Abstract Interpretation of Functional Programs using an Attribute Grammar System

Jeroen Fokker and S. Doaitse Swierstra

Dept. of Information and Computing Sciences,
Utrecht University, The Netherlands

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

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, where the tag is actually a pointer to the code of the function [11,9]. Evaluating a closure now amounts to just calling that code. 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.

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](a few left out details can be found in an accompanying technical report [7]).
The implementation is presented by giving the actual code. We use a preprocessor for Haskell that enables us to use notions derived from the realm of attribute grammars [8]. This makes the code concise enough to present it (almost) in full. 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

Using higher order functions on lists, like *map*, *filter* and *foldr*, is a good way to abstract from common patterns in functional programs. 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]. 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.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell [12] 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. The abstract syntax of the language is defined in a **syntax** declaration, which is like a Haskell **data** declaration with named fields, without the braces and commas (see section 3 for an example). Constructor function names need not to be unique between types. 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", possibly for more than one data type. For example, we can declare that both statements and expressions need to synthesize bytecode as well as listings, and that expressions can be evaluated to integer values:

$$\begin{array}{llll} \textbf{attr}\ \mathit{Expr}\ \mathit{Stat} & \textbf{syn}\ \mathit{bytecode} :: [\mathit{Instr}] & \textbf{syn}\ \mathit{listing} :: \mathit{String} \\ \textbf{attr}\ \mathit{Expr} & \textbf{syn}\ \mathit{value} \quad :: \mathit{Int} & \end{array}$$

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. These definitions are known as "semantic rules", and start with keyword **sem**. An example is:

$$\begin{array}{lll} \textbf{sem}\ \mathit{Expr} & \mid \mathit{Const}\ \textbf{lhs}.\mathit{value} = & @\mathit{num} \\ & \mid \mathit{Add}\quad \textbf{lhs}.\mathit{value} = & @\mathit{left}.\mathit{value} + @\mathit{right}.\mathit{value} \end{array}$$

This states that the synthesized (left hand side) *value* attribute of a *Const*ant expression is just the contents of the *num* field, and that of an *Add*-expression

can be computed by adding the *value* attributes of its subtrees. 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 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 [8]). 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:

$\mathbf{type}\ \mathit{Env} = [(\mathit{String}, \mathit{Int})]$
$\mathbf{attr}\ \mathit{Expr}\ \mathbf{inh}\ \mathit{env} :: \mathit{Env}$
$\mathbf{sem}\ \mathit{Expr} \mid \mathit{Var}\ \mathbf{lhs}.\mathit{value} = \mathit{fromJust}\ (\mathit{lookup}\ @\mathbf{lhs}.\mathit{env}\ @\mathit{name})$

The preprocessor translates inherited attributes into extra parameters for the semantic functions in the algebra. This overcomes problem 4.
In many situations, **sem** rules only specify that attributes a tree node inherites should be passed unchanged to its children. To scrap the boilerplate expressing this, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. A similar automated copying is done for synthesized attributes passed up the tree. When more than one child offers a candidate to be copied, normally the rightmost one is taken, unless we specify to **use** an operator to combine several candidates:

$\mathbf{attr}\ \mathit{Expr}\ \mathit{Stat}\ \mathbf{syn}\ \mathit{listing}\ \mathbf{use}\ (+\!\!+)\ [\,]$

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 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 by means of **syntax** declarations for the AG preprocessor. We do not provide a concrete syntax for the language, as Grin programs are only an intermediate representation. We start with a definition of toplevel constructs. A program consists of a name, and a list of function bindings. Each binding binds a parameterized name to an expression.

$\mathbf{syntax}\ \mathit{Program} = \mathit{Prog}\ \mathit{nm} :: \mathit{Name}\quad \mathit{bindL}\quad :: \mathit{BindL}$
$\mathbf{syntax}\ \mathit{Bind} = \mathit{Bind}\ \mathit{nm} :: \mathit{Name}\quad \mathit{argNmL} :: [\mathit{Name}]\quad \mathit{expr} :: \mathit{Expr}$
$\mathbf{type}\ \mathit{BindL} = [\mathit{Bind}]$

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.

$$
\begin{array}{lllll}
\textbf{syntax}\ \mathit{Term} = & \mathit{LitInt} & \mathit{int} :: \mathit{Int} & \\
\quad\quad\quad\quad\quad\mid & \mathit{Tag} & \mathit{tag} :: \mathit{Tag} & \\
\quad\quad\quad\quad\quad\mid & \mathit{Node} & \mathit{tag} :: \mathit{Tag} & \mathit{fldL} :: \mathit{TermL} \\
\quad\quad\quad\quad\quad\mid & \mathit{Var} & \mathit{nm} :: \mathit{Name} & \\
\textbf{type}\ \mathit{TermL} = [\,\mathit{Term}\,] & & &
\end{array}
$$

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.
Four different tags are used to label nodes: *Con*, *Fun*, *PApp* and *App*. 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. For lazy calls to functions of which the name is not statically known, special thunk nodes are used with tag *App*. The first field of such node represents the function, the other fields the arguments to which the function is applied when the thunk is forced to evaluate.

$$
\begin{array}{llll}
\textbf{syntax}\ \mathit{Tag} = & \mathit{Con} & \mathit{nm} :: \mathit{Name} & \\
\quad\quad\quad\quad\mid & \mathit{Fun} & \mathit{nm} :: \mathit{Name} & \\
\quad\quad\quad\quad\mid & \mathit{PApp} & \mathit{needs} :: \mathit{Int} & \mathit{nm} :: \mathit{Name} \\
\quad\quad\quad\quad\mid & \mathit{App} & &
\end{array}
$$

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. Eight cases in the expression syntax are relevant for this paper:

$$
\begin{array}{lllll}
\textbf{syntax}\ \mathit{Expr} = & \mathit{Unit} & \mathit{val} :: \mathit{Term} & & \\
\quad\quad\quad\quad\mid & \mathit{Seq} & \mathit{expr} :: \mathit{Expr} & \mathit{pat} :: \mathit{PatLam} & \mathit{body} :: \mathit{Expr} \\
\quad\quad\quad\quad\mid & \mathit{Case} & \mathit{val} :: \mathit{Term} & \mathit{altL} :: \mathit{AltL} & \\
\quad\quad\quad\quad\mid & \mathit{Store} & \mathit{val} :: \mathit{Term} & & \\
\quad\quad\quad\quad\mid & \mathit{FetchUpdate} & \mathit{src} :: \mathit{Name} & \mathit{dst} :: \mathit{Name} & \\
\quad\quad\quad\quad\mid & \mathit{Call} & \mathit{nm} :: \mathit{Name} & \mathit{argL} :: \mathit{TermL} & \\
\quad\quad\quad\quad\mid & \mathit{Eval} & \mathit{nm} :: \mathit{Name} & & \\
\quad\quad\quad\quad\mid & \mathit{Apply} & \mathit{nm} :: \mathit{Name} & \mathit{argL} :: \mathit{TermL} & \\
\quad\quad\quad\quad\mid & \ldots & & &
\end{array}
$$

We give an informal description of the semantics of these constructs, that is their runtime evaluation result and side effects on the heap. 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. 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. A pattern in a case alternative is a node with a known tag and names as arguments. A pattern in a *Seq* expression is quite different: it can be *Empty*, to be able to match the empty result value of the *FetchUpdate* expression, or just a variable name.

$$
\begin{array}{lllll}
\textbf{type}\ \mathit{AltL} & = [\mathit{Alt}] & & & \\
\textbf{syntax}\ \mathit{Alt} & = \mathit{Alt} & \mathit{pat} :: \mathit{PatAlt} & \mathit{expr} :: \mathit{Expr} & \\
\textbf{syntax}\ \mathit{PatAlt} & = \mathit{Node} & \mathit{tag} :: \mathit{Tag} & \mathit{fldL} :: [\mathit{Name}] & \\
\textbf{syntax}\ \mathit{PatLam} & = \mathit{Empty} & & & \\
 & \mid\ \mathit{Var} & \mathit{nm} :: \mathit{Name} & &
\end{array}
$$

Two constructs 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 *FetchUpdate*, which copies the contents of a heap location to another location, and returns the empty value. Next, we have *Call* for calling a Grin 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.

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

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. Our abstract interpretation algorithm also determines, for each *Store* expression, what type of node it can create. The abstraction of all heap cells that a particular *Store*-expression creates is known as a *Location*. In our implementation we identify locations by unique, consecutive numbers. Similarly, *Variable* is also an alias for *Int*, and we assume a function $nr :: Name \rightarrow Variable$.

We introduce a data type *AbsValue* to describe the domain in the abstract interpretation, 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. As for concrete values, 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.

```
instance Monoid AbsValue where
  mempty                                 = AbsBottom
  mappend av             AbsBottom       = av
  mappend AbsBottom      bv              = bv
  mappend AbsBasic       AbsBasic        = AbsBasic
  mappend (AbsTags ats)  (AbsTags bts)   = AbsTags (Set.union ats bts)
  mappend (AbsLocs als)  (AbsLocs bls)   = AbsLocs (Set.union als bls)
  mappend (AbsNodes am) (AbsNodes bm)
            = AbsNodes (Map.unionWith (zipWith mappend) am bm)
  mappend _              _               = AbsError "conflict"
```

The goal of the abstract interpretation algorithm is to determine the abstract value of each *Variable* and *Location*, which we collect in mutable arrays:

```
type AbsEnv s  = STArray s Variable AbsValue
type AbsHeap s = STArray s Location AbsValue
```

4.2 A constraint language

By observing a Grin program, we can deduce equations to constrain variables and locations. 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.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Equation} & = \mathit{IsKnown} & \mathit{Variable}\ \mathit{AbsValue} \\
 & \mid\ \mathit{IsSuperset} & \mathit{Variable}\ \mathit{Variable} \\
 & \mid\ \mathit{IsSelection} & \mathit{Variable}\ \mathit{Variable}\ \mathit{Int}\ \mathit{Tag} \\
 & \mid\ \mathit{IsConstruct} & \mathit{Variable}\ \mathit{Tag}\ [\mathit{Maybe}\ \mathit{Variable}] \\
 & \mid\ \mathit{IsEval} & \mathit{Variable}\ \mathit{Variable} \\
 & \mid\ \mathit{IsApply} & (\mathit{Maybe}\ \mathit{Variable})\ [\mathit{Variable}] \\
\textbf{data}\ \mathit{HeapEquation} & = \mathit{WillStore} & \mathit{Location}\ \mathit{Tag}\ [\mathit{Maybe}\ \mathit{Variable}] \\
\textbf{type}\ \mathit{Equations} & = [\mathit{Equation}] & \\
\textbf{type}\ \mathit{HeapEquations} & = [\mathit{HeapEquation}] &
\end{array}
$$

A variable may be constrained by more than one equation. These equations are cumulative. The semantics of the equations will be discussed in section 4.4.

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 *eqs* and *hEqs* stating the constraints for program variables and locations, respectively. Equations are collected for the whole *Program* but also for many substructures.

$$
\begin{array}{l}
\textbf{attr}\ \mathit{Program}\ \mathit{Bind}\ \mathit{BindL}\ \mathit{Expr}\ \mathit{Alt}\ \mathit{AltL} \\
\quad \textbf{syn}\ \mathit{eqs}\quad\ \textbf{use}\ (+\!\!+)\ [\,]\ ::\ \mathit{Equations} \\
\quad \textbf{syn}\ \mathit{hEqs}\ \textbf{use}\ (+\!\!+)\ [\,]\ ::\ \mathit{HeapEquations}
\end{array}
$$

We declare a few auxiliary attributes that collect information about nodes, viz. whether they reside in a variable or are given explicitly. The definition of these attributes is straightforward; only the nontrivial cases are given here.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{NodeInfo}\ a = \mathit{IV}\ \ \mathit{Variable} \mid \mathit{INd}\ \mathit{Tag}\ [\,a\,]} \\
\textbf{attr}\ \mathit{Term} & \textbf{syn}\ \mathit{termInfo} & ::\ \mathit{NodeInfo}\ (\mathit{Maybe}\ \mathit{Variable}) \\
\textbf{attr}\ \mathit{Alt}\ \mathit{AltL} & \textbf{inh}\ \mathit{termInfo} & ::\ \mathit{NodeInfo}\ (\mathit{Maybe}\ \mathit{Variable}) \\
\textbf{attr}\ \mathit{PatAlt}\ \mathit{PatLam} & \textbf{syn}\ \mathit{patInfo} & ::\ \mathit{NodeInfo}\ \mathit{Variable} \\
\textbf{attr}\ \mathit{Expr}\ \mathit{Alt}\ \mathit{AltL} & \textbf{inh}\ \mathit{targetInfo} & ::\ \mathit{NodeInfo}\ \mathit{Variable} \\
\textbf{sem}\ \mathit{Bind} \mid \mathit{Bind} & \multicolumn{2}{l}{\mathit{expr}.\mathit{targetInfo} = \mathit{IV}\ (\mathit{nr}\ @\mathit{nm})} \\
\textbf{sem}\ \mathit{Expr} \mid \mathit{Seq} & \multicolumn{2}{l}{\mathit{expr}.\mathit{targetInfo} = @\mathit{pat}.\mathit{patInfo}} \\
 & \multicolumn{2}{l}{\mathit{body}.\mathit{targetInfo} = @\textbf{lhs}.\mathit{targetInfo}}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{Unit}\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ (@\textbf{lhs}.\mathit{targetInfo},\ @\mathit{val}.\mathit{termInfo})\ \textbf{of}\\
\qquad\qquad (\mathit{IV}\ \ \mathit{tv}\ \ \ \ ,\mathit{IV}\ \ \mathit{sv}\ \ \ \ ) \rightarrow [\mathit{IsSuperset}\ \mathit{tv}\ \mathit{sv}]\\
\qquad\qquad (\mathit{IV}\ \ \mathit{tv}\ \ \ \ ,\mathit{INd}\ \mathit{st}\ \mathit{sns}) \rightarrow [\mathit{IsConstruct}\ \mathit{tv}\ \mathit{st}\ \mathit{sns}]\\
\qquad\qquad (\mathit{INd}\ \mathit{tt}\ \mathit{tns},\mathit{IV}\ \ \mathit{sv}\ \ \ \ ) \rightarrow \mathit{mkSelEqs}\ \mathit{sv}\ \mathit{tt}\ \mathit{tns}\\
\qquad\qquad (\mathit{INd}\ \mathit{tt}\ \mathit{tns},\mathit{INd}\ \mathit{st}\ \mathit{sns}) \rightarrow \mathit{mkUnifyEqs}\ \mathit{sns}\ \mathit{tns}\\
\textbf{sem}\ \mathit{Alt}\quad \mid\ \mathit{Alt}\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ (@\mathit{pat}.\mathit{patInfo},\ @\textbf{lhs}.\mathit{termInfo})\ \textbf{of}\\
\qquad\qquad (\mathit{INd}\ \mathit{tt}\ \mathit{tns},\mathit{IV}\ \ \mathit{sv}\ \ \ \ ) \rightarrow \mathit{mkSelEqs}\ \mathit{sv}\ \mathit{tt}\ \mathit{tns}\\
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{Store}\\
\quad \textbf{lhs}.\mathit{locnr} = @\textbf{lhs}.\mathit{locnr} + 1\\
\quad \textbf{lhs}.\mathit{hEqs} = \textbf{case}\ @\mathit{val}.\mathit{termInfo}\ \textbf{of}\\
\qquad\qquad \mathit{INd}\ \mathit{st}\ \mathit{sns} \rightarrow [\mathit{WillStore}\ @\textbf{lhs}.\mathit{locnr}\ \mathit{st}\ \mathit{sns}]\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ @\textbf{lhs}.\mathit{targetInfo}\ \textbf{of}\\
\qquad\qquad \mathit{IV}\ \mathit{tv} \rightarrow [\mathit{IsKnown}\ \mathit{tv}\ (\mathit{AbsLocs}\ (\mathit{Set.singleton}\ @\textbf{lhs}.\mathit{locnr}))]\\
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{FetchUpdate}\\
\quad \textbf{lhs}.\mathit{eqs} = \qquad\qquad [\mathit{IsSuperset}\ (\mathit{nr}\ @\mathit{dst})\ (\mathit{nr}\ @\mathit{src})]\\
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{Call}\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ @\textbf{lhs}.\mathit{targetInfo}\ \textbf{of}\\
\qquad\qquad \mathit{IV}\ \mathit{tv}\ \ \ \ \ \ \rightarrow [\mathit{IsSuperset}\ \mathit{tv}\ (\mathit{nr}\ @\mathit{nm})]\\
\qquad\qquad \mathit{INd}\ \mathit{tt}\ \mathit{tns} \rightarrow \mathit{mkSelEqs}\ (\mathit{nr}\ @\mathit{nm})\ \mathit{tt}\ \mathit{tns}\\
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{Eval}\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ @\textbf{lhs}.\mathit{targetInfo}\ \textbf{of}\\
\qquad\qquad \mathit{IV}\ \mathit{tv} \rightarrow [\mathit{IsEval}\ \mathit{tv}\ (\mathit{nr}\ @\mathit{nm})]\\
\textbf{sem}\ \mathit{Expr}\quad \mid\ \mathit{Apply}\\
\quad \textbf{lhs}.\mathit{eqs} = \textbf{case}\ @\textbf{lhs}.\mathit{targetInfo}\ \textbf{of}\\
\qquad\qquad \mathit{IV}\ \mathit{tv} \rightarrow [\mathit{IsApply}\ (\mathit{Just}\ \mathit{tv})\ (\mathit{nr}\ @\mathit{nm} : @\mathit{argL}.\mathit{varsInfo})]
\end{array}
$$

In the case of a *Unit* 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 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:

$$
\begin{array}{l}
\mathit{mkSelectEqs} :: \mathit{Variable} \rightarrow \mathit{Tag} \rightarrow [\mathit{Variable}] \rightarrow \mathit{Equations}\\
\mathit{mkSelectEqs}\ \mathit{sv}\ \mathit{tt}\ \mathit{tns}\\
\quad = [\mathit{IsSelection}\ \mathit{tv}\ \mathit{sv}\ \mathit{i}\ \mathit{tt} \mid (\mathit{tv}, \mathit{i}) \leftarrow \mathit{zip}\ \mathit{tns}\ [0\,..]\ ]\\
\mathit{mkUnifyEqs} :: [\mathit{Maybe}\ \mathit{Variable}] \rightarrow [\mathit{Variable}] \rightarrow \mathit{Equations}\\
\mathit{mkUnifyEqs}\ \mathit{sns}\ \mathit{tns}\\
\quad = [\ \textbf{case}\ \mathit{mbSvar}\ \textbf{of}\ \mathit{Nothing} \rightarrow \mathit{IsKnown}\ \ \mathit{tv}\ \mathit{AbsBasic}\\
\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{sv}\ \ \rightarrow \mathit{IsSuperset}\ \mathit{tv}\ \mathit{sv}\\
\qquad \mid (\mathit{tv}, \mathit{mbSvar}) \leftarrow \mathit{zip}\ \mathit{tns}\ \mathit{sns}\ ]
\end{array}
$$

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.
For a *Store* expression we generate a new uniquely numbered location, and a heap equation that associates it with the stored value. A normal equation equates the target variable to a pointer to the new location. The destination heap location that is updated by *FetchUpdate* can at least take all the values of the source location. In the case of a *Call* to a function we distinguish the cases that the target is a variable or a complete node. The final two cases 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. 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*) 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. Thanks to the **use** clause, we only need to specify the locations where calls and nodes are actually introduced:

$$
\begin{array}{l}
\textbf{attr}\ \mathit{Bind\ BindL\ Expr\ Alt\ AltL\ Term\ TermL}\\
\quad \textbf{syn}\ \mathit{allCalls}\quad \textbf{use}\ (+\!\!+)\ [\,] :: [(\mathit{Variable}, [\mathit{Maybe\ Variable}])]\\
\quad \textbf{syn}\ \mathit{fpaNodes}\ \textbf{use}\ (+\!\!+)\ [\,] :: [(\mathit{Tag}, \qquad [\mathit{Maybe\ Variable}])]\\
\textbf{sem}\ \mathit{Expr}\ \mid \mathit{Call}\ \ \textbf{lhs}.\mathit{allCalls}\ \ = [(\mathit{nr}\ @\mathit{nm}, @\mathit{argL.vars})]\\
\textbf{sem}\ \mathit{Term} \mid \mathit{Node}\ \textbf{lhs}.\mathit{fpaNodes} = [(@\mathit{tag}, @\mathit{fldL.vars}) \mid @\mathit{tag.isfpa}]
\end{array}
$$

Now the final set of equations is the combination of constraints that were gathered in the tree walk (that is, the synthesized *eqs* from all bindings), and those that arise from direct calls and *fpaNodes*. Note that we exploit the fact that the function and its arguments are numbered consecutively, from one more than the function number onwards.

$$
\begin{array}{l}
\textbf{sem}\ \mathit{Program} \mid \mathit{Prog}\\
\quad \textbf{lhs}.\mathit{eqs} = @\mathit{bindL.eqs}\\
\qquad\qquad +\!\!+ [\mathit{IsSuperset}\ x\ y \mid (\mathit{funnr}, \mathit{args}) \leftarrow @\mathit{bindL.allCalls}\\
\qquad\qquad\qquad\qquad\qquad\qquad , (x, \mathit{Just}\ y) \leftarrow \mathit{zip}\ [\mathit{funnr} + 1\,..]\ \mathit{args}]\\
\qquad\qquad +\!\!+ [\mathit{IsSuperset}\ x\ y \mid (\mathit{Tag\_Fun}\ \mathit{nm}, \mathit{args}) \leftarrow @\mathit{bindL.fpaNodes}\\
\qquad\qquad\qquad\qquad\qquad\qquad , (x, \mathit{Just}\ y) \leftarrow \mathit{zip}\ [\mathit{nr}\ \mathit{nm} + 1\,..]\ \mathit{args}]\\
\qquad\qquad +\!\!+ [\mathit{IsSuperset}\ x\ y \mid (\mathit{Tag\_PApp}\ \_\ \mathit{nm}, \mathit{args}) \leftarrow @\mathit{bindL.fpaNodes}\\
\qquad\qquad\qquad\qquad\qquad\qquad , (x, \mathit{Just}\ y) \leftarrow \mathit{zip}\ [\mathit{nr}\ \mathit{nm} + 1\,..]\ \mathit{args}]\\
\qquad\qquad +\!\!+ [\mathit{IsApply}\ \mathit{Nothing}\ (\mathit{map}\ \mathit{fromJust}\ \mathit{args})\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid (\mathit{Tag\_App}, \mathit{args}) \leftarrow @\mathit{bindL.fpaNodes}]
\end{array}
$$

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 the number of *Variable*s and *Location*s, and the two lists of equations that were collected in the tree walk.

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

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, the change candidate(s) obtained are fed into function *procChange* to apply the change.

```
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                    }
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                     }
```

Function *procChange* can be used for either an environment variable or a heap location. This function only changes the array when an element is actually changed, and returns a boolean that indicates whether there was a change. The fixpoint function uses that boolean to decide whether to stop: as long as one of the equations results in a change, the iteration is continued. What remains to be done is to describe how change candidates are selected for each equation. 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 → AbsEnv s → ST s (Location, AbsValue)
heapChange (WillStore locat tag args) env
 = do { absArgs ← mapM getEnv args
      ; absRes  ← getEnv (tagFun tag)
      ; return (locat, absNode tag absArgs `mappend` absRes)
      } where getEnv = maybe (return AbsBottom) (readArray env)
              tagFun (Tag_Fun nm) = Just (nr nm)
              tagFun _            = Nothing
absNode t as = AbsNodes (Map.singleton tag absArgs)
```

The changes to abstract variables that arise from processing an *Equation* are determined by function *envChanges*. 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.

```
envChanges :: Equation → AbsEnv s → AbsHeap 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
                              ; return [(d, absSelect av i t)]      }
      IsConstruct d t as → do { vars ← mapM (maybe (return AbsBasic)
                                                  (readArray env))
                                            as
                              ; return [(d, absNode t vars)]        }
      IsEval d v         → do { av  ← readArray env v
                              ; res ← absDeref av
                              ; return [(d, res)]                   }
```

```
IsApply 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    }
```

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 *IsConstruct* 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. We have local auxiliaries that do selection and dereferencing in the abstract world:

```
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
```

The fifth equation type is *IsEval 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 *IsEval* 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 last equation type *IsApply*, 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 *IsApply 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.

```
absCall f args
  = do  { ts ← mapM addArgs (getNodes (filterNodes isPAppTag f))
        ; let (sfxs, avs) = unzip ts
        ; return (concat sfxs, mconcat avs)  }
  where getNodes av = case av of AbsNodes n → Map.toAscList n
                                 AbsBottom  → []
        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 $ absNode newtag (oldArgs ++ args)
                        else  readArray env funnr
               ; return (sfx, res)
               }
```

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. 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 [10].
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.

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] Jeroen Fokker and S. Doaitse Swierstra. Abstract interpretation of functional programs. Technical report, www.cs.uu.nl/research/techreps/UU-CS-2007-049.html.

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

[9] 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.

[10] 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.

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

[12] 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.