Proceedings of the 1999 Haskell Workshop

Erik Meijer (editor)

UU-CS-1999-28
Friday October 9th, 1999, Paris, France

Proceedings of the 1999 Haskell Workshop

UU-CS-1999-28

Erik Meijer (editor)

Friday October 1st, 1999, Paris, France

1999 Haskell Workshop

The purpose of the workshop is to discuss experience with Haskell, and possible future developments of the language.

The lively discussions at the 1997 Haskell Workshop in Amsterdam about the future of Haskell led to the definition of Haskell'98, giving Haskell the stability that has so far been lacking. The road ahead to Haskell-2 has many opportunities for developing and enhancing Haskell in new and exciting ways. The third Haskell workshop presents six papers on the design, implementation, and use of Haskell.

The program committee consisted of

- Koen Claessen (Chalmers)
- Byron Cook (OGI)
- Gregory Hager (The Johns Hopkins University)
- Graham Hutton (Nottingham)
- Alexander Jacobson (shop.com)
- Fergus Henderson (Melbourne)
- Sigbjorn Finne (Glasgow)
- Erik Meijer (Utrecht) – chair
- Colin Runciman (York)
- Philip Wadler (Bell Labs, Lucent Technologies)

The Haskell Workshop was held in conjuction with PLI'99 and was sponsored by INRIA, CNRS, Microsoft Research, Trusted Logic, France Telecom, and the French Ministère de l'Education Nationale de la Recherche et de la Technology. Additional sponsoring was received by the University Research Programs group of Microsoft Research Cambridge for the Jake project. The goal of Jake is to develop a set of Perl and Tcl free conference management tools completely programmed in Haskell.

Erik Meijer
September 1999

Workshop Program

8:30 - 9:30 PLI Invited talk

Mobility in the Join-Calculus
Georges Gonthier (INRIA Rocquencourt, France)

9:45 - 10:45 Session 1

Typing Haskell In Haskell
Mark Jones (Oregon Graduate Institute, USA)

Haskell library template effort (10 minute slot)
Andy Gill (Oregon Graduate Institute, USA)

10:45 - 11:15 Coffee break

11:15 - 12:45 Session 2

Embedding Prolog in Haskell
Silvija Seres and *Mike Spivey* (Oxford University, UK)

Logical Abstractions in Haskell
Nancy A. Day, *John Launchbury* and *Jeff Lewis* (Oregon Graduate Institute, USA)

12:45 - 14:30 Lunch

14:30 - 16:00 Session 3

The Syntactical Subtleties of Haskell (Invited Talk)
Simon Marlow (Microsoft Research Cambridge, UK)

Lightweight Extensible Records for Haskell
Mark Jones (Oregon Graduate Institute, USA) and *Simon Peyton Jones* (Microsoft Research Cambridge, UK)

16:00 - 16:30 Tea time

16:30 - 18:00 Session 4

A Generic Programming Extension for Haskell
Ralf Hinze (Bonn University, Germany)

Restricted Datatypes in Haskell
John Hughes (Chalmers University, Sweden)

18:00 - 18:30 Session 5

Dependent types: Doing without them (10 minute slot)
Daniel Fridlender (BRICS, Denmark) and *Mia Indrika* (Chalmers University, Sweden)

The future of Haskell (Open-mike session)

Typing Haskell In Haskell

Mark Jones (Oregon Graduate Institute, USA)

Typing Haskell in Haskell

Mark P. Jones
Oregon Graduate Institute of Science and Technology
mpj@cse.ogi.edu

Haskell Workshop Version: September 1, 1999

Abstract

Haskell benefits from a sophisticated type system, but implementors, programmers, and researchers suffer because it has no formal description. To remedy this shortcoming, we present a Haskell program that implements a Haskell typechecker, thus providing a mathematically rigorous specification in a notation that is familiar to Haskell users. We expect this program to fill a serious gap in current descriptions of Haskell, both as a starting point for discussions about existing features of the type system, and as a platform from which to explore new proposals.

1 Introduction

Haskell[1] benefits from one of the most sophisticated type systems of any widely used programming language. Unfortunately, it also suffers because there is no formal specification of what the type system should be. As a result:

- It is hard for Haskell implementors to be sure that their compilers and interpreters accept the same programs as other implementations. The informal specification in the Haskell report [10] leaves too much room for confusion and misinterpretation. This leads to genuine discrepancies between implementations, as many subscribers to the Haskell mailing list will have seen.
- It is hard for Haskell programmers to understand the details of the type system, and to appreciate why some programs are accepted when others are not. Formal presentations of most aspects of the type system are available in the research literature, but often abstract on specific features that are Haskell-like, but not Haskell-exact, and do not describe the complete type system. Moreover, these papers often use disparate and unfamiliar technical notation and concepts that may be hard for some Haskell programmers to understand.
- It is hard for Haskell researchers to explore new type system extensions, or even to study usability issues that arise with the present type system such as the search for better type error diagnostics. Work in these areas requires a clear understanding of the type system and, ideally, a platform on which to build and experiment with prototype implementations. The existing Haskell implementations are not suitable for this (and were not intended to be): the nuts and bolts of a type system are easily obscured by the use of specific data structures and optimizations, or by the need to integrate smoothly with other parts of an implementation.

This paper presents a formal description of the Haskell type system using the notation of Haskell itself as a specification language. Indeed, the source code for this paper is itself an executable Haskell program that is passed through a custom preprocessor and then through LaTeX to obtain the typeset version. The type checker is available in source form on the Internet at `http://www.cse.ogi.edu/~mpj/thih/`. We hope that this will serve as a resource for Haskell implementors, programmers and researchers, and that it will be a first step in eliminating most of the problems described above.

One audience whose needs may not be particularly well met by this paper are researchers in programming language type systems who do not have experience of Haskell. (We would, however, encourage anyone in that position to learn more about Haskell!) Indeed, we do not follow the traditional route in such settings where the type system might first be presented in its purest form, and then related to a more concrete type inference algorithm by soundness and completeness theorems. Here, we deal only with type inference. It doesn't even make sense to ask if our algorithm computes 'principal' types: such a question requires a comparison between two different presentations of a type system, and we only have one. Nevertheless, we believe that the specification in this paper could easily be recast in a more standard, type-theoretic manner and used to develop a presentation of Haskell typing in a more traditional style.

The code presented here can be executed with any Haskell system, but our primary goals have been clarity and simplicity, and the resulting code is not intended to be an efficient implementation of type inference. Indeed, in some places, our choice of representation may lead to significant overheads and duplicated computation. It would be interesting to try to derive a more efficient, but provably correct implementation from the specification given here. We have not attempted to do this because we expect that it would obscure the key ideas that we want to emphasize. It therefore remains as a topic for future work, and as a test to assess the applicability of program transformation and synthesis to complex, but modestly sized Haskell programs.

[1] Throughout, we use 'Haskell' as an abbreviation for 'Haskell 98'.

Another goal for this paper was to give as complete a description of the Haskell type system as possible, while also aiming for conciseness. For this to be possible, we have assumed that certain transformations and checks will have been made prior to typechecking, and hence that we can work with a much simpler abstract syntax than the full source-level syntax of Haskell would suggest. As we argue informally at various points in the paper, we do not believe that there would be any significant difficulty in extending our system to deal with the missing constructs. All of the fundamental components, including the thorniest aspects of Haskell typing, are addressed in the framework that we present here. Our specification does not attempt to deal with all of the issues that would occur in the implementation of a full Haskell implementation. We do not tackle the problems of interfacing a typechecker with compiler front ends (to track source code locations in error diagnostics, for example) or back ends (to describe the implementation of overloading, for example), nor do we attempt to formalize any of the extensions that are implemented in current Haskell systems. This is one of things that makes our specification relatively concise; by comparison, the core parts of the Hugs typechecker takes some 90+ pages of C code.

Regrettably, length restrictions have prevented us from including many examples in this paper to illustrate the definitions at each stage. For the same reason, definitions of a few constants that represent entities in the standard prelude, as well as the machinery that we use in testing to display the results of type inference, are not included in the typeset version of this paper. Apart from those details, this paper gives the full source code.

We expect the program described here to evolve in at least three different ways.

- Formal specifications are not immune to error, and so it is possible that changes will be required to correct bugs in the code presented here. On the other hand, by writing our specification as a program that can be typechecked and executed with existing Haskell implementations, we have a powerful facility for detecting simple bugs automatically and for testing to expose deeper problems.
- As it stands, this paper just provides one more interpretation of the Haskell type system. We believe that it is consistent with the official specification, but because the latter is given only informally, we cannot establish the correctness of our presentation here in any rigorous manner. We hope that this paper will stimulate discussion in the Haskell community, and would expect to make changes to the specification as we work towards some kind of consensus.
- There is no shortage of proposed extensions to the Haskell type system, some of which have already been implemented in one or more of the available Haskell systems. Some of the better known examples of this include multiple-parameter type classes, existential types, rank-2 polymorphism, extensible records. We would like to obtain formal descriptions for as many of these proposals as possible by extending the core specification presented here.

It will come as no surprise to learn that some knowledge of Haskell will be required to read this paper. That said, we have tried to keep the definitions and code as clear and simple as possible, and although we have made some use of Haskell overloading and do-notation, we have generally avoided using the more esoteric features of Haskell. In addition, some experience with the basics of Hindley-Milner style type inference [5, 9, 2] will be needed to understand the algorithms presented here. Although we have aimed to keep our presentation as simple as possible, some aspects of the problems that we are trying to address have inherent complexity or technical depth that cannot be side-stepped. In short, this paper will probably not be useful as a tutorial introduction to Hindley-Milner style type inference!

Description	Symbol	Type
kind	$k, \ldots$	*Kind*
type constructor	$tc, \ldots$	*Tycon*
type variable	$v, \ldots$	*Tyvar*
– 'fixed'	$f, \ldots$	
– 'generic'	$g, \ldots$	
type	$t, \ldots$	*Type*
class	$c, \ldots$	*Class*
predicate	$p, q, \ldots$	*Pred*
– 'deferred'	$d, \ldots$	
– 'retained'	$r, \ldots$	
qualified type	$qt, \ldots$	*QualType*
scheme	$sc, \ldots$	*Scheme*
substitution	$s, \ldots$	*Subst*
unifier	$u, \ldots$	*Subst*
assumption	$a, \ldots$	*Assump*
identifier	$i, \ldots$	*Id*
literal	$l, \ldots$	*Literal*
pattern	$pat, \ldots$	*Pat*
expression	$e, f, \ldots$	*Expr*
alternative	$alt, \ldots$	*Alt*
binding group	$bg, \ldots$	*BindGroup*

Figure 1: Notational Conventions

2 Preliminaries

For simplicity, we present the code for our typechecker as a single Haskell module. The program uses only a handful of standard prelude functions, like *map*, *concat*, *all*, *any*, *mapM*, etc., and a few operations from the *List* library:

```
module TypingHaskellInHaskell where
import List (nub, (\\), intersect, union, partition)
```

For the most part, our choice of variable names follows the notational conventions set out in Figure 1. A trailing s on a variable name usually indicates a list. Numeric suffices or primes are used as further decoration where necessary. For example, we use k or k' for a kind, and ks or ks' for a list of kinds. The types and terms appearing in the table are described more fully in later sections. To distinguish the code for the typechecker from program fragments that are used to discuss its behavior, we typeset the former in an *italic* font, and the latter in a `typewriter` font.

Throughout this paper, we implement identifiers as strings, and assume that there is a simple way to generate new iden-

tifiers dynamically using the *enumId* function:

```
type Id    =  String
enumId     ::  Int → Id
enumId n   =  "v" ++ show n
```

3 Kinds

To ensure that they are valid, Haskell type constructors are classified into different *kinds*: the kind $*$ (pronounced 'star') represents the set of all simple (i.e., nullary) type expressions, like *Int* and *Char* → *Bool*; kinds of the form $k_1 \to k_2$ represent type constructors that take an argument type of kind k_1 to a result type of kind k_2. For example, the standard list, *Maybe* and *IO* constructors all have kind $* \to *$. Here, we will represent kinds as values of the following datatype:

```
data Kind  =  Star | Kfun Kind Kind
           deriving Eq
```

Kinds play essentially the same role for type constructors as types do for values, but the kind system is clearly very primitive. There are a number of extensions that would make interesting topics for future research, including polymorphic kinds, subkinding, and record/product kinds. A simple extension of the kind system—adding a new *row* kind—has already proved to be useful for the Trex implementation of extensible records in Hugs [3, 7].

4 Types

The next step is to define a representation for types. Stripping away syntactic sugar, Haskell type expressions are either type variables or constants (each of which has an associated kind), or applications of one type to another: applying a type of kind $k_1 \to k_2$ to a type of kind k_1 produces a type of kind k_2:

```
data Type  =  TVar Tyvar
           |  TCon Tycon
           |  TAp Type Type
           |  TGen Int
           deriving Eq

data Tyvar =  Tyvar Id Kind
           deriving Eq

data Tycon =  Tycon Id Kind
           deriving Eq
```

The following examples show how standard primitive datatypes are represented as type constants:

```
tChar   =  TCon (Tycon "Char" Star)
tArrow  =  TCon (Tycon "(->)" (Kfun Star
                                (Kfun Star
                                 Star)))
```

A full Haskell compiler or interpreter might store additional information with each type constant—such as the the list of constructor functions for an algebraic datatype—but such details are not needed during typechecking.

Types of the form *TGen n* represent 'generic', or quantified type variables; their role is described in Section 8.

We do not provide a representation for type synonyms, assuming instead that they have been fully expanded before typechecking. It is always possible for an implementation to do this because Haskell prevents the use of a synonym without its full complement of arguments. Moreover, the process is guaranteed to terminate because recursive synonym definitions are prohibited. In practice, however, implementations are likely to expand synonyms more lazily: in some cases, type error diagnostics may be easier to understand if they display synonyms rather than expansions.

We end this section with the definition of two helper functions. The first provides a way to construct function types:

```
infixr  4  `fn`
fn         ::  Type → Type → Type
a `fn` b   =   TAp (TAp tArrow a) b
```

The second introduces an overloaded function, *kind*, that can be used to determine the kind of a type variable, type constant, or type expression:

```
class HasKind t where
  kind  ::  t → Kind
instance HasKind Tyvar where
  kind (Tyvar v k)  =  k
instance HasKind Tycon where
  kind (Tycon v k)  =  k
instance HasKind Type where
  kind (TCon tc)    =  kind tc
  kind (TVar u)     =  kind u
  kind (TAp t _)    =  case (kind t) of
                         (Kfun _ k)  →  k
```

Most of the cases here are straightforward. Notice, however, that we can calculate the kind of an application $(TAp\ t\ t')$ using only the kind of its first argument t: Assuming that the type is well-formed, t must have a kind $k' \to k$, where k' is the kind of t' and k is the kind of the whole application. This shows that we need only traverse the leftmost spine of a type expression to calculate its kind.

5 Substitutions

Substitutions—which are just finite functions, mapping type variables to types—play a major role in type inference. In this paper, we represent substitutions using association lists:

```
type Subst  =  [(Tyvar, Type)]
```

To ensure that we work only with well-formed type expressions, we will be careful to construct only *kind-preserving* substitutions, in which variables can be mapped only to types of the same kind.

The simplest substitution is the null substitution, represented by the empty list, which is obviously kind-preserving:

```
nullSubst  ::  Subst
nullSubst  =   []
```

Almost as simple are the substitutions $(u \mapsto t)$[2] that map a single variable u to a type t of the same kind:

```
(↦)     ::  Tyvar → Type → Subst
u ↦ t   =   [(u, t)]
```

This is kind-preserving if, and only if, *kind* u = *kind* t.

Substitutions can be applied to types—or to anything containing type components—in a natural way. This suggests that we overload the operation to *apply* a substitution so that it can work on different types of object:

```
class Types t where
  apply  ::  Subst → t → t
  tv     ::  t → [Tyvar]
```

In each case, the purpose of applying a substitution is the same: To replace every occurrence of a type variable in the domain of the substitution with the corresponding type. We also include a function *tv* that returns the set of type variables (i.e., *Tyvar*s) appearing in its argument, listed in order of first occurrence (from left to right), with no duplicates. The definitions of these operations for *Type* are as follows:

```
instance Types Type where
  apply s (TVar u)   =  case lookup u s of
                          Just t    →  t
                          Nothing   →  TVar u
  apply s (TAp l r)  =  TAp (apply s l) (apply s r)
  apply s t          =  t

  tv (TVar u)        =  [u]
  tv (TAp l r)       =  tv l `union` tv r
  tv t               =  []
```

It is straightforward (and useful!) to extend these operations to work on lists:

```
instance Types a ⇒ Types [a] where
  apply s  =  map (apply s)
  tv       =  nub . concat . map tv
```

The *apply* function can be used to build more complex substitutions. For example, composition of substitutions, specified by *apply* $(s_1@@s_2)$ = *apply* s_1 . *apply* s_2, can be defined more concretely using:

```
infixr  4  @@
(@@)    ::  Subst → Subst → Subst
s1@@s2  =   [(u, apply s1 t) | (u, t) ← s2] ++ s1
```

We can also form a 'parallel' composition $s_1 +\!\!+ s_2$ of two substitutions s_1 and s_2, but the result is 'left-biased' because bindings in s_1 take precedence over any bindings for the same variables in s_2. For a more symmetric version of this operation, we use a *merge* function, which checks that the two substitutions agree at every variable in the domain of both and hence guarantees that *apply* $(s_1 +\!\!+ s_2)$ = *apply* $(s_2 +\!\!+ s_1)$.

[2]The typeset version of the symbol ↦ is written +-> in the concrete syntax of Haskell.

Clearly, this is a partial function, which we reflect by arranging for *merge* to return a result of type *Maybe Subst*:

```
merge         ::  Subst → Subst → Maybe Subst
merge s1 s2   =   if agree then Just s else Nothing
  where dom s   =  map fst s
        s       =  s1 ++ s2
        agree   =  all (\v → apply s1 (TVar v) ==
                             apply s2 (TVar v))
                       (dom s1 `intersect` dom s2)
```

It is easy to check that both (@@) and *merge* produce kind-preserving results from kind-preserving arguments.

6 Unification and Matching

The goal of unification is to find a substitution that makes two types equal—for example, to ensure that the domain type of a function matches up with the type of an argument value. However, it is also important for unification to find as 'small' a substitution as possible, because that will also lead to the most general type. More formally, a substitution s is a *unifier* of two types t_1 and t_2 if *apply* s t_1 == *apply* s t_2. A *most general unifier*, or *mgu*, of two such types is a unifier u with the property that any other unifier s can be written as $s'@@u$, for some substitution s'.

The syntax of Haskell types has been carefully chosen to ensure that, if two types have any unifying substitutions, then they also have a most general unifier, which can be calculated by a simple variant of Robinson's algorithm [11]. One of the main reasons for this is that there are no non-trivial equalities on types. Extending the type system with higher-order features (such as lambda expressions on types), or with any other mechanism that allows reductions or rewriting in the type language, will often make unification undecidable, non-unitary (meaning that there may not be most general unifiers), or both. This, for example, is why it is not possible to allow type synonyms to be partially applied (and interpreted as some restricted kind of lambda expression).

The calculation of most general unifiers is implemented by a pair of functions:

```
mgu      ::  Type → Type → Maybe Subst
varBind  ::  Tyvar → Type → Maybe Subst
```

Both of these return results using *Maybe* because unification is a partial function. However, because *Maybe* is a monad, the programming task can be simplified by using Haskell's monadic do-notation. The main unification algorithm is described by *mgu*, which uses the structure of its arguments to guide the calculation:

```
mgu (TAp l r) (TAp l' r')  =  do s1 ← mgu l l'
                                 s2 ← mgu (apply s1 r)
                                          (apply s1 r')
                                 Just (s2@@s1)
mgu (TVar u) t             =  varBind u t
mgu t (TVar u)             =  varBind u t
mgu (TCon tc1) (TCon tc2)
  | tc1 == tc2             =  Just nullSubst
mgu t1 t2                  =  Nothing
```

The *varBind* function is used for the special case of unifying a variable u with a type t. At first glance, one might think that we could just use the substitution $(u \mapsto t)$ for this. In practice, however, tests are required to ensure that this is valid, including an 'occurs check' (u `` `elem` `` $tv\ t$) and a test to ensure that the substitution is kind-preserving:

```
varBind u t | t == TVar u       =  Just nullSubst
            | u `elem` tv t     =  Nothing
            | kind u == kind t  =  Just (u ↦ t)
            | otherwise         =  Nothing
```

In the following sections, we will also make use of an operation called *matching* that is closely related to unification. Given two types t_1 and t_2, the goal of matching is to find a substitution s such that $apply\ s\ t_1 = t_2$. Because the substitution is applied only to one type, this operation is often described as *one-way* matching. The calculation of matching substitutions is implemented by a function:

```
match :: Type → Type → Maybe Subst
```

Matching follows the same pattern as unification, except that it uses *merge* rather than @@ in the case for type applications, and it does not allow binding of variables in t_2:

```
match (TAp l r) (TAp l' r')  =  do sl ← match l l'
                                   sr ← match r r'
                                   merge sl sr
match (TVar u) t
  | kind u == kind t         =  Just (u ↦ t)
match (TCon tc1) (TCon tc2)
  | tc1 == tc2               =  Just nullSubst
match t1 t2                  =  Nothing
```

7 Predicates and Qualified Types

Haskell types can be *qualified* by adding a (possibly empty) list of *predicates*, or class constraints, to restrict the ways in which type variables are instantiated[3]:

```
data Qual t  =  [Pred] :⇒ t
             deriving Eq
```

Predicates themselves consist of a class name, and a type:

```
data Pred  =  IsIn Class Type
           deriving Eq
```

Haskell's classes represent sets of types. For example, a predicate $IsIn\ c\ t$ asserts that t is a member of the class c. It would be easy to extend the *Pred* datatype to allow other forms of predicate, as is done with Trex records in Hugs [7]. Another frequently requested extension is to allow classes to accept multiple parameters, which would require a list of *Type*s rather than the single *Type* in the definition above.

[3]The typeset version of the symbol :⇒ is written :=> in the concrete syntax of Haskell, and corresponds directly to the => symbol that is used in instance declarations and in types.

The extension of *Types* to the *Qual* and *Pred* datatypes is straightforward:

```
instance Types t ⇒ Types (Qual t) where
  apply s (ps :⇒ t)  =  apply s ps :⇒ apply s t
  tv (ps :⇒ t)       =  tv ps `union` tv t

instance Types Pred where
  apply s (IsIn c t)  =  IsIn c (apply s t)
  tv (IsIn c t)       =  tv t
```

7.1 Classes and Instances

A Haskell type class can be thought of as a set of types (of some particular kind), each of which supports a certain collection of *member functions* that are specified as part of the class declaration. The types in each class (known as *instances*) are specified by a collection of instance declarations. We will assume that class names appearing in the original source code have been mapped to values of the following *Class* datatype prior to typechecking:

```
data Class  =  Class {name :: Id,
                      super :: [Class],
                      insts :: [Inst]}
type Inst   =  Qual Pred
```

Values of type *Class* and *Inst* correspond to source level class and instance declarations, respectively. Only the details that are needed for type inference are included in these representations. A full Haskell implementation would need to store additional information for each declaration, such as the member functions for the class, or their implementations in a particular instance.

A derived equality on *Class* is not useful because the data structures may be cyclic and so a test for structural equality might not terminate when applied to equal arguments. Instead, we use the *name* field to define an equality:

```
instance Eq Class where
  c == c'  =  name c == name c'
```

Apart from using a different keyword, Haskell class and instance declarations begin in the same way, with a clause of the form $preds \Rightarrow pred$ for some (possibly empty) 'context' *preds*, and a 'head' predicate *pred*. In a class declaration, the context is used to specify the immediate superclasses, which we represent more directly by the list of classes in the field *super*: If a type is an instance of a class c, then it must also be an instance of any superclasses of c. Using only superclass information, we can be sure that, if a given predicate p holds, then so too must all of the predicates in the list $bySuper\ p$:

```
bySuper :: Pred → [Pred]
bySuper p@(IsIn c t)
        =  p : concat (map bySuper supers)
           where supers  =  [IsIn c' t | c' ← super c]
```

The list $bySuper\ p$ may contain duplicates, but it will always be finite because restrictions in Haskell ensure that the superclass hierarchy is acyclic.

The final field in each *Class* structure, *insts*, is the list of instance declarations for that particular class. Each such instance declaration is represented by a clause $ps :\Rightarrow h$. Here, h is a predicate that describes the form of instances that the declaration can produce, while the context ps lists any constraints that it requires. We can use the following function to see if a particular predicate p can be deduced using a given instance. The result is either *Just ps*, where *ps* is a list of subgoals that must be established to complete the proof, or *Nothing* if the instance does not apply:

```
byInst                  ::  Pred → Inst → Maybe [Pred]
byInst p (ps :⇒ h)      =   do u ← matchPred h p
                               Just (map (apply u) ps)
```

To see if an instance applies, we use one-way matching on predicates, which is implemented as follows:

```
matchPred          ::  Pred → Pred → Maybe Subst
matchPred (IsIn c t) (IsIn c' t')
  | c == c'        =   match t t'
  | otherwise      =   Nothing
```

We can find all the relevant instances for a given predicate $p = IsIn\ c\ t$ in *insts* c. So, if there are any ways to apply an instance to p, then we can find one using:

```
reducePred                  ::  Pred → Maybe [Pred]
reducePred p@(IsIn c t)     =   foldr (|||) Nothing poss
  where poss                =   map (byInst p) (insts c)
        Nothing|||y         =   y
        Just x|||y          =   Just x
```

In fact, because Haskell prevents the definition of overlapping instances, we can be sure that, if *reducePreds* succeeds, then we have actually found the *only* applicable instance.

7.2 Entailment

The information provided by class and instance declarations can be combined to define an *entailment* relation on predicates. As in the theory of qualified types [6], we write $ps \Vdash p$ to indicate that the predicate p will hold whenever all of the predicates in ps are satisfied. To make this more concrete, we define the following function[4]:

```
(⊩)       ::  [Pred] → Pred → Bool
ps ⊩ p    =   any (p `elem`) (map bySuper ps) ||
              case reducePred p of
                Nothing   →  False
                Just qs   →  all (ps ⊩) qs
```

The first step here is to determine whether p can be deduced from ps using only superclasses. If that fails, we look for a matching instance and generate a list of predicates qs as a new goal, each of which must, in turn, follow from ps.

Conditions specified in the Haskell report—namely that the class hierarchy is acyclic and that the types in any instance declaration are strictly smaller than those in the head—are enough to guarantee that tests for entailment will terminate. Completeness of the algorithm is also important: will $ps \Vdash p$ always return *True* whenever there is a way to prove p from ps? In fact our algorithm does not cover all possible cases: it does not test to see if p is a superclass of some other predicate q such that $ps \Vdash q$. Extending the algorithm to test for this would be very difficult because there is no obvious way to choose a particular q, and, in general, there will be infinitely many potential candidates to consider. Fortunately, a technical condition in the Haskell report [10, Condition 1 on Page 47] reassures us that this is not necessary: if p can be obtained as an immediate superclass of some predicate q that was built using an instance declaration in an entailment $ps \Vdash q$, then ps must already be strong enough to deduce p. Thus, although we have not formally proved these properties, we believe that our algorithm is sound, complete, and guaranteed to terminate.

[4]The typeset version of the symbol ⊩ is written `||-` in the concrete syntax of Haskell.

8 Type Schemes

Type schemes are used to describe polymorphic types, and are represented using a list of kinds and a qualified type:

```
data Scheme  =  Forall [Kind] (Qual Type)
                deriving Eq
```

There is no direct equivalent of *Forall* in the syntax of Haskell. Instead, implicit quantifiers are inserted as necessary to bind free type variables.

In a type scheme *Forall ks qt*, each type of the form *TGen n* that appears in the qualified type *qt* represents a generic, or universally quantified type variable, whose kind is given by $ks\,!!\,n$. This is the only place where we will allow *TGen* values to appear in a type. We had originally hoped that this restriction could be enforced statically by a careful choice of the representation for types and type schemes. However, after considering several other alternatives, we eventually settled for the representation shown here because it allows for simple implementations of equality and substitution. For example, because the implementation of substitution on *Type* ignores *TGen* values, we can be sure that there will be no variable capture problems in the following definition:

```
instance Types Scheme where
  apply s (Forall ks qt)  =  Forall ks (apply s qt)
  tv (Forall ks qt)       =  tv qt
```

Type schemes are constructed by quantifying a qualified type *qt* with respect to a list of type variables *vs*:

```
quantify          ::  [Tyvar] → Qual Type → Scheme
quantify vs qt    =   Forall ks (apply s qt)
  where vs'       =   [v | v ← tv qt, v `elem` vs]
        ks        =   map kind vs'
        s         =   zip vs' (map TGen [0..])
```

Note that the order of the kinds in *ks* is determined by the order in which the variables *v* appear in *tv qt*, and not by the order in which they appear in *vs*. So, for example, the leftmost quantified variable in a type scheme will always be represented by *TGen* 0. By insisting that type schemes are

constructed in this way, we obtain a unique canonical form for *Scheme* values. This is very important because it means that we can test whether two type schemes are the same—for example, to determine whether an inferred type agrees with a declared type—using Haskell's derived equality.

In practice, we sometimes need to convert a *Type* into a *Scheme* without adding any qualifying predicates or quantified variables. For this special case, we can use the following function instead of *quantify*:

```
toScheme     ::  Type → Scheme
toScheme t   =   Forall [] ([] :⇒ t)
```

9 Assumptions

Assumptions about the type of a variable are represented by values of the *Assump* datatype, each of which pairs a variable name with a type scheme:

```
data Assump  =  Id :>: Scheme
```

Once again, we can extend the *Types* class to allow the application of a substitution to an assumption:

```
instance Types Assump where
  apply s (i :>: sc)  =  i :>: (apply s sc)
  tv (i :>: sc)       =  tv sc
```

Thanks to the instance definition for *Types* on lists (Section 5), we can also use the *apply* and *tv* operators on the lists of assumptions that are used to record the type of each program variable during type inference. We will also use the following function to find the type of a particular variable in a given set of assumptions:

```
find         ::  Id → [Assump] → Scheme
find i as    =   head [sc | (i' :>: sc) ← as, i == i']
```

We do not make any allowance here for the possibility that the variable i might not appear in as, and assume instead that a previous compiler pass will have detected any occurrences of unbound variables.

10 A Type Inference Monad

It is now quite standard to use monads as a way to hide certain aspects of 'plumbing' and to draw attention instead to more important aspects of a program's design [12]. The purpose of this section is to define the monad that will be used in the description of the main type inference algorithm in Section 11. Our choice of monad is motivated by the needs of maintaining a 'current substitution' and of generating fresh type variables during typechecking. In a more realistic implementation, we might also want to add error reporting facilities, but in this paper the crude but simple *error* function from the Haskell prelude is all that we require. It follows that we need a simple state monad with only a substitution and an integer (from which we can generate new type variables) as its state:

```
newtype TI a  =  TI (Subst → Int → (Subst, Int, a))

instance Monad TI where
  return x     =  TI (\s n → (s, n, x))
  TI c >>= f   =  TI (\s n →
                        let (s', m, x)  =  c s n
                            TI fx       =  f x
                        in fx s' m)

runTI           ::  TI a → a
runTI (TI c)    =   result
  where (s, n, result)  =  c nullSubst 0
```

We provide two operations that deal with the current substitution: *getSubst* returns the current substitution, while *unify* extends it with a most general unifier of its arguments:

```
getSubst     ::  TI Subst
getSubst     =   TI (\s n → (s, n, s))

unify        ::  Type → Type → TI ()
unify t1 t2  =   do s ← getSubst
                    case mgu (apply s t1) (apply s t2) of
                      Just u    →  extSubst u
                      Nothing   →  error "unification"
```

For clarity, we define the operation that extends the substitution as a separate function, even though it is used only here in the definition of *unify*:

```
extSubst     ::  Subst → TI ()
extSubst s'  =   TI (\s n → (s'@@s, n, ()))
```

Overall, the decision to hide the current substitution in the *TI* monad makes the presentation of type inference much clearer. In particular, it avoids heavy use of *apply* every time an extension is (or might have been) computed.

There is only one primitive that deals with the integer portion of the state, using it in combination with *enumId* to generate a new or fresh type variable of a specified kind:

```
newTVar      ::  Kind → TI Type
newTVar k    =   TI (\s n →
                       let v  =  Tyvar (enumId n) k
                       in (s, n+1, TVar v))
```

One place where *newTVar* is useful is in instantiating a type scheme with new type variables of appropriate kinds:

```
freshInst                ::  Scheme → TI (Qual Type)
freshInst (Forall ks qt) =   do ts ← mapM newTVar ks
                                return (inst ts qt)
```

The structure of this definition guarantees that *ts* has exactly the right number of type variables, and each with the right kind, to match *ks*. Hence, if the type scheme is well-formed, then the qualified type returned by *freshInst* will not contain any unbound generics of the form *TGen n*. The definition relies on an auxiliary function *inst*, which is a variation of *apply* that works on generic variables. In other

words, *inst ts t* replaces each occurrence of a generic variable *TGen n* in *t* with *ts !! n*. Although we use it at only this one place, it is still convenient to build up the definition of *inst* using overloading.

```
class Instantiate t where
  inst  ::  [Type] → t → t
instance Instantiate Type where
  inst ts (TAp l r)  =  TAp (inst ts l) (inst ts r)
  inst ts (TGen n)   =  ts !! n
  inst ts t          =  t
instance Instantiate a ⇒ Instantiate [a] where
  inst ts  =  map (inst ts)
instance Instantiate t ⇒ Instantiate (Qual t) where
  inst ts (ps :⇒ t)  =  inst ts ps :⇒ inst ts t
instance Instantiate Pred where
  inst ts (IsIn c t)  =  IsIn c (inst ts t)
```

11 Type Inference

With this section we have reached the heart of the paper, detailing our algorithm for type inference. It is here that we finally see how the machinery that has been built up in earlier sections is actually put to use. We develop the complete algorithm in stages, working through the abstract syntax of the input language from the simplest part (literals) to the most complex (binding groups). Most of our typing rules are expressed in terms of the following type synonym:

```
type Infer e t  =  [Assump] → e → TI ([Pred], t)
```

In more theoretical treatments, it would not be surprising to see the rules expressed in terms of judgments $P \mid A \vdash e : t$, where P is a set of predicates, A is a set of assumptions, e is an expression, and t is a corresponding type [6]. Judgments like this can be thought of as 4-tuples, and the typing rules themselves just correspond to a 4-place relation. Exactly the same structure shows up in types of the form *Infer e t*, except that by using functions, we distinguish very clearly between input and output parameters.

11.1 Literals

Like other languages, Haskell provides special syntax for constant values of certain primitive datatypes, including numerics, characters, and strings. We will represent these *literal* expressions as values of the *Literal* datatype:

```
data Literal  =  LitInt Integer
              |  LitChar Char
```

Type inference for literals is straightforward. For characters, we just return *typeChar*. For integers, we return a new type variable v together with a predicate to indicate that v must be an instance of the *Num* class.

```
tiLit                ::  Literal → TI ([Pred], Type)
tiLit (LitChar _)    =   return ([], tChar)
tiLit (LitInt _)     =   do v ← newTVar Star
                            return ([IsIn cNum v], v)
```

For this last case, we assume the existence of a constant *cNum* :: *Class* to represent the Haskell class *Num*, but, for reasons of space, we do not present the definition here. It is straightforward to add additional cases for Haskell's floating point and *String* literals.

11.2 Patterns

Patterns are used to inspect and deconstruct data values in lambda abstractions, function and pattern bindings, list comprehensions, do notation, and case expressions. We will represent patterns using values of the *Pat* datatype:

```
data Pat  =  PVar Id
          |  PLit Literal
          |  PCon Assump [Pat]
```

A *PVar i* pattern matches any value, and binds the result to the variable i. A *PLit l* pattern matches only the particular value denoted by the literal l. A pattern of the form *PCon a pats* matches only values that were built using the constructor function a with a sequence of arguments that matches *pats*. We use values of type *Assump* to represent constructor functions; all that we really need for typechecking is the type, although the name is useful for debugging. A full implementation of Haskell would store additional details such as arity, and use this to check that constructor functions in patterns are always fully applied.

Most Haskell patterns have a direct representation in *Pat*, but it would need to be extended to account for patterns using labeled fields, and for $(n + k)$ patterns. Neither of these cause any substantial problems, but they do add a little complexity, which we prefer to avoid in this presentation.

Type inference for patterns has two goals: To calculate a type for each bound variable, and to determine what type of values the whole pattern might match. This leads us to look for a function:

```
tiPat  ::  Pat → TI ([Pred], [Assump], Type)
```

Note that we do not need to pass in a list of assumptions here; by definition, any occurence of a variable in a pattern would hide rather than refer to a variable of the same name in an enclosing scope.

For a variable pattern, *PVar i*, we just return a new assumption, binding i to a fresh type variable.

```
tiPat (PVar i)  =  do v ← newTVar Star
                      return ([], [i :>: toScheme v], v)
```

Haskell does not allow multiple use of any variable in a pattern, so we can be sure that this is the first and only occurrence of i that we will encounter in the pattern.

For literal patterns, we use *tiLit* from the previous section:

```
tiPat (PLit l)  =  do (ps, t) ← tiLit l
                      return (ps, [], t)
```

The case for constructed patterns is slightly more complex:

```
tiPat (PCon (i :>: sc) pats)
    = do (ps, as, ts) ← tiPats pats
         t' ← newTVar Star
         (qs :⇒ t) ← freshInst sc
         unify t (foldr fn t' ts)
         return (ps ++ qs, as, t')
```

First we use the *tiPats* function, defined below, to calculate types *ts* for each subpattern in *pats* together with corresponding lists of assumptions in *as* and predicates in *ps*. Next, we generate a new type variable t' that will be used to capture the (as yet unknown) type of the whole pattern. From this information, we would expect the constructor function at the head of the pattern to have type *foldr fn t' ts*. We can check that this is possible by instantiating the known type *sc* of the constructor and unifying.

The *tiPats* function is a variation of *tiPat* that takes a list of patterns as input, and returns a list of types (together with a list of predicates and a list of assumptions) as its result.

```
tiPats        :: [Pat] → TI ([Pred], [Assump], [Type])
tiPats pats   =
  do psasts ← mapM tiPat pats
     let ps   =  [p | (ps, _, _) ← psasts, p ← ps]
         as   =  [a | (_, as, _) ← psasts, a ← as]
         ts   =  [t | (_, _, t) ← psasts]
     return (ps, as, ts)
```

We have already seen how *tiPats* was used in the treatment of *PCon* patterns above. It is also useful in other situations where lists of patterns are used, such as on the left hand side of an equation in a function definition.

11.3 Expressions

Our next step is to describe type inference for expressions, represented by the *Expr* datatype:

```
data Expr  =  Var Id
           |  Lit Literal
           |  Const Assump
           |  Ap Expr Expr
           |  Let BindGroup Expr
```

The *Var* and *Lit* constructors are used to represent variables and literals, respectively. The *Const* constructor is used to deal with named constants, such as the constructor or selector functions associated with a particular datatype or the member functions that are associated with a particular class. We use values of type *Assump* to supply a name and type scheme, which is all the information that we need for the purposes of type inference. Function application is represented using the *Ap* constructor, while *Let* is used to represent let expressions.

Haskell has a much richer syntax of expressions, but they can all be translated into *Expr* values. For example, a lambda expression like `\x->e` can be rewritten using a local definition as `let f x = e in f`, where `f` is a new variable. Similar translations are used for case expressions.

Type inference for expressions is quite straightforward:

```
tiExpr  ::  Infer Expr Type

tiExpr as (Var i)
        =  do let sc  =  find i as
              (ps :⇒ t) ← freshInst sc
              return (ps, t)

tiExpr as (Const (i :>: sc))
        =  do (ps :⇒ t) ← freshInst sc
              return (ps, t)

tiExpr as (Lit l)
        =  do (ps, t) ← tiLit l
              return (ps, t)

tiExpr as (Ap e f)
        =  do (ps, te) ← tiExpr as e
              (qs, tf) ← tiExpr as f
              t ← newTVar Star
              unify (fn tf t) te
              return (ps ++ qs, t)

tiExpr as (Let bg e)
        =  do (ps, as') ← tiBindGroup as bg
              (qs, t) ← tiExpr (as' ++ as) e
              return (ps ++ qs, t)
```

The final case here, for *Let* expressions, uses the function *tiBindGroup* presented in Section 11.6.3, to generate a list of assumptions as' for the variables defined in *bg*. All of these variables are in scope when we calculate a type *t* for the body *e*, which also serves as the type of the whole expression.

11.4 Alternatives

The representation of function bindings in following sections uses *alternatives*, represented by values of type *Alt*:

```
type Alt  =  ([Pat], Expr)
```

An *Alt* specifies the left and right hand sides of a function definition. With a more complete syntax for *Expr*, values of type *Alt* might also be used in the representation of lambda and case expressions.

For type inference, we begin by building a new list as' of assumptions for any bound variables, and use it to infer types for each of the patterns, as described in Section 11.2. Next, we calculate the type of the body in the scope of the bound variables, and combine this with the types of each pattern to obtain a single (function) type for the whole *Alt*:

```
tiAlt  ::  Infer Alt Type
tiAlt as (pats, e)
       =  do (ps, as', ts) ← tiPats pats
             (qs, t) ← tiExpr (as' ++ as) e
             return (ps ++ qs, foldr fn t ts)
```

In practice, we will often need to run the typechecker over a list of alternatives, *alts*, and check that the returned type

in each case agrees with some known type t. This process can be packaged up in the following definition:

```
tiAlts  ::  [Assump] → [Alt] → Type → TI [Pred]
tiAlts as alts t
        =  do psts ← mapM (tiAlt as) alts
              mapM (unify t) (map snd psts)
              return (concat (map fst psts))
```

Although we do not need it here, the signature for *tiAlts* would allow an implementation to push the type argument inside the checking of each *Alt*, interleaving unification with type inference instead of leaving it to the end. This is essential in extensions like the support for rank-2 polymorphism in Hugs where explicit type information plays a prominent role. Even in an unextended Haskell implementation, this could still help to improve the quality of type error messages.

11.5 Context Reduction

We have seen how lists of predicates are accumulated during type inference. In this section, we will describe how those predicates are used to construct inferred types. The Haskell report [10] provides only informal hints about this aspect of the Haskell typing, where both pragmatics and theory have important parts to play. We believe therefore that this is one of the areas where a more formal specification will be particularly valuable.

To understand the basic problem, suppose that we have run *tiExpr* over the body of a function f to obtain a set of predicates ps and a type t. At this point we could infer a type for f by forming the qualified type $qt = (ps :\Rightarrow t)$, and then quantifying over any variables in qt that do not appear in the assumptions. While this is permitted by the theory of qualified types, it is often not the best thing to do in practice. For example:

- The syntax of Haskell requires class arguments to be of the form $v\ t_1\ \ldots\ t_n$, where v is a type variable, and $t_1,\ldots,t_n$ are types (and $n \geq 0$). Predicates that do not fit this pattern must be broken down using *reducePred*. In some cases, this will result in predicates being eliminated. In others, where *reducePred* fails, it will indicate that a predicate is unsatisfiable, and will trigger an error diagnostic.
- Some of the predicates in ps may be repeated or, more generally, entailed by the other members of ps. These predicates can safely be deleted, leading to smaller and simpler inferred types.
- Some of the predicates in ps may contain only 'fixed' variables (i.e., variables appearing in the assumptions), so including those constraints in the inferred type will not be of any use [6, Section 6.1.4]. These predicates should be 'deferred' to the enclosing bindings.
- Some of the predicates in ps could be ambiguous, and might require defaulting to avoid a type error.

To deal with all of these issues, we use a process of *context reduction* whose purpose is to compute, from a given set of predicates ps, a set of 'deferred' predicates ds and a set of 'retained' predicates rs. Only retained predicates will be included in inferred types. The complete process is described by the following function:

```
reduce  ::  [Tyvar] → [Tyvar] → [Pred] → ([Pred], [Pred])
reduce fs gs ps  =  (ds, rs')
  where (ds, rs)  =  split fs ps
        rs'       =  useDefaults (fs ++ gs) rs
```

The first stage of this algorithm, which we call context splitting, is implemented by *split* and is described in Section 11.5.1. Its purpose is to separate the deferred predicates from the retained predicates, using *reducePred* as necessary. The second stage implemented by *useDefaults*, is described in Section 11.5.2. Its purpose is to eliminate ambiguities in the retained predicates, whenever possible. The *fs* and *gs* parameters specify appropriate sets of 'fixed' and 'generic' type variables, respectively. The former is just the set of variables appearing free in the assumptions, while the latter is the set of variables over which we would like to quantify. Any variable in *ps* that is not in either *fs* or *gs* may cause ambiguity, as we describe in Section 11.5.2.

11.5.1 Context Splitting

We will describe the process of splitting a context as the composition of three functions, each corresponding to one of the bulletted items at the beginning of Section 11.5.

```
split      ::  [Tyvar] → [Pred] → ([Pred], [Pred])
split fs   =   partition (all (`elem` fs) . tv)
           .   simplify []
           .   toHnfs
```

The first stage of this pipeline, implemented by *toHnfs*, uses *reducePred* to break down any inferred predicates into the form that Haskell requires:

```
toHnfs  ::  [Pred] → [Pred]
toHnfs  =   concat . map toHnf

toHnf      ::  Pred → [Pred]
toHnf p    =
  if inHnf p
    then [p]
    else case reducePred p of
           Nothing  →  error "context reduction"
           Just ps  →  toHnfs ps
```

The name *toHnfs* is motivated by similarities with the concept of *head-normal forms* in λ-calculus. The test to determine whether a given predicate is in the appropriate form is implemented by the following function:

```
inHnf              ::  Pred → Bool
inHnf (IsIn c t)   =   hnf t
  where hnf (TVar v)   =  True
        hnf (TCon tc)  =  False
        hnf (TAp t _)  =  hnf t
```

The second stage of the pipeline uses information about superclasses to eliminate redundant predicates. More precisely, if the list produced by *toHnfs* contains some predicate

p, then we can eliminate any occurrence of a predicate from *bySuper* p in the rest of the list. As a special case, this also means that we can eliminate any repeated occurrences of p, which always appears as the first element in *bySuper* p. This process is implemented by the *simplify* function, using an accumulating parameter to bulid up the final result:

```
simplify                :: [Pred] → [Pred] → [Pred]
simplify rs []          =  rs
simplify rs (p : ps)    =  simplify (p : (rs \\ qs)) (ps \\ qs)
  where qs              =  bySuper p
        rs \\ qs        =  [r | r ← rs, r `notElem` qs]
```

Note that we have used a modified version of the (\\\) operator; with the standard Haskell semantics for (\\\), we could not guarantee that all duplicate entries would be removed.

The third stage of context reduction uses *partition* to separate deferred predicates—i.e., those containing only fixed variables—from retained predicates. The set of fixed variables is passed in as the *fs* argument to *split*.

11.5.2 Applying Defaults

A type scheme $P \Rightarrow t$ is said to be *ambiguous* if P contains generic variables that do not also appear in t. From theoretical studies [1, 6], we know that we cannot guarantee a well-defined semantics for any term with an ambiguous type, which is why Haskell will not allow programs containing such terms. In this section, we describe the mechanisms that are used to detect ambiguity, and the defaulting mechanism that it uses to try to eliminate ambiguity.

Suppose that we are about to qualify a type with a list of predicates *ps* and that *vs* lists all known variables, both fixed and generic. An ambiguity occurs precisely if there is a type variable that appears in *ps* but not in *vs*. To determine whether defaults can be applied, we compute a triple (v, qs, ts) for each ambiguous variable v. In each case, *qs* is the list of predicates in *ps* that involve v, and *ts* is the list of types that could be used as a default value for v:

```
ambig :: [Tyvar] → [Pred] → [(Tyvar, [Pred], [Type])]
ambig vs ps
    = [(v, qs, defs v qs) |
            v ← tv ps \\ vs,
            let qs = [p | p ← ps, v `elem` tv p]]
```

If the *ts* component of any one of these triples turns out to be empty, then defaulting cannot be applied to the corresponding variable, and the ambiguity cannot be avoided. On the other hand, if *ts* is non-empty, then we will be able to substitute *head ts* for v and remove the predicates in *qs* from *ps*.

Given one of these triples (v, qs, ts), and as specified by the Haskell report [10, Section 4.3.4], defaulting is permitted if, and only if, all of the following conditions are satisfied:

- All of the predicates in *qs* are of the form *IsIn c* (*TVar v*) for some class *c*.
- All of the classes involved in *qs* are standard classes, defined either in the standard prelude or standard libraries. We assume that the list of these classes is provided by a constant *stdClasses* :: [*Class*].
- At least one of the classes involved in *qs* is a numeric class. Again, we assume that the list of these classes is provided by a constant *numClasses* :: [*Class*].
- That there is at least one type in the list of default types for the enclosing module that is an instance of all of the classes in *qs*. We assume that this list of types is provided in a constant *defaults* :: [*Type*].

These conditions are captured rather more succinctly in the following definition, which we use to calculate the third component of each triple:

```
defs        :: Tyvar → [Pred] → [Type]
defs v qs   =  [t | all ((TVar v) ==) ts,
                    all (`elem` stdClasses) cs,
                    any (`elem` numClasses) cs,
                    t ← defaults,
                    and [[] ⊢ IsIn c t | c ← cs]]
  where cs  =  [c | (IsIn c t) ← qs]
        ts  =  [t | (IsIn c t) ← qs]
```

The defaulting process can now be described by the following function, which generates an error if there are any ambiguous type variables that cannot be defaulted:

```
useDefaults          :: [Tyvar] → [Pred] → [Pred]
useDefaults vs ps
  | any null tss     =  error "ambiguity"
  | otherwise        =  ps \\ ps'
    where ams        =  ambig vs ps
          tss        =  [ts | (v, qs, ts) ← ams]
          ps'        =  [p | (v, qs, ts) ← ams, p ← qs]
```

A modified version of this process is required at the top-level, when type inference for an entire module is complete [10, Section 4.5.5, Rule 2]. In this case, *any* remaining type variables are considered ambiguous, and we need to arrange for defaulting to return a substitution mapping any such variables to their defaulted types:

```
topDefaults          :: [Pred] → Maybe Subst
topDefaults ps
  | any null tss     =  Nothing
  | otherwise        =  Just (zip vs (map head tss))
    where ams        =  ambig [] ps
          tss        =  [ts | (v, qs, ts) ← ams]
          vs         =  [v | (v, qs, ts) ← ams]
```

11.6 Binding Groups

The main technical challenge in this paper is to describe typechecking for binding groups. This area is neglected in most theoretical treatments of of type inference, often being regarded as a simple exercise in extending basic ideas. In Haskell, at least, nothing could be further from the truth! With interactions between overloading, polymorphic recursion, and the mixing of both explicitly and implicitly typed bindings, this is the most complex, and most subtle component of type inference. We will start by describing the treatment of explicitly typed bindings and implicitly typed bindings as separate cases, and then show how these can be combined.

11.6.1 Explicitly Typed Bindings

The simplest case is for explicitly typed bindings, each of which is described by the name of the function that is being defined, the declared type scheme, and the list of alternatives in its definition:

```
type Expl = (Id, Scheme, [Alt])
```

Haskell requires that each *Alt* in the definition of any given value has the same number of arguments in each left-hand side, but we do not need to enforce that restriction here.

Type inference for an explicitly typed binding is fairly easy; we need only check that the declared type is valid, and do not need to infer a type from first principles. To support the use of polymorphic recursion [4, 8], we will assume that the declared typing for i is already included in the assumptions when we call the following function:

```
tiExpl                  :: [Assump] → Expl → TI [Pred]
tiExpl as (i, sc, alts) =
  do (qs :⇒ t) ← freshInst sc
     ps ← tiAlts as alts t
     s ← getSubst
     let qs'      = apply s qs
         t'       = apply s t
         ps'      = [p | p ← apply s ps, not (qs' ⊢ p)]
         fs       = tv (apply s as)
         gs       = tv t' \\ fs
         (ds, rs) = reduce fs gs ps'
         sc'      = quantify gs (qs' :⇒ t')
     if sc /= sc' then
         error "signature too general"
       else if not (null rs) then
         error "context too weak"
       else
         return ds
```

This code begins by instantiating the declared type scheme *sc* and checking each alternative against the resulting type t. When all of the alternatives have been processed, the inferred type for i is $qs' :\Rightarrow t'$. If the type declaration is accurate, then this should be the same, up to renaming of generic variables, as the original type $qs :\Rightarrow t$. If the type signature is too general, then the calculation of sc' will result in a type scheme that is more specific than *sc* and an error will be reported.

In the meantime, we must discharge any predicates that were generated while checking the list of alternatives. Predicates that are entailed by the context qs' can be eliminated without further ado. Any remaining predicates are collected in ps' and passed as arguments to *reduce* along with the appropriate sets of fixed and generic variables. If there are any retained predicates after context reduction, then an error is reported, indicating that the declared context is too weak.

11.6.2 Implicitly Typed Bindings

Two complications occur when we deal with implicitly typed bindings. The first is that we must deal with groups of mutually recursive bindings as a single unit rather than inferring types for each binding one at a time. The second is Haskell's monomorphism restriction, which restricts the use of overloading in certain cases.

A single implicitly typed binding is described by a pair containing the name of the variable and a list of alternatives:

```
type Impl = (Id, [Alt])
```

The monomorphism restriction is invoked when one or more of the entries in a list of implicitly typed bindings is simple, meaning that it has an alternative with no left-hand side patterns. The following function provides a simple way to test for this condition:

```
restricted       :: [Impl] → Bool
restricted bs    = any simple bs
  where simple (i, alts) = any (null . fst) alts
```

Type inference for groups of mutually recursive, implicitly typed bindings is described by the following function:

```
tiImpls          :: Infer [Impl] [Assump]
tiImpls as bs    =
  do ts ← mapM (\_ → newTVar Star) bs
     let is    = map fst bs
         scs   = map toScheme ts
         as'   = zipWith (:>:) is scs ++ as
         altss = map snd bs
     pss ← sequence (zipWith (tiAlts as') altss ts)
     s ← getSubst
     let ps'      = apply s (concat pss)
         ts'      = apply s ts
         fs       = tv (apply s as)
         vss      = map tv ts'
         gs       = foldr1 union vss \\ fs
         (ds, rs) = reduce fs (foldr1 intersect vss) ps'
     if restricted bs then
         let gs'  = gs \\ tv rs
             scs' = map (quantify gs' . ([] :⇒)) ts'
         in return (ds ++ rs, zipWith (:>:) is scs')
       else
         let scs' = map (quantify gs . (rs :⇒)) ts'
         in return (ds, zipWith (:>:) is scs')
```

In the first part of this process, we extend *as* with assumptions binding each identifier defined in *bs* to a new type variable, and use these to type check each alternative in each binding. This is necessary to ensure that each variable is used with the same type at every occurrence within the defining list of bindings. (Lifting this restriction makes type inference undecidable [4, 8].) Next we use the process of context reduction to break the inferred predicates in ps' into a list of deferred predicates *ds* and retained predicates *rs*. The list *gs* collects all the generic variables that appear in one or more of the inferred types ts', but not in the list *fs* of fixed variables. Note that a different list is passed to *reduce*, including only variables that appear in *all* of the inferred types. This is important because all of those types will eventually be qualified by the same set of predicates, and we do not want any of the resulting type schemes to be ambiguous. The final step begins with a test to see if the monomorphism restriction should be applied, and then continues to calculate an assumption containing the principal types for each of the defined values. For an

unrestricted binding, this is simply a matter of qualifying over the retained predicates in *rs* and quantifying over the generic variables in *gs*. If the binding group is restricted, then we must defer the predicates in *rs* as well as those in *ds*, and hence we can only quantify over variables in *gs* that do not appear in *rs*.

11.6.3 Combined Binding Groups

Haskell requires a process of *dependency analysis* to break down complete sets of bindings—either at the top-level of a program, or within a local definition—into the smallest possible groups of mutually recursive definitions, and ordered so that no group depends on the values defined in later groups. This is necessary to obtain the most general types possible. For example, consider the following fragment from a standard prelude for Haskell:

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a []     = a

and xs           = foldr (&&) True xs
```

If these definitions were placed in the same binding group, then we would not obtain the most general possible type for `foldr`; all occurrences of a variable are required to have the same type at each point within the defining binding group, which would lead to the following type for `foldr`:

```
(Bool -> Bool -> Bool) -> Bool -> [Bool] -> Bool
```

To avoid this problem, we need only notice that the definition of `foldr` does not depend in any way on `&&`, and hence we can place the two functions in separate binding groups, inferring first the most general type for `foldr`, and then the correct type for `and`.

In the presence of explicitly typed bindings, we can refine the dependency analysis process a little further. For example, consider the following pair of bindings:

```
f   :: Eq a => a -> Bool
f x  = (x==x) || g True
g y  = (y<=y) || f True
```

Although these bindings are mutually recursive, we do not need to infer types for `f` and `g` at the same time. Instead, we can use the declared type of `f` to infer a type:

```
g   :: Ord a => a -> Bool
```

and then use this to check the body of `f`, ensuring that its declared type is correct.

Motivated by these observations, we will represent Haskell binding groups using the following datatype:

type *BindGroup* = ([*Expl*], [[*Impl*]])

The first component in each such pair lists any explicitly typed bindings in the group, while the second component breaks down any remaining bindings into a sequence of smaller, implicitly typed binding groups, arranged in dependency order. In choosing our representation for the abstract syntax, we have assumed that dependency analysis has been carried out prior to type checking, and that the bindings in each group have been organized into values of type *BindGroup* in an appropriate manner. For a correct implementation of the semantics specified in the Haskell report, we must place all of the implicitly typed bindings in a single group, even if a more refined decomposition would be possible. In addition, if that group is restricted, then we must also check that none of the explicitly typed bindings in the same *BindGroup* have any predicates in their type. With hindsight, these seem like strange restrictions that we might prefer to avoid in any further revision of Haskell.

A more serious concern is that the Haskell report does not indicate clearly whether the previous example defining `f` and `g` should be valid. At the time of writing, some implementations accept it, while others do not. This is exactly the kind of problem that can occur when there is no precise, formal specification! Curiously, however, the report does indicate that a modification of the example to include an explicit type for `g` would be illegal. This is a consequence of a throw-away comment specifying that all explicit type signatures in a binding group must have the same context up to renaming of variables [10, Section 4.5.2]. This is a syntactic restriction that can easily be checked prior to type checking. Our comments here, however, suggest that it is unnecessarily restrictive.

In addition to the function bindings that we have seen already, Haskell allows variables to be defined using pattern bindings of the form *pat* = *expr*. We do not need to deal directly with such bindings because they are easily translated into the simpler framework used in this paper. For example, a binding:

```
(x,y) = expr
```

can be rewritten as:

```
nv = expr
x  = fst nv
y  = snd nv
```

where `nv` is a new variable. The precise definition of the monomorphism restriction in Haskell makes specific reference to pattern bindings, treating any binding group that includes one as restricted. So, at first glance, it may seem that the definition of restricted binding groups in this paper is not quite accurate. However, if we use translations as suggested here, then it turns out to be equivalent: even if the programmer supplies explicit type signatures for `x` and `y` in the original program, the translation will still contain an implicitly typed binding for the new variable `nv`.

Now, at last, we are ready to present the algorithm for type inference of a complete binding group, as implemented by the following function:

```
tiBindGroup                :: Infer BindGroup [Assump]
tiBindGroup as (es, iss)   =
  do let as'  = [v :>: sc | (v, sc, alts) ← es]
     (ps, as'') ← tiSeq tiImpls (as' ++ as) iss
     qs ← mapM (tiExpl (as'' ++ as' ++ as)) es
     return (ps ++ concat qs, as'' ++ as')
```

The structure of this definition is quite straightforward. First we form a list of assumptions as' for each of the explicitly typed bindings in the group. Next, we use this to check

each group of implicitly typed bindings, extending the assumption set further at each stage. Finally, we return to the explicitly typed bindings to verify that each of the declared types is acceptable. In dealing with the list of implicitly typed binding groups, we use the following utility function, which typechecks a list of binding groups and accumulates assumptions as it runs through the list:

```
tiSeq   ::  Infer bg [Assump] → Infer [bg] [Assump]
tiSeq ti as []
        =   return ([], [])
tiSeq ti as (bs : bss)
        =   do (ps, as') ← ti as bs
               (qs, as'') ← tiSeq ti (as' ++ as) bss
               return (ps ++ qs, as'' ++ as')
```

11.6.4 Top-level Binding Groups

At the top-level, a Haskell program can be thought of as a list of binding groups:

```
type Program  =  [BindGroup]
```

Even the definitions of member functions in class and instance declarations can be included in this representation; they can be translated into top-level, explicitly typed bindings. The type inference process for a program takes a list of assumptions giving the types of any primitives, and returns a set of assumptions for any variables.

```
tiProgram           ::  [Assump] → Program → [Assump]
tiProgram as bgs    =   runTI $
  do (ps, as') ← tiSeq tiBindGroup as bgs
     s ← getSubst
     let ([], rs)   =   split [] (apply s ps)
     case topDefaults rs of
       Just s'      →   return (apply (s'@@s) as')
       Nothing      →   error "top-level ambiguity"
```

This completes our presentation of the Haskell type system.

12 Conclusions

We have presented a complete Haskell program that implements a type checker for the Haskell language. In the process, we have clarified certain aspects of the current design, as well as identifying some ambiguities in the existing, informal specification.

The type checker has been developed, type-checked, and tested using the "Haskell 98 mode" of Hugs 98 [7]. The full program includes many additional functions, not shown in this paper, to ease the task of testing, debugging, and displaying results. We have also translated several large Haskell programs—including the Standard Prelude, the Maybe and List libraries, and the source code for the type checker itself—into the representations described in Section 11, and successfully passed these through the type checker. As a result of these and other experiments we have good evidence that the type checker is working as intended, and in accordance with the expectations of Haskell programmers.

We believe that this typechecker can play a useful role, both as a formal specification for the Haskell type system, and as a testbed for experimenting with future extensions.

Acknowledgments

This paper has benefited from feedback from Lennart Augustsson, Stephen Eldridge, Tim Sheard, Andy Gordon, and from an anonymous referee. The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161.

References

[1] S. M. Blott. *An approach to overloading with polymorphism.* PhD thesis, Department of Computing Science, University of Glasgow, July 1991. (draft version).

[2] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, NM, January 1982.

[3] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Computer Science, University of Nottingham, November 1996.

[4] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[5] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[6] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[7] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs/`.

[8] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[9] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[10] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

[11] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 1965.

[12] P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.

Embedding Prolog in Haskell

Silvija Seres and *Mike Spivey* (Oxford University, UK)

Embedding PROLOG in HASKELL

Silvija Seres Michael Spivey

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.

Abstract

The distinctive merit of the declarative reading of logic programs is the validity of all the laws of reasoning supplied by the predicate calculus with equality. Surprisingly many of these laws are still valid for the procedural reading; they can therefore be used safely for algebraic manipulation, program transformation and optimisation of executable logic programs.

This paper lists a number of common laws, and proves their validity for the standard (depth-first search) procedural reading of PROLOG. They also hold for alternative search strategies, e.g. breadth-first search. Our proofs of the laws are based on the standard algebra of functional programming, after the strategies have been given a rather simple implementation in Haskell.

1 Introduction

Logic programming languages are traditionally explained in terms of their declarative and procedural semantics. For a given logic program, they are respectively considered its specification and its model of execution.

It is regarded as the responsibility of the programmer to ensure the consistency of the two readings. This paper aims to help in this, by codifying algebraic laws which apply equally to both readings. In some sense, a sufficient collection of these laws would provide an additional algebraic semantics for a logic language, intermediate between the declarative and procedural semantics. The general role of algebra in bridging the gap between abstract and concrete theories is argued in [8].

A proof of the validity of our laws in the declarative reading is unnecessary, because they express properties of Boolean algebra. To prove that they are true of the procedural reading requires us to construct a model of execution. This we do by implementing the operators of the logic language as a library of higher order functions in the functional language Haskell. This makes available all the algebraic reasoning principles of functional programming [3], from which it is quite straightforward to derive the laws we need. Many of these are familiar

properties of categorical monads, but a knowledge of category theory is not needed for an understanding of this paper.

It is worth stressing that our implementation is a *shallow* embedding of a logic language in a functional one; it is *not* the same as building an interpreter. We do not extend the base functional language; rather, we implement *in* the language a set of functions designed to support unification, resolution and search.

Our implementation is strikingly simple, and the basic ideas that it builds upon are not new. The embedding of a logic language to a functional one by translating every predicate to a function was explored in e.g. LogLisp [14, 15] or PopLog [11], although the base language was non-lazy in each case. The use of the lazy stream-based execution model to compute the possibly infinite set of answers is also well known, e.g. [1]. Nevertheless, we believe that the combination of these two known ideas is well worth our attention, and the *algebraic* semantics for logic programs that naturally arises from our embedding is a convincing example.

To some extent, use of our library of functions will give functional programmer a small taste of the power of a *functional logic language*. But current functional logic languages are much more powerful; they embody both rewriting and resolution and thereby result in a functional language with the capability to solve arbitrary constraints for the values of variables. The list of languages that have been proposed in an attempt to incorporate the expressive power of both functional and logic paradigms is long and impressive [2, 6]; some notable examples are Kernel-LEAF [5], Curry [7], Escher [9] and Babel [12]. Our research goal is different from the one set by these projects. They aspire to build an efficient language that can offer programmers the most useful features of both worlds; to achieve this additional expressivity they have to adopt somewhat complicated semantics. Our present goal is a conspicuous declarative and operational semantics for the embedding, rather that maximal expressivity. Nevertheless, the extensions of our embedding to incorporate both narrowing and residuation in its operational semantics do not seem difficult and we hope to make them a subject of our further work.

In this paper we use Prolog and Haskell as our languages of choice, but the principles presented are general. Prolog is chosen because it is the dominant logic language, although we only implement the pure declarative features of it, i.e., we ignore the impure but practically much used features like cut, `assert` and `retract`, although the cut is quite an easy extension of our models. Haskell is chosen because it is a lazy functional language with types and lambda-abstractions, but any other language with these properties could be used.

In the remainder of the paper we proceed to describe the syntax of the embedding and the implementation of the primitives in sections 2 and 3. In section 4 we list some of the algebraic properties of the operators and in section 5 we study the necessary changes to the system to accommodate different search strategies. We conclude the paper with section 6 where we discuss related work and propose some further work in this setting.

2 Syntax

Prolog offers the facility of defining a predicate in many clauses and it allows the applicability of each clause to be tested by pattern matching on the formal parameter list. In our implementation of Prolog, we have to withdraw these notational licences, and require the full logical meaning of the predicate to be defined in a single equation, with the unifications made explicit on the right hand side, together with the implicit existential quantification over the fresh variables.

In the proposed embedding of Prolog into a functional language, we aim to give rules that allow any pure Prolog predicate to be translated into a Haskell function with the same meaning. To this end, we introduce two data types, *Term* and *Predicate*, into our functional language, together with the following four operations:

$$(\&), (\|\|) : \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate},$$
$$(\doteq) : \mathit{Term} \longrightarrow \mathit{Term} \longrightarrow \mathit{Predicate},$$
$$\mathit{exists} : (\mathit{Term} \longrightarrow \mathit{Predicate}) \longrightarrow \mathit{Predicate}.$$

The intention is that the operators & and || denote conjunction and disjunction of predicates, $\doteq$ forms a predicate expressing the equality of two terms, and the operation *exists* expresses existential quantification. We shall abbreviate the expression *exists* $(\lambda x \to p\ x)$ by the form $\exists x \to p\ x$ in this paper, although the longer form shows how the expression can be written in any lazy functional language that has λ-expressions. We shall also write $\exists x, y \to p(x, y)$ for $\exists x \to (\exists y \to p(x, y))$.

These four operations suffice to translate any pure Prolog program, provided we are prepared to exchange pattern matching for explicit equations, to bind local variables with explicit quantifiers, and to gather all the clauses defining a predicate into a single equation. These steps can be carried out systematically, and could easily be automated. As an example, we take the well-known program for append:

```
append([], Ys, Ys) :- .
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

As a first step, we remove any patterns and repeated variables from the head of each clause, replacing them by explicit equations written at the start of the body. These equations are computed by unification in Prolog.

```
append(Ps, Qs, Rs) :-
    Ps = [], Qs = Rs.
append(Ps, Qs, Rs) :-
    Ps = [X|Xs], Rs = [X|Ys], append(Xs, Qs, Ys).
```

The head of each clause now contains only a list of distinct variables, and by renaming if necessary we can ensure that the list of variables is the same in

each clause. We complete the translation by joining the clause bodies with the || operation, the literals in a clause with the & operation, and existentially quantifying any variables that appear in the body but not in the head of a clause:

$$\begin{aligned} &\mathit{append}(\mathit{Ps}, \mathit{Qs}, \mathit{Rs}) = \\ &\qquad (\mathit{Ps} \doteq \mathit{nil} \;\&\; \mathit{Qs} \doteq \mathit{Rs}) \;\|\; \\ &\qquad (\exists \mathit{X}, \mathit{Xs}, \mathit{Ys} \rightarrow \mathit{Ps} \doteq \mathit{cons}(\mathit{X}, \mathit{Xs}) \;\&\; \mathit{Rs} \doteq \mathit{cons}(\mathit{X}, \mathit{Ys}) \;\& \\ &\qquad\qquad\qquad \mathit{append}(\mathit{Xs}, \mathit{Qs}, \mathit{Ys}) \;). \end{aligned}$$

Here *nil* is used for the value of type *Term* that represents the empty list, and *cons* is written for the function on terms that corresponds to the Prolog list constructor `[|]`. We assume the following order of precedence on the operators, from highest to lowest: $\doteq, \&, \|, \exists$.

The function *append* defined by this recursive equation has the following type:

$$\mathit{append} :: (\mathit{Term}, \mathit{Term}, \mathit{Term}) \longrightarrow \mathit{Predicate}.$$

The Haskell function *append* is constructed by making the *declarative* reading of the Prolog predicate explicit. However, the relationship between the Haskell function and the Prolog predicate extends beyond their declarative semantics. The next section shows that the *procedural* reading of the Prolog predicate is also preserved through the implementation of the functions & and ||. The embedding essentially allows the mapping of the computation of the Prolog program into lazy lists by embedding the structure of a SLD-tree of a Prolog program into a Haskell stream.

3 Implementation

The translation described above depends on the four operations &, ||, $\doteq$ and *exists*. We now give definitions to the type of predicates and to these four operations that correspond to the depth-first search of Prolog. Later, we shall be able to give alternative definitions that correspond to breadth-first search, or other search strategies based on the search tree of the program.

The key idea is that each predicate is a function that takes an 'answer', representing the state of knowledge about the values of variables at the time the predicate is solved, and produces a lazy stream of answers, each corresponding to a solution of the predicate that is consistent with the input. This approach is similar to that taken by Wadler [18]. An unsatisfiable query results in an empty stream, and a query with infinitely many answers results in an infinite stream.[1]

$$\textbf{type}\ \mathit{Predicate} = \mathit{Answer} \longrightarrow \mathit{Stream}\ \mathit{Answer}.$$

[1]For clarity, we use the type constructor *Stream* to denote possibly infinite streams, and *List* to denote finite lists. In a lazy functional language, these two concepts share the same implementation.

An answer is (in principle) just a substitution, but we augment the substitution with a counter that tracks the number of variables that have been used so far, so that a fresh variable can be generated at any stage by incrementing the counter. A substitution is represented as a list of (variable, term) pairs, and the Haskell data-type *Term* is a straightforward implementation of Prolog's term type:

$$\begin{aligned}
&\textbf{type } \mathit{Answer} = (\mathit{Subst}, \mathit{Int}),\\
&\textbf{type } \mathit{Subst} = [\,(\mathit{Var}, \mathit{Term})\,],\\
&\textbf{data } \mathit{Term} = \mathit{Func}\ \mathit{Fname}\ [\,\mathit{Term}\,] \mid \mathit{Var}\ \mathit{Vname},\\
&\textbf{type } \mathit{Fname} = \mathit{String},\\
&\textbf{data } \mathit{Vname} = \mathit{Name}\ \mathit{String} \mid \mathit{Auto}\ \mathit{Int}.
\end{aligned}$$

Constants are functions with arity 0, in other words they are given empty argument lists. For example the Prolog list `[a,b]` can be represented in the embedding as $\mathit{Func}\ \text{"}\mathit{cons}\text{"}\ [\mathit{Func}\ \text{"}a\text{"}\ [\,],\ \mathit{Func}\ \text{"}\mathit{cons}\text{"}\ [\ldots]]$. With the use of the simple auxiliary functions *cons*, *atom* and *nil* the same Prolog list can be embedded as the Haskell expression $\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil})$.

We can now give definitions for the four operators. The operators & and || act as predicate combinators; they slightly resemble the notion of *tacticals* [13], but in our case they combine the computed streams of answers, rather that partially proved statements.

The || operator simply concatenates the streams of answers returned by its two operands:

$$\begin{aligned}
&(\|) :: \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate}\\
&(p \,\|\, q)\ x = p\ x +\!\!+ q\ x.
\end{aligned}$$

This definition implies that the answers are returned in a left-to-right order as in Prolog. If the left-hand argument of || is unsuccessful and returns an empty answer stream, it corresponds to an unsuccessful branch of the search tree in Prolog and backtracking is simulated by evaluating the right-hand argument.

For the & operator, we start with applying the first argument to the incoming answer; this produces a stream of answers, to each of which we apply the second argument of &. Finally, we concatenate the resulting stream of streams into a single stream:

$$\begin{aligned}
&(\&) :: \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate}\\
&p \,\&\, q = \mathit{concat} \cdot \mathit{map}\ q \cdot p.
\end{aligned}$$

Because of Haskell's lazy evaluation, the function p returns answers only when they are needed by the function q. This corresponds nicely with the backtracking behaviour of Prolog, where the predicate $p \,\&\, q$ is implemented by enumerating the answers of p one at a time and filtering them with the predicate q. Infinite list of answers in Prolog are again modelled gracefully with infinite streams.

We can also define primitive predicates *true* and *false*, one corresponding to immediate success and the other to immediate failure:

$$\begin{array}{ll} \mathit{true} :: \mathit{Predicate} & \mathit{false} :: \mathit{Predicate} \\ \mathit{true}\ x = [x]. & \mathit{false}\ x = [\,]. \end{array}$$

The pattern matching of Prolog is implemented by the operator $\doteq$. It is defined in terms of a function *unify* which implements J.A. Robinson's standard algorithm for s unification of two terms relative to a given input substitution. The type of *unify* is thus:

$$\mathit{unify} :: \mathit{Subst} \longrightarrow (\mathit{Term}, \mathit{Term}) \longrightarrow \mathit{List}\ \mathit{Subst}.$$

More precisely, the result of $\mathit{unify}\ s\ (t,u)$ is either $[s \triangleright r]$, where r is a most general unifier of $t[s]$ and $u[s]$, or $[\,]$ if these two terms have no unifier.[2] Thus if $\mathit{unify}\ s\ (t,u) = [s']$, then s' is the most general substitution such that $s \sqsubseteq s'$ and $t[s'] = u[s']$. The coding is routine and therefore omitted.

The $\doteq$ operator is just a wrapper around *unify* that passes on the counter for fresh variables:

$$\begin{array}{l} (\doteq) :: \mathit{Term} \longrightarrow \mathit{Term} \longrightarrow \mathit{Predicate} \\ (t \doteq u)\ (s,n) = [(s',n) \mid s' \leftarrow \mathit{unify}\ s\ (t,u)] \end{array}$$

Finally, the operator *exists* is responsible for allocating fresh names for all the local (or existentially quantified) variables in the predicates. This is necessary in order to guarantee that the computed answer is the most general result. It is defined as follows:

$$\begin{array}{l} \mathit{exists} :: (\mathit{Term} \longrightarrow \mathit{Predicate}) \longrightarrow \mathit{Predicate} \\ \mathit{exists}\ p\ (s,n) = p\ (\mathit{makevar}\ n)\ (s, n+1), \end{array}$$

where *makevar* n is a term representing the n'th generated variable. The slightly convoluted flow of information here may be clarified by a small example. The argument p of *exists* will be a function that expects a variable, such as $(\lambda X \rightarrow \mathit{append}(t, X, u))$. We apply this function to a newly-invented variable $v = \mathit{makevar}\ n$ to obtain the predicate $\mathit{append}(t, v, u)$, and finally apply this predicate to the answer $(s, n{+}1)$, in which all variables up to the n'th are marked as having been used.

The function *solve* evaluates the main query. It simply applies its argument, the predicate of the query, to an answer with an empty substitution and a zero variable counter, and converts the resulting stream of answers to a stream of strings.

$$\begin{array}{l} \mathit{solve} :: \mathit{Predicate} \longrightarrow \mathit{Stream}\ \mathit{String} \\ \mathit{solve}\ p = \mathit{map}\ \mathit{print}\ (p\ ([\,], 0)), \end{array}$$

[2] We use $s \triangleright r$ to denote composition of substitutions s and r, and $t[s]$ to denote the instance of term t under substitution s. We use $s \sqsubseteq s'$ to denote the preorder on substitutions that holds iff $s' = s \triangleright r$ for some substitution r.

where *print* is a function that converts an answer to a string by having pruned it to show only the values of the original query variables This is the point where all the internally generated variables are filtered out in our present implementation, but another, possibly cleaner, solution might be to let the $\exists$ operator do this filtering task before it returns.

We do not provide proofs of the soundness and completeness relative to the procedural reading of Prolog since we feel that the encoding we have described is about the simplest possible mechanised formal definition of a Prolog-like procedural reading. Nevertheless, a soundness proof for the embedding could be carried out relative to the declarative semantics by defining of a mapping *decl* between our embedding and a declarative semantics of a logic program. Given a function *herb* with type $Answer \longrightarrow Set\ Subst$:

$$herb(s, n) = \{s; t \mid t \in Subst\},$$

where $herb(s, n)$ describes a set of all substitutions that refine (i.e. extend) the substitution part s of the input answer (s, n). The mapping from our embedding to the declarative semantics can then be defined as:

$$decl = (fold\ union)\ (map\ herb).$$

Namely, if P is a predicate then $decl \cdot P$ is its declarative semantics. A soundness proof for the embedding would then be obtained by proving the equations:

$$decl \cdot (P \parallel Q) \subseteq (decl \cdot P) \cup (decl \cdot Q),$$
$$decl \cdot (P \,\&\, Q) \subseteq (decl \cdot P) \cap (decl \cdot Q),$$

for the operators $\parallel$ and &, and similar equations for the operators $\exists$ and $\doteq$.

4 Algebraic Laws

The operators & and $\parallel$ enjoy many algebraic properties as a consequence of their simple definitions in terms of streams. We can deduce directly from the implementation of & and *true* that the & operator is associative with unit element *true*. This is a consequence of the fact that *map*, *concat* and *true* form a structure that category theory calls a *monad*, and the composition operator & is obtained from this by a standard construction called *Kleisli composition*. We wish to show that these properties of the logic programming primitives & and *true*, and several others regarding also $\parallel$ and *false*, can be alternatively proved with no reference to category theory. The proofs we sketch show how a standard tool in functional programming, equational reasoning, can be applied to logic programming by means of our embedding.

All the algebraic properties we quote here can be proved equationally using only the definitions of the operators and the standard laws (see [4]) for *concat*,

map and functional composition. As an example, given:

$$map\ f \cdot concat = concat \cdot map\ (map\ f), \tag{1}$$
$$concat \cdot concat = concat \cdot map\ concat, \tag{2}$$
$$map\ (f \cdot g) = (map\ f) \cdot (map\ g, \tag{3}$$

we can prove the associativity of & by the following rewriting:

$$\begin{aligned}
&(p\ \&\ q)\ \&\ r \\
&= concat \cdot map\ r \cdot concat \cdot map\ q \cdot p && \text{by defn. of \&} \\
&= concat \cdot concat \cdot map\ (map\ r) \cdot map\ q \cdot p && \text{by (1)} \\
&= concat \cdot map\ concat \cdot map\ (map\ r) \cdot map\ q \cdot p && \text{by (2)} \\
&= concat \cdot map\ (concat \cdot map\ r \cdot q) \cdot p && \text{by (3)} \\
&= p\ \&\ (q\ \&\ r). && \text{by defn. of \&}
\end{aligned}$$

The proofs of the following properties are at least as elementary as this. The predicate *false* is a left zero for &, but this operator is strict in its left argument, so *false* is not a right zero. This corresponds to the feature of Prolog that *false* & q has that same behaviour as *false*, but p & *false* may fail infinitely if p does. Owing to the properties of *concat* and [ ], the || operator is associative and has *false* as a left and right identity.

Other identities that are satisfied by the connectives of propositional logic are not shared by our operators because in our stream-based implementation, answers are produced in a definite order and with definite multiplicity. This behaviour mirrors the operational behaviour of Prolog. For example, the || operator is not idempotent, because *true* || *true* produces its input answer twice as an output, but *true* itself produces only one answer. The & operator also fails to be idempotent, because the predicate

$$(true \parallel true)\ \&\ (true \parallel true)$$

produces the same answer four times rather than just twice.

We might also expect

$$p\ \&\ (q \parallel r) = (p\ \&\ q) \parallel (p\ \&\ r),$$

that is, for & to distribute over ||, but this is not the case. For a counterexample, take for p the predicate $X \doteq a \parallel X \doteq b$, for q the predicate $Y \doteq c$, and for r the predicate $Y \doteq d$. Then the left-hand side of the above equation produces the four answers [X=a, Y=c]; [X=a, Y=d]; [X=b, Y=c]; [X=b, Y=d] in that order, but the right-hand side produces the same answers in the order [X=a, Y=c]; [X=b, Y=c]; [X=a, Y=d]; [X=b, Y=d].

However, the other distributive law,

$$(p \parallel q)\ \&\ r = (p\ \&\ r) \parallel (q\ \&\ r),$$

does hold, and it is vitally important to the unfolding steps of program transformation. The simple proof depends on the fact that both *map r* and *concat* are homomorphisms with respect to ++:

$$
\begin{aligned}
&((p \mid\mid q) \,\&\, r)\ x \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x +\!\!+ q\ x)) && \text{by defn. of } \mid\mid, \,\& \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x) +\!\!+ \mathit{map}\ r\ (q\ x)) && \mathit{map} \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x)) +\!\!+ \mathit{concat}\ (\mathit{map}\ r\ (q\ x)) && \mathit{concat} \\
&= ((p \,\&\, r) \mid\mid (q \,\&\, r))\ x. && \text{by defn. of } \&
\end{aligned}
$$

The declarative reading of logic programs suggests that also the following properties of $\doteq$ and $\exists$ ought to hold, where $p\ x$ and $q\ x$ are predicates and u is a term not containing x:

$$
\begin{aligned}
&(\exists x \to p(x) \mid\mid q(x)) = (\exists x \to p(x)) \mid\mid (\exists x \to q(x)), \\
&(\exists x \to x \doteq u \,\&\, p(x)) = p(u), \\
&(\exists x \to (\exists y \to p\ (x, y))) = (\exists y \to (\exists x \to p(x, y))).
\end{aligned}
$$

These properties are important in program transformations that manipulate quantifiers and equations, since they allow local variables to be introduced and eliminated, and allow equals to be substituted for equals in arbitrary formulas.

However, these properties of $\doteq$ and $\exists$ depend on properties of predicates p and q that are not shared by all functions of this type, but are shared by all predicates that are defined purely in terms of our operators. In future work, we plan to formulate precisely the 'healthiness' properties of definable predicates on which these transformation laws depend, such as monotonicity and substitutivity.

It might be seen as a weakness of our approach based on a 'shallow' embedding of Prolog in Haskell that these properties must be expressed in terms of the weak notion of a *predicate* definable in terms of our operators, when a 'deep' embedding (i.e., an interpreter for Prolog written in Haskell) would allow us to formulate and prove them as an inductive property of *program texts*. We believe that this is a price well worth paying for the simplicity and the clear declarative and operational semantics of our embedding.

5 Different Search Strategies

Our implementation of ||, together with the laziness of Haskell, causes the search for answers to behave like depth-first search in Prolog: when computing $p\ x +\!\!+ q\ x$ all the answers corresponding to the $p\ x$ part of the search tree are returned before the other part is explored. A fair *search* strategy would share the computation effort more evenly between the two parts. Similarly, our implementation of & results in a left-to-right selection of the literals of a clause. A fair *selection* rule would allow one to chose the literals in a different order.

One possible solution (inspired by [10]) is to *interleave* the streams of answers, taking one answer from each stream in turn. A function *twiddle* that interleaves two lists can be defined as:

$$\begin{aligned}
&\mathit{twiddle} :: [a] \longrightarrow [a] \longrightarrow [a] \\
&\mathit{twiddle}\ [\,]\ \mathit{ys} = \mathit{ys} \\
&\mathit{twiddle}\ (x : \mathit{xs})\ \mathit{ys} = x : (\mathit{twiddle}\ \mathit{ys}\ \mathit{xs}).
\end{aligned}$$

The operators $\|$ and & can be redefined by replacing $+\!\!+$ with *twiddle* and recalling that $\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$:

$$\begin{aligned}
&(p \,\|\, q)\ x = \mathit{twiddle}\ (p\ x)\ (q\ x) \\
&(p\ \&\ q)\ x = \mathit{foldr}\ \mathit{twiddle}\ [\,] \cdot \mathit{map}\ q \cdot p).
\end{aligned}$$

This implementation of & is fairer, producing in a finite time solutions of q that are based on later solutions returned by p, even if the first such solution produces an infinite stream of answers from q. The original implementation of & produces all solutions of q that are based on the first solution produced by p before producing any that are based on the second solution from p.

Note that this implementation of operators does *not* give breadth-first search of the search tree; it deals with infinite success but not with infinite failure. Even in the interleaved implementation, the first element of the answer list has to be computed before we can 'switch branches'; if this takes an infinite number of steps the other branch will never be reached.

To implement breadth-first search in the embedding, the *Predicate* data-type needs to be changed. It is no longer adequate to return a single, flat stream of answers; this model is not refined enough to take into account the number of *computation steps* needed to produce a single answer. The key idea is to let *Predicate* return a stream of lists of answers, where each list represents the answers reached at the same depth, or level, of the search tree. These lists of answers with the same cost are always finite since there is only a finite number of nodes at each level of the search tree. The new type of *Predicate* is thus:

$$\mathit{Predicate} :: \mathit{Answer} \longrightarrow \mathit{Stream}\ (\mathit{List}\ \mathit{Answer}).$$

Intuitively, each successive list of answers in the stream contains the answers with the same computational "cost". The cost of an answer increases with every resolution step in its computation. This can be captured by adding a new function *step* in the definition of predicates. For example, *append* should be coded as:

$$\begin{aligned}
&\mathit{append}(Ps, Qs, Rs) = \\
&\qquad \mathit{step}((Ps \doteq \mathit{nil}\ \&\ Qs \doteq Rs) \,\|\, \\
&\qquad\qquad (\exists X, Xs, Ys \rightarrow Ps \doteq \mathit{cons}(X, Xs)\ \&\ Rs \doteq \mathit{cons}(X, Ys)\ \& \\
&\qquad\qquad\qquad \mathit{append}(Xs, Qs, Ys))).
\end{aligned}$$

In the depth-first model, *step* is the identity function on predicates, but in the breadth-first model it is defined as follows:

$$step :: Predicate \longrightarrow Predicate$$
$$step\ p\ x = [\,] : (p\ x).$$

Thus, in the stream returned by *step* p, there are no answers of cost 0, and for each n, the answers of *step* p with cost $n{+}1$ are the same as the answers of p that have cost n.

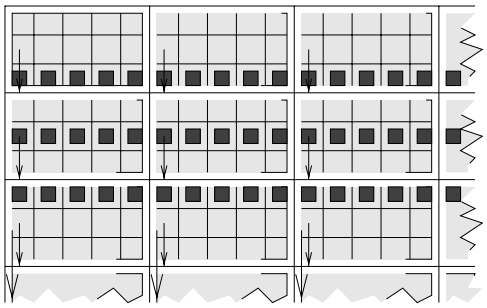
The implementations of the *Predicate* combinators || and & need to be changed so that they no longer operate on lists but on streams of lists. They must preserve the cost information that is embedded in the input lists. Since the cost corresponds to the level of the answer in the search tree, only resolution steps are charged for, while the applications of ||, & and *equals* are cost-free. The || operator simply zips the two streams into a single one, by concatenating all the sublists of answers with the same cost. If the two streams are of different lengths, the zipping must not stop when it reaches the end of the shorter stream. We give the name *mergewith* to a specialized version of *zipwith* that has this property, and arrive at this implementation of || in the breadth-first model:

$$(p \parallel q)\ x = mergewith\ (+\!+)\ (p\ x)\ (q\ x).$$

The implementation of & is harder. The cost of each of the answers to $(p\ \&\ q)$ is a sum of the costs of the computation of p and the computation of q. The idea is first to compute all the answers, and then to flatten the resulting stream of lists of streams of lists of answers to a stream of lists of answers according to the cost. This flattening is done by the *shuffle* function which is explained below. The &-operator is thus:

$$p\ \&\ q = shuffle \cdot map\ (map\ q) \cdot p$$

We write S for streams and L for finite lists for sake of brevity. The result of $map\ (map\ q) \cdot p$ is of type *SLSL*. It can be visualised as a matrix of matrices, where each element of the outer matrix corresponds to a single answer of p. Each such answer is used as an input to q and consequently gives rise to a new stream of lists of answers, which are represented by the elements of the inner matrices. The rows of both the main matrix and the sub-matrices are finite, while the columns of both can be infinite. For example, the answers of $map\ (map\ q) \cdot p$ with cost 2 are marked in the drawing below:



The function *shuffle* collects all the answers marked in the drawing into a single list, the third in the resulting stream of lists of answers. It is given an *SLSL* of answers, and it has to return an *SL*. Two auxiliary functions are required to do this: *diag* and *transpose*. A stream of streams is converted to a stream of lists by *diag*, and a list of streams can be converted to a stream of lists by *transpose*:

$$diag :: Stream\ (Stream\ a) \longrightarrow Stream\ (List\ a)$$
$$diag\ xss = [[(xss\ !\ i)\ !\ (n - i) \mid i \leftarrow [0..n]] \mid n \leftarrow [0..]],$$

$$transpose :: List\ (Stream\ a) \longrightarrow Stream\ (List\ a)$$
$$transpose\ xss = map\ hd\ xss : transpose\ (map\ tl\ xss).$$

Given *diag* and *transpose*, the function *shuffle* can be implemented as follows. The input to *shuffle* is of type *SLSL*. The application of *map transpose* swaps the middle *SL* to a *LS*, and gives *SSLL*. Then the application of *diag* converts the outermost *SS* to *SL* and returns *SLLL*. This can now be used as input to *map* $(concat \cdot concat)$ which flattens the three innermost levels of lists into a single list, and returns *SL*:

$$shuffle = map\ (concat \cdot concat) \cdot diag \cdot map\ transpose.$$

A very interesting aspect of this breadth-first model of logic programming is that all the algebraic laws listed in the previous section still hold, if we ignore the ordering of the answers within each sublist in the main stream. This can be achieved by implementing the type of predicates as a function from answers to streams of *bags* of answers. Each of the bags contains the answers with the same computational cost, so we know that all the bags are finite. This is because there are only a finite number of branches in each node in the search tree. Hence all the equalities in our laws are still computable.

To implement *both* depth-first search and breadth-first search in the embedding, the model has to be further refined. It is not sufficient to implement predicates as functions returning streams of answer lists; they have to operate on lists of trees. The operators || and & are redefined to be operations on lists of trees, where the first one connects two lists of trees in a single one and the second 'grafts' trees with small subtrees at the leaves to form normal trees. If just trees were used, rather than lists of trees, $p \parallel q$ would have to combine their trees of answers by inserting them under a new parent node in a new tree, but that would increase the cost of each answer to $p \parallel q$ by one. We describe this general model fully in [16].

It is interesting how concise the definitions of || and & remain in all three models. To recapitulate the three definitions of & in the depth-first model, breadth-first model and the tree model which accommodates both search strategies, respectively:

$$p\ \&\ q = concat \cdot map\ q \cdot p,$$
$$p\ \&\ q = shuffle \cdot map\ (map\ q) \cdot p,$$
$$p\ \&\ q = graft \cdot treemap\ q \cdot p.$$

These closely parallel definitions hint at a deeper algebraic structure, and in fact the definitions are all instances of the so-called Kleisli construction from category theory. Even greater similarities between the three models exist, and we give a more detailed study of the relation between the three in [16].

6 Further Work

The work presented in this paper has not addressed the question of an efficient implementation of these ideas, although a language implementation based on our embedding is conceivable. Rather, this work is directed towards producing and using a theoretical tool (with a simple implementation) for the analysis of different aspects of logic programs. The simplicity is the key idea and the main strength of our embedding, and it has served well in opening several directions for further research.

We are presently working on two applications of the embedding. One is a study of program transformation by equational reasoning, using the algebraic laws of the embedding. The other is a categorical study of a model in which trees are used as the data-structure for the answers, and we show that there exists a *morphism of monads* between this most general model and the two models that is presented in this paper. This line of research is inspired by [17, 19].

Among other questions that we plan to address soon are also the implementation of higher-order functions and the implementation of nested functions in the embedded predicates. Constraint logic programming also has a simple model in our embedding: one has to pass equations (instead of substitutions) as parts of answers. These equations are evaluated when they become sufficiently instantiated. An efficient language implementation is also a challenging goal in this setting.

References

[1] Abelson and Sussman. *Structure and Interpretation of Computer Programs*, chapter 4. 1985.

[2] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3(3):317–236, 1986.

[3] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[4] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[5] E. Giovanetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2), 1991.

[6] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20):583–628, 1994.

[7] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[8] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[9] J.W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[10] R. McPhee and O. de Moor. Compositional logic programming. In *Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming*, Report 96-28. Technische Universität Berlin, 1996.

[11] Mellish and Hardy. Integrating prolog in the POPLOG environment. In J. Campbell, editor, *Implementations of Prolog*. 1984.

[12] J. Moreno-Navarro and M. Roderiguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–223, 1992.

[13] L.C. Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Computer Journal*, (28), 1985.

[14] J.A. Robinson. Beyond LogLisp: combining functional and relational programming in a reduction setting. *Machine intelligence*, 11, 1988.

[15] J.A. Robinson and E.E. Sibert. LogLisp: An alternative to Prolog. *Machine Intelligence*, 10, 1982.

[16] S. Seres, J.M. Spivey, and C.A.R. Hoare. Algrebra of logic programming. submitted to International Conference on Logic Programming, 1999.

[17] J.M. Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*. Springer LNCS 375, 1989.

[18] P. Wadler. How to replace failure by a list of successes. In *2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[19] P. Wadler. The essence of functional programming. In *19'th Annual Symposium on Principles of Programming Languages*, January 1992.

Logical Abstractions in Haskell

Nancy A. Day, *John Launchbury* and *Jeff Lewis* (Oregon Graduate Institute, USA)

Logical Abstractions in Haskell

Nancy A. Day John Launchbury Jeff Lewis

Oregon Graduate Institute of Science & Technology

Abstract

We describe a generalization of the Haskell Boolean type, which allows us to use existing decision procedures for reasoning about logical expressions. In particular, we have connected Haskell with a Binary Decision Diagram (BDD) package for propositional logic, and the Stanford Validity Checker for reasoning in quantifier-free, first-order logic. We have defined referentially transparent interfaces to these packages allowing the user to ignore the details of their imperative implementations. We found that having a tight connection between the provers and Haskell allows Haskell to serve as a meta-language enhancing the capabilities of the provers. We illustrate the use of these packages for reasoning about a sort algorithm and a simple microprocessor model. In the sort example, the parametric nature of Haskell's polymorphism is used to lift the result of the BDD analysis to arbitrary datatypes.

1 Introduction

Here's some old advice: go through life like a swimming duck — remain calm and unruffled on the surface, but paddle like fury underneath. This advice applies to datatypes in programming languages. Consider the `Integer` datatype in Haskell. As far as the Glasgow Haskell compiler is concerned, the operations of `Integer` are implemented with calls to the GNU multi-precision arithmetic library, with all its ancillary mess of manipulating storage and pointers. Above the surface, however, is quite another story. As far as the user is concerned, `Integer` is just another numeric datatype with the usual arithmetic operations defined on it. It is quite possible to use `Integer` without thinking about the implementation mechanism at all. Of course, the plain numeric interface provided by Haskell is far poorer than the rich variety of methods provided by the full library. However, experience suggests that, for most users, a simple interface to a complex implementation provides far more benefit than a complex interface used simply.

The goal of this paper is set out on the same program for logical types like booleans. Excellent packages are now available that implement decision procedures for different logics, and we wondered whether clean interfaces could be built to allow the details of the decision procedures to be hidden under the surface.

Haskell's type class system was designed to solve a thorny problem in language design: how to combine overloading and polymorphism of numeric operators. The problem was motivated by the variety of numeric types. The solution was general enough to also solve several similar problems involving equality and printing. But, the notion of overloading booleans just didn't arise. However, several recent examples have made it clear that it's useful to be able to overload even simple types like booleans.

The Fran work on reactive animations demonstrates this point nicely [9]. In Fran, datatypes are lifted over time. An integer, for example, is replaced by a function from time to integer, and the numeric operations are defined pointwise. The same is done for equality. Are two time-varying integers equal? The answer is a time-varying boolean. By defining the boolean operations pointwise, it is easy to see that functions from time to `Bool` are fully "boolean".

Another example, and one which is the direct inspiration for this work, is the Voss verification system [20], used extensively for hardware verification. Voss uses a lazy functional language called FL as its interface language. In FL, booleans are implemented using Binary Decision Diagrams (BDDs) [4]. In effect, a decision procedure for propositional logic is built into the language, allowing the user to combine simulation and verification in powerful ways.

In this paper, we introduce two new flavors of booleans for Haskell. The first one follows FL by defining booleans using Binary Decision Diagrams. The improvement over FL is that we're able to do this by a mixture of type classes, the foreign function interface, and a little `unsafePerformIO` magic, rather than by designing and implementing (and maintaining!) a new language. For the second flavor of booleans, we extend the logic to quantifier-free predicate logic by using the Stanford Validity Checker (SVC) [2].

The implementations of each flavor are complex and have a strong imperative feel to them, but for both we have defined referentially transparent interfaces, allowing the underlying tools to do their work while the user simply sees the corresponding values. To some extent, this choice was forced upon us: we found that a fairly tight integration with SVC was necessary in order to avoid overly large intermediate data structures and to exploit the data sharing provided by SVC.

Even though both implement decision procedures for logics, BDDs and SVC are quite different in their approach. BDDs represent propositional formulae maintained in a canonical form. The results of operations are simplified incrementally, so equivalence between propositions is deter-

mined immediately by structural equivalence. In contrast, because it handles a richer logic, the basic SVC operations construct the problem *statement*. Much of the work is contained in testing logical equivalence, which involves a call to the prover. What pleased us about the embedding in Haskell is that both approaches are implemented in the same framework, so the user has great freedom to decide which is appropriate for the task.

The tight connection between the various provers and Haskell allows Haskell to be used very naturally as a metalanguage, in effect enhancing the capabilities of each of the logics. In the BDD case, we have an example where the parametric nature of Haskell's polymorphism can be used to lift the result of the BDD analysis to arbitrary datatypes. In the SVC case, we present an example where we introduce an uninterpreted function, but use the expressiveness of Haskell to generate a limited axiomatization of it.

In summary, the goal of this paper is to describe a new easy-to-use power tool for the Haskell programmer's workbench. Applications include verification of Haskell programs within Haskell. This suggestion immediately brings to mind visions of higher-order logics, but for now we'll forgo generality in favor of the automation of simpler logics.

The rest of the paper is organized as follows. Section 2 presents the Boolean class. In Sections 3 and 4 we describe BDDs, and provide an example leveraging the structure of Haskell. In Section 5 we do the same for SVC and in Section 6 we present a larger worked example using the power of SVC. The remaining sections present discussion.

2 Logical Type Classes

We now do for `Bool` what the `Num` class does for numeric types. That is, we define a type class signature for operations over booleans. It contains all the usual suspects, plus implication (`==>`), mutual implication (`<=>`), and if-then-else (`ifb`).

```
class Boolean b where
    true :: b
    false :: b
    (&&) :: b -> b -> b
    (||) :: b -> b -> b
    (==>) :: b -> b -> b
    (<=>) :: b -> b -> b
    not :: b -> b
    ifb :: b -> b -> b -> b
```

`Bool` is of course an instance of class `Boolean`.

We also need to refer to logical variables, which are not an aspect of the `Bool` datatype. Thus, we introduce a new class for logical variables.

```
class Var a where
    var :: String -> a
```

For example, here's a little proposition about distributing `&&` over implication:

```
(var "a" ==> var "b") && (var "c" ==> var "d")
==>
(var "a" && var "c") ==> (var "b" && var "d")
```

Of course, `Bool` is used in many places in the prelude. One place that it shows up is in the definition of equality. We define a variant of the `Eq` class where the boolean result is abstract.

```
data BinTree a t =
      Terminal t
    | Branch a (BinTree a t) (BinTree a t)
    deriving Eq

cofactor a (Terminal x) =
    (Terminal x, Terminal x)
cofactor a c@(Branch b x y) =
    if a == b then (x, y) else (c, c)

top2 x y =
    a `min` b
    where
        a = index x
        b = index y
        index (Terminal _) = maxBound
        index (Branch a _ _) = a

norm b@(Branch a x y) = if x == y then x else b
norm x = x

bddBranch a x y =
    let a' = top2 x y
        (x1, x2) = cofactor a x
        (y1, y2) = cofactor a y
    in
        if a <= a' then
            norm (Branch a x1 y2)
        else
            norm (Branch a' (bddBranch a x1 y1)
                            (bddBranch a x2 y2))
```

Figure 1: Bdd normalization

```
class Boolean b => Eql a b where
    (===) :: a -> a -> b
```

3 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a representation of boolean functions as a binary tree. The nodes are labeled by boolean variables, and the leaves, usually referred to as *terminals*, are boolean values. A BDD represents a boolean formula in the form of a case analysis.

An Ordered BDD (OBDD) is a normal form for BDDs defined as follows:

- the variables along every path are in strictly increasing order, and
- all unnecessary nodes are eliminated. An unnecessary node represents a variable upon which the subtree doesn't depend, and is recognized as a node where both left and right children are equal.

An Ordered BDD is further a canonical form for boolean formulae, thus equality of two OBDDs is reduced to structural equality.

Construction of OBDDs can be easily described in Haskell, as we show in figure 1. Creating a new terminal is easy—just use the `Terminal` constructor. Creating a new branch (`bddBranch`) is a matter of pushing the branch down the tree until it's label is in order. Notice that as a node

is moved down the tree, its branches get duplicated. This duplication may be countered by the pruning action of the second restriction (implemented by `norm`). But in general, the worst case size complexity of a BDD is exponential in the number of variables.

An important refinement that makes BDDs a practical tool for large scale verification is Reduced Ordered BDDs (ROBDDs)[1] [4]. An ROBDD is one in which the tree is reduced to a directed acyclic graph (DAG) by performing common subexpression elimination to capture all structure sharing. This has two benefits. The first is that equivalence of BDDs is reduced to pointer equality. The second is a reduction in space use, which can be considerable when there's a significant amount of regularity in the data. Despite the worst-case size complexity of BDDs, the worst case is often avoided in practice.

However, there's another aspect of sharing that isn't captured by just using a DAG. The regularity that leads to a lot of structural sharing also leads to performing the same boolean calculations on a given BDD over and over again—you may do a lot of work just to realize that you've already constructed this tree before. Thus, another important trick in a BDD implementor's bag is to keep a memo table for each basic boolean operation so that the operation is only performed once on any given BDD (or pair of BDDs).

It's an interesting question whether a Haskell implementation of BDDs would offer any improvement over a highly-tuned off-the-shelf BDD library written in C. We've pursued this question by doing prototypes that use the latest features of Hugs and GHC to implement structure sharing and memoizing. The results are promising, but it's clear that Haskell isn't really ready to beat C at its best game. So maybe we're trying the wrong strategy. All of these structure-sharing and memoizing mechanisms make BDDs strict, whereas the simple implementation of OBDDs sketched above is lazy. An even more interesting question may be whether there's some way to play off of Haskell's strengths and take advantage of laziness. This question is pursued further in Section 7.1.

3.1 The CMU BDD Library

The BDD library used in this paper is David Long's BDD library (which we'll refer to as the CMU library) [16], which is an ROBDD package written in C, and is one of several high-quality ROBDD packages that are available. We use the CMU BDD package as distributed with the VIS suite, version 1.3, from Berkeley [1]. As with other BDD packages, it comes with its own garbage collector that has been tuned to work well with BDDs.

We import the CMU BDD package into Haskell with the help of H/Direct [10]. Using the foreign function interface of either GHC or Hugs, you can interface to libraries written in various imperative languages, such as C, FORTRAN, Java, etc. H/Direct helps simplify that process by automatically generating the glue code that marshals and unmarshals datatypes from C/FORTRAN/Java to Haskell. The input to H/Direct is a specification of the external library's interface, written in IDL, an Interface Description Language used in industry. IDL interface descriptions are essentially annotated C declarations. In the case of the BDD library, it was a fairly easy matter to translate the header file from the library into an IDL description. For example, here's the snippet of IDL describing the BDD implementations for the boolean constant `true`, and boolean conjunction:

```
cmuBdd cmu_bdd_one ([in]cmuBddManager bddm);
cmuBdd cmu_bdd_and ([in]cmuBddManager bddm,
                    [in]cmuBdd x, [in]cmuBdd y);
```

The `[in]` annotations indicate that the argument is an input argument only (i.e. it doesn't return a result by side-effect).

The first parameter to both functions is common to all BDD calls and is the structure that holds all the context necessary for managing BDDs, such as hash tables, and garbage collection data.

The signatures that H/Direct generates for these two functions are as follows:

```
cmu_bdd_one :: CmuBddManager -> IO (CmuBdd)
cmu_bdd_and :: CmuBddManager ->
               CmuBdd -> CmuBdd -> IO (CmuBdd)
```

3.2 Managing BDDs

Of course, the raw imported interface to the BDD package is not exactly the svelte duck that we were after in the beginning. There are two things in the way. First, the `CmuBddManager` is an implementation artifact that we would rather hide from users. Second, the result lies in the `IO` monad. Our plan is to hide the fact that we are using an imperative implementation underneath by liberal use of `unsafePerformIO`. We will then to justify why it's really safe after all.

In the C world, the types `CmuBddManager` and `CmuBdd` are pointers to rich structures. In Haskell, all of that is hidden, and they appear as type synonyms for the type `Addr`, a type which only supports pointer addition and pointer equality. However, even this is more than we want to expose about BDDs, so we define a new abstract type for BDDs.

```
newtype BDD = Bdd CmuBdd
```

The BDD manager must ordinarily be explicitly allocated at the beginning of a program. We would rather it was allocated on demand without any muss or fuss from the programmer. We use `unsafePerformIO` to get this effect:

```
bdd_manager :: CmuBddManager
bdd_manager = unsafePerformIO (cmu_bdd_init)
```

In the instance declarations for `Boolean`, we distribute the `bdd_manager` to all the calls, and the extract the result from the `IO` monad with `unsafePerformIO`.

```
instance Boolean (Bdd Bool) where
  true = Bdd $ unsafePerformIO $
         bdd_one bdd_manager
  (Bdd x) && (Bdd y) =
      Bdd $ unsafePerformIO $
      cmu_bdd_and bdd_manager x y
  ...
```

The `$` operator is right-associated function application (`f $ g $ x = f (g x)`).

Dealing with variables brings up another detail of the machinery that we wish to hide. The BDD package uses integers as variable identifiers, but we'd rather provide the nicer interface of using strings as variable identifiers. Thus, we keep a table that assigns to each variable name encountered so far a unique integer.

[1] Subsequently, we will use the term BDD to mean ROBDD.

```
bdd_vars :: IORef [(String, Int32)]
bdd_vars = unsafePerformIO (newIORef [])
```

The instance of BDDs for the `Var` class is then defined in terms of a function that keeps track of the necessary bookkeeping. The function `newvar` takes a reference to the lookup table, and a variable identifier, and constructs a BDD representing that variable.

```
newvar :: CmuBddManager ->
          IORef [(String,Int32)] -> String ->
          IO CmuBdd
newvar m v a =
    do  vars <- readIORef v
        case lookup a vars of
          Just i -> bdd_var_with_id m i
          Nothing ->
            do  b <- bdd_new_var_last m
                i <- bdd_if_id m b
                writeIORef v ((a, i) : vars)
                return b

instance Var (Bdd Bool) where
    var a = Bdd $ unsafePerformIO $
            newvar bdd_manager bdd_vars a
```

3.3 Safe use of `unsafePerformIO`

All this use of `unsafePerformIO` may seem a bit blithe, so it's worth a few moments to informally justify why it doesn't harm referential transparency.

For starters, since the types `CmuBddManager` and `CmuBdd` are essentially abstract, we're in control of what can be observed about the imperative side. We simply do not import operations from the CMU package that reveal the imperative structure, such as a procedure that gives the size of internal hash tables. Those functions that we do import, despite all the operational goings-on with hash tables and heap allocation, are essentially functional. The imperative features affect the way the data is constructed, but not the data itself, and we give ourselves no way to observe the imperative details of how the data is constructed.

We do allow ourselves to do pointer equality on `CmuBdds`, which is how structural equality is tested. But since the pointers will be equal if-and-only-if the structures are equal, this is safe.

One point to question is the use of `unsafePerformIO` in the definition of `bdd_vars` to extract an `IORef` out of the `IO` monad. By doing this and then using the `IORef` elsewhere inside the `IO` monad, we're making the assumption that the store inside the `IO` monad is indeed persistent, and that `IORefs` are coherent between invocations of the `IO` monad. Fortunately, although this behavior is not, to the authors' knowledge, documented, it is at least the most reasonable assumption to make. After all, the `IO` monad is supposed to represent the "real world", which has the behavior of being persistent.

4 An Example: Sorting

BDDs give us the ability to show equivalence of boolean functions. This is useful of itself, but in this section we show how the structure of Haskell can be used to imply much richer results. The example we take is from sorting.

4.1 Comparison-Swap Sorting

Knuth has a famous theorem about sort algorithms that are based only on comparison-swaps (an operation that takes two elements and returns them in sorted order). The theorem states that if such an algorithm is able to sort booleans correctly, it will also sort integers correctly. Knuth's proof is based in decision-theory, and is specific to sorting. We have discovered that Knuth's result is a special case of the more general theorems coming from polymorphic parametricity.

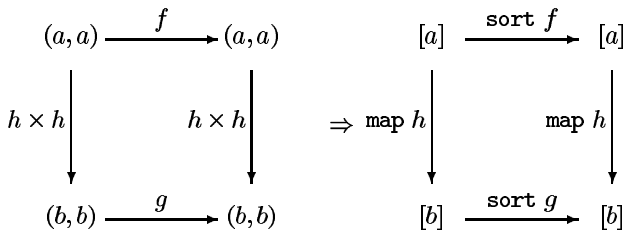
This suggests the following proof technique. Take a polymorphic function, perform some verification using BDDs at the boolean instance, and then use the parametricity theorem to deduce a corresponding result for more complex types. In effect, the boolean instance acts as an abstract interpretation of the more general algorithm, and parametricity supplies the abstraction and concretization relationships.

To see how this works in practice, we first need to consider the type of a comparison-swap sort. We want to avoid the manual examination of the program text that an application of Knuth's theorem would require. What can we learn from the type alone? The type cannot tell us that the algorithm is a sorting algorithm, but it can ensure that it makes no assumptions about the contents of the list, except via the first parameter. Consider the type:

```
sort :: ((a,a)->(a,a)) -> [a] -> [a]
```

If the sort argument is indeed a comparison-swap function then, intuitively, the type of `sort` ensures that the only data-sensitive operation `sort` can use is comparison-swap.

Let's make this precise. Consider the parametricity theorem for functions of this type [19, 23] (f, g, and h are universally quantified, and we define $j \times k = \lambda x, y.(j\ x, k\ y)$):



In Haskell, the theorem applies only when h is strict and, since the introduction of `seq` in Haskell 98, bottom-reflecting as well.

Now, instantiate a to be `Int`, and b to be `Bool`, and chose f and g to be the standard comparison/swap over integers and booleans (where `False` < `True`) respectively. If we can show that `sort` g sorts sequences of booleans correctly (using BDDs for example), then the parametricity theorem will allow us to conclude that `sort` f sorts sequences of integers correctly as well.

To see this, suppose the converse, and we will derive a contradiction. Suppose xs contains x and y, such that $x < y$. For `sort` f to be incorrect, there has to exist at least one x and y pair which appears out of order (y before x) in the list `sort` f xs. Let h be the function that is false for all inputs less than y, and true otherwise ($h(n) = y <= n$). This function commutes with f and g, and is strict and bottom-reflecting as well, thus it satisfies the precondition for the theorem. Therefore, the right-hand side of the theorem must hold.

Now, by assumption, `sort` g (`map` h xs) is sorted correctly. That is, the result is a list of booleans with all the occurrences of `False` preceding the occurrences of `True`. However, if y precedes x in the result of `sort` f xs then the result of `map` h(`sort` f xs) will contain an occurrence of `True` before the final occurrence of `False`. Thus, we have a contradiction, so the assumption that y preceded x in the result of `sort` f xs was incorrect.

In effect, parametricity has ensured that `sort` behaves coherently over all types, so that results at the boolean instance can be used to imply consequences at other types. Another perspective is that parametricity expresses the multi-faceted symmetry inherent in this problem Symmetry is vital in verification by model checking for reducing large problem spaces to manageable proportions, and that is what is achieved here [8]. Rather than model check on lists of 32-bit integers, we perform the check on single-bit integers.

4.2 Checking Comparison-Swap Sort on Booleans

The missing part of the story is using BDDs to show that `sort` g sorts lists of booleans correctly. We can only show that sorting is correct for an arbitrary, but fixed-length list. The logic of BDDs is simply not powerful enough to prove the result in general (which would require some kind of inductive argument—well beyond the scope of propositional logic). However, since the method is automated, it's no trouble to check it for a variety of lengths of list, leaving only very subtle bugs out of its reach.

The sorting algorithm we use is bitonic sort, an efficient algorithm that is particularly amenable to hardware realization. Also, for a sorting algorithm, it's fairly tricky, so it's a good candidate for verification. The code is given in Figure 2. Bitonic sort is designed to work on lists that have a length that is a power of two. It recursively divides the list into two parts and sorts each partition. To combine the two parts it swaps corresponding elements in each list. Because one list is sorted in ascending order and the other in descending order, the swapping results in all the elements in the first list being lower (or higher) than the elements in the second list. The two lists are in the correct form to repeat this swapping on the two sublists to arrive at a sorted list.

To verify the algorithm, we first need a simple predicate to indicate whether a list is sorted or not.

```
sorted test [] = true
sorted test [x] = true
sorted test (x : ys@(y : _)) =
    x `test` y && sorted test ys
```

Now, we will state the property that we want to show for a list with variable elements, but a fixed length of sixteen.

```
result = sorted lessEq (sort xs)
  where
    xs = [ var ("x" ++ show i) | i <- [0 .. 15] ]
    sort xs = bitonic_sort cmpSwap xs True
```

It remains to define the two BDD-specific functions `cmpSwap` and `lessEq`. These turn out to be particularly nice.

```
cmpSwap a b = (a && b, a || b)

lessEq a b = a ==> b
```

Now, when we query Haskell about `result`, it returns `true`.

```
bitonic_to_sorted cmpSwap [] up = []
bitonic_to_sorted cmpSwap [x] up = [x]
bitonic_to_sorted cmpSwap xs up =
  let k = length xs `div` 2
    (ys, zs) = pairwise cmpSwap (splitAt k xs)
    (ys', zs') =
      if up then (ys, zs) else (zs, ys)
  in
    bitonic_to_sorted cmpSwap ys' up ++
    bitonic_to_sorted cmpSwap zs' up

pairwise f ([], []) = ([], [])
pairwise f (x : xs, y : ys) =
  let (x', y') = f x y
    (xs', ys') = pairwise f (xs, ys)
  in
    (x' : xs', y' : ys')

bitonic_sort cmpSwap [] up = []
bitonic_sort cmpSwap [x] up = [x]
bitonic_sort cmpSwap xs up =
  let k = length xs `div` 2
    (ys, zs) = splitAt k xs
    ys' = bitonic_sort cmpSwap ys True
    zs' = bitonic_sort cmpSwap zs False
  in
    bitonic_to_sorted cmpSwap (ys' ++ zs') up

cmpSwap x y = if x < y then (x, y) else (y, x)
```

Figure 2: Bitonic Sort

4.3 Limitations to using Parametricity

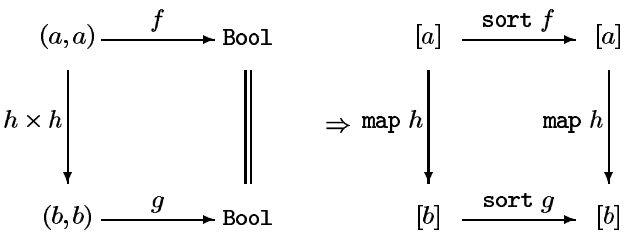
We expect the verification technique outlined above to be useful in many cases, but it's not a panacea. Sometimes parametricity is not powerful enough to capture appropriate abstractions. In effect, some types are simply not expressive and/or constraining enough to enable the boolean instance to say much about the general case. Consider the following variation on the example above.

It might seem that the parametricity argument that we used to echo Knuth's sorting theorem would apply just as easily to a regular sort algorithm based on a comparison function, with type:

```
sort :: ((a, a) -> Bool) -> [a] -> [a]
```

However, it is fairly easy to construct a pseudo-sorting algorithm of this type that will correctly sort lists of booleans but fails to sort lists of integers correctly. Consider the following: take the first element of the list as a partition value. Next, do a one-pass sort into three buckets: one for elements less then the partition, one for those equal (neither less, nor greater), and one for those greater. Finally, stick the partition element in the equal bucket, and concatenate the buckets in the order: less, equal and greater. Partitioning based upon a single element will work for booleans, because there's only two values; however, it clearly won't work in general.

So, the parametricity-based approach must fail for comparison-based sorts. Where does it break down? First, examine the "free theorem" for a comparison-based sort.



The conclusion is identical in each instance of the parametricity theorem, but the precondition of this instance is much more stringent than before. The key to proving the comparison-swap case was the ample supply of appropriate functions h to "detect" any incorrectly sorted list. However, the precondition on h in this case requires that the comparisons on the two sorts, integer and boolean say, are equivalent to one another. Thus for the case of comparison sort there are essentially no interesting choices for h relating the integer and boolean cases.

5 The Stanford Validity Checker

The Stanford Validity Checker (SVC) is an implementation of a decision procedure for a quantifier-free, first-order logic with equality [2, 7, 11]. It has been used extensively for microprocessor validation and verification [7, 11, 12, 22] and recently for requirements validation [18]. The logic allows models to include uninterpreted functions, which can be used to represent datapath operations in a pipelined architecture. SVC returns a counterexample if the formula is not valid.

formula	::=	ite (*formula*, *formula*, *formula*)
	|	(*term* = *term*)
	|	*predicate symbol* (*term*, ..., *term*)
	|	true
	|	false
term	::=	ite (*formula*, *term*, *term*)
	|	*function symbol* (*term*, ..., *term*)
	|	read (*term*, *term*)
	|	write (*term*, *term*, *term*)
	|	*distinct constant*
	|	*formula*

Figure 3: The SVC logic

5.1 Connecting SVC with Haskell

Our initial interface with this tool was file-based. We had a representation of expressions in the logic as a datatype in Haskell and wrote expressions of this form to a file that was later read by SVC. As we worked on larger examples, this approach became unmanageable. The size of the structure was extremely large and did not take advantage of possible sharing of subexpressions. While SVC's internal data structure is not canonical as is the case for BDDs, it is optimized and shares common subexpressions. Thus it quickly became apparent that a much better approach is to have a tight link between the process of generating the term and building the term in SVC. Using H/Direct we were able to create an abstract interface to the SVC C++ functions that build the expressions. We used version 1.1 of SVC.

As it is a richer logic, SVC expressions include more than just boolean-valued terms. Figure 3 contains a description of the SVC logic. The predicate and function symbols introduce uninterpreted predicates and functions. The functions `ite` and `=` are interpreted functions representing "if-then-else" and equality. The functions `read` and `write` are interpreted as acting on stores; an axiom of the logic relating these functions is, `read (write (store, index, data), index) = data`. Other logical, numeric, bit vector and record operations also have an interpreted meaning.

Using H/Direct we created an interface to SVC that has functions for building each of the kinds of terms and formulae. These functions return elements of the type `PExpr`, which are pointers to SVC expressions. The interface functions to SVC that build expressions in the logic do not distinguish between terms and formulae.

As with the BDD package, the calls to the SVC functions are wrapped in `unsafePerformIO` to extract the value from the IO monad. Because the only way to observe the SVC expressions is to check their validity, the meaning of an expression is the same regardless of its order of construction. Therefore we can use the term building functions as if they are referentially transparent.

Only a subset of the SVC expressions, the formulae, can be used to instantiate the Boolean class. Even though the underlying package doesn't distinguish between terms and formulae, we want Haskell to make this distinction so that the Boolean class is only instantiated for formulae. We create the datatypes `SvcFormula` and `SvcTerm` to wrap around the pointers to expressions that SVC returns to make them distinct types.

```
newtype SvcFormula = SvcF PExpr
```

```
newtype SvcTerm = SvcT PExpr
```

We wrap the output of the SVC functions with `SvcF` or `SvcT` as appropriate. The arguments to the function must be unwrapped.

Using Haskell's type system to distinguish between terms and formulae in SVC's logic, we instantiated the Boolean class using only the formulae of SVC.

```
instance Boolean SvcFormula where
  true = SvcF $ unsafePerformIO $ Svc.makeTrue
  (SvcF a) && (SvcF b) =
    SvcF $ unsafePerformIO $ Svc.makeAnd a b
  ...
```

The functions `Svc.makeTrue` and `Svc.makeAnd` are calls to the SVC package.

In SVC the equality operator is also used to create formulae. This operator is an instance of the generalized equality class:

```
instance Eql SvcTerm SvcFormula where
  (SvcT a) === (SvcT b) =
    SvcF $ unsafePerformIO $ Svc.makeEquals a b
```

Because SVC has both terms and formulae, there are functions that create terms. We provide wrappers for these functions as well. For example, `fcn` creates an uninterpreted function application, where the first string argument is the name of the function, and the arguments to the function are provided in a list:

```
fcn a bs =
  SvcT $ unsafePerformIO $
    (if (bs==[]) then Svc.makeSymbol a
                 else Svc.makeUninterpretedFcn a
                        (args bs))
```

The function `args` turns the Haskell list of terms into the SVC form.

The SVC package has two instantiations of the `Var` class – one for formulae, and one for terms.

```
instance Var SvcFormula where
  var a = SvcF $ unsafePerformIO $ Svc.makeSymbol a

instance Var SvcTerm where
  var a = SvcT $ unsafePerformIO $ Svc.makeSymbol a
```

Type annotations are sometimes necessary to distinguish which instance of `var` is being used in a Haskell program.

The interface includes the SVC function `checkValid` to call the prover on the constructed expression. Calls to `checkValid` are referentially transparent because our interface tells SVC to treat each check independently from any other calls to the prover. We pop its stack of knowledge about a particular proof session (context), but retain its data about the expressions that have been built.

5.2 Sort Example

SVC is able to check the sort algorithm presented in Section 4 for a fixed length list of elements without the parametricity meta reasoning because SVC can reason over arbitrary types. To do this, we provided different definitions for `lessEq` and `cmpSwap`. We made the "less than" operator an uninterpreted predicate replacing its use with `pred "lt" [a,b]`. We also used the SVC "if-then-else", namely `itet`.

```
INVALID
Falsifying Assumptions
======================
Assert:

  $92:(lt $7:a1 $11:a3)
Deny:

  $85:(lt $8:a2 $7:a1)
Deny:

  $55:(lt $8:a2 $11:a3)
Deny:

  $57:(lt $7:a1 $12:a4)
Deny:

  $13:(lt $11:a3 $12:a4)
Deny:

  $9:(lt $7:a1 $8:a2)

INVALID
Case_Splits:    7
Exprs_Generated:        57
```

Figure 4: SVC counterexample

```
cmpSwap x y =
  let test = pred "lt" [x,y] in
  (itet test x y, itet test y x)

lessEq x y = not (pred "lt" [y,x])
```

The prover was invoked to determine if a fixed length list of symbolic elements is sorted, as in:

```
result = checkValid
           (sorted lessEq (sort xs)) where
  sort xs = bitonic_sort cmpSwap xs True
  xs = [var "a1", var "a2", var "a3", var "a4"]
```

SVC returned with a counterexample found in Figure 4. The counterexample is in the form of a series of assertions and denials of subformulae. The "$" variables refer to internal subexpression names. The case provided in Figure 4 has both $\neg(a2 < a1)$ and $\neg(a1 < a2)$, which means $a2$ must equal $a1$. The case also says that $a1 < a3$ and $\neg(a2 < a3)$, which is impossible when $a1$ and $a2$ are equal, and $<$ has its intended meaning. From this counterexample, we learned that we cannot achieve our verification result without providing more information about the behavior of the "less than" operator.

SVC has an interpreted "less than" function for rational expressions that we could use. But we wished to check the sort algorithm for all types of ordered elements without any meta reasoning. SVC needed the information that the "less than" operator is irreflexive, transitive, and that $x \neq y \Rightarrow (x < y = \neg(y < x))$. We provided these in the form of antecedents to the consequent that we wanted to check. This is a limited axiomatization of the "less than" operator.

The SVC logic has no quantifiers so it was necessary to generate all the possible instantiations of these properties

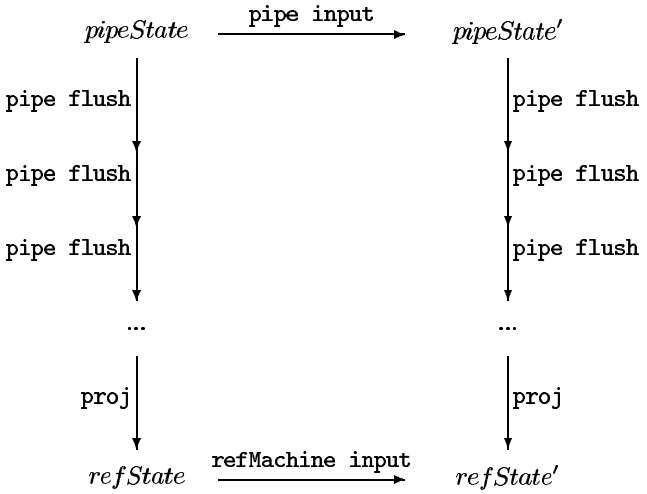


Figure 5: Burch and Dill Commuting Diagram (found in [7])

for the symbolic elements in the input list. Haskell's list comprehension syntax was very convenient for stating, in a compact form, all the antecedents that were needed. For example, transitivity of a relation `r` for a list of elements is expressed as:

```
trans r x y z = (r x y && r y z) ==> r x z

genTrans list r =
  foldr1 (&&) [trans r x y z |
                 x <- list,
                 y <- list,
                 z <- list]
```

With these antecedents, SVC returns instantly for the bitonic sort of 4 elements saying that the sort algorithm is correct.

6 Example 2: Microprocessor Verification

As a second example of the use of SVC in Haskell, we present the verification of a simple pipelined ALU used in Burch and Dill [7] (originally found in Burch et al. [6]). In their presentation, they use a simple hardware description language based on Lisp as input to their verification process. This section describes how this example can be verified in Haskell using SVC and uninterpreted functions for the datapath operations.

The Burch and Dill approach to verification automatically calculates an abstraction function relating a microprocessor pipeline to a reference machine. The calculation is done using symbolic simulation. The pipeline is equivalent to the reference machine if the diagram in Figure 5 commutes. The abstraction function consists of flushing the intermediate results of the pipeline and projecting from the pipeline only the parts of the state visible in the reference machine (`proj`).

Figure 6 is a pipeline and reference machine modeled in Haskell, translated from the descriptions in Lisp found in the appendix of Burch and Dill's paper. The state of the reference machine is simply the register file. The reference machine and the pipeline are compared on the value of the register file only. The register file element of the pipeline is projected from its state as part of the correctness statement.

Because we leave the ALU operation as an uninterpreted function, our verification using SVC does not depend on the datapath width and operations. For the operations on the register file, the interpreted write and read SVC functions are used. The SVC instantiations of the Boolean operators and equality (`===`) are chosen automatically.

To verify the pipeline, we stall it to flush its state by setting the stall signal of the input high for a certain number of steps. The flush input contains symbolic values for every input other than the stall signal.

```
flush = (true,
         var "flushDestReg",
         var "flushOpcode",
         var "flushSrc1Reg",
         var "flushSrc2Reg")

flushPipe initialState n =
  if (n == 0) then initialState
  else flushPipe (pipe flush initialState) (n-1)
```

The `initialState` also assigns symbolic values to all the internal latches of the pipe:

```
initialState = (var "registers",
                var "arg1",
                var "arg2",
                var "bubble_wb",
                var "dest_wb",
                var "result",
                var "bubble_ex",
                var "dest_ex",
                var "op_ex")
```

To calculate the left and bottom route of the commuting diagram, we flush the pipeline, project out a state for the reference machine, and run the reference machine on this initial state with symbolic input:

```
proj (registers, _,_,_,_,_,_,_,_) = registers

input = (var "stall",
         var "dest",
         var "opcode",
         var "src1",
         var "src2")

path1 n = refMachine input
            (proj (flushPipe initialState n))
```

We compare `path1` with the other side of the commuting diagram. In `path2`, we run the pipeline on the symbolic input, starting from the symbolic initial state, and then flush the pipeline:

```
path2 n = proj
        (flushPipe (pipe input initialState ) n)
```

These two paths are computed by executing the Haskell models. On any path in the pipe there are at most two latches, therefore the pipe should agree with the reference machine after two flushes. The verification condition that we pass to the prover to be checked is:

```
pipeTest = (path1 2) === (path2 2)
```

```
type Input =
    (SvcFormula,  -- stall
     SvcTerm,     -- dest
     SvcTerm,     -- opcode
     SvcTerm,     -- source1
     SvcTerm)     -- source2

type PipeState  =
    (SvcTerm,     -- register file
     SvcTerm,     -- arg1
     SvcTerm,     -- arg2
     SvcFormula,  -- bubble-writeback
     SvcTerm,     -- dest-writeback
     SvcTerm,     -- result
     SvcFormula,  -- bubble-ex,
     SvcTerm,     -- dest-ex,
     SvcTerm)     -- opcode

pipe :: Input -> PipeState -> PipeState
pipe (stall, dest,opcode,src1,src2)
  (registers,arg1,arg2,bubble_wb,dest_wb,
        result,bubble_ex,dest_ex,op_ex) =
  (registers',arg1',arg2',bubble_wb',dest_wb',
        result', bubble_ex',dest_ex',op_ex')
    where
    registers' = itet bubble_wb
          registers
          (write registers dest_wb result)
    bubble_wb' = bubble_ex
    dest_wb' = dest_ex
    result' = fcn "alu" [op_ex, arg1, arg2]
    bubble_ex' = stall
    dest_ex' = dest
    op_ex' = opcode
    arg1' = itet ((not bubble_ex) &&
                  (dest_ex === src1))
            result'
            (read registers' src1)
    arg2' = itet ((not bubble_ex) &&
                  (dest_ex === src2))
            result'
            (read registers' src2)

type RefState = SvcTerm   -- register file

refMachine :: Input -> RefState -> RefState
refMachine (stall, dest, opcode, src1, src2)
          registers =
    itet stall
    registers
    (write registers dest
              (fcn "alu" [opcode,
                          read registers src1,
                          read registers src2]))
```

Figure 6: Microprocessor models

SVC verifies `pipeTest` instantly.

Integrating SVC with Haskell creates a very convenient debugging loop when there are errors in the model. Using the information in a counterexample, concrete values can be input to the model to illustrate the error.

7 Discussion

The section discusses some interesting points that have been raised in creating these logical abstractions.

7.1 Shallow versus Deep Embedding

In order to look most like a duck, we've taken the approach of doing a shallow embedding of both BDDs and SVC. This means we directly interpret the logical operators as operations on the internal data structures of the BDD package and SVC. An alternate approach is a deep embedding, where we construct an intermediate data structure that is exactly (or close to) the term structure.

One benefit of the deep embedding is that it gives us the opportunity to tackle the normalization process in different ways that may be more efficient. By analogy, when constructing BDDs incrementally, we must use essentially a bubble sort to put the nodes in sorted order. The incremental approach is necessarily based on local decisions, but we know that sorting is suboptimal when it is restricted to making local decisions. Thus, we can imagine being able to do something analogous to mergesort to put a BDD in normal form much more efficiently.

This approach is not considered feasible in the strict setting of a C implementation, because the intermediate data structure would be huge, and space is more of a limiting factor with BDDs than speed. The intermediate data structure would not be able to take advantage of any sharing. Thus, the only feasible approach is to calculate the sharing as you go.

However, in the setting of a lazy functional programming language, we have more options. Because the intermediate data structure doesn't necessarily get built, we may be able to take advantage of laziness to process BDDs more efficiently, while not taking a hit in space usage.

Another argument in favor of a deep embedding is that we could let Haskell control decomposition or simplification before calling the decision procedure. Haskell could become a platform for building "minimal proof assistants" [17] combining evaluation for term generation, decision procedures, and theorem proving techniques.

7.2 Types

So far, we are not making too much use of Haskell's rich type system. The only type distinction that we make in SVC expressions is between booleans and any other kind of term. We are working on building a typed layer on top of SVC logical terms where we regain the typechecking benefits of Haskell. This layer will make extensive use of type classes letting Haskell do the work of choosing the correct instances of functions rather than the user.

7.3 Ambiguity

One unfortunate consequence of generalizing booleans to a type class is that ambiguity problems can arise left and right. Booleans are used all over the place as intermediate values,

especially in `if-then-else` expressions. Intermediate types in expressions don't show up in the type of the overall expression, and thus the type class system has no basis upon which to chose which instance to use. The same scenario holds for the `Num` class, but Haskell resolves this by the default mechanism. It would be helpful if the default mechanism could be made more general, such that we could talk about defaults for `Boolean` as well.

7.4 BDD Variable Order

As was pointed out in the introduction, in our zeal to put a pretty face on complex implementation packages, we give up a good deal of control.

For the sake of simplicity the interface that we provide to the BDD package leaves the user unaware of the details of variable order when building a BDD. The variables are ordered by the time of their creation. Since Haskell is free to change the order of evaluation, the variable order is not even predictable. This can have serious drawbacks, since the size of a BDD can vary greatly depending on the variable order. Figure 7 gives two BDDs for `(a1 && b1) || (a2 && b2) || (a3 && b3)` with different variable orderings. The dashed lines are false branches and the solid lines true branches.

However, in practice, trying to control variable ordering is a bit of a black art, and the problem in undecidable in general. But this situation is analogous to space allocation in Haskell, which is similarly out of the programmers hand, and has similar bad worst-case scenarios. One option, when variable order really needs to be controlled, is to use explicit sequencing via `seq`.

7.5 Applications in Verification

Why would this connection between Haskell and decision procedures be of interest to the verification community ? First, properties can be proven about Haskell programs. Free theorems from the parametricity of Haskell programs that model microprocessors may provide symmetry-like arguments for reducing the size of the state space.

Second, using Haskell allows models to be written in a strongly-typed language. Typechecking has its own benefits for a specification language, and now we are providing a link directly to verification tools for this language.

Third, Haskell works well as a meta-language for generating terms for input to the verification process. Can laziness in Haskell be exploited to avoid full generation of a term while a proof is in progress ? Laziness could be particularly important for defect-finding verification efforts.

8 Related Work

This work extends the brief description of linking BDDs with Haskell found in Launchbury, Lewis, and Cook [14].

Lava [3] is a Haskell-based hardware description language. They provide multiple interpretations of circuit descriptions for simulation, verification, and code generation. For verification, Lava interfaces to the propositional logic checker Prover [21], and two first-order logic theorem provers. The interface is file-based, breaking an expression into component subexpressions. Lava used reinterpretations of monads to create output for the different provers.

Individually decision procedures have been connected to other functional languages. For example SVC has been connected to Lisp. Voss uses BDDs for all boolean manipulations. And BDD packages such as Buddy [15] have been connected to ML and as a decision procedure in the HOL theorem prover [13]. Compared to these approaches, we use a generalized version of the Bool datatype through the class system to allow the packages to be used somewhat interchangeably. Furthermore, using Haskell we are able to provide this link in a pure functional language while preserving referential transparency.

9 Conclusion

It seems that our logical ducks swim quite well as abstract datatypes in Haskell. By generalizing the boolean and equality classes, it is possible to use the different decision procedures somewhat interchangeably. We have defined referentially transparent interfaces, allowing the underlying tools to do their work while the user simply sees the corresponding values. Having a tight connection between Haskell and the decision procedure allowed us to avoid space limitations in building the unreduced expression. The integration with Haskell also allowed us to leverage parametricity arguments in proofs.

10 Acknowledgements

The use of parametricity to redo Knuth's result was discovered in conjunction with John Matthews and Mircea Draghicescu. We thank Clark Barrett of Stanford for help with the Stanford Validity Checker. The authors are supported by Intel, U.S. Air Force Materiel Command (F19628-96-C-0161), NSF (EIA-98005542) and the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] VIS home page. `http://www-cad.eecs.berkeley.edu/~vis/`.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, 1996.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *DAC*, 1990.

[7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–79. Springer-Verlag, 1994.

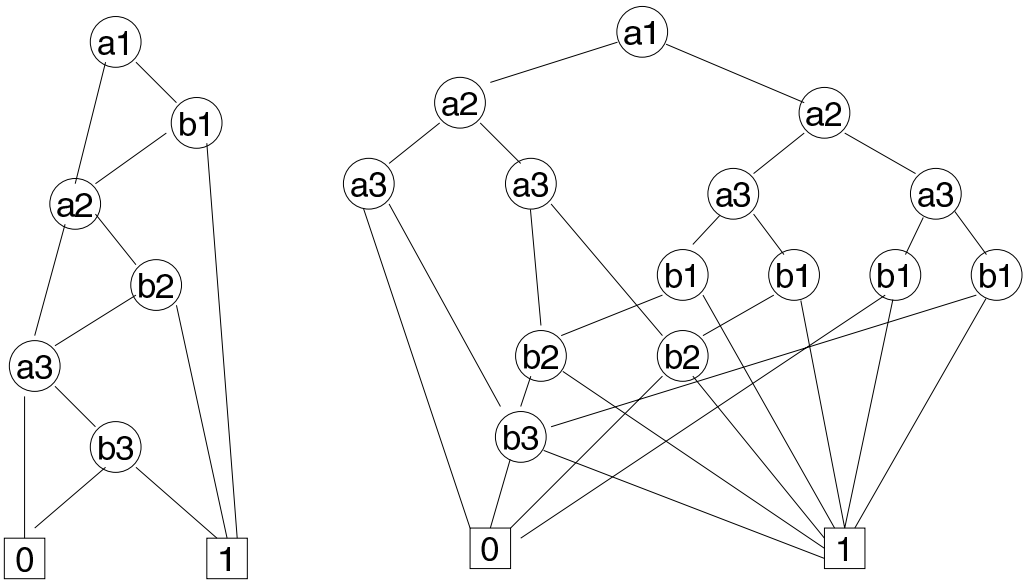


Figure 7: The formula (a1 && b1) || (a2 && b2) || (a3 && b3) with different variable orders (found in [5])

[8] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, pages 450–462, 1993.

[9] C. Elliot and P. Hudak. Functional reactive animation. In *ACM Int. Conf. on Functional Programming*, 1997.

[10] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.

[11] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD*, 1995.

[12] R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 2–17. Springer-Verlag, 1998.

[13] K. Larsen and J. Lichtenberg. MuDDy. http://www.itu.dk/research/muddy/.

[14] J. Launchbury, J. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *ACM Int. Conf. on Functional Programming*, 1999. To appear.

[15] J. Lind-Nielsen. BuDDy: Binary decision diagram package, release 1.6, 1998. Department of Information Technology, Technical University of Denmark.

[16] D. E. Long. bdd - a binary decision diagram (BDD) package. Man page.

[17] K. L. McMillan. Minimalist proof assistants. In *FMCAD*, volume 1522 of *LNCS*, page 1. Springer, 1998.

[18] D. Y. Park, J. U. Skakkebæk, M. P. Heimdahl, B. J. Czerny, , and D. L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98*, 1998.

[19] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In R. Mason, editor, *Information Processing 83*, Proceedings of the IFIP 9th World Computer Conference, 1983.

[20] C.-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, December 1993.

[21] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In *FMCAD*, number 1522 in LNCS, pages 82–99, 1998.

[22] J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV*, volume 1427 of *LNCS*, pages 98–109. Springer-Verlag, 1998.

[23] P. Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

Lightweight Extensible Records for Haskell

Mark Jones (Oregon Graduate Institute, USA) and *Simon Peyton Jones* (Microsoft Research Cambridge, UK)

Lightweight Extensible Records for Haskell

Mark P. Jones
Oregon Graduate Institute
mpj@cse.ogi.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

September 6, 1999

Abstract

Early versions of Haskell provied only a *positional* notation to build and take apart user-defined datatypes. This positional notation is awkward and error-prone when dealing with datatypes that have more than a couple of components, so later versions of Haskell introduced a mechanism for *labeled fields* that allows components to be set and extracted by *name*. While this has been useful in practice, it also has several significant problems; for example, no field name can be used in more than one datatype.

In this paper, we present a concrete proposal for replacing the labeled-field mechanisms of Haskell 98 with a more flexible system of records that avoids the problems mentioned above. With a theoretical foundation in the earlier work of Gaster and Jones, our system offers lightweight, extensible records and a complement of polymorphic operations for manipulating them. On a more concrete level, our proposal is a direct descendent of the Trex implementation ("typed records with extensibility") in Hugs, but freed from the constraints of that setting, where compatibility with Haskell 98 was a major concern.

1 Introduction

Records are one of the basic building blocks for data structures. Like a simple tuple or product, a record gathers together some fixed number of components, each of a potentially different type, into a single unit of data. Unlike a tuple however, the individual components are accessed by *name* rather than *position*. This is particularly important in languages where there is no support for pattern matching, or in cases where the number of different components would make pattern matching unwieldy and error-prone.

Haskell allows programmers to define a wide range of algebraic datatypes, but early versions of the language (up to and including Haskell 1.2) did not provide any support for records. As Haskell became an increasingly attractive platform for general purpose program development, the need for some form of records became more pressing: it is quite common to find data structures with many components in real-world applications, and it is very awkward to deal with such datatypes in a language that supports only the traditional constructor and pattern matching syntax for algebraic datatypes. Motivated by such practical concerns, the mechanisms for defining algebraic datatypes were extended in Haskell 1.3 (and carried over into Haskell 1.4 and Haskell 98) to support *labeled fields*, which allows the components of a datatype to be accessed by name. In essence, the syntax of Haskell was extended so that the definition of a particular algebraic datatype could include not just the names of its constructors, as in earlier versions of the language, but also the names for its selectors. We review the Haskell 98 record system in Section 2.

The Haskell 98 system has the merit of being explicable by translation into a simpler positional notation, and has no impact at all on the type system. However, this simplicity comes at the price of expressiveness, as we discuss in Section 2.1. This paper presents a concrete proposal for an alternative record system, designed to replace the current one (Section 3).

Our proposal is based closely on the extensible records of Gaster and Jones [4]. An implementation of their system has been available for a couple of years, in the form of the "Trex" extension to Hugs, so quite a bit of experience has accumulated of the advantages and shortcomings of the system. Trex is designed to be compatible with Haskell 98, and that in turn leads to some notational clumsiness. This proposal instead takes a fresh look at the whole language.

The resulting design is incompatible with Haskell 98 in minor but pervasive ways; most Haskell 98 programs would require modification. However, we regard this as a price worth paying in exchange for a coherent overall design. We review the differences between our proposal and Trex, Gaster's design, and other system in Section 4.

The paper contains essentially no new technical material; the design issues are mainly notational. So why have we written it? Firstly, there seems to be a consensus that some kind of full-blown record system is desirable, but debate is hampered by the lack of a concrete proposal to serve as a basis for discussion. We hope that this paper may serve that role. Second, it is not easy to see what the *disadvantages* of a new feature might be until a concrete design is presented. Our system does have disadvantages, as we discuss in Section 5. We hope that articulating these problems may spark off some new ideas.

2 Haskell 98 records: datatypes with labeled fields

The Haskell 98 record system simply provides syntactic sugar for what could otherwise be written using ordinary, positional, algebraic data types declarations. For example, to a first approximation, the following definition:

```
data List a = Nil
            | Cons {head :: a, tail :: List a}
```

can be thought of as an abbreviation for separate datatype and selector definitions:

```
data List a = Nil              -- datatype
            | Cons a (List a)

head (Cons x xs) = x           -- selectors
tail (Cons x xs) = xs
```

In fact, Haskell takes things a step further by introducing special syntax for construction and update operations. These allow programmers to build and modify data structures using only the names of components, and without having to worry about the order in which they are stored. For example, we can build a list with just one element in it using either of the constructions `Cons{head=value, tail=Nil}` or `Cons{tail=Nil, head=value}`, and we can truncate any non-empty list `xs` to obtain a list with just one element using the update expression `xs{tail=Nil}`.

The following definition illustrates another convenient feature of Haskell's record notation:

```
data FileSystem
  = File {name :: String, size :: Int, bytes :: [Byte]}
  | Folder {name :: String, contents :: [FileSystem]}
```

Values of this datatype represent a standard hierarchical file system with files contained in potentially nested folders. Notice that `name` is used as a field name in both branches; as a result, the selector function `name` that is generated from this definition can be applied to both `File` and `Folder` objects without requiring the programmer to treat the two alternatives differently each time the name of a `FileSystem` value is required.

The result of these extensions is a record-like facility, layered on top of Haskell's existing mechanisms for defining algebraic datatypes. From a theoretical perspective, of course, these features are just a form of syntactic sugar, and do nothing to make the language any more expressive. In practice, however, they have significant advantages, greatly simplifying the task of programming with data structures that have many components. Moreover, the resulting programs are also more robust because the code for selectors, update, and construction is generated directly by the compiler, automatically taking account of any changes to the datatype when fields or constructors are added, deleted or reordered.

2.1 Shortcomings of Haskell 98 records

Unfortunately, there are also some problems with the Haskell 98 approach:

- Record types are not lightweight; record values can only be used once a suitable algebraic data type has been defined. By contrast, the standard Haskell environment automatically provides lightweight tuple types—records without labels—of all sizes.
- Field names have top-level scope, and cannot be used in more than one datatype. This is a serious problem because it prevents the direct use of a datatype with labeled fields in any context where the field names are already in scope, possibly as field names in a different datatype. The only ways to work around such conflicts are to rely on tedious and error-prone renaming of field names, or by using (module) qualified names.
- Within a single datatype definition, we can only use any given field name in multiple constructors if the same types appear in each case. In the definition of the `FileSystem` datatype above, for example, it was necessary to to use different names to distinguish between the contents (i.e., `bytes`) of a `File`, and the `contents` of a `Folder`.
- There is no way to add or remove fields from a data structure once it has been constructed; records are not extensible. Each record type stands by itself, unrelated to any other record type; functions over records are not polymorphic over extensions of that record.

In the remaining sections of this paper, we present a concrete proposal for replacing the labeled field mechanisms of Haskell with a more flexible system of records that avoids the problems described above. There seems little point in retaining the current labeled field mechanisms of Haskell in the presence of our proposed extensions, as this would unnecessarily complicate the language, and duplicate functionality. Firm theoretical foundations for our system are provided by the earlier work of Gaster and Jones [5], with a type system based on the theory of qualified types [8]. This integrates smoothly with the rest of the Haskell type system, and supports lightweight, extensible records with a complement of polymorphic operations for manipulating them. On a more concrete level, our proposal is inspired by practical experience with the Trex implementation ("typed records with extensibility") in current versions of the Hugs interpreter, but freed from the constraints of that setting, where compatibility with Haskell 98 was a major concern.

3 The proposed design

This section provides an informal overview of our proposal for adding a more flexible system of extensible records to Haskell. It covers all of the key features, and sketches out our proposed syntax. Some aspects of our proposal are illustrated using extracts from a session with an interpreter for a Haskell dialect that supports our extensions. The interpreter prompts for an input expression using a single ? character, and then displays the result of evaluating that expression on the next line. At the time of writing, we have not actually built such an interpreter. However, based on our experience implementing and using the Trex system in Hugs, we are confident that our proposals are feasible, practical, and useful.

We begin by describing the basic syntax for constructing and selecting from record values (Section 3.1), and for representing record types (Section 3.3). But for a few (mostly minor) differences in syntax, these aspects our proposal are almost indistinguishable from the lightweight records of Standard ML (SML). Turning to issues that are specific to Haskell, we show that record types can be included as instances of the standard classes for equality, `Eq`, and display, `Show` (Section 3.4). One key feature of our proposal, which clearly distinguishes it from the existing mechanisms in languages like SML and Haskell 98, is the support for *extensibility* (Section 3.5). By allowing record extension to be used in pattern matching, we also provide a simple way to select or remove specific record components. A combination of extension and removal can be used to update or rename individual field in a record. In practice, we believe that these operations are useful enough to warrant a special syntax (Section 3.6). A second key feature of the underlying type system is *row polymorphism* (Section 3.7), and this leads us to introduce a general syntax for rows (Section 3.8). Finally, we turn to a collection of additional features that are less essential, but that look very attractive (Section 3.10).

3.1 Construction and Selection

In essence, records are just collections of values, each of which is associated with a particular label. For example:

```
{a = True, b = "Hello", c = 12::Int}
```

is a record with three components: an `a` field, containing a boolean value, a `b` field containing a string, and a `c` field containing the number 12. The order in which the fields are listed is not significant, so the same record value could also be written as:

```
{c = 12::Int, a = True, b = "Hello"}
```

These examples show simple ways to construct record values. We can also inspect the values held in a record using the traditional dot notation, where an expression of the form `r.l` simply returns the value of the `l` component in the record `r`. For example:

```
? {a = True, b = "Hello", c = 12::Int}.a
True
? let f r = r.b in f {a = True, b = "Hello"}
"Hello"
?
```

In all previous versions of Haskell, the '`.`' character has been used to represent function composition. It has also been used in more recent versions of Haskell in the syntax for qualified names. The first of these is clearly incompatible with our proposal, as it would allow a second reading of `r.l` as the composition of `r` with `l`. To avoid this conflict, we propose adopting a different symbol for function composition; we believe that `#` would be a good choice, but debate on that is beyond the scope of this paper. Our use of the dot notation is, however, entirely compatible with the syntax for qualified names, and with proposals for a structured module namespace in the style of Java packages. Another appealing consequence of this design is that it gives a single and consistent reading to the '`.`' character as selection, be it from a record or a module. On a practical level, this shows up in minor, but pleasing ways. For example, we can remove the rather ad-hoc restriction in the Haskell 98 syntax for qualified name that currently prohibits the use of spaces in a qualified name like `Prelude.map` that might otherwise have been confused with compositions like `Just . f`.

Repeated selections can be used to extract values from record-valued components of other records. As usual, "`.`" associates to the left, so that `r.l.k` is equivalent to `(r.l).k`:

```
? {a = True, b = {x="Hello"}, c = 12::Int}.b.x
"Hello"
?
```

3.2 Pattern matching

Record values can also be inspected by using pattern matching, with a syntax that mirrors the notation used for constructing a record:

```
? (\{a=x, c=y, b=_} -> (y,x))
  {a=True, b="Hello", c=12::Int}
(12,True)
?
```

The order of fields in a record *pattern* (unlike a record expression) is significant because it determines the order—from left to right—in which they are matched. Consider the following two examples:

```
? [x | {a=[x],b=True} <- [{b=undefined,a=[]},
                          {a=[2],b=True}]]
[2]

? [ x | {b=True, a=[x]} <- [{b=undefined, a=[]},
                            {a=[2],b=True}]]
Error: {program uses the undefined value}
?
```

In the first example, the attempt to match the pattern `{a=[x], b=True}` against the record `{b=undefined, a=[]}` fails because field `a` is matched first and `[x]` does not match the empty list; but matching the same pattern against `{a=[2],b=True}` succeeds, binding `x` to 2. Swapping the order of the fields in the pattern to `{b=True, a=[x]}` forces matching to start with the `b` component. But the first element in the list of records used above has `undefined` in its `b` component, so now the evaluation produces a run-time error message.

3.3 Record types

Like all other values in Haskell, records have types, and these are written in the form `{r}`, where `r` represents a 'row' that associates labels with types. For example, the record:

```
{c = 12::Int, a = True, b = "Hello"}
```

has type:

```
{a::Bool, b::[Char], c::Int}
```

This tells us, unsurprisingly, that the record has three components: an `a` field containing a `Bool`, a `b` field containing a `String`, and a `c` field of type `Int`. As with record val-

ues themselves, the order of the components in a row is not significant, and so the previous type can also be written as:

```
{b::String, c::Int, a::Bool}
```

In the special case when the row is empty, we obtain the empty record type {}, whose only value (other than ⊥) is the empty record, also written as {}.

Of course, the type of a record must be an accurate reflection of the fields that appear in the corresponding value. The following example produces an error because the specified type does not list all of the fields in the record value:

```
? {a=True, b="Hello", c=12} :: {b::String, c::Int}

ERROR: Type error in type signature expression
*** term           : {a=True, b="Hello", c=12}
*** type           : {a::Bool, b::[Char], c::a}
*** does not match : {b::String, c::Int}
*** because        : field mismatch

?
```

Notice that our system does not allow the kind of subtyping on record values that would permit a record like {a=True, b="Hello", c=12} to be treated implicitly as having type {b::String, c::Int}, simply by 'forgetting' about the a field. Finding an elegant and tractable way to support this kind of implicit coercion in a way that integrates properly with other aspects of the Haskell type system remains an interesting problem for future research. However, as we shall see in Section 3.7, our use of row polymorphism offers many of the benefits of subtyping.

3.4 Overloaded operations on records

Record types are automatically included in the standard Eq and Show classes of Haskell, provided that the types of each field in the records concerned are themselves instances of the appropriate class. Our interpreter uses these functions to allow comparison and display of record values in the following examples:

```
? {a=True, b="Hello"} == {b="Hello", a=True}
True
? {a = True, b = "Hello", c = 12::Int}
{a=True, b="Hello", c=12}
? {c = 12::Int, a = True, b = "Hello"}
{a=True, b="Hello", c=12}
?
```

Note that these operations always process record fields according the dictionary ordering of their labels. The fact that the fields appear in a specific (but, frankly, arbitrary) order is very important; the results of the (==) operator and the show function must be uniquely determined by their input, and not by the way in which that input is written. The records used in the last two lines of the example have exactly the same value, and so we expect exactly the same output for each. The difference in behavior between the following two examples is also a consequence of this:

```
? {a=0, b="Hello"} == {b=undefined, a=1}
False
? {b=0, a="Hello"} == {a=undefined, b=1}
Error: {program uses the undefined value}
?
```

In the first case, the equality test returns False because the two values differ in their first component, labeled as a. In the second case, where the labels have been switched, the equality test begins with an attempt to compare the string "Hello" with an undefined value, resulting in an error.

Arguably, records should automatically be instances of the classes Ord, Ix, Bounded, and Read, on the grounds that these are the classes (beyond Eq and Show) of which tuples are automatically instances. The difficulty is that the order of the fields matters even more for these four than they do for the former two. There is no difficulty in principle — fields can be lexically ordered — but the arbitrary nature of this ordering is apparent in more than just the strictness of the class methods.

3.5 Extension

An important property of our system is that the same label name can appear in many different record types, and potentially with a different type in each case. However, all of the examples that we have seen so far deal with records of some fixed shape, where the set of labels and the type of values associated with each one are fixed, and there is no apparent relationship between records of different type. In fact, all record values and record types in our system are built up incrementally, starting from an empty record and extending it with additional fields, one at a time. This is what it means for records to be *extensible*.

In the simplest case, any given record r can be extended with a new field labeled l, provided that r does not already include an l field. For example, we can construct the record {a=True, b="Hello"} by extending {a = True} with a field b="Hello":

```
? {{a=True} | b = "Hello"}
{a=True, b="Hello"}
?
```

Note that we write the record value that is being extended first, followed by a '|' character, and then by a list of the fields that are to be added. Another way to construct exactly the same result is by extending {b = "Hello"} with a field a=True:

```
? {{b = "Hello"} | a = True}
{a=True, b="Hello"}
?
```

It is often convenient to add more than one field at a time, as shown in the following example:

```
? {{b1="World"} | a=True, b="Hello", c=12::Int}
{a=True, b="Hello", b1="World", c=12}
?
```

On the other hand, a record cannot be extended with a field of the same name, even if it has a different type. The following example illustrates this:

```
? let r = {c=12::Int} in {r | c=True}
ERROR: {c::Int} already includes a "c" field
```

```
?
```

Much the same syntax can be used in patterns to decompose record values:

```
? (\{r | b=bval} -> (bval,r)) {a=True, b="Hello"}
("Hello",{a=True})
?
```

Notice that we can match, not just against individual components of a record value, but also against the portion of the record that is left after the explicitly named fields have been removed. In previous examples, we saw how a record could be extended with new fields. As this example shows, we can use pattern matching to do the reverse operation of removing fields from a record.

3.6 Update

It is often useful to update a record by changing the values associated with some of its fields. Update operations like this can be coded by hand, using pattern matching to remove the appropriate fields, and then extending the resulting record with the new values. However, it seems much more attractive to provide special syntax for these operations, using `:=` instead of `=` to distinguish update from extension. Slightly more formally, a record expression:

```
{r | x := e}
```

is treated as an abbreviation for the following update:

```
case r of {s | x=_} -> {s | x=e}
```

Providing a special syntax makes updates easier for programmers to code and also makes them easier for a compiler to recognize, which can often permit a more efficient implementation that avoids building the intermediate record `s`. For further convenience, we allow updates to be freely mixed with record extension in expressions like the following:

```
{r | x=2, y:=True}
```

Unlike the extension syntax, however, it does not seem sensible to allow the use of update syntax in a record pattern.

3.7 Row polymorphism

We can also use pattern matching to understand how selector functions are handled. For example, evaluating an expression of the form `r.l` is much like passing `r` as an argument to the function:

```
(\{_|l=value} -> value)
```

This function is polymorphic in the sense that it can be used with *any* record containing an `l` field, regardless of the type associated with that particular component, or of any other fields that the record might contain:

```
? (\{_|l=value} -> value) {l=True, b="Hello")
True
? (\{_|l=value} -> value)
  {name="Record", age=2, l="None")
"None"
?
```

To see how this works, we need to look at the type of this function, which can be inferred automatically as:

```
(r\x) => {r | x::a} -> a
```

There are two important pieces of notation here that deserve further explanation:

- `{r | l::a}` is the type of a record with an `l` component of type `a`. The *row variable* `r` represents the rest of the row; that is, it represents any other fields in the record apart from `l`. This syntax for record type extension mirrors the syntax that we have already seen in the examples above for record value extension. We discuss rows further in Section 3.8.
- The constraint `r\l` tells us that the type on the right of the `=>` symbol is only valid if "`r` lacks `l`," that is, if `r` is a row that does not contain an `l` field. If you are already familiar with Haskell type classes, then you may like to think of `\l` as a kind of class constraint, written with postfix syntax, whose instances are precisely the rows without an `l` field.

For example, if we apply our selector function to a record `{l=True,b="Hello"}` of type `{b::String, l::Bool}`, then we instantiate the variables `a` and `r` in the type above to `Bool` and `(b::String)`, respectively.

The row constraints that we see here can also occur in the type of any function that operates on record values if the types of those records are not fully determined at compile-time. For example, given the following definition:

```
average r = (r.x + r.y) / 2
```

our interpreter would infer a principal type of the form:

```
average :: (Fractional a, r\y, r\x)
        => {r | y::a, x::a} -> a
```

However, any of the following, more specific types could be specified in a type declaration for the `average` function:

```
average  :: (Fractional a) => {x::a, y::a} -> a
average  :: (r\x, r\y)
         => {r | x::Double, y::Double} -> Double
average  :: {x::Double, y::Double} -> Double
average  :: {x::Double, y::Double, z::Bool} -> Double
```

Each of these is an instance of the principal type given above.

These examples show an important difference between the system of records described here, and the record facilities provided by SML. In particular, SML prohibits definitions that involve records for which the complete set of fields cannot be determined at compile-time. So, the SML equivalent of the `average` function described above would be rejected because there is no way to determine if the record `r` will have any fields other than `x` or `y`. SML programmers usually avoid such problems by giving a type annotation that completely specifies the structure of the record. Of course, if a definition is limited in this way, then it also less useful.

With the expected implementation for our type system, as described in Section 3.9, there is an advantage to knowing the full type of a record at compile-time because it will allow the compiler to generate more efficient code. However, un-

like SML, the type system also offers the flexibility of polymorphism and extensibility over records if that is needed.

3.8 Rows

To deal more formally with record types, we extend the kind system of Haskell with a new kind, *row*: in a record type of the form `{expr}`, the expression `expr` must have kind *row*. Types of kind *row* are written using essentially the same notation that we use for records, but enclosed in parentheses rather than braces. For example:

- The empty row is written as `()`, and the empty record type `{}` is really just a convenient abbreviation for `{()}`. Note that this is a change from Haskell 98, where the symbol `()` is used to denote the unit type and its only proper (i.e., non bottom) value. With our proposal, the empty record, `{}` of type `{}`, can be used in place of a special unit value.
- Non-empty rows are formed by extension. For example, `(r|x::Int)` is the row obtained from row `r` by extending it with an `x` field of type `Int`. Multiple fields can be specified using comma-separated lists. For example, `(r|x::Int, y::Bool)` is a shorthand for `((r|x::Int)|y::Int)`. Another shorthand allows us to write extensions of the empty row as a comma-separated list of fields. For example, `(x::Int, y::Bool)` is an abbreviation for `(()|x::Int, y::Bool)`, which is in turn just an abbreviation for `((()|x::Int)|y::Bool)`. In all cases, we allow the outermost pair of parentheses to be omitted when a row expression appears inside a pair of braces. For example, `{(x::Int)}` can be abbreviated to `{x::Int}`.
- Row variables (i.e., type variables of kind *row*) represent unknown rows. As in Haskell, the kinds of all type variables are inferred automatically by the compiler using a simplified form of type inference.

Row expressions can be used anywhere that a type constructor of kind *row* is required, including the right hand side of a `type` definition, or the parameters of any programmer defined class or datatype constructor. For example, the following definitions introduce a type synonym, `Point`, of kind *row*, and then extend this to define a second type synonym `ColoredPoint` that adds an extra `Color` field:

```
-- Point :: row
type Point        = (x::Int, y::Int)

-- ColoredPoint :: row
type ColoredPoint = (Point | c::Color)
```

As the comments indicate, `Point` and `ColoredPoint` have kind *row*. (Haskell `type` declarations can already introduce type constructors of kinds other than $*$.)

This style of definition allows us to build up row types (and hence record types) in a style akin to single inheritance. It does not, however, support multiple inheritance. For example, the following definition of `ColoredPoint` is ill-formed because our proposal requires a field list to the right of a `|`, and does not permit arbitrary row expressions.

```
type Point           = (x::Int, y::Int)
type Coloring        = (c::Color)
type BadColoredPoint = (Point | Coloring) -- NO!
```

While we can model single inheritance, this style does not make it possible to define polymorphic functions. To illustrate this point, consider the following example:

```
move :: Int -> Int -> {Point} -> {Point}
move a b p = {p | x:=a, y:=b}
```

The function `move` works fine on values of type `{Point}` but it is type-incorrect to apply it to a value of type `{ColoredPoint}`.

However, it is easy to obtain a `move` that is applicable to points of all varieties by defining the types a little differently:

```
-- Point :: row -> row
type Point r = (r | x::Int, y::Int)

-- Colored :: row -> row
type Colored r = (r | c::Color)

-- ColoredPoint :: row -> row
type ColoredPoint r = Point (Colored r)
```

Notice that it is entirely legitimate for the type synonym `Point` to abstract over a row variable, so that `Point` itself has kind *row*$\rightarrow$*row*. Of course, the original definitions for each of these rows are just extensions of the empty row `()`. For example, with these definitions, we can write the type of a record `{x=0, y=0, c=Red}` as `{ColoredPoint()}`. Now we can define `move` thus:

```
move :: (r\x, r\y) => Int -> Int
                   -> {Point r} -> {Point r}
move a b p = {p | x:=a, y:=b}
```

The type neatly expresses that `move` works on any "subclass" (i.e., substitution instance) of `{Point r}`; any `{ColoredPoint s}` will do, for example. It also expresses that `move` returns a `Point` of the same variety as it is given as its argument, a well-known problem in many object systems (e.g., Cardelli and Mitchell [2, Section 2.6] discuss the "update problem" at some length). The observation that row polymorphism deals with this problem is not new [1].

Even though we have constructed `ColoredPoint` in a "sequential" way, row composition is commutative. For example, the following definition of `ColoredPoint` is entirely equivalent—to see this, just expand out the synonyms:

```
-- ColoredPoint :: row -> row
type ColoredPoint r = Colored (Point r)
```

We can also use variables of kind *row* as the parameters of user-defined datatypes, thus:

```
data T r = MkT {r | x :: Bool}
```

According to the normal rules for kind inference, `T` will be treated as a type constructor of kind *row* $\rightarrow *$, but it is clear that this kind is inaccurate; it does not seem sensible to allow `T` to be applied to *any* row argument, only to those that do not have an `x` field. Our kind system is not expressive enough to capture this restriction directly, but it can be reflected by including a constraint `r\x` in the type of `T`. From

a practical perspective, this is a minor issue; without any further restrictions, the type system will allow us to use types like `T (x::Int)` without flagging any errors, but it will not allow us to construct any values of that type, apart from ⊥. However, although it makes no real difference, it seems more consistent with other aspects of Haskell to require the definition of datatypes like `T` to reflect any constraints that are needed to ensure that their component types are well-formed. For example, we can correct the previous definition of `T` by inserting a `r\x` constraint, as follows:

```
data (r\x) => T r = MkT {r | x :: Bool}
```

(Haskell old-timers who remember the `Eval` class, may also recall that similar constraints were once required on datatypes that used strictness annotations.)

3.9 Implementation

A major merit of Gaster and Jones's record system is that it smoothly builds on Haskell's type class mechanism. This analogy applies to the implementation as well. The details are covered elsewhere [5] but the basic idea is simple enough.

Each "lacks" constraint in a function's type gives rise to an extra argument passed to that function that constitutes "evidence" that the constraint is satisfied. In particular, evidence that `r` lacks a field `l` is given by passing the offset at which `l` would be stored in the record `{r}` extended by `l`.

The size of the entire record is also required when performing record operations. This can be obtained either from the record itself, or by augmenting "evidence" to be a pair of the offset and record size.

As usual with overloading, much more efficient code can be obtained by specialisation. In the case of records, specialisation "bakes into" the code the size of the record and the offsets of its fields.

3.10 Additional Features

In this section, we collect together some small, but potentially useful ideas for further extensions of our core proposal.

3.10.1 Presentation of inferred types

One comment that some experienced users of the Trex system have made is that user-written type signatures become unreasonably large. For example, consider the type signature for `move` in Section 3.8:

```
move :: (r\x, r\y) => Int -> Int -> {Point r} -> {Point r}
move a b p = {p | x:=a, y:=b}
```

The `Point` synonym allowed us not to enumerate (twice) the details of a `Point`, but the context `(r\x, r\y)` must enumerate the fields that `r` must lack, otherwise the type is ill-formed. This is annoying, because, if we expand the type synonym, it is absolutely manifest that `r` must lack fields `x` and `y`:

```
move :: (r\x, r\y) => Int -> Int
     -> {r | x::Int, y::Int}
     -> {r | x::Int, y::Int}
```

Not only is it annoying, but it is also non-modular: adding a field to `Point` will force a change to the type signature of `move`, even if `move`'s code does not change at all. In practice, these annoyances are enough to cause programmers to omit type signatures altogether on functions with complex types — arguably just the functions for which a type signature would be most informative.

Thus motivated, an obvious suggestion is to permit constraints in a type signature to be omitted if they could be inferred directly from the rest of the signature. This is akin to the omission of explicit universal quantification. Haskell already lets us write `f::a->a`, when we really mean `f::forall a.a->a`. The "`forall a`" is inferred. In a similar way, we propose that a similar inference process adds "lacks" constraints to a type signature, based solely on the rest of the type signature (after expanding type synonyms).

Note that this is a matter of presentation only, and the actual types used inside the system do not change. Constraints of this form cannot always be omitted from the user type signature, as illustrated in the following (pathological) example:

```
g  :: (r\l) => {r} -> Bool
g x = {x | l=True}.l
```

As a slightly more realistic example where constraints cannot be omitted, consider the following datatype of trees, which allows each node to be annotated with some additional information `info`:

```
data Tree info a b
  = Leaf {info | value :: a}
  | Fork {info | left  :: Tree info a b,
                 value :: b,
                 right :: Tree info a b}
```

In an application where there are many references to the height of a tree, we might choose to add height information to each node, and hence avoid repeated unnecessary repeated computation:

```
addHeight (Leaf i) = Leaf {i | height=0}
addHeight (Fork i)
   = Fork {i | left := l, right := r,
               height = 1 + max (height l)
                                (height r) }
     where l = addHeight i.left
           r = addHeight i.right

height (Leaf i) = i.height
height (Fork i) = i.height
```

Careful examination of this code shows that the type of `addHeight` is:

```
(info\height, info\left, info\right)
  => Tree info a b -> Tree (info | height::Int) a b
```

Note here that only the first of the three constraints, `info\height`, can be inferred from the body of the type, and hence the remaining two constraints cannot be omitted. In our experience, such examples are quite uncommon, and we believe that many top-level type signatures could omit their "lacks" constraints, so this facility is likely to be very attractive in practice.

3.10.2 Tuples as values

As in Standard ML, records can be used as the underlying representation for tuples; all that we need to do is pick out canonical names for each position in a tuple. For example, if we write `field1` for the label of the first field in a tuple, `field2` for the second, and so on, then a tuple value like `(True,12)` is just a convenient shorthand for `{field1=True, field2=12}`, and its type `(Bool,Int)` is just a shorthand for `{field1::Bool, field2::Int}`. The advantages of merging currently separate mechanisms for records and tuples are clear, as it can remove redundancy in both the language and its implementations. In addition, it offers a more expressive treatment of tuples because it allows us to define functions like:

```
fst  :: (r\field1) => {r|field1::a} -> a
fst r = r.field1
```

that can extract the first component from any tuple value; in current versions of Haskell, the `fst` function is restricted to pairs, and different versions must be defined for each different size of tuple.

The exact choice of names for the fields of a tuple is a matter for debate. For example, it would even be possible (though not necessarily desirable) to use the unadorned integers themselves — e.g. `x.2`, `{r | 3=True}`.

3.10.3 Constructors as field names

Programmers often use algebraic data types to define *sums*; for example:

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
```

In such cases it is common to define projection functions:

```
just :: Maybe a -> a
just (Just x) = x

left :: Either a b -> a
left (Left x) = x

right :: Either a b -> b
right (Right y) = y
```

An obvious notational convenience would be to re-use the "dot" notation, an allow a constructor to be used after the dot to indicate a projection. That is, `m.Just` would be equivalent to `just m`, and `e.Left` would be equivalent to `left e`, and so on. This would often avoid the need to define the projection functions explicitly.

This notation is particularly convenient for Haskell 98 `newtype` declarations, which allow one to declare a new data type isomorphic to an old one. For example:

```
newtype Age = Age Int
```

Here `Age` is isomorphic to `Int`. The "constructor" `Age` has the usual type

```
Age :: Int -> Age
```

and allows one to convert an `Int` into an `Age`. (We put "constructor" in quotes, because it is implemented by the identity function, and has no run-time cost.) The reverse coercion is less convenient but, if the constructor could be used as a projection function, one could write `a.Age` to coerce a value `a::Age` to an `Int`.

The proposal here is entirely syntactic: to use the same "dot" notation for sum projections as well as for record field selection. The two are syntactically distinguishable because constructors begin with an upper-case letter, whereas fields do not.

3.10.4 Punning

It is often convenient to store the value associated with a particular record field in a variable of the same name. Motivated by practical experience with records in the Standard ML community, it is useful to allow a form of *punning* in the syntax for both record expressions and patterns. This allows a field specification of the form `var` is treated as an abbreviation for a field binding of the form `var=var`, and is referred to as a *pun* because of the way that it uses a single name in two different ways. For example, `(\{x,y,z} -> x + y + z)` is a function whose definition uses punning to sum the components of a record. Punning permits definitions such as:

```
f :: {x::Int, y::Int} -> {x::Int, y::Int}
f {x,y} = {x=y-1, y=x+1}
```

Here, in the expressions `y-1` and `x+2`, the variables `x` and `y` are bound to the fields of the same name in `f`'s argument.

Punning was also supported in some versions of Haskell prior to Haskell 98, but was removed because of concerns that it was not well behaved under renaming of bound variables. For example, in the definition

```
f :: Int -> Int
f x = x+1
```

one can rename both occurrences of "`x`" to "`y`". But in the definition:

```
f :: {x::Int} -> Int
f {x} = x+1
```

one cannot perform such a renaming, because `x` is serving as a record label. In fact punning is perfectly well-behaved under these circumstances, provided one remembers that it is simply an abbreviation, which may need to be expanded before completing the task of renaming:

```
f :: {x::Int} -> Int
f {x=x} = x+1
```

Now one can rename as follows:

```
f :: {x::Int} -> Int
f {x=y} = y+1
```

Anecdotal experience from the Standard ML community suggests that the merits of punning greatly exceed the disadvantages.

3.10.5 Renaming

It is easy to extend the set of supported operations on records to include a renaming facility, that allows us to change the name associated with a particular field in a record. This notation can be defined more formally in terms of a simple translation:

```
{r | x->y}  =  case r of {s | x=t} -> {s | y=t}
```

However, as in the case of update (Section 3.6), use of this notation makes it easier for programmers to use renaming, and easier for a compiler to implement it.

3.10.6 Kind signatures for type constructors

Earlier in this paper we used comments to indicate the kinds of type constructors in examples like:

```
-- Point :: row -> row
type Point r = (r | x::Int, y::Int)
```

The clear implication is that Haskell should provide a way to give kind signatures for type constructors, perhaps simply by permitting the above commented kind signature. Such kind signatures are syntactically distinguishable from value type signatures, because type constructors begin with an upper case letter. Another alternative would be to allow kind annotations of the form:

```
type Point (r::row) :: row = (r | x::Int, y::Int)
```

Annotations like this would also be useful elsewhere, such as in `data` of `class` declarations.

The need for explicit kind information is not restricted to extensible records. In this very workshop proceedings, Hughes writes [7]:

```
data Set cxt a = Set [a] | Unused (cxt a -> ())
```

The *only* reason for the `Unused` constructor which, as its name implies, is never used again, is to force `cxt` to have kind $* \to *$. It would be far better to say:

```
Set :: (*->*) -> * -> *
data Set cxt a = Set [a]
```

3.10.7 Topics for further work

Our proposal does not support all of the operations on records that have been discussed in the literature. Examples of this include:

- Record concatenation. This allows the fields of two distinct records to be merged to form a single record. Several researchers have studied this operator, or closely related variants. For example, Wand [14] used it as a way to describe multiple inheritance in object-oriented languages, and Rémy [13] described a technique for typing a form of record concatenation for 'free' in any language supporting record extension.
- Unchecked operations. These are variations of the operations on records that we have already seen that place slightly fewer restrictions on the types of their input parameters. For example, an unchecked extension operator guarantees that the specified field will appear in its result with the corresponding value, regardless of whether there was a field of the same name in the input record. With the checked operators that we have presented in this paper, the programmer must distinguish between the two possible cases using extension or update, as appropriate. Unchecked operations are supported, for example, in Rémy's type system for records in a natural extension of ML [12].
- First-class labels. This allows labels to be used and manipulated as program values, with a number of potentially useful applications. A prototype implementation was developed by Gaster [4], but there are still some details to be worked out.

It is not yet clear whether our proposal could be extended to accommodate these operations, and we believe that each of them would make interesting topics for future work.

4 Comparison with other systems

In this section we provide brief comparisons of our proposal with the facilities for defining an using records in other systems. We focus here on practical implementations, and refer interested readers to the work of Gaster and Jones [5] for comparisons of the underlying theory with other more theoretical proposals.

4.1 Comparison with Standard ML

The system of records in Standard ML was one of the original inspirations for this proposal, but of course our system also supports extensibility, update, and polymorphic operations over records. This last point shows up in Standard ML when we try to use a record in a situation where its corresponding set of field labels cannot be determined at compile-time, and resulting in a compile-time error.

4.2 Comparison with SML#

Based on his earlier work on type systems for records [11], Atsushi Ohori built a version of the Standard ML interpreter known as SML#, which extends the SML record system with support for update and for polymorphic operations over records[1]. Ohori's system does not provide the separation between rows and records that our proposal offers (Section 3.8), nor is it clear how records would interact with type classes, but it would be wrong to criticize SML# on the basis of these omissions, because they are much more relevant in the context of Haskell, with its advanced kind and class system, than they are in SML. Thus, apart from differences in syntax, the main advantage of our system over Ohori's is the support that it provides for extensibility.

[1] Further information about SML# is available on the World Wide Web at `http://www.kurims.kyoto-u.ac.jp/~ohori/smlsharp.html`.

4.3 Comparison with Trex

The proposal presented in this paper is closely related to the Trex implementation in current releases of Hugs 98 [10]. The only real differences are in the choice of notation:

- In record types, Trex uses `Rec r` where this proposal uses `{r}`. The latter choice could not be used with Trex because of the conflict with the syntax for labeled fields in Haskell 98.
- In record values, Trex uses `(...)` where this proposal uses `{...}`. The latter could not be used in Trex because it conflicts with the update syntax for datatypes with labeled fields in Haskell 98. For example, in Haskell 98, an expression of the form `e{x=12}` is treated as an update of value `e` with an `x` field of `12`. For the current proposal, we would expect to treat this expression as the application of a function `e` to a record `{x=12}` with just one field.
- Trex uses `(x::a | r)` where this proposal uses `(r | x::a)`. We deviate from Trex because it can be easy for the trailing "`| r`" to become lost when it follows a large block of field definitions. (In a similar way, Haskell puts guards at the beginning of an equation defining a function, rather than at the end as some languages do.) This choice is ultimately a matter of taste — we have not found any compelling technical reason to justify the use of one of these over the other.
- Like SML, Trex uses `#l` to name the selector for a field `l`; this proposal uses dot notation for field selection, and the function `#l` must be written as `(\r -> r.l)`. Dot notation could not be used in Trex because it conflicts with the use of `.` for function composition in Haskell.
- Trex does not support the update notation; update is one of several features that appeared in the original work on Trex that were not implemented in the Hugs prototype.
- Trex uses `EmptyRow` where this proposal uses `()`; the latter could not be used in Trex because it conflicts with the notation used for the unit type in Haskell 98.
- Trex does not use punning (Section 3.10.4) because of a conflict with the syntax for tuples: an expression like `(x,y)` could be read in two different ways, either as a tuple, or as an abbreviation for the record `(x=x, y=y)`.

In short, the current proposal differs in only small ways from Trex, and most of the changes were made possible only by liberating ourselves from any need to retain compatibility with the syntax of Haskell 98.

4.4 Comparison with Gaster's proposal

Our proposal is quite similar to that of [3]. Most notably, we both adopt the idea of using "`.`" for record selection.

We have gone further than Gaster by abandoning Haskell 98's current record system altogether, using "`()`" for the empty row instead of the unit tuple, providing a syntax for record updates (Section 3.6), and using constructors as selectors (Section 3.10.3). We have also elaborated a little more on the implications of row polymorphism. But the two proposals clearly share a common foundation.

5 Shortcomings of our proposal

One of the main reasons to turn a general idea into a concrete design is to highlight difficulties that deserve further attention.

5.1 Where Haskell 98 does better

We began this paper by describing some of the weaknesses of the labeled field mechanism in Haskell 98, and using those to motivate the key features of this proposal. In this section, therefore, we focus on areas where the Haskell 98 approach sometimes offers advantages over our proposal.

- *The double-lifting problem.* Part of the price that we pay for having lightweight records is that it becomes more expensive to embed a record in a datatype. For example, with our proposal, the definition of the following datatype introduces two levels of lifting:

  ```
  data P = MkP {x::Double, y::Double}
  ```

  In semantic terms, this means that the datatype `P` contains both $\perp$ and `MkP` $\perp$ as distinct elements. In implementation terms, it means that an attempt to access the `x` or `y` coordinates will require a double indirection. In comparison, the same definition in Haskell 98 introduces only one level of lifting. The same behavior can be recovered in our proposal by adding a strictness annotation to the record component of the datatype.

  ```
  data P = MkP !{x::Double, y::Double}
  ```

- *The unpacking problem.* Even if we use the second definition of the `P` datatype, it will only help to avoid two levels of indirection when we construct a value of type `P`; we still need a two stage process to extract a value from a datatype. For example, to extract the `x`, we must first remove the outer `MkP` constructor to expose the record from which the required `x` field can be obtained.

  This provides an additional incentive to adopt the use of constructors as field selectors (Section 3.10.3). This would allow the selection of the `x` component from a value `p` of type `P` to be written more succinctly as `p.MkP.x`.

  This approach does not help us to deal with examples like the `FileSystem` datatype from Section 1, where the same field name appears in multiple branches of an algebraic datatype. In Haskell 98, the compiler takes care of generating an appropriate definition for the selector function. With our proposal, this must be coded by hand:

  ```
  name            :: FileSystem -> String
  name (File r)   = r.name
  name (Folder r) = r.name
  ```

- *Strictness annotations.* Haskell 98 allows individual components of a datatype to be marked with strictness annotations, as in the following example:

  ```
  data P = MkP {x :: Double, y :: !Double}
  ```

  The proposal described in this paper does not allow this because record types are lightweight, not declared. An advantage is that the same labels can be used in different types. The disadvantage here is that there is no way to attach any special meaning, in this case a strictness annotation, to any particular label. One way to overcome this restriction would be to use lexically distinct sets of field labels to distinguish between strict and non-strict components. Alternatively, we could introduce strict versions of the extension and update operators. The problem with this approach is that strict evaluation will only be used when the programmer remembers to insert the required annotations.

The impact of these problems will depend, to a large extent on the way that records are used within algebraic datatypes.

5.2 Polymorphism

Although it is not part of the Haskell 98 standard, both Hugs and GHC allow the components of an algebraic datatype to be assigned polymorphic types. A standard example of this might be to define a concrete representation for monads as values of the following type:

```
data Mon m
  = MkMon {unit :: forall a. a -> m a,
           bind :: forall a, b. m a -> (a -> m b) -> m b}
```

To support the use of this datatype, the `MkMon` constructor, and, to a lesser degree, the `unit` and `bind` selectors are given a special status in the type checker, and are used to propagate explicit typing information to places where values of the datatype are used. (Type inference would not be possible without such information.) With our proposal, this special status is lost: all records are constructed in the same way, and all fields are selected in the same way. For example, the function to extend a record with a `unit` field is just:

```
(\r u -> {r | unit=u})
  :: (r\unit) => {r} -> a -> {r|unit::a}
```

The type variables `r` and `a` here range over monotypes (of kind *row* and $*$, respectively), and there is nothing to hint that a polymorphic value for `unit` should be expected.

Intuitively, it seems clear that we should still be able to propagate programmer-supplied type information to the places where it is needed, but the mechanisms that we need to support this are rather different from the mechanisms that are used to support the current Hugs and GHC extensions illustrated above. One promising approach, previously used in work with *parameterized signatures* [9], is to use so-called "has" predicates for records instead of the "lacks" predicates used here. These "has" predicates provide a looser coupling between records and their component types, which delays the need to resolve them to a point where more explicit typing information is likely to be available. However, it is not immediately obvious how we can integrate this approach with the main proposals in this paper, which rely instead on "lacks" predicates.

Another possibility is to provide a special typing rule for the syntactic composition of constructor application and record construction, effectively recovering the rule for constructors used by GHC and Hugs. GHC and Hugs's constructor-application rule is already restricted to the case where the constructor is applied to enough arguments to saturate all its universally-quantified arguments (e.g., `map MkMon xs` is rejected); requiring the record construction to be syntactically visible is arguably no worse.

5.3 Instances

In Section 3.4 we propose that records are automatically instances of certain built-in classes (`Eq`, `Show`, etc), and no others. Like any user-defined type, programmers may want to make a record type an instance of other classes, or to provide their own instance decalaration for the built-in classes. It is possible to define such instances for records whose shape (i.e., set of field names) is fixed. For example, we could define tuples as instances of standard Haskell classes in this manner:

```
instance (Ord a , Ord b)
    => Ord {field1::a, field2::a} where ...
```

However, some care is required to deal with instances for record types involving extension. To illustrate this point, consider the following collection of instance declarations:

```
instance C {r|x::Int}  where ... -- OK
instance C {r|x::Bool} where ...

instance D {r|x::a} where ...   -- INSTANCES
instance D {r|y::b} where ...   -- OVERLAP!
```

The first pair of instances for class `C` are acceptable, but the second pair will be rejected because they overlap. For example, these declarations provide two distinct, and potentially ambiguous ways for us to demonstrate that a type like `{x::Int,y::Bool}` is an instance of `D`. An overlap like this would not be a problem if we could be sure that both options gave the same final result, but there is no obvious way to guarantee this.

The trouble is that declarations like those for class `D` seem necessary for modular user-defined instances of record types. For example, imagine trying to declare `Eq` instances for a record. One might be led to say:

```
instance (Eq a, Eq {r}) => Eq {r | x::a} where
  {r1 | x=x1} == (r2 | x=x2}
       = x1 == x2 && {r1} == {r2}
```

But we need one such instance declaration for each distinct field label, which leads to declarations just like those for `D` above. The ambiguity in this case boils down to defining the order in which fields are compared. A way out of this impasse is an obvious piece of further work.

Acknowledgements

We would like to acknowledge our debt to John Hughes, David Espinosa, and the Haskell workshop referees, for their constructive feedback on this paper.

References

[1] L. Cardelli. Extensible records in a pure calculus of sub-typing. In Gunter and Mitchell [6], pages 373–426.

[2] L. Cardelli and J. Mitchell. Operations on records. In Gunter and Mitchell [6], pages 295–350.

[3] B. Gaster. Polymorphic extensible records for Haskell. In J. Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[4] B. Gaster. *Records, variants, and qualified types.* PhD thesis, Department of Computer Science, University of Nottingham, 1998.

[5] B. Gaster and M. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[6] C. Gunter and J. Mitchell, editors. *Theoretical aspects of object-oriented programming.* MIT Press, 1994.

[7] R. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Haskell workshop*, Paris, Sept. 1999.

[8] M. Jones. A theory of qualified types. In *European Symposium on Programming (ESOP'92)*, number 582 in Lecture Notes in Computer Science, Rennes, France, Feb. 1992. Springer Verlag.

[9] M. Jones. Using parameterized signatures to express modular structure. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 68–78. ACM, St Petersburg Beach, Florida, Jan. 1996.

[10] M. Jones and J. Peterson. Hugs 98 user manual. Technical report, Oregon Graduate Institute, May 1999.

[11] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995.

[12] D. Rémy. Type inference for records in a natural extension of ML. In Gunter and Mitchell [6].

[13] D. Rémy. Typing record concatenation for free. In Gunter and Mitchell [6].

[14] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.

A Generic Programming Extension for Haskell

Ralf Hinze (Bonn University, Germany)

A Generic Programming Extension for Haskell

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany

ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

Many functions can be defined completely generically for all datatypes. Examples include pretty printers (eg `show`), parsers (eg `read`), data converters, equality and comparison functions, mapping functions, and so forth. This paper proposes a generic programming extension that enables the user to define such functions in Haskell. In particular, the proposal aims at generalizing Haskell's `deriving` construct, which is commonly considered deficient since instance declarations can only be derived for a few predefined classes. Using generic definitions derived instances can be specified for arbitrary user-defined type classes and for classes that abstract over type constructors of first-order kind.

1 Introduction

Generic or polytypic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different datatypes. Typical examples for so-called generic functions include pretty printers (eg `show`), parsers (eg `read`), functions that convert data into a universal datatype (eg bit strings, character strings, XML documents), equality and comparison functions, mapping functions (`map`), and so forth. This paper proposes a generic programming extension for Haskell that enables the user to define such functions completely generically for all datatypes, including nested datatypes, mutually recursive datatypes, and datatypes involving function spaces. The central idea is to parameterize a function definition with a type or a type constructor and to define the function by induction on the structure of types or type constructors.

Interestingly, Haskell already provides a rudimentary form of genericity. Its derivable class methods can be seen as simple examples for generic functions. Using the `deriving` construct instance declarations (ie collections of class methods) can be automatically generated for user-defined datatypes. Every Haskell programmer probably uses this mechanism to derive, for instance, `Eq` or `Show` instances (ie `(==)` and `show` methods) for newly defined datatypes. As matters stand, the `deriving` construct suffers from several drawbacks. First of all, the definitions of the generic functions are hard-wired into the various Haskell compilers. This implies, in particular, that libraries that contain derivable classes (such as `Ix`) have a special status since the Haskell compiler must know details of their code. Clearly, this is an undesirable situation. Furthermore, derived instances are only informally defined in the Haskell Report [19, Appendix D]. For instance, Section D.4 of the Report contains the following statement.

> "A precise definition of the derived `Read` and `Show` instances for general types is beyond the scope of this report."

Second, instances can only be derived for a few predefined classes. The user can neither modify existing derivable classes (for instance, to adapt `show` to her needs) nor can she define derivable classes herself. Finally, `deriving` does not work for all user-defined types—a fact that is not very well known. For instance, if a datatype is parameterized by a type constructor as in

```
data Twice m v =  Twice (m (m v))
```

then deriving `Eq` or `Show` fails in general. The reason for this failure, which is quite fundamental, is explained in Section 3.1.

If we augment Haskell by generic definitions, all of these problems can be overcome. The meaning of derived instances can be made precise (Appendix B, for instance, precisely defines the generic `show` method), the user can define her own derivable classes, and instances can be derived for all datatypes definable in Haskell.

In the rest of this introduction let us take a brief look at generic definitions. We have already mentioned the basic idea that a generic function is parameterized by datatype and is defined by induction on the structure of types. Now, the types introduced by Haskell's `data` construct are essentially sums of products. The components of a product may depend on other user-defined datatypes or on primitive types such as `Char`, `Int`, or '`->`'. As an example, consider the datatype of polymorphic binary trees (the constructor `Bin` contains an additional size field).

```
data Tree a = Tip | Bin Int (Tree a) a (Tree a)
```

Let us make the structure of this definition more explicit by writing '`+`' for sums and '`*`' for products.

```
Tree a = 1  +  Int * Tree a * a * Tree a
```

Here `1` is an abbreviation for the unit type. The equation makes explicit that `Tree` is a binary sum; the first component of the sum is a 0-tuple and the second is a 4-tuple, which can be represented by a nested binary product. Likewise, all datatypes can be decomposed into binary sums, binary products, and primitive types. Consequently, to define a function generically for all datatypes it suffices to specify its

action on binary sums, binary products, and primitive types. As an example, the following equations define the generic function `encode(t)`, which converts elements of type `t` into bit strings implementing a simple form of data compression.

```
data Bit                = 0 | I

encode(t)               :: t -> [Bit]
encode(1)     ()        = []
encode(Char)  x         = encodeChar x
encode(Int)   x         = encodeInt  x
encode(a+b)   (Left  x) = 0 : encode(a) x
encode(a+b)   (Right y) = I : encode(b) y
encode(a*b)   (x, y)    = encode(a) x
                        ++ encode(b) y
```

The first parameter of `encode` is its type argument. The type signature makes explicit that the type of `encode(t)` depends on the type argument `t`. The definition proceeds by case distinction on `t`: we have one equation for `1`, `Char`, `Int`, and '`*`' and two equations for '`+`'. Note that `1`, '`+`', and '`*`' serve as abbreviations for the predefined types `()`, `Either`, and `(,)` which are given by the following definitions.

```
data ()          =  ()
data Either a b  =  Left a | Right b
data (a, b)      =  (a, b)
```

Let us consider each equation of `encode(t)` in turn. To encode the single element of `1` no bits are required. Characters and integers are encoded using the primitive functions `encodeChar` and `encodeInt`, whose existence we assume. To encode an element of a sum we emit one bit for the constructor followed by the encoding of its argument. Finally, the encoding of a pair is given by the concatenation of the component's encodings. Given this definition we can compress elements of arbitrary datatypes. For instance, `encode(Tree Char)` of type `Tree Char -> [Bit]` compresses binary trees with character labels.

The rest of this paper is organized as follows. Section 2 introduces the various forms of generic definitions and sketches their semantics. Section 3 explains how to blend generic definitions with Haskell's type classes. In particular, we show how to define derivable type classes. Section 4 presents several examples for generic functions. Section 5 reviews related work and Section 6 concludes. Appendix A details the necessary changes to the syntax and Appendix B lists the modifications to the Standard Prelude and to the Standard Libraries. Note that this paper describes the generic programming extension primarily from the user's point of view. The theoretical background of this extension is described in a companion paper [7]. Furthermore, note that the extension has not been implemented yet but we plan to do so in the near future.

2 Generic definitions

2.1 Type-indexed values

A generic function—or rather, a generic value is either defined inductively or in terms of other generic values. Here is the prototype for an inductive definition.

```
g(t)    :: ... t ...
g(1)    =  ...
g(Char) =  ...
g(Int)  =  ...
...
g(a+b)  =  ... g(a) ... g(b) ...
g(a*b)  =  ... g(a) ... g(b) ...
g(a->b) =  ... g(a) ... g(b) ...
...
```

Type signatures are mandatory for generic values. In fact, the type argument of `g`, which appears on the left-hand side of the signature, identifies `g` as being indexed by type. The equations for `g` employ pattern matching on types. The type patterns on the left-hand sides are restricted to type expressions of the form `T` a_1 ... a_k, ie a type constructor is applied to type variables a_1, ..., a_k, which have to be distinct. Since `g` is inductively defined, the recursive calls on the corresponding right-hand sides must have the form `g(a)` where `a` is one of the type variables a_1, ..., a_k.

Generic values are usually *not* defined for all types. For instance, `encode(t)` cannot compress functions or floating-point numbers. To be exhaustive a generic definition must include cases for `1`, '`+`', and '`*`'—these cases cover `data` declarations—and additionally for all primitive types that are not defined via a `data` declaration. All in all, there are a dozen primitive types: the Standard Prelude defines `a->b`, `Char`, `Double`, `Float`, `Int`, `Integer`, `IO a`, and `IOError` and the Standard Libraries provide `Array a b`, `Permissions`, `ClockTime`, and `StdGen`.

A generic value is instantiated or applied simply by calling it with an appropriate type argument, which may be an arbitrary type expression. We only require the type expression to be closed, ie it may not contain any type variables.

Generic values can be defined in terms of other generic values. As an example, consider the function `dump(t)`, which writes elements of type `t` in binary format to a given file (`toChar` maps a 16-bit value to a character and `rsplit n` splits its list argument into a list of lists, each of which has length `n`—only the last element may have a length smaller than `n`).

```
dump(t)          :: FilePath -> t -> IO ()
dump(t) fpath x  =  writeFile fpath
                 $  map toChar
                 $  rsplit 16
                 $  encode(t) x
```

Note that the type signature is again mandatory and that the type parameter of `dump` must not be omitted. Since type arguments are in general needed for disambiguation—Section 2.6 provides an example—we follow the design principle that value-type dependencies must be made explicit. In general, a non-inductive definition has the following form.

```
g(t) :: ... t ...
g(t) =  ... h(... t ...) ... i(... t ...) ...
```

Here `t` is a type variable. The right-hand side of the definition may involve calls of other generic values. Their type arguments may contain the type variable `t` but no other type variables.

2.2 Semantics

This section gives a brief account of the semantics of generic definitions. First of all, note that generic definitions operate on *binary* sums and products. This is a design decision dictated by pragmatic concerns. We could have based generic definitions on *n-ary* sums and products fitting them more closely to datatype declarations. It appears, however, that

n-ary type constructors are awkward to work with. Therefore, we must make precise how n-ary sums and products are translated into binary ones.

Roughly speaking, the right-hand side of a datatype declaration, $c_1 \mid \cdots \mid c_n$, is translated to the type expression $\Sigma(c_1 \ldots c_n)$ defined as follows.

$$\Sigma(t_1 \ldots t_n) = \begin{cases} t_1 & \text{if } n = 1 \\ \Sigma(t_1 \ldots t_{\lfloor n/2 \rfloor}) + \Sigma(t_{\lfloor n/2 \rfloor + 1} \ldots t_n) & \text{if } n > 1 \end{cases}$$

The translation essentially creates a balanced expression tree. For instance, $c_1 \mid c_2 \mid c_3 \mid c_4$ is mapped to $(c_1 + c_2) + (c_3 + c_4)$. All generic definitions in this paper with the notable exception of `encode(t)` and `decode(t)` (see Section 4) are insensitive to the translation of n-ary sums. For `encode(t)` this translation scheme is a sensible choice (an unbalanced or a list-like scheme would aggravate the compression rate).

Likewise, the components of a constructor, $C\ t_1\ \ldots\ t_k$, are translated to $\Pi(t_1 \ldots t_k)$ given by

$$\Pi(t_1 \ldots t_n) = \begin{cases} \texttt{1} & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ \Pi(t_1 \ldots t_{\lfloor n/2 \rfloor}) * \Pi(t_{\lfloor n/2 \rfloor + 1} \ldots t_n) & \text{if } n > 1\ . \end{cases}$$

The generic values defined in this paper happen to be insensitive to the encoding of n-ary products. It is, however, not difficult to define a function that yields different results for different translation schemes. Figure 1 contains the complete translation of `data` and `newtype` declarations. Note that the resulting type expressions include constructor names and record labels. Section 2.3 explains why this is useful. Strictness annotations are, however, ignored. Whether this is a viable choice remains to be seen.

Now, the central idea of the semantics is that type expressions can be interpreted in a fairly straightforward manner as values. To exemplify, the equation for `Tree`

```
Tree a = Tip 1 + Bin ((Int*Tree a) * (a*Tree a))
```

can be seen as defining a one-argument function, which maps type expressions to type expressions. In general, we interpret primitive types and type constructors (such as `1`, '`+`', '`*`', `Char`, `Int`, '`->`' etc) as *value constructors* and types that are defined by datatype declarations as *functions*. Since `Tree` is recursively defined, the resulting type expression is, of course, infinite. In this particular case the type expression can be represented by a circular structure but this does not hold in general, see [8]. Fortunately, Haskell is a non-strict language and is able to generate and to process infinite data structures with ease. Consequently, we may interpret generic values as 'ordinary' functions, which operate on possibly infinite type expressions. It is not possible, however, to specify this semantics via a translation into Haskell. The reason is simply that Haskell's type system is not expressive enough. To specify the type of generic values dependent types are required.

We have seen that datatypes are translated into binary sums and products. This translation must, of course, be duplicated on the value level. For instance, the argument `x` in `encode(Tree Char) x` must be converted to $[\![\texttt{x}]\!]$ given by

$$\begin{aligned} [\![\texttt{Tip}]\!] &= \texttt{Left ()} \\ [\![\texttt{Bin}\ e_1\ e_2\ e_3\ e_4]\!] &= \texttt{Right ((}[\![e_1]\!]\texttt{,}[\![e_2]\!]\texttt{),(}[\![e_3]\!]\texttt{,}[\![e_4]\!]\texttt{))}\ . \end{aligned}$$

In general, a constructor application $K_i\ e_1 \ldots e_{k_i}$, where K_i is introduced via the declaration

$$\texttt{data}\ T\ a_1 \ldots a_m \quad = \quad K_1\ t_{11} \ldots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \ldots t_{nk_n}\ ,$$

is translated as follows

$$[\![K_i\ e_1 \ldots e_{k_i}]\!] \quad = \quad \sigma_i^n(\pi([\![e_1]\!] \ldots [\![e_{k_i}]\!]))\ .$$

The function σ_i^n yields the binary encoding of the constructor K_i.

$$\sigma_i^n(v) = \begin{cases} v & \text{if } n = 1 \\ \texttt{Left}\ (\sigma_i^{\lfloor n/2 \rfloor}(v)) & \text{if } i \leqslant \lfloor n/2 \rfloor \\ \texttt{Right}\ (\sigma_{i-\lfloor n/2 \rfloor}^{n-\lfloor n/2 \rfloor}(v)) & \text{otherwise} \end{cases}$$

The definition may look daunting but it has a simple interpretation. If we draw the binary sum as a tree, then σ_i^n specifies the path from the root to the i-th leaf. The encoding of the components of a constructor is given by

$$\pi(v_1 \ldots v_n) = \begin{cases} \texttt{()} & \text{if } n = 0 \\ v_1 & \text{if } n = 1 \\ (\pi(v_1 \ldots v_{\lfloor n/2 \rfloor}), \pi(v_{\lfloor n/2 \rfloor + 1} \ldots v_n)) & \text{if } n > 1\ . \end{cases}$$

Of course, we also require the inverse translation, which decodes encoded values. The inverse translation must be applied whenever a generic function produces values whose type depends on the type argument. We leave it to the reader to fill out the details.

Generic definitions can be *implemented* in at least two ways. The semantics sketched above describes an interpretative approach where a generic value repeatedly pattern matches on its type argument. Clearly, this approach is rather inefficient especially since the type argument is statically known at compile time. The interpretative layer can be removed by specializing a generic value with respect to a given type. The process of specialization, which can be seen as a very special instance of partial evaluation, is described at length in [7]—Section 3.1 states the basic idea. Note that the restrictions on the form of generic definitions have been chosen in order to guarantee that specializing a generic definition is always feasible.

2.3 Accessing constructor and label names

The function `encode(t)` can be seen as a very simple printer. To be able to define a *pretty* printer, which produces a human-readable output, we must make the names of the constructors accessible. To this end we provide an additional type pattern of the form `c a` where `c` is a *value* variable of type `Con` and `a` is a *type* variable. The type `Con` is a new primitive type that comprises all constructor names. To manipulate constructor names the following operations can be used (Figure 3 contains a complete list).

```
data Con -- abstract
arity   :: Con -> Int       -- primitive
showCon :: Con -> String    -- primitive
```

Building upon `arity` and `showCon` we can already implement a simple, generic version of Haskell's `showsPrec` function. Recall that `showsPrec(t) d x` takes a precedence level `d` (a value from 0 to 10), a value `x` of type `t` and returns a `String`-to-`String` function. The following equations define `showsPrec(t)` for all types that do not involve any primitive types.

$$
\begin{array}{lcl}
\mathcal{D}[\![\texttt{data}\ T\ a_1 \ldots a_m = c_1 \mid \cdots \mid c_n]\!] & = & T\ a_1 \ldots a_m = \Sigma(\mathcal{K}[\![c_1]\!] \ldots \mathcal{K}[\![c_n]\!]) \\
\mathcal{D}[\![\texttt{newdata}\ T\ a_1 \ldots a_m = c]\!] & = & T\ a_1 \ldots a_m = \mathcal{K}[\![c]\!] \\
\mathcal{K}[\![K\ t_1 \ldots t_k]\!] & = & K\ (\Pi(\mathcal{T}[\![t_1]\!] \ldots \mathcal{T}[\![t_k]\!])) \\
\mathcal{K}[\![K\ \{\ell_1 :: t_1, \ldots, \ell_k :: t_k\}]\!] & = & K\ (\Pi((\ell_1 :: \mathcal{T}[\![t_1]\!]) \ldots (\ell_k :: \mathcal{T}[\![t_k]\!]))) \\
\mathcal{T}[\![!\ t]\!] & = & t \\
\mathcal{T}[\![t]\!] & = & t
\end{array}
$$

Figure 1: Translation of datatype declarations and datatype renamings

```
showsPrec(t)                :: Int -> t -> ShowS
showsPrec(a+b) d (Left x)   =  showsPrec(a) d x
showsPrec(a+b) d (Right y)  =  showsPrec(b) d y
showsPrec(c a) d x
    | arity c == 0          =  showString (showCon c)
    | otherwise             =  showParen (d >= 10)
                               (  showString (showCon c)
                               .  showChar ' '
                               .  showsPrec(a) 10 x )
showsPrec(a*b) d (x, y)     =  showsPrec(a) d x
                               .  showChar ' '
                               .  showsPrec(b) d y
```

The first two equations discard the binary constructors `Left` and `Right`. They are not required since the constructor names are accessible via the type pattern `c a`. If the constructor is nullary, its string representation is emitted. Otherwise, the constructor name is printed followed by a space followed by the representation of its arguments. If the precedence level is 10, the output is additionally parenthesized. The last equation applies if a constructor has more than one component. In this case the components are separated by a space. Note that this definition is a simplified version of `showsPrec(t)` as it neither considers infix operators nor record syntax. A full-blown version appears in Appendix B.

It should be noted that constructor names appear only on the type level; they have no counterpart on the value level as value constructors are encoded using `Left` and `Right`. If a generic definition does not include a case for the type pattern `c a`, then we tacitly assume that `g(c a) = g(a)`.

Haskell allows the programmer to assign labels to the components of a constructor. The names of the labels can be accessed via the type pattern `l::a` where `l` is a *value* variable of type `Label` and `a` is a *type* variable. If a generic definition does not include a case for this pattern, we assume that `g(l::a) = g(a)`. The full-blown version of `showsPrec(t)` uses the type pattern `l::a`.

2.4 Ad-hoc definitions

A generic function solves a problem in a uniform way for all types. Sometimes it is desirable to use a different approach for some datatypes. Consider, for instance, the function `encode` instantiated to lists over some base type. To encode the structure of an n-element list $n+1$ bits are used. For large lists this is clearly wasteful. A more space-efficient scheme stores the length of the list in a header followed by the encodings of the elements. We can specify this compression scheme for lists using a so-called ad-hoc definition.

```
encode([a]) xs =  encodeInt (length xs)
               ++ concatMap (encode(a)) xs
```

Ad-hoc definitions specify exceptions to the general rule and may be given for all predefined and for all user-defined datatypes. As before, the type pattern on the left-hand side must have the form $\texttt{T}\ \texttt{a}_1 \ldots \texttt{a}_k$, ie a type constructor is applied to type variables $\texttt{a}_1, \ldots, \texttt{a}_k$, which have to be distinct. Furthermore, `T` must be a type constructor of first-order kind. Recall that the kind system of Haskell specifies the 'type' of a type constructor [14]. The '`*`' kind represents nullary constructors like `Char` or `Int`. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The order of a kind is given by $\mathit{order}(*) = 0$ and $\mathit{order}(\kappa_1 \to \kappa_2) = \max\{1 + \mathit{order}(\kappa_1), \mathit{order}(\kappa_2)\}$. Whether the restriction to first-order kinds is severe remains to be seen.

Ad-hoc definitions can be spread over several modules. Generic definitions are handled very much like class- and instance-declarations. The type signature of a generic definition together with the equations for `1`, '`+`', and '`*`' plays the rôle of a class definition. Ad-hoc definitions are akin to instance declarations. Like instance declarations they cannot be explicitly named on import or export lists [19, Section 5.4]. Rather, they are always implicitly exported.

We have remarked in the introduction that `1`, '`+`', and '`*`' serve as abbreviations for the predefined types `()`, `Either`, and `(,)`. This is, however, not the whole truth. We use two different sets of constructor names to be able to distinguish between generic and ad-hoc equations. For instance, `showsPrec((a,b))` given by

```
showsPrec((a,b)) d (x, y) =  showChar '('
                          .  showsPrec(a) 0 x
                          .  showChar ','
                          .  showsPrec(b) 0 y
                          .  showChar ')'
```

is an ad-hoc definition, which determines the layout of pairs. By contrast, `showsPrec(a*b)` as defined in Section 2.3 is a generic definition specifying the layout of constructor components.

2.5 Mutually recursive definitions

The generic definitions we have seen so far involve only a single generic value. An obvious generalization is to allow for mutually recursive definitions. As before we require that the definitions are inductive on the structure of type expressions. Formally, a mutually recursive declaration group $\texttt{g}_1, \ldots, \texttt{g}_n$ is well-formed if there is a partial order on the $\texttt{g}_i$ such that for every recursive call

$$\texttt{g}_i(\texttt{T}\ \texttt{a}_1\ \ldots\ \texttt{a}_k) \quad = \quad \ldots\ \texttt{g}_j(t)\ \ldots$$

either (1) $t \in \{a_1, \ldots, a_k\}$ or (2) $g_i > g_j$, $t = T\ b_1\ \ldots\ b_k$, and $\{b_1, \ldots, b_k\} \subseteq \{a_1, \ldots, a_k\}$. Examples for generic definitions that employ this general form can be found in Appendix B.

Generic definitions may only appear in the top-level declaration list of a module. In principle, we could also permit generic definitions in local declaration groups. This extension is, however, hindered by Haskell's lack of scoped type variables. Recall that type signatures are mandatory for generic definitions. The lack of explicit scopes makes it impossible to distinguish between monomorphic and polymorphic uses of type variables. A suitable extension is described in [17].

2.6 Values indexed by first-order type constructors

Functions may not only be indexed by types of kind $*$ but also by type constructors of first-order kind. The archetypical example for such a function is `size(t) :: t a -> Int`, which counts the number of values of type `a` in a given structure of type `t a`. Note that the type parameter of `size` ranges over types of kind $* \to *$. To be able to define `size` generically for all type constructors of this kind we must extend the type language by type abstractions. We agree upon that `\a.t` is syntax for λ-abstractions on the type level. (We use `\a.t` instead of `\a->t` since '`->`' is also used as a type constructor and the type expression `\a->f a->g a` looks rather ugly.) For example, `\a.a` is the identity type and `\a.Char` is the constant type, which maps arbitrary types to `Char`. Using type abstractions the definition of `size` reads

```
size(t)                    :: t a -> Int
size(\a.a)        x        =  1
size(\a.1)        x        =  0
size(\a.Char)     x        =  0
size(\a.Int)      x        =  0
size(\a.f a+g a)  (Left x)  =  size(f) x
size(\a.f a+g a)  (Right y) =  size(g) y
size(\a.f a*g a)  (x, y)    =  size(f) x
                           +  size(g) y
```

To understand the equations it is helpful to replace the formal type parameter `t` in the type signature by the respective type arguments. For the first equation we obtain the type `a->Int`. Clearly, a structure of type `a` contains one element of type `a`. Substituting `\a.1` for `t` yields `1->Int`, ie we have no elements of type `a`. For sums the substitution yields `f a+g a->Int`. To determine the size of an element of type `f a+g a` we must either calculate the size of a structure of type `f a` or that of a structure of type `g a`. Finally, the size of a structure of type `f a*g a` is given by the sum of the size of the two components.

The definition of `size` employs two sorts of type patterns: the identity type `\a.a`, which may be abbreviated by `Id`, and patterns of the form `\a.T (f1 a) ... (fk a)` where the f_i are type variables of kind $* \to *$. To be exhaustive a generic definition must include cases for `Id`, `1`, '`+`', and '`*`' and additionally for all primitive types. Now, reconsider the definition of `size` and note that the cases for constant types of the form `\a.T` are all alike. We allow the programmer to combine these cases by using the type pattern `\a.u`, which may be abbreviated to `K u`. Using this shortcut the definition of `size` can be rewritten to

```
size(t)                    :: t a -> Int
size(Id)  x                =  1
size(K u) x                =  0
size(\a.f a+g a) (Left x)  =  size(f) x
size(\a.f a+g a) (Right y) =  size(g) y
size(\a.f a*g a) (x, y)    =  size(f) x
                           +  size(g) y
```

Applying a generic value works as before: the value is called with a closed type expression. The type may also involve type abstractions. For example, `size(\a->[[a]])` counts the number of elements in a list of lists.

Let us stress that type arguments cannot be unambiguously inferred by the compiler—this is the principle reason why they are mandatory. Consider, for instance, the call `size xs` where `xs` has type `[[Int]]`. To determine the type argument of `size` we must solve the equation `t u = [[Int]]`. The higher-order unification of `t u` and `[[Int]]` yields, however, three different solutions: `t` = `\a.[a]` and `u` = `[Int]`, `t` = `\a->[[a]]` and `u` = `Int`, or `t` = `\a->[[Int]]` and `u` arbitrary. To prevent ambiguity type arguments must never be omitted. Alternatively, one could use Haskell's kinded first-order unification [14], which produces unique solutions (the first one in the example above). Some practical experience is needed to decide whether this is a viable alternative.

A value may be indexed by an arbitrary first-order type constructor. In the general case, the type patterns are either projection patterns or constructor patterns. If the type parameter has n arguments, we have n different projection patterns: `\a1...an.a1`, ..., `\a1...an.an`. Constructor patterns have the general form `\a1...an.T (f1 a1...an) ... (fk a1...an)`. Note that ad-hoc definitions employ constructor patterns, as well. For instance, an ad-hoc definition for the list type has the form `g(\a1...an.[f a1...an]) = ...g(f)...`.

3 Blending generic definitions with type classes

Type classes and generic definitions are closely related concepts. This section is concerned with the integration of the two features. In particular, we show how to specify derivable class methods by generic definitions. Beforehand, let us briefly comment on the differences between type classes and generic definitions. A type class corresponds roughly to the type signature of a generic definition—or rather, to a collection of type signatures. Instance declarations are related to ad-hoc definitions. Thus, class and instance declarations can be mimicked by generic definitions. Since generic definitions furthermore allow instances to be defined in a uniform way, they are, in fact, more general than type classes. On the negative side, the user must always supply explicit type arguments. We have seen in Section 2.6 that it is not possible to infer the type arguments of generic calls. By contrast, class methods can, in general, be called without giving explicit type information.

Now, integrating generic definitions with type classes we (hope to) get the best of both worlds. To this end we allow default class methods to be specified by generic definitions. That way a class method can be defined generically for all instances making it possible to *derive* instances for newly defined datatypes. Take, for example, Haskell's `Eq` class, which is one of the few derivable classes. Here is a generic definition of `Eq`.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

```
(==)(1)    ()         ()         =  True
(==)(a+b)  (Left  v)  (Left  x)  =  (==)(a) v x
(==)(a+b)  (Left  v)  (Right y)  =  False
(==)(a+b)  (Right w)  (Left  x)  =  False
(==)(a+b)  (Right w)  (Right y)  =  (==)(b) w y
(==)(a*b)  (v, w)     (x, y)     =  (==)(a) v x
                                  && (==)(b) w y
x /= y     =  not (x==y)
```

Given this declaration instances may be derived for arbitrary types of kind $*$ and for arbitrary type constructors of first-order kind—only subsidiary primitive types must be instances of `Eq`. For type constructors of first-order kind the derived instance declaration has the form `(Eq` a_1`,...,Eq` a_k`) => Eq (T` $a_1 \ldots a_k$`)`. It is important to note that instances can be derived no matter of the internal structure of the types. For example, the following definition

```
data Matrix a = Matrix (Twice [] a) deriving (Eq)
```

which uses the second-order type constructor `Twice` defined in the introduction, is perfectly legal. Currently, this definition is rejected.

Generic default methods can also be specified for classes that abstract over first-order type constructors. Haskell's `Functor` class serves as an excellent example. The `Functor` class consists of the so-called mapping function `map(t)`, which applies a given function to each element of type `a` in a given structure of type `t a`.[1] It is well-known that `map(t)` can be defined for all type constructors of kind $* \to *$ that do not involve the function space. Alas, currently the user must program instances of `Functor` by hand (which is for the most part tedious but sometimes quite involving). Again, generic default methods save the day.

```
class Functor t where
  map :: (a -> b) -> (t a -> t b)

  map(Id)          h x          =  h x
  map(K u)         h x          =  x
  map(\a.f a+g a) h (Left x)  =  Left (map(f) h x)
  map(\a.f a+g a) h (Right y) =  Right(map(g) h y)
  map(\a.f a*g a) h (x, y)    =  (map(f) h x,
                                  map(g) h y)
```

Now, instances of `Functor` can be derived for arbitrary type constructors of kind $* \to *$ (using `deriving`). Note that Appendix B defines a slightly more general version that also takes the function space into account.

3.1 Limits of type classes

We have noted in the introduction that instances cannot be derived for all datatypes in Haskell 98. In this section we take a brief look at the reasons for this failure. Consider the following datatype declarations taken from [6] (note that `MapF` is isomorphic to `Twice`).

```
data MapF m v =  MapF (m (m v))
data MapS m v =  MapS v (MapS (MapF m) v)
                       (m (MapS (MapF m) v))
```

Both `MapF` and `MapS` are second-order type constructors of kind $(* \to *) \to (* \to *)$. The attempt to derive an instance of a predefined class, say, `Eq` fails in both cases. Recall that instance declarations for `Eq` have essentially the following form (the context may additionally refer to other classes).

`instance (Eq` a_1`,...,Eq` a_k`) => Eq (T` $a_1 \ldots a_k$`)`

Clearly, this form is too limited to handle `MapF` or `MapS`. In [20] an extension of the class system is described, which allows for more general instance contexts. Using this extension, which has been realized in GHC [18] and in Hugs 98 [15], the following `Eq` instance can be derived for `MapF` (do you guess how `(==)` is implemented?).

```
instance (Eq (m (m v))) => Eq (MapF m v)
```

However, if we try to derive `Eq` for `MapS`, both compilers loop! Even if the generalized form of instance contexts is used, the type constraints cannot be finitely represented. This example demonstrates the limits of Haskell's class system.

On the other hand, generic definitions *can* be specialized for higher-order type constructors such as `MapF` and `MapS`. Let us sketch the main idea in the rest of this section—the technique is described at length in [7]. Let `EqD a` be the dictionary type corresponding to the `Eq` class. First of all, note that an instance declaration of the form

```
instance (Eq a) => Eq (T a)
```

defines a function of type `∀a.EqD a -> EqD (T a)`, ie the dictionary for `a` is mapped to the dictionary for `T a`. Here `T` is a type constructor of kind $* \to *$. Consequently, an instance for a type constructor `H` of kind $(* \to *) \to (* \to *)$ should be a function that maps $* \to *$ instances to $* \to *$ instances.

```
∀f.(∀a.(EqD a -> EqD (f a)))
   -> (∀b.(EqD b -> EqD (H f a)))
```

This scheme can be easily generalized to type constructors of even higher-order kinds. So we have the unfortunate situation that `Eq` instances can, in principle, be derived for arbitrary type constructors but these instances cannot be represented as Haskell instance declarations. This explains, in particular, why `deriving` is restricted to type constructors of kind $*$ or of first-order kind though the type may internally depend on types of higher-order kinds.

4 Examples

This section presents further examples for generic functions. The first example, `decode(t)`, is essentially the inverse of `encode(t)`: it takes a bit string, decodes a prefix of that bit string, and returns the decoded value coupled with the unused suffix. The two functions are related by `decode(t) (encode(t) x ++ bs) = (x, bs)`.

```
decode(t)              :: [Bit] -> (t, [Bit])
decode(1) bs           =  ((), bs)
decode(Int) bs         =  decodeInt bs
decode(Char) bs        =  decodeChar bs
decode(a+b) []         =  error "decode"
decode(a+b) (0 : bs)   =  appl Left  (decode(a) bs)
decode(a+b) (1 : bs)   =  appl Right (decode(b) bs)
decode(a*b) bs         =  let (x, cs) = decode(a) bs
                              (y, ds) = decode(b) cs
                          in  ((x, y), ds)
appl f (x, y)          =  (f x, y)
```

[1] In Haskell 98 `map` is called `fmap`.

The ad-hoc definition for lists, `decode([a])`, is left as an instructive exercise to the reader.

The Standard Prelude contains a number of list processing functions, which can be generalized to arbitrary type constructors: `and`, `or`, `all`, `any`, `sum`, `product` etc. They are all instances of a more general concept termed reduction or crush [16]. A crush is a function of type `t a -> a` that collapses a structure of values of type `a` into a single value of type `a`. To define a crush we require two ingredients: a value `e :: a` and a binary operation `h :: a->a->a`. Usually but not necessarily `e` is the neutral element of `h`.

```
crush(t) :: a -> (a -> a -> a) -> (t a -> a)
crush(Id)        e h x          =   x
crush(K u)       e h x          =   e
crush(\a.f a+g a)e h (Left x) =   crush(f) e h x
crush(\a.f a+g a)e h (Right y)=   crush(g) e h y
crush(\a.f a*g a)e h (x, y)   =   crush(f) e h x
                                  `h` crush(g) e h y
```

Using `crush(t)` we can define, for instance, generic versions of `and` and `sum`.

```
and(t)   :: t Bool -> Bool
and(t)   =  crush(t) True (&&)
sum(t)   :: (Num n) => t n -> n
sum(t)   =  crush(t) 0 (+)
```

Further examples can be found in Appendix B or in [16, 10].

The function `sum(t)` shows that type signatures of generic values may also involve class constraints. The following definition, which implements a monadic mapping function [3], falls back on the `Monad` class. A monadic map applies a function of type `a->m b` to all elements of type `a` in a given structure of type `t a` and then threads the monad `m` from left to right through the resulting structure.

```
mapM(t) :: Monad m => (a->m b) -> (t a->m (t b))
mapM(Id)  h x = h x
mapM(K u) h x = return x
mapM(\a.f a+g a) h (Left x)
    = do { v <- mapM(f) h x; return(Left v) }
mapM(\a.f a+g a) h (Right y)
    = do { w <- mapM(g) h y; return(Right w) }
mapM(\a.f a*g a) h (x, y)
    = do { v <- mapM(f) h x; w <- mapM(g) h y;
           return (v, w) }
```

Note that `mapM` generalizes the function of the same name defined in the Standard Prelude (the Prelude function is specialized to lists). An important special case of `mapM` is

```
thread(t)   :: (Monad m) => t (m a) -> m (t a)
thread(t)   =  mapM(t) id
```

The function `thread(t)` essentially commutes the type constructors `t` and `m`. It generalizes the Prelude function `sequence`, which operates on lists.

5 Related work

The concept of generic functional programming trades under a variety of names: F. Ruehr refers to this concept as *structural polymorphism* [22, 21], T. Sheard calls generic functions *type parametric* [23], C.B. Jay and J.R.B. Cocket use the term *shape polymorphism* [12], R. Harper and G. Morrisett [5] coined the phrase *intensional polymorphism*, and J. Jeuring invented the word *polytypism* [13].

The mainstream of generic programming is based on the initial algebra semantics of datatypes [4] and typically requires a basic knowledge of category theory. The categorical programming language Charity [1] automatically provides `map` and catamorphisms for each user-defined datatype. Functorial ML [11] has a similar functionality but a different background. It is based on the theory of *shape polymorphism*, in which values are separated into shape and contents. The polytypic programming language extension PolyP [9] offers a special construct for defining generic functions. The generic definitions are similar to the ones given in this paper (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion (see [8] for a detailed comparison). All the approaches with the notable exception of [22] are restricted to first-order kinded, regular datatypes (or even subsets of this class), that is they cover only a small part of Haskell's type system.

The theoretical background and a possible implementation of the generic programming extension presented here are described in a companion paper [7], which contains supplementary pointers to relevant work. We show, in particular, how to specialize generic definitions for given instances of datatypes. The specialization employs a generalization of the dictionary passing translation and does not require the passing of types or representations of types at run-time. This is in contrast to the work on *intensional polymorphism* [5, 2] where a **typecase** is used for defining type-dependent operations.

6 Conclusion

This paper proposes a generic programming extension for Haskell. We feel that the extension fits quite smoothly into the language. In particular, it generalizes the `deriving` construct and makes derivable class methods explicit. The examples given in the paper furthermore demonstrate the usefulness of such an extension.

Acknowledgements

I would like to thank Phil Wadler and four anonymous referees for many helpful comments.

A Changes to the syntax

Figure 2 lists the necessary changes to the syntax. The syntax of type expressions has been extended to include type abstractions, '`+`', and '`*`'. Optional type arguments have been added to type signatures, to the left-hand sides of function definitions, and to value expressions.

B Changes to the Standard Prelude and to the Standard Libraries

The Standard Prelude is extended by a module called `Generics`, see Figure 3, which defines `Con`, `Label`, and a few generic values. Figures 4, 5, 6, and 7 contain the modifications to the Standard Prelude. Note that all derivable classes (`Eq`, `Ord`, `Enum`, `Bounded`, `Show`) with the exception of `Read` are precisely defined through generic default methods. The definition of `Read` is omitted for reasons of space. Finally, Figure 8 lists the changes to the Standard Library `Ix`.

gendecl	→	*gvars* `::` [*context* `=>`] *type*	(type signature)
	|	...	
gvars	→	$gvar_1$ `,` ... `,` $gvar_n$	($n \geqslant 1$)
gvar	→	*var*	
	|	*var tyvar*	(type-indexed value)
type	→	$type^0$	
$type^0$	→	$type^1$ [`->` $type^0$]	(function type)
$type^1$	→	$type^2$ [`+` $type^1$]	(binary sums)
$type^2$	→	$type^3$ [`*` $type^2$]	(binary products)
$type^3$	→	`\` $tyvar_1$ ... $tyvar_n$ `.` *type*	(type abstraction, $n \geqslant 1$)
	|	*var atype*	(constructor type pattern)
	|	*var* `::` *btype*	(label type pattern)
	|	*btype*	(type application)
gtycon	→	*qtycon*	
	|	`()`	(unit type)
	|	`1`	(unit type)
	|	`[]`	(list constructor)
	|	`(->)`	(function constructor)
	|	`(+)`	(binary sum constructor)
	|	`(*)`	(binary product constructor)
	|	`(,{,})`	(tupling constructors)
funlhs	→	*var atype* { *apat* }	(type-indexed value)
	|	...	
fexp	→	*qvar atype* { *aexp* }	(application of a type-indexed value)
	|	...	

Figure 2: Changes to the syntax

```
module Generics(
    Con, arity, showCon, hasLabels, Fixity(Nonfix, Infix, Infixl, Infixr, prec), fixity,
    precedence, Label, showLabel, card, crush, gsum, gproduct, size, gand, gor, gall, gany,
    gelem, gnotElem, listify ) where

-- abstract types and primitives

data Con  -- abstract
arity     :: Con -> Int     -- primitive
showCon   :: Con -> String  -- primitive
hasLabels :: Con -> Bool    -- primitive
instance Eq Con ...
instance Show Con where show = showCon

data Fixity = Nonfix | Infix {prec :: Int} | Infixl {prec :: Int} | Infixr {prec :: Int}
              deriving (Eq, Ord, Read, Show)
fixity       :: Con -> Fixity  -- primitive
precedence   :: Con -> Int
precedence c =  prec (fixity c)

data Label  -- abstract
showLabel :: Label -> String  -- primitive
instance Eq Label ...
instance Show Label where show = showLabel

-- generic definitions

card(t)    :: int     -- NB: card(t) is undefined for infinite types
card(1)    =  1
card(Char) =  ...     -- system dependent
card(Int)  =  ...     -- system dependent
card(a+b)  =  card(a) + card(b)
card(a*b)  =  card(a) * card(b)
card(a->b) =  card(b) ^ card(a)

crush(t)                          :: a -> (a -> a -> a) -> (t a -> a)
crush(Id)         e op x          =  x
crush(K u)        e op x          =  e
crush(\a.f a+g a) e op (Left x)   =  crush(f) e op x
crush(\a.f a+g a) e op (Right y)  =  crush(g) e op y
crush(\a.f a*g a) e op (x, y)     =  crush(f) e op x `op` crush(g) e op y

gsum(t), gproduct(t)     :: (Num n) => t n -> n
gsum(t)                  =  crush(t) 0 (+)
gproduct(t)              =  crush(t) 1 (*)
size(t)                  :: (Num n) => t a -> n
size(t)                  =  gsum(t) . map(t) (const 1)
gand(t), gor(t)          :: t Bool -> Bool
gand(t)                  =  crush(t) True (&&)
gor(t)                   =  crush(t) False (||)
gall(t), gany(t)         :: (a -> Bool) -> (t a -> Bool)
gall(t) p                =  gand(t) . map(t) p
gany(t) p                =  gor(t)  . map(t) p
gelem(t), gnotElem(t)    :: (Eq a) => a -> t a -> Bool
gelem(t) x               =  gany(t) (== x)
gnotElem(t) x            =  gall(t) (/= x)
listify(t)               :: t a -> [a]
listify(t) x             =  crush(t) id (.) (map(t) (:) x) []
```

Figure 3: Module `Generics`

```
module Prelude
...

import Generics

-- Equality and Ordered classes

class  Eq t  where
  (==), (/=) :: t -> t -> Bool

  (==) (t)                          :: t -> t -> Bool
  (==) (1)   ()         ()          =  True
  (==) (a+b) (Left  x1) (Left  y1)  =  (==) (a) x1 y1
  (==) (a+b) (Left  x1) (Right y2)  =  False
  (==) (a+b) (Right x2) (Left  y1)  =  False
  (==) (a+b) (Right x2) (Right y2)  =  (==) (b) x2 y2
  (==) (a*b) (x1, x2)   (y1, y2)    =  (==) (a) x1 y1 && (==) (b) x2 y2

  x /= y     =  not (x==y)

class  (Eq t) => Ord t  where
  compare              :: t -> t -> Ordering
  (<), (<=), (>=), (>) :: t -> t -> Bool
  max, min             :: t -> t -> t

  compare(1)   ()         ()          =  EQ
  compare(a+b) (Left  x1) (Left  y1)  =  compare(a) x1 y1
  compare(a+b) (Left  x1) (Right y2)  =  LT
  compare(a+b) (Right x2) (Left  y1)  =  GT
  compare(a+b) (Right x2) (Right y2)  =  compare(b) x2 y2
  compare(a*b) (x1, x2)   (y1, y2)    =  compare(a) x1 y1 `lex` compare(b) x2 y2
    where lex LT rel =  LT
          lex EQ rel =  rel
          lex GT rel =  GT
  ...

-- Enumeration and Bounded classes

class  Enum t  where
  succ, pred      :: t -> t
  toEnum          :: Int -> t
  fromEnum        :: t -> Int
  enumFromTo      :: a -> a -> [a]        -- [n..m]
  ...

  toEnum(1) n
    | n == 0      =  ()
    | otherwise   =  error "toEnum: Out of range"
  toEnum(a+b) n
    | n < card(a) =  Left  (toEnum(a) n)
    | otherwise   =  Right (toEnum(b) (n - card(a)))
  toEnum(a*b) n   =  (toEnum(a) r, toEnum(b) q)
    where (q, r)  =  divMod n (card(a))

  fromEnum(1) ()          =  0
  fromEnum(a+b) (Left x)  =  fromEnum(a) x
  fromEnum(a+b) (Right y) =  card(a) + fromEnum(b) y
  fromEnum(a*b) (x, y)    =  fromEnum(a) x + card(a) * fromEnum(b) y

  enumFromTo(a) x y =  map (toEnum(a)) [fromEnum(a) x .. fromEnum(a) y]
  ...
```

Figure 4: Changes to `Prelude`

```
class  Bounded t  where
  minBound      :: t
  maxBound      :: t

  minBound(1)   =  ()
  minBound(a+b) =  Left (minBound(a))
  minBound(a*b) =  (minBound(a), minBound(b))

  maxBound(1)   =  ()
  maxBound(a+b) =  Right (maxBound(b))
  maxBound(a*b) =  (maxBound(a), maxBound(b))

-- Monadic classes

class Functor t where
  map   :: (a -> b) -> (t a -> t b)
  comap :: (a -> b) -> (t b -> t a)

  map(Id)          h x         =  h x
  map(K u)         h x         =  x
  map(\a.f a+g a)  h (Left x)  =  Left  (map(f) h x)
  map(\a.f a+g a)  h (Right y) =  Right (map(g) h y)
  map(\a.f a*g a)  h (x, y)    =  (map(f) h x, map(g) h y)
  map(\a.f a->g a) h phi       =  map(g) h . phi . comap(f) h

  comap(K u)         h x         =  x
  comap(\a.f a+g a)  h (Left x)  =  Left  (comap(f) h x)
  comap(\a.f a+g a)  h (Right y) =  Right (comap(g) h y)
  comap(\a.f a*g a)  h (x, y)    =  (comap(f) h x, comap(g) h y)
  comap(\a.f a->g a) h phi       =  comap(g) h . phi . map(f) h
...

data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show, Functor)
...
```

Figure 5: Changes to `Prelude` (continued)

```
module PreludeList
...
import Generics

...
sum, product     :: (Num a) => [a] -> a
sum              =  gsum([])
product          =  gproduct([])
length           :: [a] -> Int
length           =  size([])
and, or          :: [Bool] -> Bool
and              =  gand([])
or               =  gor([])
any, all         :: (a -> Bool) -> [a] -> Bool
any              =  gany([])
all              =  gall([])
elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem             =  gelem([])
notElem          =  gnotElem([])
...
```

Figure 6: Changes to `PreludeList`

```
module PreludeText
...
import Generics

class  Show t  where
  showsPrec        :: Int -> t -> ShowS
  show             :: t -> String
  showList         :: [t] -> ShowS
  ...
  showsPrec(a+b)  d (Left x)  =  showsPrec(a) d x
  showsPrec(a+b)  d (Right y) =  showsPrec(b) d y
  showsPrec(a*b)  d (x, y)    =  showsPrec(a) d x . showChar ' ' . showsPrec(b) d y
  showsPrec(a->b) d phi       =  showString "<function>"
  showsPrec(c a)  d x
    | hasLabels c             =  shows c . showChar '{' . showsRecord(a) x . showChar '}'
    | arity c == 0            =  shows c
    | fixity c /= Nonfix      =  showsInfix(a) d c x
    | otherwise               =  showParen (d >= 10) (shows c . showChar ' ' . showsPrec(a) 10 x)

  showsRecord(1)    ()        =  showString ""
  showsRecord(a+b)  x         =  showsPrec(a+b) 0 x
  showsRecord(a*b)  (x, y)    =  showsRecord(a) x . showChar ',' . showsRecord(b) y
  showsRecord(a->b) phi       =  showsPrec(a->b) 0 phi
  showsRecord(c a)  x         =  showsPrec(c a) 0 x
  showsRecord(l::a) x         =  shows l . showChar '=' . showsPrec(a) 0 x

  showsInfix(a*b) d c (x, y)  =  showParen (d > precedence c) (showi (fixity c))
    where op                  =  showString (" " ++ showCon c ++ " ")
          showi (Infix  p)    =  showsPrec(a) (p+1) x . op . showsPrec(b) (p+1) y
          showi (Infixl p)    =  showsPrec(a) p     x . op . showsPrec(b) (p+1) y
          showi (Infixr p)    =  showsPrec(a) (p+1) x . op . showsPrec(b) p     y

class  Read t  --- analogous
```

Figure 7: Changes to `PreludeText`

```
module Ix
...
class  (Ord t) => Ix t  where
  range               :: (t,t) -> [t]
  index               :: (t,t) -> t -> Int
  inRange             :: (t,t) -> t -> Bool
  rangeSize           :: (t,t) -> Int

  range(1)   (l,h)           =  [()]
  range(a+b) (l,h)           =  enumFromTo(a+b) l h
  range(a*b) ((l,l'),(h,h')) =  [(i,i') | i <- range (l,h), i' <- range (l',h')]

  index(1)     (l,h) i =  0
  index(a+b) b@(l,h) i
    | inRange(a+b) b i =  fromEnum(a+b) i - fromEnum(a+b) l
    | otherwise        =  error "Ix.index: Index out of range."
  index(a*b) ((l,l'),(h,h')) (i,i')
                       =  index(a) (l,h) i * rangeSize(b) (l',h') + index(b) (l',h') i'

  inRange(1)   (l,h)           i      =  True
  inRange(a+b) (l,h)           i      =  compare(a+b) l i /= GT && compare(a+b) i h /= GT
  inRange(a*b) ((l,l'),(h,h')) (i,i') =  inRange(a) (l,h) i && inRange(b) (l',h') i'

  rangeSize(a) b@(l,h) | null (range(a) b) = 0
                       | otherwise         = index(a) b h + 1
```

Figure 8: Changes to `Ix`

References

[1] Robin Cockett and Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.

[2] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, 1999.

[3] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Informatica 94-28, University of Twente, June 1994.

[4] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.

[5] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, San Francisco, California*, pages 130–141. ACM-Press, 1995.

[6] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 1999. Accepted for publication.

[7] Ralf Hinze. A new approach to generic functional programming. Technical Report IAI-TR-99-9, Institut für Informatik III, Universität Bonn, July 1999.

[8] Ralf Hinze. Polytypic programming with ease (extended abstract). In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Lecture Notes in Computer Science. Springer-Verlag, November 1999. To appear.

[9] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM-Press, January 1997.

[10] Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden.* Department of Computing Science, Chalmers University of Technology and Göteborg University, June 1998.

[11] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[12] C.B. Jay and J.R.B. Cocket. Shapely types and shape polymorphism. In D. Sanella, editor, *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316, Berlin, 11–13 April 1994. Springer-Verlag.

[13] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[14] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[15] M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.

[16] Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.

[17] Erik Meijer and Koen Claessen. The design and implementation of Mondrian. In *Proceedings of the Haskell Workshop*, 1997.

[18] Simon Peyton Jones. Explicit quantification in Haskell, 1998. Available from `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.

[19] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.

[20] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.

[21] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.* Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[22] Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan, 1992.

[23] Tim Sheard. Type parametric programming. Technical report, Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.

Restricted Datatypes in Haskell

John Hughes (Chalmers University, Sweden)

Restricted Data Types in Haskell

John Hughes

September 4, 1999

Abstract

The implementations of abstract type constructors must often restrict the type parameters: for example, one implementation of sets may require equality on the element type, while another implementation requires an ordering. Haskell has no mechanism to abstract over such restrictions, which can hinder us from replacing one implementation by another, or making several implementations instances of the same class. This paper proposes a language extension called *restricted data types* to address this problem.

A restricted data type definition specifies a condition which argument types must satisfy for the data type to be *well-formed*. Every type in a program must be well-formed, and we add an explicit notation to express such requirements. Thus programmers can simply state that a type must be well-formed, rather than repeat its restriction explicitly.

We explain our extension via a simulation using multi-parameter classes, which serves to specify its semantics. We show its application to the design of a collection class and to the class of monads, and we discuss extensions to compile-time context reduction needed to implement it.

1 Introduction

Suppose you are designing an abstract data type of sets, represented for example by lists:

```
data Set a = Set [a]
```

When you implement the methods, you are likely to need to make assumptions about the element type `a`. For example, a function to test for membership will need to test elements for equality, and its type will reflect this:

```
member :: Eq a => a -> Set a -> Bool
```

Should you later decide to change the representation of sets, for example to use ordered binary trees for greater efficiency, then the restrictions on the element type will change. For example, the type of `member` will become

```
member :: Ord a => a -> Set a -> Bool
```

Thus a change in the *representation* of the abstract data type is reflected by a change in the *interface* which it provides.

Such a change in the interface has unfortunate consequences. When the functions which implement sets change their types, so in general will functions which use them in the rest of the program. Explicit type signatures spread throughout the program will therefore need to be changed, even though the definitions they are attached to are unaffected[1]. If there are many type signatures, either for stylistic

[1] Assuming, of course, that all the sets used in the program have elements that do actually support an ordering.

reasons or because Haskell's infamous monomorphism restriction forced their insertion, then the work of revising them may dominate that of modifying the `Set` module itself. In the worst case, the programmer may even be dissuaded from making a desirable change in one module, because of all the consequential changes that must be made to type signatures elsewhere.

An even more acute problem arises if we try to define the interface of an abstract data type as a Haskell class, so that several different implementations can be provided as instances. For example, we might wish to define a class `Collection` whose instances are lists, ordered binary trees, hash tables, etc.

```
class Collection c where
  ...
  member :: ... => a -> c a -> Bool
  ...
```

But now, what should the type of the `member` function be in the class definition? We cannot know what requirements to place on the type of collection elements, because these requirements differ from instance to instance. As a result, we cannot write an appropriate class definition at all!

The main idea of this paper is to restrict the parameters of abstract data types *when we define the type*, rather than when we define the methods. We thus define sets represented by lists as follows

```
data Eq a => Set a = Set [a]
```

with the interpretation that types `Set t` are well-formed only if `t` supports equality[2]. We call types defined in this way *restricted data types*. Now, since we state in the *type* definition that the elements must support equality, it should no longer be necessary to state it in the types of the *methods*. For example, we might now give the `member` function the type

```
member :: a -> Set a -> Bool
```

It is clear that `member` can only be used at types that support equality, because `Set a` occurs in its type signature.

Now, if the implementation of sets is changed to ordered binary trees, then the new constraint on the element type need appear only on the type definition; the types of the methods remain unchanged. Consequently the problems discussed in this section disappear: the implementations of abstract datatypes can be changed without affecting type signatures in the rest of the program, and different implementations of the same abstract datatype can be made instances of the same class, even if they place different restrictions on the type parameters.

While the basic idea of a restricted data type is very intuitive, the details of the design are surprisingly subtle, and the implementation is even subtler. We will therefore focus on *simulating* restricted data types in Haskell (extended with multi-parameter classes). This simulation has been tested using `hugs98` with the `-98` flag. However, the simulation is a little tedious to use in practice, and so at the end of the paper we will propose a language extension whose semantics (and a possible implementation) is given by the simulation we describe.

2 Simulating Restricted Data Types: A Collection Class

We shall explain our idea with reference to a simplified `Collection` class, which might be defined using restricted data types as follows:

[2]Haskell already supports this syntax, but with a much weaker meaning.

```
class Collection c where
  empty :: c a
  singleton :: a -> c a
  union :: c a -> c a -> c a
  member :: a -> c a -> Bool
```

We will show how to simulate restricted data types, so that both sets and ordered lists can be made instances of this class.

Of course, the intention of this class definition is that the element type `a` is implicitly constrained to satisfy the restriction of the data type `c`. To simulate this in Haskell without restricted data types, we must declare a `Collection` class whose methods *do* explicitly restrict the element type, but we must parameterise the class definition on the particular restriction concerned, so that different instances can impose different restrictions. If we could parameterise classes on *classes*, then we might write

```
class Collection c cxt where
  empty :: cxt a => c a
  singleton :: cxt a => a -> c a
  union :: cxt a => c a -> c a -> c a
  member :: cxt a => a -> c a -> Bool
```

and declare instances

```
instance Collection Set Eq where ...
instance Collection OrdSet Ord where ...
```

However, Haskell classes can only be parameterised on types. We therefore *represent* class constraints such as `Eq` and `Ord` by a suitable type. It is natural to represent a class by the type of its associated dictionary, and so (simplifying somewhat) we define

```
data EqD a = EqD {eq :: a -> a -> Bool}
data OrdD a = OrdD {le :: a -> a -> Bool, eqOrd :: EqD a}
```

The idea is that `EqD a` contains an implementation of the equality test, while `OrdD a` contains an implementation of `<=` and an equality dictionary (since `Eq` is a superclass of `Ord`).

Now, the constraint `Eq a` is satisfied when we have an implementation of equality available at type `a`, which is equivalent to having a value of type `EqD a` available. We therefore define a class

```
class Sat t where dict :: t
```

which we will use to simulate other constraints. For example, the constraint `Eq a` is simulated by `Sat (EqD a)`; the former is satisfied precisely when we can construct a dictionary to satisfy the latter. We declare

```
instance Eq a => Sat (EqD a) where
  dict = EqD {eq= (==)}
instance Ord a => Sat (OrdD a) where
  dict = OrdD {le= (<=), eqOrd= dict}
```

Now we can redefine the `member` function for sets so that it no longer explicitly refers to the `Eq` class, but instead just requires that an appropriate dictionary exists:

```
member :: Sat (EqD a) => a -> Set a -> Bool
member x (Set xs) = any (eq dict x) xs
```

```
data Set cxt a = Set [a] | Unused (cxt a->()) deriving Show
type SetCxt a = EqD a
type WfSet a = Set SetCxt a

instance Collection Set EqD where
  empty = Set []
  singleton x = Set [x]
  union xset@(Set xs) (Set ys) =
    Set (xs++[y | y<-ys, not (member y xset)])
  member x (Set xs) = any (eq dict x) xs
```

Figure 1: Making `Set` an instance of `Collection`.

Now, at last, we can parameterise the `Collection` class definition both on the type of collections, and on the constraint that elements must satisfy, since both are now represented by types. We might expect to write

```
class Collection c cxt where
  empty :: Sat (cxt a) => c a
  singleton :: Sat (cxt a) => a -> c a
  union :: Sat (cxt a) => c a -> c a -> c a
  member :: Sat (cxt a) => a -> c a -> Bool
```

thus making the appropriate dictionary available in all the methods.

Unfortunately this definition is still not quite right, because the class parameter `cxt` does not appear in the *types* of the methods, and so cannot be inferred when the methods are used. An attempt to use this class would therefore lead to ambiguous overloading. In fact, since we are simulating restricted data types, there is only one possible type `cxt` for each collection type `c`, but compilers cannot know this.

The solution is simply to parameterise collection types on their context, so that each type carries with it the restriction that its elements must satisfy. We rewrite the class definition as

```
class Collection c cxt where
  empty :: Sat (cxt a) => c cxt a
  singleton :: Sat (cxt a) => a -> c cxt a
  union :: Sat (cxt a) => c cxt a -> c cxt a -> c cxt a
  member :: Sat (cxt a) => a -> c cxt a -> Bool
```

and it is now accepted.

We modify the definition of the `Set` type accordingly, and we can then define it as an instance of the generic `Collection` class. The new type and instance definitions appear in figure 1[3]. A similar implementation of collections as ordered lists appears in figure 2.

[3]There is one unpleasant hack in the figure: the constructor `Unused` in the data type definition for `Set`. It is there purely in order to force the compiler to assign the parameter `cxt` the correct kind: without it, `cxt` does not appear at all in the right hand side of the definition, and is therefore assigned the (incorrect) kind `*`. The application `cxt a` forces the correct kind `*->*` to be assigned, and embedding it in the type `cxt a->()` prevents the type of the context from interfering with the derivation of a `Show` instance.

```
data OrdSet cxt a = OrdSet [a] | Unused (cxt a->()) deriving Show
type OrdSetCxt a = OrdD a
type WfOrdSet a = OrdSet OrdSetCxt a

instance Collection OrdSet OrdD where
  empty = OrdSet []
  singleton x = OrdSet [x]
  union (OrdSet xs) (OrdSet ys) = OrdSet (merge xs ys)
    where
      merge [] ys = ys
      merge xs [] = xs
      merge (x:xs) (y:ys) = if eq (eqOrd dict) x y then x:merge xs ys
                            else if le dict x y then x:merge xs (y:ys)
                            else y:merge (x:xs) ys
  member x (OrdSet xs) = any (eq (eqOrd dict) x) xs
```

Figure 2: Making `OrdSet` an instance of `Collection`.

3 Restricted Monads

Restricted data types would be of limited interest if they were useful only for defining collection types, but in fact they are generally useful in connection with constructor classes, classes whose instances are parameterised types. We will discuss one more example, the class of *monads*.

The `Monad` class is defined (slightly simplified) as follows:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

One interesting monad in the mathematical sense is the type `Set`, but our implementation of `Set` above cannot be made an instance of this class because `(>>=)` requires equality on the set elements, and the type given for `(>>=)` in the class declaration does not provide it. Categorically speaking, `Set` is a monad over a *subcategory* of the category of Haskell types and functions, but Haskell gives us no way to express that.

However, just as we parameterised the `Collection` class on a condition that elements must satisfy, so we can parameterise the `Monad` class in an analogous way. We define

```
class WfMonad m cxt where
  unit :: Sat (cxt a) => a -> m cxt a
  bind :: (Sat (cxt a), Sat (cxt b)) =>
            m cxt a -> (a -> m cxt b) -> m cxt b
```

Notice that the context for `bind` requires that *both* types `m cxt a` and `m cxt b` are well-formed.

Now we can indeed make `Set` an instance of `WfMonad`:

```
instance WfMonad Set EqD where
  unit a = Set [a]
  Set as `bind` f = foldr union empty (map f as)
```

The `union` operation in `bind` requires equality, which is available since the type `Set EqD b` is well-formed.

Another interesting example is the monad which represents computations as strings:

```
data StringM cxt a = StringM String | Unused (cxt a->())
```

Naturally, we must restrict this monad to types which can be read and shown, so we define a type to represent this constraint:

```
data TextD a = TextD {rd :: String -> a, sh :: a -> String}
instance (Read a, Show a) => Sat (TextD a) where
  dict = TextD {rd= read, sh= show}
```

Now we can make `StringM` an instance of `WfMonad` as follows:

```
instance WfMonad StringM TextD where
  unit x = StringM (sh dict x)
  StringM s `bind` f = f (rd dict s)
```

This monad is related to a library for CGI programming which I am developing, which saves computations in hidden fields of HTML forms. (In fact, the technique is applied to yet another constructor class, the class of *arrows* [Hug99]).

4 Improving the Simulation

The technique we have discussed certainly lets us use different properties of collection elements in different instances of the collection class, but not really very conveniently. In figure 2, for example, we know that the element type supports an equality operation, but we have to refer to it as `eq (eqOrd dict)`. Of course we would prefer to use the usual symbol `(==)`, not only because it is syntactically more appealing, but also because we could then use other overloaded functions that depend on equality internally. However, using the equality symbol would require a context `Eq a` rather than `Sat (OrdD a)`.

Let us therefore define new instances of `Eq` in terms of `Sat`, which extract the equality function from an available dictionary:

```
instance Sat (EqD a) => Eq a where
  (==) = eq dict

instance Sat (OrdD a) => Sat (EqD a) where
  dict = eqOrd dict
```

Given these definitions, and a similar one for `Ord`, we should be able to use `(==)` and `(<=)` freely in Figures 1 and 2.

Unfortunately, these instance declarations are rejected by Hugs, because they *overlap* with the existing instances of `Eq` and `Sat (EqD a)`. In their exploration of the design space for type classes [JJM97], Peyton-Jones, Jones and Meijer discuss the possibility of allowing overlapping instances, but conclude that it is undesirable to do so.

The main problem is that the meaning of a program may depend on which of two overlapping instances is chosen, and so it is important to specify precisely how the choice is made, but it seems to be very difficult to give a specification which is both simple and precise. However, in this particular case, the meanings of programs do *not* depend on which instance is chosen. There is only one implementation of equality for any particular type; the new overlapping instance declaration above simply provides another way of accessing it. Thus whenever there is a choice of how to obtain an implementation of equality, then we know that the choice does not

affect the semantics of the program, and the compiler is free to choose the instance declaration to apply on other grounds, such as efficiency.

However, even if overlapping instances are permitted, they do introduce a risk that type inference may loop. Indeed, Hugs provides a flag to allow instances to overlap, but if it is turned on then the type checker loops given the declarations above. To understand why, we must explain the process of *context reduction*. Suppose a programmer uses an equality test to compare two lists within a polymorphic function. The compiler infers that the comparison is well typed in a context `Eq [a]`, where `a` is the element type. Given an instance declaration `instance Eq a => Eq [a] where ...`, then the compiler can construct an implementation of equality for `[a]` from an implementation for `a`, thus reducing the problem of satisfying `Eq [a]` to the simpler problem of satisfying the context `Eq a`. This process is called context reduction, and is implemented by using instance declarations 'backwards' as rewrite rules on contexts.

Now we can see that if instance declarations overlap, then context reduction becomes non-deterministic. Worse, it may easily loop. In our example, the new instance declaration

```
instance Sat (EqD a) => Eq a
```

enables a context `Eq a` to be reduced to `Sat (EqD a)`, but the instance declaration we gave earlier

```
instance Eq a => Sat (EqD a)
```

enables this to be reduced back to `Eq a` again. Hence context reduction loops. In this case the compiler would need to search for a terminating context reduction, and this is something which existing implementations of overlapping instances do not do.

In general, it may be undecidable whether a path in the tree of possible context reductions is terminating or not. But in our particular case, loops are introduced only via instance declarations involving the `Sat` class, and by inspection, context reduction using these instances does not increase the *depth* of terms in the context. Thus even infinite context reductions using these instances will contain only a finite number of terms. A compiler can abort an attempted context reduction when a term reduced earlier appears again, since such a reduction is never helpful: to discover, for example, that an `Eq a` dictionary can be constructed from an `Eq a` dictionary is pointless. In our application, this strategy will cut off all infinite context reductions, and so type checking will terminate.

To summarise, to make our simulation of restricted datatypes convenient to use we must be able to define overlapping instance. This is normally dangerous, since in general it leads to ill-defined semantics and undecideable type checking, but in our particular application these problems do not arise. We do require, though, that compilers make an easy test to detect and avoid looping context reductions.

5 Other Approaches

The simulation we have described is certainly non-trivial, and of course, similar problems have been solved in other ways before. In this section we will review two other ways of designing a `Collection` class, which could be taken as alternative ways of simulating restricted data types, and we will argue that the approach we take in this paper is superior to both.

5.1 Peyton-Jones' Multiparameter `Collection` Class

Peyton-Jones proposed a different design for a multi-parameter `Collection` class [PJ96]. His idea was to parameterise the class on the type of elements, as well as the type of collections, thus letting instances constrain both. Applying his idea to our simplified class, we would define it as

```
class Collection c a where
  empty :: c a
  singleton :: a -> c a
  union :: c a -> c a -> c a
  member :: a -> c a -> Bool
```

and could now define instances such as

```
instance Eq a => Collection Set a where ...
```

in which the assumption `Eq a` can of course be used in the implementations of the methods.

Peyton-Jones' idea works well when there is a *single* element type which appears in all occurrences of the collection type. But it works much less well when the methods operate on collections of different types. If the `Collection` class included a method for mapping over collections,

```
mapC :: (a -> b) -> c a -> c b
```

then it would be unclear what the parameters of the `Collection` class should be.

- If only one element type appears as a parameter, for example as in

  ```
  class Collection c a where
    ...
    mapC :: (a -> b) -> c a -> c b
  ```

  then instances can constrain only one of `a` and `b`, and so an implementation of `mapC` which required equality on both types could not be made an instance of this class.

- On the other hand, if both type variables are made parameters of the class, as in

  ```
  class Collection c a b where
    ...
    mapC :: (a -> b) -> c a -> c b
  ```

  then any attempt to use the other methods of the class leads to ambiguous overloading, since the variable `b` does not occur in their types.

This problem arises also if we try to use Peyton-Jones' idea to define a restricted monad class, since the type of `return` involves only `m a`, while the type of `(>>=)` involves both `m a` and `m b`; what should the class parameters be?

Our approach, in contrast, works regardless of how many different occurrences of the restricted type constructor there are.

5.2 The 'Object-Oriented' Approach

An alternative way to simulate restricted data types is to build in the appropriate dictionary into the objects of the type. For example, if we define

```
data Set a = Set (EqD a) [a]
```

then it is clear that we can manipulate values of type `Set a` without requiring that `Eq a` hold in the context: we can obtain an implementation of equality directly from the `Set` we are working on. For example,

```
member x (Set dict xs) = any (eq dict x) xs
union xset@(Set dict xs) (Set _ ys) =
  Set dict (xs++[y | y<-ys, not (member y xset)])
```

The problem with this approach is that we cannot always guarantee that the arguments of a function will provide a suitable dictionary to construct its result. For example, `empty` and `singleton` construct `Set`s, in which they must place an equality dictionary, but they have no `Set` argument to extract it from. Likewise,

```
mapC :: (a->b) -> Set a -> Set b
```

should place an `EqD b` dictionary in its result, but can obtain only an `EqD a` dictionary from its argument. So none of these functions can be implemented.

The 'object-oriented' approach only works under strong restrictions on the types of the methods we want to implement. Our approach, on the other hand, works for all method types.

6 The Case for a Language Extension

We have now argued that restricted data types are useful, and that our simulation of them works better than other proposals. But given that restricted data types can be simulated in several dialects of Haskell already, why make a new language extension? We see three main reasons for doing so.

Firstly, our simulation requires that the designer of a constructor *class* anticipate whether or not it need support restricted datatypes as instances. In the case of a `Collection` class, it is fairly clear that it should, but in the case of the `Monad` class, for example, the class designer may not anticipate the need. The programmer who later wishes to declare `Set` to be a monad is then powerless to do so. But if restricted datatypes are built into the language, then the compiler can transform *all* class definitions appropriately, thus guaranteeing that a restricted datatype can be used anywhere an unrestricted one can.

Secondly, our simulation requires the programmer to declare the types of dictionaries for each class used in a datatype restriction. But these types are constructed internally by the compiler anyway, as part of the compilation of the class mechanism. Building restricted datatypes into the language spares the programmer from the need to duplicate the compiler's work.

Thirdly, to work really well, our simulation requires support for overlapping instances, which are in general a dangerous feature. Yet in our particular application, the overlap is safe. It is better to extend Haskell with a safe feature (restricted datatypes) than with a dangerous feature which can be used to simulate it.

We propose the following extension therefore. We introduce a new kind of context, `wft t`, to mean that the type `t` is *well-formed*. The built-in types are always well-formed; that is, there are instances

```
instance wft Int
instance wft (a, b)
instance wft (a -> b)
```

```
class Collection c where
  empty :: wft (c a) => c a
  singleton :: wft (c a) => a -> c a
  union :: wft (c a) => c a -> c a -> c a
  member :: wft (c a) => a -> c a -> Bool

data Eq a => Set a = Set [a]

instance Collection Set where
  empty = Set []
  singleton x = Set [x]
  union (Set xs) (Set ys) =
    Set (xs++[y | y<-ys, not (y `elem` xs)])
  member x (Set xs) = any (==x) xs
```

Figure 3: `Collection` and `Set` defined using restricted datatypes.

and so on.

A *restricted datatype definition*

$$\texttt{data (C}_1\ \overline{a}\texttt{,...,C}_n\ \overline{a}\texttt{) => T}\ \overline{a}\texttt{ = ...}$$

introduces instances

$$\begin{array}{c}\texttt{instance (C}_1\ \overline{a}\texttt{,...,C}_n\ \overline{a}\texttt{) => wft (T}\ \overline{a}\texttt{)}\\ \texttt{instance wft (T}\ \overline{a}\texttt{) => C}_1\ \overline{a}\\ \vdots\\ \texttt{instance wft (T}\ \overline{a}\texttt{) => C}_n\ \overline{a}\end{array}$$

Unrestricted datatype definitions are just the special case where $n = 0$.

Now, we insist that *every type appearing in a program must be well-formed.* That is, every expression must appear in a context which guarantees that every sub-expression has a well-formed type. Likewise, every type signature must carry a context which guarantees that the type itself is well-formed; every data type, newtype, and type synonym definition must carry a context which guarantees that the types on the right hand side are well-formed; and every instance declaration must carry a context which guarantees that the instance type and the types occurring in the instance methods are well-formed. Type constructors may only be applied to parameters which are themselves well-formed[4]. However, we can assume that `wft a` holds for every polymorphic type variable `a`, (since in any instance of a polymorphic type the instantiating type must be well-formed), and so such constraints need not appear in contexts.

With this extension, we could define the `Collection` class and its `Set` instance as shown in Figure 3. As we see from this example, the `wft` constraints correspond to `Sat` constraints in our simulation; for example, `wft (Set a)` corresponds to `Sat (SetCxt a)`. Notice that we can freely use operations that depend on `Eq a`, such as `(==)` and `elem`, in the methods of the `Set` instance, thanks to the generated `instance wft (Set a) => Eq a`. The instances generated from a restricted

[4]This applies only to parameters of kind `*`. Type parameters of other kinds are not restricted, but of course if they are used then their applications must be well-formed. *Example:* if `A` is defined by `data wft (c a) => A c a = A (c a)`, then in any use `A` $\kappa\ \tau$ the type τ must be well-formed by this rule, while the type constructor κ must satisfy `wft (`$\kappa\ \tau$`)` because of the context on the definition of `A`. This context must be present, since the type `(c a)` appears on the right hand side.

datatype definition are of course implemented just like the `Sat` instances we saw earlier:

$$\texttt{instance (C}_1\ \overline{a}\texttt{,...,C}_n\ \overline{a}\texttt{) => wft (T}\ \overline{a}\texttt{)}$$

constructs a dictionary for `wft (T` $\overline{a}$`)` which is a tuple of the dictionaries on the left hand side, and the instances

$$\texttt{instance wft (T}\ \overline{a}\texttt{) => C}_i\ \overline{a}$$

just select the appropriate dictionary from the tuple. As in our simulation, overlapping instances force the compiler to search for a successful context reduction, and avoid detectable loops.

Since the compiler knows that the well-formedness constraint for `Set` is `Eq` and no other, we do not need to parameterise `Set` on `Eq`, or parameterise the `Collection` class separately on the well-formedness constraint.

Well-formedness of type variables. We assume that constraints `wft a` (where `a` is a type variable) are always satisfied, since `a` can only be instantiated to a well-formed type. Consequently such constraints do not appear in contexts, and no corresponding dictionary is passed at run-time. But is this really safe? Even if we know that a dictionary for `wft a` must exist, we cannot construct one should it prove to be needed, without knowing the type `a`. Thus we must convince ourselves that such a dictionary can never be needed, and so passing it as a parameter is unnecessary.

To see this, note that the dictionary corresponding to `wft a` is of an unknown shape: it could indeed be any dictionary at all depending on which type `a` is instantiated to. Dictionaries are used to implement calls of class methods, and any such use requires that one know the dictionary shape. Thus a `wft a` dictionary can never be used.

To prove this formally, we should specify the translation of our extended language into F^ω (which would in any case be desirable, since this translation is used in the Glasgow Haskell compiler). In this translation, context constraints are translated into the types of dictionary parameters, and a constraint `wft a` would be translated into another type variable. Thus the translated code would be polymorphic in the type of the dictionary, which implies by parametricity that the dictionary is unused.

A more subtle argument is needed for type variables which are parameters of classes, because even if the method types are 'polymorphic' in these variables, their implementations are not. For example, given the class declaration

```
class BinOp a where
  binop :: a -> a -> a
```

then an instance of `binop` at the type `Set b` might very well use the fact that `Set b` is well-formed. But since no `wft a` constraint appears in the type of the class method, then calls of `binop` will pass no dictionary.

However, recall that the *instance declaration* must require that the instance type is well-formed. In this example, we would be forced to write

```
instance wft (Set b) => BinOp (Set b) where
  binop = ...
```

Thus the dictionary for `wft (Set b)` is supplied when that for `BinOp (Set b)` is created; there is no need to pass it each time the `binop` method is called.

7 Discussion

7.1 On well-formedness

The major surprise in the design we have presented is the introduction of a new kind of constraint in contexts, the `wft` constraints. Our design requires programmers to understand and write this new kind of constraint, which may seem unsatisfactory in (for example) the `Collection` class definition in Figure 3, given our initial motivation that it is 'obvious' that `Set` elements must have an equality. One might argue that it is 'obvious' from the types of the `Collection` class methods that `wft (c a)` must hold, and therefore there is no need for the programmer to write it. More generally, we might implicitly add `wft` constraints to each type signature to require that all the types occurring in it are well-formed, and thereby spare the programmer the need to know about them.

We have chosen not to follow this route, because it is not possible in general to infer which `wft` constraints must hold for the *body* of a function to be well-typed, just from its *type signature*. For example, suppose we extend the `Collection` class with a `mapC` method as in section 5.1, and define

```
existsC :: (Collection c, wft (c a), wft (c Bool)) =>
  (a -> Bool) -> c a -> Bool
existsC p c = member True (mapC p c)
```

Notice that for `existsC` to be well-typed, then `Collections` of `Bool` must be well-formed, since such a `Collection` is used internally in the definition. But this type does not appear in the type of `existsC`, and so it is impossible to implicitly insert the constraint `wft (c Bool)` just given the type signature. In general we cannot use the body of a function to decide which constraints to add to its type signature, because the type signature might appear in a class definition, for example, while the associated bodies appear scattered throughout the program in the corresponding instance declarations.

Our conclusion is that `wft` constraints should always be explicit. A half-way house would be to implicitly add the constraints which are obviously needed from the type signature, but let the programmer write (hopefully rare) additional constraints explicitly. We consider it wiser to let programmers become used to `wft` constraints by writing them often.

Indeed, we claim that `wft` constraints are naturally associated with restricted datatypes: when we declare that sets may only be built from elements with equality, we are stating that some types are well-formed and others are not. It can hardly be surprising that we then need to reason explicitly about well-formedness.

It is interesting to note that similar issues arise in Jones and Peyton Jones' proposal for extensible records [JJ99]. There a record type `{r | x::t}`, denoting record type `r` extended with the field `x`, is well-formed only if `r` 'lacks' `x`, written `r\x`. These 'lacks' constraints clutter the types of functions, and just as we do, Jones and Peyton Jones consider introducing (some of) them automatically, when their necessity can be inferred from the type of the function alone. Thus there is an interaction between these two proposals, and in a final design the same decision should be made in both cases.

7.2 On abstraction

We began this article by bemoaning the fact that the type of the `member` function reveals too much about the way that `Sets` are implemented. Specifically, replacing an implementation in terms of lists by one in terms of ordered trees will probably change the type of `member` from

```
member :: Eq a => a -> Set a -> Bool
```

to

```
member :: Ord a => a -> Set a -> Bool
```

Consequential changes to the types of other functions could force the modification of many type signatures in other modules.

With the extension we propose, both these types can be replaced by

```
member :: wft (Set a) => a -> Set a -> Bool
```

When `Set` is implemented by lists, then `wft (Set a)` is equivalent to `Eq a`, and when `Set` is implemented by ordered trees, then `wft (Set a)` is equivalent to `Ord a`.

However, while our extension *enables* the programmer to write type signatures which are robust across changes to the representation of `Set`s, it does not *force* him to do so. For example, the function

```
isOneOf x y z = member x (singleton y `union` singleton z)
```

can be given the robust type

```
isOneOf :: wft (Set a) => a -> a -> a -> Bool
```

But it can also be given the type

```
isOneOf :: Eq a => a -> a -> a -> Bool
```

when `Set`s are represented by lists, since `Eq a` implies `wft (Set a)` in that case.

Equally, the function

```
maxMember x y s = member (x `max` y) s
```

can be given the robust type

```
maxMember :: (Ord a, wft (Set a)) => a -> a -> Set a -> Bool
```

But if `Set`s are represented by ordered trees, then it can also be given the type

```
maxMember :: wft (Set a) => a -> a -> Set a -> Bool
```

since `wft (Set a)` implies `Ord a` in that case.

If the programmer chooses to write type signatures such as these, whose validity depends on the conditions under which `wft (Set a)` holds, then of course a change to the representation of `Set`s will invalidate them. Our proposal makes it possible to write robust type signatures, but does not guarantee that all type signatures are robust.

In a sense, the constraint `wft (Set a)` behaves like a *context synonym* for the context on the definition of type `Set`, and the synonym is not abstract. It would be interesting to consider ways to restrict the scope of the synonym, for example to the same scope as the datatype constructors, to force programmers to write robust type signatures elsewhere.

7.3 Overhead of dictionary passing

A major problem with the implementation we have suggested is that it requires passing many more dictionaries than usual, leading to a potentially high overhead. In particular, the definition of class `Monad` must be revised as follows:

```
class Monad m where
  return :: wft (m a) => a -> m a
  (>>=) :: (wft (m a), wft (m b)) => m a -> (a -> m b) -> m b
```

The thought of passing two dictionaries to every call of `(>>=)` is probably enough to put any implementor off.

The problem here is that programs which do not use restricted datatypes would still pay a heavy cost for their inclusion in the language, by passing a large number of empty dictionaries around. Our proposal here is to generate two versions of the code for each function whose type signature involves `wft`, one to be used when all the dictionaries involved are empty, and the other when genuinely restricted datatypes are involved. This should only affect constructor-polymorphic functions, so the amount of code duplication should be small, while the performance penalty for programs which do not use restricted datatypes is completely removed.

A related difficulty under our proposal is that adding restricted datatypes to the language might make some definitions overloaded which were not overloaded before — if the context they require contains only `wft` constraints. Such definitions would be subject to the monomorphism restriction after the extension, but not before, and this could cause type-checking to fail. We believe the correct solution here is to revise the monomorphism restriction.

7.4 Lazy vs eager context reduction

While we have explained that context reduction must search for a suitable reduction path, we have said little about *when* context reduction should occur. In Haskell as it stands, context reduction is performed *eagerly*, as part of inferring the type signature of each function. When the programmer states a context explicitly in a type signature, then it is clear that the compiler can search for a way to reduce the context that the function body requires to the given one. When the programmer leaves type signatures to be inferred by the compiler, then it is much less clear which reduction the compiler should choose. That leads us to suggest that context reduction should instead be *lazy*, that is, performed only when necessary to match contexts given explicitly by the programmer (or when a context is *tautologous*, that is can be reduced to an empty context). Non-tautologous contexts in inferred type signatures would not be reduced at all.

Peyton-Jones, Jones and Meijer come to the same conclusion in their exploration of the design space for classes [JJM97]. They point out in particular that, in combination with overlapping instances, eager context reduction can type fewer programs than lazy can. This is also true in our situation. Consider:

```
allpalin :: wft (Set [a]) => Set [a] -> Bool
allpalin (Set xss) = all palindrome xs
palindrome xs = xs==reverse xs
```

With lazy context reduction, the type signature inferred for `palindrome` is

```
palindrome :: Eq [a] => [a] -> Bool
```

and `allpalin` is well-typed, since `Eq [a]` is implied by `wft (Set [a])`. But if eager context reduction is used instead, then the type inferred for `palindrome` is

```
palindrome :: Eq a => [a] -> Bool
```

and `allpalin` becomes ill-typed — a good reason to prefer the former.

7.5 An alternative: abstracting over contexts

The extension we have proposed is not the only possible way to support restricted data types. An alternative would be to allow type and class definitions to be parameterised not just on types, but also on contexts. We could then carry out our

simulation much more easily, without needing to represent classes by types — one would declare the `Collection` class, for example, as:

```
class Collection c cxt where
  empty :: cxt a => c cxt a
  ...
```

Of course, the ability to abstract over contexts might be useful in other ways too.

A natural complement would be to give contexts names via *context synonyms*. Interestingly, we can almost do so in Haskell already. A 'synonym' for the context `(A a, B a)` can be modelled by a new class and instance

```
class (A a, B a) => AB a
instance (A a, B a) => AB a
```

The instance declaration allows a context `AB a` to be reduced to `(A a, B a)`, while the class declaration enables both `A a` and `B a` to be reduced to `AB a`. While context reduction might in principle loop here also, in fact the loop is avoided by treating class declarations specially.

However, this approach does have some important disadvantages. Firstly, even if we can abstract over a context in the `Collection` class, there is no obvious way to associate the type `Set` with the context `Eq`. We would need to make *both* the collection type, and the associated context, into parameters of the `Collection` class. To avoid ambiguous overloading, we could make sure that the context parameter appears in the method types (as we have done above). Alternatively, we might restrict instance declarations so that no two instances of the same class may have the same type parameters. Such a restriction would guarantee that the type parameters determine the context parameters, thus in effect creating an association between `Set` and `Eq`. In any case, there are subtle design choices to be made here.

Perhaps a more serious objection is that this approach would still require the class designer to *anticipate* that another programmer might later wish to make a restricted datatype into an instance. Many class designers would fail to do so, and the frustration of using restricted datatypes would remain.

8 Conclusion

It is common for the implementation of an abstract datatype to place some restrictions on its argument types. The consequent loss of abstraction when the implementation is changed, and the difficulties of making such implementations instances of more general classes, are recurring topics on the Haskell mailing list. We believe there is a crying need for a restricted datatype construct with much more power than Haskell's present sham.

The simulation in the first part of this paper enables us to explore the semantics and implementation of restricted datatypes. We argue that our solution is significantly more useful than either Peyton-Jones' approach to `Collection` classes or a more 'object-oriented' approach.

Finally, we propose a language extension based on our simulation, and argue that it is both natural, and can be implemented with reasonable efficiency. We believe the extension would be invaluable, and live in hope that Haskell implementors will take up the challenge!

Acknowledgements

Simon Peyton-Jones, Mark Jones, and Lennart Augustsson have all made very useful comments on this work at various stages. I am grateful to all of them, while

the remaining errors are of course my own.

References

[Hug99] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, to appear, 1999.

[JJ99] Mark P Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*, Paris, September 1999.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

[PJ96] Simon Peyton-Jones. Bulk types with class. In *Electronic proceedings of the Glasgow Functional Programming Workshop*, Ullapool, July 1996.