# Abstract Interpretation of Functional Programs using an Attribute Grammar System

*Jeroen Fokker*

*S. Doaitse Swierstra*

institute of information and computing sciences, utrecht university

technical report UU-CS-2007-049

www.cs.uu.nl

# Abstract Interpretation of Functional Programs using an Attribute Grammar System

Jeroen Fokker and S. Doaitse Swierstra

December 7, 2007

**Abstract**

We describe an algorithm for abstract interpretation of an intermediate language in a Haskell compiler, itself also written in Haskell. It computes approximations of possible values for all variables in the program, which can be used for optimizing the object code. The analysis is done by collecting constraints on variables, which are then solved by fixpoint iteration. The set of constraints grows while solving, as possible values of unknown functions become known. The constraints are collected by decorating the abstract syntax tree with an attribute grammar based preprocessor for Haskell. An introduction to this preprocessor is also given.

## 1 Introduction

Early implementations of lazy functional languages were usually based on graph rewriting techniques. Substitution is done in evaluation of a $\beta$-redex

$$(\lambda x \,.\, b)\; a \;\;\Rightarrow\;\; b\,[\,x\,/\,a\,]$$

either directly by walking through a copy of $b$ and replacing occurences of $x$, or indirectly by compilation to SKI-combinators [14] or super-combinators [7].

Later, other approaches were taken which compile to an abstract machine model based on stacks and continuations. The instructions of this "STG-machine" can be mapped to those of traditional hardware architectures [12].

Lazy evaluation of functional languages is implemented by, instead of calling functions directly, building "closures" of functions, i.e. heap records containing a reference to the function and to its arguments. Such a closure is forced to evaluation when the result is actually needed, viz. when it is used in a case-expression or passed in a strict argument position.

In a naive implementation, the function reference can be a tag, and a special evaluation function performs case distinction on this tag. Peyton Jones et al. describe an encoding [12], in which the tag is actually a pointer to an information table, which in turn contains a pointer to the code of the function. Evaluating a closure now amounts to just calling that code. The double indirection in this encoding can be reduced to a single indirection by having the "tag" point directly to the code, and putting the rest of the information table just before that code in memory [10].

Either way, evaluation involves calling code through an indirection pointer. On modern pipelined processors, this is a costly operation, as it stalls the prefetching pipeline. Therefore, Boquist proposes to return to the naive encoding [3]. To avoid the overhead of calling the evaluation function which does the case distinction between tags, the evaluation function is "inlined" whenever used. To prevent copying the large body of the evaluation function, each occurence of the case analysis is pruned to contain only those cases that can actually occur in that particular instance. This way, evaluation amounts to a few tests and conditional jumps, and indirect jumps are avoided completely. Branch prediction schemes that are built in in pipelined processors can deal with the

conditional jumps efficiently. Also, the conditional jumps are very local, and are likely to have their target within the instruction cache.

To do the pruning it is necessary to know for each closure what its possible tags are. This is to be determined by a global control flow analysis. Boquist sketches an algorithm for this abstract interpretation [4]. Here we present a full implementation we employ in our experimental Haskell compiler [6].

Algorithms are often described in some mathematical formalism. A problem of mathematical notation is that it lacks sophisticated data structures. We feel it is paradoxical that for describing, e.g., the subtleties of the Haskell type system, one often uses mathematical notation which itself is almost untyped. So, where a mathematical description ought to be more abstract than an implementation, sometimes it is cluttered with low-level encodings of data structures in terms of lists and tuples.

A way out of this paradox is to use Haskell itself as the description language. We consider a paper like "Typing Haskell in Haskell" [8] to be better readable than many a formal treatise on the same subject. An added benefit is that the description is actually executable code, which makes an error-prone translation from specification to implementation obsolete.

To be useful as an algorithm description intended for human readers, we try to abstract from trivial details as much as possible. Although Haskell has many mechanisms for abstraction, we think that for tree processing algorithms it is helpful to use notions derived from the realm of attribute grammars [9]. In order not to loose executability of our implementation, we use a preprocessor that translates the attribute grammar notions to plain Haskell. To make the paper self-contained, we include a description of this preprocessor as well.

The aim of this paper is twofold:

1 (technical) to give a concise, executable description of the abstract interpretation algorithm that is needed to avoid indirect jumps when evaluating a closure in a lazy functional language;
2 (methodological) to provide a case study for the use of Haskell and attribute grammar related techniques for the description of an algorithm, to show that is enables a concise and clear representation.

In section 4 we present the actual algorithm. Before that, we introduce the language to be analyzed in section 3, and the attribute grammar preprocessor for Haskell in section 2.

# 2 Tree walk methodology

## 2.1 Defining semantics

Functional languages are famous for their ability to parameterize functions not only with numbers and data structures, but also with functions and operators. The standard textbook example involves the functions *sum* and *product*, which can be defined separately by tedious inductive definitions:

$$
\begin{aligned}
&\mathit{sum} \quad [\,] &&= 0\\
&\mathit{sum} \quad (x : xs) &&= x + \mathit{sum}\ xs\\
&\mathit{product}\ [\,] &&= 1\\
&\mathit{product}\ (x : xs) &&= x * \mathit{product}\ xs
\end{aligned}
$$

but, once this pattern has been generalized in a function *foldr* that takes as additional parameters the base value and the operator to apply in the inductive case:

$$
\begin{aligned}
&\mathit{foldr}\ op\ e\ [\,] &&= e\\
&\mathit{foldr}\ op\ e\ (x : xs) &&= x\ `op`\ \mathit{foldr}\ op\ e\ xs
\end{aligned}
$$

could easily have been defined as specializations of the general case:

$$
\begin{aligned}
sum &= foldr\ (+)\ 0 \\
product &= foldr\ (*)\ 1
\end{aligned}
$$

Indeed, good generalizations might have unexpected applications in other domains:

$$
\begin{aligned}
concat &= foldr\ (+\!\!+)\ [\,] \\
sort &= foldr\ insert\ [\,] \\
transpose &= foldr\ (zipWith\ (:))\ (repeat\ [\,])
\end{aligned}
$$

The idea that underlies the definition of *foldr*, i.e. to capture the pattern of an inductive definition by having a function parameter for each constructor of the data structure, can also be used for other data types, and even for multiple mutually recursive data types. A function that can be expressed in this way was called a *catamorphism* by Bird, and the collective extra parameters to *foldr*-like functions an *algebra* [2, 1]. Thus, $((+), 0)$ is an algebra for lists, and $((+\!\!+), [\,])$ is another. In fact, every algebra defines a *semantics* of the data structure. When applying *foldr*-like functions to the algebra consisting of the original constructor functions, such as $((:), [\,])$ for lists, we have the identity function. Such an algebra is said to define the "initial" semantics.

Outside circles of functional programmers and category theorists, an algebra is simply known as a "tree walk". In compiler construction, algebras could be very useful to define a semantics of a language or, bluntly said, to define tree walks over the parse tree. The fact that this is not widely done, is due to the following problems:

1. Unlike lists, for which *foldr* is standard, in a compiler we deal with custom data structures for abstract syntax of a language, which each need a custom *fold* function. Morover, whenever we change the abstract syntax, we need to change the *fold* function and every algebra.
2. Generated code can be described as a semantics of the language, but often we need additional semantices: listings, messages, and internal structures (symbol tables etc.). This can be done by having the semantic functions in algebras return tuples, but this makes them hard to handle.
3. Data structures for abstract syntax tend to have many alternatives, so algebras end up to be clumsy tuples containing dozens of functions.
4. In practice, information not only flows bottom-up in the parse tree, but also top-down. E.g., symbol tables with global definitions need to be distributed to the leafs of the parse tree to be able to evaluate them. This can be done by using higher-order domains for the algebras, but the resulting code becomes even harder to understand.
5. A major portion of the algebra is involved with moving information around. The essense of a semantics is sparsely present in the algebra and obscured by lots of boilerplate.

Many compiler writers thus end up writing ad hoc recursive functions instead of defining the semantics by a algebra, or even resort to non-functional techniques. Others succeed in giving a concise definition of a semantics, often using proof rules of some kind, but thereby loose the executability. For the implementation they still need conventional techniques, and the issue arises whether the program soundly implements the specified semantics.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell [13] that overcomes the abovementioned problems. It is not a separate language; we can still use Haskell for writing auxiliary functions, and use all abstraction techniques and libraries available. The preprocessor just allows a few additional constructs, which can be translated into a custom *fold* function and algebras, or an equivalent more efficient implementation.

## 2.2 An Attribute Grammar based preprocessor for Haskell

We describe the main features of the preprocessor here, and explain why they overcome the five problems mentioned above. To start with, the abstract syntax of the language is defined in a **syntax** declaration, which is like a Haskell **data** declaration with named fields. The difference is that we don't have to write braces and commas, and that constructor function names need not be unique. As an example, we define a fragment of a typical imperative language:

```
syntax Stat  = Assign dest  :: String   src  :: Expr
             | While  cond  :: Expr     body :: Stat
             | Group  elems :: [Stat]
syntax Expr = Const  num   :: Int
             | Var    name  :: String
             | Add    left  :: Expr     right :: Expr
             | Call   name  :: String   args  :: [Expr]
```

The preprocessor generates corresponding **data** declarations (making the constructors unique by prepending the type name, like *Expr_Const*), and generates a custom *fold* function. This overcomes problem 1.

For any desired value we wish to compute over a tree, we can declare a "synthesized attribute". Attributes can be declared for one or more data types. For example, we can declare that both statements and expressions need to synthesize bytecode as well as pretty-printed listing, and that expressions can be evaluated to an integer value:

```
attr Expr Stat syn bytecode :: [Instr]
               syn listing  :: String
attr Expr      syn value    :: Int
```

The preprocessor generates semantic functions that return appropriate tuples, but we can simply refer to attributes by name. This overcomes problem 2.

The value of each attribute needs to be defined for every constructor of every data type which has the attribute. As this defines the semantics of the language, these definitions are known as "semantic rules", and start with keyword **sem**. An example is:

```
sem Stat | Assign
  @lhs.listing = @dest.listing ++ ":=" ++ @src.listing ++ ";"
```

This states that the synthesized *listing* attribute of an assignment statement can be constructed by combining the *listing* attributes of its *dest* and *src* children and some fixed strings. The @-symbol in this context should be read as "attribute", not to be confused with Haskell "as-patterns". At the left of the =-symbol, the attribute to be defined is mentioned; at the right, any Haskell expression can be given. The @-symbol may be omitted in the destination attribute, as is done in the next example. This example shows that it is indeed useful that any Haskell expression, with embedded occurrences of child attributes, can be used in the definition. Also, it shows how to use the value of terminal symbols ( @*num* in the example), and how to group multiple semantic rules under a single **sem** header:

```
sem Stat | While
  lhs.bytecode = let k = length @cond.bytecode
                     n = length @body.bytecode
                 in  @cond.bytecode ++ [BEQ (n + 1)]
                     ++ @body.bytecode ++ [BRA (-(n + k + 2))]
sem Expr
  | Const lhs.value =  @num
  | Add   lhs.value =  @left.value + @right.value
```

The preprocessor collects and orders all definitions in a single algebra, replacing attribute references by suitable selections from the results of the tree walk on the children. This overcomes problem 3.

To be able to pass information downward during a tree walk, we can define "inherited" attributes (the terminology goes back to Knuth [9]). As an example, it can serve to pass an environment, i.e. a lookup table that associates variables to values, which is needed to evaluate expressions:

```
type Env = [(String, Int)]
attr Expr inh env :: Env
```

**sem** *Expr* | *Var*
  **lhs**.*value* = *fromJust* (*lookup* @**lhs**.*env* @*name*)

The value to use for the inherited attributes can be defined in semantic rules higher up the tree:

**sem** *Stat* | *Assign*
  *src.env* = [("x", 37), ("y", 42)]

The preprocessor translates inherited attributes into extra parameters for the semantic functions in the algebra. This overcomes problem 4.

In the example above, an environment with two variables was just made up. In reality, a *Stat* construct probably inherited the environment from even higher constructs, say a procedure declaration. This means that the only thing that needs to be done at the *Stat* level, is to pass the inherited environment down to the children. This can be quite tedious to do:

**sem** *Stat*
  | *Assign* *dest.env* = **lhs**.*env*
           *src.env* = **lhs**.*env*
  | *While* *cond.env* = **lhs**.*env*
           *body.env* = **lhs**.*env*

Luckily, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. So, the attribute *env* that a *Stat* inherited from its parent, is automatically copied to the children which also inherit an *env*, and the tedious rules above can be omitted. A similar automated copying is done for synthesized attributes, so if they need to be passed unchanged up the tree, this needs not to be explicitly coded.

When more than one child offers a candidate to be copied, normally the last one is taken. But if we wish a combination of the copy candidates to be used, we can specify so in the attribute declaration. For example:

**attr** *Expr Stat*
  **syn** *listing* **use** (++) []

which specifies that by default, the synthesized attribute *listing* is the concatenation of the *listing*s of all children that have one, or the empty list if no child has one. This defines a useful default rule, which can be overridden when extra symbols need to be interspersed, as for example in the definition of *listing* for assignment statements given earlier.

It is allowed to declare both an inherited and a synthesized attribute with the same name. In combination with the copying mechanisms, this enables us to silently thread a value through the entire tree, updating it when necessary. See section 4.3 which maintains, in attribute *location*, a unique counter during the tree walk. This captures a pattern for which often *Reader* and *Writer* monads are introduced [8].

The preprocessor automatically generates semantic rules in the standard situations described, and this overcomes problem 5.

# 3 The Grin language

Grin (Graph Reduction Intermediate Notation) was proposed by Boquist as an intermediate language sitting between the Core language (that in Haskell compilers describes a desugared program) and an imperative backend [3].

We describe a slightly modified version here, which is more explicit than Boquist's original description about what constructs are allowed at various places. Instead of the usual BNF description, we introduce the language by means of Haskell data type declarations (or rather **syntax** declarations for the AG preprocessor). The advantage of this approach is that this explicitly mentions the types and names of child constructs of each nonterminal symbol. Also it is part of our endeavour to

make the description to serve both as the specification and as the implementation of the abstract semantics of the language.

The semantics/interpretation that we deal with in this paper is an abstract interpretation needed for analysis of the program. In the same style we are able to present other semantices. In our compiler [6] we implement a translation to bytecode that is executable by a simple interpreter, and a translation to a generic imperative language that can in turn be translated to various backend languages.

In the presentation of the language we do not provide a concrete syntax for the language, as normally is implicitly done in a BNF description. One reason for this is that a concrete syntax is unnecessary, as Grin programs are only an intermediate representation in the compilation process, and technically are merely data structures. Another reason is that the mental parsing and unparsing involved when reading the semantics description in later sections could distract the reader from the algorithm proper, and cause confusion between program fragments as data structures and their semantic values.

We start our description with a definition of toplevel constructs. A program consists of a single module, which has a name, a list of global variable definitions, and a list of function bindings. Note that in our naming, we conventionally use suffix *L* for "list", and prefix *mb* for "maybe".

**syntax** *Program* = *Prog* *mod* :: *Module*
**syntax** *Module* = *Mod* *nm* :: *Name* *globalL* :: *GlobalL* *bindL* :: *BindL*
**type** *GlobalL* = [*Global*]
**type** *BindL* = [*Bind*]

A global definition binds a name to a term, whereas a lambda binding binds a parameterized name to an expression.

**syntax** *Global* = *Global* *nm* :: *Name* *val* :: *Term*
**syntax** *Bind* = *Bind* *nm* :: *Name* *argNmL* :: [*Name*] *expr* :: *Expr*

Grin programs manipulate five kinds of values: integers, standalone tags, nodes with a known tag and a list of fields, pointers to a node stored on the heap, and the empty value. The first three have a direct syntactic representation as a *Term*, pointers and the empty value have not. Another possible *Term* is a variable, which can refer to any of the five kinds of value.

**syntax** *Term* = *LitInt* *int* :: *Int*
| *Tag* *tag* :: *Tag*
| *Node* *tag* :: *Tag* *fldL* :: *TermL*
| *Var* *nm* :: *Name*
**type** *TermL* = [*Term*]

Although the syntax above allows fields of a *Node* be any *Term*, we do not make use of nested nodes; if they are desired, the field list should contain variables that point to heap cells storing the inner nodes.

Six different tags are used to label nodes: *Con*, *Fun*, *PApp* and *App*and two special ones *Unboxed* and *Hole*. A *Con* tag labels nodes that build up data structures. They correspond to constructor functions in the Haskell source program, but unlike constructor functions, nodes with a *Con* tag are always fully saturated. A *Fun* tag labels "thunks" , i.e. function applications of which the evaluation is postponed for lazy evaluation. Nodes with a *Fun* tag are always fully saturated. A *PApp* tag indicates an unsaturated lazy function call (partial parameterization) and records, apart from the function name, also the number of parameters it still *needs* to become fully saturated.

**syntax** *Tag* = *Con* *nm* :: *Name*
| *Fun* *nm* :: *Name*
| *PApp* *needs* :: *Int* *nm* :: *Name*
| *App*
| *Unboxed*
| *Hole*

The other three tags are an extension to those proposed by Boquist. A *PApp* tag indicates an unsaturated lazy function call (partial parameterization) and records, apart from the function name, also the number of parameters it still *needs* to become fully saturated. The *Unboxed* tag is a mockup tag for constructs that conceptually are nodes, but in reality are implemented as unboxed values. Finally, the *Hole* tag is used in the implementation of recursive definitions, but plays no special role in the analysis described in this paper.

The main construct in Grin is an expression, which represents the body of a function binding. Evaluation of expressions may lead to side effects on the heap. There are twelve cases in the expression syntax:

```
syntax Expr = Unit        val  :: Term
            | Seq         expr :: Expr    pat :: PatLam    body :: Expr
            | Case        val  :: Term    altL :: AltL
            | Store       val  :: Term
            | UpdateUnit  nm   :: Name    val :: Term
            | FetchNode   nm   :: Name
            | FetchUpdate src  :: Name    dst :: Name
            | FetchField  nm   :: Name    offset :: Int    mbTag :: Maybe Tag
            | Call        nm   :: Name    argL :: TermL
            | FFI         nm   :: String  argL :: [Name]   tagL :: TagL
            | Eval        nm   :: Name
            | Apply       nm   :: Name    argL :: TermL
```

We give an informal description of the semantics of these constructs, that is their runtime evaluation result and side effects on the heap. A formal description would be a Grin interpreter, which is not the focus of this paper.

An expression *Unit val* simply evaluates to a known value *val*. Evaluation of expression *Seq expr pat body* first evaluates *expr*, binds the result to *pat* and evaluates *body* in the extended environment. Boquist uses a monadic style concrete syntax for this construct: $expr; \lambda pat \rightarrow body$, which is why we declared *pat* to have type *PatLam* (for "lambda pattern"). It can however just as well be thought of as **let** $pat = expr$ **in** *body* or even as an imperative style assignment $pat := expr; body$. Concrete syntax is immaterial; what is important is that *expr* and *body* are evaluated sequentially.

A *Case* expression selects from a list of alternatives the one with a pattern that matches the value of the variable in the *Case* header (the "scrutinee"). Each alternative consists of a pattern and a corresponding expression.

```
type AltL  = [Alt]
syntax Alt = Alt pat :: PatAlt   expr :: Expr
```

Patterns in a case alternative normally consist of a node with a known tag, and variables as arguments. Stand-alone tags and literal integers are also possible patterns:

```
syntax PatAlt = LitInt int :: Int
              | Tag    tag :: Tag
              | Node   tag :: Tag   fldL :: [Name]
```

A pattern in a case alternative is quite different from a lambda pattern in a *Seq* expression. A lambda pattern is often just a variable name. Two other possibilities are *Empty*, to be able to match for the empty result value of the *FetchUpdate* expression that only has a side effect, and a node denotation where the tag can, but needs not be, known:

```
syntax PatLam = Empty
              | Var        nm  :: Name
              | VarNode    fldL :: VarL
syntax Var    = Var        nm  :: Name
              | KnownTag tag   :: Tag
type VarL     = [Var]
```

We assume the existence of a special name

*wildcard* :: *Name*

which can serve as a "don't care variable" in a lambda pattern.

Boquist proposes two constructs which have a side effect on the heap: *Store*, which stores a node value in a new heap cell and returns a pointer to it, and *Update*, which stores a node value in an existing heap cell and returns the empty value. We do have a *Store* expression in our language, but instead of a separate *Update* expression we have *UpdateUnit*, which combines the overwriting of an existing heap cell with returning the value. This allows for a more efficient implementation of the combination. Boquist uses a single construct *Fetch* for fetching either a complete node, or a particular field of a node. Because these two variants behave quite differently, we have separate constructs *FetchNode* and *FetchField*, and a *FetchUpdate* which combines fetching a node and using it to update an existing heap cell.

Next, we have *Call* for calling a Grin function, and *FFI* for calling a foreign function. Boquist proposes the use of two builtin functions *eval* and *apply*, which can be called to force evaluation of a variable, or to apply an unknown function in a strict context, respectively. As these functions behave quite different from ordinary functions, we include special constructs *Eval* and *Apply* for these cases.

To complete our exposition of the Grin language, we define abbreviations for some groups of nonterminal symbols, which facilitates the definition of attributes that are needed for all of them:

**set** *AllDef* = *Global GlobalL Bind BindL*
**set** *AllTerm* = *Term TermL*
**set** *AllExpr* = *Expr Alt AltL PatAlt PatLam Var VarL*

# 4 Abstract interpretation

In this section we describe an abstract interpretation algorithm, which solves a set of constraints by fixpoint iteration. Constraints are first collected in a walk over the tree that represents the Grin program. We start with a description of an abstract domain, and a language for specifying the constraints.

## 4.1 An abstract domain

Grin programs largely consist of bindings from Grin expressions to function names. Expressions in turn are built from terms, of which a possible form is a single variable. Although Grin is untyped, in code generated from a correct Haskell program variables always refer to values of the same kind: the empty value, other basic values such as integers, complete nodes, standalone tags, or heap pointers. We use abstract interpretation not only to infer these kinds, but also to collect more detailed information about the runtime structure of values.

When executed, a Grin program maintains a heap of dynamically allocated nodes. More specifically, execution of a *Store* expression allocates a new heap cell, as do *Global* variable definitions. Our abstract interpretation algorithm also determines, for each *Store* expressionand each *Global* definition, what type of node it can create. The abstraction of all heap cells that a particular *Store*-expression or *Global*-binding creates is known as a *Location*. Thus, each *Location* corresponds uniquely to a *Store* or *Global*. In our implementation we identify locations simply by unique, consecutive numbers. Also, each *Variable* is also represented by a number. A preprocessing stage uniquely numbers all variable names in a program (taking care of scoping where necessary), and makes the sequence number available through a function

**type** *Location* = *Int*
**type** *Variable* = *Int*
*nr* :: *Name* → *Variable*

We introduce a data type *AbsValue* to describe the domain in the abstract interpretation. It distinguishes four cases for the five different kinds of value (both the empty value and integers are regarded as "basic"), with added bottom and error cases to form a complete lattice suitable for fixpoint iteration.

```
data AbsValue = AbsBottom
              | AbsBasic
              | AbsTags  (Set Tag)
              | AbsLocs  (Set Location)
              | AbsNodes (Map Tag [AbsValue])
              | AbsError String
```

In the *AbsTags* case, abstract interpretation reveals to which tags a variable can possibly refer. Similarly, for *AbsLocs* we determine to which locations a pointer can point. In the *AbsNodes* case, we not only determine the possible tags of the nodes, but for each of these also a list of the abstract values of their parameters. In section 3 we stipulated that nested nodes are only allowed by letting the fields be variables which refer to pointers to heap cells storing the inner nodes. This invariant propagates to *AbsNodes*: the elements of the fields of a node are never *AbsNodes* themselves, but can be *AbsLocs* pointing to locations which store inner nodes.

The fact that *AbsValue* indeed forms a lattice is expressed by the following definition, which specifies how two abstract values can be merged into one. We state that *AbsBottom* is the identity of a *Monoid*

```
instance Monoid AbsValue where
  mempty = AbsBottom
```

That is, any abstract value remains unchanged when merging it with *AbsBottom*

```
mappend a          AbsBottom = a
mappend AbsBottom b          = b
```

Abstract values of each of the four types can be merged with others of the same type:

```
mappend AbsBasic      AbsBasic      = AbsBasic
mappend (AbsTags @)   (AbsTags bt)  = AbsTags (Set.union @ bt)
mappend (AbsLocs al)  (AbsLocs bl)  = AbsLocs (Set.union al bl)
mappend (AbsNodes an) (AbsNodes bn) = AbsNodes (Map.unionWith (zipWith mappend) an bn)
```

Errors remain errors even when merged with other values:

```
mappend a@(AbsError _) _ = a
mappend _ b@(AbsError _) = b
```

New errors originate from merging abstract values from incompatible types:

```
mappend a b = AbsError (show a ++ " conflicts " ++ show b)
```

The goal of the abstract interpretation algorithm is to determine the abstract value of each variable in the program, and likewise for each abstract heap *Location*. For efficiency reasons we represent these mappings by arrays:

```
type AbstractEnv s  = STArray s Variable AbsValue
type AbstractHeap s = STArray s Location AbsValue
```

## 4.2 A constraint language

By observing a Grin program, we can deduce equations to constrain variables and locations. Before doing so, we need a language to specify such constraints. We introduce type *Equation* for describing six kinds of constraints for the abstract value of variables. Likewise, we have *HeapEquation* for constraining the abstract values of abstract heap locations.

```
data Equation = IsKnown        Variable AbsValue
              | IsSuperset     Variable Variable
              | IsSelection    Variable Variable Int Tag
              | IsConstruction Variable Tag [Maybe Variable]
              | IsEvaluation   Variable Variable
              | IsApplication  (Maybe Variable) [Variable]
```

Five out of the six equation types constrain a variable to fulfil certain properties. Only in the case of an *IsApplication* equation, the variable that is constrained appears *Maybe*, i.e. is optional.

A variable may be constrained by more than one equation. These equations are cumulative. If for example one constraint specifies that a variable "is known" to have a particular abstract value, and another constraint specifies that it is known to have another value, the abstract interpretation algorithm concludes that this variable can refer to either value.

Below we informally describe the semantics of the six equation types. A formal description is given in figure 2, which is discussed in section 4.4.

1. An equation *IsKnown* $v$ $a$ means that variable $v$ can have abstract value $a$.
2. The meaning of *IsSuperset* $v$ $w$ is that variable $v$ can have all values that variable $w$ has.
3. The equation *IsSelection* $v$ $n$ $i$ $t$ expresses that $v$ can be the selection of the $i$th component of any node tagged by $t$ which can be the value of variable $n$.
4. The meaning of *IsConstruction* $v$ $t$ $as$ is that $v$ can be a node with tag $t$ and arguments $as$. Not all arguments need to be known.
5. The meaning of *IsEvaluation* $v$ $w$ is that $v$ can refer to the evaluation result of any possible value of $w$.
6. The meaning of *IsApplication* $v$ $(f : as)$ is that $f$ is a variable that refers to a function which is applied to values referred to by variables $as$, and that the result is a possible value of $v$. For this type of constraint, mentioning a variable $v$ is optional. If it is lacking, the equation still bears information on the possible values of parameters of $f$.

For heap equations, we have only one constraint type:

```
data HeapEquation = WillStore Location Tag [Maybe Variable]
```

The meaning of *WillStore* $p$ $t$ $as$ is that location $p$ stores a node with tag $t$ and arguments $as$. A heap cell always stores a complete node, not an isolated value of other type (basic value, tag or pointer to another heap cell).

The sets of constraints for variables and locations, respectively, are collected in lists, for which we define the following types:

```
type Equations     = [Equation]
type HeapEquations = [HeapEquation]
```

## 4.3 Collecting constraints in a tree walk

In this subsection we describe a tree walk over a Grin program that collects constraints on the program variables. The tree walk is implemented using the attribute grammar (AG) based language described in section 2.

The goal of the tree walk is to synthesize *equations* stating the constraints for program variables, and *heapEqs* stating the constraints for locations (abstract results of store expressions and global definitions).

```
attr Program Module AllDef AllExpr
  syn equations use (++) [] :: Equations
  syn heapEqs  use (++) [] :: HeapEquations
```

The declarations above specify that both type of equations are not only synthesized for the whole program, but also for the intermediate levels of the program tree that have to do with definitions

and expressions. No equations are synthesized on the levels that have to do with values and variables.

The **use** clause in the declaration of the attributes expresses that the default way to synthesize equations is just to concatenate the equations synthesized on underlying levels. We will redefine the *equations* and *heapEqs* attributes for the tree positions where equations are introduced.

First, we introduce some auxiliary attributes. We need to uniquely number all abstract locations, as we represent locations by integers. For this purpose we have both a synthesized and an inherited attribute *location* for all relevant positions in the tree. With a semantic rule, value 0 is inserted for this attribute at the top of the tree.

```
attr Program Module AllDef AllExpr
  inh syn location :: Int
sem Program | Prog
  mod.location = 0
```

The AG preprocessor ensures that the inherited attributes are passed unchanged down the tree, and the synthesized values are passed up, unless there is a semantic rule which specifies that a modified value should be passed. Indeed, in figure 1 we have rules that increment the location counter when locations need to be numbered, viz. at *Store* expressions and *Global* definitions.

Before we explain the rest of the rules in figure 1, we define an auxiliary data structure needed as the type of some attributes to come. Nodes sometimes are indirectly referred to by a variable, sometimes they are directly enumerated in full. The following data type distinguishes these two cases, where the polymorphic type variable $a$ is the type of additional information that we may want to express for the parameters of the node. Function *fromInVar* retrieves the variable from a *NodeInfo* that is known to be a *InVar* case.

```
data NodeInfo a =  InVar  Variable
                |  InNode Tag [a]
fromInVar :: NodeInfo a → Variable
fromInVar (InVar v) = v
```

This data type is the type of attributes *termInfo* and *patInfo* that summarize whether terms and patterns are denoted indirectly through a variable, or directly as a node with tag and fields:

```
attr Term    syn termInfo :: NodeInfo (Maybe Variable)
attr PatAlt
     PatLam  syn patInfo  :: NodeInfo Variable
```

Some auxiliary attributes are necessary to make the summary:

```
attr Term   syn var    :: Maybe Variable
attr TermL  syn vars   :: [Maybe Variable]
attr Var    syn tag    :: Tag
            syn var    :: Variable
attr VarL   syn hdTag  :: Tag
            syn vars   :: [Variable]
```

The semantic rules for these attributes are straightforward:

```
sem Term
| Tag        lhs.termInfo = InNode @tag []
| Var        lhs.termInfo = InVar  (nr @nm)
| Node       lhs.termInfo = InNode @tag @fldL.vars
sem PatAlt
| Tag        lhs.patInfo  = InNode @tag []
| Node       lhs.patInfo  = InNode @tag (map nr @fldL)
sem PatLam
```

```
| Empty     lhs.patInfo  = InVar  wildcard
| Var       lhs.patInfo  = InVar  (nr @nm)
| VarNode   lhs.patInfo  = InNode (@fldL.hdTag) (tail @fldL.vars)
sem Term
| Var       lhs.var      = Just   (nr @nm)
| * - Var   lhs.var      = Nothing
sem TermL
| Cons      lhs.vars     = @hd.var : @tl.vars
| Nil       lhs.vars     = []
sem VarL
| Cons      lhs.hdTag    = @hd.tag
sem VarL
| Cons      lhs.vars     = @hd.var : @tl.vars
| Nil       lhs.vars     = []
sem Var
| KnownTag lhs.tag       = @tag
| Var       lhs.var      = nr @nm
```

The *patInfo* attribute defined above determines the target of each expression. For most expressions, the target is the next pattern in the sequence. For the last expression in a sequence that is the body of a function, the target is the function name bound in a *Bind* binding, and passed all the way through the *Seq* spine. This is expressed in the following semantic rule:

```
attr AllExpr
  inh targetInfo :: NodeInfo Variable
sem Bind | Bind
  expr.targetInfo = InVar (nr @nm)
sem Expr | Seq
  expr.targetInfo = @pat.patInfo
  body.targetInfo = @lhs.targetInfo
```

The *termInfo* attribute defined earlier occurs in the semantics rules for various expression forms in figure 1. The *termInfo* attribute value synthesized by the scrutinee term of a *Case* expression is also needed in the alternatives of that *Case* expression. It is therefore passed down as an inherited attribute to the alternatives:

```
attr Alt AltL
  inh termInfo :: NodeInfo (Maybe Variable)
```

No explicit semantic rules are needed here, as the AG system automatically routes the value synthesized by the first child of a *Case* expression (the scrutinee) as the value of the inherited attribute with the same name of its second child (the list of alternatives).

We are now ready to discuss the twelve syntactic positions where equations originate, as defined in figure 1. In the case of a *Unit* or *UpdateUnit* we distinguish the four combinations of target pattern and source term (each variable or node). When both are variables, the target is constrained to hold a superset of the source; when the target is a variable and the source is a node, the target can hold that node. If the target is a node and the source is a variable, all the fields of the node that are not wildcards should be projections of the source variable. When both are nodes, their corresponding fields should be unified. For the last two cases we have auxiliary functions:

```
buildSelectEquations :: Variable → Tag → [Variable] → Equations
buildSelectEquations svar ttag tnms
  = [ IsSelection tvar svar i ttag
    | (tvar, i) ← zip tnms [0..]
    , tvar ≢ wildcard
    ]
```

```
sem Expr | Unit UpdateUnit
  loc.equations1 = case (@lhs.targetInfo, @val.termInfo) of
                     (InVar tvar          , InVar svar        ) → [IsSuperset tvar svar]
                     (InVar tvar          , InNode stag snms) → [IsConstruction tvar stag snms]
                     (InNode ttag tnms    , InVar svar        ) → buildSelectEquations svar ttag tnms
                     (InNode ttag tnms    , InNode stag snms) → buildUnifyEquations snms tnms
sem Expr | UpdateUnit
  loc.equations2 = [IsSuperset (nr @nm) (nr @val.getName)]
sem Expr | Unit
  lhs.equations  = @loc.equations1
sem Expr | UpdateUnit
  lhs.equations  = @loc.equations2 ++ @loc.equations1
sem Alt | Alt
  lhs.equations  = case (@pat.patInfo, @lhs.termInfo) of
                     (InNode ttag tnms, InVar svar) → buildSelectEquations svar ttag tnms
sem Expr | FetchNode
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar → [IsSuperset tvar       (nr @nm)]
sem Expr | FetchUpdate
  lhs.equations  =                  [IsSuperset (nr @dst) (nr @src) ]
sem Expr | FetchField
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar → [IsSelection tvar (nr @nm) @offset (fromJust @mbTag)]
sem Expr | Store
  lhs.location   = @lhs.location + 1
  lhs.heapEqs    = case @val.termInfo of
                     InNode stag snms → [ WillStore @lhs.location stag snms]
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar         → [IsKnown tvar (AbsLocs (Set.singleton @lhs.location))]
sem Global | Global
  lhs.location   = @lhs.location + 1
  lhs.heapEqs    = case @val.termInfo of
                     InNode stag snms → [ WillStore @lhs.location stag snms]
  lhs.equations  =                      [IsKnown (nr @nm) (AbsLocs (Set.singleton @lhs.location))]
sem Expr | Call
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar         → [IsSuperset tvar (nr @nm)]
                     InNode ttag tnms → buildSelectEquations (nr @nm) ttag tnms
sem Expr | FFI
  loc.nodemap    = Map.fromList ([(con, [AbsBasic | con == Tag_Unboxed]) | con ← @tagL])
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar         → [IsKnown tvar (AbsNodes @loc.nodemap)]
                     InNode ttag tnms → zipWith IsKnown tnms (fromJust (Map.lookup ttag @loc.nodemap))
sem Expr | Eval
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar → [IsEvaluation tvar (nr @nm)]
sem Expr | Apply
  lhs.equations  = case @lhs.targetInfo of
                     InVar tvar → [IsApplication (Just tvar) (nr @nm : @argL.varsInfo)]
```

Figure 1: Definition of constraint equations for various expression types (discussed in section 4.3)

Finally, when both target and source are full nodes, corresponding arguments should unify. This is handled by another auxiliary function:

```
buildUnifyEquations :: [Maybe Variable] → [Variable] → Equations
buildUnifyEquations snms tnms
  = [ case mbSvar of Nothing   → IsKnown   tvar AbsBasic
                     Just svar → IsSuperset tvar svar
    | (tvar, mbSvar) ← zip tnms snms
    , tvar ≢ wildcard
    ]
```

In the case of an *UpdateUnit* expression there is one more constraint, setting the destination variable of the update equal to that of the source variable. In the semantic rules, AG keyword **loc** is used to define a local attribute common to *Unit* and *UpdateUnit*. The situation arising from an alternative *Alt* in a *Case* expression is very much like the third subcase of a *Unit* expression: the fields of the target node (which come from the pattern in each alternative) are projections of the value of the scrutinee, that for this reason was (automatically!) passed down.

We now turn to the three variants of *Fetch* expressions. When a complete node is fetched, the target variable should be equal to the value fetched. For a *FetchNode* the target is the inherited target (i.e., the next *Seq* pattern or result of a function *Bind*ing), for a *FetchUpdate* the target is specified in the expression. In case of a *FetchField* of a single field, that field should be a projection from the source.

The next semantic rule, still in figure 1, states that for a *Store* expression we need a new uniquely numbered location. A heap equation is generated that states that this location indeed stores the value, and a normal equation is generated that states that the target variable is a pointer to this location.

The situation for a *Global* variable definition is quite the same, which is why we define these situations adjacently in figure 1 (the AG preprocessor allows to handle the cases *Expr* non-contiguously, which we happily use here to group similar rules).

In the case of a *Call* to a Grin function or an *FFI* call to a foreign function we distinguish the cases that the target is a variable or a complete node. The final two cases in figure 1 state that *Eval* and *Apply* expressions give rise to corresponding constraints.

What is not handled in the cases discussed above, is that actual parameters should agree to formal parameters. The *Call* expression handled in figure 1 only matched the result, not the arguments. Function calls can either occur directly in a *Call* expression, or implicitly in an *fpaNode*, that is a node with *Fun*, *PApp* or *App* (but not *Con*or one of the other special) tags.

In a tree walk we collect the relevant calls and tagged nodes. Conceptually this is a separate tree walk, but it is merged by the AG preprocessor with the tree walk defined earlier. We declare synthesized attributes to collect *allCalls* and *fpaNodes* for nearly all syntactic positions, because this must be passed all up the tree.

```
attr AllTerm AllExpr AllDef Module
  syn allCalls   use (++) [] :: [(Variable, [Maybe Variable])]
  syn fpaNodes use (++) [] :: [NodeInfo (Maybe Variable)]
```

Thanks to the **use** clause, we only need to specify the locations where calls and nodes are actually introduced:

```
sem Expr | Call
  lhs.allCalls   = [(nr @nm, @argL.vars)]
sem Term | Node
  lhs.fpaNodes = if     @tag.isfpa
                 then [InNode @tag @fldL.vars]
                 else []
```

An auxiliary attribute decides which nodes are relevant to collect:

```
attr Tag syn isfpa :: Bool
sem Tag
  | Fun PApp App      lhs.isfpa = True
  | Con Hole Unboxed lhs.isfpa = False
```

Now the final set of equations is the combination of constraints that were gathered in the tree walk (that is, the synthesized *equations* from the entire module *mod*), and those that arise from direct calls, *Fun*, *PApp* and *App* thunk nodes:

```
sem Program | Prog
  lhs.equations
    = @mod.equations
      ++ [ IsSuperset x y
         | (funnr, args) <- @mod.allCalls
         , (x, Just y) <- zip [funnr + 1 ..] args
         ]
      ++ [ IsSuperset x y
         | (InNode (Tag_Fun nm) args) <- @mod.fpaNodes
         , (x, Just y) <- zip [nr nm + 1 ..] args
         ]
      ++ [ IsSuperset x y
         | (InNode (Tag_PApp needs nm) args) <- @mod.fpaNodes
         , (x, Just y) <- zip [nr nm + 1 ..] args
         ]
      ++ [ IsApplication Nothing (map fromJust args)
         | (InNode Tag_App args) <- @mod.fpaNodes
         ]
```

Note that we exploit the fact that the function and its arguments are numbered consecutively: the arguments are numbered from one more than the function number onwards. Without this convention, the correspondence between the number of a function and those of its parameters could have been established as a mapping that could have been defined as yet another synthesized attribute of bindings.

The trickiest equations are generated in the fifth concatenated list: it states that the arguments of an *App* node represent an application, although it is not statically known where the result is stored.

## 4.4 Solving the constraint equations

Now we've collected all equations, we can proceed to solve them. The solution is computed in function *solveEquations*. It takes two integers: the number of *Variable*s and *Location*s, and the two lists of equations that were collected in the tree walk. These were determined in an earlier stage where variables are numbered (trivial, not shown in this paper), and as synthesized attribute *location* in the tree walk.

$$solveEquations :: Int \to Int \to Equations \to HeapEquations \to (AbstractEnv, AbstractHeap, Int)$$

The *solveEquations* function starts with creating two arrays, initially holding only *AbsBottom* values, to store the abstract values of all variables and locations, respectively. Then a fixpoint iteration is done, processing in each step all constraints from both sets of equations. The *fixpoint* function is parameterized not only by the two sets of equations, but also by two procedures that process an equation. These procedures call function *envChanges* or *heapChange* respectively, to obtain the changes on the variables or locations that need to be made. In the processing procedures,

```
envChanges :: Equation → AbstractEnv s → AbstractHeap s → ST s [(Variable, AbsValue)]
envChanges equat env heap
  = case equat of
      IsKnown          d av      → return [(d, av)]
      IsSuperset       d v       → do { av ← readArray env v
                                      ; return [(d, av)]
                                      }
      IsSelection      d v i t   → do { av ← readArray env v
                                      ; let res = absSelect av i t
                                      ; return [(d, res)]
                                      }
      IsConstruction   d t as    → do { vars ← mapM (maybe (return AbsBasic) (readArray env)) as
                                      ; let res = AbsNodes (Map.singleton t vars)
                                      ; return [(d, res)]
                                      }
      IsEvaluation     d v       → do { av  ← readArray env v
                                      ; res ← absDeref av
                                      ; return [(d, res)]
                                      }
      IsApplication mbd (f : as) → do { av     ← readArray env f
                                      ; absFun ← case mbd of Nothing → absDeref av
                                                             Just _  → return av
                                      ; absArgs ← mapM (readArray env) as
                                      ; (sfx, res) ← absCall absFun absArgs
                                      ; return $ (maybe id (λd → ((d, res):)) mbd) sfx
                                      }
    where
    absSelect av i t = case av of
                         AbsNodes ns → maybe AbsBottom (!!i) (Map.lookup t ns)
                         _           → av
    absDeref av      = case av of
                         AbsLocs ls  → do { vs ← mapM (readArray heap) (Set.toList ls)
                                          ; return (mconcat (map (filterNodes isFinalTag) vs))
                                          }
                         _           → return av
    absCall f args   = do { ts ← mapM addArgs (getNodes (filterNodes isPAppTag f))
                          ; let (sfxs, avs) = unzip ts
                          ; return (concat sfxs, mconcat avs)
                          }
      where addArgs (tag @(Tag_PApp needs nm), oldArgs)
              = do { let n      = length args
                         newtag = Tag_PApp (needs − n) nm
                         funnr  = nr nm
                         sfx    = zip [funnr + 1 + length oldArgs ..] args
                   ; res ← if n < needs
                           then return $ AbsNodes (Map.singleton newtag (oldArgs ++ args))
                           else readArray env funnr
                   ; return (sfx, res)
                   }
            getNodes av = case av of
                            AbsNodes n → Map.toAscList n
                            AbsBottom  → []
```

Figure 2: Selection of change candidates for the abstract environment during fixpoint iteration (discussed in section 4.4)

the change candidate(s) obtained (exactly one in the case of an *heapEquation*, possibly more in the case of an *Equation*) are fed into function *procChange* to apply the change.

```
solveEquations lenEnv lenHeap eqs1 eqs2
 = runST $
    do { env  ← newArray (0, lenEnv  − 1) AbsBottom
       ; heap ← newArray (0, lenHeap − 1) AbsBottom
       ; let procEnv equat
                = do { cs ← envChanges equat env heap
                     ; bs ← mapM (procChange env) cs
                     ; return (or bs)
                     }
             procHeap equat
                = do { cs ← heapChange equat env
                     ; b  ← procChange heap cs
                     ; return b
                     }
       ; count ← fixpoint eqs1 eqs2 procEnv procHeap
       ; return (env, heap, count)
       }
```

Function *procChange* can be generically used for either an environment variable or a heap location. This function only changes the array (environment or heap) when an element (variable or location) is actually changed, and returns a boolean that indicates whether there was a change. The fixpoint function uses the boolean returned by *procChange* to decide whether to stop or continue processing all equations again: as long as one of the equations results in a change, the iteration is continued.

```
procChange arr (i, v1) =
  do { v0 ← readArray arr i
     ; let v2      = v0 `mappend` v1
           changed = v0 ≢ v2
     ; when changed (writeArray arr i v2)
     ; return changed
     }
```

The fixpoint function uses these booleans to decide whether to stop or continue processing all equations again: as long as one of the equations results in a change, the iteration is continued.

```
fixpoint eqs1 eqs2 proc1 proc2
 = fix 0
    where fix count
             = do
                 { let step1 b i = proc1 i >>= return.(b ∨)
                 ; let step2 b i = proc2 i >>= return.(b ∨)
                 ; changes1 ← foldM step1 False eqs1
                 ; changes2 ← foldM step2 False eqs2
                 ; if   changes1 ∨ changes2
                   then fix (count + 1)
                   else return count
                 }
```

What remains to be done is to describe how change candidates are selected for each equation. This is implemented in function *heapChange* below and function *envChanges* in figure 2.

Function *heapChange* dissects an *HeapEquation*, that states that at some location a node with given tag and argument variables is stored. It returns that the abstract contents of the location can either be the abstract node constructed from the *tag* and the abstract value of its arguments,

or, if the tag is a *Tag_Fun* thunk, the result of the function (because after evaluation, the thunk is updated with the function result).

```
heapChange :: HeapEquation → AbstractEnv s → ST s (Location, AbsValue)
heapChange (WillStore locat tag args) env
 = do {let mbres    = tagFun tag
      ; absArgs     ← mapM getEnv args
      ; absRes      ← getEnv mbres
      ; let absNode = AbsNodes (Map.singleton tag absArgs)
      ; return (locat, absNode `mappend` absRes)
      }
      where
      tagFun (Tag_Fun nm) = Just (nr nm)
      tagFun _            = Nothing
      getEnv Nothing      = return AbsBottom
      getEnv (Just v)     = readArray env v
```

The changes to abstract variables that arise from processing an *Equation* are determined by function *envChanges*in figure 2, which we will now discuss. The function returns a list of changes, unlike function *heapChange* above, which returns only a single change. For five out of six possible equation types this list is a singleton, however. Only for the last case, multiple changes may arise from one equation. For the first equation type *IsKnown*, where a variable is known to be able to have some abstract value, the variable is simply paired with that abstract value to indicate a necessary change. For the second equation type *IsSuperset d v*, the current approximation of $v$ is looked up in the abstract environment, and designated as a needed change for $d$ as well. For an *IsSelection* equation, the variable $v$ is abstractly evaluated to obtain an abstract node. From that abstract node the desired field is selected. The case of an *IsConstruction* equation is similar to the *WillStore* heap equation discussed above, in that an abstract node is created from the known tag and the abstractly evaluated argument variables. The fifth equation type is *IsEvaluation d v*, which states that $d$ may hold the evaluation result of thunk nodes pointed to by $v$. Here, we first abstractly evaluate $v$ to obtain the abstract pointers. These pointers are then abstractly dereferenced, that is looked up in the abstract heap. This results in all abstract nodes the locations can point to. By the design of the processing of heap equations, this is not only the thunk node, but also the possible evaluation results of it. As the *IsEvaluation* equation is supposed to obtain the evaluation results only, the list of all abstract nodes the locations can point to is filtered such that only those with a final tag (like *Tag_Con*) remain, and those with thunk tag (like *Tag_Fun*) are discarded. The filtering is done by an auxiliary function:

```
filterNodes :: (Tag → Bool) → AbsValue → AbsValue
filterNodes p (AbsNodes nodes) = AbsNodes (Map.filterWithKey (const.p) nodes)
filterNodes p av               = av
isFinalTag, isPAppTag :: Tag → Bool
isFinalTag (Tag_Fun _)     = False
isFinalTag Tag_App         = False
isFinalTag _               = True
isPAppTag (Tag_PApp _ _) = True
isPAppTag _              = False
```

The last equation type in figure 2, *IsApplication*, is the trickiest. It was introduced in section 4.3 in two situations: (1) for every *App* expression in the Grin program (here the *Maybe Variable* destination is *Just* a variable name), and (2) for every constructed node in the Grin program with *App* tag, (here the destination is *Nothing*). Remember from section 4.2 that *IsApplication mbv* ($f$: *as*) means that $f$ is a variable which refers to a function which is applied to values referred to by variables *as* (and the result may be stored in variable $v$ if *mbv* is *Just* $v$).

Therefore, the first thing that needs to be done is to evaluate $f$ and *as* abstractly. If the equation

was introduced from situation (2), the function variable also needs to be dereferenced abstractly. This gives us an abstract function *absFun* and abstract arguments *absArgs*. Auxiliary function *absCall* now can abstractly apply the former to the latter.

Doing an abstract call amounts to filtering the partial-application thunk nodes from the possible nodes that can represent the function, and adding the extra arguments by way of function *addArgs*. If, after adding the new parameters, the function is still not fully saturated, a new abstract node is constructed, having a *PApp* tag with lower *needs* than the original one. If the function happens to be fully saturated, the possible results are read from the environment. The resulting nodes (either the newly constructed, or those read) is tupled with the destination variable to indicate a necessary change, at least in situation (1) where such a variable exists.

But there are other changes that need to be taken into account as well, coined "side effects" or *sfx* in the code. During the abstract call, new associations between arguments and formal parameters become manifest, that are not statically available in the equations. This is why the *absCall* and *addArgs* functions, in addition to the function result, also return changes that take care of new possible abstract values for argument variables. It is because of these side effects that *envChanges* sometimes returns more than one change.

# 5 Discussion and related work

Our implementation determines, through static analysis, for each variable an approximation of its runtime value. In particular, this reveals the possible functions a closure can represent. This information can be used to replace the indirect jump in a closure evaluation by a small case analysis. It is vital that the back end which translates the case distinction to machine code does not use a jump table to implement it, as that would waste all our efforts to avoid indirect jumps. Instead, the case analysis should be translated to repeated tests and conditional jumps.

Although branch prediction makes conditional jumps with known target less expensive than indirect jumps, it is still important to keep the number of jumps as low as possible. By using a binary search, $2^n$ cases can be distinguished with $n$ comparisons and jumps. Another approach would be to test for the most probable case first. If the occurence probabilities are known, we can organize the binary search in a Huffman tree, minimizing the expected number of comparisons. Determining which cases occur most frequently is hard to detect statically, as this depends on the dynamic behaviour of a program. A interesting optimization opportunity is to gather such information during a test run, and to use the profiling information in subsequent compilations.

Extensively used higher-order library functions, like *map*, can lead to a large number of cases to be distinguished. In these situations it can be worthwhile to compile multiple copies of the function, each with a limited number of different callers.

Although our algorithm makes it possible to avoid indirect jumps on closure evaluation, there is another cause of indirect jumps: returning from a function. An idea worth investigating is to try and prevent these as well using control flow analysis: if the number of possible callers is limited, instead of returning by indirectly jumping to the adress popped from the stack, we can do a direct jump based on a case analysis of possible callers.

We collect constraints on variables by means of a tree walk, which is implemented in an attribute grammar based formalism. We think this case study shows that it is useful to be able to define attributes separately. An alternative approach to collect information on a syntax tree is using ASF [5]. In comparison, the AG approach is lightweight, in that it relies on the underlying language for the definition of semantic rules. Yet another approach would be to provide combinators that manipulate attributes within the language, instead of as a preprocessor [11]. We think that describing a tree walk algorithm explicitly in terms of inherited and synthesized attributes helps a lot in clarifying the structure of the algorithm. As tree walks are abundant in compiler construction (the algorithm described is just one of many in our compiler), we think it is worthwhile to consistently use attribute grammar based tools.

The solution of the constraints involve a standard fixpoint iteration. Special is that during solving new constraints are derived and solved on the fly for variables that are not statically known when the contraints were collected.

### Acknowledgements

The authors thank Christof Douma for writing an initial version of the implementation described in this work.

## References

[1] R. Bird and O. de Moor. *The algebra of programming.* Prentice Hall, 1996.

[2] Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

[3] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis.* Chalmers University of Technology, 1999.

[4] Urban Boquist and Thomas Johnsson. The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In *IFL*, 1996.

[5] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[6] Atze Dijkstra. *Stepping through Haskell.* PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[7] R.J.M. Hughes. Super-combinators, a new implementation method for applicative languages. In *ACM conference on Lisp and functional programming*, 1982.

[8] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[9] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[10] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J.Funct.Progr.*, 16:415–449, 2006.

[11] Oege de Moor, Simon Peyton Jones, and Eric van Wyk. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering*, number 1799 in LNCS, pages 121–133. Springer-Verlag, 1999.

[12] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *J. Funct.Progr.*, 2:127–202, 1992.

[13] S. Doaitse Swierstra, P.R. Azero Alocer, and J. Saraiava. Designing and Implementing Combinator Languages. In *3rd Advanced Functional Programming*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.

[14] D.A. Turner. A new implementation technique for applicative languages. *Software practice and experience*, 9:31–49, 1979.