements

This work has been generously supported by the EU Framework VI SCIEnce project (Symbolic Computation Infrastructure in Europe, RII3-CT-2005-026133), and by the UK's Engineering and Physical Sciences Research Council (HPC-GAP: High Performance Computational Algebra and Discrete Mathematics, EP/G 055181).

References

1. Peyton Jones, S., Hammond, K.: Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press (December 2003)
2. The GAP Group: GAP – Groups, Algorithms, and Programming, Version 4.4.12. (2008) `http://www.gap-system.org`.
3. Zain, A., Hammond, K., Trinder, P., Linton, S., Loidl, H.W., Costanti, M.: SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In: ICCS '07: Proceedings of the 7th International Conference on Computational Science, Part II, Berlin, Heidelberg, Springer-Verlag (2007) 617–624
4. Kevin Hammond, Hans-Wolfgang Loidl, *et al.*: Report on Patterns of Symbolic Computation for the Grid (May 2010) Deliverable D5.8 of the SCIEnce project.
5. Abbott, J., Díaz, A., Sutor, R.S.: A report on openmath: a protocol for the exchange of mathematical information. SIGSAM Bull. **30**(1) (1996) 21–24
6. Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D.: Symbolic Computation Software Composability. Lecture Notes in Computer Science **5144** (2008) 285–295
7. A. Al Zain and K. Hammond and P. Trinder and S. Linton and H-W. Loidl and M. Costantini: SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In: ICCS'07 — Intl. Conference on Computer Science, Beijing, China (2007)
8. Mller, A., Schwartzbach, M.I.: An Introduction to Xml And Web Technologies. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
9. Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D.: Symbolic Computation Software Composability Protocol (SCSCP) specification. `http://www.symbolic-computation.org/scscp` (2009) Version 1.3.
10. Cole, M.: Algorithmic Skeletons: Structure Management of Parallel Computations. In: Research Monographs in Parallel and Distributed Computing, MIT Press (1989)
11. GHC Team: GHC — The Glasgow Haskell Compiler. Web page `http://www.haskell.org/ghc/`.
12. Wallace, M., Runciman, C.: Haskell and XML: Generic Combinators or Type-Based Translation? In: ICFP'99 — International Conference on Functional Programming, Paris, France (September 1999)

13. Brown, C., Hammond, K.: Ever-Decreasing Circles: a Skeleton for Parallel Orbit Calculations in Eden. In: TFP10 — Symposium on Trends in Functional Programming, Oklahoma (May 2010)
14. SCIEnce Team: SymGrid-Par: Parallel Orchestration of Symbolic Computation Systems. In: ISSAC10 — Intl. Symp. on Symbolic and Algebraic Computation, Munich (July 2010) Software Demonstration.
15. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. J. of Functional Programming **8**(1) (January 1998) 23–60
16. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming **15**(3) (2005) 431–475

mHume for parallel FPGA

Abdallah Al Zain[1], Greg Michaelson[2], and Wim Vanderbauwhede[3]

[1] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
A.D.AlZain@hw.ac.uk
[2] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
G.Michaelson@hw.ac.uk
[3] University of Glasgow, Glasgow, Scotland, G12 8QQ, UK
wim@dcs.gla.ac.uk

Abstract. The formally motivated Hume language, based on the coordination of concurrent automata performing functional computations, was designed to support the development of systems requiring strong assurance that resource bounds are met. mHume is an experimental subset oriented to exploration of efficient heterogeneous multi-processors implementations. In this paper, the deployment of mHume on the FPGA MicroBlaze architecture is discussed. Preliminary results suggest very fast performance and good scalability compared with stock multi-core processors.

Keywords: Hume, FPGA.

1 Introduction

While the number of cores on stock CPUs continues to grow steadily, in accordance with Moore's Law, their internal configurations are strongly constrained by the underlying CPU architecture. In contrast, contemporary FPGAs themselves offer immediate opportunities for very large numbers of processing elements, with highly flexible connectivity and excellent scalability, albeit with poorer performance than conventional CPUs. Thus, there is considerable interest in synergistic exploitation of concurrency in *heterogeneous architectures*, typically comprising multi-core processors with SIMD acceleration augmented with an FPGA.

In the EPSRC supported Islay project, we are exploring alternative routes to deploying the functionally-flavoured, concurrent language Hume [7] on such architectures. Hume is based on autonomous *boxes*, linked point to point, and to the environment, by single buffered *wires*. Thus, we are interested in directly realising Hume boxes as loci of concurrency.

Hume's design is oriented to systems where there is need for strong assurance that bounds on resources such as time and space are met. This is enabled through Hume's rigorous formal foundations, reflected in a unified tool chain built around well-characterised Hume Abstract Machine(HAM), enabling tight articulaltion of program analysis and implementation.

We have explored the implementation of Hume on both multi-core and FPGA architectures via the HAM interpreter (hami) [10, 1], which offers consistent speedup on regular programs. However, FPGA performance is markedly poorer than on CPUs. We have also explored the direct implementation of Hume on FPGAs, via C generated from HAM through the standard tool chain, but this offers poor flexibility for multi-processor exploitation.

Thus, in our current research, we are investigating the direct generation of C from Hume itself, through a lightweight compiler for the *mHume* subset, which generates code that facilitates multi-processor implementations. In this paper, we present the first results of our experiments, which show both consistent speedup and very decent performance of multi-box programs on FPGA multi-processors.

2 mHume overview

mHume is a proper subset of Hume, itself based on an automata-like *coordination* layer, for describing boxes and wires, and a functional *expression* layer, for describing pattern matching and processing within boxes, both sharing a rich polymorphic type system and *definition* layer, for describing types, structures and functions. mHume retains an expression layer restricted to integer arithmetic but the full core coordination layer, offering considerable potential for richer expressions in future.

To get some flavour of mHume, consider the following program which generates squares of successive integers, illustrated in Figure 1.

Box `inc` generates the integers, and box `square` generates the squares by repeated adding and counting, Figure 2.

`inc` wires output `n'` (next integer) to input `n`, and output `r` to `square`'s input `i`. On each execution cycle, `inc` matches the next integer, outputs it to `square` via `r`, and also increments it and sends it back to itself via `n'`.

`square` wires outputs `s'` (sum), `c'` (count) and `v'` (current value) back to the corresponding inputs `s`, `c` and `v`. It also wires input `i` to `inc`'s output `r`, and its output `o` to the stream `output` associated with standard output `std_out`.

Note that for Hume pattern matching, there must be appropriate values on all inputs for a match to succeed, except for the pattern `*` which ignores the corresponding input. Thus, `square` is composed of three matches:

1. `(*,s,0,v) -> (s,*,*,*)`: ignoring input from `inc`, if the count is `0` then output the sum;
2. `(*,s,c,v) -> (*,s+v,c-1,v)`: ignoring input from `inc`, add the current value to the sum, decrement the count and retain the current value;
3. `(i,*,*,*) -> (*,0,i,i)`: for the next input from `inc`, set the sum to `0`, and the count and current value to that input.

The mHume syntax is summarised in Figure 3.

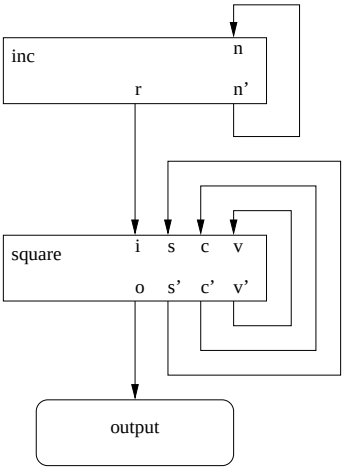


Fig. 1. Square program.

```
type integer = int 64;

box inc
in (n::integer)
out (r::integer,n'::integer)
match (n) -> (n,n+1);

box square
in (i::integer,s::integer,
      c::integer,v::integer)
out (o'::integer,s'::integer,
      c'::integer,v'::integer)
match
 (*,s,0,v) -> (s,*,*,*) |
 (*,s,c,v) -> (*,s+v,c-1,v) |
 (i,*,*,*) -> (*,0,i,i);

stream output to "std_out";

wire inc (inc.n' initially 0)
          (square.i,inc.n);

wire square
(inc.r,square.s',square.c',square.v')
(output,square.s,square.c,square.v);
```

Fig. 2. Square program Code

3 mHume execution model and compiler

mHume follows closely the Hume two stage execution model. A program executes repeatedly in cycles during which boxes are alternately in $READY$ (awaiting input) or $BLOCKED\_OUTPUT$ (output pending)states. Initially, all boxes are $READY$.

On each cycle, at the *match stage*, only a $READY$ box may attempt to match inputs. If it succeeds, it generates pending outputs and becomes $BLOCKED\_OUTPUT$. At this stage, no inputs are consumed or outputs asserted. Then, at the *super step* stage, only a $BLOCKED\_OUTPUT$ box may attempt to assert pending outputs. Provided all the previous outputs have been consumed, it consumes its inputs, asserts its outputs and becomes $READY$. Note that this input/output behaviour is very similar to AlgolW's "call by value result".

mHume is compiled to C by a very simple multi-pass, syntax directed processor, written in Haskell, and closely aligned to the execution model. We consciously generate a very restricted subset of C (declaration, assignment, condition, jump), to simplify potential cost analysis in future.

Compilation proceeds as follows:

program → [*component*`;`]$^+$
component → *box* | *wire* | *stream* | *typedef*
box → `box` *id* `in (`*links*`) out (`*links*`) match` *matches*
links → *link* [`,` *links*]*
link → *var*`::`*type*
matches → *match* [`|` *matches*]*
match → *pattern* `->` *exps*
pattern → *patt* [`,` *pattern*]*
patt → *int* | *var* | $*$
exps → *exp* [`,` *exps*]*
exp → *int* | *var* | `(` *exp* `)` | *exp op exp* | $*$
op → `+` | `-` | `*` | `/`
wire → `wire` *id* `(`*inwires*`) (`*outwires*`)`
inwires → *inwire*[`,` *inwire*]*
inwire → *id*[.*var*[`initially` *int*]]
outwires → *outwire*[`,` *outwire*]*
outwire → *id*[.*var*]
stream → `stream` *id* `{ from | to } "` *path* `"`
typedef → `type` *var* `=` *type*
type → *var* | `int` *int*

Fig. 3. mHume syntax.

1. for each box *id*, generate `int` variables:
   (a) *id*`state` for the execution state (0=$READY$; 1=$BLOCKED\_OUTPUT$);
   (b) *id*`PATT` for the number of the most recently successful *match*;
   (c) *id*`I/O`*i* for the *i*th input/output link value;
   (d) *id*`I`*i*`STATE` for the *i*th input link status flag (0=$EMPTY$; 1=$FULL$);
2. set all box states to $READY$;
3. generate wire initialisation;
4. generate match stage: for each $READY$ box; for each *match*:
   (a) for each *patt*:
      i. for non-ignore (`*`) *patt*, check if corresponding input link $FULL$;
      ii. for constant *patt*, check input has required value;
   (b) if all *patt*s satisfied:
      i. set corresponding input status flags to $EMPTY$;
      ii. remember that this *pattern* succeeded;
      iii. set appropriate output links to *exp* values, taking *var* values from corresponding input links;
      iv. set box status to $BLOCKED\_OUTPUT$.
5. generate super step stage: for each $BLOCKED\_OUT$ box, for each *match*:
   (a) if *match* succeeded, if all inputs links wired to outputs links $EMPTY$:
      i. copy non-ignore (`*`) output links to input links and set input link status flags to $FULL$;
      ii. set box status to $READY$

Note that the N *pattern*s in a *match* are numbered from 0 to $N-1$.

For example for the second match of box `square` above:

```
(*,s,c,v) -> (*,s+v,c-1,v) |
```

the C generated for intialisation, and the match and super step stages, is:

```
 1. int squareSTATE; int squarePATT;
 2. int squareI0; int squareI0STATE; ...
 3. int squareI3; int squareI3STATE;
 4. int squareO0; ... int squareO3;
    ...
 5. squareSTATE = 0;
    ...
 6. squareI0STATE = 0; ... squareI3STATE = 0;
    ...
 7. square1:
 8.    if(squareI1STATE == 0) goto square2;
 9.    if(squareI2STATE == 0) goto square2;
10.    if(squareI3STATE == 0) goto square2;
11.    squareI1STATE = 0; ... squareI3STATE = 0;
12.    squarePATT = 1;
13.    squareO1 = squareI1+squareI3;
14.    squareO2 = squareI2-1;
15.    squareO3 = squareI3;
16.    goto squareSUCC;
    ...
17. squareSUCC:
18.    squareSTATE = 1;
    ...
19.    if(squareSTATE == 0) goto squareENDSUPER;
    ...
20. squareSPATT1:
21.    if(squarePATT != 1) goto squareSPATT2;
22.    if(squareI1STATE != 0) goto squareSPATT2;
23.    if(squareI2STATE != 0) goto squareSPATT2;
24.    if(squareI3STATE != 0) goto squareSPATT2;
25.    squareI1 = squareO1; squareI1STATE = 1;
26.    squareI2 = squareO2; squareI2STATE = 1;
27.    squareI3 = squareO3; squareI3STATE = 1;
28.    squareSTATE = 0;
29.    goto squareENDSUPER;
    ...
```

Line 1 defines execution and pattern state variables. Lines 2 to 3 define variables and status flags for the input links. Line 4 defines variables for the output links. Line 5 sets the initial box execution state to $READY$. Line 6 sets the input status flags to $EMPTY$.

In the execution stage, at line 7, the first match has failed. For the second match, lines 8 to 10 check that required inputs are $FULL$. Line 11 sets the input status flags to $EMPTY$ and line 12 sets the pattern state to 1 to indicate that the second match succeeded. Lines 13 to 15 set the outputs to the corresponding expressions.

On completion of the box execution, line 18 sets the box state to $BLOCK\_OUTPUT$.

In the superstep stage, line 19 checks that the box state is $BLOCKED\_OUT$. Then, at line 20, the first match had not succeeded. Line 21 checks that the second match succeeded. Lines 22 to 24 check that inputs corresponding to outputs are $EMPTY$. Lines 25 to 27 copy outputs to inputs and set status flags to $FULL$. Line 28 sets the box execution status to $READY$.

While there is some avoidance of redundant jumps, code generation is, over all, naive. Opportunities for improvement include simplification of *self wired* matches, where a box's outputs are only to its own inputs.

4 MicroBlaze FPGA architecture

FPGAs are versatile configurable electronic devices that can be utilised as accelerators to implement tailored computational logic specific to the application being executed. Moreover, these components can be reconfigured at any time for new applications, making it possible to perform a wide range of tasks. Thanks to the continuing advances in CMOS technology, as with cores following Moore's law, FPGAs have now arrived at a stage where they form a viable alternative to Application-Specific Integrated Circuits (ASICs) for many applications.

Craven and Athanas [4] have identified major performance disadvantages of FPGAs compared to microprocessors:

- The maximum clock frequency for FPGAs is typically a few hundred MHz, while non-embedded microprocessors typically run at a few GHz.
- The FPGA's configurability comes at the cost of a large overhead (compared to equivalent-functionality ASICs).
- Floating-point arithmetic on FPGAs is very resource-intensive compared to integer arithmetic (comparable to CPUs without a FPU)

Despite these disadvantages FPGAs are still able to outperform microprocessors because:

- FPGAs are used to design specialised circuits for specific tasks.
- All the logic on the FPGA can be utilised to perform the specific task.
- FPGAs deliver a vast amount of fine-grained parallelism.
- FPGAs offer huge memory bandwidths through configurable logic, on-chip block RAMs, and local memories.

In particular, contemporary soft-core FPGA architectures enable high degrees of flexible and scalable parallelism.

The MicroBlaze soft processor core The MicroBlaze is a 32-bit RISC (reduced instruction set computer) synthesizable soft processor core developed and maintained by Xilinx Inc [16]. It is specifically designed for Xilinx FPGAs and therefore makes efficient use of their resources. The resource utilization of the

MicroBlaze is significantly smaller. Furthermore, the MicroBlaze has a solid documentation and can easily be extended by user defined IP(Intellectual Property)-blocks. Xilinx provides a complete development environment (EDK) to configure the processor and all attached IP-blocks. The EDK includes a complete GNU-tool chain for the software part.

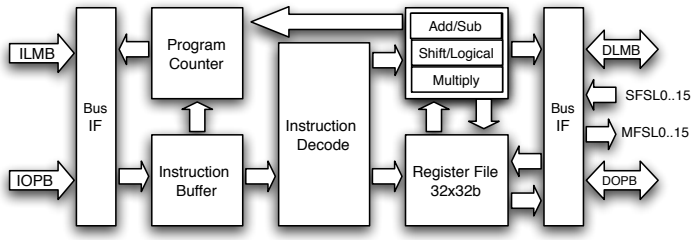


Fig. 4. MicroBlaze core structure

Figure 4 depicts the MicroBlaze building blocks which include:

- general purpose registers,
- instruction word with three operands and two addressing modes,
- instructions and data buses that comply with IBM's OPB (On-chip Peripheral Bus) and PLB (Processor Linker Bus) specification, and provide direct connection to on-chip block RAM through LMB (Local Memory Bus),
- instructions to support FSL (Fast Simplex Link),
- and, hardware multiplier.

The MicroBlaze core implements a Harvard architecture. This means that it has separate bus interface units for data and instruction access. Each bus interface unit is split further into a Local Memory Bus (LMB) and IBM's PLB and OPB buses. The LMB provides single-cycle access to on-chip dual-port block RAM. The PLB and OPB interfaces provide connection to both on and off chip peripherals and memory.

4.1 FSL: Fast Simplex Link

The FSL is a fast interface supported in the MicroBlaze architecture by dedicated low cycle count read and write operations. FSL is formed by an (asynchronous) 32-bit FIFO of configurable depth. This architecture allows high throughput with low latency and simple data handling. The FSL is always a dedicated connection to or from a single component. In the board we use for the experiment in this paper, the MicroBlaze core provides 16 input and 16 output interfaces to FSL channels. Finally, the presence of these FSL channels motivates our choice of the MicroBlaze as a soft core, as it allows us to create a true network of processors. The resulting multi-core architecture has a very high total bandwidth and is entirely free of the traditional bus bottleneck.

5 Parallel FPGA design

5.1 Hardware Apparatus

In our experiment we used the *Xilinx Univesity Program (XUP) Virtex-II Pro* Board. It provides an advanced hardware platform that consists of a high performance Virtex-II Pro FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. The Virtex-II Pro contains two embedded PowerPC 405 cores and a 10/100 Ethernet PHY device. The board provides up to 2GB of double data rate SDRAM, an RS-232 DB9 serial port, an Ethernet port, up to 256MB of CompactFlash storage, four LEDs and four switches, a 100MHz system clock and a 75MHz SATA clock. It also includes support for JTAG-over-USB FPGA configuration bit-streams.

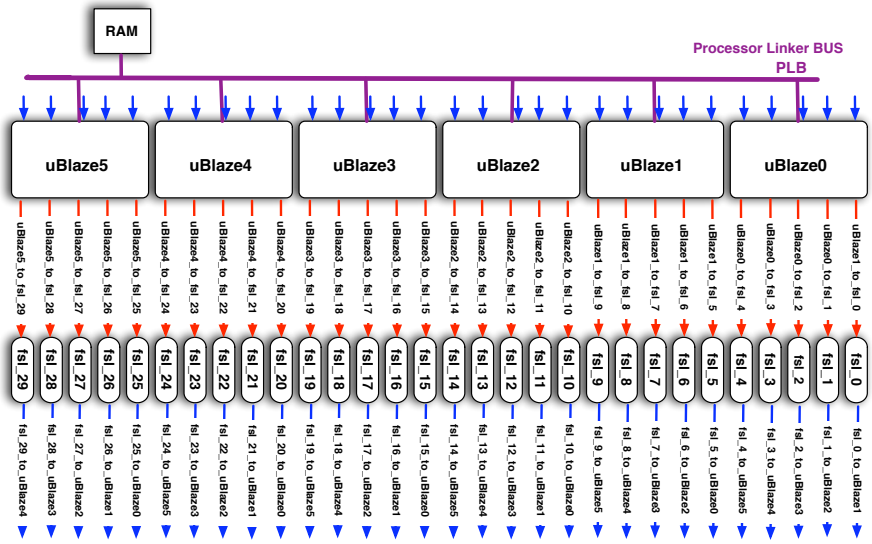


Fig. 5. Parallel Architecture Design

5.2 Parallel Design

In our parallel FPGA design we constructed six MicroBalzes which are fully connected in parallel as show in Figure 5. The communication handler in the MicroBlazes is connected to the FSL ports (subsection 4.1). To increase distribution between the MicroBlazes we design a full mesh topology. To fulfil our network topology design each MicroBlaze uses five FSL buses to communicate

with the other MicroBlazes. We, also, configure the depth of each FSL bus FIFO to one, to realise the Hume super-schedule strategy and at the same time to allow each MicroBlaze to clock at different rates. Using FSL bus for communicate between MicroBlazes provides a master-slave parallel model, i.e, when a MicroBlaze writes to the FSL bus, it will act as a master on the FSL bus and the receiver MicroBlaze from the FSL bus will be consider as a slave to the FSL bus. For instance as shown in Figure 5, MicroBlaze 0 (`uBlaze0`) uses port `uBlaze0_to_fsl_0` to write to FSL 0 (`fsl_0`) and FSL 0 uses port `fsl_0_to_uBlaze1` to write to MicroBlaze 1 (`uBlaze1`). In this scenario MicroBlaze 0 is a master and MicroBlaze 1 is a slave. In our design due to the single depth of the FSL bus, at the moment when MicroBlaze 0 (master) writes on FSL 0, FSL 0 makes the data available to MicroBlaze 1 (slave) and it prevents MicroBlaze 0 from writing again to FSL 0 until MicroBlaze 1 reads the data from FSL 0.

6 mHume on FPGA

6.1 Crafting mHume to Run on the FPGA

Sequential FPGA Design: The generated `C` code from the mHume compiler assumes the presence of a POSIX-style operating system that takes care of I/O, memory allocation and thread/process-level parallelism. However, an operating system is an unjustifiable overhead in the case of deploying mHume generated `C` code on the FPGA: the FPGA will typically be used as an accelerator on a host system, and the host will take care of I/O; as we will see further, we have no need for file system access, processes or threads either. As a consequence, it was essential to rearrange the generated `C` code to be free of OS-specific functionality. For instance, relying on the OS timer to coordinate wires had to be adjusted to use library functions which access the processor cycle count registers instead. Moreover, all memory allocations for wires and boxes had to be adapted to a static allocation instead of using *malloc*-style dynamic assignment. More importantly, embedded hardware systems like FPGA lack the file system concepts, which means mHume implementations had to be enhanced to read input files through dedicated ports on the FPGA board.

Parallel FPGA Design: The sequentially running `C` code has to be split between the designed MicroBlazes on the FPGA. Each mHume box will be assigned to a separate MicroBlaze. All communications between boxes occur through the FSL channels which achieve *completely lock-free communication* between boxes. There is no need to synchronise two communicating boxes: since any output written during one mHume scheduling cycle will never be read before the subsequent cycle, and since the subsequent cycle will not be scheduled before all output is completed, it follows that a box/MicroBlaze can never start to read data before it has completely finished its output.

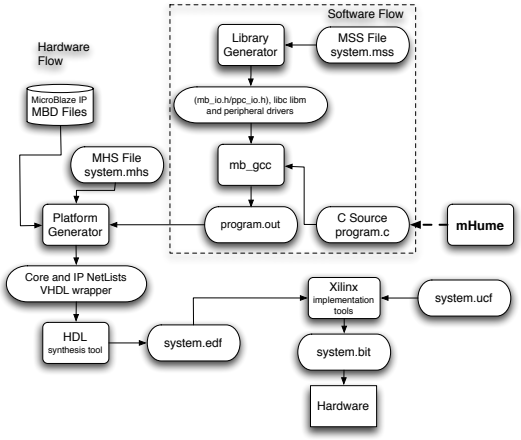


Fig. 6. From mHume to FPGA

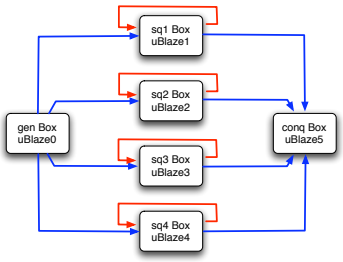


Fig. 7. multi-sq Example in Parallel FPGA

6.2 From mHume to FPGA

Figure 6 illustrates the steps of moving the generated `C` code of mHume to the FPGA. The figure is revamped from the *Xilinx*'s on-line support documentations [15]. The figure includes software and hardware flow. These flows describe the *Xilinx* development tools for MicroBlaze system building process. They include Microprocessor Hardware Specification (MHS) and Microprocessor Software Specification (MSS) files to define hardware and software systems. In our sequential approach from mHume to FPGA, those files have been generated automatically using the *Xilinx* EDK wizard system, with minor changes to simplify the memory connection, Local Access Memory (LAM), with the Multi-Port Memory Controller (MPMC), to fit with mHume's wire and box design. These hardware and software system files provide the core to build the MicroBlaze system automatically using *Xilinx* EDK tools.

For the purpose of our parallel design we had to implement manually the parallel MicroBlazes in the MHS file. Adding an extra MicroBlaze to the design is a bit more than making a replica of one generated by the *Xilinx* EDK wizard. Each MicroBlaze use the Local Memory Bus (LMB) to access the on-chip block RAM, and it uses data and instruction LMB. The on-chip block RAM has to be divided between the participant MicroBlaze processors with the consideration that each MicroBlaze can not allocate more than 8KB for itself. For the MicroBlazes to be able to communicate, FSL buses have to be implemented in the design as coprocessors. Also each MicroBlaze must be aware of the master FSLs, which transfers data to the MicroBlaze, and the slave FSLs, which receives data from the specified MicroBlaze.

Subsequently, the EDK tools integrate the MicroBlaze cores and the appropriate peripherals, and create custom-built C libraries and drivers. After this, the

EDK uses the Platform Generator and Library Generator tools, in the hardware and software flows respectively, to setup the particular hardware and software for the corresponding design. In our implementation, platform and library generator tools have to be configured to select our defined STDIN/STDOUT peripherals, map the added peripherals to the appropriate drivers, specify the correct heap and stack size, map the stack and heap to correspondent memory, and set the correct boots and debug options for the mHume.

At this stage, the generated mHume `C` file for each MicroBlaze use the Library Generator to build system-specific library C functions that map the mHume generated `C` functions with the peripherals functions and configure the C libraries. For clarification, the Library Generator uses the provided mHume configuration to setup the STDIN/STDOUT for the generated `C` files using the STDIN and STDOUT attributes in the MSS file and the *INBYTE* and *OUTBYTE* attributes in the Microprocessor Peripheral Definition (MPD) file. Moreover, the Library Generator writes a *xparameters.h* header file which must be included in the header files. The *xparameters.h* file provides essential information for driver function calls and the base addresses of the peripherals in the system. Moreover it includes the communication ID for each FSL bus which is the only identification MicroBlazes uses for communication over the FSL bus interface.

Then the EDK uses the Platform Generator to build the hardware files, which include the system netlists and HDL code and BlockRAM netlists initialised with the program code. These hardware files are then used by the synthesis toolchain to create the final hardware system (i.e. the MicroBlaze processor and its peripherals) on the FGPA.

7 Software Apparatus

To examine the implementation of our parallel FPGA design, we structured a mHume example with six boxes, *multi-sq*. The first box, *gen*, similar to `gen` above, generates an integer number and passes it to four other boxes. These boxes, *sq1*, *sq2*, *sq3* and *sq4*, all similar to `square` above, calculate the square of the integer received from the *gen* box by using iterative sum and count operations, and then each box passes its output to the *conq* box. The *conq* box sums all the values it has received and prints the output to the STDOUT. After this, the program proceeds to the next cycle and the *gen* box increases the generated value by one. The *multi-sq* program repeats this cycle N times according to the user input. The source code of the program can be obtained from [9].

To run this example in the parallel FPGA design described above, we assign each box to a different MicroBlaze as shown in Figure 7

8 Evaluation

In this section we report and analyse the performance of the *multi-sq* example, comparing the performance of the parallel FPGA with the performance of both a sequential FPGA design and an Intel PC with a 2.33 GHz CPU. We also

Input	Run-time (s)		Speedup
	6 MicroBlaze 100 MHz	Intel Xeon 2.33GHz	
10	0.0000023	0.000005	2.17
100	0.0000203	0.00026	12.80
500	0.0001003	0.006319	63.00
1000	0.0002003	0.025267	126.14
1500	0.0003003	0.053642	178.62
2000	0.0004003	0.091355	228.21
3000	0.0006003	0.199057	331.59
4000	0.0008003	0.35047	437.92
5000	0.0010003	0.545584	545.42
10000	0.0020003	2.170327	1085.00

Table 1. Run-time (s) performance

Input	Clock Cycle		Speedup
	6 MicroBlaze 100 MHz	Intel Xeon 2.33GHz	
10	230	11650	50.65
100	2030	605800	298.42
500	10030	14723270	1467.92
1000	20030	58872110	2939.19
1500	30030	124985860	4162.03
2000	40030	212857150	5317.44
3000	60030	463802810	7726.18
4000	80030	816595100	10203.61
5000	100030	1271210720	12708.29
10000	200030	5056861910	25280.51

Table 2. Clock cycle performance

compare the performance of optimised and non-optimised hand written pure `C` code for the multi-sq program in the sequential architectures and the parallel FPGA design.

8.1 Parallel FPGA Performance vs Sequential FPGA and Intel PC

In Tables 1 and 2, the first column shows the user input to the *multi-sq* program. The second column reports the run-time using the parallel FPGA design and utilising six MicroBlazes of 100MHz, in Seconds and Clock Cycle respectively. The third column, in both Tables 1 and 2, reports the run-time using a Intel Xeon PC with 2.33 GHz, in Seconds and Clock Cycle respectively. Finally, the last column, in both tables, indicates the performance improvement, speedup, between the parallel and sequential run-time.

The performance reported in Tables 1 and 2 shows a clear indication of the massive improvement in the performance under the parallel FPGA design. The advancement of the performance is more evident when clock cycles are compared due to the wide gulf here between the MicroBlaze (100MHz) and the Intel PC (2.33GHz). This immense enhancement of the performance is ascribed to a couple of reasons: the obvious one which is related to the parallel architecture and the ability to evaluate all the six boxes in the program simultaneously at the same moment.

The second reason is imputed to the realisation of the mHume super step (Section 3) in our parallel FPGA design. The super step in mHume guarantees that all boxes are at the same level of evaluation; this means they are required to wait for the slowest box regardless of whether or not its output is relevant to any of the other boxes to be in the next stage of evaluation. The super step restriction produces the quadratic behaviour of the *multi-sq* example as reported in Tables 1 and 2.

In the parallel FPGA implementation, the super step and wiring of mHume is carried out by the FSL buses. To ensure that the parallel FPGA design has

the same guaranteed level of correctness that the super step pledges by restricting the boxes to the same level of evaluation, FSL buses have been formed by a single depth of an asynchronous FIFO. For instance in the *multi-sq* example, the *gen* box generates set of data/inputs to *sq1*, *sq2*, *sq3* and *sq4* boxes, and the data is pushed to the correspond FSL buses. At this stage *gen* is not restricted to wait to coordinate with other boxes to read and evaluated the anticipate data from the buses, as expected from the mHume super step design. On the contrary, *gen* starts its next cycle and produces the next set of data. However, if the generated data is to be pushed to FSL buses where the boxes corresponding to the previous set of data had failed to pull it out, the *gen* box status becomes $BLOCKED\_OUTPUT$ until the other boxes pull out the data from the FSL buses. This will ensure the correctness of the program and avoid any possible conflicts between boxes with regards to evaluations. Moreover, this strategy enhances the performance of the program under the parallel FPGA design as shown in the linear scale of performance in regard to the size of input.

8.2 Parallel FPGA Design Performance Against Hand Written C

For a sanity check, in this subsection we compare the performance of our parallel FPGA design and an Intel Xeon PC against a hand written `C` implementation of the *multi-sq* example. In this comparison we use two different hand written `C` implementations, a fully optimised one and a completely non optimised code for the same problem. The `C` code to be run on the parallel FPGA is compiled and optimised using Xilinx `gcc` compiler.

Input	Run-time (s)				Speedup based on		
	MicroBlaze, 100 MHz		Intel Xeon, 2.33GHz				
	Seq	Par 6	Optim	NO Optim	Seq	Optim	NO Optim
10	0.00002	0.000001	0.000001	0.000003	17.89	0.83	2.50
100	0.00029	0.000009	0.000002	0.000069	31.39	0.22	7.42
500	0.00175	0.000045	0.000008	0.001550	38.67	0.18	34.22
1000	0.00376	0.000090	0.000015	0.006117	41.66	0.16	67.74
1500	0.00589	0.000135	0.000022	0.013691	43.54	0.16	101.19
2000	0.00804	0.000180	0.000029	0.023655	44.61	0.16	131.20
3000	0.01256	0.000270	0.000043	0.04901	46.47	0.16	181.32
4000	0.01712	0.000360	0.000058	0.086294	47.53	0.16	239.51
5000	0.02187	0.000450	0.000072	0.132634	48.58	0.16	294.55
10000	0.04635	0.000900	0.000143	0.518483	51.48	0.16	575.90

Table 3. Run-time performance of running C code of the multi-sq example on embedded processors versus desktop PC

In Table 3, the first column shows the input values for *multi-sq*. The second and third columns report the execution time using a single MicroBlaze and a

Input	Clock Cycle				Speedup based on		
	MicroBlaze, 100 MHz		Intel Xeon, 2.33GHz				
	Seq	Par 6	Optim	NO Optim	Seq	Optim	NO Optim
10	2147	120	2330	6990	17.89	19.42	58.25
100	29193	930	4660	160770	31.39	5.01	172.87
500	175177	4530	18640	3611500	38.67	4.11	797.24
1000	376253	9030	34950	14252610	41.67	3.87	1578.36
1500	589105	13530	51260	31900030	43.54	3.79	2357.73
2000	804405	18030	67570	55116150	44.61	3.75	3056.91
3000	1256109	27030	100190	114193300	46.47	3.70	4224.69
4000	71712709	36030	135140	201065020	47.54	3.75	5580.49
5000	2187709	45030	167760	309037220	48.58	3.73	6862.92
10000	4635317	90030	333190	1208065390	51.49	3.70	13418.48

Table 4. Clock cycle performance of running C code of the multi-sq example on embedded processors versus desktop PC

parallel design of six MicroBlazes respectively. The fourth and fifth columns outline the run-time of the optimised and non-optimised `C` code using Intel Xeon PC with 2.33 GHz CPU. The last three columns reveal the run-time improvement, speedup, of the `C` code using the parallel FPGA design of six MicroBlazes against the single MicroBlaze and the Intel PC running the optimised and non optimised `C` implementation. Table 4 discloses the clock cycle for the run-times presented in Table 3, and reports the performance based on the clock cycle.

The performance results reported in Table 3 show sustained improvement of the hand written `C` code under the parallel FPGA design, particularly in comparison with the single MicroBlaze and the non optimised code performance. A marginal improvement against the optimised `C` code is shown in the seventh column of the table. However, Table 4 exhibits preferable capabilities for the parallel design when clock cycle is considered for measurement, even against the optimised `C` code. The FPGA board used in this experiment is about 10 years old: a more recent board will have faster a clock cycle than the current 100MHz per MicroBlaze and that will result in a smaller run-time and an improvement in the performance against the Intel PC architecture.

9 Related Work

There have been some attempts to extend functional-based programming languages to use FPGAs:

- Lava [2, 3] extends Haskell with operations that allow the high-level description of FPGA circuits.
- Intel's reFL^{ect} [6] is strongly typed and similar to ML, but with quotation and anti-quotation constructs. Its features are intended for applications in hardware design and verification.

– MetaML [13] is very similar to Intel's reFL^{ect} with more direct focus on program generation, and control and optimization of evaluation.
– Template Haskell [14] is also focused on program generation, and the control and optimization of evaluation. To support this it generates code at compile time. An example of a HDL embedded in Haskell using Template Haskell can be found in [12].
– The functional derivation approach is used for deriving FPGA circuits from Haskell specifications [8].
– The Reduceron is an FPGA-based reduction machine targeting Haskell [11]

Our work, as presented in this paper, is novel in adopting a soft processor approach and in attempting to follow a complete development path from source language to target FPGA hardware.

10 Conclusion

We have shown that a simple model of, and compilation route for, mHume can deliver very decent parallel performance on an FPGA, through direct realisation of box concurrency on MicroBlaze soft cores.

mHume is, of course, an experimental language, with data and computation constructs restricted to integers. Nonetheless, as mHume includes the key features of Hume's coordination layer, our preliminary results give us confidence that equally good performance can be achieved for increasingly large Hume subsets. Thus, we next plan to systematically extend mHume with: character, float and string types; vectors; input/output; and auxilliary definitions of functions and constructed types.

Our compiler generates sequential C and substantial hand modification is needed to parallelise it. Thus, a key priority is to fully automate the generation of C with appropriate concurrency constructs, both for MicroBlaze and wider multi-core execution. This will then support mHume execution on heterogeneous systems combining multi-core with FPGA, in the first instance by user nomination of box placement, and, in the longer term, with automatic placement driven by cost analysis.

In the longer term we also plan to further optimise the C generated for mHume, in particular by identifying self-output boxes, and by implementing the Hierarchical Hume box encapsulation[5].

Acknowledgements

This research is supported by the UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072X, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds". We are also grateful to *Xilinx* for support through their University Programme.

References

1. A. Al Zain, W. Vanderbauwhede, and G. Michaelson. Hume on fpga. In *Draft Proceedings of 10th International Symposium on Trends in Functional Programming (TFP10)*, University of Oklahoma, Oklahoma, USA, 2010.
2. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.
3. K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.
4. S. Craven and P. Athanas. Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded Systems*, 2007(1):13, 2007.
5. G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. In M. Morazon, editor, *Trends in Functional Programming*, volume 8, pages 71–88. Intellect, 2008.
6. J. Grundy, T. Melham, and J. O'leary. A reflective functional language for hardware design and theorem proving. *Journal Functional Programming*, 16(2):157–196, 2006.
7. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.
8. J. Hawkins and A. Abdallah. Behavioural synthesis of a parallel hardware jpeg decoder from a functitonal specification. In *Proc. EuroPar 2002*, August 2002.
9. G. Michaelson. multisq: minihume example. ONLINE, June 2010. http://www.macs.hw.ac.uk/~ceeatia/IFL10-miniHume/multisq.hume.
10. G. Michaelson, G. Grove, and A. Al Zain. Multi-core parallelisation of hume through structured transformation. In *Draft Proc. of 21st Intl. Workshop on Implementation and Application of Functional Languages (IFL '09)*, Seton-Hall University, New Jersey, USA, 2009.
11. M. Naylor and C. Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. *Implementation and Application of Functional Languages*, pages 129–146, 2008.
12. J. Odonnell. Embedding a hardware description language in template haskell. *Domain-Specific Program Generation*, pages 195–265, 2004.
13. E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *In the International Conference on Functional Programming (ICFP 02*, pages 218–229. ACM, 2002.
14. T. Sheard and S. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.
15. Xilinx support documentation. Online White Papers. http://www.xilinx.com/support/documentation/white_papers/.
16. Xilinx. MicroBlaze Processor Reference Guide (v 4.0). Technical report, Xilinx, 2004.

The Essence of Synchronisation in Asynchronous Data Flow Programming *

Clemens Grelck

University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. We discuss the aspect of synchronisation in the language design of the asynchronous data flow language S-Net. Synchronisation is a crucial aspect of any coordination approach. S-Net provides a particularly simple construct, the synchrocell. The synchrocell is actually two simple to meet regular synchronisation demands itself. We show that in conjunction with other language feature, S-Net synchrocells can effectively do the job. Moreover, we argue that their simplistic design in fact is a necessary prerequisite to implement even more interesting scenarios, for which we outline ways of efficient implementation.

1 Introduction

The current hardware trend towards multi-core and soon many-core chip designs for increased performance creates a huge problem on the software side. Software needs to become parallel to benefit from future improvements in hardware, which will be rather in th number of cores available than in the individual computing power of a single core. So far, parallel programming has been confined to niches of software engineering, like for instance high performance computing. Now, parallel programming must become mainstream, and the real question is how existing software can be parallelised and how new software can be engineered without the cost currently attributed to, for instance, high performance computing.

S-Net [1–3] is a declarative coordination language whose design thoroughly avoids the intertwining of computational and organisational aspects through active separation of concerns: S-Net completely separates the concern of writing sequential application building blocks (i.e. *application engineering)* from the the concern of composing these building blocks to form a parallel application (i.e.*concurrency engineering).*

More precisely, S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes*, to conventional languages. An S-Net box is connected to the outside

* This work was supported by the European Union through the projects Æther and Advance.

world by two typed streams, a single input stream and a single output stream. Data on these streams is organised as non-recursive records, i.e. collections of label-value pairs.

The operational behaviour of a box is characterised by a stream transformer function that maps a single record from the input stream to a (possibly empty) stream of records on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. Boxes execute fully asynchronously: as soon as a record is available on the input stream, a box may start computing and producing records on the output stream.

The restriction to a single input stream and a single output stream per box again again is motivated by separation of concurrency engineering from application engineering. If a box had multiple input streams, this would immediately raise the question as to what extent input data arriving on the various input streams is synchronised. Do we wait for exactly one data package on each input stream before we start computing like in Petri nets? Or do we alternatively start computing when the first data item arrives and see how far we get without the other data? Or could we even consume varying numbers of data packages from the various input streams? This immediately intertwines the question of synchronisation, which is a classical concurrency engineering concern, with the concept of the box, which in fact is and should only be an abstraction of a sequential compute component. The same is true for the output stream of a box. Had a box multiple output streams, this would immediately raise the question of data routing, again a classical concurrency engineering concern, as the box code would need to decide to which stream data should be sent. Having a single output stream only, in contrast, clearly separates the routing aspect from the computing aspect of the box and, thus, concurrency engineering from application engineering.

The construction of streaming networks based on instances of asynchronous components is a distinctive feature of S-Net: Thanks to the restriction to a single-input/single-output stream component interface we can describe entire networks through algebraic formulae. Network combinators either take one or two operand components and construct a network that again has a single input stream and a single output stream. As such a network again is a component, construction of streaming networks becomes an inductive process. We have identified a total of four network combinators that prove sufficient to construct a large number of network prototypes: static serial and parallel composition of heterogeneous components as well as dynamic serial and parallel replication of homogeneous components.

Fig. 1 shows a simple example of an S-Net streaming network. The five boxes execute fully asynchronously and data items or tokens in the form of records flow through the stages of the streaming network. Neither the split point nor the merge point in the network constitute any form of synchronisation. Both

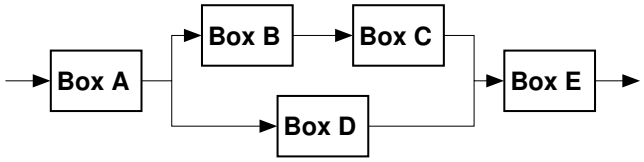


Fig. 1. Example of an S-Net streaming network of asynchronous components

are rather routing points where data can be either sent into multiple directions or received from multiple directions.

What does synchronisation mean in the streaming network context of S-Net? Due to the SISO restriction of network construction, no S-Net entity can have multiple input streams. Hence, synchronisation can only mean the combination of multiple data items or records appearing in some order on a single input stream. For this purpose S-Net provides a built-in component, the *synchrocell*. The design of the synchrocell is geared towards simplicity. In fact, a synchrocell waits for two differently typed records on its own input stream. Whichever record appears first is held within the synchrocell until the other appears as well. On this occasion the the synchrocell joins the two records and emits the resulting record on its output stream. Any synchrocell can only synchronise exactly once. Whenever a type pattern (more on this in the following chapters) has been matched any further records matching the same pattern will be forwarded to the output stream directly. Consequently, after successful synchronisation all patterns have been matched and, hence, the synchrocell becomes an identity box.

This operational behaviour may surprise. However, exactly this one-off behaviour proves essential to use synchrocells effectively in varying contexts. It is the contribution of this paper to show how and why.

The remainder of the paper is organised as follows: Section 2 provides more technical background information on S-Net while in Section 3 we focus entirely on the aspect of synchronisation and introduce S-Net synchrocells. Sections 4 and 5 describe two application scenarios for synchrocells that proves their simplistic design as an essential prerequisite. In Section 6 we sketch out directions of efficient implementation, and we conclude in Section 7.

2 S-Net in a Nutshell

S-Net is a high-level, declarative coordination language based on the concept of stream processing. As such S-Net promotes functions implemented in a standard programming language into asynchronously executed stream-processing components, termed *boxes*. Both imperative and declarative programming languages qualify as box implementation languages for S-Net, but we require any box implementation to be free of state on the coordination level. More precisely, a

box must not carry over any information between two consecutive activations on the streaming layer.

Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organized as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. The fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers, which are accessible both on the coordination and on the box level. Tag labels are distinguished from field labels by angular brackets.

Operationally, a box is triggered by receiving a record on its input stream. As soon as that happend, the box applies its box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has terminated, the box is ready to receive and to process the next record on the input stream.

On the S-Net level a box is characterized by a *box signature*: a mapping from an input type to a disjunction of output types. For example,

```
box foo ((a,<b>) -> (c) | (c,d,<e>));
```

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to qualify for any opaque data; the second argument is of type `int`.

The box signature naturally induces a *type signature*. Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain. Consequently, this step turns tuples of labels into sets of labels. Hence, the type signature of box `foo` is `{a,<b>} -> {c} | {c,d,<e>}`. We call the left hand side of this type mapping the *input type* and the right hand side the *output type*, and we use curly brackets instead of round brackets to emphasise the set nature of types.

To be precise, this type signature makes `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$. Again, the variant v is a subtype of w if and only if every label $\lambda \in v$ also appears in w.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. As mentioned previously, S-Net supports the concept of *flow inheritance* whereby excess fields and tags from incoming records are not just ignored in the input record of a network entity, but are also attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance prove to be indispensable when it comes to getting boxes that were designed separately to work together in a streaming network.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property: any network, regardless of its complexity, is an SISO entity in its own right.

Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in pipeline mode.

Parallel combination (`A|B`) constructs a network where incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the combined network. The type system controls the flow of records. Each network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record. If both branches match equally well, one is selected non-deterministically. Parallel composition can be used to route different kinds of records through different branches of the network (like branches in imperative languages) or, in the presence of subtyping, to create generic and specific alternatives triggered by the presence or the absence of certain fields or tags.

The parallel and serial combinators have their infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator `A*`*`type`* constructs an infinite chain of replicas of `A` connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. More precisely the type acts as a so-called *type pattern* and patter matching is defined via the same subtype relationship as defined above. Hence, a record leaves a serial replication context as soon as its type is a subtype of the type specified in the type pattern position.

The parallel replicator `A!<`*`tag`*`>` also replicates network `A` infinitely, but the replicas are connected in parallel. All incoming records must carry the tag; its value determines the replica to which a record is sent.

S-Net is an abstract notation for streaming networks of asynchronous components. It is a notation that allows programmers to express concurrency in an abstract and intuitive way without the need to reason about the typical annoyances of machine-level concurrent programming. Readers are referred to [3, 2, 4] for a more thorough presentation of S-Net and to [5, 6] for case studies on application programming with S-Net.

3 Synchronisation

What does synchronisation mean in the streaming network context of S-Net? Network combinators inspect records for routing purposes, but never manipulate individual records. This is the privilege of boxes and filters. They both may also split a single record into several records. However, no user-defined box or filter can ever join two records into a single one. The absence of state is an essential property of boxes. So, boxes cannot join records. Joining records is the essence of synchronisation in asynchronous data flow computing in the style of S-Net. Since synchronisation is an integral aspect of the coordination layer, we separate synchronisation as far as possible from any computing aspects and provide a special construct for this purpose: the *synchrocell*.

Syntactically, a synchrocell consists of two type patterns enclosed in `[|` and `|]` brackets, for example `[| {a,b,<t>}, {c,d,<u>} |]`. The synchrocell holds incoming records which match one of the patterns until both patterns have been matched. Only then are the records merged into a single one, which is released to the output stream. A match happens when the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification of the synchrocell: it only accepts records that match at least one of the patterns. Any record arriving at a synchrocell whose type matches a pattern that has previously been matched by a preceding record, is directly forwarded to the output stream without any further processing or even analysis. Consequently, once both patterns of a synchrocell have been matched and a joined record has been emitted to the output stream, the synchrocell is bound to forward all further records regardless of their type directly to the output stream. Effectively, once a synchrocell has successfully synchronised and joined two records, it becomes the identity box. In a more operational sense, the synchrocell should be removed from the streaming network. In essence, synchronisation in S-Net is a one-shot operation.

This definition of synchronisation, in our view, is the bare minimum that is required: a one-shot synchronisation between two records on the same stream. This is truly the essence of synchronisation in the asynchronous data flow context of S-Net. This one-shot design of the synchrocell seems almost disturbing at first glance. However, in the following sections we will demonstrate how this simplistic design and only this design allows to implement essential higher level synchronisation features like continuous pairwise synchronisation or modelling of state.

4 Continuous Synchronisation

A very common form of synchronisation in a streaming network is *continuous synchronisation*, where the first record that matches pattern A is joined with the first record that matches pattern B, the second record that matches pattern A with the second record that matches pattern B and so on. With S-Net synchrocells this behaviour can easily be achieved when embedded into a serial

replication with an exit pattern that is the union of all patterns in the synchrocell. For example, the network

```
[| {A,B},{C,D} |] * {A,B,C,D}
```

achieves continuous synchronisation for records of type {A,B} with records of type {C,D}. Let us see how this works in detail. Fig. 2 illustrates the operational behaviour.

Initially the serial replication is uninstantiated. When the first record, say {A,B} arrives, it does not match the exit pattern of the star combinator and, hence, the serial replication unfolds and instantiates a fresh synchrocell, where the record is stored. Let us assume another record {A,B} comes next. Since it neither matches the exit pattern it is routed into the first instance of the serial replication and hits the synchrocell. As the corresponding pattern has already been matched, the synchrocell passes {A,B} on. It still does not match the exit pattern and, hence, a second instance of the serial replication unfolds, which again has a fresh synchrocell where the record is stored.

If the third record is of type {C,D}, it likewise is routed into the first instance of the serial replication where it hits the synchrocell that is already primed with the first record {A,B}. The synchrocell joins the two records forming a new record {A,B,C,D}. This record does match the exit pattern of the serial replication and leaves the network. Semantically the first synchrocell now becomes the identity, but, operationally, the whole first instance of the serial replication is removed and subsequent records immediately reach the second instance.

In fact, the combination of synchrocell and serial replication allows us to implement an unbounded matching store with the means of S-Net. Of course, we could also have just defined the semantics of the synchrocell as a matching store and provide continuous synchronisation built-in. We do not do so for two reasons. Firstly, it is good programming language design to keep the number and the complexity of primitive language constructs to the minimum and use constructive features whenever possible. Secondly, and actually more important, the simplistic design of synchrocells allows for another application, more precisely the modelling of state in an otherwise state free environment. We will learn more about this in the following section.

5 Modelling State

Functional programming and state typically do not go well together, and S-Net makes no exception here. In general, the absence of state must be seen as the great advantage of function approaches when it comes to parallel processing. In the context of S-Net, for example, the statelessness of boxes allows the S-Net runtime system to schedule box computations to computing resources at will, including the relocation and migration of computations between processing elements. A common example of modelling some form of state in main stream functional programming is tail-end recursion: a function with a number of parameters applies itself (recursively) to a new set of arguments computed from

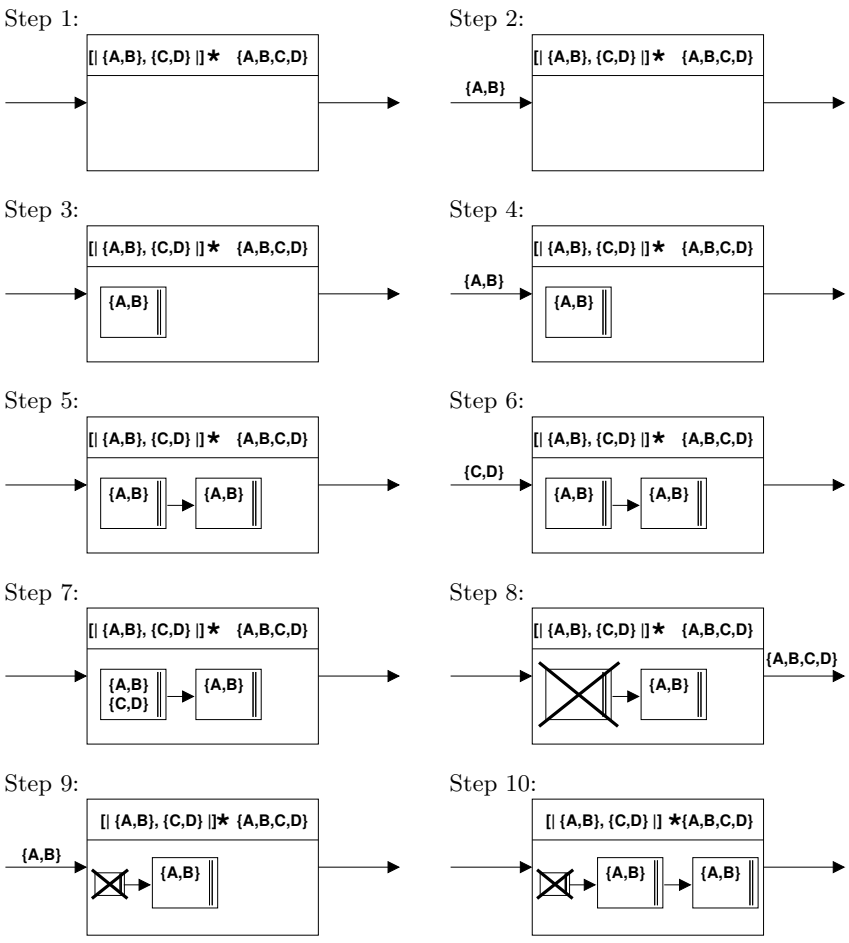


Fig. 2. Illustration of continuous synchronisation

the parameters in the last syntactic position, i.e. the value of recursive function application will become the value of the current function invocation.

Modelling state in S-Net follows the same basic idea, but, of course, has its intricacies due to the stream processing approach of S-Net. Fig. 3 shows an example implementation. The network `model_state` expects an initial state {`state`} followed by a sequence of values {`inval`} on its input stream; it will emit a sequence of values {`outval`} on its output stream, where each output value is a function of the corresponding input value and the accumulated state.

The local networks `join` and `id` wrap a synchrocell and a filter, respectively. Lifting these built-in constructs into separate networks is solely for illustration purposes and has otherwise no semantic implication. At the core of the network is the box `step` that expects a pair of state and value on the input stream. Based

```
net model_state {
  net join connect [|{state},{inval}|];
  net id connect [{inval} -> {inval}];
  box step ({state,inval} -> {state} | {outval});
}
connect (join..(id|step))*{outval};
```

Fig. 3. S-Net network that models a stateful computation.

on both the current state and the current value the box computes a new state and an output value that are individually emitted on the output stream. The network `model_state` is made up of a synchrocell that synchronises one state token with one input token. The subsequent parallel composition is crucial for the overall idea. Whenever the initial synchrocell combines state and value to one record, this record will be routed towards the `step` box because its input type is {`state,inval`}. Any subsequent value that just passes through the synchrocell will be routed towards the filter in the `id` network, which essentially is a typed identity.

The network consisting of `join`, `id` and `step` is embedded within a serial replication, i.e. the whole network is forward replicated on demand. The termination pattern {`outval`} ensures that any new value computed by the `step` box leaves the `model_state` network, whereas any new state computed by the `step` box triggers a re-instantiation of the network, including a fresh synchrocell where the new state is captured to wait for the corresponding value from the outermost input stream.

In Fig. 4 we illustrate the operational behaviour of the `model_state` network. Note that for space reasons we abbreviate {`state`}, {`inval`} and {`outval`} as {`s`}, {`iv`} and {`ov`}, respectively. Initially, the operand network of the serial replication remains uninstantiated, and only the appearance of the first state token (i.e. the initial state) triggers the instantiation of one instance. The {`s`} record is immediately captured in the synchrocell. Next, we expect the first value to appear on the input stream. It is likewise captured in the synchrocell leading to successful synchronisation and the construction of a joined {`s,iv`} record. The synchrocell becomes the identity thereafter.

The {`s,iv`} is routed towards the `step` box due to its type, which matches the input type of that box more than the input type of the `id` network, which is {`iv`}. The `step` box first emits an output value {`ov`}, which is sent to the global output stream, as it perfectly matches the termination pattern of the serial replication. Next, the `step` box emits a state token {`s`}. As this record does not match the termination pattern, it triggers a subsequent instantiation of the serial replication's operand network. The state token flows into this newly instantiated network, where is stuck in the fresh synchrocell.

In the meantime a second value appears on the input stream. It passes the first (disabled) synchrocell and bypasses the first instance of the `step` box. Note that we deliberately omit the identity filter in Fig. 4 as it can easily be identified

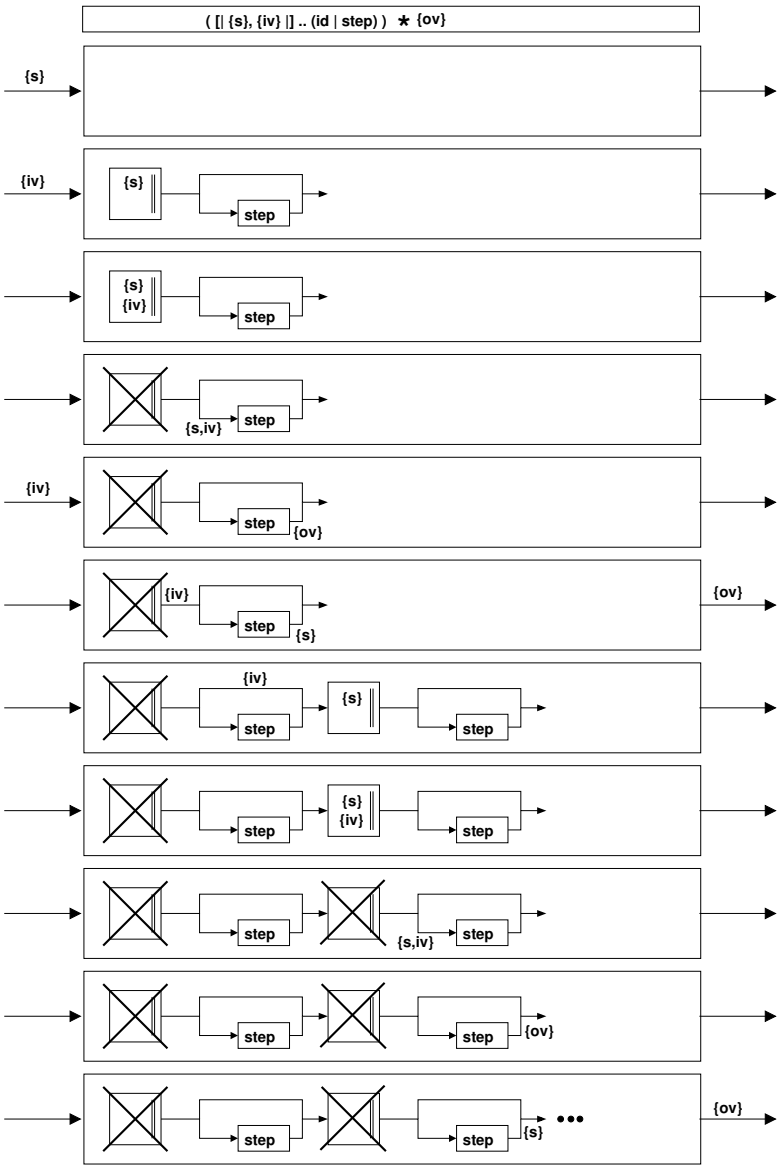


Fig. 4. Illustration of state-modelling network

by our implementation and simply leads to a bypass channel at runtime. Since {iv} does again not match the termination pattern of the serial replication, it is also routed into the second instance of the operand network. Here, it joins the state token {s} in the synchrocell. Now, history repeats itself leading to a

new output record sent to the global output stream and a new state record that triggers the next instantiation of the operand network.

The trick is that each instance of the subnetwork containing `join`, `id` and `step` processes exactly one input value and one instance of the state. After that each instance effectively becomes the identity for any subsequent input value. Conversely, each input record is routed such that it eventually reaches the front or active instance where it triggers the computation of a new output value and a new state.

6 Implementation Aspects

The current S-Net implementation for shared address space parallel architectures [7] is based on threads for boxes, including filters and synchrocells, and bounded FIFO buffers as streams from which threads read and to which threads write. In addition to the runtime components that explicitly appear in an S-Net specification, i.e. essentially boxes, filters and synchrocells, the S-Net runtime system makes use of a small further number of components that only implicitly appear on the S-Net level. These are routing components for parallel composition (routing based on best match of record type and the types of the outgoing streams), serial replication (routing into the replicated network potentially trigger a re-instantiation or routing outside) and parallel replication (routing based on the value of a named tag in the record) as well as merge components for all three underlying network combinators. In essence each network combinator except for serial composition inflicts the instantiation one routing and one merging component for network control.

The most naive implementation of synchrocells in this context is by an internal finite state machine that once it reaches its final state routes any further record on the input stream directly to the output stream. While obviously satisfying the semantics of S-Net [8], this solution is likewise obviously dissatisfying as the synchrocell component infinitely binds resources at runtime for no good and also the movement of records through the network is nothing but delayed by such useless components.

In fact, the current runtime system [7] takes a more reasonable approach. As soon as the finite state machine inherent to the synchrocell (runtime component) reaches its final state, it wraps the address of its input stream in a control message and sends this message to its own output stream. Thereafter, the component immediately terminates thus releasing all associated resources. When the subsequent component, i.e. the component that has the synchrocell's output stream as input stream, receives this control message, it releases its input stream, which in this situation is guaranteed to be empty, and uses the former synchrocell's input stream as its new input stream.

This implementation is adequate from the perspective of a single synchrocell, but is it also sufficient for the two application scenarios sketched out in the previous sections?

Let us first look at continuous synchronisation as outlined in Section 4. The runtime configuration of a synchrocell embedded within a serial replication is as sketched out in Fig. 5. As an example, we see a two-fold instantiation of the continuous synchronisation subnetwork. The serial repliation combinator leads to a sequence of routing components alternating with synchrocells as the only component embedded within the serial replication.

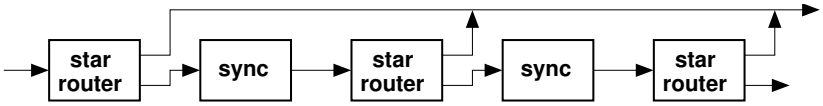


Fig. 5. Runtime configuration of continuous synchronisation

If a synchrocell terminates in this configuration, one routing component becomes directly connected to the next routing component. Since all routing component base their routing decision on the same type pattern, it is clear that serial replication with an empty operand network is semantically transparent, i.e. the instance of the serial replication may be eliminated and not just the synchrocell inside.

This situation can relatively easily be identified by the runtime system. Components register themselves with a stream when they first connect. More precisely, they store their individual identifiers withing the stream. At runtime, this allows components to identify their communication partners. Hence, a serial replication routing component, upon receiving its new input stream from the disabled synchrocell, can immediately identify whether or not the writing component to this stream belongs to the same serial replication. If so, the routing component, rather than reconnecting itself, forwards the stream to its own successor before terminates itself. As a consequence of this simple solution, any useless stage of the serial replication terminates and releases all resources along with the successful synchronisation.

The modelling of state, as proposed in the previous Section, is a much harder problem when it comes to efficient implementations. In this case, effective removal of the synchrocell or its bypassing does not suffice because for each stage of the serial replication the parallel composition within would simply remain as it is. This case is more complicated than that of continuous synchronisation, because the body of the serial replication contains a number of entities and not just a synchrocell. The trick in this case is that following successful synchronisation, the initial synchrocell can no longer produce records of the joint type, but only records of type. Consequently, all such records are guaranteed to use the bypass around the `step` box. Like in the continuous synchronisation case, the entire instance of the serial replication becomes the identity. However, this fact is much harder to identify here.

One potential approach goes as follows. When a synchrocell terminates, it not only wraps its input buffer into a control message but also a runtime representation of the synchrocell patterns. We assume that after termination of the synchrocell only records that match one of the synchronisation patterns may appear, but no records that contain the union of fields and tags. In the given example, this message arrives at the routing component derived from the parallel composition. If the routing component compares the type information contained in the message with the types that form the basis of its routing decision, it can quickly come to the conclusion that all further records must take the upper or bypass branch. Hence, the routing component can send a deactivation control message into the other branch that subsequently removes the whole (now useless) network until the corresponding join point.

Once this has happened we find the network in the same situation as in the previous scenario of continuous synchronisation. We have a serial replication with an empty body, i.e. two routing components of the star combinator have become immediate neighbours in the streaming network. Fortunately, we can use the same technique as above to finalise an entire stage in the dynamically replicated pipeline.

In either scenario the implementation tricks sketched out lead to a queue-like behaviour of serial replication. The pipeline grows at the front, and it shrinks at the end. Under normal circumstances, resource consumption should remain within reasonable bounds rather than grow unbounded as in the naive implementation.

7 Conclusion

In this paper we have explained and motivated the concept of synchronisation through synchrocells in the asynchronous data flow coordination language S-Net. Synchrocells, in our view, capture the bare essence of synchronisation in the context of S-Net. Through two examples, continuous synchronisation and the modelling of state, we demonstrate that it is this bare-bone design of synchrocells that acts as a prerequisite for the expressiveness of S-Net, when combined with other language features. In other words the deliberate restriction to a built-in synchronisation primitive that is unexpectedly simple proves to be essential for expressiveness, a seeming paradox.

We have shown how to use synchrocells to implement advanced synchronisation concepts. Their efficient implementation, however, is a slightly different matter. In fact, orthogonal language design that aims at using a minimum of built-in constructs of minimal complexity, puts a specific burden on language implementors. We have sketched out approaches to implement the scenarios used throughout the paper efficiently. Realisations of these implementation concepts are still outstanding leaving their practical evaluation as future work.

Acknowledgements

Design and implementation of S-Net is truly a team effort. The author would like to thank the other members of the S-Net team, namely Alex Shafarenko, Sven-Bodo Scholz and Frank Penczek, for endless fruitful discussions and an exciting research context.

References

1. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237
2. Grelck, C., Scholz, S.B., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
3. Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2010)
4. Shafarenko, A.: Nondeterministic coordination using s-net. In Gentzsch, W., Grandinetti, L., Joubert, G., eds.: High Speed and Large Scale Scientific Computing. Volume 18 of Advances in Parallel Computing. IOS Press (2009) 74–96
5. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)
6. Penczek, F., Herhut, S., Grelck, C., Scholz, S.B., Shafarenko, A., Barrere, R., Lenormand, E.: Parallel signal processing with S-Net. Procedia Computer Science **1** (2010) 2079 – 2088 ICCS 2010.
7. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
8. Penczek, F., Grelck, C., Scholz, S.B.: An Operational Semantics for S-Net. In Chapman, B., Desprez, F., Joubert, G., Lichnewsky, A., Peters, F., Priol, T., eds.: Parallel Computing: From Multicores and GPU's to Petascale. Volume 19 of Advances in Parallel Computing. IOS Press (2010) 467–474

An Executable Semantics for D-Clean *

Viktória Zsók[1], Pieter Koopman[2], and Rinus Plasmeijer[2]

1: Department of Programming Languages and Compilers, Faculty of Informatics,
Eötvös Lorláand University, Budapest, Hungary
zsv@inf.elte.hu
2: Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
pieter@ru.cs.nl
rinus@ru.cs.nl

Extended abstract

Abstract. D-Clean is a distributed variant of Clean for a Grid system. With D-Clean primitives the programmer distributes the computation over a number of computation nodes placed in boxes and interconnected by channels. These primitives are first class citizens which allows the coordination of a very dynamical work distributions. The computations are distributed automatically over the Grid by the middleware system. In order to obtain sufficient concurrent work, there is a subtle balance between the ordinary lazy computation in Clean and the strict computations on the channels. This paper enlighten the D-Clean extension of Clean. First, we give an executable semantics for D-Clean that defines this evaluation order. Second, we describe a graphical system that displays the maximum amount of parallelism graphically. Third, we describe some transformations that can be used to change the amount of work on channels. Finally, we state some properties of D-Clean.

1 Introduction

Distributed Clean [3, 7], or D-Clean, is a distributed version of Clean [6]. In D-Clean the programmer indicates how the work can be divided between *computation boxes* connected via *channels*. The functions placed in boxes are coordinated by a relatively small number of specific D-Clean primitives, and they are applied on the dataflow of the communication channels. Every channel is carrying data elements of a specified type of the dataflow from one computational node to another one.

D-Clean is the top layer of the distributed system for which it was designed. It has coordinating role and it is used to define the computation scheme. The intermediate level is the D-Box level, a Petri net based language [1] with the task of calculating the function of a node in the distributed computation graph. The boxes are compiled into a Clean program, and they communicate via channels according to well established protocols.

* Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003

In [3] the semantics is described informally. In this paper first we give a formal description of the semantics using Clean as the description language abstracting from the real distributed environment and leaving out the details of the generation of the boxes and channels.

In a number of examples D-Clean obtains the speedups from parallelism one hopes with relative little effort. In other examples it appeared much harder to achieve a good speedup [8]. Although the potential speedup is completely determined by the amount of channels and the computations on these channels as specified in a D-Clean program, it is sometimes hard to predict the speedup. Channel and box creation is part of the distributed system. Hence the amount of channels can be data-dependent.

An extension of the semantics allows us to depict the potential amount of parallelism graphically, described in the second part of the paper. The actual amount of parallelism, and hence the speedup, depends on the order of channel creation and the amount of work on the channels. Nevertheless, information on the potential amount of parallelism is a valuable tool in the design of D-Clean programs with good speedups. The third part of the paper provides transformations that eases the work-balance of channels. Finally properties of D-Clean programs are formulated.

2 D-Clean primitives and their executable semantics

A D-Clean coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence of input channels. The coordination primitives return the result dataflow on the specified output channels. The signature of the coordination primitive, i.e. the types of the input and output channels are inferred according to the type of the embedded Clean expressions. In the executable semantics we provide here some abstractions of the real distributed world is made. They ease our description and sanity checks over D-Clean programs can be made easily.

In the distributed system a D-Clean expression can be either a function or a composition of D-Clean expression parameterized by Clean functions to be placed in a concurrently executed box. Here we will ignore the boxes, only the established channels will be explicitly given by simple integer numbering.

```
// one function or a composition of functions
:: DExpr a b =   F String (a → b)
             | D (DFun (Ch a) (Ch b))
// a DClean expression is a state transition function
:: DFun a b      := a State → (b, State)
// single channel, here used with simplified numbering
:: Ch a          := (Int,a)
// multiple channel
:: MCh a         := [Ch a]
```

2.1 The `DStart` and `DStop` primitives

The task of the `DStart` primitive is to start building up the distributed computation by producing the communication channels for the input dataflow of the distributed graph. It has no input channels, only output channels.

The `DStart` primitive will take the input function and dataflow and starts the computation, the results are sent to the output channels. Each D-Clean program contains at least one `DStart` primitive.

The functional description using Clean functions is as follows:

```
DStart :: a (DFun a b) State → (b, State)
DStart a expr state = expr a state
```

The other coordination primitive which must be included in any D-Clean program is the `DStop` primitive.

The task of this primitive is to close all the communication channels and save the result of the computation process. It has as many input channels as the function expression requires, but it has no output channels. Each D-Clean program contains at least one `DStop` primitive, and it is the final element of a D-Clean program, i.e the last element of the process network. The function for the semantics of the primitive is given as follows:

```
DStop :: (a, State) → (a, State)
DStop (result, state) = (result, takeWorld state)
where
    takeWorld (n,t) = (n,reverse t)
```

We use `DExec` as a wrapper to lift the "normal" world of Clean into the distributed world of D-Clean, therefore we include immediately the two above described primitives by the `DExec` encapsulation.

```
DExec :: a (DFun a b) → (a, (b, State))
DExec a expr  = (a, DStop (DStart a expr (0,[])))
```

2.2 The `DApply` primitive and its variants

`DApply` with one function parameter applies the parameter to the dataflow on a single threaded computation.

```
DApply  :: String (a → b)  → DFun (Ch a)  (Ch b)  | toString b
DApply name fun = mkFunBox name fun
```

In general the `DApply` primitive is used on multiple threaded computation, illustrated by the `MCh` type. The first variant, `DApply1` applies the same function parameter n times. The second variant of `DApplyN` may apply different function expressions on the different dataflows of the computation threads. The function sequence is given in a list of expressions.

```
DApply1 :: (DExpr a b)  → DFun (MCh a) (MCh b) | toString b
DApply1 dexpr
= λinflows → DApplyN (repeatn (length inflows) dexpr) inflows
```

```
DApplyN :: [DExpr a b]  → DFun (MCh a) (MCh b) | toString b
DApplyN funs = handleDExpr funs
where
    handleDExpr :: [DExpr a b] (MCh a) State → (MCh b, State)| toString b
    handleDExpr [] [] state         = ([], state)
    handleDExpr []  inflows state   = abort "run-time error"
    handleDExpr [F name fun:funs]   [i:is] state
    # (result,state)    =   mkFunBox name fun i state
    # (results,state)   =   handleDExpr funs is state
    = ([result:results],state)
    handleDExpr [D dfun:funs] [i:is] state
    # (result,state)    =   dfun i state
    # (results,state)   =   handleDExpr funs is state
    = ([result:results],state)
```

DMap is the distributed version of the well known standard map library function. The D-Clean variant is a computational node which applies the parameter expression to every element of the incoming dataflow. The parameter function of a DMap must be an elementwise processable function. It can be observed that DMap is a special case of DApply.

```
DMap :: (a → b) → DFun (Ch [a]) (Ch [b])  | toString b
DMap fun = DApply "DMap" (map fun)
```

DReduce is another special case of DApply with some restrictions. A valid expression for DReduce has to reduce the dimension of the input dataflow.

```
DReduce :: ([b] → a) → DFun (MCh [b]) (MCh a) | toString a
```

The opposite of the DReduce is the DProduce primitive, which is another special case of DApply. The expression has to increase the dimension of the dataflows.

```
DProduce :: (a → [b]) → DFun (MCh a) (MCh [b]) | toString b
```

DFilter also a special case of DApply, and it is filtering the input dataflow according to a boolean expression.

```
DFilter :: (a → Bool) → DFun (MCh [a]) (MCh [a]) | toString a
```

2.3 The DDivide primitive

DDivideS is a static divider. The parameter expression splits the input dataflow into n parts and broadcasts them to n computational nodes. This primitive is called a static divider since the value of n is known at compilation time. The DDivideD version determines dynamically the number of computation threads it needs to sparkle.

```
DDivideS :: (Int [a] → [[a]]) Int → DFun (Ch [a]) (MCh [a]) | toString a
DDivideD :: (a → [b]) → DFun (Ch a)  (MCh b) | toString b
```

2.4 The `DMerge` primitive

The counterpart of the `DDivide` is the `DMerge` primitive, which collects the input dataflows from input channels and builds up the only one output dataflow. All the input channels must have the same type.

```
DMerge :: ([b] → a) → DFun (MCh b) (Ch a) | toString a
```

3 Transformations

Since channels behave in some sense similar to lists, we can apply a number of list-like transformations. These transformations can be used to change the operational behaviour of D-Clean programs as well as to introduce channels.

For list we have equivalencies like for instance `map f o map g` ≡ `map (f o g)` [2]. The denotational semantics of both languages constructs are equal, which implies that for all functions `f` and `g` and all argument lists these expressions produce the same result. Nevertheless, there is an operational difference since `map f o map g` produces an intermediate list which is omitted in `map (f o g)`. Hence, the later expression is expected to be operationally more efficient.

Similarly to this list transformation we can change the number of functions operating on channels by the transformation:

```
DApply1 (F "f" f) >>= DApply1 (F "g" g) ≡ DApply1 (F "f" (g o f))
```

This effect can also be achieved when we apply different functions to the channels by

```
        DApplyN [F "f1" f1, .., F "fn" fn]
     >>= DApplyN [F "g1" g1, .., F "gn" gn]
≡ DApplyN [F "fg1" (g1 o f1), .., F "fgn" (gn o fn)]
```

In a similar fashion we can transform combinations of `DApplyN` and `DApply` into a single application of `DApplyN`.

```
        DApplyN [F "f1" f1, .., F "fn" fn]
     >>= DApply1 (F "g" g)
≡ DApplyN [F "fg1" (g o f1), .., F "fgn" (g o fn)]
```

and

```
        DApply1 (F "f" f)
     >>= DApplyN [F "g1" g1, .., F "gn" gn]
≡ DApplyN [F "fg1" (g1 o f), .., F "fgn" (gn o f)]
```

The last transformation we show here covers the lifting of an ordinary list processing function in Clean to a channel processing construct over `n` channels in D-Clean.

```
map f list
```

≡

```
    DDivideD (divide n) (0,list)
>>= DApply1 (F "f" f)
>>= DMerge id
```

More transformations are under development and will be included in the final version of this paper.

4 Properties

The transformations from section 3 should be semantics preserving under the semantics from section 2. We plan to test this using the model-based test tool G∀st [5]. Apart from the generation of functions as introduced in [4], we need to check the strictness properties of the constructs. Since the handling of nonterminating expressions in G∀st is hard, we have to find a way to handle this. We plan to handle this by a non-functional extension of G∀st that is able to count the reduction of subexpressions.

5 Conclusions

In this paper we had given an executable semantics for the D-Clean extension of Clean. The functional description of the primitives enables to understand what is computed and how the computation is done in the distributed system of the D-Clean. The transformations defined can be used to change the amount of work on the computation boxes and communication channels. Finally we formulated properties of D-Clean programs.

References

1. Best, E., Hopkins, R. P.: B(PN)^2 - a Basic Petri Net Programming Notation, In: Bode, A., Reeve, M., Wolf, G. (Eds.): *Parallel Architectures and Languages Europe*, 5th International PARLE Conference, PARLE'93, Proceedings, Munich, Germany, June 14-17, 1993, Springer Verlag, LNCS Vol. 694, pp. 379-390.
2. R. Bird. Introduction to functional programming using Haskell (second edition). Prentice Hall, 1998. ISBN 0-13-484346-0.
3. Z. Horváth, Z. Hernyák, and V. Zsók. Coordination language for distributed clean. *Acta Cybernetica*, 17(2):247–271, 2005.
4. P. Koopman and R. Plasmeijer. Automatic testing of higher order functions. In N. Kobayashi (ed.) *Proceedings of the 4th Asian Symposium on Programming Languages and Systems* APLAS'06, volume 4279 of *LNCS*, pages 148–164, Sydney, Australia, 8-10, Nov. 2006. Springer-Verlag.
5. P. Koopman and R. Plasmeijer. Fully automatic testing with functions as specifications. In *Central European Functional Programming School*, volume 4164 of *LNCS*, pages 35–61, Budapest, Hungary, 4-16, July 2006. Springer-Verlag.
6. R. Plasmeijer and M. van Eekelen. *Concurrent Clean language report (version 2.0)*, Dec. 2001. `http://www.cs.ru.nl/ clean/`.

7. V. Zsók, Z. Hernyák, and Z. Horváth. Designing distributed computational skeletons in D-Clean and D-Box. In *Central European Functional Programming School*, volume 4164 of *LNCS*, pages 223–256, 2006.
8. V. Zsók, Z. Hernyák, and Z. Horváth. Improving the Distributed Elementwise Processing Implementation in D-Clean. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools* SPLST 2007, Dobogókö, Hungary, June 14-16, 2007, pp. 256-264, Eötvös University Press, 2007.

Dependency Graphs for Parallelizing Erlang Programs *

Melinda Tóth, István Bozó, Zoltán Horváth, Atilla Erdődi

Eötvös Loránd University, Budapest, Hungary
{toth_m,bozo_i,hz}@inf.elte.hu,erdodi@elte.hu

Abstract. Most of the processors deploy multiple cores to achieve power efficiency and high performance, but even in case of functional languages supporting parallel and distributed evaluation, program need to be structured an appropriate way to take the advantages of these multi-core processors. Our research focuses on determining possible parallelization of programs written in a dynamically typed functional programming language, Erlang, which has special language elements, library support and scheduler logic for parallel execution. A high abstraction level of program representation can be used to determine those program parts, that can be computed parallel in an efficient way and to identify bottlenecks. We apply dependency graphs calculated by static analysis, which graphs based on that control and data dependencies of Erlang programs which influence parallel evaluation.

1 Introduction

A number of legacy program have not been structured to run parallel, thus they can not run efficiently on multi-core architectures. Various approaches have been proposed to improve performance for sequential programs on multiple core systems, our among them focuses on Erlang [4] programs. The Erlang virtual machine (VM) has had a symmetric multiprocessing (SMP) support since 2006, so we try to determine those program parts that can be execute parallel efficiently in the Erlang VM.

The first industrial success with using SMP in Erlang systems was Ericsson's Telephony Gateway Controller [20]. Migrating the project into a dual-core processor produced an impressive 1.7x increase in performance. The migration took less than one man-year including the testing, which is a very short time for a project of this complexity.

Our goal is to provide tool support for developers to further reduce the time necessary for migrating legacy applications to multi-core systems.

To identify the various parallelisable components in the program we have to identify both control and data dependencies between the expressions of the program. That can be easily represented using a directed dependency graph, where the Erlang expressions represented as nodes and dependencies between

* Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary

the expressions as directed edges. Using the DG we can identify the parallelisable components by computing strongly connected components [17] in the graph.

The paper is structured as follows. Section 2 provides a description of Erlang Multi-Core. Section 3 introduces the different dependency relations among Erlang expressions. Section 4 shows an example to demonstrate parallelization. Finally, Section 5 describes related work, and Section 6 is a conclusion.

2 Introduction to Erlang multicore

Erlang is a general purpose functional programming language originally developed by Ericsson to develop fault tolerant telecom applications. Its dynamic type system allows rapid prototyping and hot code swapping. Processes in Erlang are lightweight and handled by the Virtual Machine.

The fist prototype for a multi-threaded Erlang/VM implementation was presented by Pekka Hedquist in his Master's thesis in 1998 [9]. Since there was not demand for multi-core support in the industry at the time, the work for multi-core implementation was only restored in 2005. Erlang/OTP R11B with official SMP support was released in May 2006 [12].

In the current implementation (R14B) each scheduler runs on one operating system level thread with an own run queue. Load balancing among the schedulers is done using a migration logic [13].

Due to Erlang's lightweight processes and message passing concurrency model, software developed in Erlang for single core processors generally scale well on multi core processors. However there are several factors to consider on migration.

There might be bottlenecks processes in the system that should be identified and removed. When a single process provides some sort of functionality used by many other processes, the increasing number of cores is also increase the workload of the singleton process. This is not just causing a performance hit on the system but the whole system can crash after a while when a process cannot handle the requests and its mailbox become flooded.

Erlang comes with a built-in term storage called Erlang Term Storage (ETS). Arbitrary Erlang terms can be stored in ETS tables implemented as hash tables. Tables can be set to public and therefore be accessed by multiple processes simultaneously. This in a sense a form shared memory and introduces the known related problems. ETS tables are heavily used by most Erlang applications.

All shared data structure is protected by locks. If lock-conflicts occurs too often during the operation of the system, it causes performance degradation. Recent improvements in the Erlang VM reduced this overhead by fine grade locking, but the programmer still needs to take the effect of lock-conflicts into account.

When splitting a process into multiple processes race conditions can also be introduced which might be hidden on a single core machine due to deterministic scheduling, but occurs on true multiprocessing systems [20]. Dialyzer, a lightweight static analyzer tool for Erlang, can detect some of the introduced race conditions [2].

3 Dependency graphs for Erlang

To calculate the dependency we use the Semantic Program Graph of RefactorErl [10, 11]. RefactorErl is a source code transformer and static analyzer tool for Erlang. The information retrieval is efficient from that graph, so using the SPG is more efficient than calculating the dependency from the source code.

The DG contains data and control dependencies. To build the control dependency graph we have to define the control-flow graph (CFG) of Erlang programs. From the CFG we can build a Post Dominator Tree (PDT) ([16]). The control dependency graph (CDG) can be built from the CFG and the PDT. The CDG eliminates the unnecessary sequencing from the control flow graph, thus only the relevant control dependencies are stored in the CDG. That makes possible to efficiently find the parallelisable component of the graph. If we do not consider that data dependency can occur among the components, then we have to synchronize the data during the parallelization. Thus we introduce a communication overhead and lose efficiency.

Therefore we build the DG containing both data and control dependency information. According to the nature of the multi-core Erlang we have to consider some kinds of hidden data dependencies. The usage of the ETS tables is one of them. We should group those statements from the program to one parallel component that use the same ETS table or tables. That can reduce the number of possibly lock-conflict in the Erlang VM.

There are expressions or expression sequences in Erlang programs which can not be evaluated in parallel (or the necessary synchronization among them is too costly). That statements have data or control dependency among them. Some of them are easily detectable, for example the head of a case expression can not be evaluated parallel with the body of the case expression, because the executed branch of the case depend on the return value of the head. A more Erlang specific example is that we can not run parallel two expression which can read and write the same ETS table. This kind of data dependencies is detectable in case of named tables, otherwise we need some kind of process identifier (Pid) analysis.

The candidates for parallel execution are the strongly connected components in the graph. Further analysis is needed to decide whether the result is appropriate or efficient. In case of the components become large, then an iterative splitting of that component could produce a more applicable result.

4 An example to demonstrate parallelization

Consider the code of a singleton process with dummy function calls shown on Figure 1.

For each request the reply is generated by the functions, `do_computation/0`, `do_computation_ets_1/1` and `do_computation_ets_2/1`. Lets assume that there is only one side effect present in `do_computation_ets_X/1` and `do_computation_ets_2/1`, that is reading data form the ets table `EtsTable`. Besides `do_computation/0` either does not have a side effect or have a side effect that is not related to `EtsTable`. In this case, the first two function calls

```
loop(EtsTable) ->
   receive
       {Pid, Request} ->
           Reply = handle_request(Request),
           Pid ! {reply, handle_request(Request, EtsTable)}
   end,
   loop(EtsTable).

handle_request(Request, EtsTable) ->
   update_ets_table(Request, EtsTable),
   Data1 = do_computation_ets_1(EtsTable),
   Data2 = do_computation_ets_2(EtsTable)
   Data3 = do_computation(),
   {{Data1, Data2}, Data3}.
```

Fig. 1. Example code of a simple server process

and the third function call can be evaluated in parallel. Of course, this is only meaningful if the delegated functions are complex enough, that the gains by the parallel evaluations are higher than the overhead caused by process spawn and message passing.

Using the erlang module `rpc` for delagating evaluation to separate processes, the transformed code is shown at Figure 2.

```
handle_request(Request, EtsTable) ->
   update_ets_table(Request, EtsTable),
   Data1_key =
       rpc:async_call(node(), ?MODULE, do_computation_ets, [EtsTable]),
   Data2_key =
       rpc:async_call(node(), ?MODULE, do_computation, []),
   Data12 = rpc:yield(ProcessedData1_key),
   Data3 = rpc:yield(ProcessedData2_key),
   {Data12, Data3}.

do_computation_ets(EtsTable) ->
   Data1 = do_computation_ets_1(EtsTable),
   Data2 = do_computation_ets_2(EtsTable),
   {Data1, Data2}.
```

Fig. 2. Parallelized code of the request handling part

The scratch of the DG of the `handle_request/2` function is shown on Figure 3. The dotted edges represent the first component, the dashed edge represents the second component.

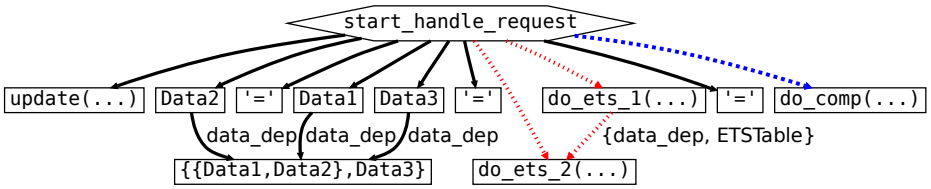


Fig. 3. Dependency Graph for `handle_request/2`

Exceptions thrown by the delegated functions are caught by the helper functions and returned by `rpc:yield/1`. Therefore the semantics of the code two code snippets are not identical which should be considered by the developer [20].

As a counterexample, parallelizing the code by delegating the whole request handling onto a separate process as show in Figure 4 is not possible.

```
loop(EtsTable) ->
   receive
       {Pid, Request} ->
           spawn(fun() ->
                         Reply = handle_request(Request),
                         Pid ! {reply, Reply}
                 end)
   end,
   loop(EtsTable).
```

Fig. 4. Invalid transformation of the server code

Since each request may update the ets table and also reads of it, the decisions made on handling a request might depend on the previous requests.

5 Related work

Dependency graphs are originally designed and used in compilers to prevent statement execution in wrong order i.e. the order that changes the meaning of the program [16].

Nowadays the usage of different dependency graphs is a commonly used technology in different software engineering tasks (in program understanding, maintenance, debugging, testing, differencing, specialization, re-engineering, optimization, parallelization, anomaly detection, etc).

The parallelization problem is a special usage of the dependency graphs. This kind of optimization is studied in different point of view. The paper [5] presents

a dependency representation useful in an optimizing compiler for a vector or parallel machine. Methods for parallelization of Prolog programs and related problems are presented in [19] and [8].

In functional programming languages there are researches for partitioning functional programs [18] that explores pottential paralelism in the source code. That paper describes an intermediate representation for an implicitly parallel functional programming language SISAL [15] and its use for compile-time partitioning.

In case of functional languages most of the researches concentrated on explicit parallelism and its efficient implementation and execution on multi-core systems [14, 6, 1].

The paper [7] introduces a data parallel, functional array processing language SAC and its concept of generic, compositional array programming. The paper presents some tasks in compiling the SAC code into efficient code for modern multi-core processors.

Efficiency in case of running Erlang programs on multi-core systems also studied [21]. The paper [3] describes race condition detection according to the nature of the Erlang programs running on multi-core systems.

6 Conclusions and future work

A number of legacy Erlang program have not been structured to run parallel, thus they can not run efficiently on multi-core architectures. Migrating an application manually is a time consuming process. Our goal is to provide a tool that helps the developers to determine the parallelisable components of the system, thus they can migrate legacy Erlang applications to efficiently executable programs on multi-core systems.

In our dependency graph representation we try to consider the special language elements and library support of Erlang. While we select the parallelisable parts of the program we also try to consider the scheduler logic for parallel execution of the Erlang runtime system.

The mentioned static analysis allows the programmers to transform the sequential source code to an explicitly parallel code in a way that it should be performed by a compiler in a language that supports implicit parallelism. With such an analyzer tool the programmer can supervise the suggested interpretation of the sequential source code into a parallel code, thus make the debugging easier and avoid arbitrary decisions of the compiler that can result in a less than optimal code.

As RefactorErl is also a source code transformer, we want to implement automatic transformation of the source code according to the result of the studied static analysis. Total semantic equivalence however will not be possible due to the limitations of static analysis. Only a limited subset of side effects could be handled and the transformation will not preserve exception semantics. Therefore interaction with the developer will be necessary to decide the applicability of the refactoring.

References

1. A. Al Zain, J. Berthold, K. Hammond, P. Trinder, G. Michaelson, and M. Aswad. Low-Pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on MultiCore Architectures. In *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP'09, Savannah, Georgia, USA*, 2009.
2. M. Christakis and K. Sagonas. Static Detection of Race Conditions in Erlang. In M. Carro and R. Pena, editors, *Practical Aspects of Declarative Languages (PADL'2010)*, volume 5937 of *LNCS*, pages 119–133. Springer, Jan. 2010.
3. K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, ICFP '09*, pages 149–160, New York, NY, USA, 2009.
4. Ericsson AB. *Erlang Reference Manual.* Latest version available online at http://www.erlang.org/doc/reference_manual/part_frame.html.
5. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, pages 9(3):319–349, 1987.
6. M. Fluet, L. Bergstrom, N. Ford, and M. Rainey. Programming in Manticore, a Heterogenous Parallel Functional Language. In *Proceeding of the Central European Functional Programming Summer School, CEFP '09, Komárom, Slovakia*, 2009.
7. C. Grelck and S.-B. Scholz. SAC: off-the-shelf support for data-parallelism on multicores. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France*, pages 25–33, 2007.
8. G. Gupta, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23:2001, 1995.
9. P. Hedqvist. A Parallel and Multithreaded ERLANG Implementation. Technical report, 1998.
10. Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, M. Tóth, I. Bozó, and R. Király. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babe-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, Jul 2009.
11. Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, and R. Király. Building a refactoring tool for Erlang. In *Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008*, Paphos, Cyprus, Jul 2008.
12. K. Lundin. Inside the Erlang VM with focus on SMP, 2008. Presented at the Erlang User Conference, Stockholm. This conference did not have formal proceedings.
13. K. Lundin. About Erlang/OTP and Multi-core performance in particular, 2009. Presented at the Erlang Factory, London. This conference did not have formal proceedings.
14. S. Marlow, S. P. Jones, and S. Singh. Runtime Support for Multicore Haskell. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09, Edinburgh, Scotland*, 2009.
15. J. e. a. McGraw. SISAL Streams and Iteration in a Single Assignment Language. Language Reference Manual, Version 1.2. M-146, 1985.

16. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kauffmann Publishers, 1997.
17. G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.
18. V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SZGPLAN Compiler Construction Conference*, 1986.
19. D. C. Sehr. Automatic Parallelization Of Prolog Programs, 1992.
20. U. Wiger. Erlang Programming for Multi-core. In L. Petersen and M. M. T. Chakravarty, editors, *DAMP*. ACM, 2009.
21. U. Wiger. Erlang Programming for Multi-core, QCON London, 2009.

Counter Automata for Parameterised Timing Analysis of Box-Based Systems

Christoph A. Herrmann[1] and Kevin Hammond[1]

University of St Andrews, Scotland KY16 9SX, UK,
{ch,kh}@cs.st-andrews.ac.uk,
WWW home page: http://www.cs.st-andrews.ac.uk/~ch

Abstract. We present a new approach for the compositional parameterised resource analysis of purely-functional box-based systems. In particular, we are interested in the worst-case execution time between an external input event and the associated output event. Each box, when receiving data on some incoming wires can, according to a set of matching rules, perform a computation and send data on some of its outgoing wires. This model is implemented with the language Hume [2], targeting embedded and safety-critical systems.
The work about compositional analysis of boxes presented in this paper abstracts from the particular execution schema of Hume, by describing potential execution orders in terms of a counter automata, finite state automata with additional counters representing sizes and resource costs and controlling repetitions.
Our approach should be applicable to a variety of systems consisting of purely functional boxes. Nevertheless, we will explain how we perform our abstractions, the effect this has for a small Hume program and how we combine the analysis results of the functional parts with the parameterised analysis results gained from the counter automata.
Our special contribution is that we can deal with systems which are scheduled according to constraints on changing parameters and in which the cost formulae are non-linear. The progress to our previous work [5] is in the reduction of computational complexity and the ability to deal with unknown control parameters modified in cycles in the system. Our method has been applied in the analysis of autonomous vehicle control software with the aim for a resource-aware reconfiguration [3].

1 Introduction

We present a formalism for the compositional modelling and analysis of cooperating computational boxes that is capable of dealing with repetitive executions of boxes in terms of unknown control parameters. In particular, we are interested in analysis of upper bounds for resource consumption like worst-case execution time. Our work is motivated by the analysis of programs in Hume [2], a language whose functional parts are known to be suited for a formally based analsis of safety-critical systems due to a tight coupling of resource consumption to the operational semantics. In this paper, we extend the capabilities to the analysis

at the Hume coordination level, i.e. the interactions of boxes and their repetitive execution controlled by unknown external parameters.

Our analysis separates the purely functional parts inside the boxes from the box interactions which is the main topic of this paper. From the ensemble of boxes inside a Hume program we construct a counter automaton, a finite state automaton which keeps track on control parameters, iteration counters and resource cost by additional variables, so-called counters, which play a crucial role in the analysis. A soon as a counter automaton has been constructed for a Hume program or a program in another purely functional box-based language, we do not look inside the boxes any more. The automaton is based on available knowledge of how the abstract data used in the static analysis is transformed and about the resource usage in terms of the input patterns. It is irrelevant whether the inside of the box is implemented in hardware or software, provided it adheres to an agreed protocol of the coordination language. However, if we wanted to interface to the actual Hume runtime environment, the encoding of data on the wires connecting boxes and the conditions for execution of boxes must be respected.

Currently, we gain the knowledge of how the boxes transform static information by manual construction of an abstract prototype in Haskell which is then subject to a symbolic analysis. This is sufficient to gain confidence about the capabilities of our analysis method. However, at a later stage we aim for an automatic extraction of this information from source programs in Hume or another box-based language.

The knowledge about worst-case execution time for particular input patterns is obtained using an amortised analysis of the functional language layer of Hume [8]. It is itself based on machine-level analysis for the instructions of the compiled byte code of Hume [4]. If another language is to be used for implementing the box functionality, an appropriate analysis of programs of that language would be necessary. Likewise, if a box represented a hardware component, the abstract behaviour of this component needs to be known.

In the next section, we describe the parts of the Hume language which are relevant to understand the need for a concise formalism such as counter automata, and the abstractions that need to be applied before. The reader who does not need a motivation and is only interested in the counter automata analysis can skip this section safely. Section 3 introduces counter automata and how we use them to derive parameterised cost formulae. Section 4 applies our technique to a larger example at the coordination level. Section 5 concludes and outlines potential further research.

2 Background: Hume

2.1 The Hume Box Execution Model

Hume programs consist of compositions of boxes, which are activated by a scheduler and exchange data via a static network of wires. Each wire connects an

output port of a box with an input port of another or the same box, and can buffer at most one data object. However, such a data object can be composed of vectors, tuples etc. If a box cannot release results because one of the values on the wires has not been consumed yet, these results remain in the box heap and the box is blocked from execution until the values are released. If a box is not blocked, it can execute if at least one particular input pattern matches, thereby arguments from input ports can be ignored, and the availability of such ignored values is not required for a successful match and box execution. The patterns are specified in the program part for the box, using the symbol $*$ for ignored inputs. Each pattern forms the left-hand side of a purely-functional computational rule, and when it matches, the values on the input ports are assigned to program variables and used in the evaluation of the rule's right-hand side, which determines the values of the output ports. The programmer can use the symbol $*$ to specify that no value is produced for a particular output port. The right-hand side expressions can apply pure functions, which are potentially recursive, potentially higher-order. The programmer can define her own data types and constructors which are then associated with individual execution time costs by an amortised analysis of the box execution [5]. The analysis of Hume box compositions presented here builds on this amortised or other kind of analysis, i.e., assumes that we already can obtain the execution time of each rule of each box. In particular, we are interested in the execution time for a particular class of tasks, starting from a set of input events from the environment, involving several box executions, and finishing with the creation of the last output event.

2.2 The Superstep Scheduling Schema

Several alternative scheduling orders are possible for Hume boxes, e.g., to allow for efficient parallelisation. However, any legitimate schedule be consistent with the denotational Hume semantics. One order which can be easily understood from an operational perspective is the *superstep scheduling* mechanism. The idea of a superstep in Hume is similar to that in the bulk-synchronous parallel programming (BSP) model [9]: within each superstep each box is executed at most once and the data that a box produces is not available to the consumer before the next superstep. It follows that within a superstep any execution order and any potential parallelisation of box executions leads to the same behaviour. However, this only holds within a single superstep, and not across several supersteps. This means that we can view all box executions within a step as being semantically independent of each other, i.e., forming part of a function which maps the system state at the beginning of a superstep to the state at the end.

Figure 1 depicts a system with two Hume boxes named P and Q and three wires labeled x, y and z. Without further knowledge of the program for the boxes, Box Q can react to either or both available inputs on y and z and Box P can be blocked from further execution as long as it wants to release a value onto wire y but cannot do so because Q has not consumed the previous value on y yet.

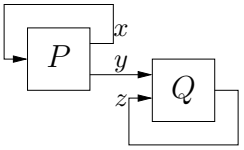


Fig. 1. Two Boxes

In the model, we treat availability of wire values explicitly. Figure 2 shows the instance of our model for a scheduling cycle of the two box composition. Because of the superstep semantics assertion of outputs does not have an impact on executions in the same step. This is established by dividing each superstep into two phases, A and B.

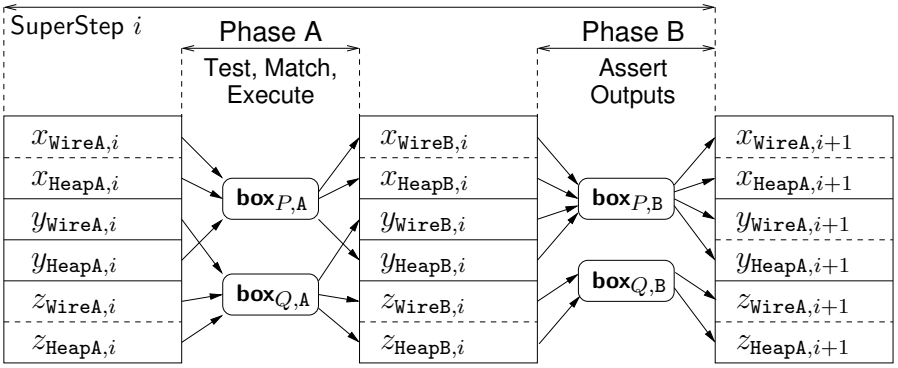


Fig. 2. Hume Superstep as a Mapping Between Wire Vectors

We view each scheduling cycle as a mapping between vectors of wire and heap values. These values are indexed by the location (`Wire`/`Heap`), the scheduling cycle (i) and the phase (`A`/`B`). The location `Wire` means that the value is ready to be consumed while `Heap` means that it resides as part of a result in the heap of a box and has not yet been released on the wire. The scheduling cycle and the distinction into phases `A` and `B` express the development of a value over time and permit to express the simulation purely functional.

Each kind of box contributes to the entire function with a single function for each of the two phases, e.g. $\texttt{box}_{P,\texttt{A}}$ and $\texttt{box}_{P,\texttt{B}}$ for Box P. Non-consumption of a value is implemented by assigning the wire value of the `A` vector to the corresponding wire value of the `B` vector, e.g., when Box Q does not consume y, function $\texttt{box}_{Q,\texttt{A}}$ copies the value $y_{\texttt{WireA},i}$ to $y_{\texttt{WireB},i}$ and this prevents $\texttt{box}_{P,\texttt{B}}$ from asserting the heap value $y_{\texttt{HeapB},i}$ to $y_{\texttt{WireA},i+1}$. Note that we do not need extra state information for keeping track on whether a box is blocked; this can be deduced from the fact that values still reside in the box heap. However, other issues which we do not deal with in this paper, like rule reordering to achieve fair merging of inputs could be regarded by adding information about the current rule ordering of each box into the wire vector to form a more general state vector.

We can describe the semantics of each box in Haskell by calculating concrete values for the heap and wire values. In the sequential scheduling order the timing information is passed in a daisy-chain fashion through all box functions which update it. Since we are interested in an analysis which covers all possible cases instead of dealing with single inputs, the calculation is carried out using symbolic expressions representated by an algebraic data type in Haskell. E.g., a box might

transform an input named `x` into an output such as `x:+:(C(-1))` (decrement of `x`) or `x:*:x` (`x` squared).

2.3 Small Example

The Hume example we have chosen here is part of an autonomous vehicle control application in a scenario in which a set of autonomous platforms (`ps`) are tracking targets (`ts`). The example calculates which targets are reachable from a particular platform. Although it only comprises one box, the use of a feedback wire to carry information from one iteration of the box to the next addresses the aspect of, in general, major difficulty of the analysis, which is control-dependent repetition.

We show below the Hume code for the computation of all reachable targets from a given platform. The selection of the target is done by iterating over the vector index `whichT`. The result list is accumulated in the second component of the first argument.

```
addReachables :: t_pInfo -> t_int -> t_tAllInfo -> t_pInfo;
addReachables p whichT ts =
 if whichT==0
  then p
  else
   case (p,(ts@whichT)::t_tInfo) of
    ((pPos,pReachables,pInfCon,pCost),
     (tPos,tClash,tPlatNo,tSearchComp,tProfile,tPlatClashes,tRedTClashes))
         -> let newP = if isInRange tPos pPos
                       then (pPos,
                             whichT : pReachables,
                             calcConstraint tPos pPos,
                             calcMetric tPos pPos)
                       else p
            in addReachables newP (whichT-1) ts;
```

The algorithm we have chosen to use has complexity O(#ps*#ts), where # denotes the length of a vector, that is, it is linear only if one of these values is fixed. Since the amortised analysis at the functional level can currently only deal with linear cost expressions, we have to lay out one of the nested iterations at the coordination level by using a box with a feedback wire. Box `collectAllReachables` shown below iterates over the index of the vector of platforms, named `n`. The box has two rules `A` and `B`, covering the normal and initial cases, respectively. An asterisk (`*`) in an input position indicates that this input value is not required and remains on the wire, if present; an asterisk in the result means that no output value is produced.

```
box collectAllReachables
in (initReach::t_ptAllInfo, loopReach::t_loopReach)
out (loopReach1::t_loopReach, outReach::t_ptAllInfo)
 match
  {-A-}   (*,(n,ps,ts)) ->
            if n==0
  {-A1-}     then (*,(ps,ts))
  {-A2-}     else let newP = addReachables ((ps@n)::t_pInfo) (length ts) ts;
                          newPs = update ps n newP
                     in ((n-1, newPs, ts), *)
  {-B-} | ((ps,ts),*) -> ((length ps, ps, ts), *);
```

The box output `loopReach1` is linked to the input `loopReach` to form a cycle, and initially has no value. The execution time for single box executions determined by the amortised analysis, excluding in the auxiliary function `addReachables` the times for `isInRange`, `calcConstraint` and `calcMetric` (which are custom-specific) are shown in Table 1. The times are given in machine cycles of the PowerPC processor core 603e.

branch	condition	time (cycles)
B	True	69356
A2	n>0	93731 * #ts + 155547
A1	n$=0$	50717

Table 1. Results of the amortised analysis for single branches

At this point, we know the worst-case executions times for single branches, and these already contain a parameter, but we still do not know the reaction time of the entire system involving several box executions. This is the focus of the rest of this paper.

3 Analysis Using Counter Automata

3.1 Construction of a Counter Automaton

A finite state automaton comprises a fixed number of states, connected by transitions which are labelled with the input token that caused the transition. A counter automaton [1] is a finite state automaton that has been extended to include additional, so-called counter variables. In a counter automaton, the transitions are annotated instead with equations between the values of specific counter variables before the transition (unprimed) and after the transition (primed). E.g. the equation `x=0` means that `x` must be zero if this transition is taken. If `x` is unprimed, this can be seen as a guard, if it is primed it refers to the value of `x` after the transition and can be seen as an assignment. We will therefore also use the alternative notation `x`$\leftarrow$`0` instead of `x'=0` to improve readability.

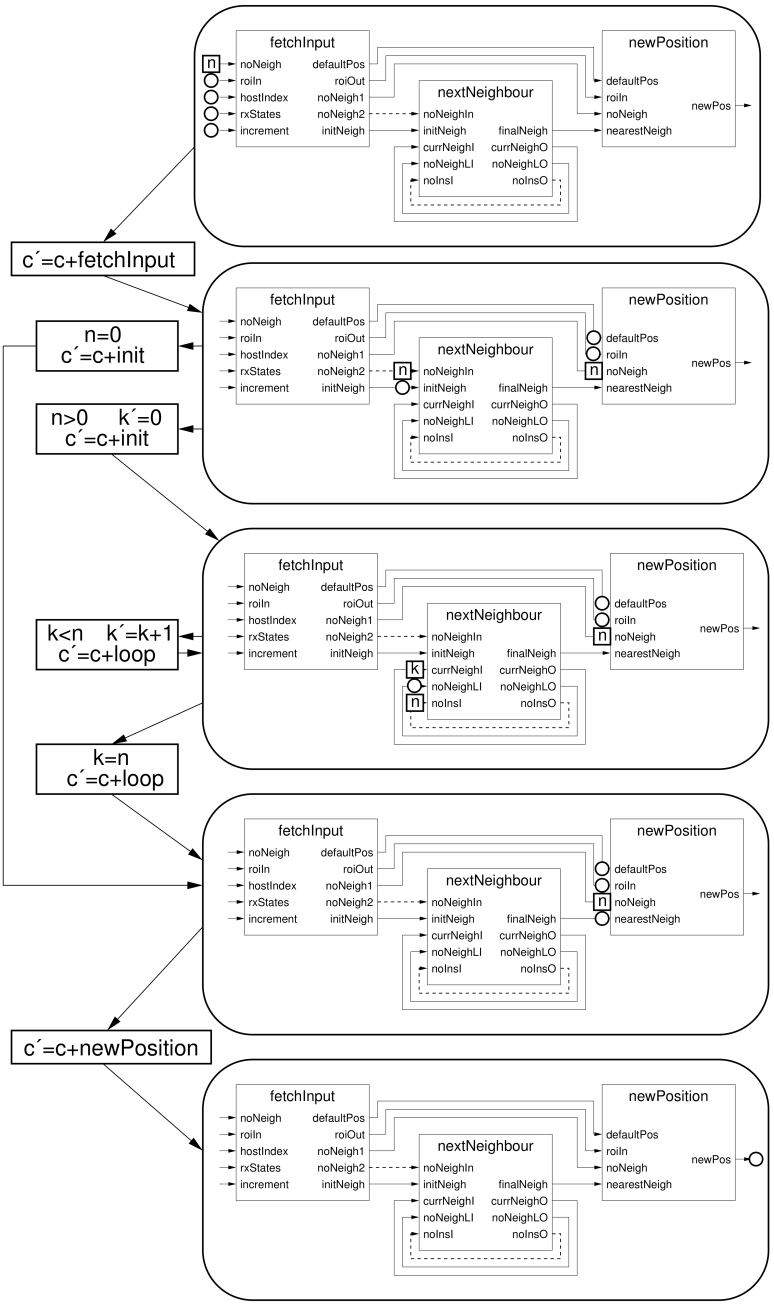


Fig. 3. Counter Automaton from Synbolic Simulation

The purpose of constructing a counter automaton for our system is to reduce the number of states that are needed to cover all the possible valuations of wire vectors over integers and other domains with large numbers of possible values. As stated before, the work of an entire Hume scheduling cycle can be described by a mapping between wire vectors. Since Hume programs are not random but have been developed to establish a particular behaviour, only a few states are usually necessary to describe iterative processes at the coordination level. We believe that similar considerations hold for other systems. For our purpose, the counters will be used to represent iteration counts, loop bounds and resource costs, e.g., worst-case execution time.

In Figure 3 we illustrate how a counter automaton is constructed. We have a system of three boxes in which the box `nextNeighbour` has a feedback wire and is repeatedly executed until the iteration counter `k` reaches the value `n`. Each state is labelled with the same Hume box composition but with different states of wire vectors. We obtain the states and transitions by a symbolic interpretation (generalisation of profiling) of an abstract prototype of the system.

Only these five states should occur in a consistent operation of the group of boxes. The wire allocations in each state are depicted either by circles or by squares with a parameter. A circle means that only the presence of a value is important, whereas a parameter indicates that the value of the parameter is also significant. In order to simplify the treatment here, this value will always be a natural number. In addition to dealing with control parameters, counters are also used to keep track of resource costs, as with the variable `c` here. The important thing is that once the annotated transitions have been constructed, the analysis does not need to look insides the states any more. This can significantly reduce the combinatorial complexity.

3.2 Analysis Using a Counter Automaton

In order to explain how analysis results from the functional and coordination level are combined, we take an example with a single Hume box. The implementation of this box has been described in Section 2, but here we only need the results of the amortised analysis from Table 1. Figure 4 shows the counter automaton for an execution sequence of this system, where `*` represents empty wires.

The system starts with data on the first wire only. Branch `B` of the box moves this data on the feedback wire and initialises the counter `n` with the number of platforms. As long as `n` has not reached 0, Branch `A2` updates the information for a particular platform and decrements `n`. At the end, `n` reaches 0 and Branch `A1` moves the updated platform information to the outgoing wire. The cost transformation could be stated in the transitions as well, using a counter variable for costs and depending on other counter values as well. Once this automaton has been constructed, no further inspection of the Hume program is necessary to carry out the compositional box analysis.

Every language recognised by a finite state automaton can be described by a regular expression, describing the set of words accepted by the automaton. Our

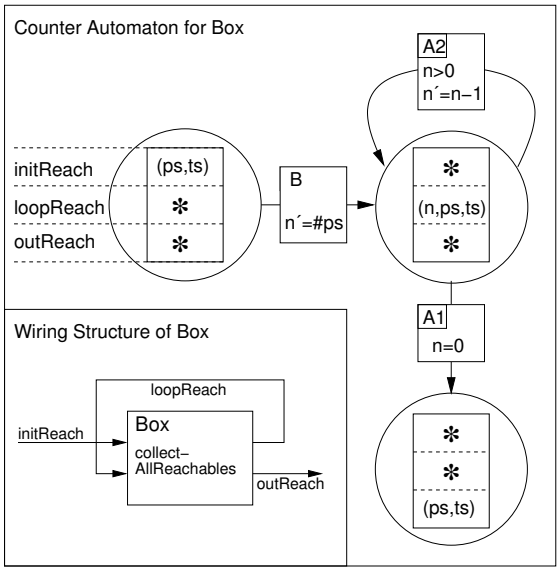


Fig. 4. Counter Automaton for Single Box with Feedback Loop

idea is that the alphabet of the formal language consists of the box rule executions, so each word describes a particular schedule. For our resource analysis we are not interested in arbitrary number of repetitions but in particular numbers, depending on parameters. Therefore, instead of having Kleene stars to express repetitions we would need arithmetic expressions for exponents. The dual on the automaton side are the counter automata [7].

The parameterised expression for the automaton in Figure 3 would be, in simplified form, $BA_2^{\#ps}A_1$, and the cost = cost(B)+#ps*cost(A_2)+cost(A_1), instantiated with results from the single box analysis: (93731*#ts+155547)*#ps+120073. Since it multiplies the values of two parameters `#ts` and `#ps`, this expression is not linear. Given a number of platforms and a limit on the required response time it is now possible to calculate the number of targets that can be tracked.

Instead of constructing a particular parameterised expression for the box branches explicitly, we simply add another counter for the cost, and increment it in each transition either by a constant number or by a variable if the value is unavailable or we want the result uninstantiated. Variables in cost expressions are uncritical for the analysis, since they should not have an influence on the function of the system. If the system function is designed to depend on time, even in form of a timeout exception, time should be made an explicit part of the abstraction of the program.

3.3 Analysis Algorithm

The idea of our algorithm stems from the direct transformation of finite state automata into regular expressions based on generalised finite state automata. These automata have transitions annotated with regular expressions instead of just input symbols, and the algorithm works by stepwise elimination of states while preserving the recognised language by updating the existing transitions with compositions of the expressions to/from the eliminated state, including loops in this state.

Counter automata are simply spoken finite state automata with additional counters and guards on them. Input symbols can be encoded within the counters and so we can assume that all transitions can be performed without consuming any input. Our algorithm also eliminates states successively, but instead of constructing a regular expression it combines constraints on the counters from the transitions to/from the eliminated state and regarding transitions from the eliminated state to itself.

Cost expressions for sequences of transitions can be easily combined by substitution, like the Hoare rule for sequence. The challenges are loops and in particular dependencies between variables. Fortunately, our approach only needs to deal with direct loops at each elimination step. We then need to factorise the set of counter variables into those which are (1) decremented by one, (2) incremented by an expression using a variable from (1) and other variables not modified inside the loop and (3) variables assigned loop-invariant expressions. When the loop is eliminated, we treat them as follows: Kind (1) is used for the exit condition and is thus assigned the final value 0 in this step. Kind (2) is assigned the symbolic sum in the Kind (1) variable. Kind (3) is assigned the loop-invariant value once.

Symbolic summation of polynomials is a generalisation of the Gauss formula for successive numbers and a standard procedure in Computer Algebra. We implement it with a small Haskell function.

Expressions are represented by DAGs in the Haskell ST monad to reduce space and time explosion in the ubiquitous presence of common subexpressions, e.g., induced by substitution. A view function expands a specified number of topmost layers of an expression tree from a DAG in order to perform pattern matching, e.g., to classify variables according to their effect in loops.

4 Larger Application Example

Our example is part of the reconfiguration algorithm for an autonomous system. Cooperating platforms are tracking targets, taking into account constraints about distance and energy consumption. The part we had been analysing is the assignment of platforms to targets, which consists of several nested iterations over targets and platforms. The abstraction of this algorithm leads to a counter automaton with 7 states, depicted in Figure 5.

It is represented below as a Haskell array. Each array entry is a list of triples containing the number of the follow-up state, the guard for the transition and

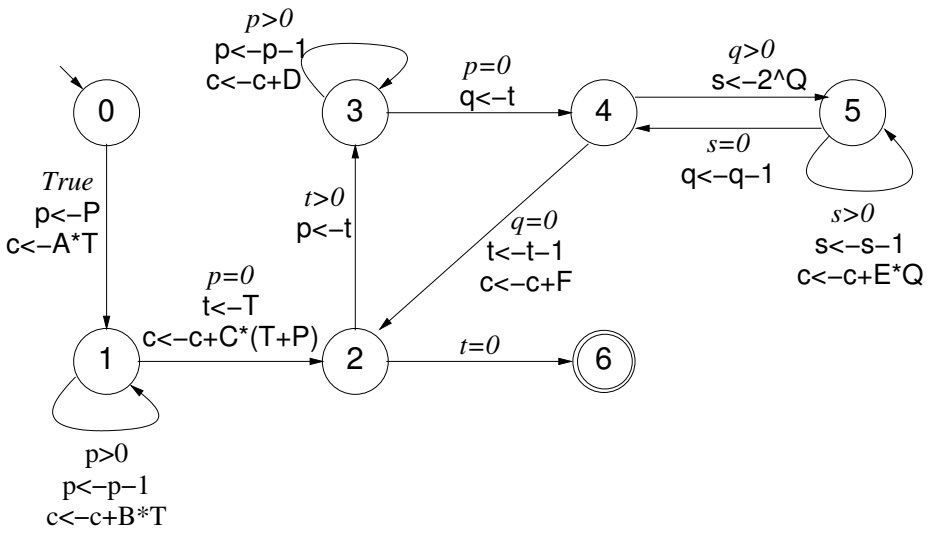


Fig. 5. Counter Automaton for Example

the associate action if the transition is being performed. The start state has the number 0 and final states are all those which do not have successors, here State 6.

```
[(0,[(1,TT, (addCost (v "A" :*: v "T"))++[(Var "p",v "P")])]),
 (1,[(1,C 0 :<: v "p", (addCost (v "B" :*: v "T"))++(decVar "p")),
     (2,C 0 :=: v "p", (addCost (v "C" :*: (v "T" :+: v "P")))
                       ++[(Var "t", v "T")])]),
 (2,[(3,C 0 :<: v "t", [(Var "p",v "t")]),
     (6,C 0 :=: v "t", [])]),
 (3,[(3,C 0 :<: v "p", (addCost (v "D"))++(decVar "p")),
     (4,C 0 :=: v "p", [(Var "q",v "t")])]),
 (4,[(5,C 0 :<: v "q", [(Var "s",v "2^Q")]),
     (2,C 0 :=: v "q", (decVar "t") ++ addCost (v "F"))]),
 (5,[(5,C 0 :<: v "s", (addCost (v "E" :*: v "Q"))++(decVar "s")),
     (4,C 0 :=: v "s", (decVar "q"))]),
 (6,[])]
```

There are three input parameters: the number of platforms (`P`), the number of targets (`T`) and the maximum number of platforms to be regarded to track a target (`Q`), and a derived input parameter (`2^Q`) for 2 to the `Q`, since our approach cannot yet deal with exponentials. There are six cost coefficients for purely functional code: `A`, `B`, `C`, `D`, `E`, `F`. Counters which are modified during execution are written in small letters: `p`, `q`, `t` and `s`. The template `addCost` increments a distinguished cost counter and `decVar` decrements a control counter by one. The list entry `(Var "q",v "t")` means that the counter `q` is assigned

the current value of `t`. Our current implementation requires that counters are decremented towards zero, thus the use of the guards `C 0 :=: v ...` and `C 0 :<: v ...`. Iterations with a fixed stride and upper bound can in principle be reduced to this form by index transformations.

Our analysis delivers the following cost expression which has been verified by manual calculation, where `%` denotes a rational fraction and `**` a power:

```
(1 % 2)*(2^Q)*E*Q*T + (1 % 2)*(2^Q)*E*Q*(T**2)
+ 1*A*T + 1*B*P*T + 1*C*P + 1*C*T + (1 % 2)*D*T
+ (1 % 2)*D*(T**2) + 1*F*T
```

5 Conclusions and Future Work

We have been able to analyse purely functional box-based systems for resource consumption and demonstrated this for the resource worst-case execution time. The choice of resource is not important for our analysis, however we know memory space tends to be simpler and energy consumption to be harder, but this is an issue of box properties, not of the compositional analysis as such. The challenge of our analysis are definitely finding appropriate abstractions of our system and finding closed forms for recursive cost expressions.

We demonstrated in the previous section that we can analyse a system with nested cyles and derive non-linear cost formulae which depend on several control parameters and cost coefficients. Our approach exploits guarded pattern matching available in the language Haskell as well as semantic treatment such as symbolic summation of polynomials.

We have thought about a transformation of the automaton representation into a synthetic small imperative loop language to apply standard analysis methods based on the Hoare calculus, such as the weakest precondition predicate transformer. But we are convinced that finding loop invariants would be more complicated than a mixed pattern / semantics based approach working on the automaton.

Future work will deal with automatic generation of the counter automaton from box-based programs as well as an extension of the range of the analysis to more sophisticated predicates and variable dependencies.

References

1. Marius Bozga, Radu Iosif, and Yassine Laknech. Flat parametric counter automata. In *ICALP 2006: Automata, Languagesvand Programming, 33rd International Colloquium*, Venice/Italy, 2006.
2. Kevin Hammond and Greg J. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proc. Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science 2830, pages 37–56. Springer-Verlag, 2003.

3. Christoph Herrmann and Kevin Hammond. Compositional resource usage analysis of parametric box-based autonomous vehicle systems. In *Proc. 5th SEAS DTC Annual Conference*, pages A8/1–11, Edinburgh, July 2010. Systems Engineering for Autonomous Systems Defence Technology Centre (SEAS DTC).
4. Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
5. Christoph A. Herrmann and Kevin Hammond. Towards compositional worst-case execution time analysis for Hume programs. In *Proc. ERCIM/DECOS Workshop*, 2008.
6. Christoph A. Herrmann and Kevin Hammond. Compositional analysis of hume box executions. In *Draft Proc. of FOPARA'09 Workshop*, 2009.
7. D. Hovland. Regular expressions with numerical constraints and automata with counters. In *Proc. ICT AC 2009,*, Lecture Notes in Computer Science 5684, pages 231–254. Springer Verlag, 2009.
8. Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proc. Principles of Programming Languages (POPL)*, 2010.
9. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Introducing the PilGRIM: a Pipelined Processor for Executing Lazy Functional Languages

Arjan Boeijink, Philip K.F. Hölzenspies, Jan Kuper

University of Twente
Enschede, The Netherlands
{w.a.boeijink,p.k.f.holzenspies,j.kuper}@utwente.nl

Abstract Processor designs specialized for functional languages received very little attention in the past 20 years. The potential for exploiting more parallelism and the developments in hardware technology, ask for renewed investigation of this topic. In this paper we use ideas from modern processor architectures and state of the art in compilation, to guide the design of our processor, the PilGRIM. We define a high-level instruction set for lazy functional languages, and show the processor architecture that can efficiently execute these instructions. Furthermore we show how a little hardware support can significantly improve the memory locality for functional languages.

1 Introduction

The big gap between the functional evaluation model based on graph reduction and the imperative execution model of most processors, has made (efficient) implementation of functional languages a topic of extensive research. Until about 20 years ago, several projects have been undertaken to solve the implementation problem by designing processors specifically for executing functional languages. However, advances in the compilation strategies for conventional hardware and the rapid developments in clock speed of mainstream processor architectures made it very hard for language specific hardware to show convincing benefits.

In the last 20 years the processors in PCs have evolved a lot; the number of transistors of a single core has grown from hundreds of thousands to hundreds of millions following Moore's law and the clock speed has risen from a few MHz to a few GHz. Introduction of deep pipelines, superscalar and out-of-order execution changed the microarchitecture of processors completely. Nowadays, processor designs are limited by power usage instead of transistor count, known as the power wall. Memory latency and bandwidth and the unpredictability of control flow became the bottleneck for processor performance, because they limit the amount of exploitable Instruction Level Parallelism (ILP). This power wall and the complexity of efficiently using more ILP shifted the focus in processor architectures from mainly increasing frequencies to building multicore processors.

Recent work on the Reduceron [9] showed encouraging results on what can be gained in parallelism by designing a processor for a functional language. The changes in hardware technology and the positive results of the Reduceron suggest it is time to evaluate processors for functional languages again in the light of modern hardware architectures.

Why design hardware for functional languages again?

The Reduceron shows there is a lot of low level parallelism in functional languages when there is enough memory bandwidth and the latency problems are bypassed (by low frequency, no pipelining, and single cycle memories). Single thread performance will be critical when all easy parallelism has been fully exploited. A significant part of the hard to parallelize code will be in applications with complex symbolic manipulations, a domain very suitable for functional languages. Functional languages are very data and control flow intensive [10]. First-class functions and laziness make control flow harder to predict for current branch prediction hardware. The amount of indirect jumps and load-branch dependencies in functional languages limits the achievable performance, unless their associated stalls can be avoided more often. Pure functional languages can offer a lots of flexibility in the order of evaluation, but no hardware exists that can effectively exploit the very small granularity of this potential parallelism. Previous research on language specific hardware considered only small additions or changes, because they were constrained by

the limited number of transistors at that time. Language specific modifications tend to add complexity to the hardware, thereby often reducing the clock speed. However, the maximum achievable clock speed is not longer the critical factor for performance. Extensive optimizations in modern hardware make it hard to prove benefits of individual language specific modifications. Lazy functional languages are good candidates to test whether language specific hardware is beneficial, because they differ the most in execution model from hardware of all widely used general purpose languages. Eventual performance gains for specialized pure functional architectures might influence conventional processor architectures, because the trend in mainstream languages is toward more abstraction, first-class functions and immutable data structures.

2 The Pipelined Graph Reduction Instruction Machine (PilGRIM)

The PilGRIM is a processor with a design specialized for executing lazy functional languages. The processor is pipelined deep enough to make a clock frequency of 1GHz is a feasible target. To our knowledge, it is the first deeply pipelined processor for performing graph reduction.

The architecture is derived from modern general purpose architectures, with a 64-bit datapath and using a standard memory hierarchy: separate L1 instruction and data caches, a L2 cache and DDR memory interface. The design targets silicon and is a realistic design for current hardware technology. This requires taking into account actual memory access times, and using basic components limited to sizes that have been used in real chips. The instruction set is designed to exploit benefits of extensive compiler optimizations (using the output of GHC). This is especially important for performance of primitive (arithmetic) operations. The processor executes a high level instruction set which is close to a functional core language, and that allows the code generator to be simple. Optimizations are aimed at improving memory locality and minimizing the effects of L1 cache and pipeline latencies. The processor has hardware support for memory management to minimize the overhead of garbage collection and to reduce the memory bandwidth the core requires.

Due to the complexity of a deeply pipelined design, a synthesizable hardware description for PilGRIM has still to be made. At the moment the PilGRIM project consists of a compiler that takes External Core from GHC as input, and an extensive simulation model of the hardware architecture. This simulation model is not yet complete, because some components still need to be pipelined.

Contributions

- A high-level and coarse-grained instruction set for lazy functional languages, with a simple compilation scheme from a functional core language.
- Design of a hardware architecture that can efficiently execute this instruction set, where this design is made with a deep pipeline in mind.
- Design of a register structure consisting of wide stacks and small queues can support a lot of local data movements, without needing register allocation.
- Techniques to improve the memory locality of data in functional languages, using pointer tagging and hardware supported reference counting.

2.1 Related work

Between roughly 1975 and 1990, a lot of work was done on the design of Lisp machines and combinator based processors. Big differences in implementation strategy and hardware technology leaves little to directly compare this work to. The only processor designs comparable to our work are Augustsson's Big Word Machine (BWM) [1] from 1991 and the recent work by Naylor and Runciman on the Reduceron [8,9]. Our work was inspired by the Reduceron, because of its promising results and the large amount of choices available in design of a lazy functional processor of which the impact is not yet known. The BWM, Reduceron and PilGRIM have in common, that they all focus on exploiting the potential parallelism in data movements inherit to functional languages, by being able to read multiple values in parallel from the stack and rearranging them through a large crossbar in every cycle. Advances in hardware technology allowed the Reduceron and the PilGRIM to go a step further than the BWM, by also using a wide heap memory and adding special purpose stacks that can be used in parallel. Both

the BWM and the Reduceron choose to encode data constructors and case expression in functions for hardware simplicity, and the Reduceron adds special hardware to speed up handling these in function application encoded case expressions. The Reduceron is based on template instantiation, while the BWM uses a small instruction set, based on the G-machine [5]. Unfortunately, the BWM was only simulated and never built. The Reduceron has been implemented on an FPGA, achieving a clock speed close to 100MHz. The Reduceron executes one complete reduction step per cycle. While the Reduceron achieves surprisingly high performance given its simplicity, the single cycle nature of its design will be the limiting factor in performance, because it will have low clock speed even in a silicon implementation. Besides deeply pipelined versus single cycle, the main difference between the PilGRIM and the Reduceron is, that the Reduceron is based on template instantiation and does not have an instruction set. Furthermore the Reduceron is focused on exploiting dynamic optimizations in favor of using extensive compiler optimizations. The current Reduceron design has no support for external memory, thus it does not address the latency and bandwidth issues external memory causes.

3 Instruction set and compilation

Before designing the hardware, we first want to find a suitable evaluation model and an instruction set, because designing hardware and an instruction set in parallel is a too complex task. The design requirements for the instruction set are firstly efficiency and secondly simplicity. We chose to use GHC as frontend, because it is a widely used Haskell compiler with an extensive set of high-level optimizations. By using the External Core feature of GHC, we have a convenient starting point to base a code generator for this new architecture on. While External Core is a much smaller language than Haskell, it is still too complex and abstract for efficient execution in hardware. Before compiling to an instruction set, we transform External Core to a simplified and low-level intermediate language, defined in the next section. The instruction set and assembly language is derived from Graph Reduction Intermediate Notation (GRIN) [3,2] (an intermediate language, in the form of a first-order monadic functional language). GRIN has been chosen as basis, because it is a simple sequential language, that has a small set of instructions and is still close to a functional language. GRIN is often used (and was originally developed) in combination with whole program optimizations. We choose not use whole program optimization, because it results in big code sizes and its scalability to bigger applications remains to be shown.

function definition:

$d ::= f\ x^\star = e$

toplevel expression:

$e ::= s$ *(simple expr.)*

$\mid$ **let** b **in** e *(lazy let expr.)*

$\mid$ **letS** b **in** e *(strict let expr.)*

$\mid$ **case** s **of** $\{\ a^+\ \}$ *(case expr.)*

$\mid$ **if** c **then** e **else** e *(if expr.)*

$\mid$ **fix** $(\lambda r.f\ r\ x^\star)$ *(fixpoint expr.)*†

$\mid$ **try** $f\ x^\star$ **catch** x *(catch expr.)*†

$\mid$ **throw** x *(throw expr.)*

† where f is saturated

$a ::= C\ x^\star \rightarrow e$ *(constructor alternative)*

$b ::= x = s$ *(binding)*

$c ::= \bowtie\ x^\star$ *(primitive comparison)*

simple expression:

$s ::= x$ *(variable)*

$\mid$ n *(integer)*

$\mid$ $C\ x^\star$ *(constructor [application])*

$\mid$ $f\ x^\star$ *(function [application])*

$\mid$ $y\ x^+$ *(variable application)*

$\mid$ $\otimes\ x^\star$ *(primitive operation)*

$\mid$ $\pi_n\ x$ *(projection of a product type)*

Figure1. Grammar of the simple core language

3.1 A simple functional core language

As an intermediate step in the compilation process, we want a language that is both simpler than and more explicit with special and primitive constructs than External Core. The simple core language is (like GRIN) structured in supercombinators, so all lambdas need to be transformed to toplevel functions (lambda lifting). The grammar of the simple core language is given in figure 1. Subexpressions and

arguments are restricted to plain variables, achieved by introducing let expressions for complex subexpressions. All constructors and primitive operations are fully saturated (all arguments applied). From the advanced type system in External Core, the types are simplified to the point where only the distinction between reference and primitive types remains. Strictness is explicit in this core language using a strict let, a strict case scrutinee and strict primitive variables. This core language has primitive (arithmetic) operations and an if-construct for primitive comparison and branching. Exception handling is supported by a try/catch construct and a throw expression. Let expressions are not recursive; for recursive values a fixpoint combinator is used instead. Selection from a product type could be done using a case expression, but is specialized with a projection expression.

3.2 Evaluation model and memory layout

Almost all modern implementations of lazy functional languages use a variation of the G-machine, which introduced compiled graph reduction [5]. The evaluation model and memory layout described this section is derived largely from a mix of GRIN [2] and STG, as used in GHC [12]. The machine model (from GRIN) consists of an environment and a heap. The environment maps variables to values, where values can be either *primitive values* or *heap references*. The environment is implemented as a stack of call frames. The heap maps references to nodes, where a node is a data structure consisting of a tag and zero-or-more arguments (values). Node tags specify the node's type and contain additional metadata, especially a bitmask denoting which of the node's arguments are primitive values and which are references.[1]

Node Type	Represents	Contents
C	Constructor	Constructor arguments
P	Partially applied function	Number of missing arguments and args. already gathered
F	Fully applied function	All arguments for the function

Table1. Node types

Tags distinguish three basic node types (see table 1). Both C- and P-nodes are in Weak Head Normal Form (WHNF), F-nodes are not. Tags for C-nodes contain a unique number for the data type they belong to and the index of the constructor therein. The P- and F-tags contain a pointer to the applied function. The F-nodes are closures, that can be evaluated when required. To implement laziness, they are overwritten (updated) with the result after evaluation.

A program is a set of functions, each resulting in an evaluated node on the stack, i.e. a C- or P-node. Every function has a fixed list of arguments and consists of a sequence of instructions. Next, we will introduce the instruction set, on the level of an assembly language, i.e. with variables as opposed to registers.

3.3 Assembly language

Similar to GRIN, PilGRIM's assembly language can be seen as a first-order, untyped, strict, monadic language. The monad, in this case, is abstract and implicit, i.e. the programmer can not access its internal state, other than through the predefined instructions. This internal state is, in fact, the state of the heap and the stack. Unlike GRIN, the syntax of PilGRIM's assembly language is not based on a "built-in bind structure," but rather on Haskell's do-notation. However, it imposes considerably more restrictions, viz. in

$$\textit{pattern} \leftarrow \textit{instruction} \ ; \ \textit{rest}$$

variables in *pattern* are bound to the corresponding parts of the result of *instruction* in *rest*, but which pattern is allowed is determined by the instruction. The grammar of the assembly language is shown in figure 2.

[1] This eases garbage collection. Using metadata in tags—as opposed to using info pointers, as in STG—makes handling tags more involved, but reduces the number of memory accesses.

function implementation:
fun ::= $\mathbf{f}\ a^\star = block$
basic block:
$block$ ::= $instr;^\star\ term$
instruction:
$instr$ ::= $x \leftarrow$ `Store` $(T\ a^\star)$
| $x \leftarrow$ $\texttt{Push}_{\texttt{CAF}}\ \mathbf{f}$
| $y \leftarrow$ `PrimOp` $\otimes\ y\ y$
| $y \leftarrow$ `Constant` n
| $T\ a^\star \leftarrow$ `Call` $call\ cont$
| $x \leftarrow$ `Force` $call\ cont$
terminator instruction:
$term$ ::= `Return` $(T\ a^\star)$
| `Jump` $call\ cont$
| `Case` $call\ cont\ [alt^\star]$
| `IfElse` $\bowtie\ y\ y\ block\ block$
| `Throw` x
case alternative:
alt ::= $T\ a^\star \rightarrow block$

callable expression:
$call$ ::= (`Eval` x)
| $(\texttt{Eval}_{\texttt{CAF}}\ \mathbf{f})$
| (`TLF` $\mathbf{f}\ a^\star$)
| (`Fix` $\mathbf{f}\ a^\star$)
evaluation continuation:
$cont$ ::= ()
| (`Apply` $a^\star$)
| (`Select` n)
| (`Catch` x)
node tag:
T ::= $\texttt{C}_{\texttt{con}}$
| $\texttt{F}_{\texttt{fun}}$
| $\texttt{P}^m_{\texttt{fun}}$
argument or parameter:
a ::= x *(ref. var.)*
| y *(prim. var.)*

Figure2. Grammar of PilGRIM's assembly language

Like the simple core language, PilGRIM programs are structured in supercombinators, i.e. only top-level functions have formal parameters. A program is a set of function definitions, where a function is defined by its name, its formal parameters and a code block. A block can be thought of as a unit, in that control flow does not enter a block other than at its beginning and once a block is entered, all instructions in that block will be executed. Only the last instruction can redirect control flow to another block permanently. Thus, we distinguish between *instructions* and *terminator instructions*. The former can not be the last of a block, whereas the latter must be. It follows, that a block is a (possibly empty) sequence of instructions, followed by a terminator instruction.

As discussed in section 3.2, nodes consist of a tag and a list of arguments. When functions return their result, they do this in the form of a node on the top of the stack. This is why the pattern to bind variables to the result of a `Call` must be in the form of a tag with a list of parameters.

First, we only consider top-level functions. A top-level function is called by forming a *callable*, using `TLF` with the name of the function to call and all the arguments to call it with. Given such a callable, `Call` pushes the return address and the arguments onto the stack and jumps to the called function's code block. In all cases, a `Call` has a corresponding `Return`. `Return` clears the stack down to (and including) the return address pushed by `Call`. Next, it pushes its result node (tag and arguments) onto the stack, before returning control flow to the return address.

Next, we consider calls to non-top-level functions. As discussed above, F-nodes contain fully applied functions. Therefore, F-nodes can be considered callable. On the level of PilGRIM's assembly language, however, different types of nodes can not be identified in the program. To this end, `Eval` takes a heap reference to any type of node, loads that node from the heap onto the stack and turns it into a callable. Calling `Eval` n thus behaves differently for different types of nodes. Since C- and P-nodes are already in WHNF, but do not represent callable code, a `Call` to such a node will implicitly return immediately, i.e. simply leave the node on the stack. F-nodes are called using the same mechanism as top-level functions.

In figure 2, `Call` has another argument, namely a *continuation*. In PilGRIM's assembly language, continuations are transformations of the result of a call, so they can be seen as part of the return. The empty continuation () leaves the result unchanged. `Select` n takes the n^{th} argument from the C-node residing at the top of the stack and (if that argument is a reference) loads the corresponding node from the heap onto the stack. Similarly, `Apply` works on the P-node at the top of the stack. Given a list of arguments, `Apply` appends its arguments to those of the P-node. If this saturates the P-node, i.e. if this reduces the number of missing arguments to nil, the resulting F-node is automatically called, as if using `Call` and `Eval`.

Instead of returning, another way to transfer control flow from the end of a block to the beginning of another is by using `Case`. `Case` can be understood as a `Call` combined with a switch statement. After the

callable returns a result and the continuation is applied, the C-node at the top of the stack is matched against a number of cases. If a case matches, the arguments of the node are bound to the corresponding parameters of the case and control flow is transferred to the case's block. Note that in this transfer of control flow, the stack is unaltered, i.e. the top-most return address remains the return address for the next `Return`.

Finally, a node can be written to the heap by the `Store` instruction. Since the heap is garbage collected, the address where a node is stored is not in the program's control. `Store` takes its arguments from the stack and pushes the reference to the newly allocated heap node.

Extensions Because some patterns of instructions are very common and, thus far, the instruction set is very small, a few extra instructions and combined instructions are added as optimizations.

Firstly, support for primitive values and operations is added. As a load immediate instruction, `Constant` produces a primitive constant. Primitives constants can be fed to primitive operations by means of `PrimOp`. Control flow can be determined by comparison operations on primitive values in `IfElse`. The implementation of arithmetic operations in PilGRIM is such, that it can be seen as an independent coprocessor. By letting the operation be a parameter of `PrimOp` and `IfElse`, the instruction set remains unchanged when support for more primitive operations is added.

Secondly, we add support for functions in Constant Applicative Form (CAF). These are functions without arguments, i.e. global constants. They are stored in a fixed position on the heap and require special treatment in case of garbage collection [11]. To this end, $\texttt{Push}_{\texttt{CAF}}$ generates a constant reference to a function in CAF. Furthermore, $\texttt{Eval}_{\texttt{CAF}}$ is used to make such a function callable. $\texttt{Eval}_{\texttt{CAF}}$ can be interpreted as a $\texttt{Push}_{\texttt{CAF}}$, followed by an `Eval`.

Thirdly, there are some useful optimizations with regards to control flow instructions. The common occurrence of tail recursion in lazy functional programming languages calls for a cheaper than having to `Call` every recursion. The `Jump` instruction is a terminator instruction that redirects control to the code block of the recursive call, without instantiating a new call frame on the stack. Another combination with calls is having a `Call` immediately followed by a `Store`. This happens in the case of evaluating references in a strict context. `Force` is a `Call`, followed immediately by a `Store` of the call's result.

Finally, for exception handling, we add a terminator instruction `Throw` and a continuation `Catch`. The latter places a reference to an exception handler on the stack. The former unwinds the stack, down to the first exception handler and calls that handler with the thrown value as argument.

3.4 Translation of core language to the instruction set

The translation from the core language (presented in section 3.1) to PilGRIM's assembly language (section 3.3) is defined by a four-level scheme. The entry-point of the translation is $\mathcal{T}$ (figure 3). This scheme translates (strict) top-level expressions of the core language. It is quite straight-forward, except maybe for the translation of function applications. At this point, the distinction must be made between saturated an unsaturated function application. The notation $\alpha(f)$ indicates the arity of function f, whereas $|x^\star|$ denotes the number of arguments in the core language expression.

Subexpressions can be translated for lazy evaluation (by means of scheme $\mathcal{V}$), or strict evaluation (scheme $\mathcal{S}$), both shown in figure 4). In $\mathcal{S}$, the translation of variables uses the only type-distinction left in the core language, i.e. between references and primitives. The lowest level of the translation is scheme $\mathcal{E}$ (figure 5). This scheme determines for every expression the calling method, i.e. both the callable *and* the continuation.

4 The basic architecture

In this section, we describe a simplified variant of the PilGRIM architecture, which is complete enough to execute every instructions defined in the previous section. This variant executes every instruction in a single cycle and does not include most hardware optimizations.

$$
\begin{array}{ll}
\mathcal{T}[\![\ x &]\!] = \texttt{Jump}\ \mathcal{E}\ [\![x]\!] \\
\mathcal{T}[\![\ y\ x^\star &]\!] = \texttt{Jump}\ \mathcal{E}\ [\![y\ x^\star]\!] \\
\mathcal{T}[\![\ \pi_n\ x &]\!] = \texttt{Jump}\ \mathcal{E}\ [\![\pi_n\ x]\!] \\
\mathcal{T}[\![\ f\ x^\star &]\!] = \begin{cases} \texttt{Jump}\ \mathcal{E}\ [\![f\ x^\star]\!] & \text{if } \alpha(f) \leq |x^\star| \\ \texttt{Return}\ (\texttt{P}_f\ x^\star) & \text{if } \alpha(f) > |x^\star| \end{cases} \\
\mathcal{T}[\![\ C\ x^\star &]\!] = \texttt{Return}\ (\texttt{C}_c\ x^\star) \\
\mathcal{T}[\![\ \textbf{let}\ x = s\ \textbf{in}\ e &]\!] = x\ \leftarrow \mathcal{V}\ [\![s\ ]\!]\,; \mathcal{T}\ [\![e]\!] \\
\mathcal{T}[\![\ \textbf{letS}\ x = s\ \textbf{in}\ e &]\!] = x\ \leftarrow \mathcal{S}\ [\![s\ ]\!]\,; \mathcal{T}\ [\![e]\!] \\
\mathcal{T}[\![\ \textbf{fix}\ (\lambda r.f\ r\ x^\star) &]\!] = \texttt{Jump}\ (\texttt{Fix}\ f\ x^\star)\ () \\
\mathcal{T}[\![\ \textbf{try}\ f\ x^\star\ \textbf{catch}\ h &]\!] = \texttt{Jump}\ (\texttt{TLF}\ f\ x^\star)\ (\texttt{Catch}\ h) \\
\mathcal{T}[\![\ \textbf{throw}\ x &]\!] = \texttt{Throw}\ x \\
\mathcal{T}[\![\ \textbf{case}\ s\ \textbf{of}\ \{\ C\ y^\star \rightarrow e\ \} &]\!] = C\ y^\star \leftarrow \texttt{Call}\ \mathcal{E}\ [\![s]\!]\,; \mathcal{T}\ [\![e]\!] \\
\mathcal{T}[\![\ \textbf{case}\ s\ \textbf{of}\ \{\ a^\star\ \} &]\!] = \texttt{Case}\ \mathcal{E}\ [\![s]\!]\ \ [C\ y^\star \rightarrow \mathcal{T}\ [\![e]\!]\ \mid (C\ y^\star \rightarrow e) \leftarrow a^\star] \\
\mathcal{T}[\![\ \textbf{if}\ \bowtie\ x^\star\ \textbf{then}\ c\ \textbf{else}\ d &]\!] = \texttt{IfElse}\ (\bowtie\ x^\star)\ \mathcal{T}\ [\![c]\!]\ \ \mathcal{T}\ [\![d]\!]
\end{array}
$$

Figure3. Toplevel expression translation

$$
\begin{array}{ll}
\mathcal{V}[\![\ x &]\!] = x \\
\mathcal{V}[\![\ f\ x^\star &]\!] = \begin{cases} \texttt{Push}_{\texttt{CAF}}\ f & \text{if } \alpha(f) = 0 \wedge |x^\star| = 0 \\ g \leftarrow \texttt{Push}_{\texttt{CAF}}\ f; \texttt{Store}\ (\texttt{F}_{\texttt{ap}}\ g\ x^\star) & \text{if } \alpha(f) = 0 \wedge |x^\star| > 0 \\ g \leftarrow \texttt{Store}\ (\texttt{F}_f\ x^\star); \texttt{Store}\ (\texttt{F}_{\texttt{ap}}\ g\ x^\star) & \text{if } \alpha(f) < |x^\star| \\ \texttt{Store}\ (\texttt{F}_f\ x^\star) & \text{if } \alpha(f) = |x^\star| \\ \texttt{Store}\ (\texttt{P}_f\ x^\star) & \text{if } \alpha(f) > |x^\star| \end{cases} \\
\mathcal{V}[\![\ C\ x^\star &]\!] = \texttt{Store}\ (\texttt{C}_c\ x^\star) \\
\mathcal{V}[\![\ y\ x^\star &]\!] = \texttt{Store}\ (\texttt{F}_{\texttt{ap}}\ y\ x^\star)) \\
\mathcal{V}[\![\ \pi_n\ x &]\!] = \texttt{Store}\ (\texttt{F}_{\texttt{Sel\_}n}\ x) \\
\mathcal{S}[\![\ n &]\!] = \texttt{Constant}\ n \\
\mathcal{S}[\![\ \otimes\ x^\star &]\!] = \texttt{PrimOp}\ \otimes\ x^\star \\
\mathcal{S}[\![\ x &]\!] = \begin{cases} x & \text{if } x \text{ is a primitive} \\ \texttt{Force}\ \mathcal{E}\ [\![x]\!] & \text{if } x \text{ is a reference} \end{cases} \\
\mathcal{S}[\![\ C\ x^\star &]\!] = \texttt{Store}\ (\texttt{C}_c\ x^\star) \\
\mathcal{S}[\![\ y\ x^\star &]\!] = \texttt{Force}\ \mathcal{E}\ [\![y\ x^\star]\!] \\
\mathcal{S}[\![\ \pi_n\ x &]\!] = \texttt{Force}\ \mathcal{E}\ [\![\pi_n\ x]\!] \\
\mathcal{S}[\![\ f\ x^\star &]\!] = \begin{cases} \texttt{Force}\ \mathcal{E}\ [\![f\ x^\star]\!] & \text{if } \alpha(f) \leq |x^\star| \\ \texttt{Store}\ (\texttt{P}_f^{\alpha(f)-|x^\star|}\ x^\star) & \text{if } \alpha(f) > |x^\star| \end{cases}
\end{array}
$$

Figure4. Lazy and strict value translation

$$
\begin{aligned}
\mathcal{E}[\![\, x \quad]\!] &= (\texttt{Eval}\ x)\ () \\
\mathcal{E}[\![\, \pi_n\ x \,]\!] &= (\texttt{Eval}\ x)\ (\texttt{Select}\ n) \\
\mathcal{E}[\![\, f\ x^\star \,]\!] &= \begin{cases} (\texttt{Eval}_{\texttt{CAF}}\ f)\ () & \text{if } \alpha(f) = 0 \wedge |x^\star| = 0 \\ (\texttt{Eval}_{\texttt{CAF}}\ f)\ (\texttt{Apply}\ x^\star) & \text{if } \alpha(f) = 0 \wedge |x^\star| > 0 \\ (\texttt{TLF}\ f\ x^\star)\ () & \text{if } \alpha(f) = |x^\star| \\ (\texttt{TLF}\ f\ x^\star)\ (\texttt{Apply}\ x^\star) & \text{if } \alpha(f) < |x^\star| \end{cases} \\
\mathcal{E}[\![\, y\ x^\star \,]\!] &= (\texttt{Eval}\ y)\ (\texttt{Apply}\ x^\star)
\end{aligned}
$$

Figure5. Evaluation expression with continuation

The structure of the architecture

Essential to the structure of PilGRIM, is a good understanding of the memory hierarchy. The main memory element is the *heap*. The heap contains whole nodes, i.e. addressing and alignment are both based on the size and format of nodes (discussed in more detail in section 4.2). Loads from and stores to the heap are performed in a single step, i.e. memory buses are also node-aligned. For the larger part, the heap consists of external DDR-memory. However, PilGRIM has a small allocation heap (see section 4.3) to exploit locality in typical functional programs.

PilGRIM's assembly language (section 3.3) is based on a stack-model. The *stack* supports reading and pushing everything that is required for the execution of an instruction in a single cycle. The stack contains nodes, values and call frames. Call frames always contain a return address and may contain an update continuation and zero or more application continuations. As discussed in section 4.1, the stack is not implemented as a monolithic component, but specialized to exploit considerable ILP.

At the logistic heart of the architecture sits a *crossbar*, which connects the stack, the heap and an ALU (used for primitive operations). The crossbar can combine parts from different sources in parallel to build a whole node, or anything else that can be stored or pushed on the stack in a single step.

PilGRIM's control comes from a *sequencer*, that calculates instruction addresses, decodes instructions and controls all other components of the architecture. The sequencer reads instructions from a dedicated *code memory*. Figure 6 shows the basic architecture with how data flows between the components, the block denoted by X is the crossbar.

4.1 Splitting the stack

The stack contains a lot of things: nodes read from memory, function arguments, return addresses, update pointers, intermediate primitive values and temporarily references to stored nodes. Every instruction reads and/or writes multiple of values from/to the stack, all these data movements make the stack and crossbar attached to it a critical central point of the core. The mix of multi word nodes, single word values and varying sized groups of arguments make it very hard to implement a stack as parallel as required, without making it big and slow. A solution to this problem is to split the stack into multiple specific purpose stacks, the Reduceron also uses multiple stacks but not as many.

RETURN/UPDATE STACK
: The return/update stack contains the return addresses, updates references and counters for the number of entries in the node and continuation stack that belong to a callframe.

CONTINUATION STACK
: The continuation stack contains application arguments or other simple continuations to be executed between a return instruction and the actual jump to the return address.

NODE STACK
: The node stack only contains complete nodes, including their tag. The top few entries from the node stack can be read directly and allows any combination of popping off some of top few entries (reading, popping and pushing of the node stack can done all in parallel).

REFERENCE QUEUE
The reference queue contains the references to the most recent few stored nodes.

PRIMITIVE QUEUE
The primitive queue contains the most recent produced primitive values.

The return stack and continuation stack are simple stacks with only push/pop functionality. The queues are register file structures with multiple read ports that contain the n most recent values written to it, when writing a new value into it the oldest value in the queue is lost.

Top of stack register A very parallel stack such as the node stack requiress lots of read and write ports which make it slow and big. By storing the top of stack in separate registers a lot of read ports can be saved because the top element is accessed the most. Pushing can be faster because the pushed data can only go to a single place and the top of stack register can be placed close to the local heap memory. The other stacks also have a top of stack register for fast access.

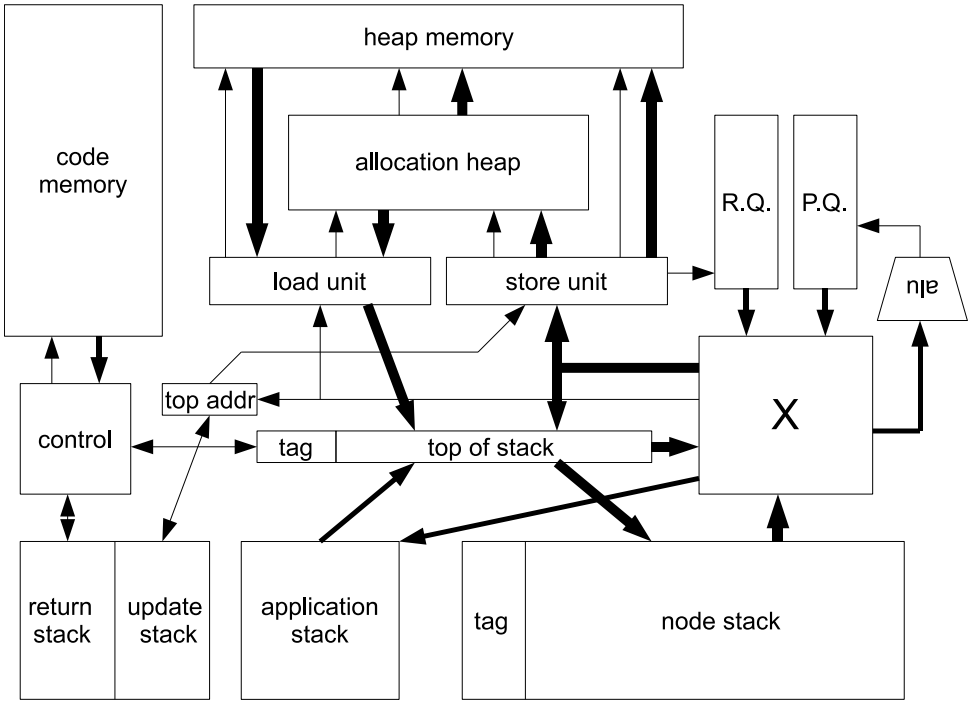


Figure6. The basic PilGRIM architecture.

4.2 Hardware component widths and constraints

We haven chosen the sizes of hardware components small enough that the cost in area and latency are reasonable, but also big enough to not be a big limitation of the performance. Some of these constraint need transformations in the code generator to deal with them.

- The datapath is 64 bits wide and all data in memory and registers is organized and addressed in 64 bits words.
- Nodes are limited to 8 words (1 tag word and 7 values). Like with the BWM and the Reduceron, the compiler has to transform the code to eliminate all constructors and function applications that are too big.

- The heap is divided in 4 word elements, because most of the nodes on the heap fit in 4 words [1] [9], more adds a lot to hardware cost and only gives a small performance benefit. Reading or writing nodes takes extra cycles if the node is more than 4 words. Updating a small F-node with a big result node is done using indirection nodes.
- Only the top 4 entries of the node stack can be read directly. The code generator needs to compact the stack or explicit spill to the heap, if nodes deeper on the stack need to be accessed.
- Both the reference queue and the primitive queue are limited to the last 16 values produced. Before values fall out the queue, they can be pushed onto the node stack.
- The continuation stack is two words wide, that is enough for most function applications [6]. For bigger applications arguments can be pushed onto the stack multiple times.

4.3 Allocation heap

Big memory structures are slow, so it is helpful to have a smaller fast memory close to the execution core. Up to 30% of the executed instructions are stores, requiring a lot of allocation bandwidth.

We can make allocation more efficient by storing all newly allocated nodes first in a sizable cyclic buffer(the allocation heap) of several hundred nodes. The allocation heap serves in effect as a fast direct mapped tagless data cache. The allocation heap is an important component in the early memory deallocation scheme described in section 5.4. We haven chosen 512 elements of 4 words as the size for the allocation heap, this is 16 kilobyte of memory. The allocation heap exploits the temporal locality between allocation and reading back of the same data. Table 4 shows that most programs expose a lot of this kind of locality.

4.4 Hardware support for the instruction set

A `Call` to an toplevel function does many things in parallel: it pushes the function arguments onto the node stack, the continuation on the continuation stack and pushes the next instruction pointer onto the return stack, while the sequencer jumps to the begin of that function. Calling `Eval` in a `Case` instruction, first loads the node references from the heap onto the top of the node stack then the tag of that node is examined. If the top node is evaluated node, then the continuation is executed, if there is any. The sequencer implements case statements by selecting an offset from a jump table that is indexed by the alternative index from a constructor tag. In case the top node is a F-node then the corresponding function is called and the reference which is being evaluated is pushed onto the update stack.

`Return` instructions push their arguments as a node onto the stack. If the current call frame has an update reference, then in parallel the same node is used to do the update. When the current call frame has a continuation it is executed. For `Apply` continuations the top node should be a P-node, number of available arguments three cases need to be handled:

- Too few arguments; the arguments are added to the top node and the new P-node is returned.
- Exactly the right number of arguments, all arguments are applied and the resulting F-node is called.
- Too many, the required arguments are applied and the rest is pushed back onto continuation stack while calling the resulting function

A `Select` continuation take the n^{th} value from the C-node on the top of the node stack, then pop the node off the stack and continue with evaluation of the selected reference. After executing the continuation, the sequencer jumps to return address popped off the return stack. To execute case statements quickly their jump table is stored together with the return stack entry, so that when returning to a `Case` instruction the alternative selection can be done in parallel.

`Store` instructions build a node from their arguments and store it in the allocation heap, the reference to the newly allocated node is pushed onto the reference queue. $\texttt{Push}_{\texttt{CAF}}$ instructions push their argument as a constant reference onto the reference queue.

`Constant` instructions push a integer value onto the primitive queue. Arithmetic operations in a `PrimOp` are executed on the ALU which gets its inputs from the crossbar, and the result is pushed onto the primitive queue. For `IfElse` instructions the ALU performs comparison and the sequencer jumps to else branch if the resulting condition is false.

4.5 Generating instructions from the assembly language

The process of generating actual instructions from the assembly language defined in section 3.3 is straightforward. First every constructor is assigned an unique number, where the numbers are grouped in such a way that the lowest bits distinguish between alternatives within a data type. Values on the node stack are indexed by a number starting with zero from the top, and values in the queues are indexed by a number starting from the most recently produced entry in the queue. Every instruction that produces a result pushes its result on either the node stack or one of the queue. Thus the code generator only have to keep track of the sequence of executed instructions within a supercombinator to translate variables in the assembly to 'register' names.

Every instruction has a mask of few bits indicating whether it needs to pop off any of top most stack nodes and to clear the reference queue. Removing nodes/references that are not used anymore from the stack/reference queue is important for accurate garbage collection, otherwise the risk of space leak is high. If an instruction contains the last use of a reference then the code generator sets the bits to pop or clear.

Before a function call or an evaluation the values on the queues need to be pushed onto the node stack if they are used afterwards. After generating all instructions in a supercombinator, the if and case instructions in it are given jump offsets for the else branch and case alternatives. The last step is linking together all supercombinators, which assigns a instruction address to every use of a function name.

5 Improving memory locality

The high latency of external memory and bandwidth limitations make optimizations in memory use very important. Functional language implementations tend to be memory intensive, so a lot can be gained here. In this section we present three optimizations in memory subsystem, avoiding memory access, exploiting locality with caches and improving locality by early deallocation of data. Locality of data is a crucial theme, because it can reduce both the used memory bandwidth and the average latency of memory accesses.

For measurements in this sections we make use of the benchmark set of the Reduceron [9]. We selected these benchmarks because they require only minimal support of primitive operations, and the memory usage and run times are small enough to use them in simulation.

5.1 Pointer tagging

Pointer tagging is putting some extra information in lowest 2 or 3 bits of a pointer. This is possible when pointers are byte addresses while they refer to word aligned data only. Pointer tagging is a commonly used optimization trick in language implementations to avoid memory reads or other indirections. In GHC pointer tagging [7] is used yielding a significant performance boost by reducing both memory accesses and branch mispredictions. Given the importance of reducing memory accesses and improving control prediction and the fact 64 bit is a lot more than what is currently physically addressable, we decided to split up each reference into a 16 bit pointer tag and a 48 bit heap address. This extreme form of pointer tagging is in itself not an optimization, but it plays an important role in various optimizations.

The first few bits in the pointer tag denote the kind of node the pointer is referring to. For constructor nodes the pointer tag contains which alternative index of its data type it is, making it possible to resolve a case statement without having to wait on the load of node. Nodes with partial applied functions have the number of missing arguments in their pointer tag.

5.2 Storing data in the reference

Some kinds of nodes are small enough that they can be stored into a reference with a pointer tag. This saves space in the heap and can make loads faster by not having to access (slow) memory. Constructor nodes without any arguments, like in enumerations consist only of a tag with number denoting the alternative, so they can fit within a reference. Partial applied function nodes that have zero arguments applied are stored in a reference, because they consist of only a function address and the count of missing arguments.

Integer values with less bits than the word width can be stored in a reference using pointer tagging. Strictness analysis in GHC removes a lot of the boxed primitives but for many programs a significant amount remains. We chose to dynamicly store smaller boxed integers (of less than 48 bits) into the reference itself, while bigger integers are stored in normal nodes. This scheme allow integers of full word width while most of them take no heap space, and some of the pointer tag bits can be used to distinguish between different constructors with . When storing primitive values in reference could is done in software it costs several instructions, but with only a little hardware support loading or storing primitives from/to references does not need additional instructions and is faster than a normal load or store.

These methods combined avoid node allocation in over 10% on average, and in over 20% of the loads the data comes from the reference itself without accessing memory, see table 2 for details.

	compact node stores [%]			compact node loads [%]		
	empty C	empty P	boxed prim	empty C	empty P	boxed prim
average	4.7	0.9	5.7	6.3	2.2	13.1
median	3.6	0.2	3.4	4.5	0.7	12.4
max	18.4	4.7	32.8	24.5	8.0	36.9

Table2. Percentage avoided load and stores by compact storage of nodes into references.

5.3 Cache structure

The allocation heap works well for allocations and reads of young data, however it is even more important to exploit the temporal locality between repeated reads of (long lived) shared data. Therefore we add a cache besides the allocation heap, this cache gets filled by data read from external memory and data spilled out of the allocation heap. Like in most processor we use a separate level one caches for instructions and data. The instruction cache is a 32 kilobyte two way associative cache with a cache line of 32 bytes (two full width 128 bit instructions).

The data cache consist of 512 cache lines of 4 words (32 bytes) each, together with the allocation heap this adds up to 32 kilobyte of local data memory. With the granularity of all data on the heap being four words, this ensures that a cache line is never shared between multiple nodes. After measurements (see table 4) we decided to make the data cache four way associative, higher associativity could improves efficiency of the cache even more, but that is probably not worth the extra cost in hardware and latency. Many high end general purpose processors use 64 byte cache lines, but for functional languages where most of data stored consist of small objects it is not optimal. While a smaller cache lines cost more hardware and are less efficient in exploiting spatial locality, we believe reducing the accidental storage of unrelated data in the same cache line is more important.

The stack register files are backed up by a small (few kilobytes) buffer, which allows the register files itself to be small, while at same time the amount of external memory accesses required for the stacks are reduced. We have not included a second level of cache (as is common on modern processors) yet, because we have not been able to run simulations big enough to determine useful parameters for a shared L2 cache. Figure 7 shows how the caches are connected with the rest of the core.

5.4 Early memory dealloction

We use reference counting mechanisms for early deallocation of nodes on allocation heap. In functional programs a lot of the data allocated on the heap is used only temporarily or is only read back once. By moving only live nodes from the allocation heap to the data cache, the number of writes to data cache is greatly reduced. Another bandwidth reducing property of the allocation heap is, that it can avoid fetching the cache line where new data is allocated into, because all data in the cache line will always been overwritten before it is read again.

We keep track of which references are potentially shared by maintaining uniqueness bits in heap references using the dashing [4] technique. The uniqueness bit is essentially a single bit reference count,

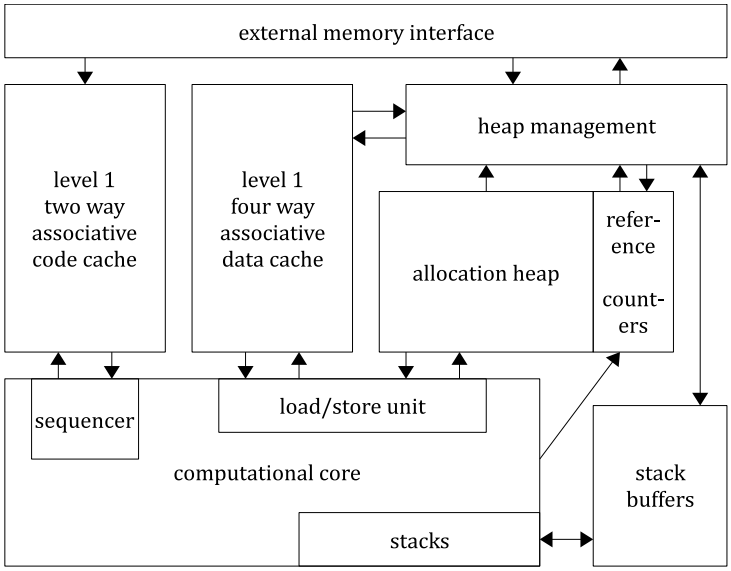


Figure7. The memory structures in the PilGRIM architecture.

with the nice property that it can be cheaply stored and maintained in the reference itself. The instruction encoding is extended so that for every reference read in an instruction, it contains a bit denoting whether that read is a copy or a destructive read. Uniqueness is used both for avoiding unnecessary updates (as in the BWM [1] and the Reduceron [9]) and for early release of memory (as in SKIM [13]). Even though it is very simple, the uniqueness bit reference count allows almost two thirds of the nodes to be deallocated early (table 3.

For references to nodes in the allocation heap that are not unique, each line in the allocation heap has an associated multibit reference counter. The update avoidance from uniqueness improves the precision of the allocation heap reference counters, because updates are a major source of reference duplication. Before a node from the allocation heap is spilled to the data cache its reference count is checked and only live nodes are moved to the data cache.

Reading a node from a unique reference from the data cache invalidates corresponding cache line, so it is a kind of destructive read. When reading from external memory using a unique reference the node is not moved into the data cache, only shared nodes are read allocated in the cache.

5.5 Evaluation of applied memory optimizations

The programs used for benchmarking are less than 100 lines of code on average and have a small working set of data in memory, this distorts the results of the benchmarks. We believe that the trends shown by the benchmarks result are valid, but that the actual numbers are too optimistic compared to big real world programs. The Braun program in this benchmark set is exceptional in two ways. It works on a data structure that does not fit well within the chosen cache and allocation heap sizes. And almost all data it allocates is only read once, making filling the cache with data read data from external memory harmful to the performance of the cache.

Table 3 shows in the first column how effective using the uniqueness bit is in avoiding unnecessary updates. The third column shows how many of the nodes could be deallocated by destructively reading nodes from unique references. In the last three columns show how many of allocated nodes are never written from the allocation heap to external memory or cache because their corresponding reference counter dropped to zero.

The first three columns in table 4 show how many external reads are required with allocation heaps of various size. This gives an indication of the locality between allocations and reads. For all numbers in this table, the lower the number of external reads the better. The next tree columns show the effectiveness of the data cache with varying cache associativity. Comparing the last column (where reference counting is disabled) with the previous one gives an indication of how well not polluting the cache with dead data improves the performance of the cache.

program	updates avoided	updates required	dead nodes found	garbage collected nodes at alloc. heap size 256	512	1024
Queens	100.0	0.0	94.9	97.3	98.9	99.5
OrdList	84.8	1.4	73.0	99.2	99.5	99.8
PermSort	76.0	2.3	54.0	98.5	98.8	99.5
Braun	78.9	0.2	75.0	35.1	68.5	95.4
Queens2	50.1	12.3	45.1	76.5	87.5	94.3
Parts	89.4	8.5	72.3	98.5	99.2	99.7
MSS	80.2	18.8	80.0	80.2	81.9	88.2
While	100.0	0.0	46.8	99.9	99.9	99.9
Taut	94.5	4.1	63.3	99.5	99.8	99.9
Clausify	13.2	59.6	17.5	86.1	86.2	86.3
Adjoxo	32.8	21.7	45.0	97.8	99.0	99.5
CountDown	86.1	11.4	79.7	91.3	93.2	94.9
Cichelli	82.3	9.0	68.8	92.5	97.3	97.9
SumPuz	83.8	7.2	79.4	92.6	95.7	97.6
Mate	67.8	24.3	61.6	95.1	95.9	96.3
average	74.6	12.1	63.7	89.3	93.4	96.6

Table3. Amount [%] of updates avoided by uniqueness and unavoidable (required), the amount of early deallocated nodes with uniqueness and different sizes of the reference counted allocation heap.

program	no cache at alloc. heap size 256	512	1024	n-way associative cache with reference counting 0	2	4	no r.c. 4
Queens	32.9	28.5	25.6	0.15	0.00	0.00	0.31
OrdList	20.5	20.2	20.0	0.02	0.00	0.00	0.35
PermSort	6.4	5.1	3.6	0.07	0.01	0.01	1.15
Braun	59.2	32.8	15.4	11.72	13.27	13.78	16.36
Queens2	31.6	24.6	20.6	1.30	0.71	0.52	2.85
Parts	23.5	22.9	22.5	0.09	0.02	0.00	0.68
MSS	65.0	27.9	2.3	0.01	0.01	0.01	0.51
While	34.9	34.7	34.1	0.15	0.00	0.00	0.18
Taut	48.9	48.8	48.8	0.01	0.00	0.00	0.06
Clausify	66.3	65.7	64.8	2.64	1.60	1.53	2.12
Adjoxo	30.4	27.4	25.2	0.17	0.00	0.00	0.28
CountDown	27.4	25.5	23.7	0.33	0.09	0.06	0.06
Cichelli	75.4	70.9	67.0	5.90	1.29	0.06	0.98
SumPuz	36.6	30.5	23.9	0.71	0.11	0.05	1.54
Mate	73.6	71.5	69.5	2.94	0.33	0.02	0.16
average	42.2	35.8	31.1	1.75	1.16	1.07	1.84
w/o Braun	_	_	_	1.03	0.29	0.16	0.80

Table4. Percentage of all memory reads that use external memory, with a reference counted allocation heap only, an allocation heap (of 512 entries) and a cache, and cache only.

6 Conclusions

It is feasible to design a processor for lazy functional languages targeting high clock frequencies using a high-level instruction set. This instruction set can be matched closely with the source language so that the code generator is (relative) simple. Lazy functional languages expose a lot of low level parallelism even in a deeply pipelined design. It is possible to remove a large part of the abstraction overhead typically, incurred by high-level languages, by adding specialized hardware. The locality of data in a functional language can be considerably improved using (local) reference counting mechanisms, minimizing indirections and avoiding unnecessary data movements. This significantly reduces both the external memory bandwidth used and the average latency of memory operations.

Unfortunately no concrete performance numbers in term of clock cycles can be given at this moment, because the simulation model is not accurate enough. While this processor is not as parallel as the Reduceron, we expect the performance per cycle to be very close to the Reduceron, partly because of the optimizations GHC provides. As for absolute performance numbers compared to desktop processors; the big open question is how often the deep pipeline of this core will stall on cache misses and branches. So far we have not found critical bottlenecks, that could not be avoided by some optimizations.

Future work

First we want to finish a cycle accurate simulation model with all optimizations applied. Then we want to try to create a synthesizable hardware description of this architecture, and make it work on a FPGA, so that real programs can be benchmarked with it. To see what the behavior of the core is when not almost all data and code is in local cache, measurements with larger programs are required. We also want to support more of Haskell, for example: floating point, big numbers, mutable reference, arrays, thread primitives and other IO primitives. Once this processor is complete, a lot of interesting research possibilities open up. Building a multicore system with it, and making the core run multiple threads at the same time by fine grained multithreading, are two examples of that.

In our next paper we will describe how this architecture is pipelined and how the cycle accurate simulation model is implemented.

References

1. Lennart Augustsson. BWM: A concrete machine for graph reduction. In *Functional Programming*, pages 36–50, 1991.
2. Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, April 1999.
3. Urban Boquist and Thomas Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, pages 58–84, 1996.
4. Geoffrey L. Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *LISP and Functional Programming*, pages 244–258, 1988.
5. Thomas Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN Symposium on Compiler Construction*, pages 58–69, 1984.
6. Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ICFP*, pages 4–15, 2004.
7. Simon Marlow, Alexey Rodriguez Yakushev, and Simon L. Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP*, pages 277–288, 2007.
8. Matthew Naylor and Colin Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In *IFL*, pages 129–146, 2007.
9. Matthew Naylor and Colin Runciman. The reduceron reconfigured. In *ICFP*, 2010.
10. Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *MSP/ISMM*, pages 44–55, 2002.
11. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
12. Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, 2(2):127–202, 1992.
13. W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *LISP and Functional Programming*, pages 159–166, 1984.

A Code example to illustrate translation between core and assembly

Simple core language	PilGRIM's assembly language

sumDouble $x\ y =$
  let $d =$ double **in**
  let $u =$ upto $x\ y$ **in**
  let $m =$ map $d\ u$ **in**
  sum m
sum $=$
  let $a =$ add **in**
  letS $x = 0$ **in**
  foldr $a\ x$
double $x =$ **case** x **of**
  Int $a \rightarrow$
  letS $b = +\ a\ a$ **in**
  Int b
add $x\ y =$ **case** x **of**
  Int $a \rightarrow$ **case** y **of**
  Int $b \rightarrow$
  letS $c = +\ a\ b$ **in**
  Int c
 	
upto $x\ y =$ **case** x **of**
  Int $a \rightarrow$ **case** y **of**
  Int $b \rightarrow$ upto$'\ a\ b$
 	
upto$'\ a\ b =$ **if** $(>\ a\ b)$ **then**
  Nil
  else
  letS $n = 1$ **in**
  letS $x = +\ a\ n$ **in**
  let $xs =$ upto$'\ x\ b$ **in**
  Cons $x\ xs$
 	
map $f\ ys =$ **case** ys **of**
  Nil $\rightarrow$ Nil
  Cons $x\ xs \rightarrow$
  let $z = f\ x$ **in**
  let $zs =$ map $f\ xs$ **in**
  Cons $z\ zs$
 	
foldr $f\ z\ ys =$ **case** ys **of**
  Nil $\rightarrow z$
  Cons $x\ xs \rightarrow$
  let $rs =$ foldr $f\ z\ xs$ **in**
  $f\ x\ rs$
	`sumDouble` $x\ y =$
	  $d \leftarrow$ `Store` $\mathtt{P}^1_{\mathtt{double}}$;
	  $u \leftarrow$ `Store` $(\mathtt{F}_{\mathtt{upto}}\ x\ y)$;
	  $m \leftarrow$ `Store` $(\mathtt{F}_{\mathtt{map}}\ d\ u)$;
	  `Jump` $(\mathtt{Eval}_{\mathtt{CAF}}\ \mathtt{sum})$ $(\mathtt{Apply}\ m)$
	`sum` $=$
	  $a \leftarrow$ `Store` $\mathtt{P}^2_{\mathtt{add}}$;
	  $x \leftarrow$ `Constant` 0;
	  `Return` $(\mathtt{P}^1_{\mathtt{foldr}}\ a\ x)$
	`double` $x =$
	  $\mathtt{C}_{\mathtt{Int}}\ a \leftarrow$ `Call` $(\mathtt{Eval}\ x)$ $()$;
	  $b \leftarrow$ `PrimOp` $+\ a\ a$;
	  `Return` $(\mathtt{C}_{\mathtt{Int}}\ b)$
	`add` $x\ y =$
	  $\mathtt{C}_{\mathtt{Int}}\ a \leftarrow$ `Call` $(\mathtt{Eval}\ x)$ $()$;
	  $\mathtt{C}_{\mathtt{Int}}\ b \leftarrow$ `Call` $(\mathtt{Eval}\ y)$ $()$;
	  $c \leftarrow$ `PrimOp` $+\ a\ b$;
	  `Return` $(\mathtt{C}_{\mathtt{Int}}\ c)$
	`upto` $x\ y =$
	  $\mathtt{C}_{\mathtt{Int}}\ a \leftarrow$ `Call` $(\mathtt{Eval}\ x)$ $()$;
	  $\mathtt{C}_{\mathtt{Int}}\ b \leftarrow$ `Call` $(\mathtt{Eval}\ y)$ $()$;
	  `Jump` $(\mathtt{TLF}\ \mathtt{upto}'\ a\ b)$ $()$
	`upto`$'\ a\ b =$ `IfElse` $>\ a\ b$
	  $(\mathtt{Return}\ \mathtt{C}_{\mathtt{Nil}})$
	  $\left(\begin{array}{l} x \leftarrow \mathtt{Store}\ (\mathtt{C}_{\mathtt{Int}}\ a); \\ n \leftarrow \mathtt{Constant}\ 1; \\ c \leftarrow \mathtt{PrimOp}\ +\ a\ n; \\ xs \leftarrow \mathtt{Store}\ (\mathtt{F}_{\mathtt{upto}'}\ c\ b); \\ \mathtt{Return}\ (\mathtt{C}_{\mathtt{Cons}}\ x\ xs) \end{array}\right)$
	`map` $f\ ys =$ `Case` $(\mathtt{Eval}\ ys)$ $()$
	  $[\ \mathtt{C}_{\mathtt{Nil}} \rightarrow \mathtt{Return}\ \mathtt{C}_{\mathtt{Nil}}$
	  $,\ \mathtt{C}_{\mathtt{Cons}}\ x\ xs \rightarrow$
	  $\left(\begin{array}{l} z \leftarrow \mathtt{Store}\ (\mathtt{F}_{\mathtt{ap}}\ f\ x); \\ zs \leftarrow \mathtt{Store}\ (\mathtt{F}_{\mathtt{map}}\ f\ xs); \\ \mathtt{Return}\ (\mathtt{C}_{\mathtt{Cons}}\ z\ zs) \end{array}\right)$
	  $]$
	`foldr` $f\ z\ ys =$ `Case` $(\mathtt{Eval}\ ys)$ $()$
	  $[\ \mathtt{C}_{\mathtt{Nil}} \rightarrow \mathtt{Jump}\ (\mathtt{Eval}\ z)\ ()$
	  $,\ \mathtt{C}_{\mathtt{Cons}}\ x\ xs \rightarrow$
	  $\left(\begin{array}{l} rs \leftarrow \mathtt{Store}\ (\mathtt{F}_{\mathtt{foldr}}\ f\ z\ xs); \\ \mathtt{Jump}\ (\mathtt{Eval}\ f)\ (\mathtt{Apply}\ x\ rs) \end{array}\right)$
	  $]$

Invited Talk: The Craft of Building with Timber

Johan Nordlander

Luleå University of Technology
johan.nordlander@ltu.se

Abstract. Timber is a progamming language that claims to be purely functional, classically object-oriented and inherently concurrent at the same time; while also fostering a purely reactive model of interaction that naturally lets its real-time behavior be controlled by declaration. This ambitious mix of facets does not come at the price of an overwhelming complexity, though – Timber rather lacks many commonplace features of contemporary languages such as reference cells, synchronization operations, blocking I/O and global system calls.

Compared to Haskell, Timber is a strict syntactic cousin, but with the IO top-level replaced by three closely related monadic constructs for forming methods and classes. Compared to OCaml, Timber is pure (in the Haskell sense), merges its object-oriented and concurrency constructs, and supports subtyping in the nominal Java-style instead of via row polymorphism. Timber furthermore shares Erlang's structuring emphasis on communicating processes (here called objects), although in a strongly typed fashion that is also able to distinguish between synchronous and asynchronous communication.

In this talk I will present an overview of Timber, starting out from the obvious question of why some of us believe the world really needs yet another programming language. I will demonstrate concrete application-building with Timber in two domains as separate as embedded automotive systems and graphical web interfaces; both with the aim of showing that the craft of functional programming is not diminished, but rather enhanced, by its inclusion in a context whose main abstraction is called reactive objects.

Time permitting I will also provide hints of some implementation techniques that have proven successful in the Timber compiler (and one that hasn't!), as well as point out a few future directions for the Timber language that I find truly exciting.

A Database Coprocessor for Haskell

George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers

Wilhelm-Schickard-Institut für Informatik,
Eberhard Karls Universität Tübingen
{george.giorgidze,torsten.grust,tom.schreiber,jeroen.weijers}@uni-tuebingen.de

Abstract. Relational database management systems (RDBMSs) provide the best understood and most carefully engineered query processing infrastructure available today. However, RDBMSs are often operated as plain stores that do little more than reproduce stored data items for further processing outside the database host, in the general-purpose programming language heap. One reason for this is that the aforementioned query processing capabilities require mastering of advanced features of query languages (e.g., SQL) in addition to the general-purpose language the application is programmed in. Moreover, the query languages are often inadequately integrated into the host programming language. One way to solve these problems is to use RDBMSs as a *coprocessor* for general-purpose programming languages, for those program parts that carry out *data-intensive* and *data-parallel* computations.

In this paper we present a library for database-supported program execution in Haskell. Data-intensive and data-parallel computations are expressed using familiar combinators from the standard list prelude and expressive list comprehension notation. The library, in addition to queries of basic types, supports computations over arbitrarily nested tuples and lists. The implementation minimises unnecessary data transfer and context switching between the database coprocessor and the programming language run-time by ensuring that the least possible number of queries is generated when executing the library functions. Although Haskell has inspired a number of language-integrated query facilities (most notably LINQ in Microsoft's .NET framework), as far as we know this is the first proposal and implementation of database-supported program execution facility for Haskell.

1 Introduction

Relational Database Management Systems (RDBMSs) provide well understood, highly efficient and scalable data processing facilities. However, very often it is the case that a program written in a general-purpose programming language uses the RDBMS merely as a data store by transferring the data from the database to the program heap for further processing. The potential amount of data that is transferred can be substantial whereas the final result of the computation might be very small. Instead, it might be much more efficient (and sometimes the only option) to transfer a part of the program to the database and then let

the database perform the computation. Database kernels are optimised for intra-query parallel execution and can thus very efficiently cary out *data-intensive* and *data-parallel* computations.

A number of approaches have been proposed providing for better integration of database systems into general-purpose programming languages; Well-known examples include: LINQ [13] and Links [5]. LINQ is an language extension that adds data querying capabilities to Microsoft's .Net family of languages. Links is a web programming language that features compilation of program fragments that deal with persistent data to SQL for database processing.

Although very successful (e.g., LINQ is distributed with .Net framework and is widely adopted), current language-integrated approaches have a number of limitations. For example, Links only supports database execution of a program fragment that computes flat result (i.e., a list of tuples of basic types), while .Net implementation of LINQ may generate a number of queries that is proportional to the size of the queried data (i.e., does not feature so called *avalanche-safety* property). In addition, LINQ standard query operators do not maintain list order.

Recently, in order to solve the aforementioned problems with the current language-integrated approaches, and more generally, to investigate to what extent one can push the idea of RDBMSs that directly and seamlessly participate in program evaluation, the Ferry language has been proposed [8]. Ferry is a functional programming language that is designed to be entirely executed on RDBMSs. The idea is that technology developed in the context of Ferry can be ported into other language-integrated database systems. So far, the most notable feature of Ferry has been its compilation technique that supports database execution of programs of nested types, maintains list order and provides avalanche-safety guarantees.

Although the Haskell programming language [14] has inspired a number of language-integrated query facilities (most notably LINQ which is based on list comprehensions) so far no such system has been proposed or implemented for Haskell. With *Ferry/Haskell* we provide a library that executes parts of a Haskell program on an RDBMS. As its name suggests the library design and implementation is influenced by Ferry and can be seen as an Haskell-embedded implementation of Ferry.

The library system is based on Haskell's list comprehensions and the underlying list-processing combinators and provides a convenient query integration into the host language. The library, just like Ferry language, in addition to queries of basic types, supports computations over arbitrarily nested tuples and lists. The implementation minimises unnecessary data transfer and context switching between the database *coprocessor* and the programming language run-time by ensuring that the least possible number of queries are generated when executing the library functions. Specifically, in Ferry/Haskell, the number of queries is only dependent on the number of list type constructors in the result type of the computation and does not depend on the size of the queried data.

Our contribution with this paper is the first proposal and implementation of a library for database-supported program execution in Haskell.

2 Ferry/Haskell by Example

Consider the database table `facilities` in Figure 1 which lists a sample of contemporary facilities (query languages, APIs, *etc.*) that are used to query database-resident data. We have attempted to categorize these facilities (see column `cat` of table `facilities`): query language (`QLA`), library (`LIB`), application programming interface (`API`), host language integration (`LIN`), and object-relational mapping (`ORM`). Furthermore, each of these facilities has particular features (table `features`). A verbose description of these features is given by the table `meanings`.

facilities

fac	cat
SQL	QLA
ODBC	API
LINQ	LIN
Links	LIN
Rails	ORM
Ferry	LIN
ADO.NET	ORM
Kleisli	QLA
HaskellDB	LIB

features

fac	feature
SQL	aval
SQL	type
SQL	SQL!
LINQ	nest
LINQ	comp
LINQ	type
Links	comp
Links	type
Links	SQL!
Rails	nest
Rails	maps
Ferry	list
Ferry	nest
Ferry	comp
Ferry	aval
Ferry	type
Ferry	SQL!
ADO.NET	maps
ADO.NET	comp
ADO.NET	type
Kleisli	list
Kleisli	nest
Kleisli	comp
Kleisli	type
HaskellDB	comp
HaskellDB	type
HaskellDB	SQL!

meanings

feature	meaning
list	respects list order
nest	supports data nesting
aval	avoids query avalanches
type	is statically type-checked
SQL!	guarantees translation to SQL
maps	admits user-defined object mappings
comp	has compositional syntax and semantics

Fig. 1. Database-resident input tables for the sample Program P.

Given this base data, an interesting question would be: What features are characteristic for the query facility *categories* introduced above? Using Ferry/Haskell, this question can be answered with Haskell Program P of Figure 2. Evaluating this program results in the nested list shown in Figure 3.

As Program P processes database-resident data, it would be most efficient to perform the computation close to the data and *let the database query engine itself execute Program P!* With Ferry/Haskell, this is exactly what we propose and provide. In the following chapter, we will describe how Ferry/Haskell compiles

and executes Program P, effectively using the RDBMS as a database coprocessor that supports the Haskell run-time.

```
hasFeatures :: Q String → Q [String]
hasFeatures f = [$qc | feat | (fac, feat) ← table features, fac ≡ f |]
means :: Q String → Q String
means f = head [$qc | mean | (feat, mean) ← table meanings, feat ≡ f |]
query :: IO [(String, [String])]
query = fromQ connection
  [$qc | (the cat, nub $ concat $ map (map means ∘ hasFeatures) fac)
   | (fac, cat) ← table facilities, then group by cat |]
```

Fig. 2. Program P

```
[("API", []),
 ("LIN", ["avoids query avalanches", "guarantees translation to SQL",
          "has compositional syntax and semantics",
          "is statically type-checked",
          "respects list order", "supports data nesting"]),
 ("LIB", ["guarantees translation to SQL",
          "has compositional syntax and semantics",
          "is statically type-checked"]),
 ("ORM", ["admits user-defined object mappings",
          "is statically type-checked", "supports data nesting"]),
 ("QLA", ["avoids query avalanches", "guarantees translation to SQL",
          "has compositional syntax and semantics",
          "is statically type-checked",
          "respects list order", "supports data nesting"])]
```

Fig. 3. Program P's result

3 Ferry/Haskell Internals

In this section, we explain how embedded Ferry/Haskell programs are represented, compiled, and then executed inside the RDBMSs.

The execution pipeline is presented in Figure 4. List comprehensions are translated into list-processing combinators at compile-time (①, Figure 4), following a regular desugaring approach [15]. With a translation technique coined

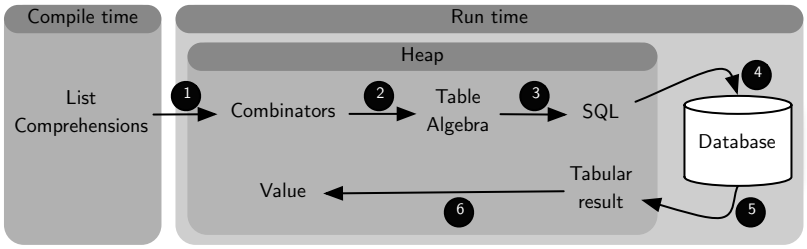


Fig. 4. Code Motion

loop-lifting [8] the list-processing combinators are compiled into the primitives of a simple variant of relational algebra (②, Figure 4). Through *Pathfinder* [6, 7] (an external algebraic optimizer and code generation facility), the algebraic primitives are compiled into SQL:1999 code (③, Figure 4). This SQL code can then be executed on off-the-shelf relational database systems (④, Figure 4). As a last step, the tabular query results are transferred back into the heap and then transformed into vanilla Haskell values (⑤ and ⑥, Figure 4).

In the remainder of this section we describe the aforementioned steps in further detail.

3.1 Ferry/Haskell Combinators

Figure 5 provides an incomplete line-up of the supported Haskell list combinators along with their types. These combinators, in the tradition of deep embedding, construct a data representation of the embedded programs they represent.

The presented combinators are very similar to the combinators for lists provided in Haskell's Prelude. In fact, they behave exactly as their counterparts in the Haskell Prelude, but instead of operating on regular lists they work on *queryable lists*. A notable omission among the supported combinators is *foldr* (and other variants of fold): general support for these combinators would require database support for recursive queries beyond the current capabilities of RDBMS engines.

The combinator *table* deserves some special attention. As can be seen in the line-up of combinators, its type features a constraint $TA\ a$, restricting the type a of the table rows to consist only of basic types (such as *Int*, *Bool*, *String etc.*) and tuples that contain such basic types. Use of the *table* combinator does not result in I/O, as it does not initiate communication with the database. In the case that the table has multiple columns, these columns are represented as fields of a tuple whose fields are ordered alphabetically. With the current Ferry/Haskell prototype, it is the user's responsibility to make sure that type a indeed matches the table's row type (otherwise an error is thrown at run-time).

```
map :: (QA a, QA b) ⇒ (Q a → Q b) → Q [a] → Q [b]
append :: QA a ⇒ Q [a] → Q [a] → Q [a]
filter :: QA a ⇒ (Q a → Q Bool) → Q [a] → Q [a]
head :: QA a ⇒ Q [a] → Q a
last :: QA a ⇒ Q [a] → Q a
tail :: QA a ⇒ Q [a] → Q [a]
init :: QA a ⇒ Q [a] → Q [a]
length :: QA a ⇒ Q [a] → Q Int
reverse :: QA a ⇒ Q [a] → Q [a]
concat :: QA a ⇒ Q [[a]] → Q [a]
take :: QA a ⇒ Q Int → Q [a] → Q [a]
drop :: QA a ⇒ Q Int → Q [a] → Q [a]
zip :: (QA a, QA b) ⇒ Q [a] → Q [b] → Q [(a, b)]
zipWith :: (QA a, QA b, QA c) ⇒ (Q a → Q b → Q c) → Q [a] → Q [b] → Q [c]
unzip :: (QA a, QA b) ⇒ Q [(a, b)] → Q ([a], [b])
groupWith :: (Ord b, QA a, QA b) ⇒ (Q a → Q b) → Q [a] → Q [[a]]
sortWith :: (Ord b, QA a, QA b) ⇒ (Q a → Q b) → Q [a] → Q [a]
the :: (Eq a, QA a) ⇒ Q [a] → Q a
table :: TA a ⇒ String → Q a
```

Fig. 5. Incomplete line-up of supported list combinators and their types.

3.2 Restricting Supported Types

In order to restrict the combinators to only work with the data types that are currently supported during compilation to relational queries, a type class *QA* (reads *queryable*) is used. This class is shown in Figure 6. The set of supported data types is a superset of the set of types that are member of the *TA* class and includes (nested) lists and tuples. In addition, the type class provides functions for converting Haskell values into queries (*i.e*, the *toQ* function) and *vice versa* (*fromQ*). The latter function triggers compilation and execution of the embedded queries, as outlined below. Function *fromQ* communicates with the database and therefore has to be provided with database connection information. As these queries generally read data that is stored in database-resident tables and thus is subject to change (via database updates), the result of *fromQ* is of type *IO a*.

```
class QA a where
  toQ   :: a → Q a
  fromQ :: Conn → Q a → IO a
```

Fig. 6. Type class *QA a* representing the queryable Haskell types.

3.3 Internal Representation

The supported list combinators have regular Haskell types and will thus be type checked by the Haskell compiler. In Figure 7, we present the data type that is used internally to represent these combinators. This datatype features types at the value-level. As a direct result of this, the internal representation is not guaranteed to represent type-correct expressions. However this datatype is not exposed to the users of the library and extra care has been taken to make sure that the combinators map to a consistent underlying representation. The combinators, however, do mention types as $Q\ a$ and not *Expr* types. This Q datatype is a wrapper type and is defined as:

data $Q\ a = Q\ Exp$

The type variable a does not occur in the type of the constructor at all, *i.e.*, $Q\ a$ is a phantom type [10, 17].

```
data Exp =
  VarE String Type
  | UnitE Type
  | IntE Int Type
  | BoolE Bool Type
  | CharE Char Type
  | TupleE [Exp] Type
  | ListE [Exp] Type
  | FuncE (Exp → Exp) Type
  | AppE Exp Exp Type
  | TableE String Type
```

Fig. 7. Internal representation of query expressions.

3.4 View Patterns

When writing a program in combinator-style, users can provide lambdas as arguments to higher-order combinators. Unlike their regular Haskell counterparts, our combinators expect functions that take a value of type $Q\ a$ as their argument. This implies that pattern matching on a value with a tuple type is not possible, limiting the usefulness of these lambdas severely. While we do not yet provide means to support pattern matching in general (see above), we do provide a limited form by making use of *view patterns* [9, 19]. View patters are a recent syntactical extension to GHC [1]. A view pattern can pattern match on values of an abstract type if it is given a *view* function that maps between the abstract and a matchable data type. In order to support multiple types with the same *view* function we introduce a type class *View* as follows:

```
class View a b | a → b where
  view :: a → b
```

Instead of projecting all datatypes onto the same matchable type, we use a type variable b that is uniquely determined by the type a.

To exemplify, such a view can now be used in lambdas as follows:

$\lambda(view \rightarrow pattern) \rightarrow expr$

GHC would then desugar this view pattern match into:

```
λx → case view x of
       pattern → expr
```

Through view patterns we allow pattern matching on tuples. In Figure 8, the instance for views on pairs is shown. The queryable tuple is transformed into a regular Haskell tuple. The fields of the Haskell tuple are both of a queryable type. The fields are constructed by applying projection combinators to the whole queryable expression, making the view pattern a convenient short-hand for projections (these projection functions are generated with Template Haskell [18]). Instances for larger tuples are similar to the instance for pairs, and, just like the projection functions, are also generated.

```
instance (QA a, QA b) ⇒ View (Q (a, b)) (Q a, Q b) where
  view q = (proj_2_1 q, proj_2_2 q)
```

Fig. 8. View instance for pairs (generated).

3.5 List Comprehension Syntax

Regular Haskell list comprehensions only support generators, guards and local bindings [14, Section 3.11]. SQL-like list comprehensions were added as a language extension to improve the expressiveness of list comprehensions [15]. Further, parallel list comprehensions are supported by GHC as an extension. The library we present supports standard list comprehensions as well as both extensions. Ferry/Haskell's list comprehension desugarer uses the combinators of Figure 5: the type of a Ferry/Haskell list comprehension is $Q\ [a]$ instead of $[a]$.

Currently, the syntax of these list comprehensions slightly deviates from regular Haskell list comprehension syntax, due to the fact that the system is implemented as a library and not as a language extension. The desugaring is implemented as a Template Haskell *quasi-quoter* [12, 18]. A Ferry/Haskell list comprehensions is thus written as:

$[\$qc \mid expr \mid quals \mid]$

3.6 Algebraic Compilation

Our compiler's intermediate language for Ferry/Haskell programs is a table algebra (see Table 1) whose operators have been designed to match the capabilities of modern SQL:1999 query processors. Featured operators include base table access (⛁), duplicate-preserving projection (π), selection (σ), cross product and join, ($\times$, $\bowtie$), and grouped aggregation (AGG). Duplicate rows are preserved by $\dot{\cup}$, $\setminus$ and are only eliminated by an explicit δ. Operators @, CAST, $\circledast$, and ϱ attach a new column to their input table. Row ranking ϱ is used to correctly encode list order and list nesting (see the discussion of ordered and nested lists below): $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{c}}$ attaches dense ranks $(1, 2, 3, \ldots)$ inside each c-group in the order given by columns $\mathsf{b}_1, \ldots, \mathsf{b}_n$. Operator ϱ thus exactly imitates SQL:1999's `DENSE_RANK` function.

Table 1. Intermediate table algebra (with n-ary operator $* \in \{+, =, \mathsf{and}, \ldots\}$ and AGG $\in \{\text{COUNT}, \text{MAX}, \text{MIN}, \ldots\}$).

Operator	Semantics
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$	project onto columns b_i, rename b_i into a_i
σ_p	select rows satisfying predicate p
$\_ \times \_$	Cartesian product
$\_ \bowtie_p \_$	join with predicate p
δ	eliminate duplicate rows
$\_ \dot{\cup} \_$	disjoint union
$\_ \setminus \_$	difference
$@_{\mathsf{a}:v}$	attach constant value v in column a
$\text{CAST}_{\mathsf{a}:(t)\mathsf{b}}$	attach value of b casted to type t in a
$\circledast_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}$	attach result of application $*(\mathsf{b}_1, \ldots, \mathsf{b}_n)$ in a
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{c}}$	group by c, attach row rank (in b_i order) in a
$\text{AGG}_{\mathsf{a}:\langle \mathsf{b}\rangle/\mathsf{c}}$	group by c, compute aggregate of b in a
⛁_R	read from database-resident table R
a b c – – –	literal table with columns a, b, c

If we want to use a relational database system as a coprocessor for Ferry/Haskell programs, then this coprocessor needs to properly reflect the semantics of (a) Ferry/Haskell's rich data model, featuring atomic types, as well as list and tuple constructors, and (b) all combinators (see Figure 5) that operate on the data model. In the following we describe the relational representation of Ferry/Haskell's values and operations that enables a SQL database back-end to act as a coprocessor that faithfully preserves the semantics of Ferry/Haskell's data model and operations.

Atomic values and (nested) tuples. Values of atomic types are directly mapped into values of a corresponding SQL type. An n-tuple $(v_1, ..., v_n), n \geqslant 1$, of such values maps into a table row of width n. A 1-tuple (v) and value v are

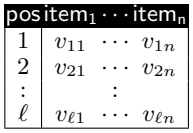
treated alike. A nested tuple $((v_1, ..., v_n), ..., (v_{n+1}, ..., v_m))$ is mapped like its flat version $(v_1, ..., v_n, ..., v_{n+1}, ..., v_m)$.

Ordered lists. Relational back-ends normally cannot provide row ordering guarantees. Therefore we let our compiler create a *runtime-accessible encoding of order*. A list value $[x_1, x_2, ..., x_l]$—where x_i denotes the n-tuple $(v_{i1}, ..., v_{in})$—is mapped into a table of width $1 + n$ as shown in Figure 9(a). A singleton list $[x]$ and its element x are represented alike. A dedicated column pos is used to encode element order in a list.

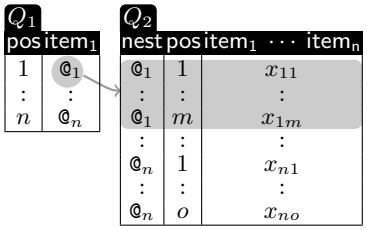
Nested lists. Relational tables are flat data structures and special consideration is given to the representation of nested lists. If a Ferry/Haskell program produces the nested list $[[x_{11}, x_{12}, ..., x_{1m}], ..., [x_{n1}, x_{n2}, ..., x_{no}]]$ where $m, n, o \geqslant 0$, then the compiler will translate the program into a *bundle* of two separate relational queries, say Q_1 and Q_2. Figure 9(b) shows the resulting tabular encodings produced by the relational query bundle:

Q_1, one query that computes the relational encoding of the outer list $[\mathtt{@}_1, \ldots, \mathtt{@}_n]$ in which all inner lists (including empty lists) are represented by *surrogate keys* $\mathtt{@}_i$, and

Q_2, one query that produces the encodings of *all* inner lists—assembled into a single table. If the ith inner list is empty, its surrogate $\mathtt{@}_i$ will not appear in the nest column of this second table.



(a) Encoding a flat ordered list.



(b) Encoding a nested list (Q_1: outer list, Q_2: all inner lists).

Fig. 9. Relational encoding of order and nesting on the database back-end.

In effect, our compiler uses a non-parametric representation for list elements [16] in which the element *types* determine their efficient relational representation: in-line (for tuples of atomic items) *vs.* surrogate-based (for lists)—we come back to this in Section 4.2. In [8] we define the compile-time analysis—coined (Un)Boxing—that we use to infer the (non-parametric) representations for all subexpressions of a Ferry/Haskell program. Note that it is exclusively the *number of list constructors* $[\cdot]$ *in the program's result type* that determines the number of queries contained in the emitted relational query bundle. For Program P and its result type $[(\mathit{String}, [\mathit{String}])]$, the bundle size thus is 2 (Figure 10). This is

radically different from related query facilities like LINQ or HaskellDB which may yield sequences of SQL statements whose size is dependent on the *size of the queried database instance* (see Section 4.1).

Operations. Relational query processors are specialists in *bulk-oriented evaluation*: in this mode of evaluation, the system applies an operation to *all* rows in a given table. In absence of inter-row dependencies, the system may process the individual rows in any order or even in parallel. To actually operate the database back-end in this bulk-oriented fashion, our compiler draws the necessary amount of independent work from Ferry/Haskell's list combinators (see Figure 5). Consider the *map* invocation

$$map\ (\lambda x \rightarrow e)\ [v_1, ..., v_n] = [\, e\,[v_1/x]\,, ..., e\,[v_n/x]\,]$$

which performs n *independent* evaluations of expression e under different bindings of x ($e\ [v/x]$ denotes the consistent replacement of free occurrences of x in e by v). The compiler exploits these semantics and applies a translation technique, coined *loop lifting* [8], that compiles *map* into an algebraic plan that evaluates $e\ [v_i/x]\ (i = 1, ..., n)$ in a bulk-oriented fashion, *i.e.*, for *all iterations*. Loop-lifting thus fully realizes the independence of the iterated evaluations and enables the relational query engine to take advantage of its bulk-oriented processing paradigm: the results of the individual evaluations of e may be produced in any order (or in parallel). A detailed account of loop-lifting is given in [8].

3.7 Query Bundle Execution

The algebraic compilation of Program P yields an initial query plan bundle featuring two algebraic queries with a total of about 150 algebra primitives from Table 1. *Pathfinder*'s query optimization [7] and SQL:1999 code generation [6] infrastructure is used to optimize this query plan bundle and transform it into a bundle of SQL statements ready for execution by the back-end. Figure 10 depicts the optimized algebraic plan bundle (of 15 operators) featuring *two roots*, one representing the outer list nesting level of the query result (Q_1), one representing the inner lists (Q_2).

After optimization, the two plan roots are fed into *Pathfinder*'s SQL:1999 code generator. The final emitted SQL:1999 code for plans Q_1 and Q_2 is shown in Figure 11.

The evaluation of the SQL code for Q_1 and Q_2 yields two tabular results as depicted in Figure 12. This table bundle holds Program P's result (see Figure 2) according to the representation introduced in Section 3.6. Note that the table schema is slightly different from the default $\mathsf{nest}|\mathsf{pos}|\mathsf{item}_1|\cdots$ because the optimizer chose columns other than the dedicated pos to faithfully represent list order (see the ORDER BY annotations attached to the plan roots in Figure 10).

Observe how the row order established by the two plan roots from Figure 10 enables Haskell value instantiation via a single sequential table scan: the surrogate orders in columns $Q_1.\mathsf{item}_1$, $Q_2.\mathsf{nest}$ coincide and Haskell value assembly merely amounts to an efficient merge between both tables.

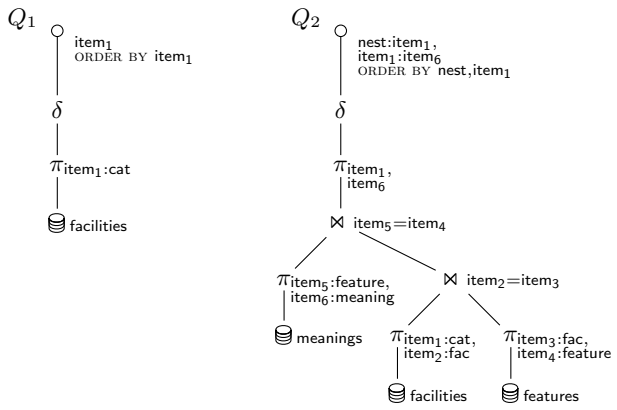


Fig. 10. Optimized algebraic plan bundle $Q_{1,2}$ implementing Program P.

```
SELECT DISTINCT t0000.cat AS item1
  FROM facilities AS t0000
ORDER BY t0000.cat ASC;
```

```
SELECT DISTINCT t0001.cat AS nest, t0000.meaning AS item1
  FROM meanings AS t0000,
       facilities AS t0001,
       features AS t0002
  WHERE t0000.feature = t0002.feature AND
        t0001.fac = t0002.fac
ORDER BY t0001.cat ASC, t0000.meaning ASC;
```

Fig. 11. SQL code generated for query Q_1 (top query) and query Q_2 (bottom query).

Q_1

item1
API
LIN
LIB
ORM
QLA

Q_2

nest	item1
LIN	avoids query avalanches
LIN	guarantees translation to SQL
LIN	has compositional syntax and semantics
LIN	is statically type-checked
LIN	respects list order
LIN	supports data nesting
LIB	guarantees translation to SQL
LIB	has compositional syntax and semantics
LIB	is statically type-checked
:	:

Fig. 12. Tabular result bundle for Program P.

4 Related Work

Embedding query facilities in programming languages is not a new idea. Many solutions for different languages already exist such as Kleisli [11], Links [5], etc.

In Section 4.1 we compare our approach with two such systems that are in our opinion closely related to Ferry/Haskell. In Section 4.2 we discuss how Ferry/Haskell's compilation technique is quite similar to the compilation strategy of Data Parallel Haskell.

4.1 LINQ and HaskellDB

Microsoft's *Language Integrated Query* (LINQ) facility [13] and HaskellDB [10] are two well-known systems that are quite close to Ferry's approach of letting developers use their normal general-purpose programming language to formulate queries against database-resident data:

LINQ seamlessly embeds a declarative query language facility into Microsoft's .NET language framework [13]. Similar to Ferry/Haskell, a LINQ query against database-resident relational tables is compiled into a sequence of SQL statements.

HaskellDB is a combinator library for Haskell that enables the construction of SQL queries in a type-safe and declarative fashion [10]. As with Ferry/Haskell, a HaskellDB query is formulated completely within Haskell without having to resort to SQL syntax.

Let us highlight two significant differences between LINQ or HaskellDB and Ferry/Haskell.

Query Avalanches Ferry/Haskell provides a guarantee that the *number of SQL queries issued* to implement a given program is exclusively *determined by the program's static type*: each occurrence of a list type constructor $[t]$ accounts for exactly one SQL query (see Section 3.6). Program P of Figure 2 and its result type of shape $[[\cdot]]$ thus led to the bundle of two SQL queries shown in Figure 11. This marks a significant deviation from LINQ and HaskellDB: in both systems, the length of the SQL statement sequence may depend on the *database instance size* resulting in an avalanche of queries. To make this concrete, we reformulated the Ferry/Haskell Program P of Figure 2 in LINQ (see Figure 13) as well as in HaskellDB (see Figure 14). Table 2 shows the number of SQL queries emitted by LINQ, HaskellDB, and Ferry/Haskell in dependence of the number of distinct categories in column cat of table `facilities`. For LINQ's and HaskellDB, the query avalanche effect is clearly visible: the number of queries generated depends on the population of column cat and a relational database back-end is easily overwhelmed by the resulting query workload [8].

List Order Preservation List element order is inherent to the Haskell data model. Ferry/Haskell relationally encodes list order (column pos) and carefully translates operations such that this order encoding is preserved (see Section 3.6). In contrast, both HaskellDB and LINQ do not provide any relational encoding of order. As a consequence, in LINQ to SQL, particular order-sensitive operations are either flagged as being unsuported or are mapped into database queries that return list elements in some arbitrary order [8].

```
from f in db.facilities
group f by f.cat into cats
select new {cat      = cats.Key,
            features = (from fac in cats
                        from feat in fac.features
                        select feat.meanings.meaning).Distinct ()}
```

Fig. 13. LINQ Version of Program P.

```
getCats :: Query (Rel (RecCons Cat (Expr String) RecNil))
getCats = do
  facs ← table facilities
  f' ← project (cat << facs ! cat)
  unique
  return f'

getCatFeatures :: String → Query (Rel (RecCons Meaning (Expr String) RecNil))
getCatFeatures cat = do
  facs  ← table facilities
  feats ← table features
  means ← table meanings
  restrict $ feats ! feature . == . means ! feature .&&.
             facs ! cat      . == . constant cat .&&.
             facs ! fac      . == . feats ! fac
  project (meaning << means ! meaning)

query :: IO [(Record (RecCons Cat String RecNil)
             ,[Record (RecCons Meaning String RecNil)])]
query = do
  cs ← doQuery getCats
  sequence $ map (λc → do
                   f ← doQuery $ getCatFeatures $ c ! cat
                   return (c, f)) cs
```

Fig. 14. HaskellDB version of Program P.

4.2 Data Parallel Haskell

RDBMSs are carefully tuned and highly efficient table processors. A look under the hood reveals that, indeed, database query engines provide a sophisticated *flat data-parallel* execution environment: most of the the engine's primitives—typically a variant of the relational algebra—apply a single operation to all rows of an input table (consider relational selection, projection, or join, for example).

In this sense, Ferry/Haskell is a close relative of Data Parallel Haskell [3, 4, 16]: both accept very similar comprehension-centric Haskell fragments, both yield code that is amenable for execution on flat data-parallel back-ends (relational

Table 2. Number of SQL queries emitted to implement Program P in dependence of the population of column cat: a comparison of LINQ to SQL, HaskellDB, and Ferry/Haskell.

# categories	**LINQ to SQL** # queries	**HaskellDB** # queries	**Ferry/Haskell** # queries
10	11	11	2
100	101	101	2
1 000	1 001	1 001	2
10 000	10 001	10 001	2

query engines or contemporary multi-core CPUs). We shed light on a few striking similarities here.

$$
\begin{aligned}
&\textbf{type } \mathit{Vector} = [:\mathit{Float}:]\\
&\textbf{type } \mathit{SparseVector} = [:(\mathit{Int}, \mathit{Float}):]\\
&\mathit{sumP} :: \mathit{Num}\ a \Rightarrow [:a:] \to a\\
&(!:) \quad :: [:a:] \to \mathit{Int} \to a\\
&\mathit{dotp} :: \mathit{SparseVector} \to \mathit{Vector} \to \mathit{Float}\\
&\mathit{dotp}\ \mathit{sv}\ v = \mathit{sumP}[:x * (v\ !:\ i) \mid (i, x) \leftarrow \mathit{sv}:]
\end{aligned}
$$

Fig. 15. Data Parallel Haskell example: sparse vector multiplication (taken from [4]).

Parallel arrays *vs.* tables Data Parallel Haskell programs operate over parallel arrays of type $[:a:]$, the primary abstraction of a vector of values that is subject to bulk computation. Positional indexing into these arrays (via $!:$) is ubiquitous. Parallel arrays are strict: evaluating one array element evaluates the entire array.

The Ferry/Haskell-generated database primitives (see Table 1) operate over (unordered) tables in which a dedicated column pos explicitly encodes element indexes (see Section 3.6). Primitive operations are always applied to all rows of their input tables.

Non-parametric data representation In Data Parallel Haskell, arrays of tuples $[:(a, b):]$ are represented as tuples of arrays $([:a:], [:b:])$ of identical length. The representation of a nested array $[:[:a:]:]$ has two components: (1) an array of $(\mathit{offset}, \mathit{length})$ descriptors, and (2) a flat data array $[:a:]$ holding the actual elements.

In Ferry/Haskell, the fields of a tuple live in adjacent columns of the same table. A nested list is represented in terms of two tables: (1) a table of surrogate keys, each of which identifies a nested list, and (2) a table of data elements, each accompanied by a foreign surrogate key to encode list membership (see Section 3.6).

Note how both representations preserve locality of the actual elements held in nested data structures.

Lifting operations In Data Parallel Haskell, operations of type $a \to b$ are lifted to apply to entire arrays of values: $[:a:] \to [:b:]$. Consider a Data Parallel Haskell variant of sparse vector multiplication (Figure 15). The comprehension defines an iterative computation to be applied to each element of sparse vector sv: project onto the components i and x, perform positional array access into v ($!:$), multiply. With *vectorisation*, Data Parallel Haskell trades comprehension notation for data-parallel combinators (*e.g.*, $fst\hat{}$, $*\hat{}$, *bpermuteP*), all operating over entire arrays (Figure 16, left).

Ferry/Haskell's translation strategy, *loop-lifting* [8], compiles a family of Haskell list combinators into algebraic primitives, all of which operate over entire tables (*i.e.*, the relational engine performs lifted application by definition—query processsors do not provide explicit iteration primitives).

Intermediate code produced by Data Parallel Haskell and Ferry/Haskell indeed exhibits structural similarities. To illustrate, we have identified the database primitives that implement the sparse vector multiplication program of Figure 15 in Figure 16 (right): *bpermuteP*, which performs bulk indexed array lookup, turns into a relational equi-join over column pos, for example.

A study of the exact relationship between Data Parallel Haskell and Ferry/Haskell still lies ahead. We conjecture that Ferry/Haskell's loop-lifting compilation strategy does have a equivalent formulation in terms of vectorisation or Blelloch's flattening transformation [2].

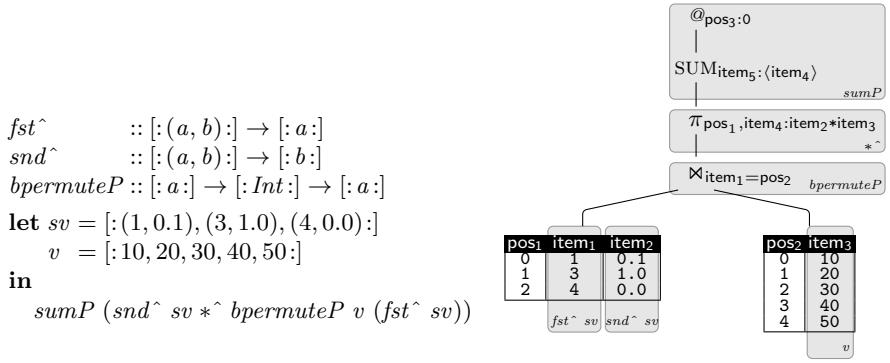


Fig. 16. Intermediate code generated for the sparse vector multiplication example of Fig. 15: Data Parallel Haskell (left) *vs.* Ferry/Haskell (right).

5 Future Work and Conclusions

In this paper we presented Ferry/Haskell, an embedding of relational database queries into Haskell. Queries are written in a style that has been designed to imitate the syntax and semantics of typical Haskell functions over lists. Haskell list comprehension syntax is supported, including two extensions for list comprehensions, with only slight deviations in syntax.

We focused on how Ferry/Haskell is implemented and embedded using Template Haskell and quasiquoting. We compared our system with two well-known query language embeddings. Our work is closely related to the work on Data Parallel Haskell and we are looking forward to explore this relationship in the near future.

We expect to continue work on the Ferry/Haskell language in the future in the following directions:

- Support for functions as first-class citizens, so that the result of a sub-query can also be a function,
- support for user-defined data types,
- complete support for pattern matching (add support for lists, and user-defined data types),
- support for (certain restricted forms of) recursion,
- an extension of the set of supported combinators over lists,
- application to large-scale data analysis problems
- an even tighter integration with Haskell, probably in the form of a true language extension.

More information on Ferry is available on the Web at `www.ferry-lang.org`. Our Ferry/Haskell library will be available in open source form soon.

Acknowledgements. Tom and Jeroen have been supported by the German Research Foundation (DFG), Grant GR 2036/3-1. George has received support from the ScienceCampus Tübingen: Informational Environments

Bibliography

[1] GHC, The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[2] G. Blelloch and G. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 1990.

[3] M. Chakravarty and G. Keller. More types for nested data parallel programming. *ACM SIGPLAN Notices*, Jan 2000.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. *Proc. of the 2007 workshop on Declarative aspects of multicore programming*, 2007.

[5] E. Cooper, S. Lindley, and P. Wadler. Links: Web programming without tiers. *Proc. of the 5th international conference on Formal methods for components and objects*, Jan 2006.

[6] T. Grust, M. Mayr, J. Rittinger, and S. Sakr. A SQL: 1999 code generator for the pathfinder XQuery compiler. *Proc. of the 2007 ACM SIGMOD international conference on Management of data*, Jan 2007.

[7] T. Grust, M. Mayr, and J. Rittinger. Let SQL drive the XQuery workhorse (XQuery join graph isolation). *Proc. of the 13th International Conference on Extending Database Technology*, Jan 2010.

[8] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. In *Proc. of the 36th International Conference on Very Large Databases (VLDB)*, Sep 2010.

[9] R. Hinze. A simple implementation technique for priority search queues. *ICFP '01: Proc. of the sixth ACM SIGPLAN international conference on Functional programming*, Oct 2001.

[10] D. Leijen and E. Meijer. Domain-specific embedded compilers. *ACM SIGPLAN Notices*, Jan 2000.

[11] W. Limsoon. Kleisli, a functional query system. *Journal of Functional Programming*, Jan 2000.

[12] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. *Proc. of the ACM SIGPLAN workshop on Haskell workshop*, Jan 2007.

[13] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. *SIGMOD '06: Proc. of the 2006 ACM SIGMOD international conference on Management of data*, Jun 2006.

[14] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[15] S. Peyton Jones and P. Wadler. Comprehensive comprehensions. *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, 2007.

[16] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. *Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2008.

[17] M. Rhiger. A foundation for embedded languages. *Transactions on Programming Languages and Systems (TOPLAS*, May 2003.

[18] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, October 2002.

[19] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. *POPL '87: Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Oct 1987.

The Design and Implementation of Feldspar: an Embedded Language for Digital Signal Processing

Emil Axelsson Koen Claessen
Mary Sheeran Josef Svenningsson
Chalmers
{emax,koen,ms,josefs}@chalmers.se

David Engdal
Ericsson
david.engdal@ericsson.com

Anders Persson
Ericsson and Chalmers
anders.persson@chalmers.se

Abstract

Feldspar is a Domain Specific Language, embedded in Haskell, for programming Digital Signal Processing algorithms. The final aim of a Feldspar program is to generate low level code with good performance. Still, we chose to provide the user with a purely functional DSL. The language is implemented as a minimal, deeply embedded core language, with shallow extensions built upon it. This paper presents, in full, the implementation, and gives the results of initial case studies carried out by domain experts from industry who were not familiar with functional programming. Our initial conclusion is that this approach works well in our domain, although much work remains.

Categories and Subject Descriptors D.3 [*Programming Languages*]

General Terms Algorithms, Languages

Keywords Haskell, Domain Specific Language, Digital Signal Processing

1. Introduction

The Feldspar project[1] aims to raise the level of abstraction at which Digital Signal Processing (DSP) algorithms are programmed [2]. Today, such algorithms are typically implemented in low level C, which is a poor match for the mathematical notations and concepts used in designing and specifying the algorithms. C is used because performance is critical in applications such as baseband processing in radio base stations. Feldspar is a Domain Specific Language (DSL) embedded in Haskell and generating C. It is designed to raise the level of abstraction at which the programmer works, without sacrificing vital performance.

Feldspar's roots in the DSP domain are reflected in the fact that it is an array programming language. Its design is deliberately minimal, so that it does not contain other DSP-specific features in its core. However, its architecture permits the addition of higher level interfaces built upon the minimal core. The intention to provide a compositional approach to expressing algorithms led to the choice of a purely functional embedded language, and indeed an early design decision was to have Feldspar programs look as much like Haskell as possible. The user works at the GHCi prompt, and the experience is very much like ordinary Haskell programming.

The following example is a Feldspar program that closely resembles the corresponding Haskell function. It computes the bitwise *and* of a mask with each integer in the range 0 to n.

```
mask :: Data Int → Data Int → DVector Int
mask m n = map (m .&.) (0...n)
```

It results in the following C code:

```
void mask(signed int  var0_0, signed int  var0_1,
          signed int *out_0 , signed int *out_1)
{
  (* out_0) = (var0_1 + 1);
  {
    signed int var2;
    for(var2 = 0; var2 < (* out_0); var2 += 1)
    {
      out_1[var2] = (var0_0 & var2);
    }
  }
}
```

The important point to note is that the code contains *one* for loop and not two, as one might expect in a naive translation. Also, the enumeration (0... n) does not result in any array allocation. Similarly, defining the function rmask m n to be reverse $ mask m n results in the following C code:

```
void rmask(signed int  var0_0, signed int  var0_1,
           signed int *out_0 , signed int *out_1)
{
  signed int var3;

  (* out_0) = (var0_1 + 1);
  var3 = ((* out_0) - 1);
  {
    signed int var2;
    for(var2 = 0; var2 < (* out_0); var2 += 1)
    {
      out_1[var2] = (var0_0 & (var3 - var2));
    }
  }
}
```

The effect of the reverse on the array appears as index manipulations inside the for loop, and once again no unnecessary intermediate data structure is created.

The close resemblance between Feldspar and Haskell programs remains, even for larger examples. However, Feldspar is restricted

[1] The Feldspar project was initiated by Ericsson. Feldspar is available open source [8].

in order to enable the generation of code with reasonable and *predictable* performance. It is the restrictiveness that allows us to find a sweet spot in which modular, reusable high level code still permits the user to control important low level details such as when memory allocation should occur or which loops in the generated code are to be fused. A major restriction is the absence of recursion over C data structures. All operations on arrays must be expressed using higher order functions like `map` and `fold`.

The contributions of this paper include full details (including all code) of how to design and implement an embedded language in Haskell that itself resembles Haskell. The choice of a minimal core language together with an API that gives the user the feeling of writing in a higher level language was fruitful, and the paper documents the implementation of an embedded DSL that follows this pattern. A novel combination of implementation techniques is presented, including mixing of deep and shallow language constructs, typed representation of expressions via GADTs [15], and smart constructors that perform optimizations on the fly. Finally, case studies from the DSP domain, conducted by users not previously familiar with functional programming, are presented.

2. Language architecture

A convenient way to implement a language is to *embed* it within an existing language. The constructs of the embedded language are then represented as functions (or similar) in the host language. It is common to distinguish between *shallow* and *deep* embedded languages. In a shallow language, the language constructs themselves perform the interpretation of the language. In a deep language, the language constructs are used to produce an intermediate data representation of the program. This representation can then be interpreted in different ways.

In general, shallow languages are more modular, allowing new constructs to be added independently of each other. In a deep implementation, each construct has to be represented in the intermediate data structure, making it much harder to extend the language. Embedded languages (both deep and shallow) can usually be interpreted directly in the host language. However, this may lead to inefficient execution, since the host compilers are not aware of possible domain-specific optimizations. For this reason, code generation is often required in order to achieve maximal performance. When code generation is needed, a deep embedding is usually beneficial, since it gives more freedom to the backends. An influential example of a code generating embedded language is Pan [7], a language for high-performance image synthesis.

The design of Feldspar tries to combine the advantages of shallow and deep implementations. We wanted the language to have the modularity and extensibility of a shallow embedding, but we also wanted to use a deep embedding in order to be able to generate high-performance code. A nice combination was achieved by using a deeply embedded core language and building high-level interfaces as shallow extensions on top of the core. The low-level core language is purely functional, but with a small semantic gap to machine oriented languages, such as C. Its intention is to be a suitable interface to the backend code generator, while being flexible enough to support any high-level interfaces.

The architecture of Feldspar is shown in Figure 1. The user interface (the "API" box) exposes the low-level core language as well as some more convenient high-level interfaces. There is currently only a single high-level interface – the *vector library*. But the generality of the core language makes it very easy to implement other interfaces, and this is something we are currently working on. The user's program generates a *core expression*, the internal data structure used as interface to the backends. At the moment, there are two backends: one for producing C code, and one for pretty printing the core expression as Haskell syntax.

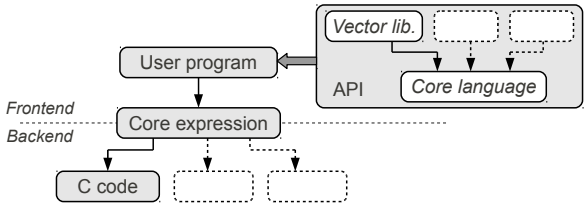


Figure 1. Feldspar architecture

In the rest of the paper, section 3 describes the core language, and section 4 describes its implementation, including the core expression data type. Section 5 describes the implementation of the vector library. The C code backend has been described in [6], and will not be covered in this paper.

3. Core language

The core language is based around the type constructor `Data`. For example, a Feldspar program that computes an integer has the type `Data Int`. Simple expressions can be formed using the interface of Haskell's `Num` class:

```
numExpr = 3*4+5 :: Data Int
```

This expression can be interpreted directly in Haskell:

```
*Main> eval numExpr
17
```

Since the core language is deeply embedded, it is possible to reify the structure of the program, using the function `printCore`:[2]

```
*Main> printCore numExpr
program = v2
  where
    v1 = 3 * 4
    v2 = v1 + 5
```

In addition to the functions of the `Num` class, the core language provides its own versions of many basic Haskell functions; for example

```
not :: Data Bool → Data Bool
div :: Data Int → Data Int → Data Int
```

These functions override the corresponding `Prelude` definitions, and we will use them without further notice throughout the examples in this paper.

More interesting programs can be built using the general constructs in Figure 2. The `Computable` class generalizes the `Data` type by allowing various Haskell structures to be treated as programs. This will be further explained in section 4.1; for now, it suffices to know that `Data a` is a member of `Computable` (for certain types `a`).

The first construct in the interface, `value`, is used to turn a Haskell value into a core language literal (a program that computes a constant value). For numeric literals, this constructor is inserted implicitly (by `fromInteger`). This allowed us to use numeric literals directly in the `numExpr` example. For convenience, Feldspar also has some pre-defined values:

```
true  = value True
false = value False
unit  = value ()
```

[2] In this example, we have turned off constant folding, which would otherwise have reduced the program to the single value `17`.

```
value :: Storable a ⇒ a → Data a

ifThenElse :: (Computable a, Computable b) ⇒
    Data Bool → (a → b) → (a → b) → (a → b)

while :: Computable st ⇒
    (st → Data Bool) → (st → st) → (st → st)

parallel :: Storable a ⇒
    Data Int → (Data Int → Data a) → Data [a]
```

Figure 2. Basic core language constructs

The conditional construct, ifThenElse, selects between two functions based on a Boolean condition. The reason for operating on functions rather than values is that this lets the user control what should go into each branch of the conditional.

The while loop, while, operates on a state of type st. The first argument is a function that computes the "continue condition" based on the current state. The second argument is the "body", which computes the next state from the current state. The result is a function from initial state to final state. Note that this while loop is a *pure* function with no side-effects. A simple example of how to use while is:

```
modulus :: Data Int → Data Int → Data Int
modulus a b = while (≥b) (subtract b) a
```

This function computes the modulus division by repeatedly subtracting b from a until the result is less than b.

The parallel construct computes an array from a length and a function that maps each index to its element. Arrays are denoted by the type Data [a] (where [a] can be arbitrarily nested). Reusing Haskell's list type for arrays results in compact and readable types. Using parallel , a program that computes the first 10 powers of two is defined as follows:

```
powersOfTwo = parallel 10 (λi → 2^i)
*Main> eval powersOfTwo
[1,2,4,8,16,32,64,128,256,512]
```

The purpose of parallel is to capture the fact that the array elements are *independent*. This means that they can be computed in any order, or even in parallel. However, our C code backend does not yet take advantage of this opportunity for parallelism.

The core language in itself is quite low-level, and may not seem to offer much advantage over C. However, Feldspar's expressiveness comes from the fact that we can use the host language, Haskell, to program powerful abstractions on top of the core language. In section 5 we introduce one such language extension: the vector library. This is a substantial extension, which more or less eliminates the need for low-level looping constructs in the user's code.

4. Core language implementation

Core expressions (see Figure 1) are represented by the generalized algebraic data type (GADT) [15] shown in Figure 3. There is a clear correspondence between the constructs in Figure 2 and the constructors of the Expr type, but also some differences. Expr is mutually recursive through the types Data and :→. These types are defined in Figures 4 and 8.

Each node in an expression tree is wrapped by the Data type, which is already familiar from earlier examples. Its purpose is to tag each node with a unique reference, in order to enable observable sharing [4]. References are handled through the interface in Figure 5. Observable sharing will be further discussed in section 4.2.

```
data Expr a where
  Value       :: Storable a ⇒ a → Expr a
  Function    :: String → (a → b) → Expr (a → b)
  Application :: Expr (a → b) → Data a → Expr b

  Variable    :: Expr a

  IfThenElse  :: Data Bool → (a :→ b) → (a :→ b)
              → (Data a → Expr b)

  While       :: (a :→ Bool) → (a :→ a)
              → (Data a → Expr a)

  Parallel    :: Storable a
              ⇒ Data Int → (Int :→ a) → Expr [a]
```

Figure 3. Core language representation

```
data Data a = Typeable a ⇒ Data (Ref (Expr a))

toData :: Typeable a ⇒ Expr a → Data a
toData = Data ∘ ref

fromData :: Data a → Expr a
fromData (Data a) = deref a
```

Figure 4. Wrapper type for expression nodes

```
data Ref a
instance Eq (Ref a)
ref   :: a → Ref a
deref :: Ref a → a
```

Figure 5. Interface to observable sharing

```
instance Storable Bool
instance Storable Int
  -- Etc. for other primitive types
instance Storable a ⇒ Storable [a]

instance Storable a ⇒ Typeable a
instance (Typeable a, Typeable b) ⇒ Typeable (a,b)
  -- Similarly for larger tuples
```

Figure 6. Supported core language types

Another purpose of Data is to make sure that each node in an expression tree stays within the set of types supported by Feldspar. Figure 6 summarizes the two classes that define two sets of supported types (not to be mixed up with the standard Haskell classes with the same name). Storable is the set of zero- or higher-dimensional arrays of primitive types. These are the types that the user works with. That is, when the user has a value of type Data a, a is generally a Storable type. The Typeable class is the set of nested tuples of Storable types. These types are only used internally. That is, the user is not allowed to work with types like Data (a,b), but is supposed to use (Data a, Data b) instead (see section 4.1).

We can now give a simple definition of the first function from Figure 2:

```
(|$|) :: Expr (a → b) → Data a → Expr b
f |$| a = Application f a

function :: Typeable b ⇒
   String → (a → b) → Data a → Data b
function fun f a = toData $ Function fun f |$| a

function2 :: Typeable c ⇒
   String → (a → b → c) → Data a → Data b → Data c
function2 fun f a b = toData $ Function fun f |$| a |$| b
```

Figure 7. Primitive function constructors

```
data a :→ b =
    Lambda (Data a → Data b) (Data a) (Data b)

freshVar :: Typeable a ⇒ () → Data a
freshVar _ = toData Variable

lambda :: Typeable a ⇒ (Data a → Data b) → (a :→ b)
lambda f = Lambda f var (f var)
  where var = freshVar ()

apply :: (a :→ b) → Data a → Data b
apply (Lambda f _ _) = f
```

Figure 8. Representation of embedded functions

```
value :: Storable a ⇒ a → Data a
value = toData ∘ Value
```

Primitive functions are constructed by the Function constructor of the Expr type. The String argument identifies the function (for use by backends), and the function argument gives the evaluation semantics. The only way to use a Function node is to apply it using the Application constructor. All the other constructors use the Data wrapper for their arguments, and the Typeable constraint for Data rules out function types. The reason for having a separate application operator is to allow nested application of curried functions. Figure 7 defines two handy combinators for defining primitive functions of one and two arguments. Note the nested use of the application operator (|$|) in function2. With these, defining new primitive functions becomes trivial:

```
not :: Data Bool → Data Bool
not = function "not" Prelude.not

mod :: Data Int → Data Int → Data Int
mod = function2 "mod" Prelude.mod
```

So far, we can define simple values and primitive functions. The remaining core language constructs deal with embedded functions. It is convenient to be able to treat Haskell functions of the form Data a → Data b as functions in the embedded language. This view is supported by the :→ type, defined in Figure 8. This type is used to represent the sub-functions of the higher-order constructs of the Expr type (for example, the body of the while loop).

In order to inspect a function of the form Data a → Data b, it has to be applied to some argument. A suitable argument can be conjured up by the freshVar function. It creates a fresh variable represented by the Variable constructor. Observable sharing makes

```
evalE :: Expr a → a
evalE (Value a)          = a
evalE (Function _ f)     = f
evalE (Application f a) = evalE f (evalD a)

evalE (IfThenElse c t e a)
    | evalD c   = evalD (apply t a)
    | otherwise = evalD (apply e a)

evalE (While cont body init) = evalD $ head
    $ dropWhile (evalD ∘ apply cont)
    $ iterate (apply body) init

evalE (Parallel l ixf) =
    map (evalD ∘ apply ixf ∘ value) [0 .. n−1]
  where n = evalD l

evalD :: Data a → a
evalD = evalE ∘ fromData
```

Figure 9. Semantics of expressions

it possible to uniquely identify each such variable[3]. An embedded function is constructed by lambda, which applies a function to a fresh variable, and stores the original function together with the variable and the applied expression. Thus, a term Lambda subst a b can be seen as a lambda expression, where a is the bound variable and b is an expression where a occurs freely. The original function subst gives an effective way of substituting a different expression for the free variable, as done by the apply function.

We now have enough building blocks to give simplified definitions of the remaining core constructs. For example, a kind of while loop can be defined as follows:

```
whileData :: Typeable st ⇒
    (Data st → Data Bool) → (Data st → Data st)
                          → (Data st → Data st)
whileData cont body =
    toData ∘ While (lambda cont) (lambda body)
```

A more general definition will be given in section 4.1.

The semantics of core expressions is given in Figure 9. This will also be generalized in section 4.1.

4.1 Extended interface

It is often desired to have multiple state variables in a loop. For example, to sum the numbers in an array (section 4.5) using a while loop, one would need two variables: one for indexing in the array and one for the running sum. The simple loop whileData, defined in section 4, has only a single value Data st in its state. One way to use multiple state variables is to make a compound state using the tuple operations in Figure 10. However, it is very inconvenient for the user to have to insert those operations explicitly in the code. Luckily, it turns out that the tupling/untupling can be automated.

Figure 11 introduces the Computable class. Generally speaking, this class provides an interface between the Data type and an open set of other, more convenient types. For example, it is much more convenient to work with (Data Int, Data Bool) than Data (Int,Bool), since the former can be constructed and decomposed using Haskell's ordinary tuple syntax. Computable

[3] One might argue that this use of observable sharing is dangerous, since we might loose more here than just sharing! Indeed, we have to be careful here that the compiler does not destroy sharing for this implementation technique to work.

```
tup2 :: Typeable (a,b) ⇒ Data a → Data b → Data (a,b)
tup2 = function2 "tup2" (,)

get21 :: Typeable a ⇒ Data (a,b) → Data a
get22 :: Typeable b ⇒ Data (a,b) → Data b
get21 = function "get21" fst
get22 = function "get22" snd

  -- Similarly for larger tuples:
  -- tup3, get31, get32, get33, tup4, etc.
```

Figure 10. Tuple operations

```
class Typeable (Internal a) ⇒ Computable a where
  type Internal a
  internalize :: a → Data (Internal a)
  externalize :: Data (Internal a) → a

instance Storable a ⇒ Computable (Data a) where
  type Internal (Data a) = a
  internalize = id
  externalize = id

instance (Computable a, Computable b) ⇒
         Computable (a,b) where
  type Internal (a,b) = (Internal a, Internal b)
  internalize (a,b) =
    tup2 (internalize a) (internalize b)
  externalize ab =
    (externalize (get21 ab), externalize (get22 ab))

  -- Similarly for larger tuples

lowerFun :: (Computable a, Computable b) ⇒
    (a → b) → (Data (Internal a) → Data (Internal b))
lowerFun f = internalize ∘ f ∘ externalize

liftFun :: (Computable a, Computable b) ⇒
    (Data (Internal a) → Data (Internal b)) → (a → b)
liftFun f = externalize ∘ f ∘ internalize
```

Figure 11. Computable class

allows us to convert easily between the two types using the internalize / externalize functions.

It is possible to automate the insertion of internalize / externalize so that the user will never see a type like Data (Int,Bool). For example, here is the general definition of the while loop:

```
while :: Computable st ⇒
    (st → Data Bool) → (st → st) → (st → st)
while cont body = liftFun (toData ∘ While contL bodyL)
  where contL = lambda (lowerFun cont)
        bodyL = lambda (lowerFun body)
```

Now we can use this loop to make a *for loop*:

```
for :: Computable s ⇒ Data Int → Data Int
    → st → (Data Int → st → st) → st
for start end init body =
                  snd $ while cont body' (start,init)
  where cont  (i,s) = i ≤ end
        body' (i,s) = (i+1, body i s)
```

This loop takes a initial and final index, an initial state and a body. The body computes the next state from the current index and current state. Note that the definition uses a pair as state in the while loop. As an example of using the for loop, we define a program for computing the sum of the squares of the numbers between 1 and n:

```
sumSq n = for 1 n 0 $ λi s → i*i+s
```

In order for the system to remain sound, the functions internalize and externalize are not allowed to change the semantics of the program. This is formalized by the following rule:

```
evalID ∘ internalize ∘ externalize == evalID
```

Computable is very powerful, as it gives a modular way to extend the language. For example, in section 5, we extend the language with a new type of vectors (seen in the introductory examples).

The remaining part of the core language implementation is given in Figure 12.

```
eval :: Computable a ⇒ a → Internal a
eval = evalID ∘ internalize

ifThenElse :: (Computable a, Computable b) ⇒
    Data Bool → (a → b) → (a → b) → (a → b)
ifThenElse cond t e =
    liftFun (toData ∘ IfThenElse cond thenSub elseSub)
  where
    thenSub = lambda (lowerFun t)
    elseSub = lambda (lowerFun e)

parallel :: Storable a ⇒
    Data Int → (Data Int → Data a) → Data [a]
parallel l ixf = toData $ Parallel l (lambda ixf)
```

Figure 12. Remaining core language definitions

```
program v0 = v7
  where
    v3 = v0 * v0
    v7 = parallel 10 ixf
      where
        ixf v1 = v6
          where
            v4 = v3 * v1
            v6 = parallel 10 ixf
              where
                ixf v2 = v5
                  where
                    v5 = v2 + v4
```

Figure 13. Result of printCore program

4.2 Reification

In earlier examples, we have used the function printCore to display the core program generated by a given Feldspar user program. This reification is done by an algorithm that turns the user program into an internal data structure. This data structure serves as an interface to the various backends. Instead of showing the real data structure here, we will present the reification algorithm informally using the Haskell syntax produced by printCore.

Consider the following program, whose core output is shown in Figure 13:

```
program x = parallel 10 $ λi → parallel 10 (+x*x*i)
```

In order to reify this program, we must first apply it to a fresh variable. This is done using the function lambda from Figure 8:

```
Lambda _ i o = lambda program
```

This gives us an expression tree o, in which the variable i appears as a leaf. However, because each node is tagged with a reference (see Figure 4), it is possible to detect when two sub-trees are equal by comparing the references. This makes it possible to view the expression as a directed graph. A depth-first traversal of this graph is then performed to create a list of all internal nodes. Sub-functions of constructs like while and parallel are already constructed by lambda, so their expressions are traversed directly as part of the main traversal.

For the above program, the traversal results in the following list of nodes:

```
program = λv0 → v7
v3      = v0 * v0
v7      = parallel 10 (λv1 → v6)
v4      = v3 * v1
v6      = parallel 10 (λv2 → v5)
v5      = v2 + v4
```

Regarded as a Haskell program, this code is not well-formed: the lambda-bound variables are used outside their scope. But since all variables are unique, it is possible to reconstruct a correct program by pushing variable definitions into the appropriate lambda expressions. For example, every variable that depends on v1 (in this case, v4, v5 and v6) should be defined locally to the function $\lambda v1 \rightarrow v6$. The result of doing this transformation is shown in Figure 13.

Note that there are often multiple choices for where to place a variable. For example, v3 and v4 are only used inside the index function $\lambda v2 \rightarrow v5$, so we could have defined them locally to that function. However, our reification uses the principle of placing variables *as globally as possible*, because this avoids recomputing values that are constant throughout a repeated computation. In the above case, v3 is computed once and used $10 \cdot 10$ times. Similarly, v4 is computed once for each index of the outer vector, but remains constant throughout the computation of each inner vector.

This way of placing computations as globally as possible resembles an optimization technique known as *loop-invariant hoisting*. It should be noted, however, that our goal with the reification algorithm was not to optimize the code, but simply to construct valid syntax. The optimization was just a nice side effect of this.

4.3 Size inference

When generating C code for the parallel construct (and for arrays in general; see section 4.5), decisions about memory allocation must be made. In embedded applications, it is generally preferred to allocate arrays statically, but this is not possible for a program like

```
λlen → parallel len (const 1)
```

where the array length is given as input to the program.

On the other hand, there are many cases where it is possible to infer an upper bound on an expression, even if the actual value is not known at compile time. For example, in the following program,

```
parallel 10 $ λlen → parallel len (const 1)
```

the length of the outer vector will obviously not exceed 10. However, even though the lengths of the inner vectors are not constant, it is statically known that len will only vary between 0 and 9, so in this case, the resulting array is guaranteed to fit in a memory block of 10×9 elements.

```
class Set (Size a) ⇒ Typeable a where
  type Size a

class Set a where
  empty     :: a
  universal :: a
  union     :: a → a → a
```

Figure 14. Adding Size type to Typeable class

```
data Ord a ⇒ Range a = Range { lowerBound :: Maybe a
                             , upperBound :: Maybe a }

instance (Ord a, Num a) ⇒ Num (Range a)
instance Ord a          ⇒ Set (Range a)

rangeLess :: Ord a ⇒ Range a → Range a → Bool
rangeLess r1 r2 | isEmpty r1 || isEmpty r2 = True
rangeLess (Range _ (Just u1))
          (Range (Just l2) _)              = u1 < l2
rangeLess _ _                              = False

rangeLessEq :: Ord a ⇒ Range a → Range a → Bool
rangeLessEq (Range _ (Just u1))
            (Range (Just l2) _) = u1 ≤ l2
rangeLessEq _ _                 = False

instance Typeable Int where
  type Size Int = Range Int

instance Typeable Float where
  type Size Float = Range Float
```

Figure 15. Representation of ranges

The purpose of size inference is to statically approximate an upper bound for numeric expressions so that whenever an expression is used to determine the length of an array, the upper bound can be used to determine the size of the allocated memory. It is convenient to perform size inference while building the expressions; thus we add a Size field to the Data type:

```
data Data a = Typeable a ⇒ Data (Size a) (Ref (Expr a))

dataSize :: Data a → Size a
dataSize (Data s _) = s
```

Size is an associated type synonym of the Typeable class, which now looks according to Figure 14. In general, Size a is a representation of a set of values of type a. General manipulation of sizes is done through the methods of the Set class.

We will not go into all the details of how the core library is redefined to deal with size information, but just highlight some key aspects. The function constructor from Figure 7 now takes an extra size propagation function Size a → Size b as argument:

```
function :: Typeable b ⇒ String → (Size a → Size b)
                        → (a → b) → (Data a → Data b)
```

The size field of the argument Data a gets passed to the size propagation function, whose result sets the size of the result Data b. Higher-arity functions (function2/3/etc.) are constructed similarly.

This allows us to define the following general Num instance supporting size inference:

```
instance (Storable a, Num a, Num (Size a)) ⇒
         Num (Data a) where
  (+) = function2 "(+)" (+) (+)
  (∗) = function2 "(∗)" (∗) (∗)
    — Etc.
```

This instance assumes a Num instance for the type Size a. A good candidate for such a size representation is the Range type, defined in Figure 15. This type represents a (potentially unbounded) range of possible values, and it is used to represent the sizes of Feldspar numeric types. One could imagine more accurate ways to represent sets of possible values, but we have found ranges to suffice for our purposes. For non-numeric types, we just use () as the size, for example:

```
type Size Bool = ()
```

The general language constructs also need to be redefined to deal with size propagation. For example, ifThenElse is now

```
ifThenElse :: (Computable a, Computable b) ⇒
    Data Bool → (a → b) → (a → b) → (a → b)
ifThenElse cond t e a =
    liftFun (toData szb ∘ IfThenElse cond tLam eLam) a
  where
    sza  = dataSize $ internalize a
    tLam = lambda sza (lowerFun t)
    eLam = lambda sza (lowerFun e)
    szb  = resultSize tLam `union` resultSize eLam
```

The input size sza is used to set the argument size of the sub-functions. This is done by the new version of lambda, which takes the size as an extra argument. The size of the total result is obtained as the union of the sizes of the two branches.

The design for specifying sizes that we have outlined above has been arrived at by trial and error. In an earlier implementation, arrays had size information as part of their type. Type-level size information is convenient from an implementation point of view because it lets Haskell's type system do much of the work in propagating the information. Having sizes as part of type signatures is also potentially useful in the same way as type signatures in general can provide good documentation and aid understanding. However, type-level size information turned out to be very hard to program with. The types became unwieldy and error messages incomprehensible. Making Feldspar programs pass the type checker was next to impossible for anyone but the language implementers. We therefore decided to move away from type-level size information and settled on the design presented here.

4.4 Partial evaluation

In our implementation we employ partial evaluation to optimize core programs. First of all, we use the standard technique of evaluating functions whose arguments are all known, that is when all arguments are constructed by the Value constructor. But we go further than this. The way in which size inference is implemented gives an additional bonus; it enables a form of partial evaluation almost for free. The size information that is propagated to infer the upper bound on array lengths can also be used to optimize certain operations statically.

Using the ranges from Figure 15, we can implement optimizing versions of several functions. As an example, we show the comparison function

```
(<) :: (Storable a, Ord b, Size a ∼ Range b)
    ⇒ Data a → Data a → Data Bool
a < b
  | a Prelude.== b       = false
  | sa `rangeLess`   sb = true
  | sb `rangeLessEq` sa = false
```

```
data a :> b = a :> b
infixr 5 :>

instance Storable a ⇒ Typeable [a]
  where
    type Size [a] = Range Int :> Size a
```

Figure 16. Size of arrays

```
  | otherwise =
          function2 "(<)" (λ_ _ → ()) (Prelude.<) a b
  where
    sa = dataSize a
    sb = dataSize b
```

The function contains three cases that perform optimization. The first case compares the pointers used by observable sharing to see if the two arguments are actually the same value. The next two cases use the range representation to see if the ranges are disjoint, and in that case return the answer statically. The final case is the default case which uses function2 to invoke the Feldspar primitive comparison function.

4.5 Arrays

Core language arrays are denoted by the type Data [a]. Constant arrays are constructed using the value function. For example,

```
value [[1,2,3],[4]] :: Data [[Int]]
```

creates a constant 2×3 matrix whose first row is initialized to [1,2,3] and the second row to [4]. Arrays are always rectangular, so the above constant has two uninitialized values at the end of the second row.

It is possible to increase the allocated size beyond the given value by using the array function:

```
array :: Storable a ⇒ Size a → a → Data a
```

This function expects the size of the array as its first argument. The size of arrays is defined in Figure 16. For example, an uninitialized 10×20 array can be constructed as follows:

```
array (10:>20:>universal) []
```

There are a few more functions that deal with arrays. We have already seen parallel which constructs an array from an index function. In addition, we have the two primitive functions

```
getIx :: Storable a ⇒ Data [a] → Data Int → Data a
setIx :: Storable a ⇒
    Data [a] → Data Int → Data a → Data [a]
```

The expression getIx arr i returns the element at index i in the array arr. Similarly, setIx arr i a returns a modified version of arr where the element at index i has been replaced by a.

5. Vector library

Many algorithms in the DSP domain operate on ordered collections of data, which is why we have added special support for vectors. A vector in Feldspar is much like an array, with one important difference: vectors are guaranteed *not to be represented in memory* at runtime, unless they are explicitly converted into a core-level array. This difference is why we sometimes call vectors *virtual*. The implementation details below discuss how a process akin to fusion takes care of this.

```
map :: (a → b) → Vector a → Vector b
map f (Indexed l ixf) = Indexed l (f ∘ ixf)

take :: Data Int → Vector a → Vector a
take n (Indexed l ixf) = Indexed (min n l) ixf

drop :: Data Int → Vector a → Vector a
drop n (Indexed l ixf)
    = Indexed (max 0 (l − n)) (λx → ixf (x + n))

tails :: Vector a → Vector (Vector a)
tails vec = Indexed (length vec + 1) (λn → drop n vec)

(...) :: Data Int → Data Int → Vector (Data Int)
(...) m n = Indexed (n − m + 1) (+ m)

zipWith :: (a → b → c)
        → Vector a → Vector b → Vector c
zipWith f aVec bVec = map (uncurry f) $ zip aVec bVec

memorize :: Storable a ⇒
            Vector (Data a) → Vector (Data a)
memorize (Indexed l ixf)
    = Indexed l (getIx (parallel l ixf))
```

Figure 17. Examples of vector functions

The support for vectors in Feldspar is implemented as a shallow embedding on top of the core language. Implementing vectors as a shallow embedding has had the benefit of allowing us to experiment with various different vector implementations easily, without having to change any other aspect of the language and its implementation. Furthermore the backend need not be aware of vectors and can therefore be simpler.

The vector library provides a set of functions inspired by standard list processing functions found in Haskell and other functional languages. This allows the programmer to write very high level code that is typically rather close to the mathematical specification of the algorithm. Some example vector functions are shown in Figure 17. Vector is the type of our virtual vectors and its argument is the type of its elements. It is very common that the elements of vectors are of type Data. We often use the abbreviation DVector in those cases.

An example of how to program with the vector library is the function below which computes the moving average of a vector. The moving average can be specified as $s_i = \frac{1}{n}\Sigma_{j=i}^{i+n-1} a_j$.

```
movingAvg :: Data Int → DVector Int → DVector Int
movingAvg n = map ((`div` n) ∘ sum ∘ take n) ∘ tails
```

The function tails produces a vector of all the suffixes of the input vector. The use of take in the argument of map creates a window into the original vector of a fixed size. To take the average of this window it is summed and then divided by its length.

The implementation of the vector type in Feldspar can be seen in Figure 18 The vector is represented by a pair consisting of the length and an index function similarly to core arrays constructed by parallel . The index function is used to index into the vector. All vectors are zero indexed.

This particular representation has the advantage that all elements in the vector are computed independently and can therefore possibly be computed in parallel. Examples of how to define functions using this representation can be seen in Figure 17. The Computable instances enable vectors to work seamlessly together with the Core language as explained in section 4.1.

```
data Vector a = Indexed
  { length :: Data Int
  , index  :: Data Int → a
  }

type DVector a = Vector (Data a)

instance Storable a ⇒ Computable (Vector (Data a))
  where type Internal (Vector (Data a)) =
                              (Int, [Internal (Data a)])
```

Figure 18. Implementation of Vectors

The chosen representation also allows for a very lightweight yet powerful implementation of vector fusion. Indeed, fusion comes as a byproduct of the way we have chosen to represent vectors. It is best illustrated by an example. Consider the following toy function:

```
squares :: Data Int → DVector Int
squares n = map square (1...n)
  where square x = x * x
```

When given an argument m, squares will reduce as follows:

```
map square (1...m)
⇒
map square (Indexed m (+1))
⇒
Indexed m (square ∘ (+1))
```

The vector computation has been reduced to a single vector. No intermediate vector will be used in the computation.

One function in Figure 17 which might require some extra explanation is memorize. The effect of a call to this function is to store a vector in memory. As a small example of when memorize is needed is if we choose to compose two movingAvg with each other. The code generated for

movingAvg n ∘ movingAvg m

contains doubly nested loops inside a parallel giving it cubic complexity. The core will look as follows (where we have elided some code for clarity):

```
program (v0_0,v0_1,(v0_2_0,v0_2_1)) = (v2,v71)
  where
    [code elided for clarity]
    v71 = parallel v2 ixf
      where
        ixf v3 = v70
          [code elided]
            (v55_0,v55_1) = while cont body (0,0)
                [code elided]
                  (v38_0,v38_1) = while cont body (0,0)
                    [code elided]
```

If we choose to insert memorize in between like this:

movingAvg n ∘ memorize ∘ movingAvg m

then the code will contain two separate loops with memory allocation to store the result of the first loop giving it quadratic complexity. This can be seen in the generated core.

```
program (v0_0,v0_1,(v0_2_0,v0_2_1)) = (v2,v74)
  where
    [code elided]
    v54 = parallel v1 ixf
      where
        ixf v19 = v53
          [code elided]
            (v38_0,v38_1) = while cont body (0,0)
```

```
            [code elided]
v74 = parallel v2 ixf
  where
    ixf v3 = v73
      [code elided]
        (v58_0,v58_1) = while cont body (0,0)
            [code elided]
```

The style of fusion described above has a significant advantage: vectors are always guaranteed to be fused away and take up no memory during runtime. This is a very strong guarantee and by far exceeds the kinds of guarantees a typical optimizing compiler gives. If the programmer wishes to avoid fusing a vector and store the vector in memory it is a simple matter of inserting a call to memorize.

This scheme provides a simple and easy to remember contract to the programmer which offers both *predictability* and *control*. It is predictable because fusion will always happen unless the programmer says otherwise, and the programmer can control memory allocation and prevent fusion using memorize.

6. Case study

To get feedback on the usability of Feldspar, we engaged Ericsson research engineers in developing a set of signal processing applications. The Ericsson research engineers are signal processing experts that know the domain well. However, they had no prior experience in functional programming.

As one test application, they chose the reference signal generation of the channel estimation of Ericsson's LTE Advanced testbed. LTE, or Long Term Evolution, is the name of the next generation mobile broadband technology as specified by 3GPP, the 3rd Generation Partnership Project.

The application constitutes a straightforward series of integer, fractional and complex number calculations which are fully specified in the LTE standard 3GPP TS 36.211 [1], section 5.5.

For each reference signal sequence, a calculation chain of approximately 25 formulas and a few tables is specified. In order to make the application development easy, our code was written to match most of these 3GPP formulas or tables to their own Feldspar functions. When the final C code is produced, most of these definitions will be inlined, and only one or two of them will be generated as separate C functions. These generated functions will then be compiled and linked together with the rest of the channel estimation application.

The following examples were chosen by the Ericsson research engineers as code that they needed to develop for reasons unrelated to the Feldspar project.

6.1 Example: Gold pseudo random sequence

As an example, consider the calculation for the pseudo-random sequence $c(n)$ which is defined in section 7.2 of 3GPP TS 36.211 as follows:

$$\begin{aligned} c(n) &= (x_1(n+N_c) + x_2(n+N_c)) mod 2 \\ x_1(n+31) &= (x_1(n+3) + x_1(n)) mod 2 \\ x_2(n+31) &= (x_2(n+3) + x_2(n+2) + x_2(n+1) + x_2(n)) mod 2 \end{aligned}$$

where $N_c = 1600$, and the x_1 sequence is initialized with

$$x_1(n) = \begin{cases} 1, & n = 0 \\ 0, & n = 1, 2, \ldots, 30 \end{cases}$$

The initialization of x_2 is given by the equation

$$c_{init} = \sum_{i=0}^{30} x_2(i) \cdot 2^i$$

where c_{init} depends on the application of the sequence.

The Feldspar code for $c(n)$ is for efficiency reasons based on 32-bit vectors.

```
type U32 = Unsigned32
c :: Data U32 → Data Int → Data U32
c c_init n = x1 `xor` x2
  where
    x1_init = 1
    x2_init = c_init
    _N_C = 1600
    (x1,x2) = for 31 (_N_C + n) (x1_init,x2_init) step
    step i = x1_step *** x2_step
    x1_step = (≫1) ∘ combineBits xor [3,0] 31
    x2_step = (≫1) ∘ combineBits xor [3,2,1,0] 31

-- Multi-bit bitwise operations for U32
combineBits :: (Data U32 → Data U32 → Data U32)
            → [Data Int] → Data Int → Data U32
            → Data U32
combineBits op is i reg = reg .|. (resultBit ≪ i)
  where
    resultBit = (Prelude.foldr1 op $
                 Prelude.map (reg ≫) is) .&. bit 0
```

The second and third equations are represented by x1_step and x2_step respectively, while the first equation is represented as the result of the function. Addition in modulo 2 is equivalent to exclusive or (xor). The (***) operator is imported from Control.Arrow.

Due to the bitwise operations, the structure of this code deviates a bit from the specification. However, as is clear from the other examples in this paper, the code is usually very close to the original specification. We suggest that the similarity between the specification and implementation is an indicator of the suitability of Feldspar in the domain of signal processing. It is often possible to transcribe the specification directly in Feldspar relying on the fusion to optimize and remove intermediate steps. The generated code for $c(n)$ is provided in Figure 19.

6.2 Example: Relying on fusion

The second example is part of a preprocessing stage in the signal processing chain.[4] For presentation purposes the types were changed from Complex to Float and hence, the primitive function sqrt has the semantics of id for evaluation.

```
ts_mf :: DVector Float → DVector Float → DVector Float
      → DVector Float
ts_mf xs ts mf = zipWith (*) ys mf
  where ys = zipWith (*) xs ts

norm :: DVector Float → DVector Float
norm ys = map (/cNorm ) ys
  where
    cNorm = sqrt ( scalarProd ys ys )

reorder :: Vector a → Vector a
reorder = permute $ λlen i → len − i

preprocess :: DVector Float → DVector Float
           → DVector Float → DVector Float
preprocess ys xs ps = reorder yrs
  where yrs = ts_mf (norm ys) xs ps
```

The function preprocess consists of several steps preparing an input vector for further processing, including scaling of individual elements (ts_mf), normalization (norm) and reordering (reorder). There are in all 4 loops over the input vector ys. In the generated

[4] The code has been sanitized to protect Ericsson IP, but has the same structure and complexity as the original.

```
void c(unsigned int var0_0,
       signed int var0_1,
       unsigned int * out)
{
  signed int var22_0;
  unsigned int var22_1_0;
  unsigned int var22_1_1;

  var22_0 = 31;
  var22_1_0 = 1;
  var22_1_1 = var0_0;
  {
    while((var22_0 ≤ (1600 + var0_1)))
    {
      var22_0 = (var22_0 + 1);
      var22_1_0 = ((var22_1_0 |
        ((((var22_1_0 >> 3) ^
           var22_1_0) & 1) << 31)) >> 1);
      var22_1_1 = ((var22_1_1 |
        ((((var22_1_1 >> 3) ^
           ((var22_1_1 >> 2) ^
            ((var22_1_1 >> 1) ^
             var22_1_1))) & 1) << 31)) >> 1);
    }
  }
  (* out) = (var22_1_0 ^ var22_1_1);
}
```

Figure 19. Generated C-code for pseudo random generator $c(n)$

code (Figure 20), only two loops remain – one for generating the normalization factor, and one to do the rest of the processing.

The original code consists of three individually hand-optimized functions that are called in sequence. It is not possible to optimize these away since they are needed for verification. In Feldspar we can generate both the individual functions and the fused composition from the same source. Thus, our fused code was 30% faster while maintaining the same properties for verification.

As some of the reference signal parameters are only used for certain configurations not interesting for the Ericsson LTE release 10 testbed, the application programmer could have analyzed the specification and manually transformed it to a less complex version. We chose not to do this, as we instead kept the more mathematical notation of the 3GPP specification and at a later stage fixed the parameters controlling functions that were of no interest to us. In all, the way the application programmer would have worked if he had developed the reference signal generation function directly in C, would have been quite different.

We did not have to change the Feldspar language to accommodate the case studies.

The users expressed that the strong typing of the host language and having access to the interactive environment of GHCi was very helpful in learning and using Feldspar. However, they would have appreciated more comprehensive tutorial material. We will address this as part of our future work.

7. Related work

Embedded Domain Specific Languages are growing in popularity, and are used in many different domains. We cannot survey the entire field, but will restrict our attention to work that has influenced ours, or has aspects in common. Feldspar is compiled, rather than interpreted or used as a library in the host language. Two early forerunners are Pan [7] and HaskellDB [12]. Of these, Pan is most closely related to Feldspar. In particular, Pan's treatment of images is similar to Feldspar's vectors, with their associated combinator-

```
void preprocess(signed int  var0_0_0, float *var0_0_1,
                signed int  var0_1_0, float *var0_1_1,
                signed int  var0_2_0, float *var0_2_1,
                signed int *out_0   , float *out_1)
{
  int var5;
  signed int var6;
  int var7;
  signed int var21_0;
  float var21_1;
  float var22;

  var5 = (var0_1_0 < var0_0_0);
  if(var5)
  { var6 = var0_1_0; }
  else
  { var6 = var0_0_0; }
  var7 = (var0_2_0 < var6);
  if(var7)
  { (* out_0) = var0_2_0; }
  else
  { (* out_0) = var6; }
  var21_0 = 0;
  var21_1 = 0.0f;
  {
    while((var21_0 ≤ (var0_0_0 − 1)))
    {
      signed int var15_0;
      var15_0 = var21_0;
      var21_0 = (var15_0 + 1);
      var21_1 = (var21_1 + (var0_0_1[var15_0] *
                            var0_0_1[var15_0]));
    }
  }
  sqrt(var21_1, &var22);
  {
    signed int var9;
    for(var9 = 0; var9 < (* out_0); var9 += 1)
    {
      signed int var10;
      var10 = ((* out_0) − var9);
      out_1[var9] = (((var0_0_1[var10] / var22) *
        var0_1_1[var10]) * var0_2_1[var10]);
    }
  }
}
```

Figure 20. Generated C-code for preprocess

based style of programming. However, Feldspar is more general and can handle a larger domain. In implementation, Feldspar differs from Pan in that it uses observable sharing to control intermediate code size, and supports fusion of vectors as an optimization.

There are a range of embedded languages that support vector programming in the style of Feldspar. Obsidian [18] is an embedded language for GPU programming that is in many ways similar to Feldspar. Feldspar's vectors were inspired by a similar construct in Obsidian. The main differences arise because Obsidian is specifically targeted to the particular concurrency model of graphics processors. So, for example, loops are unrolled in Obsidian, and the programmer has greater control over the location of data in memory than is currently the case in Feldspar.

Atom [10] is yet another language embedded in Haskell targeted at embedded systems. Compared to Feldspar, its focus is rather on control issues in realtime applications without dynamic memory allocation. Atom bases its programming model on monads.

Repa [11] is a library for array programming in Haskell that shares many similarities with Feldspar's vectors; the two approaches were developed independently. Repa uses the same model of fusion as Feldspar, and also offers programmer controlled memory allocation. It provides greater reusability through a notion of shape polymorphism, which allows functions to work over vectors with different shapes. We intend to adopt this approach in Feldspar.

Fusion is a well-established technique for lists and arrays [5, 9, 14]. The style of fusion we employ in Feldspar is similar but there are also differences. Firstly, the contract to the programmer is simpler in Feldspar: all vectors are guaranteed to be fused away and not to require any memory allocation – unless the function memorize is used. Previous fusion techniques have a more complex story: functions are classified as good producers and/or good consumers and only when good consumers meet good producers are intermediate structures removed. Secondly, the sets of functions that can be defined and fused differ in Feldspar and shortcut fusion. The differences have two sources: in Feldspar vectors have fixed length and every element is computed independently. Having a fixed length enables writing and fusing functions like reverse, while it poses problems for list based fusion. For example reverse $\circ$ reverse simply becomes the identity function for vectors. Optimizing this example is much harder for lists because they may be infinite. On the other hand, vectors have problems with functions like filter because it is not possible to compute the length of the result without computing the whole vector first. scan-like functions also pose problems. The elements in a vector are computed independently whereas the result of a scan depends on (potentially) all of the previous elements.

Other projects aimed at the DSP domain include Spiral [16], Single Assignment C (SAC) [17] and Embedded MATLAB [13]. Spiral automates the production of high performance libraries for DSP applications (among others). To that end, it has a high-level language, SPL, for specifying transforms. SPL has no notion of time or space usage; instead search is used to try to find the best implementation. SAC is a language aimed at efficient array programming and is similar to Feldspar in many respects, including the fact that the array programming model is implemented modularly as a library. However, SAC inherits from C in the sense that it is a sequential, first-order language; thus, the programming experience is rather different and in particular less modular than in Feldspar. Embedded MATLAB is an effort to compile MATLAB to C suitable for running on embedded hardware. Of necessity, Embedded MATLAB is a subset of full MATLAB. Since it is common to develop and prototype DSP algorithms in MATLAB, it makes sense to compile these prototypes directly. The compiler is developed using standard methods, and, as with many optimizing compilers, it can be difficult to predict the results when many optimizations are combined.

Several methods exist to aid the embedding of a language in Haskell. The finally tagless technique [3] provides a very powerful and compositional way of embedding languages. It could perhaps have been used in Feldspar to separate the implementation of the generation of the core expression type from the size inference. We chose not to use it for two reasons. Firstly, the types become more awkward; the (result-) type of an embedded program is simply a qualified type variable. We find it hard to motivate this for Feldspar beginners. Also, it exposes details of the implementation to the user. Secondly, we have found finally tagless to be incompatible with our use of observable sharing. Since language constructs in finally tagless are overloaded, they are typically implemented as projection functions on dictionaries. When the type is known at compile time, the compiler might choose to remove the dictionary as an optimization. However, this optimization will influence whether the term will be shared under observable sharing or not.

8. Discussion and future work

The language presented in this paper has some good sides and some sides that need more work. Our main impression, though, is that it actually works! And this result was not at all obvious when we started working on Feldspar. One of the keys to this result was the choice of using a minimalistic low-level core language with a high-level interface implemented as a shallow extension to the core. The minimal core language is quite close to the hardware, making it relatively easy for the backend to produce C code with good enough performance. At the same time, the core is flexible enough to support different kinds of high-level interfaces. This separation between the frontend and backend was particularly fruitful in our project, as these parts were developed in two different countries.

Feldspar aims to offer the same style of programming as in ordinary Haskell: pure functions, list-like processing, higher-order functions, etc. It was not obvious that this would be a good fit for the DSP domain. But now, based on the experience of Ericsson research engineers, we can say that pure functional programming appears to be quite well-suited for this task. Admittedly, we need to work on larger examples, and get feedback from more users before we can draw any real conclusions. But the initial results are very promising. We believe that a key to achieving high-level code with good performance is the vector library, which enables powerful code optimization in a predictable and controllable manner.

The implementation of Feldspar has benefited a lot from using Haskell; the first thing one notices is of course the flexible and lightweight syntax. The powerful type system has been invaluable; we have made essential use of GADTs, type functions and overloading.

While our current language shows great potential, we are also aware of some quite serious problems. Our simple core language has been a success. It supports powerful high-level interfaces, such as the vector library, while enabling decent code generation. However, when going for maximal performance (which, ultimately we are), the current core language fails to produce code of sufficiently high performance in commonly occurring cases.

As an example, take the append function $++$. When compiled to C, it generates the following loop:

```
for(var2 = 0; var2 < (* out_0); var2 += 1)
{
  var8 = (var2 < var0_0_0);
  if(var8)
  { out_1[var2] = var0_0_1[var2]; }
  else
  { out_1[var2] = var0_1_1[(var2 - var0_0_0)]; }
}
```

The outer loop goes over the whole result vector (out_1). In each iteration, a conditional is used to decide whether to pick elements from the first argument (var0_0_1) or the second (var0_1_1).

In general, having a conditional inside a loop can prevent the C compiler from doing crucial optimizations. It would be better to have two loops in sequence; the fist loop reading from the first argument and the second loop from the second argument. The problem with our current core language is that *this desired loop structure is not even possible to express*. There are a large number of useful C code patterns that are out of reach from the core language. To deal with this problem, we are working on improving the core language.

The problem with $++$ is also related to our virtual vector model, which uses a single index function to compute every element. In this model, the $++$ operator has no choice but using a conditional to determine which argument to read from. Thus, we are exploring more refined vector models that could take advantage of improvements in the core language.

Another limitation is the lack of control over memory, due to referential transparency. The user can control whether or not to use memory for vectors (via the memorize function), but it is not possible to control memory layout and memory reuse. We hope to tackle these problems by developing a "system layer" on top of current Feldspar, where things like memory usage and concurrency can be handled. In principle, this system layer will act as Feldspar's "IO monad", but the aim is to have a more declarative interface.

The openness of Feldspar makes it easy to add new domain-specific combinators that capture patterns commonly used in DSP. We are currently working on combinators for describing streaming computations with feedback, which is the natural way to define, for example, digital filters. We are also working on a more extensive library for algebraic description of DSP transforms, heavily inspired by the Spiral project [16].

Given that we've chosen to keep Feldspar very close to Haskell one might wonder why we didn't program the DSP algorithms directly in Haskell and spend our efforts improving the compilation of Haskell programs. Haskell programs needs an extensive runtime system in order to run which takes up precious space on embedded platforms where resources are scarce. Furthermore the cost model of Haskell is complex, both when it comes to time and space consumption. One of the key design philosophies of Feldspar was to keep these things predictable and under programmer control.

As future work we would like to extend the backend to support generation of LLVM code. However, this has not yet been a priority for us due to the lack of LLVM support for DSP processors.

The development of the kinds of data-processing DSP algorithms for which the current version of Feldspar is designed is not a major source of functional bugs (compared to more control-oriented software). The comparison of implementations with MATLAB models is standard practice (and we did this in our case studies). We will also investigate the use of QuickCheck-style testing and of automated formal verification based on first-order theorem provers and SMT solvers. Both of these approaches build on formal specifications and they seem promising as we are in the unusual situation of working with well-specified functions.

9. Conclusion

We have shown through case studies performed by domain experts that pure functional programming is quite well-suited to the description of algorithms in the DSP domain. Perhaps more surprisingly, from such programs we can also generate C code that looks quite close to the code a programmer would write by hand.

Feldspar is implemented as an embedded language in Haskell, and it makes essential use of advanced Haskell features, such as GADTs and overloading. The implementation is based around a simple, low-level, functional core language, which can be fairly easily translated to C code. The power of the implementation comes from the ability to program high-level interfaces as shallow extensions to the core language. We have presented one such extension – the vector library – which enables list-like processing and powerful fusion of vector traversals. Future work will include a revision of the core language to give access to a greater range of C programming patterns commonly used in the domain. We will also add a system layer to deal with the deployment in space and time of the data-processing algorithms currently described in Feldspar.

Acknowledgments

This research is funded by Ericsson, Vetenskapsrådet, and the Swedish Foundation for Strategic Research. The Feldspar project is an initiative of and is partially funded by Ericsson Software Research and is a collaboration between Chalmers, Ericsson and ELTE University, Budapest. We wish to thank Peter Brauer of Ericsson for working with us on the case studies.

References

[1] 3rd Generation Partnership Project. Technical Specification; Evolved Universal Terrestrial Radio Access. `http://www.3gpp.org/ftp/specs/archive/36_series/36.211/`.

[2] E. Axelsson, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In *Proc. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MemoCode*. IEEE Computer Society, 2010.

[3] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[4] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Proc. of Asian Computer Science Conference (ASIAN)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. 12th ACM SIGPLAN Int. Conf. on Functional Programming*. ACM, 2007.

[6] G. Dévai, M. Tejfel, Z. Gera, G. Páli, G. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, and A. Persson. Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In *Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, assoc. with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[7] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Func. Prog.*, 13:3:455– 481, 2003.

[8] Feldspar. Functional Embedded Language for DSP and PARallelism. http://feldspar.inf.elte.hu/feldspar/.

[9] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Int. Conf. on Functional programming languages and computer architecture (FPCA)*, pages 223–232. ACM, 1993.

[10] T. Hawkins. Atom. `http://tomahawkins.org/`.

[11] G. Keller, M. Chakravarty, R. Leshchinskiy, S. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of ICFP 2010*. ACM SIGPLAN, 2010.

[12] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. *ACM SIGPLAN NOTICES*, 35(1):109–122, 2000.

[13] G. Martin and H. Zarrinkoub. From MATLAB to Embedded C, The Mathworks, 2009. available at `http://www.mathworks.com/company/newsletters/news_notes/2009/matlab-embedded-c.html`.

[14] S. Peyton Jones, A. Tolmach, and C. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proc. Haskell Workshop*. ACM SIGPLAN, 2001.

[15] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. eleventh ACM SIGPLAN int. conf. on Functional programming (ICFP)*, pages 50–61. ACM, 2006.

[16] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. IEEE*, 93(2):232– 275, 2005.

[17] S. Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(06):1005–1059, 2003.

[18] J. Svensson, M. Sheeran, and K. Claessen. GPGPU Kernel Implementation and Refinement using Obsidian. In *Proc. Seventh International Workshop on Practical Aspects of High-level Parallel Programming, ICCS*. Procedia, 2010.

Combinators for Local Search In Haskell

Richard Senington and David Duke

School of Computing, University of Leeds, Leeds, LS2 9JT

Abstract. "Local search" refers to a collection of methods for tackling combinatorial problems. In practice such problems are often very large, for example optimizing train network timetables. For these, stochastic local search methods are competitive with the best known exhaustive search methods.
This paper reports on an implementation of local search using Haskell to provide a set of composable transformations over search spaces. In contrast with the monolithic stochastic processes found in imperative implementations, the Haskell transformations can be assembled both into standard stochastic search techniques, and into new methods. The paper describes challenges in formulating local search combinators in Haskell, in particular the management of stochastic methods and large solution spaces. We report initial runtime performance on two classic optimisation problems.

Keywords: Search, Optimisation, Stochastic, Combinatorial

1 Introduction

Local search refers to a collection of methods for tackling combinatorial problems ubiquitous in areas including the sciences, engineering, economics, business and logistics [8]. These methods stand in contrast to "exhaustive" (global) search, such as Branch & Bound (B&B) which implicitly and/or explicitly examines all candidates in the solution space. Despite elegant functional approaches to global search [15], local search algorithms are implemented, without exception[1], using procedural languages and/or specialised constraint programming systems. We believe there are three reasons for this:

1. The premium placed on speed, leading to implementations structured as monolithic, tightly-wound iterative loops with the aim of accessing more of the space per unit of resource.
2. The extensive use of stochastic methods, which - probalistic functional programming [4] not withstanding - appear a poor match to the determinism of pure functional programming.
3. Compositional approaches have only just begun to emerge as an alternative to "monolithic" methods.

[1] The authors have not identified an implementation of local search in a pure functional language.

This paper describes a framework for *compositional* local search and an implementation as a library of combinators in Haskell [13]. We believe that a functional approach offers two particular benefits to the local search community:

1. Compositional tools: although the merits of compositional problem-solving are widely known [9], they have particular value in local search. The structure of the search space is highly problem-dependent, and as we show in the paper there is value in being able to modify the search strategy cheaply, something that can be difficult using current tools. By reconstructing local search using finer grained primitives and combinators, we also hope to develop a deeper understanding of the relationship between the structure of the problem space and effective search strategies.
2. Access to parallelism: as noted above, local search methods are usually written as highly iterative loops. Although search spaces can be distributed across machines using explicit threading, the algorithms themselves do not parallelise well without extensive restructuring, or make effective use of the now ubiquitous shared-memory multi-core architectures.

1.1 Problems

We consider only discrete combinatorial problems, such as the Travelling Salesperson Problem (TSP) [8]. Continuous problems, such as optimising variables in sets of linear constraints, can be solved effectively by the Simplex method devised by Dantzig [3], and is used by many modern systems such as AMPL [6].

1.2 Alternative Approaches

Other techniques for tackling combinatorial problems have previously been the subject of work in Haskell and these are discussed here for completeness.

Constraint Programming is an approach which focuses on the description of the solution space. A problem is modelled as a set of variables, sets of potential values for each variable and constraints that relate the variables to each other and to the objective. These models can be simplified, or *tightened*, by transformations of the constraints, however it is rarely possible to solve a problem completely by the use of this method, however when pared with a search system constraint solvers become powerful tools.

Constraint programming has been previously combined with local search in a system called COMET, described by Van Hentenryck & Michel [17]. This project has also resulted in work on parallelism within local search and combinators for describing constraint based problems for local search [18].

Constraint based techniques have also been explored in Haskell. In 2007 Lam and Sulzmann [11] created an implementation of Concurrent Constraints Handling Rules (CCHR), an efficient technique for the representation and manipulation of constraints previously implemented in Prolog. In 2009 Schrijvers, Stuckey and Wadler [14] demonstrated an alternative method for implementing constraints using Monads.

Exhaustive/Global Search are those which guarantee to explore every useful solution in a strongly structured way, a characteristic usually referred to as *completeness*. Depth First Search (DFS) and B&B utilise a tree to give structure to the exploration. In the case of B&B, heuristics derived from the problem are used to select branches of the tree which are likely to be of interest, and to determine which branches of the tree can never be of interest as soon as possible in the process. These heuristics are usually problem specific and for a new problem discovering them can be a difficult task. In addition to this difficulty, as the size of problem instances increases, B&B and DFS tend to an exponential time complexity to complete and so practical usage of them involves limiting time and taking the best solution that is found. These techniques have been investigated in Haskell by Spivey [15] and Gibbons [7].

As stated above, local search will be the focus of this paper, so these alternatives will not be discussed further in this paper.

2 Local Search Methods

As problem size increases, global search methods are unable to guarantee termination within a practical time limit. Local search sacrifices the completeness characteristic of global search, accepts that the time available may not be enough to explore the entire solution space, and instead focuses is on finding the best solutions possible within the time limit. When problem sizes are large local search provides powerful tools for producing practical results.

A local search algorithm usually has the following structure;

```
seed <- construct solution
solutions <- {}
while (completeness test) do
  solutions <- solutions union {seed}
  ns <- generate neighbourhood
  n <- selectOne ns
  seed <- n
od;
find best of solutions
```

A higher level description can be found in the standard text books such as the one by Hoos and Stützle [8]. They propose that a general definition of local search consisting of;

- a set of solutions
- a function that can return the value of a solution
- a neighbourhood function that takes a solution and returns a subset of the solution set

By analogy with global search, the set of solutions would form the nodes of a tree and the neighbourhoods give the child relationships that form the edges of the tree. Local search algorithms are not dependent on the neighbourhood

functions, but are rather differentiated from one another by how they select the next node from the current neighbourhood. Most share the idea that they only look ahead into the neighbourhood of the current seed.

2.1 A Traditional Local Search Implementation In Haskell

The traditional structure of local search can be defined in Haskell as follows;

$$
\begin{aligned}
&\textbf{type } \mathit{Price} && = \ldots \\
&\textbf{type } \mathit{Pricer}\ a && = a \to \mathit{Price} \\
&\textbf{type } \mathit{Neighbourhood}\ a && = a \to [a] \\
&\textbf{type } \mathit{LocalSearch}\ a && = \mathit{Pricer}\ a \to \mathit{Neighbourhood}\ a \\
& && \quad \to a \\
& && \quad \to [a]
\end{aligned}
$$

Pricer and *Neighbourhood* definitions are problem-dependent, so they are not discussed in detail here. While this definition of local search works well for *deterministic* local search algorithms, other approaches make use of a stochastic factor. To accommodate these, we modify the definition of *LocalSearch*, providing a stream of random numbers, as an initial parameter. One common use of randomness is to select a value from a list; we lift this operation into the function *choose*.

$$
\begin{aligned}
&\textbf{type } \mathit{Random} = \ldots \\
&\textbf{type } \mathit{LocalSearch}\ a = [\mathit{Random}] \to \mathit{Pricer}\ a \\
& \qquad\qquad\qquad\qquad\quad \to \mathit{Neighbourhood}\ a \\
& \qquad\qquad\qquad\qquad\quad \to a \\
& \qquad\qquad\qquad\qquad\quad \to [a] \\
&\mathit{choose} :: [a] \to \mathit{Random} \to a
\end{aligned}
$$

Note that an implementation of *choose* itself is deterministic; for the present the precise implementations of random and of choice is not finalised and for practical problems we will require more efficient solutions to the processes of neighbourhood generation, pricing and choosing than have currently been explored.

2.2 Common Algorithms

The four families of algorithms that have been the subject of this work so far are; *Random Walk*, *Iterative Improvers*, *TABU* and *Simulated Annealing*.

The simplest approach is the Random walk, where the next node is chosen at random at each step. Using the *traditional* formulation of local search suggested earlier, this would be described as;

$$
\begin{aligned}
&\mathit{randomWalk} :: \mathit{Ord}\ b \Rightarrow [\mathit{Random}] \to (a \to b) \to (a \to [a]) \to a \to [a] \\
&\mathit{randomWalk}\ (r : rs)\ p\ n\ x = \\
&\quad \textbf{let } ns = (n\ x)
\end{aligned}
$$

```
      c  = choose ns r
  in  x : (randomWalk rs p n c)
```

Iterative improvers operate on the principle that at each stage the solution that is moved to should give an improvement in the value of the solution. These algorithms complete when they reach a point where no improvement is possible. They will usually move towards much better solutions more rapidly than a random walk can, the members of this family are;

- *Best* improvement
- *Minimal* improvement
- *First* improvement
- *Random* improvement

As examples of these iterative improvers using the traditional formulation, here are best and random improvement.

```
bestImprovement :: Ord b ⇒ (a → b) → (a → [a]) → a → [a]
bestImprovement p n x =
  let x' = maximumBy (λa b → p a < p b) $ n x
  in  if p x' < p x
      then x : (bestImprovement p n x')
      else [x]
randomImprovement :: Ord b ⇒ [Random] → (a → b) → (a → [a]) → a → [a]
randomImprovement (r : rs) p n x =
  let ns = filter (λk → p k < p x) $ n x
      x' = choose ns r
  in if null ns
     then [x]
     else x : (randomImprovement rs p n x')
```

TABU search is a general concept for constructing local search methods rather than a single algorithm. The principle is that once you have made a change to the solution, by selecting a neighbour, you should not backtrack, as is possible using a random walk. The basic operation is a hill climbing algorithm, which does not stop when it reaches a point with no further improvement, but instead will select the least bad next neighbour not on the TABU list.

```
tabu :: Ord b ⇒ [a] → (a → b) → (a → [a]) → a → [a]
tabu q p n x =
  let ns  = n x
      ns' = filter (λa → ¬ $ elem a q) ns
  in if null ns'
     then [x]
     else x : (tabu (x : q) p n (head ns'))
```

A size limit is usually applied to the size of the list, and when the limit is reached the oldest elements of the list are removed first, making this a TABU queue.

The final variant is Simulated Annealing which is modelled after the physical process annealing. The algorithm combines characteristics of random walk and iterative improvement, allowing it to more widely explore a solution set, while still improving on the current solutions. It introduces the concept of a constantly falling *temperature* which controls the acceptance criteria of any given neighbour, relative to the current solution. When the temperature is high, a neighbour that is weaker than the current solution still has a high chance of being accepted and moved to, but as the temperature drops the algorithm will tend to only accept neighbours that improve upon the current solution.

It is commonly accepted that Simulated Annealing and TABU search are among the most successful of local search techniques. A very recent example of Simulated Annealing being used for a practical problem can be seen in this paper by Brown, Cochrane and Krom [1], where they apply the algorithm to the problem of irrigation in agriculture. A similarly recent example of TABU search in use can be seen in this paper by Venditti, Pacciarelli and Meloni [19], which deals with a scheduling problem for the pharmaceutical industry.

2.3 Limitations

A fundamental problem with this approach to local search is that it still encourages the programmer to retain the monolithic opacity of the traditional implementations, hiding similarities between related algorithms and not generalising similar characteristics. It is not obvious how to take two existing algorithms and combine them. For example, if there were two local search algorithms, both producing an infinite sequence of solutions, how could they be combined?

3 Pipeline Model Of Local Search

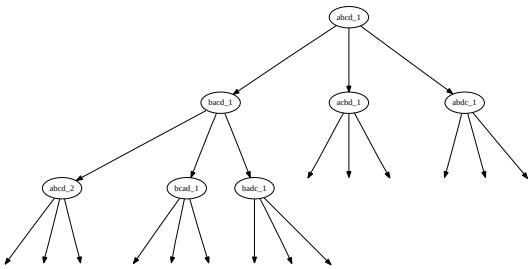


Fig. 1: A Tree Approximation To A Solution Space Graph

We initially investigated both direct functional iteration, and the use of state monads, to structure local search implementation. However neither delivered a level of clarity and composability that went significantly beyond imperative approaches. Instead, we turned to the graph model of local search for inspiration.

Rather than seeking a complete representation of the graph, we use a tree to represent the part of the space through which we are navigating (see Figure 1). Here there similarities to King and Launchbury's use of forests [10] for implementing graph algorithms. However our trees are potentially infinite, if for example two nodes are neighbours of each other, naively expanding the neighbourhoods will generate an infinite tree. In principle memoization can be used to restructure these trees as graphs. However this is not as useful as it sounds for reasons of memory residency and the *transformations* that will be carried out on the tree.

The subsequent tree is then passed to a *navigator*, a function which finds a path through the tree and yields a list of the nodes on that path. Specific algorithms can be created from combinations of different transformations and navigators. These transformations are flexible, and allow the threading of state, memory and random numbers through the trees in various ways, while retaining simplicity of expression and modularity.

3.1 The Tree Data Structure

The tree model is simple to implement. Creation of a graph is supported by a smart constructor that takes the pricing and neighbourhood functions given in section 2, and applies these to an initial solution, recursively.

```
data LSTree a = LSTree a Float [LSTree a]
makeGraph :: Neighbourhood a → Pricer a → a → LSTree a
makeGraph n p nme
  = let ns = map (makeGraph nme p) (n nme)
    in  LSTree nme (priceF nme) ns
```

Once a problem has been described using a pricing and neighbourhood function, the creation of such a graph is simple. For example, let us assume that such an efficient neighbourhood and pricing function exists for a particular instance of the TSP problem, a function that takes constructed seed solution, and gives back a tree approximation to the local search graph can now be obtained like so;

```
tspGraph = makeGraph tspNeighbourhood tspPricer
```

3.2 Navigation Of Trees

Based upon the data structure for local search trees, we can define a variety of navigators to process them, returning a list of *(solution, price)* pairs. These will all have a common definition, except for Random walk, which adds a list of Randoms;

```
type Navigator a = LSTree a → [(a, Price)]
```

– *First Choice Navigator*, this is a very simple navigator, that simply takes the first neighbour of each node

$$
\begin{aligned}
&\mathit{firstChoice}\ (\mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}) = \\
&\quad (\mathit{nme}, p) : (\mathit{firstChoice}\ (\mathit{head}\ \mathit{ns}))
\end{aligned}
$$

– *Random Walk Navigator*, this slightly more complex navigator allows the streaming of a sequence of random numbers into the process to allow the introduction of a stochastic element, for this two new types will be defined, the concept of a random value, and a list of random values

$$
\begin{aligned}
&\mathit{randomWalk} :: [\mathit{Random}] \rightarrow \mathit{LSTree}\ a \rightarrow [(a, \mathit{Price})] \\
&\mathit{randomWalk}\ (r : \mathit{rs})\ (\mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}) = \\
&\quad \textbf{let}\ c = \mathit{choose}\ \mathit{ns}\ r \\
&\quad \textbf{in}\ (\mathit{nme}, p) : (\mathit{randomWalk}\ \mathit{rs}\ c)
\end{aligned}
$$

– *First Improvement*, is a form of iterative improver, which will move to the first improvement it finds, or end if there is no improvement

$$
\begin{aligned}
&\mathit{firstImp}\ (\mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}) \\
&\qquad \mid \mathit{null}\ \mathit{imp}\ \ = [(\mathit{nme}, p)] \\
&\qquad \mid \mathit{otherwise} = (\mathit{nme}, p) : (\mathit{firstImp}\ (\mathit{head}\ \mathit{imp})) \\
&\quad \textbf{where} \\
&\qquad \mathit{imp} = \mathit{filter}\ (\lambda n \rightarrow \mathit{price}\ n < p)\ \mathit{ns}
\end{aligned}
$$

Various other navigators can also be defined, such as the variants on the iterative improvers, TABU and Simulated Annealing. However none of these navigators offer improved composability, or make the algorithms any less opaque to the programmer than the definitions given in section 2.1.

The Tree data structure does offer a limited improvement, in that it is no longer necessary to thread the pricing and neighbourhood functions through the recursive process of local search.

3.3 The Trees Give Composability

The *LSTree* provides an opportunity to refactor navigation into composable transformations over search trees. The first transformation separates the stochastic element from tree navigation. Currently the creation of an iterative improver with a stochastic element would require a new navigator, however a significant level of randomness can be achieved by shuffling the neighbourhoods of each node of the tree. This can be done using a recursive transformation of the tree. The assistant function *shuffle* can be found in Appendix A.

$$
\begin{aligned}
&\mathit{nShuffle} :: [\mathit{Random}] \rightarrow \mathit{LSTree}\ a \rightarrow \mathit{LSTree}\ a \\
&\mathit{nShuffle}\ \mathit{rs}\ (\mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}) = \textbf{let}\ (\mathit{rs}', \mathit{ns}') = \mathit{shuffle}\ \mathit{rs}\ \mathit{ns} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{ns}'' \qquad = \mathit{map}\ (\mathit{nShuffle}\ \mathit{rs}')\ \mathit{ns}' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}''
\end{aligned}
$$

This transformation can be applied to an initial tree, creating a random ordering of neighbours at each level. If it is assumed that such a process has been performed on a tree, the *randomWalk* navigator is reduced to a *first choice* navigator. Similarly to create a stochastic TABU, or random improvement local search, it is now possible to apply this shuffling process to a tree rather than creating a new navigator with the characteristic desired.

All iterative improvement algorithms share the idea that the next node must be an improvement on the current node. This can be captured by a filter on the neighbours of each node of the tree, where a neighbour is only kept if it improves on the current solution.

$$\begin{aligned}&\mathit{improvement} :: \mathit{LSTree}\ a \to \mathit{LSTree}\ a\\&\mathit{improvement}\ (\mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}) = \mathbf{let}\ \mathit{ns}' = \mathit{filter}\ (\lambda n \to \mathit{price}\ n < p)\ \mathit{ns}\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{LSTree}\ \mathit{nme}\ p\ \mathit{ns}'\end{aligned}$$

Like the *nShuffle* transformation for the tree, the *improvement* transformation removes the need for the first improvement navigator, reducing it to the first choice navigator. It is also possible to compose these two transformations to yield a random improvement algorithm;

$$\begin{aligned}&\mathit{rImprovement} :: \mathit{Randoms} \to \mathit{LSTree}\ a \to \mathit{LSTree}\ a\\&\mathit{rImprovement}\ \mathit{rs} = (\mathit{nShuffle}\ \mathit{rs}) \circ \mathit{improvement}\end{aligned}$$

To create the other two forms of improvement, maximal and minimal, a new transformation is needed, one that selects only one element of a neighbourhood and replaces the neighbourhood by this single element.

$$\begin{aligned}&\mathit{selectOne}\ f\ (\mathit{LSTree}\ n\ p\ \mathit{ns}) = \mathbf{let}\ \mathit{ns}' = [\mathit{selectOne}\ f\ (f\ \mathit{ns})]\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{LSTree}\ n\ p\ \mathit{ns}'\end{aligned}$$

This combinator can then be used with the standard *maximumBy* and *minimumBy* functions from the *Data.List* library to describe maximal and minimal improvement algorithms;

$$\begin{aligned}&f = (\lambda x\ y \to (\mathit{value}\ x) < (\mathit{value}\ y))\\&\mathit{maxImp} = (\mathit{selectOne}\ (\mathit{maximumBy}\ f)) \circ \mathit{improvement}\\&\mathit{minImp} = (\mathit{selectOne}\ (\mathit{minimumBy}\ f)) \circ \mathit{improvement}\end{aligned}$$

It should be noted that once a *selectOne* transformation has been performed there is little more that can be done to a tree using these transformations, due to the neighbourhood being reduced to a singleton.

A very basic TABU search uses a simple list as its history of previously visited nodes. Such a list can be threaded through a transformation process of a tree, filtering each neighbourhood based upon the nodes that would have to be visited to reach that point.

$$\begin{aligned}&\mathit{basicT}\ ::\ \mathit{Eq}\ a \Rightarrow [a] \to \mathit{LSTree}\ a \to \mathit{LSTree}\ a\\&\mathit{basicT}\ q\ (\mathit{LSTree}\ n\ p\ \mathit{ns}) = \mathit{LSTree}\ n\ p\ \mathit{ns}''\end{aligned}$$

where

$$\begin{aligned} q' &= n : q \\ ns' &= map\ (basicTABU\ q')\ ns \\ ns'' &= filter\ (\lambda(LSTree\ x\ \_\ \_) \rightarrow \neg\ (elem\ x\ q'))\ ns' \end{aligned}$$

It should be noted that this is not TABU search, but rather a form of TABU filter. Usually a TABU filter limits the history and it becomes a TABU queue, defined like this.

$$basicTABUQ :: Eq\ a \Rightarrow Int \rightarrow [a] \rightarrow LSTree\ a \rightarrow LSTree\ a$$

Traditional implementations of TABU are more sophisticated, comparing only characteristics of nodes and not whole names, however these ideas have not been fully explored for this paper.

Using this TABU transformation, it is now possible to describe a stochastic TABU search, or a simple hill climbing TABU search like so:

$$\begin{aligned} stochasticT\ rs &= (basicTABUQ\ 5\ [\,]) \circ (nShuffle\ rs) \\ climbingT\ \ \ \ &= (selectOne\ (maximumBy\ f)) \circ (basicTABUQ\ 5\ [\,]) \end{aligned}$$

3.4 Simulated Annealing

The final common algorithm, Simulated Annealing, has not been discussed so far, because thus far it has resisted attempts to clearly describe it using this model. Implementations have been created, though the transformations used are more specific than those used to define the other common algorithms.

The Simulated Annealing algorithm introduces a *temperature* element in addition to a stochastic element that must be threaded through the transformation process. This can be done by *zipping* together a list of the temperatures with a list of random values, as seen below. The function *saAccept* can be found in appendix A.

type $Temperature = ...$

$$saSelect :: [(RValue, Temperature)] \rightarrow LSTree\ a \rightarrow LSTree\ a$$
$$saSelect\ ((r, t) : rts)\ (LSTree\ n\ p\ ns) = LSTree\ n\ p\ [c']$$

where

$$\begin{aligned} c &= head\ (dropWhile\ acc\ ns) \\ acc\ x &= saAccept\ r\ t\ (value\ x - p) \\ c' &= saSelect\ rts\ c \end{aligned}$$

The temperature will usually drop in a regular way for each step of the process, this can be modelled with an iteration, which returns a list of Floats. This selection process can now be used by zipping the temperatures with a stream of random numbers, giving this;

$$saTemp\ p\ iTemp = iterate\ (*p)\ iTemp$$

$f\ rs\ i = saSelect\ (zip\ rs\ (saTemp\ 0.9\ 100))\ i$

Like the other selection transformations, this is likely to be the last transformation that will be carried out in the implementation of any local search algorithm, due to the neighbourhoods being reduced to a singleton.

3.5 The Pipeline

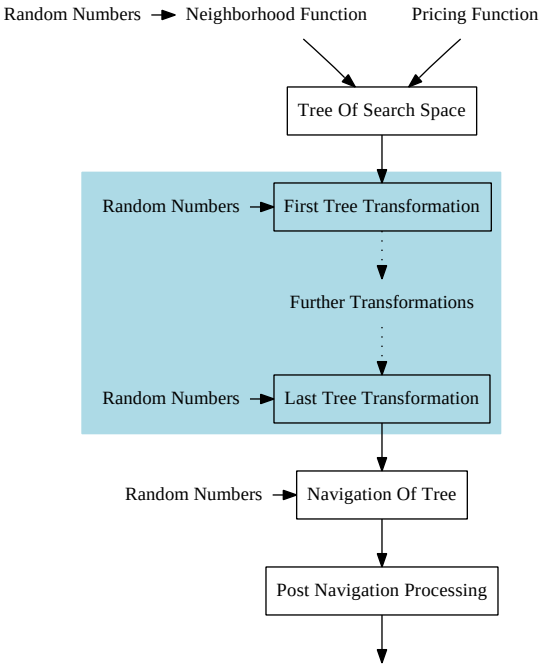


Fig. 2: The Pipeline Model Of Local Search

The process of local search can now be understood as a pipeline, illustrated in Figure 2. The pipeline begins with the construction of an initial tree from a neighbourhood function and a pricing function, and this is followed by a series of transformations, which may or may not thread state and random numbers into the process. Once all the transformations have been completed the pipeline ends with the navigation of the tree, which may also stream in random numbers, and any further processing of the list of solutions that the process has discovered.

This final step is application specific. It may simply be taking the best solution using standard Haskell functions, or it could involve performing a more in depth statistical analysis of the solutions investigated.

4 Performance Analysis

The goal of this work was to gain confidence that the approach chosen was expressive enough to capture a range of local search strategies. None the less it is useful to examine performance, to highlight where work should be focussed in the future. To this end a "first found" iterative improvement algorithm was selected to obtain a first cut evaluation of the library, as compared to a version written in C. The C implementation was written by the authors.

The instances of the TSP selected for the investigation were of two types, all drawn from the TSPLIB [16] repository maintained by Ruprecht-Karls Universität. Most were symmetric TSPs, based on geographic data, however others were symmetric or asymmetric based upon other problems.

To test performance 100 starting solutions were randomly constructed for each problem instance, and each starting solution was then used as the seed solution for a local search and the time taken for all 100 to complete was recorded using the Linux *time* command. The same starting solutions were fed to each implementation of the local search, to control the conditions of the experiment as fully as possible.

The Haskell system used was the *GHC, 6.10.3* with the -O2 Flag. The C compiler used was the *GCC 4.3.0 20080428 (Red Hat 4.3.0-8)*. The Java version used was *1.6.0_10* for both runtime and compiler. The machine used for execution had 2GB of ram and an Intel Core2 CPU running at 2.13 GHz, with the Fedora 9 operating system, with a Linux Kernel version 2.6.27.28.

4.1 Runtime Performance

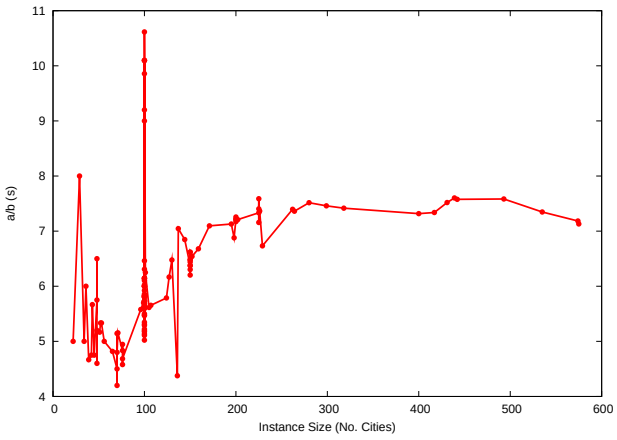


Fig. 3: A graph of the ratio of C program performance to the Haskell pipeline performance, against problem size

The graph in figure 3 shows the ratio of performance of the simple imperative improvement algorithm written in C to the same algorithm written in Haskell using the new library, with problem size on the X axis.

On the surface this indicates that the C version is approximately 8 times faster than the Haskell version. It is still unclear why the jagged pattern occurs in this graph. One hypothesis is that it is related to the garbage collection and memory management system of Haskell, suggesting that for some specific problem instances significantly more of the neighbourhood must be evaluated than in others, causing Haskell to allocate and deallocate more memory than is required to evaluate other problems.

4.2 Memory Profiling

Profiling of the Haskell version of the local search program, compiled for a single core machine only, has been done to check where memory is being used and what for. Peak memory use appears commensurate with the scale of the problem, however total memory allocation and deallocation is significantly worse (up to 2000 times), suggesting a high churn and overhead from the garbage collection.

This high turnover is caused by the creation of neighbourhoods and the subsequent freeing up of the memory once a single choice is made and the algorithm moves on. This suggests that the data structure is resisting memory optimisation that the GHC may be trying to use, such as fusion techniques.

4.3 Expression Of Local Search

	Value Mean	Itterations Mean
Random Walk	4611.32	1000.0
Max Improv	3577.35	11.16
Min Improv	3739.23	103.83
Rev Min Improv	4714.04	1.00
TABU 10	4461.17	200.0
Max TABU 10	2698.78	200.0

Fig. 4: Statistical analysis of characteristics of local search algorithms.

To illustrate the ease of expressing different variations of local search using the library several different combinations of transformations were chosen. All the algorithms used the simple navigator for final evaluation. The problem chosen was an asymmetric TSP problem from the TSPLIB. For each combination of transformations 1000 tests were run and means of the values and the number of iterations required were calculated, these results can be seen in figure 4.

The techniques chosen were making use of shuffling transformations, maximum and minimum selections, and a TABU queue, taking a parameter that indicates the maximum size of the queue during the transformation;

1. **Random Walk**, was chosen to be the baseline and is expressed using the recursive neighbourhood shuffle. Since this would be an infinite process a limit of 1000 solutions was taken from the result of the navigation and the best taken as the result.
2. **Maximal Improvement**, selects the best solution in the neighbourhood, when that solution is an improvement on the current solution, and finds better solutions than a random walk in far fewer steps.
3. **Minimal Improvement**, tries to climb slowly rather than greedily.
4. **Reversed Minimal Improvement**, changing the order of composition of the components used for the minimal improvement can radically change the character of the algorithm, as is seen here.
5. **TABU 10**, changes the basic transformation from *improvement* to the TABU queue filter, with a maximum size of 10. Alone this is an ineffective approach, much worse than the improvement strategies, but slightly better than the pure random walk. Like random walk this is expected to be an unending process and so a limit was fixed at 200 iterations, with the actual result being the best of the 200 solutions examined.
6. **Maximal TABU 10**, by extending the basic TABU filter with the selection of the best possible element, TABU search becomes the best search method seen.

One consideration regarding these results was the selection of the limit on the iterations for the infinite processes of the Random Walk and TABU search, where the number of solutions being examined is significantly higher than those for the improvement algorithms. In light of this some additional sensitivity testing was carried out, changing the number of iterations available to these strategies. It was found that as the number of iterations was reduced the performance of the TABU search also reduced, however even at 30 iterations the Max TABU variant still outperformed all other variants, but by a smaller margin.

5 Conclusion

This paper has shown that a flexible, modular local search library is possible in Haskell by using a tree data structure and transformations of that data structure as a framework. The library presented has been shown to support several of the most common algorithms used in local search and constructs these algorithms by composition of independent components.

Other types of transformations on trees have been considered by the authors such as neighbourhood enrichment and annotation of the trees. These will be investigated more fully to evaluate their potential uses in the expression of characteristics of various local search algorithms.

At this early stage of the work it is the belief of the authors that it will be more productive to take a step back from the current library and re-evaluate the field of local search itself. This activity would consist of a more extensive survey of the techniques of local search, including a more comprehensive investigation of

the variations on TABU search and the population based approaches of swarms and evolutionary algorithms.

The objective of creating the taxonomy will be the identification of key shared functional characteristics. To provide the best approach to implementation various techniques within Haskell are being considered including State Monads and the work of Fischer, Kiselyov and Shan on Non-deterministic programming [5].

Once the authors are comfortable with the expressiveness of the library, attention will be turned to the performance characteristics it engenders within algorithms it is used for. It has been shown that for a simple local search algorithm an implementation in the library has the same asymptotic performance as an imperative implementation in C, however the raw performance is rather weaker. It is felt that various techniques being explored by the Haskell community will be investigated in this phase of the project, including stream fusion [2] and supercompilation [12].

References

1. Brown, P.D., Cochrane, T.A., Krom, T.D.: Optimal on-farm irrigation scheduling with a seasonal water limit using simulated annealing. Agricultural Water Management 97(6), 892–900 (2010)
2. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: from lists to streams to nothing at all. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. pp. 315–326. ACM, New York, NY, USA (2007)
3. Dantzig, G.: Linear Programming and Extensions. Princeton University Press (1963)
4. Erwig, M., Kollmansberger, S.: Functional pearls: Probabilistic functional programming in haskell. J. Funct. Program. 16(1), 21–34 (2006)
5. Fischer, S., Kiselyov, O., Shan, C.c.: Purely functional lazy non-deterministic programming. In: ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. pp. 11–22. ACM, New York, NY, USA (2009)
6. Gay, R.F.D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Brooks/Cole Publishing Company / Cengage Learning, 2nd edn. (2002)
7. Gibbons, J.: Algebras for Tree Algorithms. D. Phil. thesis, Programming Research Group, Oxford University (1991), available as Technical Monograph PRG-94. ISBN 0-902928-72-4
8. Hoos, H., Sttzle, T.: Stochastic Local Search: Foundations & Applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
9. Hughes, J.: Why functional programming matters pp. 17–42 (1990)
10. King, D.J., Launchbury, J.: Structuring depth-first search algorithms in haskell. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 344–354. ACM, New York, NY, USA (1995)
11. Lam, E.S.L., Sulzmann, M.: A concurrent constraint handling rules implementation in haskell with software transactional memory. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. pp. 19–24. ACM, New York, NY, USA (2007)

12. Mitchell, N., Runciman, C.: A supercompiler for core haskell pp. 147–164 (2008)
13. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. J. Funct. Program. 13(1), 0–255 (Jan 2003)
14. Schrijvers, T., Stuckey, P.J., Wadler, P.: Monadic constraint programming. J. Funct. Program. 19(6), 663–697 (2009)
15. Spivey, J.m.: Algebras for combinatorial search. J. Funct. Program. 19(3-4), 469–487 (2009)
16. TSPLIB. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`, accessed June 2009
17. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
18. Van Hentenryck, P., Michel, L., Liu, L.: Contraint-based combinators for local search. Constraints 10(4), 363–384 (2005)
19. Venditti, L., Pacciarelli, D., Meloni, C.: A tabu search algorithm for scheduling pharmaceutical packaging operations. European Journal of Operational Research 202(2), 538–546 (2010)

A Additional Code

```
shuffle :: [Float] → [a] → ([Float], [a])
shuffle rs xs = (rest, map snd (sortBy tupleCompare (zip use xs)))
where
  (use, rest) = splitAt (length xs) rs
  tupleCompare (x, _) (y, _) = compare x y
```

```
saAccept :: Float → Float → Float → Bool
saAccept r t d | p > 1      = True
               | otherwise = p > r
where
  p = exp (d / t)
```

Modular Components with Monadic Effects

Tom Schrijvers[1] and Bruno C. d. S. Oliveira[2]

[1] Katholieke Universiteit Leuven
tom.schrijvers@cs.kuleuven.be
[2] ROSAEC Center, Seoul National University
bruno@ropas.snu.ac.kr

Abstract. Limitations of monad stacks get in the way of developing highly modular programs with effects. This paper demonstrates that Functional Programming's abstraction tools are up to the challenge. Of course, abstraction must be followed by clever instantiation: Huet's zipper for the monad stack makes components jump through unanticipated hoops.

1 Introduction

Features like higher-order functions and parametric polymorphism make functional languages very suitable for developing highly modular code (libraries or components) that can be reused in many different settings.

Purely functional languages like Haskell, also provide good means to reason about (modular) code, for instance through equational reasoning and parametricity. Purity bans implicit side-effects because it destroys the reasoning capabilities. Instead, monads are the preferred way for introducing explicit effects, without losing equational reasoning and parametricity.

Unfortunately, the combination of modularity and monads is currently the Achilles heel of the purely functional approach. Monads seem to be subpar to imperative object- and aspect-oriented languages for highly modular and effectful applications. In fact, functional programmers do not easily explore such applications, because working with monads quickly becomes "awkward".

Monad transformers are the most prominent work on modular monads. However, they only provide limited modularity. Only components with distinct effects can be composed without problems. This imposes a serious restriction on the number of effectful components that can be composed while reusing existing monad transformer implementations.

In this article we provide solutions to the construction of highly modular software with monadic effects. Our approach does not impose restrictions on the number of components or the reuse of existing monad implementations. It neither requires the component implementation to anticipate particular system configurations nor does it restrict the eventual configurations.

2 Setting

To illustrate our approach, we consider a setting that provides modular effects through monad transformers and modular components through open recursion:

- We use the monad transformer variant defined by the Monatron library (1) that provides an essential operation on monad transformer stacks. This operation can be defined for other transformer libraries too, but would have to be done from scratch.
- We use the mixin approach to open recursion pioneered by Cook (2), because it provides a minimal yet highly expressive approach (e.g. in terms of control flow) to tightly coupled components. Alternative approaches in terms of type classes and imposed control flow patterns are possible too.

2.1 Monad Transformers in Monatron

We assume that the reader is already familiar with the general concept of monad transformers, and summarize here only the particulars of Monatron. The type class $MonadT\ t$ expresses that t is a monad transformers. This means that t provides the usual lifting functionality $lift :: m\ a \to t\ m\ a$. Furthermore, it also supplies a bind operator $tbind :: t\ m\ a \to (a \to t\ m\ b) \to t\ m\ b$ for the transformed monad; and $treturn :: a \to t\ m\ a$, which is simply defined by default as $lift \circ return$. A distinguishing feature of Monatron's monad transformers is the *tmixmap* method:

$$\begin{aligned} tmixmap\ &:: (Monad\ m, Monad\ n) \\ &\Rightarrow (forall\ a \circ m\ a \to n\ a) \\ &\to (forall\ b \circ n\ b \to\ m\ b) \to t\ m\ c \to t\ n\ c \end{aligned}$$

The *tmixmap* operation takes a natural isomorphism—two natural transformations, from the monad functor m to the monad functor n and vice-versa, that are each other's inverse—and returns a natural transformation from the monad functor $t\ m$ to the monad functor $t\ n$. In other words, *tmixmap* is an operation similar to the *fmap* operation of the *Functor* class, but mapping the monad functor instead. However, unlike *fmap* the (higher-order) functor t can have both co-variant and contra-variant occurrences (for the continuation monad transformer in particular) of m, which explains the need for the natural isomorphism.

Monad Transformer Implementations Monatron provides the usual range of monad transformer implementations, summarized in Figure 1.

As an example of the underlying implementation approach, consider the state monad with its methods for reading and writing the state. Monatron provides an explicit *dictionary* type

type *MakeWith s m*

that encapsulates the functionality for accessing a state of type s in monad m. The actual methods can be retrieved from this dictionary with helper functions:

```
  -- identity monad
newtype 𝕀 a
𝕀      :: a → 𝕀 a
run𝕀 :: 𝕀 a → a

  -- identity transformer
newtype 𝕀T m a
𝕀T     :: m a → 𝕀T m a
run𝕀T :: 𝕀T m a → m a

  -- reader transformer
newtype ℝT e m a
ℝT     :: (e → m a) → ℝT e m a
runℝT :: e → ℝT e m a → m a

  -- reader class
class Monad m ⇒ ℝM e m | m → e
ask    :: ℝM e m ⇒ m e

  -- state transformer
newtype 𝕊T s m a
𝕊T     :: (s → m (a, s)) → 𝕊T s m a
run𝕊T :: s → 𝕊T s m a → m (a, s)

  -- state class
class Monad m ⇒ 𝕊M s m | m → s
get    :: 𝕊M s m ⇒ m s
put    :: 𝕊M s m ⇒ s → m ()

  -- exception transformer
newtype 𝔼T x m a
𝔼T     :: m (Either x a) → 𝔼T x m a
run𝔼T :: 𝔼T x m a → m (Either x a)

  -- exception class
class Monad m ⇒ 𝔼M x m | m → x
throw :: 𝔼M x m ⇒ x → m a
```

Fig. 1. Monatron quick reference.

```
getX :: Monad m ⇒ MakeWith s m → m s
putX :: Monad m ⇒ MakeWith s m → s → m ()
```

In addition, the state functionality can be lifted through other monad transformers that reside above the state transformer in the monad stack. The lifting is uniform: there is a single implementation for lifting the state transformer methods through all other monad transformers:

```
liftMakeWith :: (Monad m, MonadT t) ⇒
                MakeWith z m → MakeWith z (t m)
```

For convenience Monatron uses Haskell's type class mechanism to make the dictionaries implicit. Figure 2 shows how this is done for the state monad. The first instance implements the specific functionality for the $\mathbb{S}_T$ monad using *makeWithStateT*. The second instance is more interesting as it shows how Monatron makes use of *liftMakeWith* to achieve uniform lifting through *any* monad transformer t. Note that other monad transformers are implemented in essentially the same way: one instance provides the functionality specific to the particular monad in question; whereas another instance provides uniform lifting.

```
class Monad m ⇒ 𝕊M z m | m → z where
  stateM :: MakeWith z m
instance Monad m ⇒ 𝕊M z (𝕊T z m) where
  stateM = makeWithStateT
instance (𝕊M z m, MonadT t) ⇒ 𝕊M z (t m) where
  stateM = liftMakeWith stateM
get :: 𝕊M z m ⇒ m z
get = getX stateM
put :: 𝕊M z m ⇒ z → m ()
put = putX stateM
```

Fig. 2. Overloading of state operations in Monatron

2.2 Mixins and Open Recursion

We briefly summarize the notion of mixins, and refer the interested reader to previous literature on the topic for a more indepth treatment. The basis of our mixin implementation in Haskell is:

```
type Mixin s = s → s
fix :: Mixin s → s
fix a = a (fix a)
(⊛) :: Mixin s → Mixin s → Mixin s
a1 ⊛ a2 = λproceed → a1 (a2 proceed)
```

The type *Mixin s* is a synonym for a function with type $s \to s$ representing open recursion. The parameter of that function is called a *join point*, that is, the point in the component at which another component is added. The operation $\circledast$ defines component composition. The function *fix* is a fixpoint combinator used for closing, or sealing, an open and potentially composed component.

Consider the following open functions:

```
fib1 :: Mixin (Int → Int)
fib1 proceed n = case n of
  0 → 0
  1 → 1
```

```
advfib :: Mixin (Int → Int)
advfib proceed n = case n of
  10 → 55
  _  → proceed n
```

The open function fib_1 defines the standard fibonacci function, except that recursive calls are replaced by *proceed*. The open function *advfib* optimizes one call of the fibonacci function by returning the appropriate value immediately. Different combinations of open functions are closed through *fix*:

```
slowfib1 , optfib :: Int → Int
slowfib1 = fix fib1
optfib   = fix (advfib ⊛ fib1 )
```

2.3 Effectful Components

Components with effects are obtained by combining mixins and monads. For instance, consider the following memoization component and a monadic fibonacci function.

```
memo :: 𝕊M (Map Int Int) m ⇒ Mixin (Int → m Int)
memo proceed x =
  do m ← get
     if member x m then return (m ! x)
                   else do y ← proceed x
                           m′ ← get
                           put (insert x y m′)
                           return y
fib2 :: Monad m ⇒ Mixin (Int → m Int)
fib2 proceed n = case n of
  0 → return 0
  1 → return 1
  _ → do y ← proceed (n − 1)
         x ← proceed (n − 2)
         return (x + y)
```

We can instantiate different monads, using the corresponding run functions of Figure 1, to recover variations of the fibonacci function. For example, the identity monad recovers the effect-free function

```
slowfib2 :: Int → Int
slowfib2 = run𝕀 ∘ fix fib2
```

while a fast fibonacci function is obtained by adding the memo advice and suitably instantiating the state monad:

```
fastfib :: Int → Int
fastfib n = run𝕀 ∘ eval𝕊T empty $ fix (memo ⊛ fib2) n
```

where

```
eval𝕊T :: Monad m ⇒ s → 𝕊T s m a → m a
eval𝕊T s m = run𝕊T s m ≫= return ∘ fst
```

Another component for profiling is

```
prof :: 𝕊M Int m ⇒ Mixin (a → m b)
prof proceed x =
  do c ← get
     put (c + 1)
     proceed x
```

which allows us to count the number of calls to the fibonacci function

```
proffib n = run𝕀 ∘ eval𝕊T 0 $ fix (prof ⊛ fib2) n
```

Transformer Conflicts Of course, we would also like to profile the memoized fibonacci function to get an idea of how much more efficient it is.

$$\begin{aligned}&\mathit{profmemofib} :: \mathit{Int} \to \mathbb{S}_{\mathrm{T}}\ \mathit{Int}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ \mathbb{I})\ \mathit{Int}\\&\mathit{profmemofib} = \mathit{fix}\ (\mathit{prof} \circledast \mathit{memo} \circledast \mathit{fib}_2)\end{aligned}$$

Unfortunately, the type checker complains that *Int* and *Map Int Int* are distinct types. The problem is that there are two uses of *get* in our features: one in *evalMem*; and another in *evalVar*. Due to automatic lifting, both *get* methods read the state from the same top-level $\mathbb{S}_{\mathrm{T}}$, which happens to contain an *Int* value. This is the right thing to do for *prof*, but wrong for *memo* that expects a value of type *Map Int Int*.

This type error is only a symptom of the real problem though. Namely, we expect the *get* calls in *prof* and *memo* to pick, or automatically lift out, different states in the monad stack, but the type checker does not distinguish between the two calls.

The lifting is biased towards the top of the monad stack. If the stack contains two $\mathbb{S}_{\mathrm{T}}$ instances, then the top one is in focus. Obviously, we cannot simply rearrange the layers in the monad stack to fix the problem, because this also alters the semantics and still the bottom instance remains inaccessible.

In Liang et al.'s modular interpreters this problem is solved using *lift* methods to explicitly disambiguate the access to the monad stack. However, this solution would not work for us because, unlike Liang et al., we are interested in having modular components that can be reused in several different configurations; and where potentially many different interpreters can coexist at the same time. The use of *lift* entails adapting existing code for the library components, which is fine when a single instance of a modular interpreter is in use, but it leads to fundamentally incompatible components when multiple interpreters with different configurations exist.

A dilemma At this stage it seems that we are left with a dilemma. On the one hand automatically lifted methods like *get* are nice because they do not pollute the code and they interact well with abstraction, implicitly lifting the monad into the right layer. Unfortunately, they do not allow multiple instances of the same monad in the monad stack, which is just too constraining for realistic applications. On the other hand explicit *lift* methods are nice to disambiguate uses of automatically lifted methods, which allows multiple monads of the same type to be used in a component. However *lift* methods can also lead to a significant loss of abstraction and reuse.

Is there a way out of this dilemma?

3 The Monad Zipper

We want to combine multiple instances of the same monad without touching the library components. In order to have our cake and eat it too, we must make the most of the provided abstraction. Indeed, we can influence the behavior

of the library components from the outside by instantiating the type variables appropriately. Of course, doing so in the obvious way did not get us anywhere earlier. So we need to reconsider what we expect from the instantiation: it should focus the automatic lifting to the desired layer in the stack.

3.1 Stacks and Zippers

Sometimes type-level problems get easier when we shift them to the value level. Let's reify the structure of the monad stack in a data type

$$\begin{aligned}&\textbf{data}\ \mathit{Stack} &&= \mathit{Push}\ \mathit{Trans}\ \mathit{Stack} \mid \mathit{Bottom}\ \mathit{Monad}\\ &\textbf{data}\ \mathit{Trans} &&= T_1 \mid \ldots \mid T_n\\ &\textbf{data}\ \mathit{Monad} &&= M_1 \mid \ldots \mid M_n\end{aligned}$$

where the T_i represent the different transformers and M_i are plain monads like $\mathbb{I}$.

(3) taught us how to shift the focus to any position in a data structure, with his *zipper*. Here is the *Zipper* for *Stack*:

$$\begin{aligned}&\textbf{data}\ \mathit{Zipper} &&= \mathit{Zipper}\ \mathit{Path}\ \mathit{Trans}\ \mathit{Stack}\\ &\textbf{data}\ \mathit{Path} &&= \mathit{Pop}\ \mathit{Trans}\ \mathit{Path} \mid \mathit{Top}\end{aligned}$$

where $\mathit{Zipper}\ p\ l\ s$ denotes a stack with layer l in focus, remainder of the stack s and path p back to the top of the stack. The path is a reversed list, where the first element is closest to the layer in focus and the last element is the top of the stack.

The *zipper* function turns a stack into a zipper with the first element in focus:

$$\begin{aligned}&\mathit{zipper} :: \mathit{Stack} \to \mathit{Zipper}\\ &\mathit{zipper}\ (\mathit{Push}\ t\ s) = \mathit{Zipper}\ \mathit{Top}\ t\ s\end{aligned}$$

while the *up* and *down* functions allow shifting the focus one position up or down:

$$\begin{aligned}&\mathit{up}, \mathit{down} :: \mathit{Zipper} \to \mathit{Zipper}\\ &\mathit{up}\ (\mathit{Zipper}\ (\mathit{Pop}\ t_1\ p)\ t_2\ s) &&= \mathit{Zipper}\ p\ t_1\ (\mathit{Push}\ t_2\ s)\\ &\mathit{down}\ (\mathit{Zipper}\ p\ t_1\ (\mathit{Push}\ t_2\ s)) &&= \mathit{Zipper}\ (\mathit{Pop}\ t_1\ p)\ t_2\ s\end{aligned}$$

It's all well and good to zip around a reified form of the monad stack, but can we do it on the real thing too?

3.2 Monad Zipper

The answer is yes. Here is how the monad zipper ($\triangleright$) is defined:

$$\textbf{newtype}\ (t_1 \triangleright t_2)\ m\ a = \mathbb{Z}_{\mathrm{T}}\ \{\mathit{run}\mathbb{Z}_{\mathrm{T}} :: t_1\ (t_2\ m)\ a\}$$

where the type $(p \triangleright t)\ s$ corresponds to the reified data structure $\mathit{Zipper}\ p\ t\ s$. However, the monad zipper only changes the type representation: the newtype

indicates that no actual structural change to the monad stack $t_1\ (t_2\ m)$ takes place. Even though the zipper shifts the focus to t_2, it does keep t_1 around, which is essential for recursive calls that need to shift the focus back to t_1. Such recursive calls are impossible using *lift* because the information about t_1 is lost in lifted code.

Mixing Stack and Zipper The type system will not allow values of type *Zipper* to be used when values of type *Stack* are expected. This segregation is not the case at the type level: the monad zipper type ($\triangleright$) can appear as part of a monad stack. Indeed, we define $t_1 \triangleright t_2$ to be a monad transformer that is the composition of t_1 and t_2:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{MonadT}\ t_1, \mathit{MonadT}\ t_2)\\
&\quad \Rightarrow \mathit{MonadT}\ (t_1 \triangleright t_2)\ \textbf{where}\\
&\qquad \mathit{lift} \qquad\quad = \mathbb{Z}_{\mathrm{T}} \circ \mathit{lift} \circ \mathit{lift}\\
&\qquad \mathit{tbind}\ m\ f \quad = \mathbb{Z}_{\mathrm{T}}\ \$\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ m \ggg\!= \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ f\\
&\qquad \mathit{tmixmap}\ f\ g = \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ (\mathit{tmixmap}\ f\ g)\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{tmixmap}\ g\ f) \circ \mathit{run}\mathbb{Z}_{\mathrm{T}}
\end{aligned}
$$

Focus The interesting behavior of $t_1 \triangleright t_2$, where it deviates from a plain monad transformer composition, lies in the methods of the monad classes: for looking up the method implementations it ignores (looks through) t_1 and only considers $t_2\ m$.

Consider the state monad $\mathbb{S}_{\mathrm{M}}$ defined by Figure 2. In the case of the monad zipper transformer $t_1 \triangleright t_2$, t_2 should be on focus and t_1 should be ignored. Thus adopting the uniform lifting functionality provided by the second instance would be the wrong thing to do. With such implementation, the definition of *stateM* would be equivalent to:

$$\mathit{stateM} = \mathit{isoMakeWith}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ \mathit{stateM}$$

using an auxiliary function

$$
\begin{aligned}
&\mathit{isoMakeWith}\ ::\ (\forall a.m\ a \to n\ a) \to (\forall a.n\ a \to m\ a)\\
&\qquad\qquad \to \mathit{MakeWith}\ s\ m \to \mathit{MakeWith}\ s\ n
\end{aligned}
$$

that changes a *MakeWith* dictionary using a given monad isomorphism. The above *stateM* code would merely adopt the $\mathbb{S}_{\mathrm{M}}$ implementation of $t_1\ (t_2\ m)$ for $(t_1 \triangleright t_2)\ m$ through the $(\mathbb{Z}_{\mathrm{T}}, \mathit{run}\mathbb{Z}_{\mathrm{T}})$ monad isomorhism, and thereby prefer the $\mathbb{S}_{\mathrm{M}}$ implementation of t_1 before any in $t_2\ m$. However, that is not the desired behavior for the monad zipper. Instead, using *liftMakeWith*, we lift the *stateM* implementation of $t_2\ m$ through t_1, *ignoring* any possible *stateM* implementation available for t_1.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{MonadT}\ t_1, \mathit{MonadT}\ t_2, \mathit{Monad}\ m,\\
&\quad \mathbb{S}_{\mathrm{M}}\ s\ (t_2\ m)) \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ ((t_1 \triangleright t_2)\ m)\ \textbf{where}\\
&\qquad \mathit{stateM} =\\
&\qquad\quad \mathit{isoMakeWith}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ (\mathit{liftMakeWith}\ \mathit{stateM})
\end{aligned}
$$

Example Now let's have a look at how to actually use the zipper monad. Consider the following two examples. The first example runs $put\ 1$ in the regular stack $\mathbb{S}_\mathrm{T}\ Int\ (\mathbb{S}_\mathrm{T}\ Int\ \mathbb{I})$, and hence updates the state of the topmost $\mathbb{S}_\mathrm{T}$ transformer. The second shifts the focus to the other $\mathbb{S}_\mathrm{T}$ transformer with a monad stack of the form $(\mathbb{S}_\mathrm{T}\ Int \rhd \mathbb{S}_\mathrm{T}\ Int)\ \mathbb{I}$.

$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ put\ 1$
$(((), 1), 0)$
$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{Z}_\mathrm{T}\ \$\ put\ 1$
$(((), 0), 1)$

On the surface, $run\mathbb{Z}_\mathrm{T}$ does not provide any expressive power over *lift*:

$> run\mathbb{I}\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ run\mathbb{S}_\mathrm{T}\ 0\ \$\ lift\ \$\ put\ 1$
$(((), 0), 1)$

where $put\ 1$ has type $\mathbb{S}_\mathrm{T}\ Int\ \mathbb{I}\ ()$. Note that the topmost $\mathbb{S}_\mathrm{T}\ Int$ does not appear in this type. In contrast, unlike *lift*, $run\mathbb{Z}_\mathrm{T}$ does not lose information about any monadic layers. Despite the deceiving similarity between $run\mathbb{Z}_\mathrm{T}$ and *lift*, we will see that $run\mathbb{Z}_\mathrm{T}$ has great advantages when it comes to modular components. First though, we further develop the correspondence between Huet's zipper and our monad zipper.

Relative Navigation Suppose we have a monad transformer stack $t_1\ (t_2 ... (t_n\ m))$. Then the focus lies by default on the topmost transformer t_1. Analogous to what the *zipper* function does with *Stack*, we can change this monad transformer stack into explicit zipper form:

$$zipper :: t\ m\ a \to (\mathbb{I}_\mathrm{T} \rhd t)\ m\ a$$
$$zipper = \mathbb{Z}_\mathrm{T} \circ \mathbb{I}_\mathrm{T}$$

where the identity monad transformer $\mathbb{I}_\mathrm{T}$ acts as the *Top* sentinel. However, this change is entirely unnecessary: $t\ m\ a$ and $(\mathbb{I}_\mathrm{T} \rhd t)\ m\ a$ have exactly the same automatic lifting behavior. Indeed, we have added $\mathbb{I}_\mathrm{T}$ to subsequently ignore it again with $\mathbb{I}_\mathrm{T} \rhd t$.

The monad zipper becomes useful only when we shift the focus away from t_1 to t_2. We have already seen that $\mathbb{Z}_\mathrm{T}$ accomplishes that shift of focus, but how can we navigate further down, and back up?

Let us start with moving the focus one step further down:

$$step2to3 :: (t_1 \rhd t_2)\ (t_3\ m)\ a \to (? \rhd t_3)\ m\ a$$

What should come in the place of the question mark? Following Huet's zipper, we should push t_2 on a reversed stack that already contains t_1. Pleasingly, if we denote this reversed stack as $t_1 \rhd t_2$, we obtain the following very simple implementation for *step2to3*:

$$step2to3 :: (t_1 \rhd t_2)\ (t_3\ m)\ a \to (t_1 \rhd t_2 \rhd t_3)\ m\ a$$
$$step2to3 = \mathbb{Z}_\mathrm{T}$$

A further step down:

$$step3to4 :: (t_1 \rhd t_2 \rhd t_3)\ (t_4\ m)\ a \to (t_1 \rhd t_2 \rhd t_3 \rhd t_4)\ m\ a$$
$$step3to4 = \mathbb{Z}_{\mathrm{T}}$$

The pattern should now be obvious. A single step down at any position in the stack is defined as:

$$\downarrow :: t_1\ (t_2\ m)\ a \to (t_1 \rhd t_2)\ m\ a$$
$$\downarrow = \mathbb{Z}_{\mathrm{T}}$$

Stepping back up is similar:

$$\uparrow :: (t_1 \rhd t_2)\ m\ a \to t_1\ (t_2\ m)\ a$$
$$\uparrow = run\mathbb{Z}_{\mathrm{T}}$$

such that $\downarrow \circ \uparrow \equiv id$ and $\uparrow \circ \downarrow \equiv id$ hold.

3.3 Abstraction with the Zipper

How does the monad zipper solve the monad stack abstraction problem, and avoid clashing monad transformer instances? Simple, we shift the focus on a different layer in the stack for each feature. That way the different monad transformer instances do not all have to be at the top of the stack.

A simple application of this idea consists of defining a combinator $\otimes$ that shifts each monadic layer one level to the right.

$$\begin{aligned}(\otimes)\ ::\ & Mixin\ (a \to t_1\ (t_2\ m)\ b) \\ & \to Mixin\ (a \to (t_1 \rhd t_2)\ m\ b) \\ & \to Mixin\ (a \to t_1\ (t_2\ m)\ b) \\ c1 \otimes c2 = {} & \lambda proceed\ x \to \\ & c1\ (\uparrow \circ c2\ (\downarrow \circ proceed))\ x\end{aligned}$$

Here component *c1* focusses on the current layer, and *c2* looks one position down – that's why we have to bring *proceed* down ($\downarrow$) to its level and shift the whole back up ($\uparrow$) to the current level.

This combinator is very useful whenever we have a set of features that uses a disjoint set of monads (that is, each feature will use different monads). No additional work is needed to make the two state transformers of *prof* and *memo* happily coexist.

$$profmemofib :: Int \to \mathbb{S}_{\mathrm{T}}\ Int\ (\mathbb{S}_{\mathrm{T}}\ (Map\ Int\ Int)\ (\mathbb{I}_{\mathrm{T}}\ \mathbb{I}))\ Int$$
$$profmemofib = fix\ (prof \otimes memo \otimes fib2')$$

Note that every component has its own state transformer, notably $\mathbb{I}_{\mathrm{T}}$ for *fib2′*, and we use the base monad $\mathbb{I}$ at the bottom of the stack.

3.4 Effect Encapsulation

To evaluate *profmemofib*, the use has to supply the appropriate run functions:

$$n = run\mathbb{I} \circ run\mathbb{I}_{\mathrm{T}} \circ eval\mathbb{S}_{\mathrm{T}}\ empty \circ run\mathbb{S}_{\mathrm{T}}\ 0\ \$\ profmemofib\ 7$$

We can encapsulate the components more tightly by bundling them with their run functions and hiding the effect types.

$$\begin{aligned}
&\textbf{data}\ Component\ f\ a\ b = \forall\ t_2.MonadT\ t_2 \Rightarrow \\
&\quad Component\ \{\, behavior\ :: \forall\ t_1\ m.(MonadT\ t_1, Monad\ m) \\
&\qquad\qquad\qquad \Rightarrow Mixin\ (a \to (t_1 \rhd t_2)\ m\ b) \\
&\qquad\quad , run\qquad :: \forall\ m\ x.Monad\ m \Rightarrow t_2\ m\ x \to m\ (Run\ f\ x)\,\} \\
&\textbf{type family}\ Run\ f\ x
\end{aligned}$$

In the *Component* type definition, t_2 is an existentially quantified type. That means it is hidden from users of the component. In contrast, m and t_1 are universally quantified. This means that the user of the component gets to choose, and the component must work for all possible choices. In other words, every component is only aware of its own effects.

The *behavior* field contains the mixin, while *run* captures the effect evaluation. The latter should handle any return type x, and is allowed to change that return type according to a type family *Run*. The need for this becomes clear when we look at particular component implementations:

$$\begin{aligned}
&fibC :: Component\ ()\ Int\ Int \\
&fibC = Component\ \{\, behavior = fib2', run = run\mathbb{I}_{\mathrm{T}}\,\} \\
&memoC :: Component\ ()\ Int\ Int \\
&memoC = Component\ \{\, behavior = memo, run = eval\mathbb{S}_{\mathrm{T}}\ empty\,\}
\end{aligned}$$

Both *fibC* and *memoC* leave the result type unmodified. This is captured in the type family instance.

$$\textbf{type instance}\ Run\ ()\ x = x$$

In contrast, the profiling component augments the result type with the profiling information.

$$\begin{aligned}
&profC :: Component\ (RPair\ Int)\ Int\ Int \\
&profC = Component\ \{\, behavior = prof, run = run\mathbb{S}_{\mathrm{T}}\ 0\,\} \\
&\textbf{data}\ RPair\ s \\
&\textbf{type instance}\ Run\ (RPair\ s)\ x = (x, s)
\end{aligned}$$

The operators $\times$ and *fixC* are the component counterparts of the mixin operators $\otimes$ and *fix*.

$$\begin{aligned}
&fixC :: Component\ f\ a\ b \to (a \to Run\ f\ b) \\
&fixC\ (Component\ bhC\ runC) = run\mathbb{I} \circ runC \circ run\mathbb{I}_{\mathrm{T}} \circ run\mathbb{Z}_{\mathrm{T}} \circ fix\ bhC
\end{aligned}$$

Note that *fixC* instantiates the top and bottom of them monad stack with identity effects $\mathbb{I}_{\mathrm{T}}$ and $\mathbb{I}$ respectively.

$$
\begin{aligned}
&(\times) :: \mathit{Component}\ f_1\ a\ b \to \mathit{Component}\ f_2\ a\ b \to \mathit{Component}\ (f_1, f_2)\ a\ b \\
&(\mathit{Component}\ \mathit{bh1}\ \mathit{run1}) \times (\mathit{Component}\ \mathit{bh2}\ \mathit{run2}) = \mathit{Component}\ \{ \\
&\quad \mathit{behavior} = \lambda \mathit{proceed} \to \\
&\qquad \mathbf{let}\ \mathit{proceed}' = \mathbb{Z}_{\mathrm{T}} \circ \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ \mathit{run}\mathbb{Z}_{\mathrm{T}}\ \mathbb{Z}_{\mathrm{T}} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{proceed} \\
&\qquad\quad \mathit{bh2}' \quad = \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{bh2}\ \mathit{proceed}' \\
&\qquad\quad \mathit{bh1}' \quad = \mathbb{Z}_{\mathrm{T}} \circ \mathit{tmixmap}\ \mathbb{Z}_{\mathrm{T}}\ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \circ \mathit{bh1}\ \mathit{bh2}' \\
&\qquad \mathbf{in}\ \mathit{bh1}' \\
&\quad , \mathit{run} \quad = \mathit{run2} \circ \mathit{run1} \circ \mathit{run}\mathbb{Z}_{\mathrm{T}} \} \\
&\mathbf{type\ instance}\ \mathit{Run}\ (f_1, f_2)\ x = \mathit{Run}\ f_2\ (\mathit{Run}\ f_1\ x)
\end{aligned}
$$

The implementation of component composition $\times$ is particularly complicated by attaining the appropriately focused shapes of the same monad stack for the different components and their resulting composition.

Finally, we may compose a number of different functions as follows:

$$
\begin{aligned}
&\mathit{proffib}' :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{proffib}' = \mathit{fixC}\ (\mathit{profC} \times \mathit{fibC}) \\
&\mathit{profmemofib}' :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{profmemofib}' = \mathit{fixC}\ (\mathit{profC} \times \mathit{memoC} \times \mathit{fibC})
\end{aligned}
$$

$$
\begin{aligned}
&> \mathit{proffib}'\ 20 \\
&(6765, 21891) \\
&> \mathit{profmemofib}'\ 20 \\
&(6765, 39)
\end{aligned}
$$

4 Discussion

After this exercise on abstraction, it is time to reflect on some of the design choices and summarize the main ideas.

4.1 Monad Transformers and Data Types à la Carte

As shown in Section 2 composing modular components with effects is not straightforward. The approach described in this paper can be viewed as an improvement of the existing techniques.

The approach of (4) to modular interpreters is an important step towards the goal of modularizing interpreters (and programs in general). However their approach does not support separate compilation nor the concurrent development of several interpreters, because all the features are entangled through hard references.

The Data types à la Carte approach (5) avoids these hard references: it shows how to abstract away from the concrete compositions of datatypes. Unlike Liang et al., Swierstra does not consider the issue of modular implementations of *effectful* features of an interpreter. He does, however, apply his technique to free monads, obtaining a modular way to combine different monads. This provides an alternative to monad transformers, but we expect similar issues to the ones identified in Section 2 to occur for stacks of free monads. Thus, a monad zipper suitably adapted to stacks of free monads would be desirable.

Unlike Liang et al. and Swierstra, we use open recursion instead of type classes. Type classes are very good for the ultimate automation as we do not even have to bother explicitly composing features. However, this approach does not allow multiple implementations for the same feature to coexist, since type classes do not permit more than one instance per (feature) type. At a relatively small cost of explicitly composing features by hand we gain increased flexibility, expressive power and ability to reuse components.

4.2 Other Stacks for The Zipper

Although the Monatron library is used in this paper to present the monad zipper, it is certainly possible to use other monad transformer libraries such as the MTL, which is a library inspired by the original design proposed by (4). We have two main reasons to prefer Monatron over the MTL.

- The first reason is that in the MTL, the class for monad transfomers
  ```
  class MonadTrans t where
    lift :: m a → t m a
  ```
  provides only the *lift* method. However, in order to lift the operations of the various monads through the monad zipper an operation like the *tmixmap* provide by Monatron is necessary. MTL is not fundamentally incompatible; one work-around consists of adding *tmixmap* in a new subclass of *MonadTrans*.
- The second, more fundamental, reason to prefer Monatron over the MTL is that the MTL design prevents certain operations from being lifted. In particular, as noted by (6), the *listen* operation of the writer monad seems to be impossible to lift. This would preclude the use of writer monads, which is not desirable. Nevertheless, if we would be willing to give up the *listen* operation or the writer monad, then it would be possible to use the monad zipper in the MTL.

While we focus here on the range of monad transformers available in Monatron, it should be possible to use the monad zipper in stacks of free monads. It would be also be interesting to adapt the zipper to other effect stacks such as *applicative functors* (7) or *arrows* (8).

4.3 Open Recursion

Cook (2) was one of the first to propose open recursion as a means to extend existing behavior in an OOP inheritance setting. Recently, Oliveira et al. (9)

have done the same to model effectful aspect oriented programming advice using monads to model effects.

The modularity concerns addressed in these works are orthogonal to our own: they are concerned with augmenting the behavior of a feature, rather than adding more features. In fact, they can be readily combined with our work to, for example, add profiling as a separate component that can be (re-)used to advise different features.

5 Conclusion

With this paper we have shown that highly modular *and* effectful systems can be realized in Haskell. Our solution borrows heavily from the literature, in particular Liang's modular interpreters and Swierstra's data types à la carte, but the monad zipper is the key ingredient that ties everything together. The code is available on Hackage, as part of the Monatron library.

In ongoing work we are investigating two applications of our approach:

1. solutions to the expression problem such as Data Types à la Carte (5), and
2. combinators for compositional search heuristics as outlined in the Monadic Constraint Programming framework (10).

At a more abstract level, we would like to build a name-based approach on top of our structural technique, comparable to named variables versus De Bruijn indices.

Acknowledgements

We are grateful to Jeremy Gibbons, heisenbug, Mauro Jaskelioff, Wonchan Lee, Wouter Swierstra, Phil Wadler and Stephanie Weirich for their help and feedback on an earlier version of this text.

Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders. Bruno Oliveira is supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant number R11-2008-007-01002-0.

Bibliography

[1] Jaskelioff, M.: Monatron: An extensible monad transformer library. In: IFL '08: Symposium on Implementation and Application of Functional Languages. (2008)

[2] Cook, W.R.: A Denotational Semantics of Inheritance. PhD thesis, Brown University (1989)

[3] Huet, G.: Functional Pearl: The Zipper. Journal of Functional Programming **7**(5) (September 1997) 549–554

[4] Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL'95. (1995)

[5] Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4) (2008) 423–436

[6] Jaskelioff, M.: Modular monad transformers. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2009) 64–79

[7] Mcbride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1) (2008) 1–13

[8] Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37** (1998) 67–111

[9] Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: Effectiveadvice: Disciplined advice with explicit effects. In: AOSD '10: ACM SIG Proceedings of the 9th International Conference on Aspect-Oriented Software Development. (2010)

[10] Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. J. Funct. Program. **19**(6) (2009) 663–697

Experiences using F# for developing analysis scripts and tools over search engine query log data

Stefan Savev[1] and Peter Bailey[2]

[1] Northeastern University, Boston MA, USA,
savev@ccs.neu.edu
[2] Microsoft Corp., USA,
pbailey@microsoft.com

Abstract. We describe our experience using the programming language F# for analysis of text query logs from the Bing search engine. The goals of the project were to develop a set of scripts for enabling ad-hoc query analysis, clustering and feature extraction as well as to provide a subset of these within a data exploration tool developed for non-programmers. Where appropriate we describe programming patterns for the text analysis domain that we used in our project. Our motivation for using F# was to explore the benefits and weaknesses of F# and functional programming in such an application environment. Our investigations showed that for the target application, common use cases involve the creation of ad-hoc pipelines of data transformations. F#'s language and library-level support for such pipelined data manipulation and lazy sequence evaluation made these aspects of the work both simple and succinct to express. We also found that when operating at the extremes of data scale, the ability of F# to natively interoperate with C# provided noticeable programmer efficiency. At these limits, reusing existing C# applications may be desirable.

1 Introduction

Query log analysis is one of the techniques that can drive improvement of the quality of search engine results. Typically, analyses of query logs require both extensive hardware resources and complex modeling techniques. Our goal in this research is to explore the suitability of the F# programming language [17] for ad-hoc analysis of query log data. We focus mainly on the support of F# for investigative programming and tool building. While the scripts and the application that we developed work on multi-core machines and out-of-main-memory scenarios, efficiency considerations were secondary to ease of use and the ability to quickly evaluate models over the data. To carry out our evaluation of F#'s suitability to the domain of query analysis we implemented a few clustering and feature selection algorithms. We incorporated the techniques that work best on our data in an application with a graphical frontend. The application allows browsing, filtering, grouping of the data and display of aggregated patterns.

The field of large scale query log data analysis is driven by the need to improve search engine quality. The dominant systems approaches in the field are MapReduce [7], SCOPE [4] and DryadLINQ [10] where the focus is on scalability. Due to the high latency caused by the large amounts of data and the sharing of cluster resources between many users, those approaches may not offer quick turnaround times. An alternative for many information retrieval researchers and practitioners is to analyze a sample of the data using interactive tools like Matlab or external commands. Those usually require that the data fits in main memory. The approach that we took with F# offers a middle ground by not requiring the data to fit in main memory while still allowing for quick compositions of primitives in scripts. Thus, the capabilities of F# for lazy evaluation over sequences that may not fit in memory, higher-order functions, succinct syntax and the possibility to define custom abstractions (e.g. pipelines of data transformations) were important for this project. We also took advantage of F# model for parallel computations. Even though we consider F# a good match to data analysis tasks such as ours, reports detailing similar experiences are rare [6]. While some of the algorithms we implemented were a good match to functional programming, we also found that some data mining algorithms that involve mutation of state did not fit the functional programming paradigm. We were still able to express them in F# using imperative programming.

The majority of open source tools in our field are written in Java. Various data processing scripts are typically written in Python or Perl. On the hand, ideas from functional programming underpin the design of MapReduce and DryadLINQ which have proven effective solutions to data analysis problem similar to ours. In our project we reap both the benefits of scripting and high-level programming abstractions that have proven useful in our field using the same programming language. We were able to build both research style scripts and seamlessly integrate them into a tool.

We describe the project in the next section. We discuss related work in Section 3. In Section 4 we detail our experience with F# and show typical use cases.

2 Project Description

Our project consisted of two stages: a) investigation of clustering algorithms applied to query logs (Table 2) and b) implementation of a graphical exploration tool (Figure 2). In the first part we implemented a few standard clustering algorithms with custom modifications. Even though clustering is well-studied topic in the statistics, data-mining and machine learning literature, finding a clustering method that works well on our dataset was a challenging task. The main reason is that most of the queries are very short. This makes it hard to estimate distances between the queries which in turn precludes off-the-shelf clustering tools to work well. Additionally, text clustering algorithms are quite sensitive to initialization and data preprocessing. While the programming language used for implementation does not guarantee success for data analysis, the use of F# did

Discriminative Words By Cluster
type, function, operator, pattern, types, class, cast, match, discriminated, union, interface, operators, active, matching, functions, generic, units, record, string, unit, syntax, patterns, module, int, member, measure, static, return, loop, constructor
list, array, seq, map, sample, sequence, code, monad, fold, math, fibonacci, performance, arrays, computation, recursion, tail, append, matrix, examples, cheat, comprehension, samples, sheet, immutable, monads
download, visual, studio, ctp, powerpack, 2008, 2010, express, pack, mono, power, compiler, install, 2009, split, 2, october, shell, 0, linux, documentation, beta, vs2010, 4, runtime, september, release, emacs, framework
tutorial, f, programming, async, c, language, vs, parallel, interactive, net, asp, wiki, book, tutorials, blog, center, workflows, developer, books, lazy, workflow, specification, msdn, line, command, versus, asynchronous
scale, major, minor, melodic, dorian, chords, natural, harmonic, piano, ukulele, chart, key , descending, clarinet, sax, uukulele, phrygian, ascending, pentatonic, comparison, iv, v, hash, fingering, violin, progression, min, statistical, octaves
...

Table 1. Example of discriminative words extracted after clustering queries containing the word F#

help us in the evaluation of multiple possible algorithms, parameters and data inputs. We consider that F# was a good match for this problem because of its ability to create small and easily modifiable programs. We implemented model-based K-means [21] and LDA [16]. In the second stage we incorporated the most successful clustering algorithms in a query browser application with a graphical user interface (Figure 2). We organized the application around datasets derived through various operations. Given that many machine learning algorithms take a dataset and produce a new one (e.g. by filtering or weighting the data points), the selected abstraction seems natural. It is further reinforced by the F# library design which encourages that data transformations be obtained by chaining a few primitives. Thus F#'s standard library gave us a good design pattern to follow. The operations on datasets always return a new dataset without modifying an existing one in place. This approach is both for convenience, so that the user can return to an existing dataset later, and because the data may not fit in memory. We used three different kinds of datasets: 1) datasets containing queries, 2) datasets containing urls and 3) datasets containing textual features derived from the query or url datasets. We implemented the following groups of operations for query and url datasets: 1) filtering; 2) clustering; 3) selection of top data points (words, queries, urls) by various criteria; 4) grouping; 5) transformations; 6) merging; 7) extraction; 8) comparison;

All operations except extraction, comparison and feature analysis return new datasets as results. Based on insights from executing various operations the user can feed new datasets via queries from observed patterns into the system and proceed iteratively.

We carried out all computations on a high-end desktop using parallel programming techniques for the core operations. Due to the large size of the query log, multiple levels of caching take place before we can import the data in our application. We import the data from an existing distributed application written in C# via a limited number of filtering commands. We store the datasets in binary files on disk.

We demonstrate the usefulness of our tool by an example. The task of this example is to extract from the query log a list of cooking recipes. To gain an intuition about the properties of the data, the user can proceed in the following order: 1) search for queries containing the phrase "cooking recipes"; 2) explore various analysis views of top features and clusters; 3) notice that the pattern "/Recipe/X/" where X is the name of a recipe appears more than expected by chance in the urls; 4) observe that this pattern is associated with websites which have a large number of hits on cooking recipes.

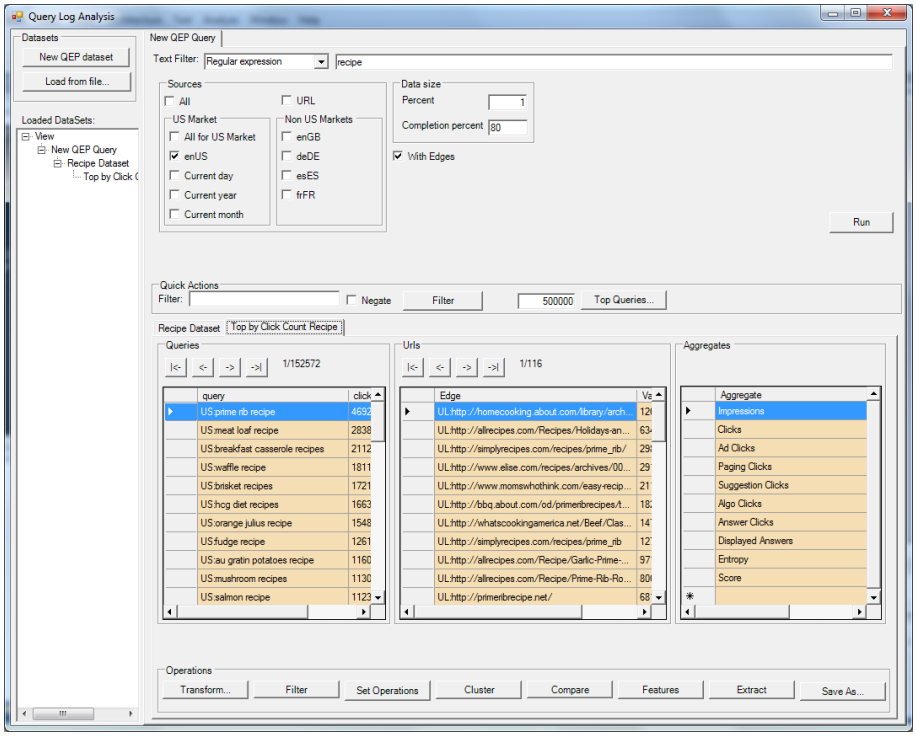


Fig. 1. Our application main screen

3 Related Work

A popular approach for large scale data processing is MapReduce[7]. MapReduce was inspired by the functions map and reduce from the Lisp programming language. MapReduce, however, is somewhat inappropriately named because it combines the functions map and reduce with a third operation: group by. A similar monolithic function was proposed in [19] before MapReduce became popular in order to solve elegantly the problem of histogram creation in functional languages. At the core of the approach of MapReduce is the use of associative operators to split the problem whose intermediate results will be aggregated. Sawzall [13] is a special purpose scripting language that is a generalization of MapReduce. The generalizations in Sawzall are in two ways: a) nested grouping and b) application of multiple associative operations within groups computed by a single pass over the data. In our application we also use associative operators (counts, random samples, filters, groups and within group statistics) and in that respect we are similar to MapReduce and Sawzall. DryadLINQ[10] is a more general approach based on a much larger number of combinators from functional programming than MapReduce. Among those operations are map, filter, group, sort, concatenate. DryadLINQ offers a distributed execution of LINQ queries by dynamic generation of query execution plans. As part of the query execution DryadLINQ also avoids out-of-memory conditions by data partitioning and serialization via reflection. While some of our code in F# uses similar combinators as Dryad we do not compute query plans but manually point intermediate results to disk when necessary. Another large scale data processing system in use is SCOPE [4]. SCOPE is modeled after SQL but allows users to plug arbitrary extractors and reducers written in any .Net language. A SCOPE program is very much like a skeleton or framework used to piece together user-defined C# functions and orchestrate the distributed execution.

Our code for file input and data transformations look similar to the Haskell code in [8]. However, the F# code is simpler because F#'s type system does not track side-effects with monads.

A basic research-style search engine in F# is described in [14]. Their implementation uses the F#'s standard library and a library for external memory algorithms. Some of our code follows a similar style. We also use a combinator-based approach similar to Haskell.Binary for specifying serializers.

F# was used for a different type of data driven research task in the TrueSkill system [6]. TrueSkill models the skill level of players in a Bayesian framework. The data experiments in TrueSkill were carried out in F#.

4 Experiences

Our experience with F# is very positive. It allowed us to immediately focus on the domain problem of exploring algorithms on query log datasets without presenting software engineering challenges. We found it beneficial to always start with the smallest program that could possibly solve the problem and then work,

if necessary, towards making it more robust and efficient. We wanted to explore multiple clustering and feature selection algorithms, as well as multiple parameterizations, data inputs and initializations. F# was useful for this research problem in the following ways: 1) it helped implement well-understood baseline methods quickly and gain intuition about properties of the data, 2) allowed us to easily achieve flexible parameterization via higher-order functions and thus explore multiple inputs to the algorithms. The only bottleneck related to F# that we encountered had to do with sorting tuples of strings. We were able to work around it effectively. Other bottlenecks were related to multiple disk writes or were of algorithmic nature.

```
type Counts = Counts of int*Dictionary<string,int>
type Document = Document of (string*int)[]
let documentToWordsAndTf (Document wordsAndTf) = wordsAndTf

let countsFromStream (strm: (string*int) seq) =
    let d = strm |> Seq.countBy fst |> Dict.fromSeq
    Counts (d |> Dict.getValues |> Seq.sum, d)

let kmeans (numClusters: int) (documents: Document []): int [] =
    let numUniqueWords =
           documents
           |> Seq.collect documentToWordsAndTf
           |> Seq.distinctBy fst
           |> Seq.length
           |> float

    let negLogProb (Document wordsAndTF)  (Counts (total, counts)) =
        wordsAndTF
        |> Seq.sumBy(fun (word,tf) ->
            let wordProb = float ((Dict.getDefault 0 counts word) + 1) / ((float total) + numUniqueWords)
            - (float tf)*log wordProb)

    let docsToAssignments (clusters: (int*Counts) []) =
        let docToCluster(d: Document) = Seq.minBy (snd>>(negLogProb d)) clusters|>fst
        documents |> Array.map docToCluster

    let assignmentsToClusters (assignments: int []) =
        Array.map2 (fun clusterId doc -> (clusterId,doc)) assignments documents
        |> Seq.groupBy fst
        |> Seq.map (snd >> Seq.collect (snd>>documentToWordsAndTf) >> countsFromStream)
        |> Seq.mapi (fun i counts -> (i,counts))
        |> Array.ofSeq

    let rec loop times assignments =
        if times <= 0 then assignments
        else assignments
             |> assignmentsToClusters
             |> docsToAssignments
             |> loop (times - 1)

    let rand = System.Random 1234
    Array.init (Array.length documents) (fun _ -> rand.Next(numClusters))
    |> loop 10
```

Fig. 2. Basic implementation of model-based K-means for document clustering in F# (after [21]). This is an example of a data mining algorithm that can be expressed in functional style with the standard F# library.

```
//create a parser by composing parsing combinators
let lineParser = tuple2 (parseTill "\t")
                        (restOfLine |>>= fun s -> s.Split([|'\x15'|]))
File.readLines @"input.txt"                        // read a file into a stream of lines
|> Seq.parseWith lineParser                        // parse each line using user-defined parser
|> Seq.collect (fun (query,relatedQueries) ->     // create pairs of query,relatedQuery
                    relatedQueries
                    |> Seq.map (fun relatedQuery -> (query,relatedQuery)))
|> Query.objectsToQueryNodes snd                   // use in-house library to expand the query string
                                                   // to a query object with urls
|> Seq.collect
       (fun ((query,relatedQuery), queryNode) ->
             queryNode
             |> Query.getEdges                     // extract urls from the query object
             |> Seq.map (fun (url,count) ->
                      [query; relatedQuery; url; count.ToString()] // format result
                      |> String.concat " "))
|> File.writeLines "outputFile.txt"
```

Fig. 3. An example of a typical query log processing task in F#. Similar scripts are usually used only once. Their purpose is usually to change the format or join datasets.

4.1 Experiences with Algorithm Implementation

We found that the functional programming style fitted well the implementation of model-based K-means via standard combinators like map and group by. We show a simple implementation in figure 4 to illustrate the functional programming style we tend to use. This script gives results comparable to the ones described in the literature for this algorithm [21]. We used this program as a starting point. We extended the program with other smoothing methods and used it for the related bisecting K-means algorithm [15]. We found that on published datasets modifications to K-means initialization improved results. Thus, the ability to easily modify programs is valuable for data processing tasks. Proponents of functional programming have often emphasized that small changes in the specification have to be reflected as small changes in the implementation. We found that this was often the case when we the used functional programming style in F#. We did an equally simple implementation in C# by imperatively updating matrix values and explicitly manipulating indexes. The resulting implementation was around 200 lines vs. 50 in F#. The high-level implementation can be easily made parallel by changing Array.map with Array.Parallel.map in the assignDocsToCluster function.

Unfortunately, we could not use the functional style for the LDA clustering algorithm. The reason is that efficient implementation of this algorithm involves mutation of state. For this algorithm we lose the benefits of functional programming, but still retain some of the benefits of F# including code conciseness resulting from the type inference system.

We found that the kinds of data mining algorithms that benefit from rapid prototyping similar to k-means include grouping, filtering and applications of statistical functions over a data collection.

Run on a sample first	Run on the complete dataset

`input`
`|> sample 5000`
`|> step1`
`|> ...`
`|> output`
	`input`
	`// remove step |> sample 5000`
	`|> step1`
	`|> ...`
	`|> output`

Fig. 4. An example from our practice that applicative style is easily amenable to modifications.

4.2 Experience with Applicative Style

A large amount of the code we wrote used applicative style with F#'s function application operator as in:

```
input |> step1 param1 |> step2 |> ... |> output
```

This code is equivalent to:

```
output(... (step2 (step1 param1 input))))
```

It is a convention to use the former style in F# code. Typically each of the steps does not use global mutable state except I/O. However, the internal implementation of most of the standard library functions and our extensions are imperative for efficiency. They may use local mutable variables as opposed to a more mathematical formulation via recursion. It is usually the case that many of the applied steps are quite simple when considered in isolation. However, when one attempts to construct directly the code which corresponds to a long pipeline, which itself may use nested pipelines, one is forced to code loops, nested loops and mix indices from different conceptual stages together. Such code quickly becomes incomprehensible and unmodifiable. In our use cases the ability to easily modify the code is highly desirable. We found this style is preferable because one can easily add and remove processing stages. A common idiom is when one wants to first run the program on a small sample of the data (figure 4.2). As can be seen in Figure 4.2 the corresponding change is very small. Had the logic been fused into a single loop such a small change would be more difficult.

A different canonical example from Information Retrieval is a document processing pipeline. In F# we use:

```
document |> tokenize |> stopword |> stem
```

Some object-oriented implementations of Information Retrieval Systems simulate this pattern by a special Pipeline interface which only applies to a stream of words. As shown in [14] and [12] this pattern does not only apply to processing words but to the index construction and query matching components of search engines.

4.3 Experience with Streams

Typical query log datasets do not fit in memory. Due to this reason the ability to stream over the data is very important. F# supports a good syntax for generating sequences and a useful library of common operations over sequences. F#'s library implements imperative streams as enumerators. There are multiple possible designs for streams each with different trade-offs. Functional streams are a more flexible alternative to imperative streams. Functional streams themselves could be implemented via recursion [2] or iteration [5]. We experimented with stream implementation via lazy lists as described in [20]. While we could create a good syntax due to F#'s workflow support, experiments revealed that lazy lists were not acceptable for our data loads. The main reason seems to be that a large number of thunks were created. While one could blame the .Net virtual machine for not optimizing such patterns, we found that even an optimizing compiler such as MLton may fail in certain complex cases (even when using the iterative implementation from [5]).

The F# workflow syntax for sequences works well when the iteration resembles a for-loop, but is harder to use in other more complex cases, for example when merging or joining two sorted sequences.

We learned to be careful when using lazy evaluation of streams and external mutable state, because the result depends on the order of evaluation. There is no protection against such problems in F# and in some cases the obtained result might be intended (e.g inserting print statements while debugging).

One should select carefully the tradeoffs between arrays and sequences. In general, one has to be careful not to unintendedly reuse a sequence twice because this might repeat long computations or produce a different result. A common example of the latter is a random number generator. Frequent materialization of sequences using arrays might end up slower than sequences because of memory accesses and cause of out-of-memory exceptions on high data loads.

F# does not have sophisticated code rewrite system for library writers to implement Haskells optimized streams. F# relies instead on the virtual machine to optimize pipelines of sequences. While sequence of pipelines (of maps, filters, etc.) are slower than direct loops, we did not find this to be a bottleneck in practice.

Programs that are expressed as a pipeline composed from various stages can be considered declarative. Therefore, the internal implementation of the combinators matters for efficiency only. The default combinators are in F#'s Seq module, but one could also use LINQ in the same way, or user-defined implementations. We have experimented with lazy functional streams, "push-based" streams, and streams based on message passing. In "push-based" streams the producer pushes items into the pipeline. The Unix command line offers a similar, but more restricted programming model, the major practical difference in our use cases being that one cannot nest other commands within a command. Stream-based interfaces are ubiquitous in functional programming.

4.4 Experience with Scripting

We give an example of a possible use of F# for scripting in our domain. In this example we read a file whose lines are formatted as "headQuery \t relatedQuery1 0x15 relatedQuery2 ...". The goal in this example is to expand each pair of query and related query to a list of corresponding urls and click counts. We use an in-house service to fill in the required data. The final result is a list of (headQuery1,relatedQuery1,url1,count1) tuples written to a file. We use the collect function twice to emit multiple values in the resulting stream. In this example, we use parsing combinators from FParsec [18] as a way to declare the input format. FParsec an implementation of the parsing combinators described in [11] for F#. Our experience is that it is possible to use parsing combinators for many ad-hoc formats which arise in our practice. We found it very useful that simple tasks such as the one described can be expressed both in short and readable code.

Custom code for input/output formatting is typical for many text processing tasks. While FParsec may be quite useful, its use can be avoided if one has control over the formatting specification. For those cases we describe the format using a simple library of serialization combinators, the most common of which are int,string, tuple2, list, etc. for parsing objects of the corresponding types. The type of the record for output can in principle be obtained using reflection, but the input type needs to be specified somehow because of the static typing in F#. We found the approach of building a schema with functions convenient. Here is some typical code:

```
// specification of the format
let schema = string @ (list string) // use of @ operator to denote a tuple of two elements
// output code
someSequence
|> Seq.outputRecords schema "filename"

// input code
Seq.inputRecords schema "filename"
|> processSequence
```

This pattern avoids the need to deal with parsing and output formatting and error prone issues such as string escaping and byte manipulation. It is also very efficient because there is no string parsing. This is a very successful example of the power of functional programming combinators because it gives gains in both programmer and program efficiency. Similar combinators are found in Haskell's Data.Binary library.

4.5 Experience with Parallel Computations

Common cases of parallel computations in our application involve filters, transformations, and groupings. Typically for each group we compute a number of statistics. In many cases the the operations we encountered were associative or could be derived from associative operations. Example of associative operations are sums, counts, random samples, filters. Textbook examples of operations that

can be decomposed into associative operations are the average and the standard deviation. An important characteristic in our use cases and in many other Information Retrieval applications is that the input is usually large and read from a file. In some cases, the generated output is also large. In the field of Information Retrieval the major paradigms for grouping and reductions within each groups are MapReduce [7] and its Sawzall [13] generalization, SCOPE [4] and DryadLINQ [10]. Programs written for those systems require dedicated clusters and specialized software. The simplest strategy we used was to split the input file into chunks with number equal to the number of CPUs, run computations for each chunk and aggregate the results.
We utilized a more complex strategy for computing groups and results within the groups. F#'s Seq module contains a groupBy function. This function, however, materializes intermediate values that fall within each group. If the input is large, usage of this function will cause the program to run out of memory. In some cases, the values within groups may not be required but only some statistics that can be computed in limited memory. We use "push-based" streams to avoid storing lists of values within groups. Instead only a few numbers are stored within each group. To produce multiple results from the same stream we use the split combinator. We can handle nested groups and support a few more combinators in addition to "by", "len", "sum" and "split". Those are collect (for emitting multiple values); map (for transformation of values); distinct (for obtaining distinct values), topBy (for obtaining top values by a criterion); values (for collecting intermediate results). We used on object-oriented implementation which allowed for chaining transformers and consumers. By using our combinators we create a specification which is passed to a "run" function. The run function takes an parallel stream, which can be split into chunks. The run function applies the specification to each chunk to produce intermediate results. When all intermediate results are computed the the run function aggregates them. Essentially, our specifications allow for composable and more general "MapReduce" style programs.

An important point is that even with a push-based stream implementation one can write programs that run out of memory. One option for us would have been to extend our combinators to handle out-of-memory conditions, but we found out that multiple disk writes are detrimental to speed-ups that can be achieved a muticore desktop. Therefore, we took care to avoid materialization of large intermediate results by decomposing a program into multiple programs. The following example shows a case we encountered. In this example, the input data is in the form of a stream of queries, each query pointing to a list of urls and click counts:

```
seq{(query1, [(url11, clickCount11);(url12,clickCount12);...)]);
    (query2, [(url21, clickCount21); ...])}.
```

We would like to a) extract features from each url; b) compute top url features by number of queries generating the feature; c) compute two views for the top features: query view and url view. The features we have in mind are strings like "city=?" and are useful because they allow for extraction of values from a

```
// group by key, compute the number of values and their sum for each group
// input is a sequence of (key,value) tuples

// Solution using standard F# Seq module
input
|> Seq.groupBy fst   //materializes values within groups
|> Seq.map (fun (key,values) ->
               key, (values |> Seq.length
                    ,values |> Seq.sumBy snd))
//Result is seq<'key*(int*int)>

// Solution using our "push-based" stream implementation to avoid
// unnecessary materialization of intermediate results
let spec = by (fst, split2(len(), sumBy snd))
input
|> toParallelStream
|> run spec
//Result is Dictionary<'key, (int*int)>
```

Fig. 5. A comparision between F#'s groupBy and our push-based grouping operator "by". Our implementation avoids storing intermediate results.

category. An alternative feature is the website extracted from the url. Therefore this program is useful for organizing queries by website as well. Instead of writing the program as a single expression we split the program into two phases: 1) extraction of top features; 2) given top features, compute both query and url views simultaneously (using the split combinator). In this way, we read the input data twice but do not write to disk. An alternative that is shorter to write, but slower to execute because of disk writes, would be to compute the results for each feature and its "score" in the same pass and then select best features. The solution to this task is given in figure 4.5.

While we investigated possible solutions to the simultaneous processing of split streams, we observed two styles for processing sequences: "pull-style" corresponding to F# sequences and lazy evaluation, and "push-style" corresponding to message passing and reactive programming with events. "Pull-style" can easily handle merging or joining of sequences, while it fails if a sequence needs to be split. "Push-style" fails on merging two sorted sequences. The duality between both styles has also been observed for event processing in user interfaces [1].

In addition to the combinators described above, we implemented parallel filters, histograms, transposition of a sparse matrix of (query,url) pairs in external memory and random sampling.

4.6 Experience with Tool Building

F# was also very useful for developing the user interface of the application. We used the C# user interface designer but implemented the behaviors for graphical elements from F#. Our application is parameterization rich, i.e. the user can input parameters from multiple graphical elements. To translate the user input to executable code, we mapped input from each selected user-interface element to a higher-order function. We composed all selections to obtain a function that

```
let emitFeatures cont = collect (fun (query,urlAndClicks) ->
                                 urlAndClicks
                                 |> Seq.collect (fun (url,clickCount) ->
                                      url
                                      |> featurize
                                      |> Seq.map (fun feature ->
                                                      feature,(query,url,clickCount))))
                                 cont
let features = parallelStream
               |> run (emitFeatures (by(fst, len())))
let query (query,_,_) = query
let url (_,url,_) = url
let clickCount (_,_,clickCount) = clickCount
let topFeatures = features |> Dict.toSeq |> Seq.topBy 50 (snd>>desc) |> Dict.fromSeq
let spec = emitFeatures //generate features and query-url data
               (filter (fst>>Dict.hasKey topFeatures)  //filter top features
                  (by (fst, (map snd                   //take (query,url,clickCount) data
                               (split2 (by  (query,sumBy clickCount) //compute query view
                                       ,by (url, sumBy clickCount))))))) //compute url view

parallelStream
|> run spec
```

Fig. 6. A realistic example of the use of grouping and stream splitting combinators which operate in parallel on a stream read from disk. This example also illustrates the need to split the program into two programs to avoid materialization of unnecessary results.

is passed as a parameter to an operation over a dataset.
The basic design that we followed when connecting behaviors to the user interface was to use closures as callbacks. The benefit for us was that parameters such as datasets that are used in a user operation are automatically captured. This programming pattern may cause resource leaks since references to datasets are kept in closures. To solve this problem, any callback that we create returns an IDisposable object representing the assigned closure. We gather all objects corresponding to a view such as a tab and assign them to a field in the tab. Thus, when a tab is closed we release the captured resources deterministically.

4.7 Efficiency

Query mining can be quite CPU and I/O intensive depending on the task. Of the CPU intensive operations string processing and especially string sorting are the most notorious. String sorting using the generic .Net sorting functions was unacceptably slow because .Net does not implement a specialized string sorting algorithm as in [3]. An investigation of standard library implementation of sorting in Java, Haskell and Ocaml revealed lack of efficient implementation of string sorting in those libraries as well. Our implementation of string sorting was based on the reference C implementation from [3]. It was easier to translate this implementation to C# than to F#. Thus, C# can be used as a lower-level imperative language when needed. Since both C# and F# share a common representation of types no foreign-function interface is necessary.

This is in contrast to other functional languages such as OCaml or Haskell where C is the foreign-function language. When interfacing with C polymorphism cannot be easily achieved without explicitly passing a dictionary of conversion functions. Boxing and unboxing penalties are usually incurred in those cases. Thus, compared to other functional languages F# has the advantage of running on the same virtual machine as a lower-level language.

We also hit a performance issue by using comparisons of tuples. Those issues were due to F#'s new powerful equality or comparison constructs. Unfortunately, under high loads, when hashing or sorting tuples of objects we found that those constructs did not perform well. In such circumstances, in order to resolve the efficiency issue, it is best practice to switch from unnamed tuples to using named types as hash or sorting keys.

In a few cases we had to modify our code to gain efficiency but lost some readability. For example, instead of using strings we had to remap them to integer ids. To make this common task easier, we implemented a function with the following signature:

```
val withConvertToIds: ((‘a -> int) -> ‘b) -> (‘a Ids*‘b)
where
type ’a Ids =
  class
    member idToObject : int -> ’a
    member objectToId : ’a -> int
  end
```

We can run a computation within this function that remaps the ids and returns the actual result and an object which can perform the reverse map.

Another example of inefficiency is transposing a sparse matrix of queries and urls. Instead of using the obvious algorithm with Seq.collect and Seq.groupBy that works well in main memory, we had to hash manually the strings to integers and group by integer ids to avoid string sorting.

Except the issue caused by tuple comparisions, all of those pitfalls would have occurred independently of the programming language. They are either representation or algorithm dependent. During the early stages of development efficiency is not a requirement but may become later on. Due to this reason the possibility for variable mutation and C# interfacing is a strong point of F# in our use cases.

5 Conclusion

We described our experience using the programming language F# for ad-hoc analysis of query logs. Our usage of F# focused on key text analysis tasks and resulted in a graphical application for browsing logs. We focused on algorithm implementation as well as ad-hoc scripts. In some cases the algorithm could be easily expressed in functional programming style, while in others we had

to resort to imperative programming. The F# language allowed us to easily formulate scripts to carry out typical tasks and we believe offered productivity gains in implementation. In our view F# is a very practical language which proved to be a good match for our usage. There are not many programming languages which can combine the conciseness of popular scripting languages like Python or Ruby with the speed of C# for typical use cases, and at the same time encourage functional programming style. Some of the programming abstractions we used have appeared in various domain specific languages for the analysis of search engine data. Those abstractions have roots in functional programming and their use is encouraging because they bring elegant and useful techniques from functional programming into mainstream practice. We believe we are the first to apply those abstractions to log analysis tasks using a language with heritage from the functional programming community.

6 Acknowledgements

We are thankful to Don Syme for his useful feedback on our paper.

References

1. Reactive extensions for .Net (Rx).
2. Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.
3. Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.
4. Ronnie Chaiken, Bob Jenkins, P. Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
5. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.
6. Pierre Dangauthier, Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill through time: Revisiting the history of chess. In *NIPS*, 2007.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
8. J. R. Heard. Beautiful code, compelling evidence. functional programming for information visualization and visual analytics.
9. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
10. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

11. Daan Leijen and Erik Meijer. Parsec: A practical parser library, 2001.
12. Marc Najork Stephen Robertson Nick Craswell, Dennis Fetterly and Emine Yilmaz. Microsoft research at trec 2009: Web and relevance feedback tracks. In *18th Text Retrieval Conference (TREC)*, 2009.
13. Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.
14. Stefan Savev. A search engine in a few lines.: yes, we can! In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 772–773, New York, NY, USA, 2009. ACM.
15. M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. Technical Report 00-034, University of Minnesota, 2000.
16. M. Steyvers and T. Griffiths. Probabilistic topic models.
17. Don Syme, Adam Granicz, and Antonio Cisternino. Expert F#, Apress, 2008.
18. Stephan Tolksdorf. Fparsec: http://www.quanttec.com/fparsec/, 2009.
19. Philip Wadler. A new array operation. In *Proc. of a workshop on Graph reduction*, pages 328–335, London, UK, 1987. Springer-Verlag.
20. Philip Wadler, Walid Taha, and David Macqueen. How to add laziness to a strict language without even being odd.
21. Shi Zhong and Joydeep Ghosh. Generative model-based document clustering: a comparative study. *Knowl. Inf. Syst.*, 8(3):374–384, 2005.

A Comparison of Lock-based and Lock-free Taskpool Implementations in Haskell

Michael Lesniak

University of Kassel
Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73
Kassel, Germany
mlesniak@uni-kassel.de

Abstract. Today, synchronization of shared data structures in multi-threaded software is mostly implemented using locks, which leads to difficult to understand and error-prone programs. Software Transactional Memory allows lock-free parallel programming by handling the synchronization of shared variables implicitly. We show how to implement different instances of the widely-used Taskpool-pattern (global and private taskpools with and without task stealing) using both lock-based and lock-free synchronization mechanisms. We examine their performance using two synthetic algorithms and LU decomposition and report our observations about parallel performance and the complexity factor of the implementation. Our results show that lock-free taskpools are not only on par with lock-based implementations concerning parallel performance but are also easier to comprehend and develop.

1 Introduction

Today, one of the hardest problem in parallel programming is to keep data that is shared by multiple threads consistent. The classic (manual) approach is to restrict access to the data by wrapping the modification right around a *lock*: Before a thread is allowed to modify data, it has to gain the lock. Until the lock is released, no other thread must modify the shared data (depending on the particular scenario, even reading can be unsafe). This technique has well-known drawbacks that makes the development of correct *and* performant parallel programs even for experts difficult. A modern approach in concurrent software development named *Software Transactional Memory*, coming historically from database research, uses the idea of *transactions* to allow concurrent access to shared data structures without explicit locking and unlocking; instead, functions access shared data structures in transactions and the runtime systems takes care of consistency issues, e.g. by restarting transactions in case of conflicts.

In this paper we analyze the advantages and disadvantages of the lock-based and lock-free synchronization models with respect to performance and ease of development by implementing the well-known Taskpool-pattern in Haskell.

A *Taskpool* is a design pattern to distribute independent tasks to threads: it stores processable tasks in a shared data structure, such that threads can take tasks out of and add new tasks to the pool. Taskpools are good candidates to measure and compare the parallel performance of the two synchronization models: First, the number of tasks and thus accesses to a taskpool are often large, the underlying parallel runtime system and especially the building blocks of the particular synchronization mechanism are therefore constantly under load. Second, the application of taskpools in real-world applications is enormous (for example, Java's Executor Framework[1] is based on this pattern). Results of a comparison have therefore practical relevance both for the implementation of similar parallel programs and parallel runtime systems.

Haskell is a pure and non-strict functional programming language with support for different parallelization approaches (see [2–4]), especially supporting both lock-based and lock-free parallel programming. It is far from obvious how the Taskpool-pattern, typically implemented in imperative languages, maps onto the respective (parallel) Haskell counterparts and which of various implementation possibilities deliver the best performance.

Our contributions in this paper are

- The implementation of three taskpool variants in Haskell. We examine global taskpools and private taskpools with and without task stealing.
- A comparison of the parallel performance of these taskpools with focus on the different synchronization models. For benchmarking we used three example problems: two synthetic algorithms with varying task structures and LU decomposition.
- A list of observations comparing the development effort of both approaches.

To briefly state our results, the development of parallel programs using STM is easier and more comprehensible when compared with standard lock-based approaches. The performance of STM-based taskpool variants is comparable to manual locking and in most cases even better.

This paper is structured as follows: In section 2, we give a brief overview of lock-based and lock-free mechanisms in Haskell. In section 3, we explain the used Taskpool variants. In Section 4 we introduce the problems we chose to benchmark and motivate our choices. In Section 5 we describe the various implementations. In Section 6 we benchmark the parallel performance of the various taskpools and discuss the results. Section 7 compares related work and Section 8 concludes and gives an outlook to possible future work.

2 Lock-based and Lock-free Synchronization In Haskell

Manual locking is implemented with `MVars`. An `MVar` can contain a single value of an arbitrary type; its state is either filled or empty. For locking, we use the property that trying to read from an empty `MVar` blocks the reading thread until a value is stored. The example of Figure 1 shows a variable `sharedVar`,

which is shared between two threads (created by `forkIO`) and exemplifies the usage of `MVars` to implement locking. `threadWait` loops until the shared variable contains the value 1, `threadFill` modifies textttsharedVar directly. Locking and

```
main = do
  sharedVar <- newMVar 0

  forkIO (threadWait sharedVar)
  forkIO (threadFill sharedVar)
  getLine >> return ()

  -- Waits until the shared
  -- value is 1, then adds 1.
threadWait :: MVar Int -> IO ()
threadWait mvar = do
  -- block
  value <- takeMVar mvar
  if value == 1
    then -- unblock and update
         putMVar mvar (value+1)
    else do
        -- unblock and retry
        putMVar mvar value      (*)
        threadWait

threadFill :: MVar Int -> IO ()
threadFill mvar = do
  -- block, update and unblock
  takeMVar mvar                 (+)
  putMVar mvar 1
```

Fig. 1. Lock-based example.

```
main = do
  sharedVar <- atomically $
    newTVar 0
  forkIO (threadWait sharedVar)
  forkIO (threadFill sharedVar)
  getLine >> return ()

threadWait :: TVar Int -> STM ()
threadWait tvar = do
  value <- readTVar tvar
  when (value /= 1) retry
  writeTVar tvar (value+1)

threadFill :: TVar Int -> STM ()
threadFill tvar =
  writeTVar tvar 1
```

Fig. 2. Lock-free example.

unlocking of the shared variable have to be done manually: at `(*)` the lock has to be released by writing to the variable even though the stored value is not modified; the same holds for `(+)`: even though the thread wants to only write a value, it has to take the lock in advance.

Implicit locking is utilized by using Software Transactional Memory (STM) and its concept of transactions. A transaction is a sequence of operations which execution is *atomic* to all threads: if a variable used in different transactions is modified, exactly one transaction is committed and the other transactions are restarted with the updated committed value. To guarantee the ability of restarting transactions at any time, a thread can only modify shared variables inside the `STM` monad, preventing the execution of IO actions. The example can be modified for using STM as shown in Figure 2: Shared variables are stored in

`TVar`s and STM actions are initiated inside the IO monad by using `atomically`. The `retry`-operation is used to restart a transaction in advance, for example when the value of a shared variable is not as expected. By using STM, threads do not have to implement the lock/unlock pattern on shared data structures. Instead, from a thread's point of view, the program flow is sequential.

3 Taskpool variants

While selecting and implementing the taskpools we were guided by the following requirements:

- The taskpool implementation should not imply a large overhead, leaving most of the processing time for the actual calculations.
- The taskpool's interface should be consistent and simple to understand; different variants and their implementations should be easily interchangeable to ease experimentation.
- The functioning of the taskpool should be task independent. Besides simple task structures it should support complex task structures, where subtasks are spawned, dependencies between tasks exists and advanced termination detection is needed.

We study three well-known variants of taskpools:

- With a **global taskpool** all threads access a single homogeneous data structure to take tasks and add new subtasks. A global pool is sufficiently performant when accesses to the pool are infrequently. If threads access the pool often, they can delay each other due to the needed synchronization.
- A **private taskpool** uses a data structure that consists of a public pool as well as a private thread-local storage for each thread. On each access to the public pool a set of tasks is transferred into the private storage. Successive takes are served by this storage until it becomes empty. A private taskpool prevents frequent synchronized accesses to the pool when tasks are small. If large tasks are stored successively, taking a set of tasks of the public pool can leave idle threads workless.
- A **shared taskpools** resembles a private taskpool but has an extended operation to get new tasks: If the public pool is empty, an idle thread accesses the private storage of a still working thread and steals some of its tasks. This approach prevents idle threads but enforces additional synchronization on thread-local storage.

4 Example problems

We choose example problems for using the taskpools based on the following considerations: First, the problems should be processable independently of problem-dependent data structures: A problem where each task must store its result in a

set of synchronized linked lists would be unfavorable, since the additional synchronizations are non-trivial and allow many taskpool-independent optimizations[5]. Second, the tasks should have varying and possibly unpredictable computation time, such that the approach of using taskpools is reasonable. Third, the tasks generated by the problem should not use much (dynamic) memory: since we test the implementations with many thousands and millions of tasks, they would stress the (parallel) garbage collector and thus complicate the interpretation of the results even more.

4.1 Calculating Digits of π

Our most basic and somewhat artificial example task refers to the calculation of π to an arbitrary number of digits. Benchmark problems consist of a list of numbers that each specify the calculation of π to the given precision. This problem has the following properties: First, tasks do not spawn subtasks, hence if the pool is empty, all given tasks have been processed. Second, since only arithmetical operations are used, the calculation is memory efficient. Third, since no subtasks are spawned we can manually assign the tasks for threads beforehand, leaving out the taskpool. Therefore we can calculate the possible maximal speedup and deduce the overhead of the taskpools for these specific cases.

4.2 A Synthetic algorithm

The next example problem is a synthetic algorithm[6] that involves spawning of subtasks, whereby for each task two subtasks are generated:

$$A(i) = \begin{cases} \{10f\} & \text{for } i \leq 0 \\ \{1f\}A(i-2)\{5f\}A(i-1)\{10f\} & \text{for } i > 0 \end{cases}$$

In the base case ($i \leq 0$) one final calculation is done and no subtasks are spawned. In the normal case ($i > 0$) varying sized calculations are interleaved with spawning of two smaller-sized subtasks. The values in curly braces describe simulated computational intensive tasks (we chose the calculation of π as indicated in the last section), $A(i)$ denotes spawning of a new subtask. By varying f, the amount of computation per task can be modified, by varying the initial i the number of tasks and the degree of irregularity.

4.3 LU Decomposition

Solving a system of linear equations described by $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$ is common in many scientific applications. A well-known approach is the Gaussian Elimination Algorithm and its specializations. If the system has to be solved for multiple b, a more efficient approach than solving the system repeatedly uses the *LU-decomposition* of A.

There are efficient approaches to parallelize the LU decomposition and we refer to [7] for an overview and an extended description of our approach. Due to the available space we omit a thorough description and solely describe the task structure. There are two distinct differences in comparison with the previous problems: First, both the number of tasks and the computation time per task becomes smaller with progressing decomposition. Second, the calculation works in iterations. To calculate values for the current iteration, all values from the previous one are needed; hence, additional data dependencies occur and all taskpool implementations need to support iterations and waiting on their processing.

We explicitly want to clarify that our chosen parallelization does not achieve optimal parallel performance; for better performance one would switch to a sequential algorithm when tasks become too small. Nevertheless, we chose this particular example to examine task structures that are difficult to handle for taskpool implementations.

5 Taskpool Implementations

In this section we describe a general taskpool interface for Haskell and lock-based and lock-free implementations for global, private and shared taskpools with a focus on synchronization.

5.1 The Abstract Taskpool Interface

The abstract taskpool interface defines a common interface to all taskpools and consists of two parts: a typeclass defines the necessary functions of a taskpool and a monad defines the environment in which these functions are executed. The typeclass is defined by

```
class Taskpool pool task | pool -> task where
    put  :: task -> TPMonad pool ()
    get  :: TPMonad pool (Maybe task)
    wait :: TPMonad pool ()
```

The `put`-operation adds a task to the taskpool and can both be called by the main thread for initialization and by a working thread. The `get`-operation returns `Just t` if a task `t` is available and `Nothing` if the calculation is finished. If the pool is empty and other tasks are still working, `get` blocks until either all threads are finished or a new task becomes available. The `wait`-operation, only used by the main-thread that created and initially filled the taskpool, blocks until all tasks have been processed. It is used to wait both for the the finish of the final calculation and for fulfilling of task dependencies between iterations (see below).

We used functional dependencies to correctly check the result-type of the `get`-operations: since each pool `Pool a` contained tasks of type `a` and instantiates the taskpool class with `Taskpool (Pool a) a`, the dependency `pool -> task` is sound.

The monad `TPMonad` is used to hide pool-internal data from the interface. It encapsulates a state containing a thread-specific index (which use is explained in Section 5.2) and the used pool and is defined by

```
type TPMonad pool a = StateT (Int,pool) IO a
```

We could have used a Reader-like monad but this approach leaves more flexibility for future extensions.

The following example shows the taskpool independent code for the synthetic algorithm, clarifies the use of the described operations and demonstrates the simplicity of their usage:

```
synthetic :: Taskpool pool Int => Int -> Int -> IO pool -> IO ()
synthetic constFactor initTask poolGenerator = do
  pool <- poolGenerator
  taskpool pool $ do
    forkN numCapabilities (thread (task constFactor))
    put initTask
    wait

task :: Taskpool pool Int => Int -> Int -> TPMonad pool ()
task constFactor t = do
  if t > 0
    then do io $ calcPi (1*constFactor)
            put (t-2)
            io $ calcPi (5*constFactor)
            put (t-1)
            io $ calcPi (10*constFactor)
    else io $ calcPi (10*constfactor)
```

In the example, `numCapabilities` threads are forked, the initial task `initTask` is put in the pool and the main-thread waits until all tasks have been processed. The functions `taskpool`, `forkN` and `thread` are internally defined and used to initialize the taskpool state, fork a number of threads and simplify execution of functions, respectivly. The `io`-function lifts functions with type `IO a` into `TPMonad`.

Our approach supports dependencies between tasks as for example needed in the parallelization of the LU decomposition: an *iteration* consists of initial tasks added to the pool by the main-thread and waiting for these tasks to be processed. In the following example the main-thread waits that all tasks for a particular `i` and all `j` have been processed before adding tasks for the next iteration:

```
forkN ...
forM_ [1..10] $ \i -> do
    forM_ [1..10] $ \j -> do
        put (i,j)
    wait        -- wait for dependencies between iterations
wait            -- wait for final calculation
```

5.2 Lock-based Taskpool variants

We describe the implementation of the global taskpool in detail since it is the basis for the other variants and explain private and shared pools by focusing on the differences and additions.

Global Pool. The implementation of global pools comes with the difficulties of preventing busy-waiting and implementing general termination detection. After a description of the pool's type we will describe the general control flow of the taskpool operations by using schematic flowcharts and explain them in detail.

The global pool is defined by a type

```
data GPool a = GPool {                -- put    get    wait
      gChan   :: Chan a               --  W      R
    , gWork   :: MVar (Set ThreadId)  --  RW     RW     RW
    , gState  :: IORef GState         --         R      W
    , gWait   :: IORef [MVar ()]      --  RW     RW
    , gFinish :: MVar (MVar ())       --         W      R
}
data GState = Put | Wait deriving Eq
```

The comments after each field mark the different access modes for each of the taskpool functions, `R` stands for reading access, `W` for writing and `RW` for both.

The channel `gChan` stores tasks of an arbitrary type `a` and is the only part of the pool responsible for task storage; the other parts are used for synchronization and termination detection. A channel is used solely for its FIFO-like interface; the possibility of thread safe access was not needed since we used our own locking.

The pool data structure is concurrently accessed by multiple threads and access is synchronized by the lock `gWork`. Besides being the global lock we use `gWork` to store the set of `ThreadId`s to check the number of working threads. We have chosen to specify thread identifiers instead of counting working threads because it allows a more thorough overview of the taskpool's functionality; since the number of threads is small and the set-operations insert and delete are $O(\log n)$, usage does not induce a measurable performance penalty.

The taskpool can be in one of two states, `Put` or `Wait`. Initially, before `wait` in the main-thread is called, it is in `Put`-state: initial tasks can be inserted into the pool by the main-thread, idle worker threads need to wait if they can not get tasks. After the pool changes into `Wait`-state, all initial tasks for the current iteration have been inserted. In `Wait`-state, when the pool is empty and all threads are idle, the next iteration can start.

In general, there are two possibilities to block a thread that is waiting on an event: First, the thread can create an empty `MVar` and block until it is filled (*blocked waiting*). Second, the thread can repeatedly acquire a lock, check if the event occurred, release the lock, wait some time and retry again (*busy waiting*). By using busy waiting, the event source does not need to be aware of all listeners, but the lock-check-unlock cycle creates unnecessary locking operations. Since performance was important, we used the first approach which is implemented as follows: the thread that holds the global lock and wants to wait creates a new empty `MVar` `w`. Depending on the event it wants to be informed of, it either adds `w` to `gWait` to be informed on the arrival of new tasks or sets `gFinish` to be informed on the start of a new iteration (see `get`-operation below). Afterwards it releases the global lock and tries to read `w`. The thread is later unblocked when a value is written to its `MVar`.

The *put*-operation works by writing a new task into the channel and unblocking a waiting thread (in `gWait`), if one exists (Figure 3, put).

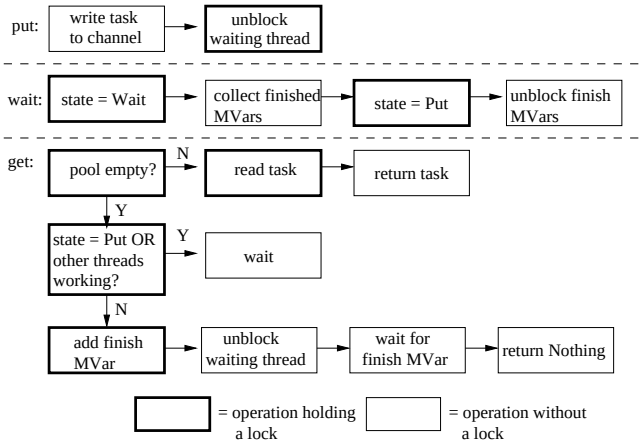


Fig. 3. Schematic flowchart of the taskpool-operations for a global taskpool

When the *wait*-operation is called the pool's state is changed to `Wait`. An empty `MVar` is collected for each finished and idle thread using `gFinish`. Afterwards, the pool's state is changed to `Put` to recreate the initial state for a possible next iteration and threads are unblocked (Figure 3, wait).

The *get*-operation is called by each thread to receive a new task or to be informed that the current iteration has been finished. Since termination detection is implemented in get, it is the most complex of the three operations (Figure 3, get): After the pool structure is locked, the availability of unprocessed tasks is checked. If tasks are available, one is read and returned. If the pool is empty but the pool's state still in `Put`-state or other tasks are still working and could add tasks, the thread starts waiting. If on the other hand all threads have finished their work, the remaining one is the last non-waiting thread; after creating a `MVar` for `gFinish`, it unblocks one of the other waiting threads until all are unblocked.

Private Pool. A private taskpool is similar to a global pool but has in addition to the pool data structure a thread-local (non-synchronized) storage for tasks. The implementation was complicated because all details of the local storage should be hidden from the forked function. We have chosen to use an additional field with type `IOArray Int a`; the type `GPool` is extended as follows:

```
data PPool a = PPool {
    -- ... same as GPool
    , pPrivate :: IOArray Int (ArrayList a)
    , pSize    :: Int
```

```
}
```

`ArrayList` is a custom data structure in the IO monad based on `IOArrays` which supports additional functions, e.g. getting blocks of elements. We generate a thread-specific index in `forkN` and modify the state for each forked thread. The `get` function can therefore access the thread-local `ArrayList` by reading the value at its specific index of `pPrivate`. The access to the local data of other threads is not forbidden but solely enforced by the discipline of the programmer. Nonetheless, this approach has the advantages of being performant, easy to implement and comprehend. A preliminary version of our implementation used the `String`-representation of `myThreadId` to generate an appropriate index by `reading` its value (and thus omitting the need to carry a thread-specific index), but was slow and consumed memory that needed to be garbage collected. Another solution for thread-local storage uses an additional map `MVar (Map ThreadId a)` to store arbitrary values. It has the advantage of not needing any additional information besides the `ThreadId`. But due to the synchronizing needed, threads would need to lock against all others when accessing their private storage!

Shared Pool. A shared pool uses the same approach as the private pool but allows threads to access the local storage of still working threads to steal tasks; other implementations divide the shared pool in a thread-local and private and one public-pool (cf. [8]) but since we were mainly interested in the synchronization behavior we implemented sharing as follows: The thread-local `ArrayList` needs to be extended to be concurrency-safe; the `SPool`-type has the additional attributes

```
data SPool a = SPool {
    -- ... same as PPool
    -- not needed anymore:
    -- pPrivate :: IOArray Int (ArrayList a)
    , sPrivate :: IOArray Int (ArrayListLock a)
    , sSize    :: Int
}
```

The `ArrayListLock` is the lock-based equivalent of a non-synchronized `ArrayList`, where each of the access functions uses a lock to prevent concurrent access.

5.3 Lock-free Taskpool Variants

In the following sections we describe our implementation of the lock-free taskpool variants, where we used STM instead of manual locking. After an explanation of the global lock-free taskpool we describe briefly the differences of the other variants to the lock-based approaches.

Global Pool. The lock-free global taskpool uses the same ideas and techniques as the lock-based one, but we need fewer fields:

```
data STMGPool a = STMGPool {              -- put    get    wait
      stmChan     :: TChan a              --  W      R      R
    , stmState    :: TVar STMState        --         R      W
    , stmFinished :: TChan (TMVar ())     --         W      R
    , stmWorking  :: TVar (Set ThreadId)  --         RW     R
}
data STMState = SPut | SWait deriving Eq
```

The fields follow the same naming scheme and serve the same purpose as in the lock-based example, the main differences are

- the use of STM-supported types (`TChan`, `TVar`, `TMVar`), substituting `Chan` and `MVar`.
- no explicit locking. We leave the details of keeping the pool structure consistent to the runtime system.
- no explicit blocked waiting. Instead of having an explicit list of waiting threads, a thread calls `retry` if it wants to be informed of events and automatically blocks until then.

This advantages directly map onto the implementation complexity of the task-pool-specific operations: the put-operation does not need to inform waiting threads about new tasks and the get-operation does not need to keep track of manual notifications for waiting; all operations ignore the different lock- and unlock occurrences.

Private and Shared Pools. From a synchronization point of view the implementation of private and shared pools is identical. Both pools use a structure `ArrayListSTM` to store thread-local data; instead of an `IOArray`, a `TArray` is used internally. Since the synchronization is done implicitly, we did not need to develop both a locking and non-locking version for the shared and private pool, respectivly.

5.4 Observations and Suggestions

While developing the different variants we made some observations and have some suggestions for future research and development:

STM lets the developer forget about locking, which makes the implementation and usage of shared data structures more easy. Since the STM implementation provides different types that mirror the types provided by the lock-based interface, it allows an easy transition for a programmer which knows the traditional system. Nevertheless, the developer has to understand the concepts of transactions and the timing of commits to write performant code. Using STM for parallel programming makes the development of parallel programs easier but does not (yet) compare to sequential programming.

Unfortunately, debugging of problems and unexpected behavior, occurring frequently in parallel programming, does not become easier. While the lock-based programming model works in the IO monad and lets the developer insert output

and logging statements on the fly, more preparation is needed to understand what actually happens while using STM. We used a global `TChan` and an additional thread to log messages, as described in [9].

The support for profiling performance problems that go beyond space leaks and heavy use of the garbage collector, for which the heap profiler and threadscope exists, is quasi non-existing. Manual timing of critical sections with systematic guessing worked sufficiently well but was quite time-consuming for the lock-based method. Not being in the IO monad (and thus making timing difficult) in combination with transactions being implicitly restarted made reasoning about and profiling of performance problems in the STM monad much more difficult and was unsatisfying.

Summarizing our experiences, developing in the STM monad allows to transform an idea much easier into a working and comprehensible parallel program. Common problems of lock-based programming, e.g. deadlocks, can not occur; nevertheless, it currently has no advantage when it comes to finding bugs or performance bottlenecks.

6 Benchmarks

We tested the different taskpool implementations against each problem on a 2.2 GHz 8-core AMD Opteron 875, Linux-kernel 2.6.18 with GHC 6.12.1. We ran each benchmark, consisting of a particular problem, a taskpool variant and its synchronization model, five times from one to eight cores and used the mean value for speedup calculation. The local storage had space for 256 tasks. We chose our instance sizes such that the absolute runtime on eight cores was around thirty seconds. Since the observations and their interpretation are similar, we explain the benchmark for the π-calculation in detail and focus on the differences for the other ones.

6.1 Pi Calculation

We benchmarked the pi calculation problem using two task scenarios: first, using 131272 tasks with a random task size of 100 ± 10 (*pi-small*, ca. 0.001s/task), second with 8192 tasks with a random size of 1000 ± 100 (*pi-large*, ca. 0.12s/task). Since the tasks in pi-small are so short-lived, the pool is accessed frequently to obtain new tasks. This effect is weakened in pi-large. To compare the taskpool's overhead, we split the tasks manually to threads beforehand.

Figure 4 show the speedup for pi-small and pi-large. In the following we describe our observations and give possible explanations:
a) For short-lived tasks the speedup is better when a lock-based variant is used: Since accesses occur extremely often (tasks are processed fast), the impact of the overhead of the STM implementation is huge.
b) The lock-based shared pool is better than manual distribution for small tasks: Since tasks are randomly generated, threads become idle when they have processed their individual chunks. This is prevented when threads are allowed to

Speedup for pi-small								
Variant	Lock-based				Lock-Free			
	1	2	4	8	1	2	4	8
manual	1.04	1.92	3.61	6.10				
global	0.99	1.89	3.52	5.23	0.89	1.71	3.39	5.72
private	1.03	1.86	3.52	5.99	0.89	1.75	3.26	5.47
shared	1.07	1.92	3.55	6.32	0.89	1.69	3.27	5.47

Speedup for pi-large								
Variant	Lock-based				Lock-Free			
	1	2	4	8	1	2	4	8
manual	1.00	1.93	3.73	6.48				
global	1.00	1.89	3.48	6.22	0.91	1.81	3.63	6.88
private	1.01	1.90	3.56	6.32	0.91	1.82	3.60	6.75
shared	1.01	1.92	3.62	6.42	0.91	1.80	3.63	6.84

Fig. 4. Speedup for the pi-problem

steal tasks.
c) The global lock-free pool performs better than the lock-based one: The speedup for both variants show that the lock-free taskpools scales better with many threads. We believe that waiting threads are restarted earlier than in the lock-based implementation where they always wait until the lock is released.
d) For large tasks all lock-free variants perform better than their lock-based counterparts: the pool is not accessed as often, hence the impact of the lock-free overhead is not as severe. At the same time, the STM variants still scale better (see c)), thus delivering the better speedup.

6.2 Synthetic Algorithm

We benchmarked the synthetic algorithm using two problem instances: first, with very small tasks (default size of 1) and a depth of 24 (196418 tasks) (*syn-small*), second with larger tasks (default size 100) and a depth of 15 (2584 tasks). The results are shown in Figure 5. In addition to the mostly similar results of a) to d) we make the following observations:
e) The performance of the private and shared lock-free variants for small tasks is worse. We believe that the the impact of the overhead for (synchronized) local storage is even higher, since the tasks are smaller than in pi-small.
f) The private variants are slower than the respective global pools: Since tasks are stored thread-local and task stealing is not enabled, large tasks remain unreachable for idle threads.

6.3 LU Decomposition

For benchmarking the LU problem we measured the decomposition time for a randomly generated 5000×5000 matrix. As we've mentioned before, our chosen

Speedup for syn-small								
Variant	Lock-based				Lock-Free			
	1	2	4	8	1	2	4	8
global	0.97	1.87	3.30	5.34	0.89	1.75	3.31	5.88
private	0.96	1.85	3.50	6.13	0.90	1.71	3.02	3.76
shared	0.97	1.87	3.57	6.21	0.88	1.72	3.04	3.75

Speedup for syn-large								
Variant	Lock-based				Lock-Free			
	1	2	4	8	1	2	4	8
global	1.00	1.92	3.73	6.47	0.90	1.79	3.59	6.75
private	1.00	1.89	3.62	5.98	0.90	1.77	3.42	6.55
shared	1.00	1.92	3.74	6.53	0.90	1.79	3.57	6.77

Fig. 5. Speedup for the synthetic algorithm

parallelization strategy is first and foremost a stress test for the taskpools due to the task structure. Figure 6 shows the speedup. We made the following observations:

Variant	Lock-based					Lock-Free				
	1	2	4	7	8	1	2	4	7	8
global	0.64	1.14	1.98	2.41	0.81	0.37	0.69	1.18	1.45	1.47
private	0.64	1.14	1.94	2.22	1.04	0.36	0.67	1.05	1.18	1.26
shared	0.63	1.18	2.11	2.77	1.33	0.37	0.67	1.10	1.36	1.46

Fig. 6. Speedup for LU-decomposition

g) The drop in speedup for 8 cores for the lock-based variant is a well-know problem with the GHC parallel runtime system. Currently we can not explain why this does not happen with the lock-free variants and instead they even gain performance.
h) All implementations have bad speedup. The sequential implementation was a straightforward nested loop implementation of solely math operations that perform naturally fast. We believe that the overhead of the taskpools in conjunction with the difficult task structure leads to these results and that the better performance of the lock-based implementations are due to the better handling of short-lived tasks (see a)) and utilization of thread-local storage.

6.4 Summary

The benchmarks have shown that for typical scenarios of task pools lock-free programming is performance-wise a viable alternative to the traditional lock-based approach. The gap between lock-based and lock-free implementations for

corner cases can be closed with the implementation of native thread-local storage (either through a library or compiler extension) or performance improvements for `TArray`s.

7 Related work

The de-facto standard Haskell compiler Glasgow Haskell compiler supports two other parallelization models that are feasible for simple task structures, e.g. when tasks do not spawn subtasks, involve additional operations in the IO monad or need access to shared data structures: By using *data parallel Haskell* vectorized operations can be executed very fast; approaches to port this model to GPUs exist[10][11]. By using *semi-implicit* parallelism, it is possible to leave all details of parallelization besides annotating potential sources to the compiler; an analysis of different problems solely for semi-implicit parallelism can be found in [12]; the examined problems from the nofib-benchmark suite have structural similar irregular tasks, i.e. calculation of mandelbrot sets, matrix multiplication and raytracing.

An analysis similar to ours for concurrent linked-list implementations, but using different implementations (STM, `MVars` and `IORefs`), can be found in [5]. The authors came to similar results concerning the scalability of STM, even though the different variants perform quite differently, hence a future comparison of these variants in the context of taskpools would be interesting. The idea of comparing different taskpool variants and irregular tasks with synthetic problems has been examined for Java in [6] and OpenMP in [8] and came to similar results concerning the differences between global, private and shared taskpools.

8 Conclusion and Future Work

We have shown how to implement different variants of the Taskpool-pattern (global and private pools with and without task stealing) in Haskell using both manual lock-based and automatic lock-free approaches. The lock-free taskpools were implemented using Software Transactional Memory (STM). We benchmarked each taskpool using two synthetic problems and the LU decomposition of a matrix. We made the following observations:

- The implementation of parallel programs using STM is easier and less error-prone than their lock-based counterparts, since the developer does not have to keep synchronization issues in mind.
- The performance of lock-based and lock-free taskpools is comparable, and in particular scenarios, even better. Our benchmark results suggest that STM-based implementations scale better with a large number of cores.
- The reasons for worse performance were not in the concept of STM per se, but lay either in implementation details, for example, inefficient arrays or very unusual scenarios, i.e. with task durations around 10^{-3} seconds.

– Finding performance bottlenecks is difficult in both synchronization models, since there exists no advanced profilers or other development tools for either implementation.

Our results suggest that the implementation of STM in GHC is mature enough to use it to develop parallel programs that are easier to comprehend and perform comparable to or even outperform their lock-based variants.

We see more topics for future research on lock-based and lock-free taskpools in Haskell: first, it remains interesting to examine if the lock-free variants continue to scale better with more cores. Second, by implementing more performant alternatives for thread-local storage for STM, the advanced taskpool variants should deliver better parallel performance. Third, since taskpool are so widely used and our results are equal to those of imperative languages, developing and analyzing advanced taskpool variants in Haskell is a viable alternative to implementations in imperative languages.

References

1. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional (2005)
2. Jones, S.P., Singh, S.: A tutorial on parallel and concurrent programming in haskell. In: Lecture Notes in Computer Science, Springer Verlag (2008)
3. Jones, S.P., Gordon, A., Finne, S.: Concurrent haskell, ACM Press (1996) 295–308
4. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2005) 48–60
5. Sulzmann, M., Lam, E.S., Marlow, S.: Comparing the performance of concurrent linked-list implementations in haskell. In: DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming, New York, NY, USA, ACM (2008) 37–46
6. Korch, M., Rauber, T.: A comparison of task pools for dynamic load balancing of irregular algorithms. Concurrency and Computation: Practice and Experience **16** (January 2004) 1–47
7. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to parallel computing: design and analysis of algorithms. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1994)
8. Wirz, A., Süß, M., Leopold, C.: A comparison of task pool variants in openmp and a proposal for a solution to the busy waiting problem. Int. Workshop on OpenMP (2006)
9. Marlow, S.: Developing a high-performance web server in concurrent haskell. Journal of Functional Programming **12**(4+5) (July 2002) 359–374
10. Jones, S.P., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the multicores: Nested data parallelism in haskell (2008)
11. Peyton Jones, S.: Harnessing the multicores: Nested data parallelism in haskell. In: APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2008) 138–138
12. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore haskell. SIGPLAN Not. **44**(9) (2009) 65–78

A High Level Implementation of STM Haskell using the Trasactional Locking 2 Algorithm

André Rauber Du Bois

Dept Informática,
Universidade Federal de Pelotas,
Pelotas - RS, Brazil
{dubois}@ufpel.edu.tche.br

1 Extended Abstract

Transactional Memory is a concurrency control abstraction that is considered a promissing approach to facilitate software development for multi-core processors. In this model, sequences of operations that modify memory are grouped into atomic actions. The run-time support for transactions must guarantee that these actions will appear to have been executed atomically to the rest of the system. STM Haskell [2] is a Haskell extension that provides *composable memory transactions*. The programmer defines *transactional actions* that are composable i.e., they can be combined to generate new transactions, and are first-class values. Haskell's type system forces threads to access shared variables only inside transactions. As transactions can not be executed outside a call to `atomic`, properties like *atomicity* (the effects of a transaction must be visible to all threads all at once) and *isolation* (during the execution of a transaction, it can not be affected by other transactions) are always manteined.

This paper describes an implementation of STM Haskell in Haskell, using the Transactional Locking 2 (LT2) algorithm by Dice, Shalev and Shavit [1]. The main reason to choose this algorithm is that, unlike most other lock-based STMs, it safely avoids periods of unsafe execution [1], meaning that transactions are guaranteed to always operate on consistent memory states. For example, in the current implementation of STM Haskell, threads can see an inconsistent view of memory that might lead to non-termination. The solution to the problem was to modify the scheduler of the Haskell virtual machine, so that every time it is about to switch to a thread that is executing a transaction, it should validate its state to check if the transaction is not already doomed.

The contributions of this paper are as follows:

- A High-level implementation of STM Haskell in Haskell is described. Such a high-level implementation of transactions works as an executable specification, that is more flexible and customizable then the current implementation in C. Since STM Haskell was first presented, some extensions to the basic primitives were proposed e.g., [3, 4]. The implementation presented in this paper could work as a testbed to experiment with new extensions for STM Haskell.

– We describe the TL2 algorithm as a state passing monad. We demonstrate that the basic infrastructure needed to implement the TL2 algorithm is enough to support high-level transactional constructs such as `retry` and `orElse`.

Acknowledgment The work presented here was supported by a CNPq/FAPERGS grant.

References

1. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.
2. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*. ACM Press, 2005.
3. N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero. Unreadtvar: Extending Haskell software transactional memory for performance. In *Trends in Functional Programming*, volume 8. Intellect Books, 2008.
4. M. Sulzmann, E. S. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in haskell. *SIGPLAN Not.*, 44(5):11–20, 2009.

Twilight in Haskell

Software Transactional Memory with Safe I/O and Typed Conflict Management

Annette Bieniusa[1], Arie Middelkoop[2], and Peter Thiemann[1]

[1] University of Freiburg, Germany
[2] Universiteit Utrecht, The Netherlands

Abstract. Software transactional memory (STM) is a promising paradigm for the development of concurrent software. It coordinates the potentially conflicting effects of concurrent threads on shared memory by running their critical regions isolated from each other in transactions. However, this automatic coordination is sometimes too restrictive: library calls such as I/O operations are disallowed in a transaction and some transactions fail for minor conflicts that could easily be resolved.
Twilight STM is an extension of the STM approach which addresses these restrictions. It separates a transaction into a *functional* transactional phase, and an imperative *irrevocable* phase, which supports a safe embedding of I/O operations as well as a repair facility to resolve minor conflicts.
This paper introduces our Haskell implementation of Twilight STM and its formalization as an extension of a call-by-value lambda calculus. We use Haskell's type system to restrict operations to the phases of a transaction were they are safe to use. We also introduce a new, type-safe tagging facility to localize conflicts easily.

1 Introduction

Software Transactional Memory (STM) is a promising paradigm for the development of concurrent software. It provides fine-grained deadlock-free mutual exclusion in a scalable and composable way. The underlying concept is simple: computations on shared memory are grouped into blocks that are executed atomically and in isolation on a consistent memory snapshot by transactions. Conflicting accesses to shared memory are detected at run time and resolved transparently. The prevailing transaction continues and commits its write operations while the other transactions are aborted and restarted (rolled back).

The advantages of STM come at a price. Due to the transparent rollback mechanism, non-reversible operations do not mingle well with transactions. Most implementations rely on programming conventions to keep non-reversible (I/O) operations out of transactions. In contrast, the Haskell STM provided with GHC [4], prohibits the use of I/O operations through the use of the type system. Using the STM monad to encapsulate all read and write operations, the outcome of a transaction is a pure function of the values read. Yet, GHC offers the unsafe

lifting of I/O operations into the STM monad via $unsafeIOToSTM :: IO\ a \rightarrow STM\ a$, which is considered a highly dangerous loophole that breaks the functional guarantee. However, the need of this functionality has been demonstrated by several extensions of STM with irreversible and irrevocable transactions [10, 8]. As further elaborated in Section 2, each of these proposals severely constrains concurrency by serializing the execution of irreversible transactions.

Another problem of STM is contention management. High contention on variables can lead to poor performance and scalability if a transaction is repeatedly restarted because of conflicts. While the problem with contention is shared with all concurrency paradigms, the transparent nature of TM hinders the detection of high contention variables and often it is impossible to come up with an application-specific solution using the means offered inside a transaction.

This paper presents Twilight STM in Haskell, an STM implementation which safely augments the STM monad with irreversible actions and allows introspection and modification of a transaction's state to improve contention management (for example). Twilight splits the code of a transaction into a (functional) atomic phase, which behaves as in GHC's implementation, and an (imperative) twilight phase. Code in the twilight phase executes before the decision about a transaction's fate (restart or commit) is made and can affect its outcome based on the actual state of the execution environment. To this end, the Twilight API has operations to detect and repair read inconsistencies as well as operations to overwrite previously written variables. A transaction can only finish successfully if the twilight code resolved all inconsistencies. Otherwise it restarts the transaction.

Twilight also permits the safe embedding of I/O operations with the guarantee that each I/O operation is executed only once. In contrast to other implementations of irrevocable transactions, twilight code may run concurrently with other transactions including their twilight code in a safe way. However, the programmer is obliged to prevent deadlocks and race conditions when integrating I/O operations that perform locking schemes.

Contributions. The unique feature of the Twilight STM is the specification of the code for a transaction in two phases, atomic and twilight. The latter permits code that may detect and repair inconsistencies and thus make a transaction commit. It further provides facilities to safely integrate irreversible operations, the side-effects of which are immediately visible, into a transaction's work flow.

1. We describe our design of the Twilight STM API for Haskell.
   - The twilight code of a transaction can execute concurrently with any other kind of thread: outside of transactions, inside of transactions, or the twilight code of another transaction.
   - The Twilight API prevents the misapplication of features by assigning suitable types to the twilight operations.
   - Twilight code can exploit the full power of the IO monad. That is, I/O, system calls, and standard concurrency control mechanisms are available. Twilight guarantees concurrent, one-shot execution for these operations in twilight code.

   - After a commit preparation, the twilight code can access and modify the state of its pending transaction. It can turn a failing transaction into a successful one by resolving inconsistencies detected during a commit preparation.
2. We formalize the static and dynamic semantics of Twilight STM.
3. We prove serializability for Twilight transactions including repair actions.
4. We provide a Haskell implementation of the Twilight STM. The code is available on the web[3].

The Twilight STM algorithm has been implemented in imperative programming languages. We have developed prototype implementations for Java and C. On the C implementation, we have run benchmarks and compared the results with a state-of-the-art TL2 implementation [2]. We were able to show that twilight code does not introduce extra overhead on top of TL2. On the contrary, the repair mechanism can reduce the abort rates significantly and so improve overall run-time performance. These results can be found in a technical report [1].

Outline. We show a simple example that motivates the need for Twilight STM in Section 3. In Section 4, we look at the API in more detail. We then reveal some aspects of the implementation in Section 5. Finally, we formalize Twilight STM as the language Λ_{Twi} in Section 6, and show that reduction traces of Λ_{Twi} programs can be serialized.

2 Related Work

Transactional Memory (TM) has a huge design space which is investigated by numerous researchers. In addition, research on transactions in data bases as well as concepts used in operating systems provide a fertile ground from which many TM ideas have re-emerged.

STM and Irrevocability. Welc et al. [10] propose an irrevocability mechanism to support flexible contention management and the execution of non-reversible actions (I/O, system calls, precompiled libraries, ...) within transactions. To ensure safety, they use a protocol with single-owner read locks. A transaction becomes irrevocable by executing a special statement that tries to acquire locks on the read set so far and all upcoming reads. To avoid deadlocks, this approach enforces that only a single irrevocable transaction can run at a time. The system is implemented as an extension of the Java-based McRT-STM and uses dynamic method dispatch to enforce the correct usage and interaction with other language constructs.

Similarly, Spear and coworkers [8] compare five mechanisms which all require that at most one irrevocable transaction runs at a time and that these transactions do not abort (they use the term inevitability instead of irrevocability).

[3] `http://proglang.informatik.uni-freiburg.de/projects/twilight/`

Twilight STM introduces a notion of irrevocability, too. I/O operations can also be safely integrated into transactions, but this requires the programmer to first compensate for possible inconsistencies. The main difference to the afore mentioned systems is that Twilight permits arbitrarily many transactions with possibly irrevocable actions to run concurrently.

Welc and coworkers [9] propose a dual-mode implementation of monitors for Java which switches at run time between a lock-based implementation and a transactional one. The switch is based on the level of contention where high contention triggers the use of the transactional implementation. To integrate irrevocable actions, they rely on the same mechanism as in the previously described paper [10]. Here, mode switching is handled transparently to the programmer. Twilight does not switch modes but offers integration with lock-based code in the twilight code. It further reduces the conflict potential of transactions by allowing to inspect and repair read conflicts.

Harris [3] proposes a mechanism to integrate exceptions and side effects with transactions. The proposal relies on a transaction being able to register external actions that execute at commit-time of the transaction.

Harris and Stipić [5] implement the concept of an abstract nested transaction (ANT) in STM. Failure of an ANT does not cause failure of the enclosing transaction. Instead the ANTs are retried when the enclosing transaction is ready to commit. Side effects like I/O and system calls are disallowed inside ANTs. ANTs can be implemented with the Twilight API by performing potential re-execution of computations in the twilight code. This approach is also implemented in Haskell.

STM in Haskell Harris and coworkers [4] report an implementation of STM in Haskell. They introduce the STM monad, which is a special type of computation inside an atomic section. This design enables a clean separation between unrestricted computations outside the STM monad and which restricts operations in an atomic section to reads and writes to transactional variables. There is a special *orElse* operation that enables the specification of alternatives in case of failing transactions. But these alternatives cannot preform repairs as in our approach because all alternatives must be consistent with respect to the start of the original transaction.

3 Counting Committing Transactions

Applications can benefit from twilight operations in a diversity of ways. In this section, we give a concise example that features transactional variables with high contention, avoids rollback via recalculation, and makes use of safe I/O actions in an atomic block.

Figure 1 shows the code for a worker thread which executes concurrently with other threads on shared memory using STM for synchronization and information interchange. For debugging purposes or for measuring the application's progress, a programmer wants to trace the order in which transactions commit successfully.

```
worker :: TVar Int → Int → IO ()
worker counter tid =
  atomically $ do
    r0 ← newTag
      -- expensive calculation on variables tagged with r0
    r1 ← newTag
    (pos, rc) ← readTVar counter r1
    wc ← writeTVar counter (pos + 1)
    twilight
    test1 ← isInconsistent r1
    test0 ← isInconsistent r0
    pos ← if (test1 ∧ (¬ test0))
      then do
        reload
        pos ← rereadTVar rc
        rewriteTVar wc (pos + 1)
        return pos
      else do
        tryCommit
        return pos
    safeTwiIO $ safePutStrLn
      (show tid ++ " at position " ++ show pos)
```

Fig. 1. Example: Counting committing transactions.

To this end, a global variable *counter* is used for assigning each worker thread a unique commit stamp. As every transaction reads and writes the counter, the counter is heavily contended and causes read inconsistencies in (almost) all concurrently running worker threads. As each such inconsistency aborts the respective transaction and restarts it, a lot of work is done repeatedly and the program becomes very inefficient.

Instead of using a *TVar* for the counter, the programmer might contemplate using other synchronization primitives such as mutexes or placing the counter outside the transaction. However, primitives like mutexes do not mingle well with transactions, because the programmer is responsible for their correct use to prevent deadlocks and data races. The latter may not be possible because an STM implementation may perform transparent rollbacks anytime during a transaction. With the remaining alternative, placing the counter outside the scope of the transaction, the resulting information might be too imprecise due to the nondeterministic thread scheduling by the runtime system.

In this situation, Twilight comes to the programmer's aid. Instead of aborting and restarting a transaction whose only inconsistency is caused by the counter variable, the programmer may write code to repair this inconsistency. To this end, each read operation in the atomic block attaches a tag to the read variable. In the twilight code (i.e., the code after the *twilight* operation), the query

isInconsistent t determines if variables with tag t have been found to be inconsistent.

If the counter is the only cause for inconsistencies, then the code obtains a consistent view on the memory using *reload*, recalculates the counter's value, and updates it before finally committing. To avoid deadlocks in the underlying implementation of transactions, the Twilight code can only read and write transactional variables through handles which are returned by read and write accesses in the body of the transaction. As the transaction is now known not to abort anymore, it is safe to output the logging message.

On the other hand, if the counter is found to be consistent, *tryCommit* restarts the transaction in case there are inconsistencies involving the remaining variables. If the remaining variables are also consistent, then *tryCommit* continues the Twilight code knowing that the transaction cannot fail anymore.

4 The Twilight API

4.1 The Parameterized Monad STM

Figure 2 lists the operations of the API.

The parameterized monad $STM\ t\ p\ q\ a$ encapsulates a computation as the body of a transaction. The type parameters describe a computation of this type more closely:

- t is a static transaction identifier which restricts the scope of tags and variable handles to one transaction using monadic encapsulation [7];
- p and q statically indicate the phase of the transaction in the STM monad before and after the computation;
- a denotes the result type of the computation.

The function $atomically :: (\forall t.\, STM\ t\ p\ q\ a) \rightarrow IO\ a$ creates a new transactional scope with a fresh static transaction identifier t and executes its body computation atomically. The operations *gbind* and *gret* generalize the $\gg=$ and *return* operations of the standard *Monad* class to parameterized monads [6][4].

Twilight distinguishes three different phases in a transactional scope, which are indicated by the instantiation of the p and q parameters.

- Code in the *Atm* (atomic) phase enjoys full transactional execution. The STM implementation provided with GHC[4] provides only this phase. Code that runs in this phase is fully isolated from external changes to variables and vice versa. It always sees memory in a consistent state.
- In the *Twi* (twilight) phase, the consistency of the variable values read within the *Atm* phase of the transaction may be checked with respect to their current values. In the presence of inconsistencies a transaction is doomed to fail, unless the programmer switches to the safe phase.

[4] Our examples exploit GHC's convenient customization feature of the **do** notation through a simple local redefinition of $\gg=$ and *return* by *gbind* and *gret*.

```
  -- STM data
data STM t p q a = ...   -- abstract type of computations
data TVar a      = ...   -- transactional variables
data RTVar t a   = ...   -- handle for rereading
data WTVar t a   = ...   -- handle for rewriting
  -- static states of a transaction
data Atm
data Twi
data Safe
  -- STM parameterized monad
atomically :: (∀t. STM t p q a) → IO a
gbind      :: STM t p q a → (a → STM t q s b) → STM t p s a
gret       :: STM t p p a
  -- transfer between phases
twilight     :: STM t Atm Twi Bool
reload       :: STM t Twi Safe ()
tryCommit    :: STM t Twi Safe ()
  -- read and write operations
newTVar      :: a → STM t p p (TVar a)
readTVar     :: TVar a → Tag t a → STM t Atm Atm (a, RTVar t a)
writeTVar    :: TVar a → a → STM t Atm Atm (WTVar t a)
rewriteTVar  :: WTVar t a → a → STM t p p ()
rereadTVar   :: RTVar t a → STM t p p a
  -- tags
newTag :: STM t Atm Atm (Tag t a)
isInconsistent :: Tag t a → STM t p p Bool
getInconsistencies :: Tag t a → STM t Safe Safe [(RTVar t a, Maybe (WTVar t a))]
  -- embedding IO
unsafeTwiIO :: IO a → STM t p p a
safeTwiIO   :: IO a → STM t Safe Safe a
```

Fig. 2. Twilight API

– Once the *Safe* phase is reached, the transaction does not fail anymore unless explicitly requested by the programmer. The twilight code has exclusive access to the variables in the transaction's write set. This concurrency guarantee ensures that the I/O effects coincide with the outcome of the transaction. In this phase, the code may perform operations for repairing inconsistencies. It is also possible to perform irreversible operations like I/O immediately.

Each STM operation is indexed by its start and end phases. Hence, the type checker guarantees that it is not possible to perform the operations out of order or in the wrong phase. The *twilight* operation switches from the *Atm* phase to the *Twi* phase. Similarly, *reload* finishes the *Twi* phase and starts the *Safe* phase. The operation *tryCommit* also switches from *Twi* to *Safe*, but it aborts and restarts if the transaction is still in an inconsistent state. Otherwise, it proceeds in the *Safe* phase.

4.2 Reading and Writing Shared Memory

As in the STM implementation for Haskell provided with GHC, a shared memory location which can be accessed within a transaction has type $TVar\ a$. It holds a value of type a. The operation *newTVar* creates a new transactional variable with an initial value. Within an atomic section, a $TVar\ a$ can get accessed via *readTVar*. The operation returns the current value of type a as well as a handle of type $RTVar\ t\ a$ where t is the current transaction identifier. This handle is associated to the same location as the underlying $TVar$ and it may be used in the *Safe* phase to read the new value of the variable if it was the cause of an inconsistency. Similarly, the atomic write operation *writeTVar* returns a write handle of type $WTVar\ t\ a$ to enable writing this variable in the *Safe* phase.

After entering the Twilight zone, transactional variables can only be read or written via the read and write handles. Admitting a read operation on a $TVar$ might read a memory location that has not been touched in the preceding *Atm* phase. Similarly, the *Safe* phase must not write variables that have not yet been written to in the *Atm* phase. We impose this restriction to keep the transaction's read and write set constant, which we need in order to give sufficient concurrency guarantees and deadlock freedom.

The operation *rereadTVar* returns the value of a variable as it is currently found in the read set. Within the *Atm* phase, it returns the same value as the *readTVar* operation on the associated $TVar$. After issuing a *reload*, in the *Safe* phase, the *rereadTVar* operation may return a new value if the underlying variable has caused an inconsistency and *reload* has obtained a new value for it. The operation *rewriteTVar* updates the variable corresponding to the $WTVar$ handle, but this update only takes effect when the transaction commits.

4.3 Tags

A tag $Tag\ t\ a$ names a group of variables of type $TVar\ a$. The operation *newTag* returns a fresh tag without any variables attached to it. Its scope is restricted to the execution of the STM monad with static transaction identifier t. Each read operation associates a $TVar$ with a tag unless the variable was already tagged before: A $TVar$ may belong to only one tag in a transaction.

In the Twilight zone, the programmer can apply *isInconsistent* to a tag to determine whether the tag is associated to an inconsistent variable. In the *Safe* phase, the function *getInconsistencies* returns a list of read handles and corresponding write handles (if the variable has been written to in the *Atm* phase) to the inconsistent $TVar$s associated with this tag.

4.4 Embedding I/O into STM

The STM monad whipped with GHC prohibits performing I/O within a transaction because I/O might violate the transactional semantics. Yet, it might sometimes be desirable to include "transaction-safe/harmless" actions, like reading

the system time or printing debugging output, into transactional code. The operation *unsafeTwiIO* injects an I/O action into the STM monad without giving guarantees about when and how often the action may be executed. In contrast, an action performed with *safeTwiIO* in the *Safe* phase of a transaction is guaranteed to be executed exactly once.

5 Implementation

The STM data type is a composition of the IO monad with error and state monads. The state maintains a global counter to obtain unique time stamps that marks write to transactionally managed memory.

We implemented the Twilight STM as a conservative extension of the TL2 algorithm [2]. A detailed description of the implementation can be found in the accompanying technical report [1], including proofs of transactional properties related to atomicity and isolation.

TL2 alike, the implementation relies on a global counter/timer `T`. Each shared variable is associated with a version number that represents the time of its last modification. The first (transactional) read of a variable creates an entry in the read set comprising the variable, its value, and its version number at the time of the read operation. Write operations to shared variables are performed lazily. They are first recorded locally in the transaction's write set.

5.1 Technical Challenges

The Twilight implementation in Haskell has to deal with several issues that require some trickery:

Parameterized monads A parameterized monad [6] serves to separate and order the different phases that a transaction passes through. Further, we use monadic encapsulation [7] to restrict the scope of tags and handles to single transactions.

Enumerating TVars Each *TVar* has a lock to grant exclusive access to the variable. The locking protocol which is underlying Twilight requires that these locks are ordered to avoid deadlocks of the system. As in Haskell, pointers (even stable pointers) are not instance of the *Ord* class, we were forced to use integers to enumerate and order all *TVar*s. This numbering introduces a bottleneck in the implementation and consumes space, but we are not aware of a better solution without reimplementing *IORef*s and *MVar*s.

Wait and Notify. Both the *twilight* and the *reload* operations require exclusive access to *TVar*s. To avoid spin-locking during lock acquisition, we use a semaphore based on a wait and notify mechanism. It allows several transactions to proceed in the lock acquisition in either *twilight* or *reload*. The semaphore is biased towards the *reload* operation to avoid that conflicting transactions accumulate in the twilight phase.

$$
\begin{array}{lll}
x \in \mathrm{Var} & & l \in \mathrm{Ref} \\
v \in \mathrm{Val} & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid () \mid \lambda x.e \mid \texttt{error} \\
e \in \mathrm{Exp} & ::= & v \mid x \mid e\ e \mid \texttt{if}\ e\ e\ e \\
& & \mid \texttt{return}\ e \mid e \gg\!= e \mid \texttt{twilight} \mid \texttt{tryCommit} \mid \texttt{reload} \\
& & \mid \texttt{spawn}\ e \mid \texttt{atomic}\ e \mid (e, W_i, R_i, i, e, \mathcal{H}, f) \\
& & \mid \texttt{newref}\ e \mid \texttt{readref}\ e \mid \texttt{writeref}\ e\ e \\
& & \mid \texttt{update}\ e\ e \mid \texttt{reread}\ e \mid \texttt{inconsistent}\ e \mid \texttt{error}
\end{array}
$$

Fig. 3. Syntax of Λ_{Twi}. Expressions marked in gray arise only during evaluation.

$$
\begin{array}{llll}
l & \in \mathrm{Ref} & & \\
\mathcal{P} & \in \mathrm{Program} & = \mathrm{ThreadId} \rightharpoonup \mathrm{Exp} & \\
T & \in \mathrm{Transaction} & = \mathrm{Exp} \times \mathrm{Store} \times \mathrm{Store} \times \mathrm{Id} \times \mathrm{Exp} \times \mathrm{Store} \times \mathrm{Flag} & \\
\mathcal{H}, R_i, W_i & \in \mathrm{Store} & = \mathrm{Ref} \rightharpoonup \mathrm{Val} \times \mathrm{Id} & \\
\alpha & \in \mathrm{Effect} & = \{at_i, ab_i, co_i(\bar{l}), r_i(l), \epsilon\} & \\
f & \in \{ok, bad, \cdot\} & &
\end{array}
$$

Fig. 4. State related definitions.

6 Formalization

This section introduces Λ_{Twi}, a formalization of the Twilight STM. We define a call-by-value lambda calculus with references, thread spawning and monads. With Λ_{Twi}, we can focus on the relevant parts of the semantics: the access to the shared memory.

6.1 Syntax

Figure 3 shows the syntax of Λ_{Twi}. Values are either references, boolean constants, the unit constant, functions, errors, or monadic expressions. Expressions comprise these values, variables, function application, spawning of threads, transactions, and operations on references. As usual, the expression $e_1; e_2$ abbreviates $(\lambda x.e_2)\ e_1$ where x does not appear free in e_2 and $e[v/x]$ denotes the capture-avoiding substitution of x by v in e.

We can define a monadic type system for Λ_{Twi}in the standard way and assume that all programs we consider for now are type-correct with respect to this type system.

6.2 Operational Semantics

For the operational semantics we introduce some further definitions in Figure 4. A program state consists of a heap, a mapping of thread identifiers to expressions to be evaluated concurrently, and the transactions which is currently executing its twilight code. The execution of a program is represented by a sequence of program states.

Evaluation contexts:

$$\begin{aligned}\mathcal{E} &= [] \mid \mathcal{E}\ e \mid (\lambda x.e)\ \mathcal{E} \mid \texttt{if}\ \mathcal{E}\ e\ e' \\ \mathcal{M} &= [] \mid \texttt{newref}\ \mathcal{E} \mid \texttt{readref}\ \mathcal{E} \mid \texttt{writeref}\ \mathcal{E}\ e \mid \texttt{writeref}\ l\ \mathcal{E} \\ &\quad \mid \texttt{return}\ \mathcal{E} \mid \mathcal{M} \gg\!= e \\ &\quad \mid \texttt{update}\ \mathcal{E}\ e \mid \texttt{update}\ l\ \mathcal{E} \mid \texttt{reread}\ \mathcal{E} \mid \texttt{inconsistent}\ \mathcal{E}\end{aligned}$$

Expression evaluation $\rightarrow$:

$$\begin{aligned}\mathcal{E}[(\lambda x.e)\ v] &\rightarrow \mathcal{E}[e[v/x]] \\ \mathcal{E}[\texttt{if tt}\ e\ e'] &\rightarrow \mathcal{E}[e] \\ \mathcal{E}[\texttt{if ff}\ e\ e'] &\rightarrow \mathcal{E}[e']\end{aligned}$$

$$\frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']} \qquad e \rightarrow^* e \qquad \frac{e \rightarrow^* e' \qquad e' \rightarrow e''}{e \rightarrow^* e''}$$

Monadic evaluation $\curvearrowright$:

$$\begin{aligned}\mathcal{M}[\texttt{return}\ v \gg\!= m] &\curvearrowright \mathcal{M}[m\ v] \\ \mathcal{M}[\texttt{error} \gg\!= m] &\curvearrowright \mathcal{M}[\texttt{error}]\end{aligned}$$

$$\frac{e \rightarrow e'}{\mathcal{M}[e] \curvearrowright \mathcal{M}[e']} \qquad m \curvearrowright^* m \qquad \frac{m \curvearrowright^* m' \qquad m' \curvearrowright m''}{m \curvearrowright^* m''} \qquad \frac{m \curvearrowright m'}{\mathcal{M}[m] \curvearrowright \mathcal{M}[m']}$$

Fig. 5. Semantics: Evaluation contexts.

A transaction T_i is a tuple $(e, W_i, R_i, i, e, \mathcal{H}, f)$. It consists of the expression that is currently evaluated, the write set and the read set of the transaction, a (unique) transaction identifier, a copy of the whole expression that is to be evaluated transactionally for rollbacks, a copy of the heap taken at the begin of the transaction or during a reload, and a flag denoting the transaction's status.

A reference corresponds to a heap location. All stores, i.e., the heap, the read set, and the write set of a transaction, map references to pairs of values and transaction identifiers denoting the transaction which committed or, in case of the write set, attempts to commit the value to the global store. $S(l)$ defines the lookup of a reference l in a heap S if $l \in \mathrm{dom}(S)$. An updated heap $S[l \mapsto y]$ corresponds to the heap S, but maps l to y. For two stores S_1 and S_2, we write $S_1[S_2]$ for the updated version of S_1 with all entries of S_2.

Finally, operations can have different effects on the global state: the begin (at_i), abort (ab_i), read ($r_i(l)$), and commit ($co_i(\bar{l})$) operations to the global shared heap, or an empty effect (ϵ).

The evaluation of a program starts in an initial state $\langle\rangle, \{t_0 \mapsto e\}, \cdot$ with an empty heap and a main thread t_0. A terminal state has the form $\mathcal{H}, \{t_0 \mapsto v_0, \ldots, t_n \mapsto v_n, \cdot$. The rules in Figures 5-7 define the semantics of the language constructs.

$\mathcal{E}[\bullet]$ denotes the evaluation context for expressions and $\mathcal{M}[\bullet]$ the corresponding one for monadic expressions.

$$\frac{\mathcal{P}(t) = m \qquad m \curvearrowright m'}{\mathcal{H}, \mathcal{P}, s \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto m'], s} \text{ IO-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{spawn}\ m] \qquad t' \text{ fresh}}{\mathcal{H}, \mathcal{P}, s \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{return}\ ()], t' \mapsto m], s} \text{ Spawn}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{atomic}\ m] \qquad T = (m, \langle\rangle, \langle\rangle, i, m, \mathcal{H}, \cdot) \qquad i \text{ fresh}}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{at_i} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[T]], \cdot} \text{ Atomic}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}', \cdot)] \qquad m \curvearrowright m''}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(m'', W_i, R_i, i, m', \mathcal{H}', \cdot)]], \cdot} \text{ STM-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{newref}\ v], W_i, R_i, i, m', \mathcal{H}', \cdot)] \qquad l \notin \mathrm{dom}(\mathcal{H}) \cup \mathrm{dom}(W_i)}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ l],], W_i[l \mapsto (v, i)], R_i, i, m', \mathcal{H}') \cdot], \cdot} \text{ Alloc}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\ \texttt{writeref}\ l\ v], W_i, R_i, i, m', \mathcal{H}', \cdot)]}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()],], W_i[l \mapsto (v, i)], R_i, i, m', \mathcal{H}') \cdot], \cdot} \text{ Write}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}', \cdot)] \\ l \notin \mathrm{dom}(W_i) \cup \mathrm{dom}(R_i) \qquad \mathcal{H}(l) = \mathcal{H}'(l) = (v, j)\end{array}}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{r_i(l)} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ v],], W_i, R_i[l \mapsto (v, j)], i, m', \mathcal{H}') \cdot], \cdot} \text{ ReadGlobal}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}', \cdot)] \\ l \notin \mathrm{dom}(W_i) \qquad R_i(l) = (v, i)\end{array}}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ v],], W_i, R_i, i, m', \mathcal{H}') \cdot], \cdot} \text{ ReadRSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}', \cdot)] \qquad W_i(l) = (v, i)}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ v],], W_i, R_i, i, m', \mathcal{H}') \cdot], \cdot} \text{ ReadWSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}', \cdot)]}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{ab_i} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{atomic}\ m']], \cdot} \text{ Rollback}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{twilight} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', \cdot)] \qquad \mathit{check}(R_i, \mathcal{H}) = \mathit{ok}}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[m\ \texttt{tt}], \mathcal{H}, W_i, R_i, i, \mathcal{H}, \mathit{ok})]], t} \text{ TwiOk}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{twilight} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', \cdot)] \\ \mathit{check}(R_i, \mathcal{H}) = \mathit{bad}\end{array}}{\mathcal{H}, \mathcal{P}, \cdot \xRightarrow{ab_i} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[m\ \texttt{ff}], \mathcal{H}, W_i, R_i, i, \mathcal{H}, \mathit{bad})]], t} \text{ TwiBad}$$

Fig. 6. Operation semantics I.

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)] \qquad R_i(l) = \mathcal{H}(l)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return ff}], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)]], t} \textsc{Incons-1}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], \mathcal{H}, W_i, R_i, i, f,]) \qquad R_i(l) \neq \mathcal{H}(l)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return tt}], \mathcal{H}, W_i, R_i, i, f,])], t} \textsc{Incons-2}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], \mathcal{H}, W_i, R_i, i, f,]) \qquad R_i(l) \neq \mathcal{H}(l)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], \cdot} \textsc{InconsErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{tryCommit} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', ok)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[m\ ()], W_i, R_i, i, m', \mathcal{H}', ok)]], t} \textsc{TryCommitOk}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{tryCommit} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', bad)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{ab_i} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{atomic}\ m']], s} \textsc{TryCommitBad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reload} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', ok)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{at_j, r_j(l)} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[m\ ()], W_i, R_i, i, m', \mathcal{H}', ok)]], t} \textsc{ReloadOk}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reload} \gg\!= m], W_i, R_i, i, m', \mathcal{H}', bad)] \\ j \text{ fresh} \qquad R_j = \{l \mapsto \mathcal{H}(l) | l \in \mathrm{dom}(R_i)\}\end{array}}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[m\ ()], W_j, R_j, j, m', \mathcal{H}', ok)]], t} \textsc{ReloadBad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{update}\ l\ v], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)] \qquad l \in \mathrm{dom}(W_i)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], \mathcal{H}, W_i[l \mapsto (v, i)], R_i, i, \mathcal{H}, f)]], t} \textsc{Update}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{update}\ l\ v], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)] \qquad l \notin \mathrm{dom}(W_i)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], \cdot} \textsc{UpdateErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reread}\ l], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)] \qquad R_i(l) = (v, j)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ v], \mathcal{H}, W_i, R_i, \mathcal{H}, i, f)]], t} \textsc{Reread}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reread}\ l], \mathcal{H}, W_i, R_i, i, \mathcal{H}, f)] \qquad l \notin \mathrm{dom}(R_i)}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], \cdot} \textsc{RereadErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\texttt{return}\ v, W_i, R_i, i, m', \mathcal{H}', ok)] \qquad \mathcal{H}' = \mathcal{H}[W_i']}{\mathcal{H}, \mathcal{P}, t \xrightarrow{co_i(\bar{l})} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[v]], \cdot} \textsc{Commit}$$

$$\frac{\forall l \in \mathrm{dom}(R_i) : R_i(l) = \mathcal{H}(l)}{check(R_i, \mathcal{H}) = ok} \textsc{Check-Ok}$$

$$\frac{\exists l \in \mathrm{dom}(R_i) : R_i(l) \neq \mathcal{H}(l)}{check(R_i, \mathcal{H}) = bad} \textsc{Check-Bad}$$

Fig. 7. Operational semantics II

The `IO` monad is the top-level evaluation environment. Each reduction step $\stackrel{\alpha}{\Longrightarrow}$ chooses an expression from the thread pool $\mathcal{P}$. The non-determinism in this choice models an arbitrary scheduling of threads.

An atomic expression at the top-level (Atomic) creates a new transaction with the expression to be evaluated, an empty read and write set, and a fresh transaction identifier. Further, a copy of the expression is needed for possible rollbacks, and a copy of the current heap to mark the beginning of the transaction. The transaction then becomes the new innermost evaluation context.

Inside a transaction, the global state only influences the read operation on references (ReadGlobal). If a reference cannot be read from the local read or write set, it is accessed in the current global heap. To maintain the transaction's consistency, the read operation is successful only if the value has not been updated since the transaction's beginning. The value and transaction identifier as registered in the heap for this reference are then added to the read set, and the value is returned to the transactional computation.

A transaction may be aborted and rolled back via Rollback . This rollback may happen nondeterministically.

Before committing, the transaction must switch from the `STM` monad to the `TWI` monad with the function `twilight`. At this point, the heap is checked for updates to the references which are found in the transaction's read set since the start of the transaction. There are two cases:

Rule TwiOk applies if the check is successful: none of the variables read by the transaction have been committed by another transaction in the mean time.

Rule TwiBad applies if the check fails. It aborts the transaction so far, and continues the evaluation of the following twilight code in a new transactional context. This context inherits the read and write set of the aborted transaction. If in the `TWI` monad the transaction's consistency is restored with a reload via `reload`, the transaction can then repair inconsistencies and still commit successfully.

Errors are thrown by invalid read or write operations inside the twilight zone. A read/write operation is illegal in the twilight code if its location has not been read/written in the preceding `STM` phase of the transaction. These errors are propagated to the top-level with rule IO-Error . They abort the enclosing transaction and terminate the execution of the associated thread.

To ensure that reduction sequences are serializable, we require that $\operatorname{dom} W_i \subseteq \operatorname{dom} R_i$. Reading a heap location yields either the value which the transaction itself has written or the value which is stored in the read set.

6.3 Serializability of Twilight Transactions

The standard property most STM systems provide is serializability. It states that any allowed interleaving of transactions must have an equivalent result compared to an execution where the transactions are executed after each other.

We can prove that the semantics for Λ_{Twi} satisfies serializability under certain restrictions. To this end, we propose a definition for well-formedness of execution traces in terms of the effects they exhibit.

Definition 1 (Well-formed traces). *For a sequence of reduction steps* $R = \mathcal{H}_0, \mathcal{P}_0 \xRightarrow{\overline{\alpha_1}} \ldots \xRightarrow{\overline{\alpha_n}} \mathcal{H}_n, \mathcal{P}_n$*, its* trace *is the sequence* $tr(R) = \overline{\alpha_1}, \ldots, \overline{\alpha_n}$*. The trace is* well-formed *iff the following conditions hold:*

- *For* $\alpha_j \in tr(R)$*,* $\alpha_j \in \{at_i, ab_i, co_i(\bar{l}), r_i(l), \epsilon\}$*.*
- *There is no read or commit effect for a transaction* i *before its atomic or abort effect.*
- *There is no read or commit effect for a transaction* i *after its commit or abort effect.*
- *A transaction may have either a commit or an abort effect, but not both. A transaction is pending if it has neither a commit or abort effect.*

One can then show that reordering certain evaluation steps leads to equivalent reductions sequences. Reductions are considered equivalent if each read operation returns the same value, each commit operation commits the same values, and each transaction's outcome (abort or commit) is the same. To see which reordering will give equivalent reductions, we define a notion of dependency on effects. Only independent effects may get reordered while preserving the result of evaluating the program.

Finally, we can show that all reduction sequences produced by a program whose twilight zones either aborts a transaction or repairs all inconsistencies accordingly to the calculations in the atomic section are equivalent to some reduction sequence with a serial trace, up to the assignment of unique labels to the transactions.

Further details will be provided in the final version of the paper.

7 Conclusion

We presented Twilight STM, an implementation of software transactional memory that extends the commit phase of the transaction with the execution of twilight code. Twilight code consists of arbitrary user code that runs with special concurrency privileges. Consequently, the code to safely perform irrevocable I/O actions. Furthermore, the twilight code may use the Twilight API to introspect and modify the transaction's state, e.g. to implement sophisticated contention management. The notion of tags offer an intuitive and practical mechanism to identify areas where read conflicts arose.

We experimented with implementations of Twilight for several languages. Unlike the imperative paradigm, the functional paradigm imposes a clear separation between transactional and twilight code. The transactional code is a pure atomic function between inputs and outputs, whereas side effect may only occur in twilight code, where it can be used safely. Additionally, through Haskell's type system, we enforce sanitary restrictions on the twilight code. For example, we statically enforced that only those variables that actually occur in the write set of the transaction, can be updated.

References

1. Bieniusa, A., Middelkoop, A., Thiemann, P.: Actions in the Twilight: Concurrent irrevocable transactions and inconsistency repair (extended version). Tech. Rep. 257, Institut für Informatik, Universität Freiburg (2010)
2. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006. pp. 194–208. LNCS 4167, Springer (2006)
3. Harris, T.: Exceptions and side-effects in atomic blocks. Science of Computer Programming 58(3), 325–343 (2005)
4. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: Sixteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 48–60. ACM Press, Chicago, IL, USA (Jun 2005), `http://research.microsoft.com/~simonpj/papers/stm/stm.pdf`
5. Harris, T., Stipić, S.: Abstract nested transactions. In: TRANSACT '07. Portland, OR, USA (Aug 2007), `\url{http://www.cs.rochester.edu/meetings/TRANSACT07/papers/harris.pdf}`
6. Kiselyov, O., chieh Shan, C.: Lightweight monadic regions. In: Gill, A. (ed.) Haskell 2008. pp. 1–12. ACM, Victoria, BC, Canada (Sep 2008)
7. Launchbury, J., Peyton Jones, S.L.: State in Haskell. Lisp and Symbolic Computation 8(4), 293–341 (Dec 1995)
8. Spear, M.F., Michael, M.M., Scott, M.L.: Inevitability mechanisms for software transactional memory. In: TRANSACT '08 (2008)
9. Welc, A., Hosking, A.L., Jagannathan, S.: Transparently reconciling transactions with locking for Java synchronization. In: 20th ECOOP. LNCS, vol. 4067, pp. 148–173. Springer, Nantes, France (Jul 2006)
10. Welc, A., Saha, B., Adl-Tabatabai, A.R.: Irrevocable transactions and their applications. In: SPAA '08: Proc. Twentieth Annual Symposium on Parallelism in Algorithms and Architectures. pp. 285–296. ACM, New York, NY, USA (2008)

Towards Orthogonal Haskell Data Serialisation

— Work in progress, preliminary version —

Jost Berthold

e-Science Center
University of Copenhagen
berthold@diku.dk

Abstract. This paper investigates a novel approach to serialisation of Haskell data structures with a high degree of flexibility, based on runtime support for parallel Haskell on distributed memory platforms. This serialisation has highly desirable and so-far unrivalled properties: it is truly orthogonal to evaluation and does not require any type class mechanisms. Especially, (almost) any kind of value can be serialised, including functions and IO actions. We outline the runtime support on which our serialisation is based, and present different versions of the wrapper code in Haskell which can ensure type safety of the serialisation process, as well as application ideas.

1 Introduction

Serialisation of data is a crucial feature of real-world programming systems. Being able to write out and read in data structures without complications provides a simple and straightforward way of saving and restoring an application's configuration, and generally enables a program's state to persist between runs. Mainstream and scripting languages like Java, Ruby, Perl and Python provide powerful libraries for this purpose by default, concentrating on ease of use (consider e.g. Python's pickle interface or the omnipresent JSON format for object-oriented web programming). There is considerable interest and demand for general serialisation features in Haskell. In the past year, we have come across related requests on mailing lists or talked about it in personal discussion at several opportunities [5, 1, 3, 2]. For the functional world, the characteristic equal treatment of a program (functions) and its (heap) data poses additional challenges to the design and implementation of serialisation features, complicated further in case of lazy languages (Haskell, in particular).

This paper proposes and explores a route to serialisation which uses techniques from parallel Haskell. Implementations of parallel Haskell variants for distributed memory systems obviously require to transfer data from one heap to another. In essence, this means nothing else than to serialise and deserialise data structures of various kind, in any evaluation state. This is by far the most crucial and error-prone part of the parallel Haskell implementations Eden and GUM [12, 8, 17]. At the same time, it needs to be integrated part of the runtime

system. Outsourcing its functionality into a library appears to require rather specialised and complex support features (we investigated in [7] and [6]). On the other hand, it offers the needed runtime support for a Haskell serialisation mechanism, with interesting properties. In this paper, we explore the technical and conceptual limits of using the parallel Haskell graph packing routines, and necessary extensions, for a Haskell serialisation approach which is, by its very nature, orthogonal to evaluation and not restricted to particular data types.

The remainder of this paper is organised as follows: After briefly summarising related work in Sec. 2, Sec. 3 presents the essentials of a serialisation feature for Haskell, discussing implementation and potential problems. Applications of the approach are discussed in Sec. 4, and the final section concludes with future work.

This research is work in progress, which we wish to present for discussion at the IFL symposium.

2 Related Work

As already mentioned in the beginning, serialisation is a common standard feature present in many programming languages. We put the focus of our discussion on approaches specific to lazy functional languages, considering mechanisms for serialisation and persistence.

A very simple, yet unsatisfying, serialisation (and thereby data persistence) is to use the Show and Read class instances. Data types which define instances for these classes can simply be written to a string (`show`) and parsed back in (`read`). Efficiency can be drastically improved by using binary I/O (various libraries have been proposed), and integrated approaches like [16], which uses additional type classes, offer more programming comfort. More recently, we also see efforts to improve efficiency by reproducing the original sharing structure when reading in data [9].

However, such approaches will imply that the data is going to be evaluated before serialisation, and cyclic data structures will make the serialisation loop[1]. The real challenge in combining serialisation and laziness is to serialise only partially evaluated values. Only if data evaluation and data serialisation are truly orthogonal, a library for persistence can be established where previous evaluations are reused later. Efforts have been made in the past to join lazy functional languages and persistence in an orthogonal way.

McNally [14, 13] has pioneered these questions with the STAPLE programming system. STAPLE adds persistence to a lazy functional language, an integrated whole-system approach with interpreted purely functional interface (related to Miranda and the then-upcoming Haskell). Stream I/O instead of monads and the simplistic interpreter interface characterises it as early pioneering work

[1] In the Haskell wiki [4], we have found allusions to a library SerTH that allegedly supports cyclic data structures and uses template haskell. All provided links are however dead, this library seems to have perished (including all substantial information about its implementation techniques).

in the field. To our knowledge, no existing system is similar to the STAPLE persistent store, but it has set directions for several successors.

One such strand of work is [15], which describes a concept for adding persistence to the GHC runtime, more precisely the GUM [17] system. As in GUM, special closures with fetch semantics are used for retrieving persistent data, which is stored in an external generic persistent object store. While these mechanisms remain completely transparent, the programming interface to the system requires to explicitly open a store and retrieve and store values. The approach of the authors is based on the same runtime system features as our own proposal, yet the paper stays with a high-level design and does not present working solutions to the inherent problems (some of which are nevertheless discussed).

The system which comes closest to what we outline here is Clean which provides dynamics [18]. Clean dynamics solve both the problem of runtime typing and retain the laziness of stored data, while allowing to transfer data between different applications. However, as we will see subsequently, this requires an integrated systemic design around an "application repository" which contains all functions referenced by a persistent data item.

Very limited support for dynamics is included in the Haskell base library [11] as well, in `Data.Dynamic`. We mention it here because we are going to use the underlying `Data.Typeable` to solve typing probelms in our approach. Based on the module `Data.Typeable` which provides runtime type reification and guarded type casts, data can be converted to a `Dynamic` and back to its original type (failing at runtime if the type is wrong). Haskell Dynamics are very limited, since a Dynamic can only be used in the same *run* of the same program (binary).

3 Implementation

3.1 Runtime System Support

Our implementation of heap data serialisation is based on functionality needed for the Eden[12] and GUM[17] parallel variants of Haskell, namely graph packing. The runtime system (RTS) for Eden and GUM contains methods to traverse and pack a computation graph in the heap in breadth-first manner, and the respective counterpart for unpacking. During the traversal, unevaluated data (thunks) are packed as the function to apply and its arguments, so they can be evaluated on the receiver side. Cyclic graph structures are broken up by using back references into the previous data.

Heap closures in GHC are laid out in memory as a header section containing metadata, followed by all pointers to other graph nodes, and then by all non-pointers[2]. The essential and first header field is the *info pointer*, pointing to information as e.g. the amount of pointers and non-pointers in the particular heap closure, and the entry code.

[2] The packing implementation of the Eden and GUM runtime system is described here only for the standard layout, leaving out special cases.

When packing data, the respective computation graph structure is traversed and serialised. Newly met closures are packed as their info pointer address and non-pointer data. Pointers will be reestablished from the global packet structure when unpacking. When a closure is met again, it will not be packed a second time. Instead, the packet will contain a special marker (REF) for a back reference and the relative position of the previously packed closure in the packet. As this requires a starting marker field, normal closures will start by another marker (CLO). A third type of marker indicates static closures (constant applicative forms).

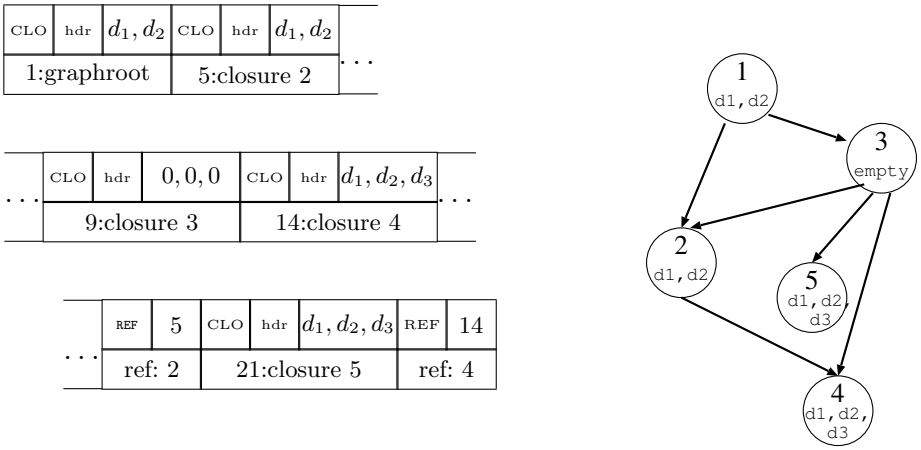


Fig. 1. Example serialisation packet for a computation graph

Figure 1 shows the resulting packet layout for a small example (with header data assumed to be just one word, the info pointer). The example packet contains a complete subgraph of 5 nodes, the graph structure depicted on the right. Pointer fields are left out and will be filled with references to the subsequent closures upon unpacking, using a closure queue.

The packing algorithm in its current form uses static memory addresses of the program's (and libraries') functions, thus assuming the exact same binary is receiving the data. Furthermore, packing may currently fail because a graph structure in the heap is bigger than a configurable maximum buffer size. Graph packing works on most closure types in GHC and is therefore to a large extent independent of the data type to be packed. The notable exception is mutable variables (MVars) and transactional variables, which cannot be packed. Data structures which contain MVars typically do not make sense to transfer to a new location, since they imply and store system state. Examples include IO file handles, as well as semaphores and other synchronisation structures.

3.2 Heap to Array: Unsafe Dynamics

As the first step towards our Haskell serialisation, we define a primitive operation which, instead of sending a serialised graph over the wire, returns the data in a

Haskell byte array. A byte array is a primitive data type in GHC which provides a chunk of heap for unboxed non-pointer values. This type is the basis of every higher-level array which contains unboxed values (as opposed to "boxed" values, which are pointers to other heap structures).

```
-- primitive operations
serialize# :: a -> State# s -> (# State# s, ByteArray# s #)
deserialize# :: ByteArray# -> State# s -> (# State# s, a #)

-- IO Monad wrapper
heapToArray :: a -> IO (UArray Int Word)
heapFromArray :: UArray Int Word -> IO a
```

Shown here are the types of the primitive operations and two small wrapper functions which provide an IO-monadic version and lift the result type to an unboxed array (`UArray`) of Words. These two functions `heapToArray` and `heapFromArray` provide the minimum wrapper around the primitive operation: the functions return a regular Haskell data structure in the IO monad.

When used in a disciplined manner, these functions can already prove useful, for they contain all necessary data to reconstruct the serialised structure in its current evaluation state. However, no type safety is provided: any type of value can be serialised and later (attempted to) unpacked and used as any other data type, using the raw data in the array. A programmer might well be aware of this problem, but make wrong assumptions on the type inference defaults in the compiler, as the following example illustrates.

An unintended type cast

```
lucky = do let list = map (2^) [30..40]          -- defaults to [Integer]
           pack <- heapToArray list
           copy <- heapFromArray pack ::IO [Int] -- explicit (wrong) type
           putStrLn (show copy)
```

In our example, a list of integer values is computed (its default type is a large (GMP-based) `Integer`). When unpacking the serialised data, addresses are misinterpreted (for the large numbers) as a 32bit `Int`, leading to the following wrong program output:

Output when running `lucky`*:*

```
[1073741824,17004549,33781765,33781765,33781765,33781766,33781766,...
```

Unpacking a serialised value into a value of the wrong type can lead to all kinds of runtime errors. The subsequent code makes false assumptions about its heap representation and enters the data structure at the wrong point. In the best case, the two types have a "similar" underlying heap representation, but might still be misinterpreted, as in our example above. Other cases might lead to complete failure.

3.3 Phantom Types: Type-safe Dynamics in one Program Run

In order to provide more type safety, we wrap the array containing the serialised data inside a structure which uses a phantom type:

```
data Serialized a = Serialized { packetSize :: Int
                               , packetData :: ByteArray#}
serialize :: a -> IO (Serialized a)
deserialize :: Serialized a -> IO a
```

Including the original type into the `Serialized` type in this way ensures that the type checker refuses ill-typed programs with a meaningful error message. The type cannot be freely given any more, but is encapsulated in the data structure which is passed to the deserialisation. Data can be restored in a typesafe manner within the same program run now. In our example, the type `[Int]` explicitly given to the unpacked value propagates up to the original.

Type propagating up

```
works = do let list = map (2^) [30..40]        -- inferred: [Int]
           pack <- serialize list              -- inferred: Serialize [Int]
           copy <- deserialize pack ::IO [Int] -- this fixes the type
           putStrLn (show copy)

-- output: [1073741824,-2147483648,0,0,0,0,0,0,0,0,0]
```

3.4 Type-safe Persistence

With the previously presented code in place, a self-suggesting idea is to store and read back in the array which represents the serialised data. We can define Show and Read instances for `Serialized a` types, which allows one to write a representation to a file and read it later. Essentially this is what we need in order to realise *persistence*, i.e. keeping partially evaluated data in an external store and loading it into a program. Two problems become apparent when doing so, only one of which can be solved easily.

Dynamic Type Checks. The *first problem* is again one of typing: when reading `Serialized a` from a file and deserializing the represented data, the type a of these data has got to be accurate, either given by the programmer or inferred. Since a user can (attempt to) read from any arbitrary file, this type check needs to happen at runtime. Figure 2 shows how to realise this dynamic type check by the library `Data.Typeable`, which provides type reification mechanisms to Haskell for predefined types (but restricting the serialisation to monomorphic types since `Typeable` excludes polymorphic types).

First, the `Serialized` data structure now needs to include type information. Second, the represented type has got to be established and checked when reading in and deserialising data. With the phantom type, the right place to do this type

```
data Serialized a = Serialized { packetSize :: Int
                               , packetType :: TypeRep
                               , packetData :: ByteArray#}

instance Typeable a => Read (Serialized a)
    where readsPrec _ input
           = case parseP input of
              [((size,tp,dat),_)] ->
                    let !(UArray _ _ _ arr# ) = listArray (0,size-1) dat
                        t = typeOf (undefined::a)
                    in if show t == tp
                           then [(Serialized size t arr# , r)]
                           else error ("type error during packet parse: "
                                       ++ show t ++ " vs. " ++ tp)
              other -> error "No parse"

-- parser function, not shown
parseP :: :: ReadS (Int,String,[Word])
parseP = ... -- returns packet size, string for type and values list
```

Fig. 2. Serialisation structure including type, and Read instance

check is inside the `Read` instance for `Serialized`, requiring a `Typeable` context.[3] The code checks not the type itself but its string representation – type representation IDs can change from run to run. Also note the use of lexically scoped type variable `a`.

References to Raw Memory Addresses. The *second, much more serious* limitation of the approach is the use of memory addresses (info pointers and static function addresses) in the packing code. The packing algorithm in its current form uses static memory addresses of the program's (and libraries') functions, thus assuming the exact same binary is receiving the data, and that no code relocation takes place.

Dynamic assignment of code addresses (relocatable code) can be dealt with by packing relative offsets to a known reference point (as also mentioned in [15]). Another possibility is to inspect the application at runtime using binary utilities like `nm`. However, if an application is recompiled after making changes to its code, the addresses of static data and the compiler-generated names will necessarily change, thereby invalidating previously produced packet data without a chance of correction for the new binary.

Well-understood, the ability to store and retrieve only partially evaluated data is the main advantage of our proposal, and this property *conceptually requires* to keep references to a program's functions if they are needed in the

[3] This is in contrast to dynamics in Clean [18], where a pattern match on types is performed at the time of unpacking a value from dynamic, and which allows even several different type matches to be performed.

serialised computation. Clean [18] achieves this by a system-wide application store which contains all code referenced from a saved dynamic, requiring special tools for its maintenance (transfer to other machines, deletion, garbage collection of the application store). We consider this a slightly too system-invasive and demanding solution. Better would be to achieve a compromise design where serialisation packets stay independent and self-contained (at the price of higher failure potential at runtime).

To enable data transfer between several applications (and there is no fundamental difference between this and data exchange between different versions of one application), big changes to the packing format will be necessary. Not only does the code need to replace static addresses by relative references, the packet needs to include the static data itself, metadata as well as chunks of code for functions of the program, which has to be dynamically linked at runtime. On the contrary, it is easy to make the system produce an understandable runtime error message when loading data with the wrong version. This can be achieved by including a hash value of the program code into the `Serialized` data and checking that value after parsing.

4 Potential Applications

4.1 Checkpointing Long-running Applications

Checkpointing is the process of storing a snapshot of an application's current state during runtime, in order to restart the application in this state after interruptions.

With the presented serialisation approach, we can realise checkpointing mechanisms to recover long-running Haskell applications from external failures and interruptions, by serialising suitable parts of the program as IO actions. The only condition is that in the program, a variable needs to be bound to the sequence of subsequent IO actions to be carried out from a certain stage of execution. Previous variable bindings referred to by the sequence will then automatically be included in the serialisation, in their current evaluation state. A checkpoint will typically be established at a particular stage of execution, for instance after each step of an iteration. Therefore, the easiest way to achieve this is by an explicit checkpointing loop construct.

We intend to explore and exemplify checkpointing and present language constructs for it at a later time. The final paper will present small demo programs and propose checkpointing language constructs with their implementation.

4.2 Persistent Memoisation for Frequently Used Applications

A second potential application area for the approach (with its present limitations) can be to alleviate computational load of frequently used applications in a production environment. Similar to memoisation techniques in just one program run, it is conceivable to use a *persistent memo table* for supportive data

structures and functions which are repeatedly used in the typical computation performed.

In order to realise persistent memoisation, a program using memoised functions has got to be extended by suitable init and shutdown mechanisms. The shutdown routine will serialise all memoised functions to a file, with their memo tables obtained so far. The init routine of a subsequent run will then load these existing memo table dumps if they exist, or otherwise use the "fresh" definition in the program. At the time of writing, it is unclear to what extent this approach can be realised, yet it appears very promising to build on existing work for generic memoisation in this way.

5 Conclusions and Future Work

We have presented an approach to serialisation for Haskell data structures which is orthogonal to evaluation. This means that data can be serialised independent of its evaluation state, and recovered later by the same program (but potentially in a different run of it). Our approach is, to a large extent, also independent of the data type to be serialised; no special serialisation type class is used. Suitable Haskell wrapper functions ensure that ill-typed usage is detected and leads to meaningful runtime errors instead of unspecific internal failure.

The base implementation for our approach is directly carried over from previous research in the area of parallel Haskell implementations. As such, our prototype currently has conceptual and technical limits. In its current form, we can realise a checkpointing mechanism for iterative Haskell computations (which is ongoing work at the time of writing). Similar approaches have been investigated in the past, but to our knowledge, no previous work has investigated the technical details in comparable depth.

As future work, we plan to investigate how the Haskell code and the runtime system routines should be extended to provide better support for serialisation. The version checksums for serialised data which we have described earlier appear to be a straightforward extension at the Haskell level. In contrast, considerable modifications to the packing routine need to be made in order to support data exchange between several (versions of) applications. The goal is to make the packet format more self-contained, to avoid referencing static data of any kind. In the long run, we can even consider to include dynamically linkable code chunks in the packet. We believe that the parallel implementation can as well profit from these improvements, in the form of extended platform independence and flexibility.

Acknowledgements. We would like to thank all colleagues from Marburg, Edinburgh and St.Andrews who were and are supporting the development of parallel Haskell on distributed memory platforms. Special thanks go to Phil Trinder, who brought up the initial idea to work on serialisation as a separable strand.

References

1. Haskell Café: Discussion on "bulk synchronous parallel". accessed 2010-07-20, http://www.haskell.org/pipermail/haskell-cafe/2010-April/076593.html
2. Haskell Café: Discussion on "how to serialize thunks?". accessed 2010-07-23, http://www.haskell.org/pipermail/haskell-cafe/2006-December/020786.html
3. Haskell Café: Discussion on "persist and retrieve of IO type?". accessed 2010-07-20, http://www.haskell.org/pipermail/haskell-cafe/2010-April/076121.html
4. Haskell Wiki. Wiki, http://www.haskell.org/haskellwiki/, accessed 2010-07-20
5. Augustsson, L.: Personal communication about Haskell serialisation (September 2009)
6. Berthold, J.: Explicit and implicit parallel functional programming: Concepts and implementation. Ph.D. thesis, Philipps-Universität Marburg, Germany (June 2008), http://archiv.ub.uni-marburg.de/diss/z2008/0547/
7. Berthold, J., Loidl, H.W., Al Zain, A.: Scheduling Light-Weight Parallelism in ArTCoP. In: Hudak, P., Warren, D. (eds.) PADL'08 — Practical Aspects of Declarative Languages. Springer LNCS 4902, San Francisco, USA (January 2008)
8. Berthold, J., Loogen, R.: Parallel Coordination Made Explicit in a Functional Setting. In: Horváth, Z., Zsók, V. (eds.) Implementation of Functional Languages (IFL 2006). Springer LNCS 4449, Budapest, Hungary (2007)
9. Corona, A.: Refserialize: Write to and read from Strings maintaining internal memory references. Haskell Library on Hackage, http://hackage.haskell.org/package/RefSerialize-0.2.7
10. Davie, T., Hammond, K., Quintela, J.: Efficient Persistent Haskell. In: Clack, C., Hammond, K., Davie, T. (eds.) Implementation of Functional Languages (IFL'98). Draft Proceedings. London, UK (September 1998), available online
11. Haskell Hierarchical Libraries: Base library. Haskell Library on Hackage, http://hackage.haskell.org/package/base-4.2.0.1, accessed 2010-07-20
12. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming 15(3), 431–475 (2005)
13. McNally, D.J.: Models for Persistence in Lazy Functional Programming. Ph.D. thesis, University of St.Andrews (1993)
14. McNally, D.J., Davie, A.J.T.: Two models for integrating persistence and lazy functional languages. SIGPLAN Not. 26(5), 43–52 (1991)
15. Quintela, J.J., Sánchez, J.J.: Persistent Haskell. In: Moreno-Díaz, R., Buchberger, B., Freire, J.L. (eds.) Computer Aided Systems Theory - EUROCAST 2001, Las Palmas de Gran Canaria, Spain, February 19-23, 2001, Revised Papers. vol. LNCS 2178, pp. 657–667. Springer (2001), presented earlier, at IFL '98, as [10]
16. Santos, A., Abdon Monteiro, B.: A Persistence Library for Haskell. In: Musicante, M.A., Haeusler, E.H. (eds.) SBLP'2001 - V Simpósio Brasileiro de Linguagens de Programação. Proceedings. Curitiba (May 2001)
17. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96. ACM Press (1996)
18. Vervoort, M., Plasmeijer, M.J.: Lazy Dynamic Input/Output in the Lazy Functional Language Clean. In: Pena, R., Arts, T. (eds.) Implementation of Functional Languages (IFL'02). vol. LNCS 2670. Springer (2002)

From Bayesian Notation to Pure Racket
via Discrete Measure-Theoretic Probability in $\lambda_{\mathbf{ZFC}}$

Neil Toronto and Jay McCarthy

PLT @ Brigham Young University
Provo, Utah, USA
neil.toronto@gmail.com and jay@cs.byu.edu

Abstract. We are developing languages to express Bayesian models and queries about models. Answers to queries are generally not finitely computable, so we must settle for converging approximations. To ensure that we have correct answers to converge to, we first create an *exact* semantics. It outputs uncomputable programs in λ_{ZFC}, an untyped call-by-value lambda calculus extended with sets. We identify the approximations we must make, incorporate them into an *approximating* semantics, and implement the approximating semantics in Racket.
Here, we describe preliminary work, in which we have created semantics and an implementation for discrete (countable) probability. The work should extend naturally to uncountable sample spaces.

1 Introduction

We give preliminary results in developing a language for probabilistic modeling and queries. We target **Bayesian practitioners**, whose style of modeling both complicates designing the language and motivates it.

Bayesian practitioners define *models*, or probabilistic relationships among objects of study, without regard to the closed-formedness or tractability of future calculations. They are loathe to make simplifying assumptions. (If some probabilistic phenomenon is best described by an unsolvable integral or infinitely many random variables, so be it.) When they must approximate, they often create two models: an "ideal" model first, and a second model that approximates it.

Because they create models without regard to future calculations, they usually must accept approximate answers to queries about them. Typically, they adapt algorithms that compute converging approximations in programming languages they are familiar with. The process is tedious and error-prone, and involves much performance tuning and manual optimization. It is by far the most time-consuming part of their work—and also the most automatable part.

They follow this process to adhere to an overriding philosophy: an approximate answer to the right question is worth more than an exact answer to an approximate question. Thus, they put off approximating as long as possible.

We also adhere to this philosophy, partially because practicing it tends to reduce compounded errors. More importantly, Bayesian practitioners are unlikely to use a language that requires them to approximate early, or that approximates

earlier than they would. We have found that a good way to put the philosophy into practice in language design is to create two semantics: an "ideal," or *exact* semantics first, and an *approximating* semantics that converges to it.

Theory of Probability. Measure-theoretic probability is the most successful theory of probability in precision, maturity, and explanatory power. In particular, it explains every Bayesian model. We therefore define the exact semantics as a transformation from Bayesian notation to measure-theoretic calculations.

Measure theory treats finite, countably infinite, and uncountably infinite outcomes uniformly, but with significant complexity. Here, we deal with at most countably infinite sets of probabilistic outcomes. We avoid most of the complexity that way while retaining most of the structure. The exact semantics appears to extend naturally to uncountable sample spaces.

Approach and Target Language. For three kinds of Bayesian notation, we

1. Manually interpret an unambiguous subclass of typical notation.
2. Mechanize the interpretation with a semantic function.
3. If necessary, give approximations and prove convergence.
4. Implement the semantics in Racket [1] (formerly PLT Scheme).

This approach is most effective if the target language can express measure-theoretic calculations and is similar to Racket in structure and semantics.

Our target language, λ_{ZFC}, is an untyped call-by-value lambda calculus extended with the von Neumann hierarchy of sets V as values and set operators as described in the ZFC axioms. Booleans, pairs, real numbers, and lambdas are encoded as pure sets. The equality operator "=" thus applies extensionally to all values (viewing lambdas as formulas). We assume that all infinite values are expressible. Extra primitive operators include pair indexing and arithmetic.

λ_{ZFC} contains all set-theoretic functions, or **mappings**. A lambda f can be made into a mapping by restriction to a domain $A \in V$:

$$f|_A = \{(x, f\ x) \mid x \in A\} = \textbf{\textit{image}}\ (\lambda x.\, (\textbf{\textit{pair}}\ x\ (f\ x)))\ A \tag{1}$$

where ***image***, a concurrent `map` on sets, is from the collection axiom schema. We often write mappings as $\lambda x \in A.e = (\lambda x.e)|_A$. The pair (f, A) is a *lazy* mapping.

By far, most λ_{ZFC} programs are uncomputable, but the goal of the exact semantics is to exactly express measure-theoretic calculations. The approximating semantics will be finitely representable and computable in Racket.

We write almost all our mathematics in λ_{ZFC}, but freely use syntactic sugar like *true* and *false*, numerals, infix and summation, set comprehensions, and pattern-matching definitions. In short, we write calculations in contemporary mathematics, but with untyped, first-class lambdas.

When we use types, $A \to B$ is a set of mappings, $A \Rightarrow B$ is a lambda or mapping type, and sets are membership propositions. We write common keywords in **bold** and invented keywords in ***bold italics***. Proofs are omitted for space.

2 Random Variables

Various accounts of probability allow practitioners to understand random variables in different ways, most commonly as free values with ambient probabilities. In measure-theoretic probability, a **random variable** X is a total mapping

$$X : \Omega \to S_X \tag{2}$$

Ω and S_X are sets called **sample spaces**, with elements called **outcomes**. Outcomes in S_X are ***observable***. Random variables must be **measurable**, which is analogous to continuity and always holds for discrete random variables.[1]

Example 1 (even/odd die outcome). Suppose we want to encode, as a random variable E, observing whether the outcome of a fair die roll is even or odd.

A complicated way is to define Ω as the states of the universe. Then $E : \Omega \to \{\texttt{even}, \texttt{odd}\}$ is a process that simulates the universe until the die is still, and recognizes the outcome. Hopefully, the probability that $E\ \omega = \texttt{even}$ is $\frac{1}{2}$.

A realistic way defines $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $E\ \omega = \texttt{even}$ if $\omega \in \{2, 4, 6\}$, otherwise `odd`. The probability that $E\ \omega = \texttt{even}$ is the sum of probabilities of the even Ωs, or $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

If we observe only evenness, we can define $\Omega = \{\texttt{even}, \texttt{odd}\}$, each with probability $\frac{1}{2}$, and $E\ \omega = \omega$. □

Random variables define and limit what is observable about any outcome $\omega \in \Omega$, which enables a kind of probabilistic abstraction. The example does it twice. The first makes calculating the probability that $E\ \omega = \texttt{even}$ tractable. The second is an optimization. In fact, redefining Ω, its random variables, and the probabilities of outcomes without changing the probabilities of *observable* outcomes is the essence of measure-theoretic optimization.

Defining random variables as functions not only enables optimization, but is a good factorization: it separates nondeterminism, observing nondeterminism, and assigning probabilities. It allows us to interpret expressions involving random variables without considering nondeterminism or probabilities at all.

2.1 Random Variables As Computations

When random variables are regarded as free variables, random variable expressions are no different from deterministic expressions. In measure-theoretic probability, as in vector arithmetic and analysis, operations are implicitly lifted to operate pointwise. For example, if A, B and C are random variables, $C = A + B$ means $C\ \omega = (A\ \omega) + (B\ \omega)$, and $B = 4 + A$ means $B\ \omega = 4 + (A\ \omega)$.

If we allow random variables to be lambdas, unnamed random variables are easy to express in λ_{ZFC}: $4 + A$ means $\lambda\omega.\,(+\ 4\ (A\ \omega))$. Lifting constants allows

[1] Measurability is generally regarded as weaker than continuity. Under very mild conditions, continuity implies measurability, but the converse is difficult to ensure.

$$\mathcal{R}[\![X]\!] = X \qquad \mathcal{R}[\![x]\!] = \textbf{\textit{pure}}\ x \qquad \mathcal{R}[\![v]\!] = \textbf{\textit{pure}}\ v$$
$$\mathcal{R}[\![(e_f\ e_1\ \ldots\ e_n)]\!] = \textbf{\textit{ap}}^*\ \mathcal{R}[\![e_f]\!]\ \mathcal{R}[\![e_1]\!]\ \ldots\ \mathcal{R}[\![e_n]\!]$$
$$\mathcal{R}\left[\!\left[\lambda x_1 \ldots x_n.e\right]\!\right] = \lambda\omega.\lambda x_1 \ldots x_n.\,(\mathcal{R}[\![e]\!]\ \omega)$$
$$\textbf{\textit{pure}}\ c = \lambda\omega.c, \quad \textbf{\textit{ap}}^*\ F\ X_1\ \ldots\ X_n = \lambda\omega.\,((F\ \omega)\ (X_1\ \omega)\ \ldots\ (X_n\ \omega))$$

Fig. 1. Random variable expression semantics. The source and target language are both λ_{ZFC}. Conditionals and primitive operators are trivial special cases of application.

expressions to be interpreted uniformly. For example, by interpreting + and 4 as $Plus = \lambda\omega.+$ and $Four = \lambda\omega.4$, the meaning of $4 + A$ is equivalent to

$$\lambda\omega.\,((Plus\ \omega)\ (Four\ \omega)\ (A\ \omega)) \tag{3}$$

The following combinators abstract lifting and application:

$$\begin{aligned} \textbf{\textit{pure}}\ c &= \lambda\omega.c \\ \textbf{\textit{ap}}^*\ F\ X_1\ \ldots\ X_n &= \lambda\omega.\,((F\ \omega)\ (X_1\ \omega)\ \ldots\ (X_n\ \omega)) \end{aligned} \tag{4}$$

so that $4 + A$ means $\textbf{\textit{ap}}^*\ (\textbf{\textit{pure}}\ +)\ (\textbf{\textit{pure}}\ 4)\ A = \cdots = \lambda\omega.\,(+\ 4\ (A\ \omega))$. These combinators define an **idiom** [12]. Idioms embed effectful computations with only a partial order. The ***random variable idiom*** is an instance of the **environment idiom** with idiom type constructor $I\ a = \Omega \Rightarrow a$ for some Ω.

$\mathcal{R}[\![\cdot]\!]$ (Fig. 1), the semantic function that interprets random variable expressions, targets this idiom. It does mechanically what we have done manually, and additionally interprets lambdas. For simplicity, it follows probability convention by assuming single uppercase letters are random variables. We assume syntactic sugar has been removed; e.g. application is in prefix form.

$\mathcal{R}[\![\cdot]\!]$ returns lambdas instead of mappings, but we can always recover a mapping if we need it. It may return lambdas that do not converge when applied. These lambdas do not represent random variables, which are total.

2.2 Implementation in Racket

Figure 2 shows `RV` and a snippet of `RV/kernel`, the macros that implement $\mathcal{R}[\![\cdot]\!]$. `RV` fully expands expressions into Racket's kernel language. This allows `RV/kernel` to transform any pure Racket expression into a random variable using Racket's new `syntax-parse` library [3]. `RV/kernel` raises a syntax error on `set!`, but there is no way to disallow applying functions that have effects.

Rather than differentiate between kinds of identifiers by case, `RV` takes a list of random variable identifiers as an additional argument. Other identifiers are wrapped with ***pure***, allowing arbitrary values in random variable expressions.

```
(define-syntax (RV/kernel stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     (syntax-parse #'e #:literal-sets (kernel-literals)
       [X:id  #:when (free-id-in? #'Xs #'X)  #'X]
       [x:id        #'(pure x)]
       [(quote c)   #'(pure (quote c))]
       [(%#plain-app e ...)  #'(ap* (RV/kernel Xs e) ...)]
       [....)]))]
(define-syntax (RV stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     #`(RV/kernel Xs #,(local-expand #'e 'expression empty))]))
```

Fig. 2. A well-embedded implementation of $\mathcal{R}[\![\cdot]\!]$.

3 Probability Distributions and Queries

In practice, functions called **distributions** assign probabilities or probability *densities* to observable outcomes. ***Primitive*** random variables have specified distributions. Distributions of ***derived*** random variables are calculated from their defining random variable expressions and the specified distributions.

In measure-theoretic probability, distributions are **probability measures**, which generalize assigning probabilities and densities by assigning probabilities to *sets* of outcomes. There are typically no special random variables: all distributions are calculated from one global probability measure.

3.1 Global Probability Measure and Queries

A probability measure's domain is a σ*-algebra* of a sample space, which is generally coarser than the powerset.[2] The σ-algebra of a countable sample space is usually its powerset; in that case, it has a **discrete** probability measure $\mathbb{P} : \mathcal{P}(S) \to [0, 1]$. A discrete probability measure is uniquely determined by its value on singletons, or by a **probability mass function** $P : S \to [0, 1]$.

Suppose we want to calculate, as in Example 1, the probability of an even die outcome. We do this by applying the global probability measure $\mathbb{P}$ to the correct subset of Ω. If $\Omega = \{1, 2, 3, 4, 5, 6\}$, then $P = [1, 2, 3, 4, 5, 6 \to \frac{1}{6}]$ uniquely determines $\mathbb{P}$. The probability that E outputs `even` is

$$\mathbb{P}\,\{\omega \in \Omega \mid E\ \omega = \texttt{even}\} = \mathbb{P}\,\{2, 4, 6\} = \sum_{\omega \in \{2,4,6\}} P\ \omega = \tfrac{1}{2} \tag{5}$$

This is a **probability query**. We could instead use a **distribution query** to find E's distribution $\mathbb{P}_E$, and apply it to $\{\texttt{even}\}$. Because $\mathbb{P}$ is discrete, so is $\mathbb{P}_E$;

[2] The σ-algebra requirement is imposed by ZFC set theory (see *Borel-Tarski paradox*).

therefore we only need E's probability mass function P_E:

$$P_E\ e = \sum_{\omega \in (E^{-1}\ \{e\})} P\ \omega, \quad P_E\ \texttt{even} = \sum_{\omega \in \{2,4,6\}} P\ \omega = \tfrac{1}{2} \tag{6}$$

As with ***pure*** and $\boldsymbol{ap}^*$ and $\mathcal{R}[\![\cdot]\!]$, it will be helpful to have some lambda abstractions before making making semantic functions for probability and distribution queries. The following converts probability mass functions to measures:

$$\boldsymbol{prob}\ P\ A = \sum_{\omega \in A} P\ \omega \tag{7}$$

So $\mathbb{P} = \boldsymbol{prob}\ P$. To calculate distributions like P_E, we define

$$\boldsymbol{dist}\ X\ (\Omega, P) = \lambda x \in S_X.\boldsymbol{prob}\ P\ (\boldsymbol{preimage}\ X\ \Omega\ \{x\}) \tag{8}$$

where $S_X = \boldsymbol{image}\ X\ \Omega$, and ***preimage*** calculates preimages of lambdas. We will need to thread Ω and P further on rather than regard them as free variables, so we have started now. The pair (Ω, P) is called a discrete **probability space**.[3]

3.2 Propositions As Predicates

When random variables are regarded as free variables, special notation $\mathrm{P}[\cdot]$ replaces applying $\mathbb{P}$ and sets turn into propositions. For example, a common way to write "the probability of an even die outcome" in practice is $\mathrm{P}[E = \texttt{even}]$.

We have a semantic function $\mathcal{R}[\![\cdot]\!]$ that turns propositions about random variables into predicates on Ω. The corresponding set is easy to calculate. For $E = \texttt{even}$, for example, it is $\boldsymbol{preimage}\ \mathcal{R}[\![E = \texttt{even}]\!]\ \Omega\ \{true\}$. In general,

$$\boldsymbol{prob}\ P\ (\boldsymbol{preimage}\ \mathcal{R}[\![e]\!]\ \Omega\ \{true\}) = \boldsymbol{dist}\ \mathcal{R}[\![e]\!]\ (\Omega, P)\ true \tag{9}$$

calculates $\mathrm{P}[e]$ when e is a proposition; i.e. when $\mathcal{R}[\![e]\!] : \Omega \Rightarrow \{true, false\}$.

Probability queries have common notation, but there seems to be no common notation that denotes distributions *per se*. The typical workarounds are to use $\mathrm{P}[\cdot]$ in implicit formulas like $\mathrm{P}[E = e]$ or to name the distributions with human-parseable names like P_E. Some theorists use $\mathcal{L}[\cdot]$, with $\mathcal{L}$ for *law*, an obscure synonym of *distribution*. We will define $\mathbf{D}[\![\cdot]\!]$ in place of $\mathcal{L}[\cdot]$.

Though we could define semantic functions $\mathbf{P}[\![\cdot]\!]$ and $\mathbf{D}[\![\cdot]\!]$ right now, we are putting them off until after interpreting the notation used for modeling.

3.3 Approximating Discrete Queries

Probabilities are real numbers. They remain real in the approximating semantics; we use floating point approximation and exact rationals in the implementation.

[3] For uncountable Ω, the probability space is $(\Omega, \Sigma, \mathbb{P})$, where Σ is a σ-algebra.

```
(define-struct mapping (domain proc)
  #:property prop:procedure (λ (f x) ((mapping-proc f) x)))
(define-struct fmapping (default hash)
  #:property prop:procedure
  (λ (f x) (hash-ref (fmapping-hash f) x (fmapping-default f))))

(define appx-z (make-parameter +inf.0))
(define (finitize ps)
  (match-let* ([(mapping Ω P)  ps]
               [Ωn  (cotake Ω (appx-z))]
               [qn  (apply + (map P Ωn))])
    (mapping Ωn (λ (ω) (/ (P ω) qn)))))

(define ((dist X) ps)
  (match-define (mapping Ω P) ps)
  (fmapping 0 (for/fold ([h  (hash)]) ([ω  (in-list Ω)])
                (hash-set h (X ω) (+ (P ω) (hash-ref h (X ω) 0))))))
```

Fig. 3. Implementation of finite approximation and distribution queries in Racket.

Arbitrary countable sets are not finitely representable. In the approximating semantics, Ω is restricted to recursively enumerable sets. The implementation represents them as lazy lists. We trust users to not create "sets" with duplicates.

prob P A calculates a countable sum. If A is a lazy list, it is easy to compute a converging series. But then approximate answers to distribution queries sum to values less than 1. To fix this, we instead approximate Ω and normalize P, which makes the sum finite and the distributions proper.

Suppose $(\omega_1, \omega_2, \ldots)$ is an enumeration of Ω. Let $z \in \mathbb{N}$ be the length of the prefix $\Omega_z = \{\omega_1, \ldots, \omega_z\}$ and $P_z\ \omega = (P\ \omega)/(\textbf{\textit{prob}}\ P\ \Omega_z)$. Then P_z converges to P by the arithmetic rules of limits, and so do distributions. We define ***finitize*** $(\Omega, P) = (\Omega_z, P_z)$ with $z \in \mathbb{N}$ as a free variable.

3.4 Implementation in Racket

Fig. 3 shows the implementations of ***finitize*** and ***dist*** in Racket. The free variable z appears as a *parameter* `appx-z`: a variable with static scope but dynamic extent. The `cotake` procedure returns the prefix of a lazy list as a finite list.

To implement ***dist***, we need to represent mappings in Racket. The applicable struct type `mapping` represents lazy mappings with possibly infinite domains. A `mapping` named `f` can be applied with `(f x)`. We do not ensure `x` is in the domain. (Checking is semidecidable, and nontermination is a terrible error message.) For distributions, checking is not important; the observable domain is.

However, we do not want `dist` to return lazy mappings. Doing so is inefficient: every application of the mapping would filter `Ω`. Further, `dist` will always receive a `finitize`d probability space. We therefore define `fmapping` for mappings that

are constant on all but a finite set. For these values, `dist` builds a hash table by computing the probabilities of all preimages in one pass through `Ω`.

We do use `mapping`, but for probability spaces and *specified* distributions.

4 Conditional Queries

For Bayesian practitioners, the most meaningful queries are **conditional** queries: those *conditioned on*, or *given*, some random variable's value. (For example, the probability an email is spam given it contains words like "madam," or the distribution over suspects given security footage.) A language without conditional queries is of little more use to them than a general-purpose language.

A review of measure-theoretic conditional probability is too involved to accurately summarize here. When $\mathbb{P}$ is discrete, however, the conditional probability of A given B (i.e. asserting that $\omega \in B$), simplifies to

$$\mathbb{P}[A \mid B] = (\mathbb{P}\ A \cap B)/(\mathbb{P}\ B) \tag{10}$$

The left-hand side of (10) is special notation, not an application. In practice, $\mathrm{P}[\cdot \mid \cdot]$ is also expressed using simpler queries: $\mathrm{P}[A \mid B] = \mathrm{P}[A \wedge B]/\mathrm{P}[B]$.

Example 2 (low/high die outcome). Extend Example 1 with random variable $L = \texttt{low}$ if $\omega \leq 3$, otherwise `high`. The probability E is `even` given L is `low` is

$$\mathrm{P}[E = \texttt{even} \mid L = \texttt{low}] = \frac{\mathrm{P}[E = \texttt{even} \wedge L = \texttt{low}]}{\mathrm{P}[L = \texttt{low}]} = \frac{\sum_{\omega \in \{2\}} P\ \omega}{\sum_{\omega \in \{1,2,3\}} P\ \omega} = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3} \tag{11}$$

as opposed to $\mathrm{P}[E = \texttt{even}] = \frac{1}{2}$, the *unconditional* probability. □

Conditional *distribution* queries determine how one random variable's value influences the distribution of another. Like unconditional distribution queries, there is no common notation for conditional distributions *per se*, and this is usually worked around the same way as before. For example, the distribution of E conditioned on L could be written $\mathrm{P}[E = e \mid L = l]$ or $P_{E|L}$.

It is tempting to define $\mathbf{P}[\![\cdot \mid \cdot]\!]$ in terms of $\mathbf{P}[\![\cdot]\!]$, and $\mathbf{D}[\![\cdot \mid \cdot]\!]$ in terms of $\mathbf{D}[\![\cdot]\!]$, as in (10). However, we get more flexibility defining conditioning as an operation on probability spaces instead of queries, and it better matches the unsimplified measure theory. The following abstraction returns a discrete probability space in which Ω is restricted to the subset where random variable Y returns y:

$$\begin{aligned} \textbf{\textit{cond}}\ Y\ y\ (\Omega, P) = (\Omega', P') \ \text{ where } \ & \Omega' = \textbf{\textit{preimage}}\ Y\ \Omega\ \{y\} \\ & P' = \lambda\omega \in \Omega'.(P\ \omega)/(\textbf{\textit{prob}}\ P\ \Omega') \end{aligned} \tag{12}$$

Then $\mathrm{P}[E = \texttt{even} \mid L = \texttt{low}] = \textbf{\textit{dist}}\ E\ (\textbf{\textit{cond}}\ L\ \texttt{low}\ (\Omega, P))\ \texttt{even}$.

We approximate ***cond*** by applying ***finitize*** to the probability space. Its implementation simply uses finite list procedures instead of set operators.

5 Conditional Probabilistic Theories

A probability space is a global mechanism by which random variables influence each other. When random variables are regarded as free variables, however, there is no way to refer to its properties. Instead, practitioners state facts about random variables, which constrain their distributions. Though they call such collections of statements *models*,[4] to us they are ***probabilistic theories***. A ***model*** is a probability space and random variables that imply the facts.

Bayesian practitioners tend to state conditional distributions directly. Their ***conditional theories*** generally guarantee that measure-theoretic models exist. We develop a semantic function $\mathcal{M}[\![\cdot]\!]$ that transforms conditional theories into models, and define $\mathbf{P}[\![\cdot]\!]$ and $\mathbf{D}[\![\cdot]\!]$ to calculate answers to queries about them.

5.1 Conditional Theories and the Product Model

Example 3 (die outcome conditional theory). Suppose we only need to know whether a die outcome is even or odd, high or low. L's distribution is the unconditional $P_L = [\texttt{low}, \texttt{high} \mapsto \frac{1}{2}]$, but E's distribution depends on L:

$$P_{E|L}\ l = \begin{cases} [\texttt{even} \mapsto \frac{1}{3}, \texttt{odd} \mapsto \frac{2}{3}] & \text{if } l = \texttt{low} \\ [\texttt{even} \mapsto \frac{2}{3}, \texttt{odd} \mapsto \frac{1}{3}] & \text{if } l = \texttt{high} \end{cases} \tag{13}$$

With P_L and $P_{E|L} : S_L \to S_E \to [0,1]$,[5] the theory is $L \sim P_L;\ E \sim P_{E|L}\ L$.[6] □

$L \sim P_L$ is a constraint on (Ω, P): for any model of the theory, L's distribution is P_L. Similarly, $E \sim P_{E|L}\ L$ means E's conditional distribution is $P_{E|L}$. We have been using the model $\Omega = \{1,2,3,4,5,6\}$, $P = [1,2,3,4,5,6 \mapsto \frac{1}{6}]$, with the obvious E and L. It is not hard to check that this is also a model:

$$\begin{aligned} &\Omega = \{\texttt{low}, \texttt{high}\} \times \{\texttt{even}, \texttt{odd}\},\ \ L\ (l,e) = l,\ \ E\ (l,e) = e \\ &P = [(\texttt{low}, \texttt{even}), (\texttt{high}, \texttt{odd}) \mapsto \tfrac{1}{6}, (\texttt{low}, \texttt{odd}), (\texttt{high}, \texttt{even}) \mapsto \tfrac{2}{6}] \end{aligned} \tag{14}$$

This construction of Ω, L and E clearly generalizes, but P is a little trickier. Fully justifying the generalization (including that it meets implicit independence assumptions that we have not mentioned) is rather tedious, so we will not do it here. But, for the present example, it is not hard to check that

$$\begin{aligned} &P\ \omega = (P_L\ (L\ \omega)) \times \big(P_{E|L}\ (L\ \omega)\ (E\ \omega)\big) \\ &\text{or}\ \ P = \mathcal{R}[\![(P_L\ L) \times \big(\big(P_{E|L}\ L\big)\ E\big)]\!] \end{aligned} \tag{15}$$

Let $K_L = \mathcal{R}[\![P_L]\!]$ and $K_E = \mathcal{R}[\![\big(P_{E|L}\ L\big)]\!]$. Then $P = \mathcal{R}[\![(K_L\ L) \times (K_E\ E)]\!]$. This is easy to generalize, but we need a technical restriction on theories first.

[4] In the colloquial sense, probably to emphasize their essential incompleteness.
[5] Usually, $P_{E|L} : S_E \times S_L \to [0,1]$. We reorder and curry to simplify interpretation.
[6] There are other popular ways to specify conditional distributions. We interpret only "$\sim$" notation because we can easily do so unambiguously.

Definition 1 (well-formed). *A conditional theory is* ***well-formed*** *when no e_j refers to X_i unless $j > i$ (roughly, no circular causes).*

Bayesian conditional theories can always be made well-formed.

Definition 2 (discrete product model). *Given a well-formed, discrete conditional theory $X_1 \sim e_1; \ldots; X_n \sim e_n$, let $K_i : \Omega \Rightarrow S_i \to [0,1]$, $K_i = \mathcal{R}[\![e_i]\!]$ for each $1 \leq i \leq n$. The* ***product model*** *of the theory is*

$$\Omega = \bigtimes_{i=1}^{n} S_i, \quad X_i\ \omega = \omega_i\ (1 \leq i \leq n), \quad P = \mathcal{R}\left[\!\!\left[\prod_{i=1}^{n} (K_i\ X_i)\right]\!\!\right] \tag{16}$$

Theorem 1 (correctness). *The discrete product model induces the correct conditional distributions and meets implicit independence assumptions.*

In their distribution statements, practitioners tend to apply first-order distributions to simple random variables. The discrete product model allows any λ_{ZFC} term e_i that, when interpreted, is a discrete **transition kernel** $K_i : \Omega \Rightarrow S_i \to [0,1]$. In measure theory, transition kernels are used to build **product spaces** such as (Ω, P).[7] $\mathcal{R}[\![\cdot]\!]$ links Bayesian practice to measure theory and represents an increase in expressive power in specifying distributions, by turning properly typed λ_{ZFC} terms into precisely what measure theory requires.[8]

5.2 Conditional Theories as Stateful Computations

Some conditional theories assert global conditions [11, 18]. Having conditioning *and* distribution statements requires a recursive interpretation. Well-formedness amounts to lexical scope, so we define it in terms of the state monad with probability-space-valued state, which we call the ***probability space monad***.

We assume the typical $\textbf{\textit{return}}_s$ and $\textbf{\textit{bind}}_s$ for the state monad. Fig. 4 shows additional lambdas $\textbf{\textit{cond}}_{ps}$, $\textbf{\textit{dist}}_{ps}$ and $\textbf{\textit{extend}}_{ps}$. $\mathcal{M}[\![\cdot]\!]$ expands into applications of these, wired up with $\textbf{\textit{bind}}_s$. The first two are straightforward reimplementations of ***cond*** and ***dist*** as stateful computations.

According to the product model, interpreting $X_i \sim e_i$ results in $\Omega_i = \Omega_{i-1} \times S_i$, with S_i extracted from $K_i : \Omega_{i-1} \Rightarrow S_i \to [0,1]$. A more precise type for K_i is the dependent type $(\omega : \Omega_{i-1}) \Rightarrow S'_i\ \omega \to [0,1]$, which reveals a complication. We cannot extract S_i, we can only extract the random variable $S'_i : \Omega_{i-1} \to \mathcal{P}(S_i)$.

Let $S'_i\ \omega = \textbf{\textit{domain}}\ (K_i\ \omega)$; then $S_i = \bigcup (\textbf{\textit{image}}\ S'_i\ \Omega_{i-1})$. But this makes query implementation inefficient: if the union has little overlap or is disjoint, P will assign 0 to most ω. In more general terms, we actually have a *dependent* Cartesian product $(\omega \in \Omega_{i-1}) \times (S'_i\ \omega)$, a generalization of the Cartesian product.[9] Calculating the Cartesian product instead leads to inefficiency.

[7] P in the discrete product model is a special case of Ionescu-Tulcea's product measure construction, which assumes pre-existing transition kernels.

[8] For uncountable spaces, the product model requires $\mathcal{R}[\![e_i]\!] : \Omega \Rightarrow \Sigma_i \to [0,1]$ where Σ_i is a σ-algebra. It is easy to ensure that every conditional distribution used in practice is defined in such a way that applying it in e_i yields this type.

[9] The dependent Cartesian product also generalizes disjoint union to arbitrary index sets. As such, it is usually called a *dependent sum* and denoted $\Sigma a : A.(B\ a)$.

$$
\begin{aligned}
&\boldsymbol{cond}_{ps}\ Y\ y\ (\Omega, P) = \big(\Omega', P', \_\big) \quad \text{where}\ \ \Omega' = \boldsymbol{preimage}\ Y\ \Omega\ \{y\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad P' = \lambda\omega \in \Omega'.(P\ \omega)/\big(\boldsymbol{prob}\ P\ \Omega'\big) \\
&\boldsymbol{dist}_{ps}\ X\ (\Omega, P) = (\Omega, P, P_X) \quad \text{where}\ \ S_X = \boldsymbol{image}\ X\ \Omega \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad P_X = \lambda x \in S_X.\boldsymbol{prob}\ P\ (\boldsymbol{preimage}\ X\ \Omega\ \{x\}) \\
&\boldsymbol{extend}_{ps}\ K_i\ (\Omega_{i-1}, P_{i-1}) = (\Omega_i, P_i, X_i) \\
&\quad \text{where}\ \ S'_i\ \omega = \boldsymbol{domain}\ (K_i\ \omega), \quad \Omega_i = (\omega \in \Omega_{i-1}) \times \big(S'_i\ \omega\big) \\
&\qquad\quad X_i\ \omega = \omega_j\ \ (\text{where } j = \text{length of any } \omega \in \Omega_{i-1}), \quad P_i = \mathcal{R}[\![P_{i-1} \times (K_i\ X_i)]\!]
\end{aligned}
$$

Fig. 4. Additional probability space monad functions used to interpret statements.

$$
\begin{aligned}
\mathcal{M}[\![X_i := e_i;\ t; \ldots]\!] &= \boldsymbol{bind}_s\ (\boldsymbol{return}_s\ \mathcal{R}[\![e_i]\!])\ \lambda X_i.\mathcal{M}[\![t; \ldots]\!] \\
\mathcal{M}[\![X_i \sim e_i;\ t; \ldots]\!] &= \boldsymbol{bind}_s\ (\boldsymbol{extend}_{ps}\ \mathcal{R}[\![e_i]\!])\ \lambda X_i.\mathcal{M}[\![t; \ldots]\!] \\
\mathcal{M}[\![e_a = e_b;\ t; \ldots]\!] &= \boldsymbol{bind}_s\ (\boldsymbol{cond}_{ps}\ \mathcal{R}[\![e_a]\!]\ \mathcal{R}[\![e_b]\!])\ \lambda\_.\mathcal{M}[\![t; \ldots]\!] \\
\mathcal{M}[\![\epsilon]\!] &= \boldsymbol{return}_s\ (X_1, \ldots, X_n) \\
\\
\mathbf{D}[\![e]\!]\ m &= \boldsymbol{run}_{ps}\ \big(\boldsymbol{bind}_s\ m\ \lambda(X_1, \ldots, X_n).\boldsymbol{dist}_{ps}\ \mathcal{R}[\![e]\!]\big) \\
\mathbf{D}[\![e_X \mid e_Y]\!]\ m &= \lambda y.\mathbf{D}[\![e_X]\!]\ \big(\boldsymbol{bind}_s\ m\ \lambda(X_1, \ldots, X_n).\mathcal{M}[\![e_Y = y]\!]\big) \\
\mathbf{P}[\![e]\!]\ m &= \mathbf{D}[\![e]\!]\ m\ \mathit{true}, \quad \mathbf{P}[\![e_A \mid e_B]\!]\ m = \mathbf{D}[\![e_A \mid e_B]\!]\ m\ \mathit{true}\ \mathit{true}
\end{aligned}
$$

Fig. 5. The conditional theory and query semantic functions.

Dependent Cartesian products are elegantly calculated using the set monad:

$$
\begin{aligned}
\boldsymbol{return}_v\ x &= \{x\}, \quad \boldsymbol{bind}_v\ m\ f = \bigcup(\boldsymbol{image}\ f\ m) \\
(a \in A) \times (B\ a) &= \boldsymbol{bind}_v\ A\ \lambda a.\ \big(\boldsymbol{bind}_v\ (B\ a)\ \lambda b.\ (\boldsymbol{return}_v\ (a, b))\big)
\end{aligned}
\tag{17}
$$

5.3 Semantic Functions

Fig. 5 defines $\mathcal{M}[\![\cdot]\!]$, which interprets conditional theories containing definition, distribution, and conditioning statements as probability space monad computations. After the statements are exhausted, it returns the random variables. Returning their names as well is an obfuscating complication, which we avoid by implicitly extracting them from the theory before interpretation. (Extracting and returning names is explicit in the implementation, however.)

$\mathbf{D}[\![e]\!]$ expands to a distribution-valued computation and runs it using

$$
\boldsymbol{run}_{ps}\ m = x \quad \text{where} \quad (\Omega, P, x) = m\ (\Omega_0, P_0),\ \ (\Omega_0, P_0) = \big(\{()\}, \lambda\omega.1\big) \tag{18}
$$

We call (Ω_0, P_0) the ***empty probability space***. $\mathbf{D}[\![e_X \mid e_Y]\!]$ conditions and hands off to $\mathbf{D}[\![e_X]\!]$. $\mathbf{P}[\![\cdot]\!]$ is defined in terms of $\mathbf{D}[\![\cdot]\!]$.

5.4 Approximation

$(a \in A) \times (B\ a)$ with recursively enumerable sets can be computed using lazy lists in a way similar to enumerating $\mathbb{N} \times \mathbb{N}$. (It cannot be done with a monad as in the exact semantics, but we do not need it to.) The approximating versions of $\boldsymbol{dist}_{ps}$ and $\boldsymbol{cond}_{ps}$ apply ***finitize*** to the probability space.

5.5 Racket Implementation

$\mathcal{M}[\![\cdot]\!]$'s implementation is `MDL`. Like `RV`, it passes random variable identifiers, but it accumulates them. For example, `(MDL [] ([X ~ Px]))` expands to

```
([X] (bind/s (extend/ps (RV [] Px)) (λ (X) (ret/s (list X)))))
```

which is a list of random variable identifiers and a model computation.

We store theories in transformer bindings so queries can expand them later. For example, `(define-model die-roll [L ~ Pl] [E ~ (Pe/l L)])` expands to

```
(define-syntax die-roll #'(MDL [] ([L ~ Pl] [E ~ (Pe/l L)])))
```

The macro `with-model` introduces a scope in which a theory's variables are visible: `(with-model die-roll (Dist L E))`, for example, looks up `die-roll` and expands it into its identifiers and computation. Using the identifiers as lambda arguments, `Dist` (the implementation of $\mathbf{D}[\![\cdot]\!]$) builds a query computation as in Fig. 5, and runs it with the empty probability space `(mapping (list empty) (λ (ω) 1))`.

Using these identifiers would break hygiene, except that `Dist` replaces the lambda arguments' lexical context. This puts the theory's exported identifiers in scope, even when the theory and query are defined in separate modules. It is safe, in the sense that queries can access only the exported identifiers.

Aside from passing identifiers and monkeying with hygiene, the macros are almost transcribed from the semantic functions.

Examples. The Geometric conditional distribution has countable domain:

```
(define (Geometric p)
  (mapping N1 (λ (k) (* p (expt (- 1 p) (- k 1))))))
```

where `N1` is a lazy list of natural numbers starting at `1`. Nahin [14] gives an interesting use for this distribution in his book of probability puzzlers.

Two idiots duel with one gun. They put only one bullet in it, and take turns spinning the chamber and firing at each other. They know that if they each take one shot at a time, player one usually wins. Therefore, player one takes one shot, and after that, the next player takes one more shot than the previous player, spinning the chamber after each. How probable is player two's demise?

The distribution over the number of shots when the gun fires is `(Geometric 1/6)`. This procedure determines if the first player wins if the gun fires at shot `n`:

```
(define (p1-wins? n [shots 1])
  (cond [(n . <= . 0)  #f]
        [else  (not (p1-wins? (- n shots) (add1 shots)))]))
```

The probability that the first player wins is therefore approximately

```
(with-model (model [N ∼ (Geometric 1/6)])
  (Pr (p1-wins? N)))
```

Nahin computes 0.523919127555099524791984 3—25 decimal digits—with custom MATLAB code. At `appx-z` $\geq$ `321`, our solution affixes 9 to the same digits. Implementing it took us about five minutes—but the problem is not Bayesian.

This is: suppose player one slyly suggests a coin flip to decide whether they spin the chamber before firing. You do not see the duel, but find out that player two won. What is the probability they spun the chamber? A good theory is

```
(define-model half-idiot-duel
  [S ∼ (Bernoulli 1/2)]
  [N ∼ (cond [S     (Geometric 1/6)]
             [else  (UniformInt 1 6)])])
```

With a model of this, `(Pr S (not (p1-wins? N)))` converges to about `0.588`.

`N`'s conditional distribution is easy to specify using `cond`. We can do it because `RV` operates on Racket's kernel language, and because $\mathcal{M}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]$ allow expressive specifications. The sample space is built using a dependent Cartesian product for efficiency. These features support the Bayesian style of specifying models as directly as possible. (The most direct specification of this problem has infinitely many statements. Supporting theories like that is for future work.)

6 Commutativity and Equivalence in Distribution

Whether queries should be allowed inside theories is a decision with subtle effects.

Theories are sets of facts. Well-formedness imposes a partial order, but every linearization should be interpreted equivalently. Thus, we can determine whether two kinds of statements can coexist in theories by determining whether they can be exchanged without changing the interpretation. This is equivalent to determining whether the corresponding monad functions commute.

The following definitions suppose a conditional theory $t_1; \ldots; t_n$ in which exchanging some t_i and t_{i+1} (where $i < n$) is well-formed. Applying semantic functions in the definitions yields definitions that are independent of syntax but difficult to read, so we give the syntactic versions.

Definition 3 (commutativity). *We say that t_i and t_{i+1}* ***commute*** *when* $\mathcal{M}[\![t_1; \ldots; t_i; t_{i+1}; \ldots; t_n]\!]\ (\Omega_0, P_0) = \mathcal{M}[\![t_1; \ldots; t_{i+1}; t_i; \ldots; t_n]\!]\ (\Omega_0, P_0)$.

This notion of commutativity is too strong: distribution statements would never commute with each other. We need a weaker test than equality.

Definition 4 (equivalence in distribution). *Suppose $X_1, \ldots, X_m$ are defined in $t_1, \ldots, t_n$. Let $m = \mathcal{M}[\![t_1, \ldots, t_n]\!]$, and m' be a (usually different) probability space monad computation. We write $m \equiv_{\mathbf{D}} m'$ and call m and m'* ***equivalent in distribution*** *when* $\mathbf{D}[\![X_1, \ldots, X_m]\!]\ m = \mathbf{D}[\![X_1, \ldots, X_m]\!]\ m'$.

The following says $\equiv_{\mathbf{D}}$ is like observational equivalence with query contexts:

Theorem 2 (context). $\mathbf{D}[\![e_X \mid e_Y]\!]\ m = \mathbf{D}[\![e_X \mid e_Y]\!]\ m'$ *for all random variables* $\mathcal{R}[\![e_X]\!]$ *and* $\mathcal{R}[\![e_Y]\!]$ *if and only if* $m \equiv_{\mathbf{D}} m'$.

Definition 5 (commutativity in distribution). *We say* t_i *and* t_{i+1} *commute* ***in distribution*** *when* $\mathcal{M}[\![t_1; \ldots; t_i; t_{i+1}; \ldots; t_n]\!] \equiv_{\mathbf{D}} \mathcal{M}[\![t_1; \ldots; t_{i+1}; t_i; \ldots; t_n]\!]$.

Theorem 3. *The following table summarizes commutativity of* $\boldsymbol{cond}_{ps}$, $\boldsymbol{dist}_{ps}$ *and* $\boldsymbol{extend}_{ps}$ *in the probability space monad:*

$\boldsymbol{cond}_{ps}$	$=$		
$\boldsymbol{extend}_{ps}$	$=$	$\equiv_{\mathbf{D}}$	
$\boldsymbol{dist}_{ps}$	$\not\equiv_{\mathbf{D}}$	$=$	$=$
	$\boldsymbol{cond}_{ps}$	$\boldsymbol{extend}_{ps}$	$\boldsymbol{dist}_{ps}$

By Thm. 3, if we are to maintain the idea that theories are sets of facts with an optional partial order imposed by well-formedness, conditioning and query statements cannot both be allowed in the same theory.

7 Related Work

Our approach to semantics is similar to abstract interpretation: we have a concrete (exact) semantics and a family of abstractions parameterized by z (approximating semantics). We have not framed our approach this way because our approximations are not conservative, and would be difficult to formulate as abstractions when parameterized on a random source (which we intend to do).

Bayesian practitioners occasionally create languages for modeling and queries. It is usually difficult to analyze their properties, as they tend to be defined by implementations. Almost all of them compute converging approximations and support conditional queries. When they work as expected, they are useful.

Koller and Pfeffer [8] efficiently compute exact distributions for the outputs of programs in a Scheme-like language. BUGS [10] focuses on efficient approximate computation for probabilistic theories with a finite number of primitive random variables, with distributions that practitioners typically use. BLOG [13] exists specifically to allow stating distributions of infinite vectors of primitive random variables. BLAISE [2] allows stating both distribution and approximation method for each random variable. Church [4] is a Scheme-like probabilistic language with approximate inference, and focuses on expressiveness.

Kiselyov [7] embeds a probabilistic language in O'Caml for efficient computation. It uses continuations to enumerate or sample random variable values, and has a `fail` construct for the *complement* of conditioning. The sampler looks ahead for `fail` and can handle it efficiently. This may be justified by commutativity (Thm. 3), depending on interaction with other language features.

There is a fair amount of semantics work in probabilistic languages. Most of it is not motivated by Bayesian concerns, and thus does not define conditioning. Kozen [9] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [5] proves

properties about programs with binary random choice by encoding programs and portions of measure theory in HOL.

Jones [6] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete incarnation is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [17] define the probability monad measure-theoretically and implement a language for finite probability. We do not build on this work because the probability monad does not build a probability space, making it difficult to reason about conditioning.

Pfeffer also develops IBAL [16], apparently the only lambda calculus with probabilistic choice that also defines conditional queries. Park [15] extends a lambda calculus with probabilistic choice, defining it for a very general class of probability measures using inverse transform sampling.

8 Conclusions and Future Work

For discrete Bayesian theories, we explained a large subclass of notation as measure-theoretic calculations by transformation into λ_{ZFC}. There is now at least one precisely defined set of expressions that denote discrete conditional distributions in conditional theories, and it is very large and expressive. We gave a converging approximating semantics and implemented it in Racket.

We could have interpreted notation as first-order logic and ZFC, in which measure theory is developed. Defining the exact semantics compositionally would have been excruciating, and deriving an implementation from the semantics would have involved much hand-waving. With our approach, the path from notation to exact meaning to approximation to implementation is clear.

Now that we are satisfied that it works, we turn our attention to uncountable sample spaces and infinitely many primitive random variables.

We expect that targeting λ_{ZFC} and following measure-theoretic structure in our preliminary work will make the transition to uncountable spaces fairly smooth. The overall structure of the exact calculations will not change, but some details will. The random variable idiom will be identical, but will require measurability proofs. We will still use the state monad, but the state will be general probability spaces instead of discrete probability spaces. We will use regular conditional probability in $\boldsymbol{cond}_{ps}$, $\boldsymbol{extend}_{ps}$ will calculate product σ-algebras and transition kernel products, and $\boldsymbol{dist}_{ps}$ will return probability measures. We will not need to change $\mathcal{R}[\![\cdot]\!]$, $\mathbf{D}[\![\cdot]\!]$ or $\mathbf{P}[\![\cdot]\!]$. Many approximations are available. We will likely choose a sampling method that is reliable and easy to parallelize.

The most general constructive way to specify theories with infinitely many primitive random variables is with recursive abstractions, but it is not clear what kind of abstraction we need. Lambdas are suitable for most functional programming, in which it is usually good that intermediate values are unobservable. However, they do not meet Bayesian needs: practitioners define theories to study them, not to obtain single answers. If lambdas were the only abstraction, returning every intermediate value from every lambda would become *good*

practice. Because we do not know what form abstraction will take, we will likely develop it independently by allowing theories with infinitely many statements.

Model equivalence in distribution extends readily to uncountable spaces. It defines a standard by which to carry out measure-theoretic optimizations, which can only be done in the exact semantics. Examples are variable collapse, a probabilistic analogue of constant folding that can increase efficiency by an order of magnitude, and a probabilistic analogue of constraint propagation to speed up conditional queries.

References

1. The Racket programming language. http://racket-lang.org
2. Bonawitz, K.A.: Composable Probabilistic Inference with Blaise. Ph.D. thesis, Massachusetts Institute of Technology (2008)
3. Culpepper, R.: Refining Syntactic Sugar: Tools for Supporting Macro Development. Ph.D. thesis, Northeastern University (2010), to Appear
4. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: Uncertainty in Artificial Intelligence (2008)
5. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
6. Jones, C.: Probabilistic Non-Determinism. Ph.D. thesis, University of Edinburgh (1990)
7. Kiselyov, O., Shan, C.: Monolingual probabilistic programming using generalized coroutines. In: Uncertainty in Artificial Intelligence (2008)
8. Koller, D., McAllester, D., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: 14th National Conference on Artificial Intelligence. pp. 740–747 (August 1997)
9. Kozen, D.: Semantics of probabilistic programs. In: Foundations of Computer Science. pp. 101–114 (1979)
10. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS – a Bayesian modelling framework: Concepts, structure, and extensibility. Statistics and Computing 10(4), 325–337 (2000)
11. Mateescu, R., Dechter, R.: Mixed deterministic and probabilistic networks. Annals of Mathematics and Artificial Intelligence (2008)
12. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1) (2008)
13. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: International Joint Conference on Artificial Intelligence. pp. 1352–1359 (2005)
14. Nahin, P.J.: Duelling Idiots and Other Probability Puzzlers. Princeton University Press (2000)
15. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. Transactions on Programming Languages and Systems 31(1) (2008)
16. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Statistical Relational Learning. MIT Press (2007)
17. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages. pp. 154–165 (2002)
18. Toronto, N., Morse, B.S., Seppi, K., Ventura, D.: Super-resolution via recapture and Bayesian effect modeling. In: Computer Vision and Pattern Recognition (2009)

On the relation of call-by-need and call-by-name in a natural semantics setting

Lidia Sánchez-Gil
Sistemas Informáticos y Computación
Universidad Complutense de Madrid
lidiasg@mat.ucm.es

Mercedes Hidalgo-Herrero
Didáctica de las Matemáticas
Universidad Complutense de Madrid
mhidalgo@edu.ucm.es

Yolanda Ortega-Mallén
Sistemas Informáticos y Computación
Universidad Complutense de Madrid
yolanda@sip.ucm.es

Abstract

We study the relation between Launchbury's natural semantics for lazy evaluation and an alternative call-by-name version, both given for a λ-calculus extended with recursive lets. This relation considers not only the final values computed by each semantics when evaluating the same expression in the same context, but also the obtained final heaps are compared.

1 Motivation

Call-by-name evaluation, or more precisely its more efficient implementation *call-by-need* —which avoids repeated computations— is the semantic foundation for non-strict —or so-called "lazy"— functional programming languages like Haskell or Clean.

An important number of works comparing the different evaluation strategies for the λ-calculus have been published. Among the most influential is the one by Plotkin[10], which studies in detail the relation between *call-by-value* and *call-by-name* by means of simulations between two languages, and through the prism of a contextual equivalence, where two terms are considered equivalent whenever applied in any context, the first term converges if and only if the second converges too. Clearly, *call-by-value* and *call-by-name* are essentially different, because for some terms the latter is able to produce a value in contexts where the former is lost in unending computations. However, the relation between *call-by-need* and *call-by-name* is more subtle and it has been analyzed, among other works, in [7] and [2], where observational equivalences based on reducibility to weak-head-normal-form are given, but only for λ-calculi with non recursive `let`-constructions, leaving unresolved the equivalence for calculi extended with `letrec`. More recently, in [11], a contextual equivalence between a *call-by-need* letrec-calculus and a *call-by-name* one has been presented.

However, we are not interested in the equivalence of the equational theories for *call-by-need* and *call-by-name*, but in a more specific problem. Launchbury defines in [6] a natural semantics for lazy evaluation (*call-by-need*) where the set of bindings is explicitly managed to make possible their sharing. In order to prove that this lazy semantics is correct and computationally adequate —with respect to a standard denotational semantics— Launchbury introduces some variations in his natural semantics so that it becomes a *call-by-name* semantics where the bindings are not longer updated with the values obtained; moreover, in the alternative semantics applications are achieved by introducing indirections instead of by performing the β-reduction through substitution. Unfortunately, the proof of the equivalence between these two semantics is nowhere detailed, and a simple induction turns out to be insufficient, while some problems arise that are interesting in themselves. In the present work we investigate these problems and show how to achieve a proof of the equivalence between these two natural semantics.

In Section 2, we present the letrec-calculus and the two natural semantics described in [6]. We clarify the technical differences between both semantics, and we introduce two intermediate semantics that help us to deal with the equivalence problem. In order to avoid α-conversion and name clashing, in Section 3 we use a nameless representation of our calculus. We give a brief introduction to the de Bruijn notation, and the syntax and the semantics given in Section 2 are translated to this notation. Section 4 is devoted to the analysis of the similarities and differences between the derivations obtained with the distinct semantics. We explain how to cope with the equivalence problem and we define some relations between the heaps and values obtained with the different evaluation rules. In the last section we draw the conclusions and give hints of our future work on this subject.

2 Lazy natural semantics

In this section we explain the natural semantics defined by Launchbury in [6] for lazy evaluation. We also present the alternative rules for application and variables that transform the lazy semantics in a call-by-name semantics. In order to facilitate the comparison of these semantics, we focus individually in each change and introduce two intermediate semantics.

2.1 Natural semantics

The language described in [6] is a normalised lambda calculus extended with recursive *lets*. The restricted abstract syntax is

$$\begin{array}{lcll} x & \in & \mathit{Var} & \\ e & \in & \mathit{Exp} & ::= \quad \lambda x.e \\ & & & \mid \quad (e\ x) \\ & & & \mid \quad x \\ & & & \mid \quad \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e. \end{array}$$

Normalization is achieved in two steps. First an α-conversion is performed, i.e. bound variables are renamed with completely fresh names. In a second phase, it is assured that arguments in functional applications are always variables.

Launchbury defines a natural semantics for this language [6]. This natural semantics follows a call-by-need strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : z$, that is, the term e in the context of the heap Γ reduces to the value z together with the (modified) heap Δ. *Values* are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions of variables into expressions. Each pair (variable, expression) is called a *binding*, and it is represented as $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings that bind variables to unevaluated terms may be updated to their computed value. The rules of this semantics are shown in Figure 1, where $\hat{z}$ represents an α-conversion of the value z.

2.2 Denotational semantics

To define a denotational semantics of the calculus described above, a domain of values and environments to associate values to variables are needed.

An *environment* is a function mapping variables into values,

$$\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Value},$$

where *Value* is some appropriate domain containing at least a lifted version of its own function space, i.e.,

$$z \in \mathit{Value} = [\mathit{Value} \to \mathit{Value}]_{\perp}.$$

Notice that *Value* corresponds to the standard lifted function domain D described in [1].

An ordering is defined on environments, such that if ρ is *less or equal* than ρ' (denoted as $\rho \leq \rho'$) then ρ' may bind more variables than ρ, but otherwise is equal to ρ, i.e. $\forall x.\rho(x) \neq \perp \Rightarrow \rho(x) = \rho'(x)$.

Launchbury defines a denotational semantics of the language described by the syntax given in Subsection 2.1 in [6]. The semantic function is $[\![—]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Value}$, whose definition is as follows:

$$\begin{array}{l} [\![\lambda x.e]\!]_\rho = Fn(\lambda\nu.[\![e]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\ [\![(e\ x)]\!]_\rho = ([\![e]\!]_\rho)\!\downarrow_{Fn} ([\![x]\!]_\rho) \\ [\![x]\!]_\rho = \rho(x) \\ [\![\texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e]\!]_\rho = [\![e]\!]_{\{\!\{(x_i \mapsto e_i)_{i=1}^{n}\}\!\}\rho}. \end{array}$$

$$\textsf{Lambda} \qquad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e$$

$$\textsf{Application} \qquad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : z}{\Gamma : (e\ x) \Downarrow \Delta : z}$$

$$\textsf{Variable} \qquad \frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}}$$

$$\textsf{Let} \qquad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : z}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : z}$$

Figure 1: Natural Semantics

The function $\{\!\{-\}\!\} : \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$ should be considered as an environment modifier which is defined as follows:

$$\{\!\{(x_i \mapsto e_i)_{i=1}^n\}\!\}\rho = \mu\rho'.(\rho \sqcup (x_i \mapsto [\![e_i]\!]_{\rho'})_{i=1}^n),$$

where μ stands for the least fixed point operator and $\sqcup$ for

$$(\rho \sqcup (x \mapsto [\![e]\!]_{\rho'}))\ y\ =\ \begin{cases} \rho(y) & \text{if } y \neq x \\ [\![e]\!]_{\rho'} & \text{if } y = x \end{cases}$$

This definition only makes sense on environments ρ that are *consistent* with the heap $(x_i \mapsto e_i)_{i=1}^n$ (i.e. if the environment and the heap bind the same variables, then they are bound to values for which an upper bound exists). The consistency between environments and heaps in the denotational semantics definition given above is ensured thanks to the normalisation of the terms, that requires that all bound variables are distinct.

The correctness of the natural semantics (Figure 1) with respect to the denotational semantics has been proved in [6].

Theorem 1 (Correctness of the natural semantics) *Let $\Gamma, \Delta \in \mathit{Heap}$, $e \in \mathit{Exp}$ and $z \in \mathit{Val}$. If $\Gamma : e \Downarrow \Delta : z$ then for any environment $\rho \in \mathit{Env}$ consistent with Γ, $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$ and $\{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$.*

2.3 Alternative natural semantics

In order to prove the computational adequacy of the natural semantics (Subsection 2.1) with respect to the standard denotational semantics (Subsection 2.2), Launchbury introduces the alternative rules for Application and Variable shown in Figure 2.

From now on we will use $\Downarrow^A$ for derivations using these alternative rules.

The effect of the new application rule is to add indirections instead of performing a β-reduction. In this way the heaps are more closely related to the environments of the denotational semantics. The side-effect is an increase in the number of bindings in the heap.

Example 1 *We show in this example how the heaps obtained by the alternative natural semantics are more "alike" to the environments of the denotational semantics than those calculated with the original lazy semantics.*

We evaluate the expression $e \equiv \texttt{let}\ \{x_1 = \lambda t_1.t_1, x_2 = (x_1\ x_1)\}\ \texttt{in}\ (x_2\ x_1)$ in the context of the empty heap with both operational semantics and the following heap/term pairs are obtained:

$$\{\,\} : e \Downarrow \{x_1 \mapsto \lambda t_1.t_1, x_2 \mapsto \lambda t_3.t_3\} : \lambda t_4.t_4$$
$$\{\,\} : e \Downarrow^A \{t_5 \mapsto x_1, t_2 \mapsto x_1, x_1 \mapsto \lambda t_1.t_1, x_2 \mapsto x_1\ x_1\} : \lambda t_4.t_4$$

Now, we calculate the denotation of e in the initial undefined environment $\rho_\perp$ which maps every variable to $\perp$:

$$[\![e]\!]_{\rho_\perp} = \mathit{Fn}(\lambda\nu.[\![t_4]\!]_{\rho' \sqcup \{t_4 \mapsto \nu\}})$$

where $\rho' = \mu\rho'.(\rho_\perp \sqcup \{t_5 \mapsto [\![x_1]\!]_{\rho'}, t_2 \mapsto [\![x_1]\!]_{\rho'}, x_1 \mapsto [\![\lambda t_1.t_1]\!]_{\rho'}, x_2 \mapsto [\![(x_1\ x_1)]\!]_{\rho'}\})$. Notice the exact correspondence of the bindings in the final heap for $\Downarrow^A$ with the variables defined in the environment ρ'.

$$\textsf{AApplication} \qquad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad (\Theta, y \mapsto x) : e' \Downarrow \Delta : z}{\Gamma : e\ x \Downarrow \Delta : z}$$

$$\textsf{AVariable} \qquad \frac{(\Gamma, x \mapsto e) : \hat{e} \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow \Delta : z}$$

Figure 2: Alternative Natural Semantics

The AVariable rule removes updating from the semantics, hence there is no longer the possibility of detecting black holes produced by self-references. Notice that with the original natural semantics a proof of a judgement may fail in one of two ways: either there is no *finite* derivation for the judgement (infinite loop), or there is no applicable rule in a subderivation (*black hole*). Denotationally both cases correspond to the undefined value $\bot$.

In the following, we will refer to the original (call-by-need) natural semantics —rules in Figure 1— as the NS, and to the alternative (call-by-name) semantics —with the alternative rules in Figure 2— as the ANS.

We have proved that the ANS is correct with respect to the denotational semantics given in Subsection 2.2.

Theorem 2 (Correctness of the ANS) *Let $\Gamma, \Delta \in Heap$, $e \in Exp$ and $z \in Val$. If $\Gamma : e \Downarrow^A \Delta : z$ then for any environment $\rho \in Env$ consistent with Γ, $[\![e]\!]_{\{\!|\Gamma|\!\}\rho} = [\![z]\!]_{\{\!|\Delta|\!\}\rho}$ and $\{\!|\Gamma|\!\}\rho \leq \{\!|\Delta|\!\}\rho$.*

Proof. By induction on the derivation of the judgement. We need to check only the new rules in Figure 2.

AAplication. Let ρ be an environment consistent with Γ.

$$\begin{aligned}
[\![(e\ x)]\!]_{\{\!|\Gamma|\!\}\rho} &= ([\![e]\!]_{\{\!|\Gamma|\!\}\rho})\downarrow_{Fn} ([\![x]\!]_{\{\!|\Gamma|\!\}\rho}) \\
&\overset{I.H.}{=} ([\![\lambda y.e']\!]_{\{\!|\Theta|\!\}\rho})\downarrow_{Fn} ([\![x]\!]_{\{\!|\Theta|\!\}\rho}) \\
&= (\lambda\nu.[\![e']\!]_{\{\!|\Theta|\!\}\rho \sqcup \{y \mapsto \nu\}})([\![x]\!]_{\{\!|\Theta|\!\}\rho}) \\
&= [\![e']\!]_{\{\!|\Theta|\!\}\rho \sqcup \{y \mapsto [\![x]\!]_{\{\!|\Theta|\!\}\rho}\}}) \\
&= [\![e']\!]_{\{\!|\Theta, y \mapsto x|\!\}\rho} \\
&\overset{I.H.}{=} [\![z]\!]_{\{\!|\Delta|\!\}\rho}
\end{aligned}$$

By the transitivity of the *less or equal* relation on heaps:

$$\{\!|\Gamma|\!\}\rho \overset{I.H.}{\leq} \{\!|\Theta|\!\}\rho \leq \{\!|\Theta, y \mapsto x|\!\}\rho \overset{I.H.}{\leq} \{\!|\Delta|\!\}\rho.$$

AVariable. Let ρ be an environment consistent with $(\Gamma, x \mapsto e)$.

$$\begin{aligned}
[\![x]\!]_{\{\!|\Gamma, x \mapsto e|\!\}\rho} &= \{\!|\Gamma, x \mapsto e|\!\}\rho\ (x) \\
&= [\![e]\!]_{\{\!|\Gamma, x \mapsto e|\!\}\rho} \\
&= [\![\hat{e}]\!]_{\{\!|\Gamma, x \mapsto e|\!\}\rho} \\
&\overset{I.H.}{=} [\![z]\!]_{\{\!|\Delta|\!\}\rho}
\end{aligned}$$

And $\{\!|\Gamma, x \mapsto e|\!\}\rho \leq \{\!|\Delta|\!\}\rho$ by induction hipothesis. □

Theorems 1 and 2 indicate that the NS and the ANS are "denotationally" equivalent in the sense that if they are able to reduce an expression (in some heap context) then the values obtained have the same denotation. But this is insufficient for our purposes, because we need to assure that if for some heap/term pair a reduction exists in one of the natural semantics, then there must exist a reduction in the other too. Next section is devoted to this question.

2.4 Two intermediate semantics

At first sight, the changes introduced by the ANS do not seem to involve serious difficulties to prove its equivalence with the NS in the sense that any derivation from a heap/term pair with the NS implies some derivation with the ANS, such that the final heap/value pairs are somehow "equivalent", and viceversa. Unfortunately things are not so easy. On the one hand, the alternative rule for variables transforms the original call-by-need/lazy semantics into a call-by-name semantics because computed values are not longer shared. On the other hand, the addition of indirections complicates considerably the task of comparing the heap/value pairs obtained by each semantics.

We deal with the problems arisen from these modifications by introducing two intermediate semantics which correspond to the insertion of

just one of the modifications into the NS.

The rules of the *Indirected Natural Semantics* (INS) are the same as those of the NS (Figure 1) except for the application rule, that corresponds to the one in the alternative version, i.e. AApplication in Figure 2. Analogously, the rules of the *No-update Natural Semantics* (NNS) are those of the NS but for the variable rule, that corresponds to the alternative AVariable rule in Figure 2. In the following, we will use $\Downarrow^I$ in the derivations of the INS and $\Downarrow^N$ for those of the NNS.

Figure 3 resumes the characteristics of the four natural semantics introduced so far.

The correctness of these intermediate semantics with respect to the denotational semantics of Subsection 2.2 can be easily proved by combining the proofs of Theorems 1 and 2.

Theorem 3 (Correctness of the INS and the NNS) *Let* $\Gamma, \Delta \in Heap$, $e \in Exp$, *and* $z \in Val$. *If* $\Gamma : e \Downarrow^K \Delta : z$ *for* $K \in \{I, N\}$ *then, for any environment* $\rho \in Env$ *consistent with* Γ, $[\![e]\!]_{\{\!|\Gamma|\!\}\rho} = [\![z]\!]_{\{\!|\Delta|\!\}\rho}$ *and* $\{\!|\Gamma|\!\}\rho \leq \{\!|\Delta|\!\}\rho$.

3 A nameless representation

The rules Variable and AVariable imply α-conversion of terms, and thus they introduce fresh names. This makes more difficult the task of comparing the heaps and values obtained with the different natural semantics. Therefore, in order to simplify proofs, we have decided to use a nameless representation of the λ-calculus. More concretely, we use the de Bruijn notation [5]. We start with some simple examples to clarify the main ideas.

For instance, the expression $\lambda x.x$ is written using the de Bruijn indices as $\lambda.0$. That is, the index 0 indicates that the variable is bound to the λ and no variable name is needed. The same happens with the expression $\lambda y.y$, that it is written as $\lambda.0$ too. Therefore, both expressions have exactly the same representation.

The term $\lambda x.\lambda y.(y\ x)$ becomes in the de Bruijn notation $\lambda.\lambda.(0\ 1)$, where each index indicates to which λ are bound the variables.

Notice that the same number can be used for different variables names, and that the same variable name can be represented with different indices depending on the position where it occurs in the expression. For example, $\lambda x.\lambda y.((y\ x)\ (\lambda u.(x\ u)))$ is represented as $\lambda.\lambda.((0\ 1)\ (\lambda.(2\ 0)))$, where 0 is used for y and z because both are bound to the nearest λ (from inside out), while the two occurrences of x are represented with the indices 1 and 2.

Let-expressions are similar to λ-expressions, but with more bound variables. Therefore, an order should be given to these bound variables. For instance, $\texttt{let}\ \{x_1 = \lambda y.(y\ x_2), x_2 = x_3, x_3 = \lambda u.u\}\ \texttt{in}\ x_1$ becomes $\texttt{let}\ \{\lambda.(0\,2), 0, \lambda.0\}\ \texttt{in}\ 2$. Notice that, as it happens with the λ-abstractions, the names of the variables do not appear, only the expressions to which they are bound. The order of appearance is relevant to establish the indices, which grow from right to left.

The syntax of the *de Bruijn expressions* is formalized as follows:

Definition 1 *Let* $\mathcal{BE}xp$ *be the smallest family of sets* $\{\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \dots\}$ *such that*

1. $k \in \mathcal{B}_n$ *whenever* $0 \leq k < n$;
2. *if* $t \in \mathcal{B}_n$ *and* $n > 0$, *then* $\lambda.t \in \mathcal{B}_{n-1}$;
3. *if* $t \in \mathcal{B}_n$ *and* $0 \leq k < n$, *then* $(t\ k) \in \mathcal{B}_n$;
4. *if* $t, t_1, \dots, t_m \in \mathcal{B}_n$ *and* $n - m > 0$, *then* $\texttt{let}\ \{t_i\}_{i=1}^m\ \texttt{in}\ t \in \mathcal{B}_{n-m}$.

The elements of $\mathcal{B}_n$ are expressions with at most n free variables, numbered between 0 and $n-1$. In the case of a λ-abstraction, if t has at most n free variables, then $\lambda.t$ has at most $n-1$ because the *first* free variable in t becomes bound. Analogously, in a *let*-expression m variables are bound, so that it can have at most $n-m$ free variables. The set of variables (or indices) in the de Bruijn notation is represented by $\mathcal{BV}ar$ and it corresponds to the set of natural numbers. The set of values (i.e. *whnf*) is represented by $\mathcal{BV}al$.

Definition 2 *Let* $\forall k : 0 \leq k \leq n \in \mathcal{BV}ar : t_k \in \mathcal{BE}xp \cup \{\bot\}(\mathcal{BE}xp_\bot)$. *A* de Bruijn-heap Γ_B *is an ordered set* $\{t_n, \dots, t_0\}$, *denoted also as* $\{t_k\}_{k=n}^0$, *where each* t_k *corresponds to the*

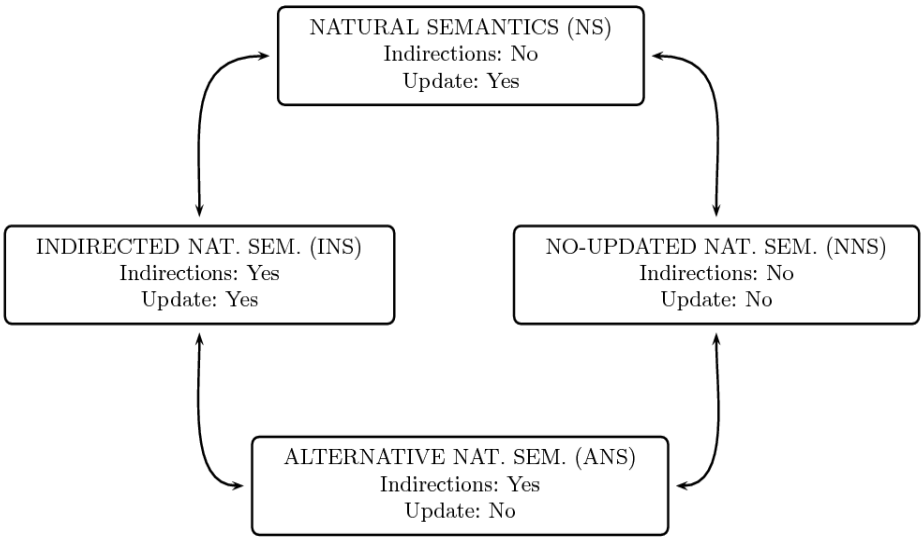


Figure 3: The lazy natural semantics and its alternatives

expression bound to the position k. $t_k = \perp$ indicates that the variable corresponding to position k is undefined and the access to it is not allowed. The set of the de Bruijn-heaps is denoted as $\mathcal{B}Heap$.

Definition 3 *Let $\Gamma_B = \{t_k\}_{k=n}^{0} \in \mathcal{B}Heap$, $i \in \mathcal{B}Var$ with $0 \leq i \leq n$, and $t \in \mathcal{B}Exp_{\perp}$. $\Gamma_B\{i \mapsto t\}$ represents the* modified heap *that equals Γ_B except for t_i, which is replaced by t, i.e. $\Gamma_B\{i \mapsto t\} = \{t_n, \ldots, t_{i+1}, t, t_{i-1}, \ldots, t_0\}$.*

The "*shifting*" function is an auxiliary operation for renumbering the indices of some free variables in an expression. In $\uparrow_c^d (t)$, d is a number that indicates how much the indices should be augmented (or decremented if d is negative), and c is a number that determines from which position the indices should be modified.

Definition 4 *Let $d \in \mathbb{Z}$, $c \in \mathcal{B}Var$, and $t \in \mathcal{B}Exp_{\perp}$. The* d-place shift of an expression t above cutoff c, *written $\uparrow_c^d (t)$, is defined as follows:*

1. $\uparrow_c^d (k) = \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases}$;
2. $\uparrow_c^d (\lambda.t) = \lambda.\uparrow_{c+1}^d (t)$;
3. $\uparrow_c^d (t\ k) = ((\uparrow_c^d (t))\ (\uparrow_c^d (k)))$;
4. $\uparrow_c^d (\texttt{let}\ \{t_i\}_{i=1}^m\ \texttt{in}\ t) = \texttt{let}\ \{\uparrow_{c+m}^d (t_i)\}_{i=1}^m\ \texttt{in}\ \uparrow_{c+m}^d (t)$;
5. $\uparrow_c^d (\perp) = \perp$.

This *shifting* function can be extended to a de Bruijn-heap by applying it to every expression in the heap:

Definition 5 *Let $\Gamma_B = \{t_k\}_{k=n}^{0} \in \mathcal{B}Heap$, $d \in \mathbb{Z}$ and $c \in \mathcal{B}Var$. We define the* shifting *of Γ_B as $\uparrow_c^d (\Gamma_B) = \{\uparrow_c^d (t_k)\}_{k=n}^{0}$.*

Now, we can formalize the substitution of an expression for a variable in the de Bruijn notation:

Definition 6 *Let $s \in \mathcal{B}Exp$, $t \in \mathcal{B}Exp_{\perp}$, and $j \in \mathcal{B}Var$. The* substitution of an expression s for a variable j *in an expression t, written $t[s/j]$, is defined inductively as follows:*

1. $k[s/j] = \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases}$;
2. $(\lambda.t)[s/j] = \lambda.t[\uparrow_0^1 (s)/j + 1]$;

3. $(t\ k)[s/j] = (t[s/j]\ k[s/j])$;

4. $(\texttt{let}\ \{t_i\}_{i=1}^{m}\ \texttt{in}\ t)[s/j] = \texttt{let}\{t_i[\uparrow_0^m(s)/j+m]\}_{i=1}^{m}\texttt{in}\ t[\uparrow_0^m(s)/j+m]$;

5. $\bot\ [s/j] = \bot$.

Again, substitution can be extended to a de Bruijn-heap by applying it to every expression in the heap:

Definition 7 *Let* $\Gamma_B = \{t_k\}_{k=n}^{0} \in \mathcal{B}Heap$, $s \in \mathcal{B}Exp$, *and* $j \in \mathcal{B}Var$. *The* substitution of a term s for a variable j *in a heap* Γ_B, *written* $\Gamma_B[s/j]$, *is defined as* $\Gamma_B[s/j] = \{t_k[s/j]\}_{k=n}^{0}$.

When a de Bruijn-heap is extended with some new bindings the indices in the old bindings have to be incremented correspondingly.

Definition 8 *Let* $\Gamma_B = \{t_k\}_{k=n}^{0} \in \mathcal{B}Heap$, *and* $\forall i : 1 \leq i \leq m : t'_i \in \mathcal{B}Exp_{\bot}$. *An* extension *of* Γ_B *is*

$$\Gamma_B + \{t'_i\}_{i=1}^{m} = \{\uparrow_0^m(t_n), \ldots, \uparrow_0^m(t_0), t'_1, \ldots, t'_m\}.$$

In Figures 4 and 5 we express in de Bruijn notation () the operational rules given in Section 2. The notation $|.|$ indicates the cardinal of a de Bruijn-heap.

Notice how β-reduction is done in the BApplication rule. First of all, the λ-abstraction (i.e. $\lambda.t'$) is obtained, but this implies that the position of the variable k may have changed in the new heap Δ_B; thus k must be augmented by the difference between the cardinals of Δ_B and Γ_B. Then, a substitution of the argument (augmented by one) for the variable 0 (the index representing the variable bound by λ) in t' is performed. After that, as the λ is eliminated, the free variables should be decreased by one, including the one corresponding to the argument.

In the BVariable rule, the expression t_k is replaced by a $\bot$ value in the heap. Just removing from the heap the expression t_k, as it is done in the rule Variable (Figure 1), would imply renumbering every index in the expression and in the de Bruijn-heap each time that a variable is evaluated. Moreover, if some other expression has a reference to position k and this has been eliminated, which should be the new index for this position? For this reason, we have changed the expression t_k for $\bot$. In this way, if the variable represented by k is evaluated again, the $\bot$ value is found and the evaluation cannot continue (*black hole*).

In the rule BLet, the heap is extended with the new local variables.

In the rule BAApplication of the alternative semantics, the heap is extended with a new indirection.

4 Equivalence

In this section we define several ways of relating heap/term pairs, which help us to establish the equivalence between the semantics defined in Section 2. We cope first with the problem of adding indirections in a heap, and afterwards we study the problems that arise when there is no update.

We begin with some auxiliary definitions.

First of all, we have to calculate the free variables of an expression that is written with the de Bruijn notation.

Definition 9 *Let* $t \in \mathcal{B}Exp$, $L \subseteq \mathcal{B}Var$, *and* $n \in \mathbb{N}$. *The* free variables of t in a context (L, n), *written* $\mathcal{FV}(t, L, n)$, *is a list of variables defined inductively as follows:*

1. $\mathcal{FV}(k, L, n) = \begin{cases} L & \text{if } k < n \\ L +\!\!+ [k-n] & \text{if } k \geq n \end{cases}$;

2. $\mathcal{FV}(\lambda.t, L, n) = \mathcal{FV}(t, L, n+1)$;

3. $\mathcal{FV}((t\ k), L, n) = \mathcal{FV}(t, L, n) +\!\!+ \mathcal{FV}(k, L, n)$;

4. $\mathcal{FV}(\texttt{let}\ \{t_i\}_{i=1}^{m}\ \texttt{in}\ t) = \mathcal{FV}(t, L_m, n+m)$, *where* $L_j = \mathcal{FV}(t_j, L_{j-1}, n+m)$, $j = 1, \ldots, m$ *and* $L_0 = L$.

In this definition $+\!\!+$ represents the usual concatenation operation on lists. The argument n indicates how many variables are already bound. We do not return in the list the index that the free variable has in the expression, but the free variable number itself without the context of the expression.

Example 2 *We calculate the free variables of* $t \equiv \lambda.(\lambda.(5\ 0)\ 3)$. *It is easy to realize that indices* 5 *and* 3 *represent free variables, but the*

$$\textsf{BLambda} \qquad \Gamma_B : \lambda.t \Downarrow \Gamma_B : \lambda.t$$

$$\textsf{BApplication} \qquad \frac{\Gamma_B : t \Downarrow \Delta_B : \lambda.t' \qquad \Delta_B : \uparrow_0^{-1} (t'[\uparrow_0^1 (k + |\Delta_B| - |\Gamma_B|)/0]) \Downarrow \Theta_B : z}{\Gamma_B : (t\ k) \Downarrow \Theta_B : z}$$

$$\textsf{BVariable} \qquad \frac{\Gamma_B\{k \mapsto \bot\} : t_k \Downarrow \Delta_B : z}{\Gamma_B : k \Downarrow \Delta_B\{k + |\Delta_B| - |\Gamma_B| \mapsto z\} : z} \qquad \text{if } t_k \neq \bot$$

$$\textsf{BLet} \qquad \frac{\Gamma_B + \{t_i\}_{i=1}^m : t \Downarrow \Delta_B : z}{\Gamma_B : \texttt{let}\ \{t_i\}_{i=1}^m\ \texttt{in}\ t \Downarrow \Delta_B : z}$$

Figure 4: Natural Semantics in the de Bruijn notation

$$\textsf{BAApplication} \qquad \frac{\Gamma_B : t \Downarrow \Delta_B : \lambda.t' \qquad \Delta_B + \{k + |\Delta_B| - |\Gamma_B| + 1\} : t' \Downarrow \Theta_B : z}{\Gamma_B : (t\ k) \Downarrow \Theta_B : z}$$

$$\textsf{BAVariable} \qquad \frac{\Gamma_B : t_k \Downarrow \Delta_B : z}{\Gamma_B : k \Downarrow \Delta_B : z} \qquad \text{if } t_k \neq \bot$$

Figure 5: Alternative Rules in the de Bruijn notation

function $\mathcal{FV}$ defined above do not return 5 *and* 3*, but the "correct" indices that do not depend on the expression itself. We start with L as the empty list (represented as $[\,]$), and we set number n to* 0*, because outside the expression t there are no bound variables.*

$$\begin{aligned}
\mathcal{FV}(t, [\,], 0) &= \mathcal{FV}((\lambda.(5\ 0)\ 3), [\,], 1) \\
&= \mathcal{FV}(\lambda.(5\ 0), [\,], 1) \\
&\mathbin{+\!\!+} \mathcal{FV}(3, [\,], 1) \\
&= \mathcal{FV}((5\ 0), [\,], 2) \mathbin{+\!\!+} [2] \\
&= (\mathcal{FV}(5, [\,], 2) \\
&\mathbin{+\!\!+} \mathcal{FV}(0, [\,], 2)) \mathbin{+\!\!+} [2] \\
&= ([3] \mathbin{+\!\!+} [\,]) \mathbin{+\!\!+} [2] \\
&= [3, 2]
\end{aligned}$$

When a position is referred neither in the heap nor in the expression to be evaluated, then we can say it is free of pointers.

Definition 10 *Let $t \in \mathcal{BE}xp_\bot$, and $i \in \mathcal{BV}ar$. Position i is said to be* free of pointers with respect to t *if $i \notin \mathcal{FV}(t, [\,], 0)$.*

Definition 11 *Let $\Gamma_B = \{t_k\}_{k=n}^0 \in \mathcal{BH}eap$, and $i \in \mathcal{BV}ar$. Position i is said to be* free of pointers with respect to Γ_B *if $\forall k : 0 \leq k \leq n : i \notin \mathcal{FV}(t_k, [\,], 0)$.*

When a position is free of pointers, it can be eliminated from the heap, but the other indices have to be updated.

Definition 12 *Let $\Gamma_B = \{t_k\}_{k=n}^0 \in \mathcal{BH}eap$, and position $i \in \mathcal{BV}ar$ is free of pointers with respect to Γ_B. Then we define a new de Bruijn-heap by deleting position i from the heap Γ_B:*

$$\Gamma_B - i = \uparrow_i^{-1}\{t_n, \ldots, t_{i+1}, t_{i-1}, \ldots, t_0\}.$$

4.1 Indirections

We start by investigating how to relate heap/term pairs where the only difference is the addition of indirections. An *indirection* is a binding from a variable to a variable, i.e. a binding of the form $x \mapsto y$, where x and y are variables. Using the de Bruijn notation we say that position k is an indirection if $t_k \in \mathcal{BV}ar$. To illustrate the situation, we give the following example.

Example 3 *Let us consider the expression $t \equiv \texttt{let}\ \{1, \lambda 0, \lambda.(3\ 0)\}\ \texttt{in}\ (1\ 0)$ to be evaluated*

in the context of the empty heap. The results obtained with the different semantics are:

$$\begin{array}{ll} \{\} : t & \Downarrow \quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \\ \{\} : t & \Downarrow^I \quad \{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0) \\ \{\} : t & \Downarrow^N \quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \\ \{\} : t & \Downarrow^A \quad \{2, \lambda.0, \lambda.(4\ 0), 1\} : \lambda.(4\ 0) \end{array}$$

We can observe that there are some differences between the final heap obtained with $\Downarrow^N$ and the one obtained with $\Downarrow^A$. For instance, there is an extra indirection, 1 in position 0 (i.e. the rightmost), on the heap of $\Downarrow^A$. This extra indirection can be easily eliminated in this case because it is free of pointers. It can also be observed that some indices are not the same (more exactly the free variables are incremented by one), but this is due to the extra indirection.

We would like to eliminate extra indirections. How can we determine if an indirection is extra? Let us consider again the final heap produced by $\Downarrow^A$ in the previous example. The indirection in position 3 (i.e. the variable number 2) should not be eliminated because it also appears in the final heap obtained with $\Downarrow^N$ (as variable number 1 in position 2). In fact, this indirection is introduced by the BLet rule, and not by the BAApplication rule).

We define a relation between heap/term pairs such that they are related if either they are exactly the same or if one can be obtained from the other by deleting some indirections. When eliminating indirections we have to be sure that no expression appearing in the heap loses its meaning.

Definition 13 *Let $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$, and $t, t' \in \mathcal{B}Exp$. $\Gamma_B : t$ is* indirection-related *to $\Gamma'_B : t'$ (denoted by $\Gamma_B : t \succsim_I \Gamma'_B : t'$) if:*

- *$\Gamma_B = \Gamma'_B$ and $t \equiv t'$, or*
- *$\Gamma_B[j/k] - k : \uparrow_k^{-1}(t[j/k]) \succsim_I \Gamma'_B : t'$, for some $k : 0 \leq k \leq |\Gamma_B|$, such that $t_k \equiv j \neq k$ and $j \in \mathcal{B}Var$.*

Example 4 *Let us consider again Example 3. We take $\Gamma_B : z = \{2, \lambda.0, \lambda.(4\ \ 0), 1\} : \lambda.(4\ 0)$, the result obtained with the ANS. The heap/term pairs related with $\Gamma_B : z$ are shown below:*

$$\begin{array}{ll} (a) & \{2, \lambda.0, \lambda.(4\ 0), 1\} : \lambda.(4\ 0) \\ (b) & \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \\ (c) & \{\lambda.0, \lambda.(3\ 0), 1\} : \lambda.(3\ 0) \\ (d) & \{\lambda.0, \lambda.(2\ 0)\} : \lambda.(2\ 0) \end{array}$$

We observe that (b) corresponds with the heap/term pair obtained with the NNS.

To define a similar relation between the final heap/term pairs obtained by the NS and the INS is not so straightforward, because these semantics do update bindings. Therefore, if some indirection, say $t_k \equiv j$, has been updated, it becomes $z \in \mathcal{B}Val$. But this value z must coincide with the one in position j, that is, to t_j in the final heap.

Example 5 *In Example 3, in the final heap obtained with the INS, the expression in position 0 coincides with the value in position 1 (i.e. $t_0 \equiv \lambda.(4\ 0) \equiv t_1$), because somewhere in the derivation $\Downarrow^I$ position 0 was bound to 1. We show a fragment of the derivations $\Downarrow$ and $\Downarrow^I$:*

$$\begin{array}{ll} \Downarrow & \{\} : \texttt{let}\ \{1, \lambda.0, \lambda.(3\ 0)\}\ \texttt{in}\ (1\ 0) \\ & \quad \{1, \lambda.0, \lambda.(3\ 0)\} : (1\ 0) \\ & \quad\quad \{1, \lambda.0, \lambda.(3\ 0)\} : 1 \\ & \quad\quad \vdots \\ & \quad\quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.0 \\ & \quad\quad \{\mathbf{1}, \lambda.\mathbf{0}, \lambda.(\mathbf{3\ 0})\} : \mathbf{0} \\ & \quad\quad \vdots \\ & \quad\quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \\ & \quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \\ & \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0) \end{array}$$

$$\begin{array}{ll} \Downarrow^I & \{\} : \texttt{let}\ \{1, \lambda.0, \lambda.(3\ 0)\}\ \texttt{in}\ (1\ 0) \\ & \quad \{1, \lambda.0, \lambda.(3\ 0)\} : (1\ 0) \\ & \quad\quad \{1, \lambda.0, \lambda.(3\ 0)\} : 1 \\ & \quad\quad \vdots \\ & \quad\quad \{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.0 \\ & \quad\quad \{\mathbf{2}, \lambda.\mathbf{0}, \lambda.(\mathbf{4\ 0}), \mathbf{1}\} : \mathbf{0} \\ & \quad\quad \vdots \\ & \quad\quad \{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0) \\ & \quad \{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0) \\ & \{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0) \end{array}$$

We have emphasized the point where the application is evaluated, i.e. the moment where the indirection due to the BAAplication *rule is introduced in the derivation $\Downarrow^I$, while a β-reduction is performed in the derivation $\Downarrow$.*

Thus, we cannot directly use the indirection-relation. We need to define a relation that eliminates from the heap some bindings that are *redundant* (i.e. several positions that are bound to the same value).

Definition 14 *Let $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$, and $t, t' \in \mathcal{B}Exp$. $\Gamma_B : t$ is* redundant-related *to $\Gamma'_B : t'$ (denoted by $\Gamma_B : t \succsim_R \Gamma'_B : t'$) if:*

- *$\Gamma_B = \Gamma'_B$ and $t \equiv t'$, or*
- *$\Gamma_B[j/k] - k : \uparrow_k^{-1}(t[j/k]) \succsim_R \Gamma'_B : t'$, for some $k, j : 0 \leq k, j \leq |\Gamma_B|$ such that $t_k \equiv t_j$, $t_k \not\equiv k$, and $k \neq j$.*

This relation by itself is not valid for our purposes, since some indirections may not be updated because their evaluation is not required during the derivation. In Example 3, t_3 in the final heap obtained with the INS is not a value because it was not needed to evaluate the initial expression, therefore it remains as an indirection in the final heap. For this reason, we define a transformation that combines the two relations defined above.

Definition 15 *The* heap-transformation *(denoted by $\succsim_H$) is defined as $\succsim_H = \succsim_I ; \succsim_R$.*

Example 6 *We consider the final heap/term pair obtained with the INS in Example 3:*

$$\Gamma_B : z = \{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0).$$

This is heap-transformed to the following heap/term pairs:

(a) $\{2, \lambda.0, \lambda.(4\ 0), \lambda.(4\ 0)\} : \lambda.(4\ 0)$
(b) $\{1, \lambda.0, \lambda.(3\ 0)\} : \lambda.(3\ 0)$
(c) $\{\lambda.0, \lambda.(3\ 0), \lambda.(3\ 0)\} : \lambda.(3\ 0)$
(d) $\{\lambda.0, \lambda.(2\ 0)\} : \lambda.(2\ 0)$

We observe that the final heap/term pair computed by the NS coincides with (b).

We can now prove the equivalence between a semantics and its version *including indirections.*

Theorem 4 (Equivalence between the NS and the INS) *Let $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$, and $t, t' \in \mathcal{B}Exp$ such that $\Gamma'_B : t' \succsim_H \Gamma_B : t$. $\Gamma_B : t \Downarrow \Delta_B : z$ if and only if $\Gamma'_B : t' \Downarrow^I \Delta'_B : z'$ and $\Delta'_B : z' \succsim_H \Delta_B : z$.*

Theorem 5 (Equivalence between the NNS and the ANS) *Let $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$, and $t, t' \in \mathcal{B}Exp$ such that $\Gamma'_B : t' \succsim_I \Gamma_B : t$. $\Gamma_B : t \Downarrow^N \Delta_B : z$ if and only if $\Gamma'_B : t' \Downarrow^A \Delta'_B : z'$ and $\Delta'_B : z' \succsim_I \Delta_B : z$.*

The proofs of these theorems are done by induction on the derivations.

4.2 No-update

Now, we investigate how to relate heap/term pairs when the difference lies on the no-update of evaluated bindings. As it was done in the previous section, we use an example to illustrate the situation.

Example 7 *Let us consider the expression $t \equiv$* `let` $\{\lambda.\lambda.(1\ 0), (3\ 3), (3\ 2), (2\ 1)\}$ `in` $(1\ 0)$ *to be evaluated in the context of the empty heap. The results obtained with the different semantics are:*

$\{\} : t \Downarrow \{\lambda.\lambda.(1\ 0), \lambda.(4\ 0), \lambda.(3\ 0), (2\ 1)\} : \lambda.(1\ 0)$
$\{\} : t \Downarrow^N \{\lambda.\lambda.(1\ 0), (3\ 3), (3\ 2), (2\ 1)\} : \lambda.(1\ 0)$
$\{\} : t \Downarrow^I \{\lambda.\lambda.(1\ 0), \lambda.(3\ 0), \lambda.(5\ 0), (7\ 6), \lambda(3\ 0), 5, \lambda.\lambda.(1\ 0), 3, 1\} : \lambda.(1\ 0)$
$\{\} : t \Downarrow^A \{\lambda.\lambda.(1\ 0), (8\ 8), (8\ 7), (7\ 6), 7, 5, 8, 3, 1\} : \lambda.(1\ 0)$

We observe that in the final heap achieved with $\Downarrow$ the expressions are updated with the values obtained during the derivation, while in the case of $\Downarrow^N$ they are not. Nevertheless, if these expressions are evaluated with the NNS in the context of the final heap obtained with $\Downarrow^N$, then the corresponding values are the same as those in the final heap for $\Downarrow$.

Relating the final heap/term pairs obtained with the INS and the ANS becomes more awkward. When a non-updated expression is evaluated in the context of the final heap obtained

with the ANS, some new terms are possibly added to the heap, and the computed value could depend on them. Therefore, the obtained value may not correspond exactly to the one obtained with the INS. The following example illustrates this situation.

Example 8 *Let us consider the expression in Example 7, and let us denote by* Γ_B^I *the final heap achieved with* $\Downarrow^I$*, and by* Γ_B^A *the one obtained with* $\Downarrow^A$*. The following table shows the differences between* Γ_B^I *and* Γ_B^A*:*

Position	Γ_B^I	Γ_B^A
2	$\lambda.\lambda.(1\ 0)$	8
4	$\lambda.(3\ 0)$	7
6	$\lambda.(5\ 0)$	$(8\ 7)$
7	$\lambda.(3\ 0)$	$(8\ 8)$

Observe, for instance, that the value $\lambda.(5\ 0)$ *is in position* 6 *in* Γ_B^I*, while in* Γ_B^A *the expression* $(8\ \ 7)$ *is in position* 6*. However,* $\Gamma_B^A : (8\ 7) \Downarrow^A \Gamma_B^A + \{8\} : \lambda.(1\ 0)$*, where an indirection has been added to the heap. The value* $\lambda.(1\ 0)$ *coincides with* $\lambda.(5\ 0)$ *(the value in* Γ_B^I*) except for the free variable. We could consider that both expressions are equivalent if the free variables evaluate to equivalent values in their respective heaps. Index* 5 *in* $\lambda.(5\ 0)$ *refers to position* 4 *in* Γ_B^I*, and* $\Gamma_B^I : 4 \Downarrow^I \Gamma_B^I : \lambda.(3\ 0)$*. Whereas index* 1 *in* $\lambda.(1\ \ 0)$ *refers to position* 0 *in* $\Gamma_B^A + \{8\}$*, and* $\Gamma_B^A + \{8\} : 0 \Downarrow^A (\Gamma_B^A + \{8\}) + \{10\} : \lambda.(10)$*. Once again the difference between* $\lambda.(3\ 0)$ *and* $\lambda.(1\ 0)$ *is the free variable. If we repeat the process, then we obtain the closed value* $\lambda.\lambda.(1\ 0)$ *in both cases.*

The question now is: How far is it necessary to compute the expressions contained in the final heap obtained with the ANS to achieve the same values as with the INS? There is also another problem: When evaluating a let-expression in the ANS, the expressions that represent the bound variables are added to the heap each time that the let-expression is called during the evaluation. This means that, in general, there are more expressions in the heap obtained with the ANS than in the heap obtained with the INS. But these expressions are redundant (in the sense that they have been introduced in the heap several times). Therefore, if they are eliminated, the resulting de Bruijn-heap differs only in the updating of terms from the one obtained with the INS.

Before giving the formal definition of this relation, we need some auxiliary definitions.

Two expressions have the same structure if they have the same syntax except for the indices of the free variables.

Definition 16 *Let* $t, t' \in \mathcal{B}Exp$*. We say that* t *and* t' *have the* same structure*, denoted by* $t' \sim_S t$*, if either*

- $t \equiv k$ *and* $t' \equiv k'$ *where* $k, k' \in \mathcal{B}Var$*, or*
- $t \equiv \lambda.t_1$ *and* $t' \equiv \lambda.t_2$ *and* $t_1 \sim_S t_2$*, or*
- $t \equiv (t_1\ k)$ *and* $t' \equiv t_2\ k'$ *and* $t_1 \sim_S t_2$ *where* $k, k' \in \mathcal{B}Var$*, or*
- $t \equiv \texttt{let}\ \{t_i\}_{i=1}^m\ \texttt{in}\ t_1$ *and* $t' = \texttt{let}\ \{t'_i\}_{i=1}^m\ \texttt{in}\ t_2$ *and* $\forall i : 1 \leq i \leq m : t_i \sim_S t'_i$ *and* $t_1 \sim_S t_2$*.*

Proposition 6 $\sim_S$ *is an equivalence relation*

Proof. By induction on the structure of t. □

Two heap/value pairs are equivalent respect to the NNS and the NS either if the values are closed terms and syntactically equal or, if they are open then their free variables evaluate to equivalent heap/value pairs.

Definition 17 *Let* $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$*,* $z, z' \in \mathcal{B}Val$*. We say that* $\Gamma_B : z$ *is* update-equivalent *to* $\Gamma'_B : z'$ *(denoted by* $\Gamma_B : z \approx_U \Gamma'_B : z'$*) if:*

- z *and* z' *are closed terms and* $z \equiv z'$*, or*
- z *and* z' *are not closed,* $z \sim_S z'$*, and if* $\mathcal{FV}(z, [\ ], 0) = [i_1, \ldots, i_n]$ *and* $\mathcal{FV}(z', [\ ], 0) = [i'_1, \ldots, i'_n]$ *then* $\forall j : 1 \leq j \leq n : \Gamma_B : i_j \Downarrow^N \Delta_B : z_j \Leftrightarrow \Gamma'_B : i'_j \Downarrow \Delta'_B : z'_j$ *and* $\Delta_B : z_j \approx_U \Delta'_B : z'_j$*.*

A similar heap/value relation can be defined for the ANS and the INS:

Definition 18 *Let* $\Gamma_B, \Gamma'_B \in \mathcal{B}Heap$*,* $z, z' \in \mathcal{B}Val$*. We say that* $\Gamma_B : z$ *is* alternative-update-equivalent *to* $\Gamma'_B : z'$ *(denoted by* $\Gamma_B : z \approx_{AU} \Gamma'_B : z'$*) if:*

- z *and* z' *are closed terms and* $z = z'$, *or*
- z *and* z' *are not closed,* $z \sim_S z'$, *and if* $\mathcal{FV}(z, [\,], 0) = [i_1, \ldots, i_n]$ *and* $\mathcal{FV}(z', [\,], 0) = [i'_1, \ldots, i'_n]$ *then* $\forall j : 1 \leq j \leq n : \Gamma_B : i_j \Downarrow^N \Delta_B : z_j \Leftrightarrow \Gamma'_B : i'_j \Downarrow \Delta'_B : z'_j$ *and* $\Delta_B : z_j \approx_{AU} \Delta'_B : z'_j$.

Now we can give a formal definition to relate a heap/term pair of the NNS to other of the NS.

Definition 19 *Let* $\Gamma_B, \Gamma'_B, \Delta_B, \Delta'_B \in \mathcal{BH}eap$, $t, t' \in \mathcal{BE}xp$, *and* $z, z' \in \mathcal{BV}al$ *such that*

- $\Gamma_B : t \Downarrow^N \Delta_B : z$, *and*
- $\Gamma'_B : t' \Downarrow \Delta'_B : z'$.

We say that $\Gamma_B : t$ *is* update-related *to* $\Gamma'_B : t'$ *(denoted by* $\Gamma_B : t \succsim_U \Gamma'_B : t'$*) if there exists* $\Delta''_B \in \mathcal{BH}eap$, *and* $z'' \in \mathcal{BV}al$ *such that*

- $\Delta_B : z \succsim_R \Delta''_B : z''$ *with* $|\Delta''_B| = |\Delta'_B| = n$,
- $\Delta''_B : z'' \approx_V \Delta'_B : z'$, *and*
- $\Delta''_B : i \succsim_U \Delta_B : i$ *for all* $0 \leq i \leq n$.

Similarly we desire to relate a heap/term pair of the ANS to one of the INS as follows:

Definition 20 *Let* $\Gamma_B, \Gamma'_B, \Delta_B, \Delta'_B \in \mathcal{BH}eap$, $t, t' \in \mathcal{BE}xp$, *and* $z, z' \in \mathcal{BV}al$ *such that*

- $\Gamma_B : t \Downarrow^A \Delta_B : z$, *and*
- $\Gamma'_B : t' \Downarrow^I \Delta'_B : z'$.

We say that $\Gamma_B : t$ *is* alternative-update-related *to* $\Gamma'_B : t'$ *(denoted by* $\Gamma_B : t \succsim_{AU} \Gamma'_B : t'$*) if there exists* $\Delta''_B \in \mathcal{BH}eap$, *and* $z'' \in \mathcal{BV}al$ *such that*

- $\Delta_B : z \succsim_R \Delta''_B : z''$ *with* $|\Delta''_B| = |\Delta'_B| = n$,
- $\Delta''_B : z'' \approx_{AV} \Delta'_B : z'$, *and*
- $\Delta''_B : i \succsim_{AU} \Delta_B : i$ *for all* $0 \leq i \leq n$.

The following theorems establish the equivalence between two semantics where the first updates the heaps, while the second does not.

Theorem 7 (Equivalence between the NS and the NNS) *Let* Γ_B, Γ'_B *be heaps and* t, t' *be expressions such that* $\Gamma_B : t \succsim_U \Gamma'_B : t'$. $\Gamma'_B : t' \Downarrow \Delta'_B : z'$ *if and only if* $\Gamma_B : t \Downarrow^N \Delta_B : z$ *and* $\Delta_B : z \succsim_U \Delta'_B : z'$.

Theorem 8 (Equivalence between the INS and the ANS) *Let* Γ_B, Γ'_B *be heaps and* t, t' *be expressions such that* $\Gamma_B : t \succsim_{AU} \Gamma'_B : t'$.
$\Gamma'_B : t' \Downarrow^I \Delta'_B : z'$ *if and only if* $\Gamma_B : t \Downarrow^A \Delta_B : z$ *and* $\Delta_B : z \succsim_{AU} \Delta'_B : z'$.

The proofs of these theorems are done by induction on the derivations.

5 Conclusions and future work

The variations introduced by Launchbury in its alternative semantics (ANS) do affect two rules: the application rule (where indirections are added to the heap) and the variable rule (where no update is done). Instead of dealing with both transformations simultaneously, we have defined two intermediate semantics, each introducing a single change with respect to the original natural semantics (NS): the INS (with indirections) and the NNS (with no update). In this way we have been able to study separately the influence of each modification.

In order to avoid the usual problems with variable names and α-conversion, we have translated Launchbury's semantics to the de Bruijn notation [5], where variable names are removed and replaced by natural numbers. Using this notation we have defined three relations between heap/term pairs that help us to compare the values and final heaps obtained with/without indirections, so that Theorems 4 and 5 establish the equivalence between the NS and the INS, and between the NNS and the ANS, respectively.

Similarly, in order to be able to prove the equivalence between the NS and the NNS, and between the INS and the ANS (Theorems 7 and 8), we have defined another set of relations between heap/term pairs with/without updates.

The proof of the equivalence between the natural semantics and its alternative has in-

volved an ample amount of technical propositions and lemmas, whose proofs are usually long, plenty of subcases, tedious and error-prone. Proof assistants are helpful to deal with this kind of proofs. The development of automatic theorem provers is nowadays at its height. Some of the most relevant proof systems are Isabelle [8], Coq [3], AGDA [4] and PVS [9]. We have chosen Isabelle [8] to implement the four operational semantics given in Section 2. One of the reasons is that the standard λ-calculus has been defined already in Isabelle in several ways. Moreover, our language has recursive local declarations, and the way of representing this kind of calculus in Isabelle is actually by using the de Bruijn notation.

Our concern for reproducing the proof of the equivalence between these semantics is not arbitrary. We want to extend the λ-calculus with a new expression that introduces parallelism when performing functional applications. This *parallel application* would create new processes to distribute the computation, and these distributed processes would exchange values through communication channels.

ACKNOWLEDGEMENTS We want to thank J. Launchbury for his comments. We also appreciate the help of Stefan Berghofer. This work is partially supported by the following grants: TIN2009-14599-C03-01, BES-2007-16823, S2009/TIC-1465.

References

[1] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

[2] Z M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[3] Y. Bertot. Coq in a hurry. *CoRR*, abs/cs/0603118, 2006.

[4] A. Bove, P. Dybjer, and A. Sicard-Ramírez. Embedding a logical theory of constructions in AGDA. In *PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 59–66, New York, NY, USA, 2008. ACM.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[6] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.

[7] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[9] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-calvert. PVS system guide, 2001.

[10] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Journal of Theoretical Computer Science*, 1(2):125–159, 1975.

[11] M. Schmidt-Schauß. Equivalence of call-by-name and call-by-need for lambda-calculi with letrec. Technical report, 2006.

Automating Derivations of Abstract Machines from Reduction Semantics:
A Generic Formalization of Refocusing in Coq

Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki

Institute of Computer Science, University of Wrocław

We present a generic formalization of Danvy and Nielsen's refocusing transformation for functional languages [7] in the Coq proof assistant [1]. The refocusing technique allows for a mechanical transformation of an evaluator implementing a reduction semantics (i.e., a small-step operational semantics with explicit representation of evaluation contexts) into an equivalent abstract machine via a succession of simple program transformations. Whereas the original derivation method dealt with substitution-based reduction semantics and accounted for local contractions only, it has later been extended by Biernacka and Danvy to facilitate derivations of environment-based machines from reduction semantics using explicit substitutions [3], and of abstract machines derived from context-sensitive reduction semantics [4]. The refocusing method is steadily gaining currency—e.g., recently, it has been used to derive abstract machines for Scheme [5] and for call-by-need lambda calculi [6,8]. However, refocusing has been used as an informal procedure: the conditions required of a reduction semantics have not been formally captured, and the entire transformation has not been formalized and proven correct. In the original article introducing the vanilla version of refocusing Danvy and Nielsen propose a set of conditions on a reduction semantics, and they sketch a correctness proof of the refocused evaluation function. However, they do not consider refocusing the same way as we do—as a succession of simple transformations; rather, they focus on the final efficient definition of an evaluation function. Moreover, we find the representation of reduction semantics used in their paper not adequate for a formalization on a computer.

The aim of this work is to formalize and prove correct the refocusing technique. To this end, we first propose an axiomatization of reduction semantics that is sufficient to automatically apply the refocusing method. Next, we prove that any reduction semantics conforming to this axiomatization can be automatically transformed to an abstract machine equivalent to it. We formalize each intermediate step of the derivation and we state and prove its correctness with respect to the previous one. Our work is based on preliminary results by Biernacka and Biernacki [2] which we extend to a general framework. In addition, we provide a number of case studies ranging from simple to more sophisticated languages (including Mini-ML). They serve both as examples illustrating how to use the formalization in practice and as sanity checks on the axiomatization. The formalization is carried out in the Coq proof assistant. The properties required of the reduction semantics are gathered into module types, allowing to build a module containing reduction semantics along with the proofs of required proper-

ties. The transformation consists of a series of functors that implement different steps of transformation. Thus, one can derive an abstract machine automatically from the module implementing the reduction semantics by simply applying the sequence of functors to it.

Apart from the formalization of the basic refocusing transformation, we also consider its several extensions. We first treat two extensions proposed by Biernacka and Danvy [3,4]: a refocusing procedure for variants of calculi of closures leading to abstract machines with environments (rather than meta-level substitutions), and a refocusing procedure for context-sensitive reduction semantics used, e.g., for expressing languages with control operators such as `call/cc`. The final extension we present consists in adjusting the evaluation function to account for diverging computations as well as for terminating ones. In order to achieve this, we apply coinductive reasoning and we prove refocusing correct in that setting. As a result, we obtain a more faithful statement of equivalence between a reduction semantics and its corresponding abstract machine that covers both successful finite computations and infinite ones. All these extensions are formalized similarly to the basic refocusing transformation and are also provided in the Coq development. [1]

To summarize, the main contribution of this article is a generic formalization of the refocusing transformation along with its known extensions. The starting point of this formalization is a minimal axiomatization of reduction semantics allowing for an automatic derivation of its corresponding abstract machine. Machine-checked proofs of correctness for this derivation are provided. Another contribution is a new extension of refocusing that accounts for diverging computations via coinductive reasoning and trace-based semantics, together with a proof of its correctness.

References

1. The Coq Proof Assistant: `http://coq.inria.fr/`.
2. Małgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. In J. Giesl, editor, *Preliminary Proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*. Paris, France, 2007.
3. Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Logic*, 9(1):6, 2007.
4. Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, 2007.
5. Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the scheme programming language, part ii: Reduction semantics and abstract machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *Lecture Notes in Computer Science*, pages 186–206. Springer, 2009.

[1] The Coq development accompanying this article can be found at `http://fsieczkowski.com`.

6. Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer.
7. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Series RS-04-26, BRICS, Department of Computer Science, University of Aarhus, November 2004. iii+44 pp. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.
8. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 153–164. ACM, 2009.

Towards Strategies for Dataflow Programming

Joaquín Aguado and Michael Mendler

Faculty of Information Systems and Applied Computer Sciences,
University of Bamberg, Germany,
{joaquin.aguado,michael.mendler}@uni-bamberg.de

Abstract. A pure functional lazy language, like Haskell, provides powerful features for implementing important notions such as Haskell *Evaluation Strategies* and *Dataflow* (DF) programming styles. A natural step is to use strategies for DF applications. Nevertheless, there is an issue that, to our knowledge, has not been observed before in the literature. This is that present Haskell strategies infrastructure and DF like developments do not work properly together. In this paper, we propose a refinement of strategies for DF combinators, which does not rely on inductive constructions, but instead captures co-inductive data types (streams) and causality aspects of asynchronous DF directly.

1 Introduction

Haskell *Evaluation Strategies* (*Strategies* for shorth) are combinators that as such have all the advantages and properties of other lazy higher-order functions *e.g.*, clear semantics, compositionality, type safety, *etc.* In a nutshell, strategies describe a set of deterministic rules that provide an alternative (and ideally more effective) way of evaluating expressions. It has been shown that strategies are a powerful mechanism that can help to speedup the execution of programs by means of an elegant abstraction for separating *what* is computed (algorithm) from *how* the computation is carried out (dynamic behaviour) [1, 2].

On the other hand, computations are performed in time and space (memory) and controlled through programs which may introduce causality relations. The *Dataflow* (DF) abstraction takes each (memory) location as a *stream* (unbounded fifo buffer) of data changing over time and studies the functional and causal relationships between streams. The DF technology has turned into a highly successful approach in the design of embedded and reactive systems, in particular in the automotive and avionics industries [3]. In truth, we may argue that DF is well suited not only for engineering domains but also to describe a wide-range of market relevant applications since decision-support, user interfaces, interactive graphics and similar systems can be expressed (in a device independent fashion) as interchanges of data between processes.

A classical abstraction applied to represent DF models, known as *Kahn Process Network* (Kpn) [4], organises an application into a set of nodes connected

to one another through edges. Nodes represent sequential processes that transform data by performing computations and edges correspond to input and output streams for data and inter-process communications. On the positive side, HASKELL is an ideal platform for implementing DF constructs since the *lazy evaluation strategy* (which is also referred as the *default-strategy* in this paper) gives to the programmer the ability to process and manipulate (potentially) infinite streams. But this is not really new. A less common observation that deserves some attention is that all stream processing functions that the programmer can construct in HASKELL are in fact DF nodes since, in particular, these functions are sequential. For example, if we concretise streams as HASKELL infinite lists. Then, in this view, for some arbitrary unary function, say `(+1)`, we can think of `map (+1)` as a DF node able to apply `(+1)` in a point-wise fashion to all the elements of a stream. In other words, `map (+1)` would correspond to the lifted to streams version of `(+1)`. If so, the following DF feedback (recursive) network computes the infinite stream of natural numbers:

```
nats = 0 : map (+1) nats
```

There is, however, a fundamental conflict which, to our knowledge, has not been reported before in the literature. This is that the strategies, as they are defined, do not support properly programs that manipulate co-inductive data types like streams. But, can strategies change the behaviour of a DF program in such a way that it returns a different stream of values? In principle this is impossible because strategies provide only semantics preserving transformations. However, if we replace the `map (+1)` of `nats` by the (strategic function) parallel version of `map` that reduces (to *weak-head-normal-form*) each element in the list, then we will get:

```
nats_0_bot = 0 : parMap rwhnf (+1) nats_0_bot
```

This `nats_0_bot` after delivering the first element of the stream will loop unproductively for ever. Now, although both `nats` and `nats_0_bot` have the same semantics (*i.e.*, non-terminating denotation $\perp$), yet it is impractical to regard these DF networks as equivalent since, clearly, they exhibit a different observable behaviour. At this point, it is worth mentioning that the reasons for the `nats_0_bot` unproductive behaviour have more to do with the current design of strategies rather than with the (speculative) parallel nature of `parMap`, the strictness of `rwhnf` or the recursive construction (feedback) of the program. This unsatisfactory effect is hardwired inside the code of the strategies. In order to justify this point, let us define the strategic function `lazySeqMap`. The details of which will become clearer in Section 2. For now it is enough to indicate that `lazySeqMap` is a sequential (instead of parallel) version of `map` that lazily (instead of whnf) reduces the elements of a list.

```
lazySeqMap fun list = map fun list `using` seqList Lazy
```

Now, let us take as a reference the construction: `map (+1) [0,0..]`, which produces the constant stream of `1`'s and which, by the way, does not contain feedbacks (self references) at the top-level. Unfortunately, the observed behaviour

is also different in this case since `lazySeqMap (+1) [0,0..]` loops infinitely without producing even a single element of the stream.

The present paper starts investigations on the possibility of writing (and adapting) strategies for a DF style of programming and reports our experiences so far in this process. There are available two version of HASKELL strategies. The original one as described in [1] and a version developed recently for keeping a compositional approach and avoiding space leaks due to garbage collection (from GHC 6.10 forwards) [2]. In this paper we employ the latter. That is, the module `Control.Parallel.Strategies` (CPS) as in `parallel-2.2.0.1`.

2 Understanding Strategies

In principle, HASKELL strategies can be used to describe: (*i*) the *evaluation degree* of the different phases of a computation, (*ii*) the *possible parallelism* involved on it and (*iii*) the *order of execution* that must be followed [1].

2.1 Background

The evaluation degree refers to partially evaluated subexpressions or *thunks* during the course of a computation. The minimum degree represented by the completely-unevaluated object. All subsequent more elaborated evaluations are in *weak-head-normal-form* (whnf). The last level (which is also in whnf) corresponding to the completely-evaluated value is said to be in *normal-form* (nf). Evaluating an expression in HASKELL could mean reducing it to any of these levels. In the default-strategy, this is systematically approached by keeping down evaluation to the minimum possible degree (for the maximum possible amount of time) in order to complete a computation.

The two most basic combinators provided by the `Control.Paralle` module are `par, pseq :: a -> b -> b` [2, 5].

Intuitively, `par` indicates to the runtime system that it could (if possible) evaluate (reduce to whnf) the the first argument in parallel with the second argument. The resulting value of `par` is always that of the second argument. So, `x 'par' y` (in infix notation) has the same value as `y`. Then, for a given program, it is always possible to replace any construction of the form `x 'par' y` simply by `y` (or the other way around) without affecting its semantics. Combinator `par` *sparks* its first argument. What this means is that the first argument is stored as a thunk in the spark pool (an allocation area) and the main thread continues evaluating the second argument. Ideally, at some point in time, an idle processor may take this spark, creates a new thread end executes it. The `par` combinator is extensively used inside strategies because even though threads are created and scheduled dynamically, the semantics of a program remains deterministic. Note, however, that this parallelism is speculative in the sense that expressions might (or might not) be evaluated in parallel.

The combinator `pseq` forces, in the first place, the evaluation (reduction to whnf) of the first argument then, and only then, the evaluation of the second argument takes place. Therefore, the result of `pseq` is $\perp$ when the first argument is also $\perp$, otherwise the resulting value of `pseq` is that of the second argument. In the strategies the `pseq` combinator is used to control the sequencing of computations. But some care must be taken because the strictness imposed on the first argument together with the order of evaluation guarantee may lead to undesirable situations in which the response of a program may be altered.

For illustrating how `par` and `pseq` can be combined to improve performance, let us rework an example from [6]. Consider two intensive and independent integer functions `fib` and `sumEuler`. The objective is to add up the results computed by these functions from some values, say `38` and `5300`, respectively. This, of course, can be done by the following simple program:

```
fe1 = let (f,e) = (fib 38, sumEuler 5300) in f + e
```

Now, if both computations are going to be parallelised and the evaluation order of `+` could be arbitrary (due for example to some compiler optimisations) then the following construction would do the job:

```
fe2 = let (f,e) = (fib 38, sumEuler 5300)
      in f `par` (e `pseq` (f + e))
```

In `fe2` the computation of `fib` is sparked for (speculative) parallel execution with the main thread. On the other hand, the main thread takes the subexpression corresponding to the `pseq` combinator. Thus, the main thread executes first the computation specified by `sumEuler` since this is the first argument of `pseq`. If everything goes fine, the spark for `fib` will be converted by the runtime system and, then, it will be executed on another core in parallel with the computation of `sumEuler`. Finally, the main thread computes the addition of the two results.

2.2 Basic Mechanism of Strategies

All strategies have type `Strategy a`, which is a type synonym of `a -> Eval a`. The intuition is that Haskell strategies may perform some of the dynamic behaviour of expressions of type `a`. If so, the strategy will produce an object (of type `Eval a`) that will act as a proxy of a (possibly not yet known) result of type `a`. In this manner, the type `Strategy` is defined with the help of the algebraic data type `Eval` as follows:

```
type Strategy a = a -> Eval a
data Eval a = Seq a | Par a | Lazy a
```

The constructors `Seq`, `Par` and `Lazy` indicate how the resulting value of the computation must be obtained, particularly, when strategies are combined. Hence, `Eval` is treated as a monad and its instantiation is as follows:

```
instance Monad Eval where
   return = Lazy
   e >>= f = case e of
               Seq  x -> x `pseq` f x
               Par  x -> x `par`  f x
               Lazy x -> f x
```

The implementation of the injective function `return` (that makes a generic value of type `a` into a monad `Eval a`) simply uses the `Lazy` constructor. In general, from an object `e` of type `Eval a` and a monadic function `f` of type `a -> Eval b`, the *binding* operator, namely `>>=`, passes around the intermediate result wrapped inside `e` and applies `f` to it. In the context of strategies, this monadic function `f` will be typically of type `Strategy a`. So, the strategy (or monadic function) `f` will act on the intermediate (unknown) result of `e` and this latter will be computed (with respect to the application of `f`) either before (`pseq`), possibly at the same (`par`) or in the usual default lazy manner. Using the `Eval` monad, we can implement a new (not quite right yet!) version of `fe2` as follows:

```
fe3 = let (f,e) = (fib 38, sumEuler 5300)
      in Par f >>= \f' -> Seq e >>= \e' -> return (f' + e')
```

Moreover, as with any other monad, we can employ the very intuitive, yet convenient, do-notation for aligning vertically monadic computations. Here it is an equivalent version of `fe3` using this do-notation:

```
fe4 = let (f,e) = (fib 38, sumEuler 5300)
      in do f' <- Par f
            e' <- Seq e
            return (f' + e')
```

The above seems to be fine except that `fe3` and `fe4` are of type `Eval Int` whereas the type of `fe2` is `Int`. In other words, the result of `fe3` (`fe4`) is obtained from the function `return` that simply creates the monadic value `Lazy 47625790`. What we require is a way of getting an `Int` out of an `Eval Int` or, in general, to get a value of type `a` from a value of type `Eval a`. Nevertheless, that is the very thing a monad cannot do and there are good reasons for this. This does not mean that such an operation cannot be implemented. Indeed, the module CpS contains a function `unEval :: Eval a -> a` coded up in the obvious manner and for precisely this purpose. This sort of operation is in general considered harmful and should not be used unless some proof obligations regarding its timing relative to all other operations (of the same monad) are complied. For the Haskell strategies is highly advisable not to use directly `unEval`, but instead rely on the function `using` which evaluates a value employing a given strategy which is safer.

```
using :: a -> Strategy a -> a
using x s = unEval (s x)
```

Going back to our working example, the dynamic behaviour discussed so far can be encapsulated by the following code:

```
stg :: Strategy (a,a)
stg (f,e) =
   do f' <- Par f
      e' <- Seq e
      return (f',e')
```

Note that `stg` works on pairs where each component corresponds to a computation. The intentions is that `stg` performs the two computations in parallel, and returns the two respective results inside a pair. What to do with these results is leaved to the function that employs `stg`. Now, let us annotate the code of `fe1`, which contains the algorithmic core of our problem, to get:

```
fe5 = let (f,e) = (fib 38, sumEuler 5300) `using` stg in f + e
```

In general terms, this is the main design idea behind strategies. In addition, the compiled code has been run in a multi-core machine and the execution traces have been observed using the ThreadScope profiler tool [2]. As expected, programs `fe2` and `fe5` exhibit similar parallel activity and its execution times are in the same order which is, roughly speaking, just above half of the time taken by `fe1`. So, in this example, the main objective of strategies has been also fulfilled, namely to improve performance.

The module CPS includes the following. The basic strategies related to evaluation degree, namely: (*i*) `r0` (or `Lazy`) that evaluate lazily its argument, (*ii*) `rwhnf` (or `Seq`) that reduces its argument (to whnf) as far as the topmost constructor concerns and (*iii*) `rdeepseq` that fully evaluates (to nf) its argument. Regarding parallelism and order of execution, the most elemental strategies that can be employed to evaluate an argument in parallel is `rpar` (or `Par`) and the most elemental strategies for imposing order of execution is `Seq`.

3 Working with Strategies

The module CPS contains a number of combinators that provide support for tuples and the general class of traversable structures (`Data.Traversable`). Streams, or for the case unbounded lists, are the particular instance of `Traversable` in which we are interested. But, before discussing in more detail these strategies, let us turn our attention to DF programming.

3.1 Writing Dataflow Programs

For the purpose of illustrating how strategies can be defined for DF, we consider a very simple kernel DF language (code-named DFL just for reference), which is based on the core operators of languages such as LUSTRE and LUCID SYNCHRONE. In DFL we have expressions which define flows of values. The abstract syntax of DFL is in Figure 1.

$$
\begin{array}{lll}
e ::= k & & \text{constant of a definable type } k \in \mathbb{T}_1 \cup \mathbb{T}_2 \ldots \cup \mathbb{T}_n \\
\mid\ b & & \text{boolean constant } b \in \mathbb{B} \\
\mid\ x & & \text{flow variable} \\
\mid\ \texttt{uop}\ e & & \text{unary operator } \texttt{uop} \in \{\neg, +1, \ldots\} \\
\mid\ e\ \texttt{bop}\ e & & \text{binary operator } \texttt{bop} \in \{+, \wedge, \ldots\} \\
\mid\ e\ \texttt{fby}\ e & & \text{initialised delay} \\
\mid\ \texttt{rec}\, x.\, e & & \text{recursive flow}
\end{array}
$$

Fig. 1. DFL Syntax

As it stands, the syntax basically models a first-order flow algebra over the discrete data domain $\mathbb{D} = \mathbb{B} \cup \mathbb{T}_1 \cup \mathbb{T}_2 \ldots \cup \mathbb{T}_n$. In the HASKELL implementation of DFL we also have functional abstraction $\lambda x.\, e$ to build higher-order processes and arbitrary user-defined data types. We do not make use of these here. Now, a closed expression e computes a stream $[\![e]\!] \in \mathbb{D}^\infty$. Since we are not interested in the scheduling of a DF expression but only its limit stream behaviour, we use the standard denotational Kahn semantics for DFL primitives as in Figure 2. Note that all functions are total stream processing functions where the empty flow is denoted by ϵ. Moreover, these stream processing functions are continuous in domain $(\mathbb{D}^\infty, \leq)$ and thus can be used to define the semantics of arbitrary feedback KPN. Specifically, recursion can be mapped to the least fixed point operator by: $[\![\texttt{rec}\, x.\, e]\!] =_{\text{df}} \mu(\lambda x.\, [\![e]\!])$.

$$
\begin{array}{lll}
[\![k]\!] & =_{\text{df}}\ k \cdot [\![k]\!] & \\
[\![b]\!] & =_{\text{df}}\ b \cdot [\![b]\!] & \\
[\![\texttt{uop}]\!]\ s & =_{\text{df}}\ \epsilon & \text{if } s = \epsilon \\
[\![\texttt{uop}]\!]\ v \cdot s & =_{\text{df}}\ (\texttt{uop}\ v) \cdot [\![\texttt{uop}]\!]\ s & \\
[\![\texttt{bop}]\!]\ s_1\ s_2 & =_{\text{df}}\ \epsilon & \text{if } s_i = \epsilon,\ i = 1, 2 \\
[\![\texttt{bop}]\!]\ (v_1 \cdot s_1)\ (v_2 \cdot s_2) & =_{\text{df}}\ (v_1\ \texttt{bop}\ v_2) \cdot [\![\texttt{bop}]\!]\ s_1\ s_2 & \\
[\![\texttt{fby}]\!]\ \epsilon\ s_2 & =_{\text{df}}\ \epsilon & \\
[\![\texttt{fby}]\!]\ (v_1 \cdot s_1)\ s_2 & =_{\text{df}}\ v_1 \cdot s_2 & \\
[\![\texttt{rec}\, x.\, e]\!] & =_{\text{df}}\ [\![e\{\texttt{rec}\, x.\, e/x\}]\!]. &
\end{array}
$$

Fig. 2. DFL Recursive Kahn Semantics

The implementation of DFL as a set of HASKELL programming combinator is pretty much straightforward and can be done in various manners. One possibility is presented in Figure 3.

DFL considers streams as primitive types which are employed for representing continuous input and output for processes. Streams are defined as lists in which `[]` (the empty list constructor) is never used. In this characterisation every definable HASKELL type can be lifted to streams. It is worth noticing that all

```
type Stream a = [a]

lift0 :: a -> Stream a
lift0 x = x : (lift0 x)

lift1 :: (a -> b) -> Stream a -> Stream b
lift1 f (x:xs) = (f x) : (lift1 f xs)

lift2 :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
lift2 f (x:xs) (y:ys) = (f x y) : (lift2 f xs ys)

fby :: Stream a -> Stream a -> Stream a
fby (x:_) ys = x : ys
```

Fig. 3. DFL Implementation in HASKELL

stream processing functions that we could construct in HASKELL are DF nodes (Kahn processes) because they are sequential. This is a consequence of the deterministic nature of the default-strategy (reduction in the lazy λ-calculus).

The static lifting of constants and operators embeds standard functions canonically to operate point-wise on streams. The unit delay or unit memory buffer `fby` s_1 s_2 (read "*follow by*") computes a stream in which the first value is the same as that of s_1 and the rest is exactly s_2.

In DFL, as in LUSTRE and LUCID SYNCHRONE, a system S is codified as a finite set of DF equations $S = \{x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n\}$ for some distinct variables $x_1, x_2, \ldots, x_n$. These specify a system of simultaneous and mutually recursive fixed-points on flows. Since the feedback construct `rec` $x.\, s$ iterates s through variable x, then `rec` $x.\, e$ is expressed in terms of a DF equation $x = e$. This permits easy specification of feedback (recursive) DF networks. Thus, in DFL we switch from `rec` $x.\, e$ to the equational presentation $x = e$ and use directly the mechanism of recursion provided by HASKELL. For instance, in DFL the stream of natural numbers can be codified as follows:

```
nts :: Str Integer
nts = let s_zeros = (lift0 0); s_plus1 = (lift1 (+1))
      in s_zeros `fby` (s_plus1 nts)
```

The `fby` takes the first element `0` from `s_zeros` and delivers it as the first element of `nts`. The next element in `nts` is obtained from `s_plus1 nts`. Since `0` is already there, then the second element of `nts` is `1`. Similarly, the third element is derived from `s_plus1 nts`. Since `1` is the second element in `nts`, then the third stream element is `3`. This infinite recursive process continues in the same fashion.

3.2 List Strategies

The combinator `parList :: Strategy a -> Strategy [a]` is the most representative (regarding strategies) of the list type. As it may be expected, `parList` traverses a list sparking each of its elements that, in turn, should be reduced accordingly to the given strategy. Concretely, `parList` has the following algorithmic shape[1]:

```
parList strat []     = return []
parList strat (x:xs) = do y  <- Par (x `using` strat)
                          ys <- parList strat xs
                          return (y:ys)
```

Nevertheless, `parList` turns stream functions into unproductive ones. But why? Let us see, take the constant stream of `1`'s and modify its evaluation accordingly to `parList rwhnf`:

```
ones_bot = (lift0 1) `using` parList rwhnf
```

After desugaring and performing some reductions `y` and `xs` are respectively bound to the expressions: `1 `using` rwhnf` and `lift0 1`, and we get:

```
y `par` (parList strat xs >>= \ys -> return (y:ys))
```

Trouble is that the application of the binding operator `>>=` forces the reduction of `parList strat xs` in order to match it with one of the constructors of `Eval`. Thus, after some more reductions we obtain:

```
y `par`
    ((y' `par` (parList strat xs' >>= \ys' -> return (y':ys')))
     >>= \ys -> return (y:ys))
```

where `y'` and `xs'` are respectively bound to `1 `using` rwhnf` and `lift0 1`. Continuing with the unfolding in the same fashion, we will get longer and longer expressions of the form:

```
y `par` (y' `par` (y'' `par` ...) >>= \ys -> return (y:ys))
```

Since co-inductive objects (*e.g.*, streams) do not have base (*e.g.*, empty) constructors, `ys` gets never instantiate and this process continues indefinitely without returning ever the topmost construction `(y:ys)` making, in this form, `ones_bot` not only unbounded but also infinitely unproductive.

Certainly, other CpS combinators like `seqList` and `parListChunk` present the very same difficulty discussed above. The former traverses a list sequentially evaluating each element with the given strategy. So, its unfolding would result basically in the same pattern as that of `parList` but on which `pseq` replaces `par`. The latter sparks fixed-size and independent parts (chunks) of the list. Thus,

[1] This is not the actual implementation of `parList` but an equivalent one used here for illustration.

`parList` is really a special case of `parListChunk` but in which the chunks are of unit size. However, CPS includes also `parListN` and `parBuffer` that are better fit to deal with streams. Let us discuss these in more detail.

The strategy `parListN :: Int -> Strategy a -> Strategy [a]` sparks the first n^{th} element in a list and, from then on, the evaluation continues in the default (lazy) manner. More specifically, `parListN` takes an arbitrary large but finite part of a stream and manipulates this as `parList` would do it but it leaves untouched the rest of the infinite stream. It is not hard to see that `parListN` would fix `ones_bot` by replacing the `parList rwhnf` by, say `parListN 100 rwhnf`, where the `100` is just one possibility, indeed this value could be any finite natural number. Moreover, this strategy would also work for `nts` as follows:

```
nts1 = nts `using` parListN 100 rwhnf
```

where again the `100` is just one representative of an immense number of possibilities. If that were all, this would be the end of the story, but it is not! Let us try to annotate the DF network `nts` from inside as follows:

```
nts2 = let s_zero = lift0 0; s_plus1 = lift1 (+1)
       in (s_zero `fby` s_plus1 nts2) `using` parListN 100 rwhnf
```

It will not work. Well, except if we replace the input argument `100` by either `0` or `1`. None of which is really interesting. The former makes the evaluation to proceed in the default lazy manner, so nothing is gained. The latter sparks just the first element, so really it does not parallelise that much neither. Fair enough, but why are we making a fuss out of this, if after all `nts1` does the job? Firstly, the power of `parListN` is limited in the sense that only a finite part (arbitrary large as it can be) of the stream is manipulated by the strategy and the reader must agree that this could be an issue for some domain applications with continuous unbounded interaction (*e.g.*, web services). Imagine one of such system running with a certain performance for some time that suddenly slows down when the change of strategies occurs. Secondly and more important, it is a matter of abstraction and compositionality. For instance, let us take the following two versions of `(+1)` lifted to streams:

```
pls stream =  lift1 (+1) stream
slp stream = (lift1 (+1) stream) `using` parListN 100 rwhnf
```

Both of which independently provide identical results when executed with the same stream. In principle, in a compositional approach, there will be no harm in replacing one version for the other in any context. Now take:

```
nts3 = let s_zero = lift0 0 in s_zero `fby` pls nts3
nts4 = let s_zero = lift0 0 in s_zero `fby` slp nts4
```

In this case, `nts3` produces the desired sequence while `nts4` becomes unproductive after delivering the first element.

Now, let us have a look at `parBuffer`. The first point we would like to recall is that `parBufffer :: Int -> Strategy a -> [a] -> [a]` and `parMap` are

(strategic) functions rather than strategies. This means that they are constructed on the top of strategies and, therefore, they are not applied or combined in the usual strategic manner of previous examples. In particular, they do not bring back objects encapsulated in the `Eval` type, but the objects themselves. Initially, `parBuffer` sparks the first n elements of the list, subsequently it sparks more elements as the first elements of the list are consumed. Proceeding as before, we can apply `parBuffer` to the computation of, say of the stream of `1`'s:

```
ones1 = parBuffer 100 rwhnf (lift0 1)
```

Nevertheless, using `parBuffer` inside `lift0` will bring us some troubles, and the following:

```
lift0_parBuffer x = x : parBuffer 100 rwhnf (lift0' x)
ones2 = lift0_parBuffer 1
```

will not work unless we replace the buffer size `100` by `0`.

3.3 Stream Strategies

At a first sight, the problem that strategies face when dealing with DF code seems to be the lack of base constructors. Of course, this problem is related to the issue since (among other things) the unfolding of streams using, say `parList`, never reaches the topmost constructor, namely `(y:ys)`. So, one approach that can be taken is to develop combinators that manipulate only finite parts of streams. This will lead us to strategies similar to `parListN` (with or without base constructors) and its associated undesired effects regarding streams. Instead, the code corresponding to the base case could be leaved as it is (in case the strategy operates on lists) but the construction of the resulting object (of type `Eval [a]`) should be done in a less strict manner. This can be done by the following strategy:

```
parStrm strat []     = return []
parStrm strat (x:xs) = do y  <- Par  (x `using` strat)
                          ys <- Lazy (xs `using` parStrm strat)
                          return (y:ys)
```

Strategy `parStrm` operates productively on both lists and streams. The reason is that now an expression such as: `ones3 = (lift0 1) `using` parStrm rwhnf` will be unfolded in such a way that `y` and `ys` are bound, respectively, to the subexpressions `1 `using` rwhnf` and `(lift0 1) `using` parStrm strat`. Then, the strategy will `return` an object of the form:

```
Lazy((1 `using` rwhnf):((lift0 1) `using` parLS strat)
```

which, in turn, will be converted by the `using` combinator into the (ready-to-use) stream : `(1 `using` rwhnf):((lift0 1) `using` parStrm strat)`. Moreover, `parStrm` will spark the thunk corresponding to `(1 `using` rwhnf)`. In this form, this strategy sparks each of the elements of a stream (list) while the

structure is traversed. So far, our experiments have shown that the performance gains induced by both `parList` (when used on lists) and `parStrm` (when used on streams) are of the same order. As an illustration, Figure 4 presents the logs resulting from: `test = sum $ take 20` ***input***, for two different versions of ***input***, namely:

```
-- parList version
input1 = map (fib) [20..39] `using` parList rwhnf

-- parStrm version
input2 = let s = (lift0 20) `fby` (lift1 (+1) s)
         in lift1 fib s `using` parStrm rwhnf
```

```
SPARKS: 20 (19 converted, 0 pruned)

INIT  time  0.02s  ( 0.05s elapsed)
MUT   time 14.80s  ( 9.23s elapsed)
GC    time  0.17s  ( 0.09s elapsed)
EXIT  time  0.00s  ( 0.00s elapsed)
Total time 14.98s  ( 9.38s elapsed)
```

(a) `parList` version

```
SPARKS: 20 (20 converted, 0 pruned)

INIT  time  0.02s  ( 0.00s elapsed)
MUT   time 15.00s  ( 9.28s elapsed)
GC    time  0.09s  ( 0.08s elapsed)
EXIT  time  0.00s  ( 0.00s elapsed)
Total time 15.11s  ( 9.36s elapsed)
```

(b) `parStrm` version

Fig. 4. Runtime System Statistics of `test`

As it is mentioned above, we can use `parStrm` in the context of lists. For example, `map (fib) [20..39]` and `map (fib) [20..39] `using` parStrm rwhnf` will produce the same list. This, however, will be not case for `map (fib) [20..]` and `map (fib) [20..] `using` parList rwhnf`. The latter is unproductive.

It is not hard to see that more stream strategies can be designed (in the same fashion) based on the corresponding list strategies counterparts. Here it is, to give one example, `pStN` (the stream version of `parListN`):

```
pStN 0 _     xs     = return xs
pStN _ strat []     = return []
pStN n strat (x:xs) = do y  <- Par  (x `using` strat)
                         ys <- Lazy (xs `using` pStN (n-1) strat)
                         return (y:ys)
```

On top of all this, strategies like `parStrm` and `pStN` do not affect the compositionality of systems. For example, the following:

```
slp1 stream = (lift1 (+1) stream) `using` parStrm rwhnf
slp2 stream = (lift1 (+1) stream) `using` pStN 100 rwhnf
```

```
nts5        = let s_zero = lift0 0 in s_zero `fby` slp1 nts5
nts6        = let s_zero = lift0 0 in s_zero `fby` slp2 nts6
```

will do the job that `nts4` (in combination with `slp`) was unable to realise.

4 Conclusions

The main objective of the strategies concept has been centered on improving the performance of otherwise correct code. However, such an endeavour would have been very hard to realise outside an adequate design framework. This framework, we claim, has been structured around two fundamental *design conditions*, which relate subexpressions with the strategies that annotate them. These conditions have been so far: (*i*) Strategies must preserve (in a compositional manner) the denotational semantics of the subexpressions and (*ii*) strategies must not impose any modification (particularly algorithmic) other than annotating the subexpressions. For DF programming, as this paper shows, condition (*i*) is too weak. Instead, we have suggested an approach for constructing strategies based on a new design condition: (*i*) Strategies must preserve (in a compositional manner) the observed behaviour of programs. This has lead us to produce strategies that support DF while keeping the clean modular design of functional programming. Although, at this stage of the research, we have leave the treatment of performance very brief and for further benchmarks. At the same time, we have attempted to keep our strategies as closely related as possible to the existing strategies infrastructure (interface consistency). In this form, we believe, strategies for DF can take advantage of the already tested technology which, in turn, may induce execution performance gains.

References

1. Trinder, P., Hammond, K., Loidl, H.W., Jones, S.P.: Algorithm + Strategy = Parallelism. Journal of Functional Programming **8** (1998) 23–60
2. Marlow, S., Jones, S.P., Singh, S.: Runtime support for multicore haskell. (In: The 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09))
3. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages twelve years later. In: Proceedings of the IEEE, Special Issue on Embedded Systems. Volume 91. (2003) 64–83
4. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing: Proceedings of the IFIP Congress '74, Stockholm, Sweden, North-Holland Publishing Company (1974) 471–475
5. Jones, S.P., Gordon, A., Finne, S.: Concurrent haskell. In: Proceedings of the 23rd ACM-SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'96), New York, NY, USA (1996) 295–308
6. Jones, S.P., Singh, S.: A tutorial on parallel and concurrent programming in haskell. (Microsoft Research Cambridge – Draft)

Extensible Pattern Matching in an Extensible Language (Extended Abstract)

Sam Tobin-Hochstadt

Northeastern University

Abstract. Pattern matching is a widely used technique in functional languages, especially those in the ML and Haskell traditions. Although pattern matching is typically not built into languages in the Lisp tradition, it is often available via libraries built with macros. We present a sophisticated pattern matcher for Racket, which extends the language using macros, supports novel and widely-useful pattern-matching forms, and is itself extensible with macros.

1 Extending Pattern Matching

The following Racket[1] [3] program finds the magnitude of a complex number, represented in either Cartesian or polar form as a 3-element list:

```
(define (magnitude n)
  (cond [(eq? (first n) 'cart)
         (sqrt (+ (sqr (second n)) (sqr (third n))))]
        [(eq? (first n) 'polar)
         (second n)]))
```

While this program accomplishes the desired purpose, it's far from obviously correct, and commits the program to the list-based representation. Additionally, it unnecessarily repeats accesses to the list structure making up the representation. Finally, if the input is `'(cart 7)`, it produces a hard-to-decipher error.

In contrast, the same program written using pattern matching is far easier to understand:

```
(define (magnitude n)
  (match n
    [(list 'cart x y) (sqrt (+ (sqr x) (sqr y)))]
    [(list 'polar r theta) r]))
```

The new program is shorter, more perspicuous, does not repeat computation, and produces better error messages.

The function can also be easily converted to arbitrary-dimensional coordinates:

```
(define (magnitude n)
  (match n
    [(list 'cart xs ...) (sqrt (apply + (map sqr xs)))]
    [(list 'polar r theta ...) r]))
```

[1] Racket is the new name of PLT Scheme.

This definition is much improved from the original, but it still commits us to a list-based representation of coordinates. By switching to custom, user-defined pattern matching forms, this representation choice can be abstracted away:

```
(define (magnitude n)
  (match n
    [(cart xs ...)
     (sqrt (apply + (map sqr xs)))]
    [(polar r theta ...) r]))
```

Our custom pattern matching form can use other features of Racket's pattern matcher to perform arbitrary computation, allowing us to simplify the function further by transparently converting Cartesian to polar coordinates when necessary:

```
(define (magnitude n)
  (match n
    [(polar r theta ...) r]))
```

In the remainder of the paper, we describe the implementation of all of these examples, focusing on user-extensibility. We begin with a history of pattern matching in Scheme, leading up to the current implementation in Racket, touching briefly on the adaptation of standard techniques from ML-style matching [4] and their implementation via macros [2,1]. Then we describe the implementation of sequence patterns (seen above with the use of `...`) and other pattern forms not found in conventional pattern-matching systems. Third, we describe how to make patterns user-extensible by exploiting the flexibility of Racket's macro system.

References

1. Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proceedings of the Eighth Workshop on Scheme and Functional Programming*, 2007.
2. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
3. Matthew Flatt and PLT. Reference: Racket. Reference Manual PLT-TR2010-reference-v5.0, PLT Scheme Inc., June 2010. `http://racket-lang.org/techreports/`.
4. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM.

Where are you going with those types?

Vincent St-Amour[†], Sam Tobin-Hochstadt[†], Matthew Flatt[*], Matthias Felleisen[†]

[†]Northeastern University, [*]University of Utah
{stamourv,samth,matthias}@ccs.neu.edu, mflatt@cs.utah.edu

Abstract. The use of unboxed data representations often increases the efficiency of programs, especially numerical ones. Operations on unboxed data do not need to mask out type tags or dereference pointers. As a result, certain operations, such as floating point arithmetic, become a single machine instruction.

Type-based techniques [1–3] directly enable the use of unboxed data representations in the presence of polymorphic functions. The problem is, however, that type information, computed in the front end, has to be carried through the entire compilation process, all the way to the code-generation phase, where unboxing decisions are made. Doing so increases the complexity of the compiler and results in a significant overhead in compilation time.

This work presents an alternative approach. Instead of pushing types through the compiler, it relies on exposing specialized primitives that operate on unboxed data of a given type. The typechecker then rewrites programs to use these primitives where it is safe to do so. That is, when there exists a specialized equivalent to the original primitive that can operate on values of the arguments' types, the type checker replaces the generic primitive with an equivalent that allows unboxing.

For example, Racket provides specialized arithmetic primitives that are only valid for floating-point numbers: `fl+`, `fl-`, etc. Typed Racket's typechecker replaces generic arithmetic primitives—regular Racket `+`, `-` and so on—with their specialized equivalents if it can prove that their arguments are floating-point numbers. That is, the type checker employs rewriting rules such as the following:

$$\texttt{(+ x y)} \rightarrow \texttt{(fl+ x y)} \quad \text{if } \Gamma \vdash x : \texttt{Float} \text{ and } \Gamma \vdash y : \texttt{Float}$$

We have implemented this approach as part of the Typed Racket [4, 5] system. Preliminary results show speedups comparable to the traditional typed-based approaches. We conjecture that this approach could be applied to other typed languages with generic operations such as Haskell and Standard ML.

References

1. Leroy, X.: Unboxed objects and polymorphic typing. In: Symposium on Principles of Programming Languages. (1992) 177–188
2. Shao, Z., Appel, A.W.: A type-based compiler for Standard ML. In: Conference on Programming Language Design and Implementation. (1995) 116–129
3. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: Conference on Programming Language Design and Implementation. (1996) 181–192
4. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: Symposium on Principles of Programming Languages. (2008) 395–406
5. Culpepper, R., Tobin-Hochstadt, S., Flatt, M.: Advanced macrology and the implementation of Typed Scheme. In: Workshop on Scheme and Functional Programming. (2007) 1–13

Purity in Erlang*

Mihalis Pitidis and Konstantinos Sagonas

School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
mpitid@gmail.com kostis@cs.ntua.gr

Abstract. Motivated by a concrete goal, namely to extend Erlang with the ability to employ user-defined guards, we developed a parameterized static analysis tool called PURITY, that classifies functions as referentially transparent (i.e., side-effect free with no dependency on the execution environment and never raising an exception), side-effect free with no dependencies but possibly raising exceptions, or side-effect free but with possible dependencies and possibly raising exceptions. We have applied PURITY on a large corpus of Erlang programs and report experimental results showing the percentage of functions that the analysis definitely classifies in each category. Moreover, we discuss how our analysis has been incorporated on a development branch of the Erlang compiler in order to allow extending the language with user-defined guards.

Keywords: purity, static analysis, Erlang

1 Introduction

Purity plays an important role in functional programming languages as it is a cornerstone of *referential transparency*: the same language expression produces the same value when evaluated twice. Referential transparency helps in writing easy to test, robust and comprehensible code, makes equational reasoning possible and aids program analysis and optimization. In pure functional languages like Clean or Haskell, any side-effect or dependency on the state is encapsulated by the type system and is reflected in the types of functions. In a language like ERLANG, which has been developed primarily with concurrency in mind, pure functions are not the norm and impure functions can freely be used interchangeably with pure ones. Still, even in these languages, being able to reason about the purity of functions can prove useful in various situations.

This extended abstract discusses properties that functions must satisfy in order to be classified as having a certain level of purity, and describes the design and implementation of a fully automatic parameterized static analyzer, called PURITY, that determines the purity of ERLANG functions. Although the analysis is quite simple and ultra conservative, we were able to determine the purity of roughly 90% of the functions in the code bases we tested.

* This is an extended abstract intended for inclusion to the IFL'2010 draft proceedings.

As a practical application, our analysis has been integrated in a development branch of the ERLANG compiler, allowing functions that the analyzer determines as pure to be used in guard expressions, something not previously possible in ERLANG and also a very frequent user request for extending the language. Furthermore, our analysis could make way for some types of optimisations in the ERLANG compiler (common subexpression elimination, useless call elimination, automatic parallelization, etc.).

To make this extended abstract relatively self-contained, the next section reviews the ERLANG language and aspects of its evolution and implementation which are relevant to the topic we discuss. Section 3 describes the analyses we employ to determine the purity of ERLANG functions, followed by a section (Sect. 4) that presents experiences from running these analyses on a large corpus of ERLANG libraries and applications. Section 5 describes how the analysis information can be used to allow for user-defined guards in ERLANG, and the paper ends with reviewing purity in other languages (Sect. 6) and some concluding remarks.

2 Erlang: The Language and its Features

ERLANG is a concurrent functional programming language with dynamic types. What sets ERLANG apart from other functional languages is its support for concurrency, fault tolerance and distributed programming. Other notable features include hot-code reloading whereby the code of some module of an executing ERLANG program can be replaced with a newer version of that module without interrupting the program's execution. The language also provides soft real-time guarantees.

The aforementioned features make ERLANG ideal for building highly scalable, reliable and robust systems. While initially conceived to develop software for telecommunication systems, ERLANG has outgrown this particular niche and with the advent of the multi-core era it is being used for the development of a growing number of diverse software applications. This includes web and chat servers, distributed document stores, network servers and more.

To achieve these goals, ERLANG employs a mixture of purely functional programming — in the form of immutable data structures and single assignment — combined with a limited set of impure functions and expressions, in order to support concurrency and distribution. In particular, ERLANG implements the *actor* model of concurrency. Its implementation can be summarised as share-nothing concurrency based on lightweight processes communicating via asynchronous message passing. This helps express complex concurrency schemes in a more natural and declarative manner.

Impurities in ERLANG originate in particular expressions and functions. An example of the first case is the `receive` expression which is used to extract messages from the mailbox of a process. With regard to the second case, we first need to mention the general concept of built-in functions, or BIFs as they are usually known in the ERLANG community. BIFs are functions native to the

Erlang virtual machine, implemented in the language the VM is written in, in this case C. Besides primitive operations which otherwise cannot be expressed in pure Erlang, BIFs often substitute commonly used functions for optimisation purposes. As it happens, many BIFs are impure, usually because they interface with the runtime system in various ways.

Like many functional languages, Erlang supports pattern matching, a way of matching a sequence of values against a corresponding sequence of patterns. The result, if successful, is a mapping of variables from the pattern to the various terms in the sequence of values. Pattern matching plays a central role in expressing control flow in Erlang. Additional constraints can be placed on pattern matches with the use of guard tests. Guard tests consist of boolean expressions which are evaluated for each pattern matched and only if their result is true will the match be successful. With guards it is possible to extend the expressiveness of pattern matching significantly, adding support for value ranges for numbers, or tests for abstract types like process identifiers, references and function objects.

Erlang however imposes certain limitations on guard tests. Specifically, they must lack side-effects and execute in bounded time, preferably constant. To this end, they are limited by the Erlang language to a predefined set of built-in functions also known as guard BIFs [1, § 6.20, p. 103].

The example in Figure 1 showcases the use of pattern matching in Erlang and how it can be further extended with guard tests. The end result is concise and declarative code, a combination which often boosts programmer productivity significantly.

```
area({square, Side}) when is_integer(Side) ->
  Side * Side;
area({circle, Radius}) when is_number(Radius) ->
  3.14 * Radius * Radius;     %% well, almost
area({triangle, A, B, C}) ->
  S = (A + B + C) / 2, math:sqrt(S * (S-A) * (S-B) * (S-C)).
```

Fig. 1. Examples of pattern matching and guard tests

As mentioned in the beginning of this section, Erlang is dynamically typed. Furthermore, the compiler does not perform any form of type analysis. This has certain implications on Purity, which are discussed in more detail in Section 3.

3 Purity Analysis in Erlang

In order to determine the purity of Erlang functions we designed and implemented a fully automated static analyzer which operates on Erlang source code. The analysis is flexible and allows the user to select between different purity criteria, depending on the intended use of the analysis' results.

3.1 Flavours of Purity

We should first clarify what we mean by pure and impure functions. A *pure* function is one that is *referentially transparent*, i.e., it can be replaced by its return value without changing the semantics of the program in any way. This is the strongest definition of purity. Our analysis can chose between a few progressively weaker criteria, depending on the intended use of its results.

In general, a function may lose its referential transparency and be classified as impure: a) either due to *modifying* the execution environment in some way other than returning a value, or b) by *depending* on the environment of execution in some way other than its arguments.

A function that falls into the first category is said to have side-effects. Such a function will always be considered impure by our analysis. Regarding the second category, the user may choose to ignore such violations of referential transparency and still consider functions which fall in it as pure. We shall elaborate on this design decision at a later section.

Besides the fundamental categories described above, our analysis distinguishes between yet another condition of purity, one that is specific to Erlang. This condition concerns *exceptions*, which represent a non-local return out of a function and are somewhat problematic in their classification. First of all, it is not clear whether exceptions break referential transparency. While we can no longer replace the function by its value, we can still replace it by an exception raising expression, preserving the semantics of the program. This is not the whole truth however, since exceptions usually carry context sensitive information, specifically the series of function calls leading up to them, otherwise referred to as a *stack trace*. In the case of Erlang, this is not much of an issue, since exceptions are regular Erlang terms, and can be pattern matched on. The stack trace may or may not be part of the exception value, depending on the expression used to *catch* it. The older `catch` expression converts exceptions to tuples which often contain this stack trace, so using `catch` will break referential transparency. The newer and more robust `try-catch` construct however, does not capture the stack trace, which is otherwise available through a specific Erlang built-in function, aptly named `get_stacktrace()` [2]. So, in the absence of a call to this function, `try-catch` blocks can be considered pure.

Still, when it comes to certain applications of the analysis' results — such as optimisations like common subexpression elimination — it makes sense to consider exceptions impure altogether. This is why our analysis is flexible in this respect as well.

A final note regarding exceptions concerns the semantics of process termination in Erlang. If a process is terminated by an exception which is not a member of the `exit` class, then this event is reported by the Erlang runtime system to the error logging service [2, § 2.7]. We choose to ignore this potential side-effect since it does not directly influence the execution of the program, but primarily because we wish to maintain a conceptual separation between exceptions and side-effects. This is important, as we will see in Section 5 that exceptions can be safely ignored in certain contexts.

To sum up, when using the analysis we will describe in the rest of this section, one can select between three progressively stronger criteria of purity, namely considering a function as impure if it: 1) contains side-effects (the default), 2) has dependencies on the environment (lack of determinism) and 3) possibly raises an exception.

3.2 The Analysis

Our analysis is relatively straightforward. It operates on ERLANG modules which are first compiled to CORE ERLANG[1], and consists of two distinct stages. The first stage collects necessary information by traversing the *Abstract Syntax Tree*. This information consists of a list of dependencies for each function, and besides function calls includes impure expressions like `receive`. Along with each dependency, a small context is kept to facilitate the analysis stage later on. Essentially, these dependency lists represent the equivalent of a call graph.

The second stage, the core of the analysis, is responsible for converting each function's list of dependencies to a value representing its purity. This is achieved by means of an iterative process which is completed when a fixed point is reached. The process starts by selecting the subset of functions in the call graph whose purity is predetermined. The purity of each member of this initial set is then propagated to the set of functions which depend on that member, following this simple rule: if impure, the dependent function is contaminated and becomes impure too; if pure, this particular function is removed from any dependency lists. Whenever a function's dependency list becomes empty, this function can be safely marked pure. The process is repeated, with the initial set becoming the set of functions whose purity was just conclusively determined.

The final product of our analysis is a lookup table which maps functions to their purity. The key is a triple of *module*, *function* and *arity*, while the value can be either a concrete result, i.e., true or false, or a list of dependencies, which means the analysis was inconclusive.

At this point it is important to note that our analysis is conservative in regard to pure functions: no false positives are allowed, while false negatives are tolerated. It follows that any function whose purity has not been conclusively determined is considered impure.

3.3 Higher Order Functions

Higher order functions, i.e., functions that return other functions or accept functions as arguments, are common in functional programming languages. Considering the latter type of higher order functions, if a call is made to one of the arguments in the body of the higher order function, it follows that its purity depends on that argument, and cannot be resolved to a fixed value. The only exception to this is when the function depends on other *impure* functions as well.

[1] CORE ERLANG is an intermediate, simpler representation of ERLANG source code, more suitable as a starting point for static analysis.

```
%% A higher order function which depends on its first argument.
fold(_Fun, Acc, []) -> Acc;
fold(Fun, Acc, [H|T]) -> fold(Fun, Fun(H, Acc), T).

%% A pure closure is passed to a higher order function
%% so function g1/0 is pure as well.
g1() -> fold(fun erlang:'*'/2, 1, [2, 3, 7]).

%% An impure closure is passed to a higher order function
%% so function g2/0 is classified as impure.
g2() -> fold(fun erlang:put/2, computer, [ok, error]).
```

Fig. 2. An example of a higher order function

Let us consider the example of a higher order function, `h`, which just makes a call to its first argument. Clearly this has an unfixed purity. But what can be said about a function `g`, which depends on `h`? This function would either be a higher order function itself, taking another function as argument, and passing it along to `h`, or it would pass a *concrete* function `f`, as argument to `h`. It is thus possible in the second case —assuming the purity of `f` is known— to resolve the purity of this specific instance of `h` and consequently that of `g`. Figure 2 shows such an example.

This is a fairly simple example, but it manages to capture the most common use of higher order functions in Erlang. Another important case, albeit a less frequent one, has to do with higher order functions which do not call their arguments directly, passing them instead to other higher order functions. This way, multiple levels of indirection are present between a call with a concrete function as argument and the actual higher order function which will end up using it. This is better illustrated by way of an example, like the one in Figure 3. To detect such cases and analyse them correctly in the absence of type information, some kind of data flow of data flow analysis is required. The current implementation has some limitations in this area which we hope will be worked out in future versions. Regardless, such cases account for less than 10% of the functions in the code bases we examined.

Another limiting factor is the fact that functions may be passed as parts of more complex data structures instead of directly as arguments. Common cases include, but are not limited to lists, tuples and records. In fact, most of these cases require runtime information in order to be properly resolved.

3.4 Implementation Aspects

Some aspects relating to the implementation of Purity deserve further elaboration. The most important of these aspects is the way the analysis is bootstrapped, in other words the way we obtain the initial set of functions whose purity is predefined.

```
%% One level of indirection: it is not apparent this is a higher
%% order function since no direct call to its argument is made.
fold1(Fun, Acc, Lst) ->
  fold(Fun, Acc, Lst).

%% Two levels of indirection. The function argument has also
%% changed position.
fold2(Lst, Fun) ->
  fold1(Fun, 1, Lst).

g3() -> fold1(fun erlang:put/2, ok, [computer, error]).

g4() -> fold2([2, 3, 7], fun erlang:'*'/2).
```

Fig. 3. An example of indirect higher order functions

This set includes all functions built-in to the ERLANG runtime system. Since these are implemented in C instead of ERLANG, they cannot be analysed. Therefore, it was necessary to extract them from the ERLANG runtime system and hard-code their purity. The values assigned were derived from their semantics, not the actual implementation.

Beyond this, it is possible to bootstrap the analysis with a more generalised mechanism, the *persistent lookup table* or PLT for short. The PLT is used to store all the information necessary to repeat the analysis as well as cached versions of the analysis results for a given set of modules. This way, the user does not have to re-analyse every library his application depends on. The PLT also plays an important role in contexts were only one module can be analysed at a time but information regarding functions in other modules is necessary.

4 Experiences

In the course of testing our implementation diverse code bases were analysed providing some insight as to the current practices of ERLANG programmers. The applications analysed were primarily high profile open source projects:

Erlang/OTP The latest open source ERLANG distribution. Among other things, it includes the ERLANG bytecode and native code compilers, the standard library, static analysis tools like DIALYZER, an XML parsing library, and the Open Telecom Platform with its various networking applications.

Wings3D A subdivision modeler, used for generation of polygon models in computer graphics.

CouchDB A distributed, fault-tolerant and schema-free document oriented distributed database system.

ejabberd A server for the Extensible Messaging and Presence Protocol (XMPP), an open standard used primarily for instant messaging.

Yaws A high performance HTTP 1.1 server.
ibrowse An HTTP 1.1 client, also a dependency of Yaws.
erlssom Another XML parsing library and dependency of Yaws.
purity The analyzer described in this paper.

Table 1 includes further information about each application, while Table 2 presents the results of the analysis with the default options. Tables 3 and 4 present alternate runs of the analysis with progressively stronger purity criteria.

Table 1. Details of analysed applications

Application	Version	Modules	Functions	LOC
Erlang/OTP	R14A	1,901	120,807	739,010
Wings3D	1.2	168	9,507	78,996
ejabberd	2.1.4	149	5,168	53,881
CouchDB	0.11.0	97	2,502	22,938
Yaws	1.88	42	1,560	19,438
erlsom	1.2.1	18	568	9,562
ibrowse	1.6.1	7	226	2,683
purity	0.2	13	389	2,687

Table 2. Analysis results with only side-effects impure

Application	Pure	Impure	Undefined	Limited	Time
Erlang/OTP	43.8%	41.4%	1.3%	13.5%	4:01
Wings3D	54.5%	34.7%	1.4%	9.4%	0:26
ejabberd	39.0%	51.1%	5.9%	4.0%	0:15
CouchDB	43.4%	44.4%	2.3%	10.0%	0:07
Yaws	44.5%	46.5%	1.7%	7.3%	0:07
erlsom	46.0%	9.2 %	0.5%	44.4%	0:04
ibrowse	43.8%	56.2%	0.0%	0.0%	0:02
purity	66.3%	21.6%	1.5%	10.5%	0:05

The columns of Table 2 labeled Pure and Impure are self-explanatory. The Undefined column represents the percentage of functions which cannot be analysed statically. These include functions like `erlang:apply(M, F, Args)` which applies function `F` in module `M` to some argument list of terms `Args`. Since this list can be of any length we cannot know the exact arity of the function being called at compile time. The percentage also includes functions which depend on such functions or on functions whose source code was not available during the analysis. The Limited column represents the percentage of functions which could not be conclusively analysed because of limitations in our implementation. Finally, the last column shows the CPU time required to analyse each application, as reported by the `erlang:statistics/0` function.

Table 3. Analysis results with side-effects and non-determinism impure

Application	Pure	Impure	Undefined	Limited	Time
Erlang/OTP	37.1%	58.1%	0.8%	4.0%	3:55
Wings3D	44.8%	47.2%	1.2%	6.8%	0:25
ejabberd	33.8%	63.1%	1.8%	1.3%	0:14
CouchDB	41.2%	48.6%	1.3%	8.9%	0:07
Yaws	40.9%	51.6%	1.3%	6.2%	0:06
erlsom	38.4%	41.5%	0.5%	19.5%	0:04
ibrowse	38.9%	61.1%	0.0%	0.0%	0:02
purity	60.9%	27.8%	1.5%	9.8%	0:04

Table 4. Analysis results with side-effects, non-determinism and exceptions impure

Application	Pure	Impure	Undefined	Limited	Time
Erlang/OTP	5.3%	94.5%	0.1%	0.1%	3:45
Wings3D	5.6%	94.0%	0.3%	0.1%	0:21
ejabberd	9.4%	89.8%	0.5%	0.3%	0:12
CouchDB	6.2%	92.7%	0.4%	0.7%	0:06
Yaws	7.2%	92.3%	0.3%	0.3%	0:06
erlsom	3.9%	96.1%	0.0%	0.0%	0:02
ibrowse	6.2%	93.8%	0.0%	0.0%	0:02
purity	5.4%	93.6%	0.5%	0.5%	0:01

To better interpret the above results one should keep the following in mind: first of all, Erlang is primarily a concurrent language and is thus expected of most applications to make extended use of this impure feature. Furthermore, it only takes one impure function call to characterise all dependent functions as impure. Finally, reasoning about the purity of a specific function is not always straightforward for a programmer according to our experience. Consider for example a function like `filename:basename/1` which is part of the Erlang standard library. This function takes a filename and returns it with the leading path component removed, e.g., `filename:basename("/usr/bin/purity")` will return `"purity"`. This function is obviously used for the value it returns and since strings are lists in Erlang, we expect this function to perform some simple list manipulation operations. This is verified by taking a quick look at the actual source code. Most programmers would therefore consider its use consistent with programming in a purely functional style. It is however impure as our analysis —and some more careful consideration— demonstrates. The reason has to do with portability. In order for this function to be useful across different operating systems, its behaviour needs to vary according to the character used to separate paths in each one. It is thus dependent on the environment of execution and is not referentially transparent.

The results of Table 4 in particular appear disheartening. If one wishes to use the analysis results in contexts were exceptions cannot be regarded as pure, there is little one can gain from it. All hope is not lost however, since some of these results may be misleading. The reason so many functions appear to potentially

```
foo(42) -> ok;
foo(N) when is_integer(N) ->
  {error, N}.

bar(N) ->
  case foo(N) of
    ok -> ok;
    {error, _} -> error
  end.
```

(a) Code as written by programmer

```
foo(42) -> ok;
foo(N) when is_integer(N) ->
  {error, N};
foo(_) -> erlang:error(badarg)

bar(N) ->
  case foo(N) of
    ok -> ok;
    {error, _} -> error;
    _ -> erlang:error(case_clause)
  end.
```

(b) Code with exceptions inserted

Fig. 4. An example where the compiler fails to remove the redundant exception raising clauses that it has inserted due to lack of type information.

raise exceptions is that the Erlang compiler adds extra clauses at function definitions and catch expressions, which raise the corresponding clause failure exception if no pattern is matched. Later optimisation passes try to remove any such clauses which are redundant, when a function is total for instance, or when it takes no arguments. Without some form of type analysis however, it is not possible to safely remove such clauses in more complex cases. An example of an Erlang function which warrants such a clause is that of Figure 1. It is apparent from its definition that the `square` function does not cover all possible arguments it might be called with. On the other hand the exception raising clauses will not be removed for function `bar` in the example of Figure 4. The reason is that the compiler cannot determine that the pattern matching on the return value of the call to `foo` is complete, since it does not keep any information regarding `foo`'s return value.

Furthermore, these percentages do not account for the masking of exception by other functions. Consider the example in Figure 5 where an exception is raised by one function but is later masked in the body of another. With a more sophisticated analysis it is possible that some of these cases can be detected.

```
foo(X) ->
  throw(X).

bar() ->
  try foo(42) of
    Val -> {ok, Val}
  catch
    throw:E -> {error, E}
  end.
```

Fig. 5. An example of exception masking

5 Application: Adding User-Defined Guards

5.1 Motivation

As a direct application of our analysis, we set out to extend the ERLANG runtime and language with support for arbitrary pure function as guards. Such an extension further increases the expressiveness of the language and allows for more compact and descriptive code.

We mentioned in Section 2 that guard expressions are limited in ERLANG to a predefined set of built-in functions. The reason for this is that guard expressions require no observable side-effects and completion in bounded time. The careful reader might notice that the prerequisites do not mention determinism. In fact, valid ERLANG guard expressions include the `erlang:node/0` and `erlang:node/1` BIFs, which depend on the environment of execution. Specifically, the first function returns the name of the current ERLANG node, [2] while the second returns the name of the node a specific process belongs to. This name is subject to change from within the ERLANG runtime system by calling the `net_kernel:start/1` and `net_kernel:stop/0` functions. The example in Figure 6 shows an excerpt from a session in the ERLANG shell, illustrating how a guard expression might succeed on one call and fail on another.

```
1> F = fun () when node() =:= nonode@nohost -> error;
           () -> {ok, node()}
      end.
#Fun<erl_eval.6.13229925>
2> F().
error
3> net_kernel:start([test@localhost]).
{ok,<0.36.0>}
(test@localhost)4> F().
{ok,test@localhost}
(test@localhost)5> net_kernel:stop().
ok
6> F().
error
```

Fig. 6. Example of a non-deterministic guard expression

Another aspect of guards which has not been discussed yet, is that any exceptions which may be raised as part of a guard test are caught and converted to the value *false*. That is to say that even functions which may raise an exception during normal execution, will not do so when used as part of a guard expression.

[2] Taken from the ERLANG manual "A node is an executing Erlang runtime system which has been given a name"[3, ch. 12].

Since pursuing the development of such an extension was the primary motivation behind the development of our analysis, it should be clear by now why we chose to support differing criteria of purity. The guard in the previous example is not a referentially transparent function and a more strict analysis would reject it. Other valid guard tests, e.g. functions like `erlang:node/1` and `erlang:length/1`, would raise an exception when called with invalid arguments outside of guards. Obviously we did not want to break existing code with our extension.

In our opinion, one of the biggest advantages such an extension has to offer, is the fact that it enables custom tests for abstract data types in guards. The problem with ADTs in Erlang is that their structural information is exposed, as the language allows inspection through pattern matching and primitive type tests. Allowing arbitrary type tests as guards can help make code cleaner and could even discourage programmers from breaking ADT contracts. The example in Figure 7 illustrates the benefits of such an approach.

```
foo(Set) ->
  case gb_sets:is_set(Set) of
    true ->
      handle_gb_set(Set);
    false ->
      case sets:is_set(Set) of
        true ->
          handle_set(Set);
        false ->
          error
      end
  end.
```

(a) Custom tests not allowed as guards

```
foo(Set) when gb_sets:is_set(Set) ->
  handle_gb_set(Set);
foo(Set) when sets:is_set(Set) ->
  handle_set(Set);
foo(_) ->
  error.
```

(b) When user-defined guards are allowed

Fig. 7. Two ways of writing a function that operates on different term representations when user-defined tests are forbidden as guards 7(a) and when they are allowed 7(b). The code is not only more succinct, but it is also significantly more clear.

5.2 Prototype Implementation

To this end, we implemented a proof of concept on top of the Erlang compiler. Two distinct aspects of the compilation process have been be altered, while a third modification has been identified in the runtime system. First of all, an added compiler pass performing purity analysis is placed in the compiler front-end, just after the pass which converts Erlang source to Core Erlang. Additionally, errors regarding illegal guard expressions are silenced until the purity of the functions in question can be verified by looking up their values. Secondly, the compiler back-end is changed, specifically the code generation stage

up to the point where bytecode is produced. This is the trickiest part, since this phase of the compilation works on the assumption that only built-in functions might be called from within guard expressions. BIFs differ significantly from a regular ERLANG function call w.r.t. the bytecode that needs to be generated.

A third aspect we identified but did not implement has to do with the ERLANG loader. As mentioned earlier, ERLANG supports loading new code while the system is still running. It should be evident that a check must be placed at this stage, to verify that the same properties hold for the newly loaded code, specifically with regard to its purity.

Other engineering issues not addressed by our proof of concept have to do with optional user annotations of pure functions. Such annotations would make the programmer's intentions more explicit and could be further used by the code loader. Besides purity analysis, some type of conservative termination analysis would most likely be necessary to properly support user-defined functions as guards. If sufficient interest from the ERLANG community is present, we plan to formulate this extension as an Erlang Enhancement Proposal (EEP).

6 Purity in Other Languages

While we are not aware of any related work regarding ERLANG, purity concerns most functional languages to varying degrees. In most cases some sort of type system is employed and the purity of functions can be derived from their type signature. This sections examines some interesting approaches in more detail.

Clean is a general purpose, strongly typed, pure and lazy functional programming language. Clean handles side-effects and non-determinism by means of a uniqueness typing system [5, ch. 9]. This extends a traditional type system by allowing the user to specify that a given argument to a function is unique. Such an annotation guarantees the function will have private access to the argument, therefore destructively updating it will not violate the semantics of the function during the execution of the program. Besides side-effects, uniqueness typing can be used to convert pure operations to mutable state transformations without violating the pure semantics of the operation.

With a uniqueness guarantee it is trivial to verify referential transparency, as the function will never be called with the same argument. Furthermore, any side-effects of the function will never influence another function in an unforeseen manner [7].

Haskell is similar to Clean in many respects. It is purely functional, strongly typed and also features non-strict evaluation. However, Haskell takes a different approach with respect to purity, utilizing the more general concept of *monads*. Like Clean, this information is reflected in the type system and can be automatically inferred by way of a type inference scheme. Besides side-effects, monads can be used to express more general — and not necessarily impure — computations [4].

BitC was developed as a systems programming language with the goal of supporting formal verification. Unlike the languages previously mentioned, BitC is not purely functional. It does however support user level type annotations regarding the purity of functions, by means of an effect type system [6, ch. 10]. Such annotations associate expressions with an effect type variable which can have a value of *pure*, *impure* or *unfixed*. By verifying that certain parts of a program are pure, the BitC compiler can perform certain kinds of optimisations, like automatic parallelization.

7 Concluding Remarks

Having described the defining properties of pure functions according to three independent criteria: presence of side-effects, dependency on the environment of execution and possibility of raising an exception, we presented the design and implementation of a parameterized static analysis for determining such properties in the context of ERLANG. As a direct application of this analysis, we explored the possibility of enhancing the ERLANG language with user-defined guards and developed a suitable patch to the compiler which can be used as a prototype. To seriously consider such an extension however, further assurances for functions which are to be used in guard tests are necessary. In particular, some sort of termination analysis is required to satisfy the constraint that guard tests must evaluate in bounded time. Similarly, engineering issues, such as support for code reloading, need to be worked out as well.

In the course of testing our implementation, we analyzed diverse code bases, providing concrete data regarding the practices of ERLANG coders with respect to purity. The percentage of functions classified as pure by our analysis ranges on average between 30% and 50% for the applications we examined. This percentage is significant considering that ERLANG is primarily a concurrent language.

References

1. Barklund, J., Virding, R.: Erlang 4.7.3 reference manual (Feb 1999), `http://www.csd.uu.se/ftp/mirror/erlang/download/erl_spec47.ps.gz`
2. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlang's exception handling revisited. In: Proceedings of the ACM SIGPLAN Workshop on Erlang. pp. 16–26. ACM (2004)
3. Ericsson AB: Erlang Reference Manual User's Guide (Jun 2010), version 5.8, `http://www.erlang.org/doc/reference_manual/users_guide.html`
4. Newbern, J.: All about monads. `http://www.haskell.org/all_about_monads/`
5. Plasmeijer, R., van Eekelen, M.: Clean Language Report (Nov 2002), version 2.1, `http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf`
6. Shapiro, J., Sridhar, S., Doerrie, M.S.: The origins of the BitC programming language. `http://www.bitc-lang.org/docs/bitc/bitc-origins.html` (Apr 2008)
7. de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) Implementation of Functional Languages. LNCS, vol. 5083, pp. 201–218. Springer (2007)

Theory, Practice and Pragmatics of Fusion

Ralf Hinze, Thomas Harper, and Daniel W. H. James

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze,tom.harper,daniel.james@comlab.ox.ac.uk

Abstract. There are a number of related approaches for eliminating intermediate data structures in functional programs using fusion. These strategies are built upon using various but related recursion schemes, such as folds and unfolds. Using the concept of a *recursive coalgebra*, we attempt to expose the underlying mechanism that enables fusion in each of these techniques in a common theoretical and notational framework. We introduce the calculational properties of this theory and demonstrate its use with proofs and derivations in a calculational style. We then show various relationships among different fusion techniques and elucidate their syntactic transformations by showing them in a clear setting.

1 Introduction

Functional programmers love modular programs. It is easy to take simple functions and compose them to create clear, concise, and reusable code. Such programs commonly take the form of pipelines over lists. As an example, consider the following Haskell program:

$$f = sum \cdot map\ sq \cdot filter\ odd \cdot between$$

where *between* generates an enumeration between two natural numbers. Unfortunately, the clarity of this program comes at the cost of performance. The constituent functions of this program communicate with each other using intermediate data structures. Because these functions are recursive, even a compiler such as GHC [13] does not do any optimisation to try and remove these intermediate structures.

Such optimisation, however, is possible. We can produce a function that performs the same tasks as the one above without producing the intermediate data structure.

$$\begin{array}{l} f'\,(x,y) = go\,x \\ \quad \textbf{where} \\ \quad\quad go\,x \mid x > y \quad\;\; = 0 \\ \quad\quad\quad\;\; \mid otherwise = \textbf{if}\ odd\,x \\ \quad\quad\quad\quad\quad\quad\quad\;\; \textbf{then}\ sq\,x + go\,(x+1) \\ \quad\quad\quad\quad\quad\quad\quad\;\; \textbf{else}\;\; go\,(x+1) \end{array}$$

This new function loses the desirable qualities of the first one; the intention of the programmer is less clear and the concise, reusable code above has been hammered into a long, opaque and extremely specialised function. In doing so, however, we accomplish our goal of removing the intermediate data structures by combining the recursive of the traversals of the original program into a single one. This process of rewriting multiple functions into a single, equivalent function is called *fusion*.

While we can perform fusion by hand, as above, this quickly becomes infeasible for code of non-trivial length. Furthermore, it can be difficult to see all the opportunities for fusion in programs with complex functions. Instead, such a program transformation belongs in the realm of compile-time optimisations. Such transformations of recursive functions, however, are beyond the abilities of standard compiler tactics.

Programmers have developed methods of programming that make fusion easier to mechanise. These methods involve exposing the consumption and production of a data structure syntactically. Simple rules, usually in the form of GHC rewrite rules [7], can apply a syntactic transformation that removes intermediate data structures. Exposing the consumption and production of these structures requires encapsulating transformations using a fixed recursion scheme. The choice of recursion scheme is often either a *fold* or an *unfold*. A pipeline of folds or unfolds is then converted into a single fold or unfold. The underlying (co)algebras of these (un)folds, which are non-recursive, are then fused into a single (co)algebra by the compiler.

The implementation of these fusion techniques is often described syntactically, in terms of the combinators used and the rewrite rules used to transform them. This approach, however, obscures the underlying mechanism of fusion. This makes it difficult to prove the correctness of these approaches. It also makes it difficult to relate various fusion approaches to one another, despite the fact that such close relations exist. In this paper, we move fusion to a clearer setting, where the syntactic details of fusion fall away.

Category theory provides such a setting because the semantics of the recursion schemes use are often grounded in category theory. Although fusion techniques have been subject to this analysis before, such approaches have not been able to explain the inherent relationships between fusion techniques within the same framework. In this paper, we propose using *recursive coalgebras* as a such a setting. We will show how recursive coalgebras allow us to explain the fusion rules underlying the various fusion techniques and give short, simple proofs of correctness.

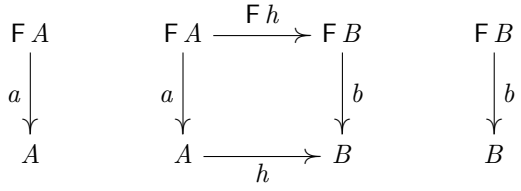
The rest of the paper is structured as follows. In Section 2 we review algebras, folds and the associated laws. In Section 3 we dualize this by introducing recursive coalgebras as a more advanced recursion scheme; we also give a proof relating algebras to recursive coalgebras. In Section 4 we generalize the calculational properties of fold and unfold. We present some basic examples in Section 5 to warm up for Section 6, where we apply our framework to *foldr*/*build* fusion (Sec-

tion 6.1), *destroy/unfoldr* fusion (Section 6.2) and stream fusion (Section 6.3). Finally, we summarise the related work and conclude.

2 Background: Algebras and Coalgebras

The category theory concept of an *initial algebra* is key in the theory of functional programming languages, specifically for giving a semantics to inductive datatypes [6]. These datatypes are defined as fixed points of functors and their interpretation as initial algebras of these functors. The remainder of this section will refresh the salient details. Appendix A serves as a further refresher of the categorical concepts that we will assume knowledge of: functors, natural transformations and subcategories.

Let $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ be a functor. An F-*algebra* is a pair $\langle a, A\rangle$ consisting of an object $A : \mathbb{C}$ and an arrow $a : \mathbb{C}(\mathsf{F}\,A, A)$. An F-*algebra homomorphism* between algebras $\langle a, A\rangle$ and $\langle b, B\rangle$ is an arrow $h : \mathbb{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\,h$.



The fact that functors preserve identity and composition entails that identity is a homomorphism and that homomorphisms compose. Consequently, F-algebras and F-algebra homomorphisms form a category, called $\mathbf{Alg}(\mathsf{F})$. We abbreviate an arrow in the category, $h : \mathbf{Alg}(\mathsf{F})(\langle a, A\rangle, \langle b, B\rangle)$, by $h : a \to b : \mathbf{Alg}(\mathsf{F})$ if the objects are obvious from the context, or simply by $h : a \to b$ if the functor F is also obvious.

If $\langle a, A\rangle$ is an F-algebra, then $\langle \mathsf{F}\,a, \mathsf{F}\,A\rangle$ is an F-algebra, as well. If $\langle a, A\rangle$ is an F-algebra, and $\mathfrak{a} : \mathsf{G} \dot{\to} \mathsf{F}$ is a natural transformation, then $\langle a \cdot \mathfrak{a}\,A, A\rangle$ is a G-algebra. Homomorphisms in $\mathbf{Alg}(\mathsf{F})$ are also homomorphisms in $\mathbf{Alg}(\mathsf{G})$:

$$h : a \cdot \mathfrak{a}\,A \to b \cdot \mathfrak{a}\,B : \mathbf{Alg}(\mathsf{G}) \quad \Longleftarrow \quad h : a \to b : \mathbf{Alg}(\mathsf{F}) \ . \tag{1}$$

The proof rests on the naturality of $\mathfrak{a}$.

If the category $\mathbf{Alg}(\mathsf{F})$ possesses an initial object, then we call it the *initial* F-*alegbra*, $\langle in, \mu\mathsf{F}\rangle$. Initiality means that there is a unique arrow from $\langle in, \mu\mathsf{F}\rangle$ to any other F-algebra $\langle a, A\rangle$. This unique arrow, called *fold*, is written $(\!|a|\!) : in \to a$. It is informative to consider the approach of using initial algebras as a semantics for datatypes. Loosely speaking, $(\!|a|\!)$ replaces constructors, the initial algebra in, by functions, given by the algebra a. Folds satisfy the following *universal property.*[1]

$$h = (\!|a|\!) \quad \Longleftrightarrow \quad h : in \to a \quad \Longleftrightarrow \quad h \cdot in = a \cdot \mathsf{F}\,h \tag{2}$$

[1] The formula $P \Longleftrightarrow Q \Longleftrightarrow R$ has to be read *conjunctively* as $P \Longleftrightarrow Q \wedge Q \Longleftrightarrow R$. Likewise, $P \Longleftarrow Q \Longleftrightarrow R$ is shorthand for $P \Longleftarrow Q \wedge Q \Longleftrightarrow R$.

The universal property has several important consequences.

Reflection law
Setting $h := id$ and $a := in$, we obtain $(\!|in|\!) = id$, the *reflection law*.

Computation law
Substituting the left-hand side into the right-hand side gives the *computation law*: $(\!|a|\!) : in \to a$, or expressed in terms of the base category $(\!|a|\!) \cdot in = a \cdot \mathsf{F}\,(\!|a|\!)$.

Fold fusion law
One of the most important consequences of the universal property is the the *fold fusion law* for fusing an arrow with a fold to form another fold.

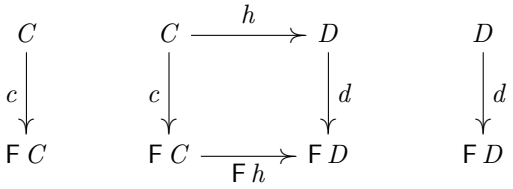
$$h \cdot (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h : a \to b \quad \Longleftrightarrow \quad h \cdot a = b \cdot \mathsf{F}\,h \tag{3}$$

The precondition requires h to be an F-algebra homomorphism from the algebra a to b.

There is an additional law that folds enjoy: a functor fusion law. To be able to formulate the it, we have to turn μ into a higher-order functor of type $\mathbb{C}^{\mathbb{C}} \to \mathbb{C}$. The object part of this functor maps a functor to its initial algebra. (This is only well-defined for functors that have an initial algebra.) The arrow part, which maps a natural transformation $\mathfrak{a} : \mathsf{F} \dot{\to} \mathsf{G}$ to an arrow $\mu\mathfrak{a} : \mathbb{C}(\mu\mathsf{F}, \mu\mathsf{G})$, is given by $\mu\mathfrak{a} = (\!|in \cdot \mathfrak{a}|\!)$. (To reduce clutter we have omitted the argument of $\mathfrak{a}$ on the right-hand side, which should read $(\!|in \cdot \mathfrak{a}\,(\mu\mathsf{F})|\!)$.) The *functor fusion law* states that we can fuse a fold after a map to form another fold: $(\!|b \cdot \mathfrak{a}|\!) = (\!|b|\!) \cdot \mu\mathfrak{a}$, for all $\mathfrak{a} : \mathsf{F} \dot{\to} \mathsf{G}$. Functor fusion follows from the definition of μ and fold fusion. Functoriality of μ is then an immediate consequence of reflection and functor fusion.

Finally, the initial algebra $\mu\mathsf{F}$ is a fixed point of F — this is known as Lambek's Lemma [8]. One direction of the isomorphism $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ is given by in, its inverse is $in^\circ = (\!|\mathsf{F}\,in|\!)$. The proof is left as an instructive exercise to the reader.

An F-*coalgebra* is a pair $\langle C, c\rangle$ consisting of an object $C : \mathbb{C}$ and an arrow $c : \mathbb{C}(C, \mathsf{F}\,C)$. An F-*coalgebra homomorphism* between coalgebras $\langle C, c\rangle$ and $\langle D, d\rangle$ is an arrow $h : \mathbb{C}(C, D)$ such that $\mathsf{F}\,h \cdot c = d \cdot h$.



F-coalgebras and F-coalgebra homomorphisms also form a category, called $\mathbf{Coalg}(\mathsf{F})$. The dual of initial algebras are final coalgebras, however, we will *not* make use of these objects and instead focus on a restricted species of coalgebras.

3 Recursive Coalgebras

In this section we will introduce *recursive* coalgebras. We follow the work of Capretta *et al.* and their development in [2], where they motivate the use of recursive coalgebras as a structured recursion scheme.

A coalgebra $\langle C, c\rangle$ is called *recursive* if for *every* algebra $\langle a, A\rangle$ the equation in the unknown $h : A \leftarrow C$,

$$h \;=\; a \cdot \mathsf{F}\, h \cdot c \;\;,$$

has a *unique* solution. The unique solution is called a *hylomorphism* or *hylo* for short and is written $(\!|\, a \leftarrow c \,|\!)_{\mathsf{F}} : A \leftarrow C$. The notation is meant to suggest that h takes a coalgebra to an algebra. If the functor F is obvious from the context, we abbreviate $(\!|\, a \leftarrow c \,|\!)_{\mathsf{F}}$ by $(\!|\, a \leftarrow c \,|\!)$. Hylos satisfy the following universal property.

$$h = (\!|\, a \leftarrow c \,|\!) \quad \Longleftrightarrow \quad h = a \cdot \mathsf{F}\, h \cdot c \tag{4}$$

The category of recursive coalgebras and homomorphisms forms a full subcategory of $\mathbf{Coalg}(\mathsf{F})$, called $\mathbf{Rec}(\mathsf{F})$. If the category has a final object $\langle F, \mathit{out}\rangle$, then there is a unique arrow from any other *recursive* coalgebra $\langle C, c\rangle$ to $\langle F, \mathit{out}\rangle$. This arrow, called *unfold*, is written $[\!(c)\!] : c \to \mathit{out}$. It satisfies the following universal property.

$$h = [\!(c)\!] \quad \Longleftrightarrow \quad h : c \to \mathit{out} \quad \Longleftrightarrow \quad \mathsf{F}\, h \cdot c = \mathit{out} \cdot h \tag{5}$$

(This amounts to the usual property of unfolds, except that we are working in the category $\mathbf{Rec}(\mathsf{F})$, not in the category $\mathbf{Coalg}(\mathsf{F})$.) Just as in the case of folds, the universal property implies the *reflection law*, $[\!(\mathit{out})\!] = \mathit{id}$, the *computation law*, $[\!(c)\!] : c \to \mathit{out}$ or expressed in terms of the base category $\mathsf{F}\,[\!(c)\!] \cdot c = \mathit{out} \cdot [\!(c)\!]$, and the *unfold fusion law*:

$$[\!(c)\!] = [\!(d)\!] \cdot h \quad \Longleftarrow \quad h : c \to d \quad \Longleftrightarrow \quad \mathsf{F}\, h \cdot c = d \cdot h \;\;. \tag{6}$$

We have already seen that *in* has an inverse — Lambek's Lemma. If $\langle F, \mathit{out}\rangle$ is the final recursive coalgebra, then *out* has an inverse, as well. For the proof we require the following lemma.

Lemma 1. *Let $\langle C, c\rangle$ be a recursive F-coalgebra. Then $\langle \mathsf{F}\, C, \mathsf{F}\, c\rangle$ is also recursive.*

Proof. We have to show that $h = a \cdot \mathsf{F}\, h \cdot \mathsf{F}\, c$ has a solution, and furthermore, that it is unique. We shall accomplish this concurrently.

$$
\begin{aligned}
& h = a \cdot \mathsf{F}\, h \cdot \mathsf{F}\, c \\
\iff & \quad \{\ \mathsf{F} \text{ functor } \} \\
& h = a \cdot \mathsf{F}\, (h \cdot c) \\
\iff & \quad \{\ \text{logic and } f = g \Longrightarrow f \cdot c = g \cdot c\ \} \\
& h = a \cdot \mathsf{F}\, (h \cdot c) \quad \text{and} \quad h \cdot c = a \cdot \mathsf{F}\, (h \cdot c) \cdot c \\
\iff & \quad \{\ \text{universal property and assumption: } c \text{ is recursive } \} \\
& h = a \cdot \mathsf{F}\, (h \cdot c) \quad \text{and} \quad h \cdot c = (\!|\, a \leftarrow c \,|\!) \\
\iff & \quad \{\ \text{Leibniz } \} \\
& h = a \cdot \mathsf{F}\, (\!|\, a \leftarrow c \,|\!) \quad \text{and} \quad h \cdot c = (\!|\, a \leftarrow c \,|\!) \\
\iff & \quad \{\ \text{computation and assumption: } c \text{ is recursive } \} \\
& h = a \cdot \mathsf{F}\, (\!|\, a \leftarrow c \,|\!) \quad \text{and} \quad h \cdot c = a \cdot \mathsf{F}\, (\!|\, a \leftarrow c \,|\!) \cdot c \\
\iff & \quad \{\ \text{logic and } f = g \Longrightarrow f \cdot c = g \cdot c\ \} \\
& h = a \cdot \mathsf{F}\, (\!|\, a \leftarrow c \,|\!)\ \square
\end{aligned}
$$

Lemma 2. *A recursive coalgebra is final if and only if it is invertible:*

1. *If $\langle F, out \rangle$ is the final recursive coalgebra, then out is invertible with $out^\circ = [\!(\mathsf{F}\, out)\!]$.*
2. *If $\langle C, c \rangle$ is a recursive coalgebra and c is invertible, then $\langle C, c \rangle$ is final. Furthermore,*

$$[\!(d)\!] \;=\; (\!|\, c^\circ \leftarrow d \,|\!)\ .$$

Proof. 1. First of all, $[\!(\mathsf{F}\, out)\!]$ is well-defined, since $\mathsf{F}\, out$ is recursive. The isomorphism between out and $[\!(\mathsf{F}\, out)\!]$ follows from the functor and unfold laws. The proofs is left as an exercise.

2. We have to prove the universal property of unfolds (5) with $out := c$ and $[\!(d)\!] := (\!|\, c^\circ \leftarrow d \,|\!)$.

$$
\begin{aligned}
& h = (\!|\, c^\circ \leftarrow d \,|\!) \iff \mathsf{F}\, h \cdot d = c \cdot h \\
\iff & \quad \{\ \text{inverses } \} \\
& h = (\!|\, c^\circ \leftarrow d \,|\!) \iff c^\circ \cdot \mathsf{F}\, h \cdot d = h
\end{aligned}
$$

The latter equivalence is an instance of the universal property of hylos (4). $\square$

Theorem 1. *Initial F-algebras and final recursive F-coalgebras coincide:*

1. *If $\langle C, out \rangle$ is the final recursive F-coalgebra, then $\langle out^\circ, C \rangle$ is the initial F-algebra. Furthermore, $(\!|\, a \,|\!) = (\!|\, a \leftarrow out \,|\!)$.*
2. *If $\langle in, A \rangle$ is the initial F-algebra, then $\langle A, in^\circ \rangle$ is the final recursive F-coalgebra. Furthermore, $[\!(c)\!] = (\!|\, in \leftarrow c \,|\!)$.*

Proof. 1. By Lemma 2–1 *out* has an inverse. We have to prove the universal property of folds (2) with $in := out^\circ$ and $(\!|a|\!) := (\!|a \leftarrow out|\!)$.

$$\begin{array}{ll} & h = (\!|a \leftarrow out|\!) \iff h \cdot out^\circ = a \cdot \mathsf{F}\, h \\ \iff & \quad \{ \text{ inverses } \} \\ & h = (\!|a \leftarrow out|\!) \iff h = a \cdot \mathsf{F}\, h \cdot out \end{array}$$

The latter equivalence is an instance of the universal property of hylos (4).

2. We first have to show that $\langle A, in^\circ \rangle$ is recursive.

$$\begin{array}{ll} & h = a \cdot \mathsf{F}\, h \cdot in^\circ \\ \iff & \quad \{ \text{ inverses } \} \\ & h \cdot in = a \cdot \mathsf{F}\, h \\ \iff & \quad \{ \text{ universal property of folds (2) } \} \\ & h = (\!|a|\!) \end{array}$$

The statement then follows from Lemma 2–2 with $c := in^\circ$. □

4 Calculational Properties

In this section we will cover the calculational properties of our hylomorphisms. In a similar fashion to folds and unfolds, hylomorphisms have an identity law and a computation law, and similarly they follow from the universal property.

Identity law

$$(\!|a \leftarrow c|\!) = id \quad \iff \quad a \cdot c = id \tag{7}$$

Computation law

$$(\!|a \leftarrow c|\!) \;=\; a \cdot \mathsf{F}\, (\!|a \leftarrow c|\!) \cdot c \tag{8}$$

We have three fusion laws: algebra fusion, coalgebra fusion, and composition.

Algebra fusion

$$h \cdot (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!) \quad \Longleftarrow \quad h : a \to b \quad \iff \quad h \cdot a = b \cdot \mathsf{F}\, h \tag{9}$$

The precondition requires h to be an F-algebra homomorphism.

Proof.

$$\begin{array}{ll} & h \cdot (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!) \\ \iff & \quad \{ \text{ universal property of hylos (4) } \} \\ & h \cdot (\!|a \leftarrow c|\!) = b \cdot \mathsf{F}\, (h \cdot (\!|a \leftarrow c|\!)) \cdot c \end{array}$$

The obligation is discharged as follows:

$$\begin{aligned}
& h \cdot (\!| a \leftarrow c |\!) \\
= \quad & \{ \text{ hylo computation (8) } \} \\
& h \cdot a \cdot \mathsf{F}\,(\!| a \leftarrow c |\!) \cdot c \\
= \quad & \{ \text{ assumption: } h : a \to b \ \} \\
& b \cdot \mathsf{F}\,h \cdot \mathsf{F}\,(\!| a \leftarrow c |\!) \cdot c \\
= \quad & \{ \ \mathsf{F} \text{ functor } \} \\
& b \cdot \mathsf{F}\,(h \cdot (\!| a \leftarrow c |\!)) \cdot c \ \square
\end{aligned}$$

Coalgebra fusion

$$(\!| a \leftarrow c |\!) = (\!| a \leftarrow d |\!) \cdot h \quad \Longleftarrow \quad h : c \to d \quad \Longleftrightarrow \quad \mathsf{F}\,h \cdot c = d \cdot h \tag{10}$$

The precondition requires $h : c \to d$ to be an F-coalgebra homomorphism. The proof is the dual of that for algebra fusion.

Composition law

$$(\!| a \leftarrow c |\!) \cdot (\!| b \leftarrow d |\!) = (\!| a \leftarrow d |\!) \quad \Longleftarrow \quad c \cdot b = id \tag{11}$$

Composition is, in fact, a simple consequence of algebra fusion as $(\!| a \leftarrow c |\!) : b \to a$ is an F-algebra homomorphism.

$$\begin{aligned}
& (\!| a \leftarrow c |\!) \cdot b \\
= \quad & \{ \text{ hylo computation (8) } \} \\
& a \cdot \mathsf{F}\,(\!| a \leftarrow c |\!) \cdot c \cdot b \\
= \quad & \{ \text{ assumption: } c \cdot b = id \ \} \\
& a \cdot \mathsf{F}\,(\!| a \leftarrow c |\!) \ .
\end{aligned}$$

Alternatively, we can derive the composition law from coalgebra fusion by showing that $(\!| b \leftarrow d |\!) : d \to c$ is an F-coalgebra homorphism. The composition law, together with the next law, generalises the functor fusion law of folds.

Hylo shift law *(or base change law)*

If we have a natural transformation $\mathfrak{a} : \mathsf{G} \dot{\to} \mathsf{F}$, then

$$(\!| a \cdot \mathfrak{a}\,A \leftarrow c |\!)_{\mathsf{G}} \ = \ (\!| a \leftarrow \mathfrak{a}\,C \cdot c |\!)_{\mathsf{F}} \ . \tag{12}$$

Proof. We have to show that $\mathfrak{a}\,C \cdot c$ is recursive.

$$\begin{aligned}
& h = a \cdot \mathsf{F}\,h \cdot \mathfrak{a}\,C \cdot c \\
\Longleftrightarrow \quad & \{ \ \mathfrak{a} \text{ natural } \} \\
& h = a \cdot \mathfrak{a}\,A \cdot \mathsf{G}\,h \cdot c \\
\Longleftrightarrow \quad & \{ \text{ universal property of hylos (4) } \} \\
& h = (\!| a \cdot \mathfrak{a}\,A \leftarrow c |\!)_{\mathsf{G}} \ \square
\end{aligned}$$

Note that the laws above are independent of the existence of initial algebras.

The *fold/unfold law* is a direct consequence of Theorem 1 and any of the fusion laws.

$$(\!|a|\!) \cdot [\!(c)\!] \;=\; (\!|a \leftarrow c|\!) \tag{13}$$

From left to right we are performing fusion and deforesting an intermediate data structure. From right to left we are turning a control structure into a data structure.

5 Examples

5.1 Warm-up: Type Functors

We have seen in Section 2 that μ is a functor, whose action on arrows is defined

$$\mu\mathfrak{a} \;=\; (\!|in \cdot \mathfrak{a}|\!) \;=\; (\!|in \cdot \mathfrak{a} \leftarrow out|\!) \;=\; (\!|in \leftarrow \mathfrak{a} \cdot out|\!) \;=\; [\!(\mathfrak{a} \cdot out)\!] \;.$$

Using Theorem 1 and the shift law we can express $\mu\mathfrak{a}$ both as a fold and as an unfold.

The parametric datatype List has the base functor

$$\mathbf{data}\,\mathsf{L}\;a\;b = \mathit{Nil} \mid \mathit{Cons}\,(a, b)$$

This is a higher-order functor of type $\mathsf{L} : \mathbb{C} \to \mathbb{C}^{\mathbb{C}}$ that takes object to functors and arrows to natural transformations. The list datatype is given in terms of its base functor by $\mathsf{List}\,A = \mu(\mathsf{L}\,A)$. The map function for lists is its action on arrows, defined by $\mathsf{List}\,f = \mu(\mathsf{L}\,f)$. Again, L takes an arrow f in $\mathbb{C}$ to a natural transformation in $\mathbb{C}^{\mathbb{C}}$, $\mathsf{L}\,f$. As noted above, $\mathsf{List}\,f$ can be expressed as both a fold and an unfold.

We shall see that μ is surprisingly useful when we encounter stream fusion in Section 6.3.

5.2 Program optimisations

Using algebras and recursive coalegbras, we can express certain programming optimisations. For example, the program

$$\mathit{sum} \cdot \mathit{between}$$

creates a list of integers and then sums them together. We can express this program as a fold after an unfold:

$$(\!|\mathfrak{s}|\!) \cdot [\!(\mathfrak{b})\!]$$

where

$$\begin{aligned}
&\mathfrak{s} : \mathsf{L}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N}\\
&\mathfrak{s}\;\mathit{Nil} \qquad\quad = 0\\
&\mathfrak{s}\,(\mathit{Cons}\,(x, y)) = x + y\\
&\mathfrak{b} : (\mathbb{N}, \mathbb{N}) \to \mathsf{L}\,\mathbb{N}\,(\mathbb{N}, \mathbb{N})\\
&\mathfrak{b}\,(m, n) = \mathbf{if}\; m > n\; \mathbf{then}\; \mathit{Nil}\; \mathbf{else}\; \mathit{Cons}\,(m, (\mathit{succ}\; m, n))
\end{aligned}$$

Note that $\mathfrak{b}$ has to be not only a coalgebra, but a recursive coalgebra.

Expressed thus, an intermediate list is used to build the sequence of numbers and then consumed to calculate the sum. It would instead be more efficient to write a program that sums the numbers as they are produced. The existence of such a transformation is given by the *fold/unfold law* which we can apply to obtain to obtain the equality

$$(\!|\mathfrak{s}|\!) \cdot [\![\mathfrak{b}]\!] = (\!|\mathfrak{s} \leftarrow \mathfrak{b}|\!)$$

In this case, the hylo corresponds to the recursive function

$$\begin{aligned} &f : (\mathbb{N}, \mathbb{N}) \to \mathbb{N} \\ &f\,(m, n) = \mathbf{if}\ m > n\ \mathbf{then}\ 0\ \mathbf{else}\ m + f\,(\mathit{succ}\,m, n)\ . \end{aligned}$$

This example is one of many cases where laws of algebras and recursive coalgebras correspond to program transformations in functional programming. We can take advantage of this correlation to provide a setting in which we can explain and analyse various program transformations. In particular, we use recursive coalgebras to explain various approaches to automated fusion techniques within the same framework.

6 Fusion

In the previous section, we have discussed fusion laws and shown their use as a proof strategy. These laws are not just useful tools for our proofs but also represent practical program transformations like the one in Section 5.2. In this section, we will use a combination of fusion laws for algebras, recursive coalgebras, and hylomorphisms to explain various approaches to optimising programs by eliminating intermediate data structures in recursive programs using fusion. By using recursive coalgebras as the setting for this analysis, we can explain and compare these approaches within the *same* framework. This allows us to more rigourously examine the relationships among these fusion approaches which are not readily apparent when examining their individual implementations.

6.1 Shortcut or *foldr/build* Fusion

The mother of all fusion rules is algebra-fusion. In order to use it, we derive a new algebra that fuses two algebras together. For example:

$$\mathit{sum} \cdot \mathit{filter}\ \mathit{odd}$$

expressed as folds

$$(\!|\mathfrak{s}|\!) \cdot (\!|\mathfrak{f}|\!)$$

where the algebra $\mathfrak{f}$ is given

$$\begin{aligned} &\mathfrak{f} : \mathsf{L}\,\mathbb{N}\,(\mu(\mathsf{L}\,\mathbb{N})) \to \mu(\mathsf{L}\,\mathbb{N}) \\ &\mathfrak{f}\,\mathit{Nil} &&= \mathit{in}\,\mathit{Nil} \\ &\mathfrak{f}\,(\mathit{Cons}\,(x, y)) &&= \mathbf{if}\ \mathit{odd}\,x\ \mathbf{then}\ \mathit{in}\,(\mathit{Cons}\,(x, y))\ \mathbf{else}\ y \end{aligned}$$

It it not hard to derive $\mathfrak{sf}$ so that $(\!|\mathfrak{s}|\!) \cdot \mathfrak{f} = \mathfrak{sf} \cdot \mathsf{F}\,(\!|\mathfrak{s}|\!)$.

$$\begin{aligned}
&\mathfrak{sf} : \mathsf{L}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N} \\
&\mathfrak{sf}\;\mathit{Nil} &&= \mathfrak{s}\;\mathit{Nil} \\
&\mathfrak{sf}\;(\mathit{Cons}\;(x, y)) &&= \mathbf{if}\;\mathit{odd}\;x\;\mathbf{then}\;\mathfrak{s}\;(\mathit{Cons}\;(x, y))\;\mathbf{else}\;y
\end{aligned}$$

Since $(\!|\mathfrak{s}|\!)$ replaces *in* by $\mathfrak{s}$, we simply have to replace the occurrences of *in* in $\mathfrak{f}$ by $\mathfrak{s}$. While this is an easy task to perform by hand, it is potentially difficult to mechanise.

The central idea of *foldr/build* fusion is to expose *in* so that replacing *in* by a is simple to implement. Consider algebra fusion again.

$$h \cdot (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h : a \to b$$

A fold $(\!|-|\!)$ is a transformation that takes an algebra to a homomorphism. Assume that we have another such transformation

$$h \cdot \mathfrak{B}\;a = \mathfrak{B}\;b \quad \Longleftarrow \quad h : a \to b \tag{14}$$

We can obtain such a natural transformation by creating a function whose argument is an algebra and which folds that algebra over our data structure. This is also known as the Church encoding of that structure.

The general form of shortcut fusion, the so-called *acid rain* [12] rule, is then

$$(\!|a|\!) \cdot \mathfrak{B}\;\mathit{in} \;=\; \mathfrak{B}\;a \;. \tag{15}$$

Using $\mathfrak{B}$ we expose *in* so that replacing *in* by a is achieved through a simple function application. Instead of building a structure and then folding over it, we eliminate the *in* and pass a directly to $\mathfrak{B}$. The proof of correctness is embarrassingly simple.

$$\begin{aligned}
&(\!|a|\!) \cdot \mathfrak{B}\;\mathit{in} = \mathfrak{B}\;a \\
=\;& \quad \{\ \text{assumption (14)}\ \} \\
&(\!|a|\!) : \mathit{in} \to a
\end{aligned}$$

Have we made any progress? After all, before we can apply (15), we have to prove (14). The fold $(\!|-|\!)$ satisfies this property, but this instance of (15) is trivial.

It turns out that in a *relationally parametric* programming language, the proof obligation (14) amounts to the free theorem [15] of the polymorphic type

$$\mathfrak{B} : \forall\, A\;.\;(\mathsf{F}\;A \to A) \to (B \to A)\;\;,$$

where B is some fixed type. In other words, the proof obligation can be discharged by the type checker.

$$(\!|\mathfrak{s}|\!) \cdot (\lambda\, a\;.\;(\!|\phi\;a|\!))\;\mathit{in} = (\!|\phi\;\mathfrak{s}|\!)$$

where

$$\begin{aligned}&\phi : (\mathsf{L}\,\mathbb{N}\,b \to b) \to (\mathsf{L}\,\mathbb{N}\,b \to b)\\ &\phi\,a\,\mathit{Nil} &&= a\,\mathit{Nil}\\ &\phi\,a\,(\mathit{Cons}\,(x,y)) &&= \mathbf{if}\ \mathit{odd}\,x\ \mathbf{then}\ a\,(\mathit{Cons}\,(x,y))\ \mathbf{else}\ y\end{aligned}$$

The reader should convince herself that $\lambda\,a\,.\,(\!|\phi\,a|\!)$ is of the right type. (Note in this regard that ϕ is polymorphic.)

The *acid rain* rule is unstructured in that hylos are hidden inside the abstraction $\lambda\,a$. Without performing any beta-reductions, we can apply the rule only once. We obtain a more structured rule if we shift the abstraction to the algebra and achieve *cata-hylo fusion:* If τ satisfies,

$$h : \tau\,a \to \tau\,b : \mathbf{Alg}(\mathsf{G}) \quad \Longleftarrow \quad h : a \to b : \mathbf{Alg}(\mathsf{F})\ , \tag{16}$$

then

$$(\!|a|\!)_{\mathsf{F}} \cdot (\!|\tau\,\mathit{in} \leftarrow c|\!)_{\mathsf{G}} \;=\; (\!|\tau\,a \leftarrow c|\!)_{\mathsf{G}}\ . \tag{17}$$

If τ is $\lambda\,a\ .\ a$, then this is just the fold/unfold law. For $\tau\,a = a \cdot \mathbb{Q}$, this is essentially functor fusion (note that (1) is an example of (16)). Again, the proof of correctness is straightforward.

$$\begin{aligned}&(\!|a|\!)_{\mathsf{F}} \cdot (\!|\tau\,\mathit{in} \leftarrow c|\!)_{\mathsf{G}} = (\!|\tau\,a \leftarrow c|\!)_{\mathsf{G}}\\ =\ &\quad \{\ \text{algebra fusion (9)}\ \}\\ &(\!|a|\!)_{\mathsf{F}} : \tau\,\mathit{in} \to \tau\,a : \mathbf{Alg}(\mathsf{G})\\ =\ &\quad \{\ \text{assumption (16)}\ \}\\ &(\!|a|\!)_{\mathsf{F}} : \mathit{in} \to a : \mathbf{Alg}(\mathsf{F})\end{aligned}$$

The proof obligation (16) once again amounts to a theorem for free, this time of the polymorphic type

$$\tau : \forall\,A\,.\,(\mathsf{F}\,A \to A) \to (\mathsf{G}\,A \to A)\ .$$

Using this rule, the running example simplifies to

$$(\!|\mathfrak{s}|\!) \cdot (\!|\phi\,\mathit{in}|\!) = (\!|\phi\,\mathfrak{s}|\!)$$

We can now also simplify a composition of folds:

$$(\!|a|\!) \cdot (\!|\tau_1\,\mathit{in}|\!) \cdot \cdots \cdot (\!|\tau_n\,\mathit{in}|\!) \cdot [\![c]\!] = (\!|(\tau_n \cdot \cdots \cdot \tau_1)\,a \leftarrow c|\!)$$

This demonstrates how the rewrite rule of *foldr/build* is able to achieve fusion over an entire pipeline of functions.

6.2 Shortcut or *destroy/unfoldr* Fusion

The *foldr/build* brand of shortcut fusion and its generalisation to arbitrary datatypes is fold-centric. This is limits the kinds of functions that we fuse, because some functions are not folds. We can dualise *foldr/build* fusion to achieve an unfold-centric approach, called *destroy/unfoldr* [11]. For example:

$$zip \cdot (between \times between)$$

expressed as unfolds

$$[\![\mathfrak{z}]\!] \cdot ([\![\mathfrak{b}]\!] \times [\![\mathfrak{b}]\!])$$

where

$$\begin{array}{l}
\mathfrak{z} : (\mu(\mathsf{L}\ a), \mu(\mathsf{L}\ b)) \to \mathsf{L}\ (a, b)\ (\mu(\mathsf{L}\ a), \mu(\mathsf{L}\ b)) \\
\mathfrak{z}\ (x, y) = \mathbf{case}\ (out\ x, out\ y)\ \mathbf{of} \\
\qquad\qquad \begin{array}{lll}
(Nil, & _ &) \to Nil \\
(_, & Nil &) \to Nil \\
(Cons\ (a_1, b_1), Cons\ (a_2, b_2)) & & \to Cons\ ((a_1, a_2), (b_1, b_2))
\end{array}
\end{array}$$

We can dualise the *acid rain* rule to fuse these functions. If

$$(\!|\mathfrak{b}\ c|\!) = (\!|\mathfrak{b}\ d|\!) \cdot h \quad \Longleftarrow \quad h : c \to d \tag{18}$$

then

$$(\!|\mathfrak{b}\ c|\!) \;=\; (\!|\mathfrak{b}\ out|\!) \cdot [\![c]\!] \ . \tag{19}$$

Similarly, we can dualise our more structured *cata-hylo fusion* fusion to achieve *hylo-ana fusion:* If τ satisfies,

$$h : \tau\ c \to \tau\ d : \mathbf{Rec}(\mathsf{G}) \quad \Longleftarrow \quad h : c \to d : \mathbf{Rec}(\mathsf{F}) \ , \tag{20}$$

then

$$(\!|a \leftarrow \tau\ c|\!) \;=\; (\!|a \leftarrow \tau\ out|\!) \cdot [\![c]\!] \ . \tag{21}$$

Beware: τ has to transform a *recursive* coalgebra into a *recursive* coalgebra! This is not guaranteed by the type alone.

This rule demonstrates how to fuse unfolds in a similar manner to our refinement of the *foldr/build* rule. However, the rules are not yet expressive enough to deal with our example; we have *two* producers to the right of *zip*. To fuse such a function, we need to employ *parallel hylo-ana fusion:* If τ satisfies,

$$h_1 \times h_2 : \tau\ (c_1, c_2) \to \tau\ (d_1, d_2) \quad \Longleftarrow \quad h_1 : c_1 \to d_1 \ \wedge \ h_2 : c_2 \to d_2 \ ,$$

then

$$(\!|a \leftarrow \tau\ (c_1, c_2)|\!) \;=\; (\!|a \leftarrow \tau\ (out, out)|\!) \cdot ([\![c_1]\!] \times [\![c_2]\!]) \ .$$

Using this rule, we are now able to fuse our example program:

$$[\![\zeta\,(out, out)]\!] \cdot ([\![\mathfrak{b}]\!] \times [\![\mathfrak{b}]\!]) = [\![\zeta\,(between, between)]\!]$$

where

$$\begin{aligned}
&\zeta : (b_1 \to \mathsf{L}\ a_1\ b_1, b_2 \to \mathsf{L}\ a_2\ b_2) \to (b_1, b_2) \to \mathsf{L}\,(a_1, a_2)\,(b_1, b_2)\\
&\zeta\,(c_1, c_2)\,(x_1, x_2) = \mathbf{case}\,(c_1\ x_1, c_2\ x_2)\ \mathbf{of}\\
&\qquad\qquad (\mathit{Nil}, \quad\ _ \quad\ ) \to \mathit{Nil}\\
&\qquad\qquad (_, \quad\ \mathit{Nil} \quad\ ) \to \mathit{Nil}\\
&\qquad\qquad (\mathit{Cons}\,(a_1, b_1), \mathit{Cons}\,(a_2, b_2)) \to \mathit{Cons}\,((a_1, a_2), (b_1, b_2))
\end{aligned}$$

6.3 Stream Fusion

The *foldr/build* fusion approach is inherently fold-centric, and *destroy/unfoldr* fusion inherently unfold-centric. We mentioned previously that *foldr/build* is limiting in requiring all programs to be expressed as folds. At first, it would appear the *destroy/unfoldr* is more flexible. For example, in Section 6.1 we expressed *filter* as a fold, but we can also express it as an unfold: $[\![\mathfrak{f}]\!]$ where

$$\begin{aligned}
&\mathfrak{f} : \mu(\mathsf{L}\,\mathbb{N}) \to \mathsf{L}\,\mathbb{N}\,(\mu(\mathsf{L}\,\mathbb{N}))\\
&\mathfrak{f}\,x = \mathbf{case}\ \mathit{out}\ x\ \mathbf{of}\\
&\qquad\quad \mathit{Nil} \qquad\quad \to \mathit{Nil}\\
&\qquad\quad \mathit{Cons}\,(x, y) \to \mathbf{if}\ \mathit{odd}\ x\ \mathbf{then}\ \mathit{Cons}\,(x, y)\ \mathbf{else}\,\mathfrak{f}\ y\ \ .
\end{aligned}$$

A crucial factor, however, is that the algebra $\mathfrak{f}$ is recursively defined. While this is not a problem in theory ($\mathfrak{f}$ is a recursive coalgebra — productivity is not an issue here), it is a problem in practice, as it defeats the purpose of the whole exercise; the algebra must be non-recursive for GHC to fuse it with others. Moreover, we need two definitions of *filter* for the two approaches. Is it possible to arrange things so that *filter* is both a fold and an unfold?

Perhaps surprsingly, the answer is yes. Let us first try to eliminate the recursion from the definition above — the rest will then fall out. The idea is to use a different base functor, one that allows us to express that we skip a list element. We draw inspiration from stream fusion[4] here:

$$\mathbf{data}\,\mathsf{S}\ a\ b = \mathit{Done} \mid \mathit{Yield}\,(a, b) \mid \mathit{Skip}\ b$$

The *filter* coalgebra can now be written as a composition of *out* with

$$\begin{aligned}
&\mathfrak{f} : \mathsf{S}\,\mathbb{N}\ b \to \mathsf{S}\,\mathbb{N}\ b\\
&\mathfrak{f}\,\mathit{Done} \qquad\quad = \mathit{Done}\\
&\mathfrak{f}\,(\mathit{Yield}\,(x, y)) = \mathbf{if}\ \mathit{odd}\ x\ \mathbf{then}\ \mathit{Yield}\,(x, y)\ \mathbf{else}\ \mathit{Skip}\ y
\end{aligned}$$

So, $\mathit{filter} = [\![\mathfrak{f} \cdot \mathit{out}]\!]$. Something interesting has happened: since $\mathfrak{f}$ is a natural transformation, we also have $\mathit{filter} = (\!|\mathit{in} \cdot \mathfrak{f}|\!)$. In other words, filter is both a fold and an unfold. Morever, it is an application of a mapping function: $\mathit{filter} = \mu\mathfrak{f}$.

In general, consumers are folds, transformers are maps, and producers are unfolds.

$$(\!|\, a \,|\!) \cdot \mu\mathfrak{a}_1 \cdot \cdots \cdot \mu\mathfrak{a}_n \cdot [\![c]\!] = (\!|\, a \cdot \mathfrak{a}_1 \cdot \cdots \cdot \mathfrak{a}_n \leftarrow c \,|\!)$$

Inspecting the types, the rule seems pretty obvious:

$$A \prec\!\!\xleftarrow{(\!|\, a \,|\!)} \mu\mathsf{F}_0 \prec\!\!\xleftarrow{\mu\mathfrak{a}_1} \mu\mathsf{F}_1 \quad \cdots \quad \mu\mathsf{F}_{n-1} \prec\!\!\xleftarrow{\mu\mathfrak{a}_n} \mu\mathsf{F}_n \prec\!\!\xleftarrow{[\![c]\!]} C$$

In a sense, the introduction of *Skip* keeps the recursion in sync. Each transformation consumes a token and produces a token. Before, *filter* possibly consumed several tokens before producing one. For a larger example, see appendix:

$$(\!|\, \mathfrak{p} \,|\!) \cdot \mu(\mathfrak{m}\ sq) \cdot \mu(\mathfrak{f}\ odd) \cdot [\![\mathfrak{b}]\!] = (\!|\, \mathfrak{p} \cdot \mathfrak{m}\ sq \cdot \mathfrak{f}\ odd \leftarrow \mathfrak{b} \,|\!)$$

It is important to note that the introduction of *Skip* is not specific to lists, but can be done for every datatype. A consumer typically has a case $Skip\ b \rightarrow b$; a transformer has a case $Skip\ b \rightarrow Skip\ b$; a producer typically doesn't produce a *Skip*. Although stream fusion is the first fusion system to make use of augmentation, we note its relation to Capretta's representation of general recursion in type theory [1], which proposed adding a "computation step" constructor to coinductive types. This relation is clear in the definitions of stream consumers, which use general recursion to consume *Skip*s.

7 Related Work

Wadler first introduced the idea of simplifying the fusion problem with his deforestation algorithm [16]. This was limited to so-called *treeless* programs, a subset of first-order programs. Wadler includes two extensions to this algorithm. The first is named *blazing*, and deals with basic types that do not need to be removed. The second extension allows some higher-order programs by treating them as macros. The fusion transformation proposed by Chin [3] generalizes Wadler's deforestation. It uses a program annotation scheme to recognize the terms that can be fused and skip the terms that cannot. Combined with Chin's prior work on higher-order removal, the transformation can take any first-order and higher-order program as input.

Sheard and Fegaras focus on the use of folds for algebraic types as a recursion scheme [10]. Their algorithm for normalizing the nested application of folds is based on the fold fusion law. Their work is suitable general to handled functions such as *zip* that recurse over multiple data structures simultaneously. Gill et al. first introduced the notion of short-cut fusion with *foldr/build* fusion [5] for Haskell. This allowed programs written as folds to be fused. It was subsquently introduced into the List library for Haskell in GHC. Takano and Meijer [12] provided a calculational view of fusion and generalised it to arbitrary data structures. It generalised the fusion law by using hylomorphisms. The dual of this fusion law, *destroy/unfoldr*, was hinted at but not studied. It was then

studied and implemented by Svenningson [11], who noted that *filter* and *zip* could be expressed this way. Svenningson did not, however, solve the issue of recursive algebras like *filter*, which would not be fused even though they could be written as unfolds. This was addressed by Coutts et al., who introduced stream fusion [4], which introduced the *Skip* constructor as a way to encode non-productive computation steps, similar to Capretta [1].

The correctness and generalisation of fusion has been explored in many different settings. In addition to the work of Takano and Meier, Ghani et al. generalised *foldr*/*build* to work with data types "induced by inductive monads". Johann and Ghani further showed how to apply initial algebra semantics, and thus *foldr*/*build* fusion, to nested datatypes [6]. Voigtländer has also used free theorems to show correctness, specifically of *destroy*/*build* rule [14].

8 Conclusions

We have brought three fusion techniques together and explicated them all within a single framework. We have exploited recursive coalgebras as 'the rug that ties the room together'. They enabled us to describe and reason about various fusion rules under the same notation. This served to remove the syntactic interference of the various fusion implementations.

The purpose of this more inclusive view is to be able to understand the relationship among various fusion techniques. In doing so, we have more formally expressed the implications of the relationship between the *foldr*/*build* and *destroy*/*unfoldr* techniques as well as the use of *Skip* in stream fusion to increase the expressiveness of the stream fusion technique. Doing so can provide a better understanding of these fusion techniques and how they may be applied in a more pragmatic setting. We have also shown how recursive coalgebras and the associated laws enabled us to write clear and concise correctness proofs about various fusion laws.

References

1. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1, 1–28 (2005)
2. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437 – 468 (2006), `http://www.sciencedirect.com/science/article/B6WGK-4JF97DV-1/2/201ba519595e41892408c1271f608302`, seventh Workshop on Coalgebraic Methods in Computer Science 2004
3. Chin, W.N.: Safe Fusion of Functional Expressions. In: LISP and functional programming. pp. 11–20. ACM, New York, NY, USA (1992)
4. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream Fusion: From Lists to Streams to Nothing At All. In: ICFP '07. pp. 315–326. ACM, New York, NY, USA (2007)
5. Gill, A., Launchbury, J., Peyton Jones, S.L.: A Short Cut to Deforestation. In: Functional programming languages and computer architecture. pp. 223–232. ACM, New York, NY, USA (1993)

6. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Typed Lambda Calculi and Applications. LNCS, vol. 4583, pp. 207–222. Springer-Verlag, Berlin, Heidelberg (2007), http://www.springerlink.com/index/10.1007/978-3-540-73228-0
7. Jones, S.P., Tolmach, A., Hoare, T.: Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In: Haskell Workshop. pp. 203–233. ACM SIGPLAN (2001), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1330
8. Lambek, J.: A fixpoint theorem for complete categories. Math. Zeitschr. 103, 151–161 (1968)
9. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, Springer-Verlag, Berlin, 2nd edn. (1998)
10. Sheard, T., Fegaras, L.: A Fold for All Seasons. In: Functional programming languages and computer architecture. pp. 233–242. ACM, New York, NY, USA (1993)
11. Svenningsson, J.: Shortcut fusion for Accumulating Parameters & Zip-like Functions. In: ICFP '02. pp. 124–132. ACM, New York, NY, USA (2002)
12. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Functional programming languages and computer architecture. pp. 306–313. ACM, New York, NY, USA (1995)
13. The GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.1, http://www.haskell.org/ghc/index.html
14. Voigtländer, J.: Proving correctness via free theorems: the case of the destroy/build-rule. ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation (2008)
15. Wadler, P.: Theorems for free! In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 347—-359. ACM, London (1989)
16. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science 73(2), 231 – 248 (1990), http://www.sciencedirect.com/science/article/B6V1G-45W3WV5-3K/2/2659383ba14773eb043709b0c6a981fe

A Category Theory Refresher

Functors Every structure comes equipped with structure-preserving maps, and for categories these maps are called *functors*. Since a category consists of two parts, objects and arrows, a functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ consists of a mapping on objects and a mapping on arrows. It is common practise to denote both mappings by the same symbol. The action on arrows has to respect the types: if $f : \mathbb{C}(A, B)$, then $\mathsf{F}\, f : \mathbb{D}(\mathsf{F}\, A, \mathsf{F}\, B)$. Furthermore, F has to preserve identity and composition:

$$\mathsf{F}\, id_A = id_{\mathsf{F}\, A} \ , \tag{22}$$

$$\mathsf{F}\, (g \cdot f) = \mathsf{F}\, g \cdot \mathsf{F} f \ . \tag{23}$$

The force of functoriality lies in the action on arrows and in the preservation of composition. There is an identity functor, $Id_{\_}(\mathbb{C}) : \mathbb{C} \to \mathbb{C}$, and functors can be composed: $(\mathsf{G} \circ \mathsf{F})\, A = \mathsf{G}\, (\mathsf{F}\, A)$ and $(\mathsf{G} \circ \mathsf{F})\, f = \mathsf{G}\, (\mathsf{F} f)$. This data turns small

categories and functors into a category, called **Cat**.[2] We let F, G etc range over functors.

Natural Transformations Let $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$ be two functors. *A natural transformation* $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ is a collection of arrows, so that for each object $A : \mathbb{C}$ there is an arrow $\alpha\, A : \mathbb{D}(\mathsf{F}\, A, \mathsf{G}\, A)$ such that

$$\mathsf{G}\, h \cdot \alpha\, A_1 \;=\; \alpha\, A_2 \cdot \mathsf{F}\, h, \tag{24}$$

for all arrows $h : \mathbb{C}(A_1, A_2)$. Given α and h, there are essentially two ways of turning $\mathsf{F}\, A_1$ things into $\mathsf{G}\, A_2$ things. The coherence condition (24) demands that they are equivalent. There is an identity transformation $id_{\mathsf{F}} : \mathsf{F} \dot{\to} \mathsf{F}$ defined $id_{\mathsf{F}}\, A = id_{\mathsf{F}\, A}$. Natural transformations can be composed: if $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ and $\beta : \mathsf{G} \dot{\to} H$, then $\beta \cdot \alpha : \mathsf{F} \dot{\to} H$ is defined $(\beta \cdot \alpha)\, A = \beta\, A \cdot \alpha\, A$. Thus, functors of type $\mathbb{C} \to \mathbb{D}$ and natural transformations between them form a category, the functor category $\mathbb{D}^{\mathbb{C}}$. (Functor categories are exponentials in **Cat**, hence the notation.) We let α, β etc range over natural transformations.

Subcategories A *subcategory* SC of a category $\mathbb{C}$ is a collection of some of the objects and some of the arrows of $\mathbb{C}$, such that identity and composition are preserved to ensure SC constitutes a category. In a full subcategory, $SC(A, B) = \mathbb{C}(A, B)$, for all objects A and B.

B Source Code

$$(\!|- \leftarrow -|\!) : (\mathit{Functor}\, f) \Rightarrow (f\ a \to a) \to (c \to f\ c) \to (c \to a)$$
$$(\!|a \leftarrow c|\!) = a \cdot \mathit{fmap}\, (\!|a \leftarrow c|\!) \cdot c$$

$$\mathbf{data}\, \mu f = \mathit{in}\, \{\, \mathit{out} : f\, (\mu f)\, \}$$

$$(\!|-|\!) : (\mathit{Functor}\, f) \Rightarrow (f\ a \to a) \to (\mu f \to a)$$
$$(\!|a|\!) = (\!|a \leftarrow \mathit{out}|\!)$$
$$[\!(-)\!] : (\mathit{Functor}\, f) \Rightarrow (c \to f\ c) \to (c \to \mu f)$$
$$[\!(c)\!] = (\!|\mathit{in} \leftarrow c|\!)$$

$$\mu - : (\mathit{Functor}\, f) \Rightarrow (\forall\, a\ .\ f\ a \to g\ a) \to (\mu f \to \mu g)$$
$$\mu \alpha = (\!|\mathit{in} \cdot \alpha \leftarrow \mathit{out}|\!)$$

Example: lists.

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\, (\mathsf{L}\ a)\ \mathbf{where}\\
&\quad \mathit{fmap}\, f\ \mathit{Nil} &&= \mathit{Nil}\\
&\quad \mathit{fmap}\, f\, (\mathit{Cons}\, (a, b)) &&= \mathit{Cons}\, (a, f\ b)
\end{aligned}$$

[2] To avoid paradoxes, we have to require that the objects of **Cat** are small, where a category is called small if the class of objects and the class of all arrows are sets.

$$
\begin{aligned}
&\mathfrak{m} : (a_1 \to a_2) \to \mathsf{L}\ a_1\ b \to \mathsf{L}\ a_2\ b \\
&\mathfrak{m}\ f\ \mathit{Nil} &&= \mathit{Nil} \\
&\mathfrak{m}\ f\ (\mathit{Cons}\ (a, b)) &&= \mathit{Cons}\ (f\ a, b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{p} : \mathsf{L}\ \mathbb{N}\ \mathbb{N} \to \mathbb{N} \\
&\mathfrak{p}\ \mathit{Nil} &&= 1 \\
&\mathfrak{p}\ (\mathit{Cons}\ (a, b)) &&= a \times b
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{f} : (a \to \mathit{Bool}) \to (\mathsf{L}\ a\ b \to b) \to (\mathsf{L}\ a\ b \to b) \\
&\mathfrak{f}\ p\ a\ \mathit{Nil} &&= a\ \mathit{Nil} \\
&\mathfrak{f}\ p\ a\ (\mathit{Cons}\ (x, y)) &&= \mathbf{if}\ p\ x\ \mathbf{then}\ a\ (\mathit{Cons}\ (x, y))\ \mathbf{else}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{test1} = (\!|\mathfrak{p} \leftarrow \mathit{out}|\!) \cdot (\!|\mathit{in} \cdot \mathfrak{m}\ \mathit{sq} \leftarrow \mathit{out}|\!) \cdot (\!|\mathfrak{f}\ \mathit{odd}\ \mathit{in} \leftarrow \mathit{out}|\!) \cdot (\!|\mathit{in} \leftarrow \mathfrak{b}|\!) \\
&\mathit{test2} = (\!|\mathfrak{p}|\!) \cdot (\!|\mathit{in} \cdot \mathfrak{m}\ \mathit{sq}|\!) \cdot (\!|\mathfrak{f}\ \mathit{odd}\ \mathit{in}|\!) \cdot [\!(\mathfrak{b})\!] \\
&\mathit{test3} = (\!|\mathfrak{f}\ \mathit{odd}\ (\mathfrak{p} \cdot \mathfrak{m}\ \mathit{sq}) \leftarrow \mathfrak{b}|\!)
\end{aligned}
$$

$$
\mathit{sq}\ x = x \times x
$$

Example: stream fusion.

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ (\mathsf{S}\ a)\ \mathbf{where} \\
&\quad \mathit{fmap}\ f\ \mathit{Done} &&= \mathit{Done} \\
&\quad \mathit{fmap}\ f\ (\mathit{Yield}\ (a, b)) &&= \mathit{Yield}\ (a, f\ b) \\
&\quad \mathit{fmap}\ f\ (\mathit{Skip}\ b) &&= \mathit{Skip}\ (f\ b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{m} : (a_1 \to a_2) \to \mathsf{S}\ a_1\ b \to \mathsf{S}\ a_2\ b \\
&\mathfrak{m}\ f\ \mathit{Done} &&= \mathit{Done} \\
&\mathfrak{m}\ f\ (\mathit{Yield}\ (a, b)) &&= \mathit{Yield}\ (f\ a, b) \\
&\mathfrak{m}\ f\ (\mathit{Skip}\ b) &&= \mathit{Skip}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{p} : \mathsf{S}\ \mathbb{N}\ \mathbb{N} \to \mathbb{N} \\
&\mathfrak{p}\ \mathit{Done} &&= 1 \\
&\mathfrak{p}\ (\mathit{Yield}\ (a, b)) &&= a \times b \\
&\mathfrak{p}\ (\mathit{Skip}\ b) &&= b
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{f} : (a \to \mathit{Bool}) \to \mathsf{S}\ a\ b \to \mathsf{S}\ a\ b \\
&\mathfrak{f}\ p\ \mathit{Done} &&= \mathit{Done} \\
&\mathfrak{f}\ p\ (\mathit{Yield}\ (x, y)) &&= \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{Yield}\ (x, y)\ \mathbf{else}\ \mathit{Skip}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathfrak{b} : (\mathbb{N}, \mathbb{N}) \to \mathsf{S}\ \mathbb{N}\ (\mathbb{N}, \mathbb{N}) \\
&\mathfrak{b}\ (m, n) = \mathbf{if}\ m > n\ \mathbf{then}\ \mathit{Done}\ \mathbf{else}\ \mathit{Yield}\ (m, (\mathit{succ}\ m, n))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{test4} = (\!|\mathfrak{p}|\!) \cdot \mu(\mathfrak{m}\ \mathit{sq}) \cdot \mu(\mathfrak{f}\ \mathit{odd}) \cdot [\!(\mathfrak{b})\!] \\
&\mathit{test5} = (\!|\mathfrak{p} \cdot \mathfrak{m}\ \mathit{sq} \cdot \mathfrak{f}\ \mathit{odd} \leftarrow \mathfrak{b}|\!)
\end{aligned}
$$

Tail recursion.

$$
\mathbf{data}\ \mathsf{T}\ a\ b = \mathit{Stop}\ a \mid \mathit{Loop}\ b
$$

$\mu(\mathsf{T}\ A)$ is the partiality monad (*Now*, *Later*)?

```
instance Functor (T a) where
  fmap f (Stop a) = Stop a
  fmap f (Loop b) = Loop (f b)

𝔯 : T a a → a
𝔯 (Stop a) = a
𝔯 (Loop b) = b
```

This is the codiagonal.

```
𝔣𝔩 : (a → b → a) → (a, [b]) → T a (a, [b])
𝔣𝔩 f (e, [])     = Stop e
𝔣𝔩 f (e, a : x) = Loop (f e a, x)

shunt : ([a], [a]) → [a]
shunt = (|𝔯 ← 𝔣𝔩 (flip (:))|)
```

Composing Reactive GUIs in F# with WebSharper

Joel Björnson
IntelliFactory

Anton Tayanovskyy
IntelliFactory

Adam Granicz
IntelliFactory

Abstract

We present a generic library for constructing composable and interactive user interfaces in a declarative style. The paper introduces *flowlets*, an extension of *formlets*[2, 3] that uses functional reactive programming (FRP) to provide interactivity. Real-world examples are given using the current implementation that compiles flowlets defined in F# to JavaScript with WebSharper[1].

1 Motivation

One of the remaining challenges for functional programming is to tackle the construction of graphical user interfaces with demonstrable advantages over the mainstream object-oriented techniques. The challenge is to make user interfaces practical to construct and yet easy to reason about – ideally, using equational reasoning. There are two aspects:

1. *Avoiding side-effects.* Side effects permeate mainstream GUI programming as a natural model for expressing change. Current approaches to avoid side effects include the one taken by *Fudgets*[1], which model change in the pure subset of Haskell, and the one used in *FranTK*[5] and *Grapefruit*[2], which encourages functional reactive (FRP) abstractions to describe changing state.

2. *Achieving compositionality.* Functional programming naturally favors a combinatory design. Ideally a GUI library would be closed under composition, providing an algebra on widgets. This is successfully solved by *formlets*[2, 3], a type-safe, compositional abstraction for web forms running in the CGI environment.

[1] http://www.websharper.com
[2] http://www.haskell.org/haskellwiki/Grapefruit

2 Overview

Flowlets build on the compositional design of formlets by extending it to use FRP to describe interactive state changes. Flowlets support dynamic dependencies, input validation and other interactive scenarios, and are designed to run in a rich GUI environment like the desktop or the web browser.

2.1 Design

We assume the availability of a push-based FRP implementation that defines a *stream* type, a typed channel of notifications:

```
type 'a stream
type 'a s = 'a stream
val const  : 'a -> 'a s
val map    : ('a -> 'b) -> 'a s -> 'b s
val bind   : 'a -> ('a -> 'b s) -> 'b s
val merge  : 'a s -> 'a s -> 'a s
val subscr : ('a -> unit) -> 'a s -> unit
```

A flowlet of type *'a* interactively collects values of type *'a* by presenting the user with a dynamically changing list of visual *body* elements. Conceptually:

```
type 'a flowlet = {
    values : 'a stream
    body   : (body list) stream
}
```

For example, a textbox is modelled as a *string flowlet* that sends a message to the *values* stream on every edit initiated by the user.

As with formlets, the idiom[3] operators provide the default flowlet composition, where the body of the composed flowlet concatenates the bodies of its constituents:

```
let user : user flowlet =
    pure (fun x y -> {name = x; age = y})
    <*> textBox
    <*> intBox
```

Flowlets extend formlets in two ways:

1. Streams allow interactive updates to the body of the flowlet. This makes interactive input validation possible.
2. Flowlets form a monad[4], allowing dynamic dependencies.

2.2 Rendering

Formlets are rendered to the underlying widget framework using a *render* function, which must make use of mutation to embed a stream of widgets into a widget:

```
val render : body list stream -> body
```

The typical render implementations provide horizontal and vertical table layouts. Dependent flowlets constructed with *bind* are seen by the user as dynamically expanding or contracting tables.

Receiving notifications with the complete visual state is inefficient, however, as it requires the rendering engine to redraw the flowlet on every update. We avoid this problem by using a binary tree to represent the state and propagating changes as tree edits, which requires to redraw only the affected parts of the flowlet:

```
type 'a tree =
 | Empty
 | Leaf of 'a
 | Fork of 'a tree * 'a tree

type 'a edit =
 | Replace of 'a tree
 | Left of 'a edit
 | Right of 'a edit

type 'a flowlet = {
    values : 'a stream
    body   : body edit stream
}
```

2.3 Implementation

The current implementation uses F#. Streams and their operations are provided by the Reactive Extensions for .NET[3] (streams are modelled as *IObservable*). Compiled to JavaScript with WebSharper, flowlets run in the browser.

As the design of the library is generic, it can be easily adapted to other environments. We are currently working on another F# implementation that uses Windows Presentation Foundation and runs on the desktop.

[3] msdn.microsoft.com/en-us/devlabs/ee794896.aspx

3 Summary

By combining functional reactive programming with formlets, flowlets offer an interactive user interface abstraction that is simple to use and reason about. It is our hope that functional programming techniques, in the form of flowlets or otherwise, will gain more ground in the UI world.

References

[1] Magnus Carlsson and Thomas Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330, New York, NY, USA, 1993. ACM.

[2] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Sixth Asian Symposium on Programming Languages and Systems*, 2008.

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical report, University of Edinburgh, 2008.

[4] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[5] Meurig Sage. FranTk - a declarative GUI language for Haskell. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 106–117, New York, NY, USA, 2000. ACM.