Higher-Order Redundancy Elimination

Peter Thiemann
Wilhelm-Schickard-Institut, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
Email address: thiemann@informatik.uni-tuebingen.de

Abstract

Functional programs often define functions by pattern matching where patterns may inadvertedly overlap through successive function calls. This leads to hidden inefficiencies since the recursively called function possibly repeats redundant tests while trying to match the pattern. An analysis which is based on conservative symbolic execution (similar to higher order constant propagation) is proposed for a strict higher-order language to drive an arity raiser which generates specialized versions for functions with partially known arguments. To ensure termination only the definitely consumed part of the partially known arguments is considered.

1 Introduction

Pattern matching is ubiquitous in modern functional programming languages like ML or Haskell. It is a convenient tool to build readable programs that process algebraic datatypes. However, pattern matching is a high-level concept that the compiler must transform into sequences of test operations. Patterns must be unnested to yield flat patterns which can then be implemented as constructor tests and selector operations. Methods to achieve this are well known [Aug85].

Nested patterns are often a subtle source of inefficiency. In their presence it is quite often the case that function arguments are partially known so that the patterns of successive calls overlap. But overlapping patterns cause redundant constructor tests for the overlapping part. For a structure of size n this can amount to $O(n)$ tests the outcome of which is known in advance.

We solve this annoying problem for a strict higher-order language. First, we define a safe notion of symbolic execution and specify a consumption analysis which determines the amount of scrutiny performed on the arguments of a function. Using these two analyses we obtain a complete set of call patterns which are used to guide a specializer. In order to take advantage of partially known structures the specializer uses the control flow to memorize data constructors that are already tested and decomposed. The arguments of the remembered constructor are added to the function's argument list (this technique is called *arity raising* [Rom90]) and the structure remains unallocated.

On the machine level functional values (partial applications) are represented as closures, *i.e.* tuples consisting of a code address and the values of some arguments. Thus, the code address (the name of the function) is considered a data constructor and closures are treated in a similar way as constructed data. The difference lies in their consumption. While constructed data is consumed by being subject of a `case` expression, closures are consumed by becoming saturated applications. If a closure is definitely consumed it is not allocated but instead its contents are passed along as additional parameters. Thus data construction can sometimes be avoided.

Although conceived and presented for a strict language the proposed method can also improve the performance of programs in lazy languages. In the implementation of a non-strict language a constructor test means not just one test, but two tests. The first test determines if the argument is a suspension (a representation of an unevaluated expression) and starts its evaluation if that is the case. The second test is the actual constructor test. In an overlapping situation as outlined above both tests are repeated although their result is known.

In the next section we give some examples for our method at work. Section /refsec:analysis first introduces our example language and gives an instrumented semantics for it. Later on, a symbolic evaluation function is abstracted from that semantics. A consumptions analysis which uncoveres scrutinized parts of values along with a description how call patterns are collected and pruned completes the range of analyses needed. Section 4 describes how to apply the results of the analyses. Section 5 discusses related work and Section 6 gives some conclusions and discusses further work.

2 Examples

The following examples demonstrate that the proposed method applies to real programs that occur in practice. We make use of a subset of ML [MTH90] which is formally defined later on in Fig. 1. We take the liberty of using list brackets and the infix list constructor `::` so as to improve readability.

2.1 Mergesort

Consider the following part of a program which sorts a list by merging sorted sublists. The sublists are constructed by `deco xs` which creates the lists of the even and odd numbered elements from list `xs`.

```
sort [ ] = [ ]
sort [x] = [x]
sort xs  = let (xs1, xs2) = deco xs in
```

```
            merge (sort xs1, sort xs2)

deco [ ]     = ([ ], [ ])
deco (x::xs) = let (dx, dy) = deco xs in
                   (x::dy, dx)
```

Analysis yields that deco is called from sort only with lists that have at least two elements. The call to deco returns a pair of non-empty lists. Hence the recursive calls of sort all have a non-empty list as a parameter. Our method first yields the following specialized version deco2 of deco derived from the call deco (x1::x2::xs).

```
deco2 (x1, x2, xs) = let (dx, dy) = deco (x2::xs) in
                         (x1::dy, dx)
```

deco2 still has a call of deco with a non-empty list parameter. Hence the version deco1 of deco is generated and deco2 is modified to call deco1.

```
deco1 (x1, xs)     = let (dx, dy) = deco xs in
                         (x1::dy, dx)
deco2 (x1, x2, xs) = let (dx, dy) = deco1 (x2, xs) in
                         (x1::dy, dx)
```

Since deco1 is only called once in the program in can safely be unfolded into deco2.

```
deco2 (x1, x2, xs) = let (dx', dy') =
                     let (dx, dy) = deco xs in
                         (x2::dy, dx)
                     in  (x1::dy', dx')
```

Simplification results in

```
deco2 (x1, x2, xs) = let (dx, dy) = deco xs in
                         (x1::dx, x2::dy)
```

Since the result of deco2 is known to be a pair of non-empty lists and those lists are immediately consumed by the function calls sort xs1 and sort xs2, deco2 is unfolded into the third equation of sort.

```
sort  [ ] = [ ]
sort  [x] = [x]
sort  (x1::x2::xs) =
        let (xs1, xs2) = deco2 (x1, x2, xs) in
            merge (sort xs1, sort xs2)
```

Now deco2 can be eliminated by unfolding its only call.

```
sort  [ ] = [ ]
sort  [x] = [x]
sort  (x1::x2::xs) = let (xs1, xs2) =
                     let (dx, dy) = deco xs in
                         (x1::dx, x2::dy)
                     in  merge (sort xs1, sort xs2)
```

Propagation of the outermost let-binding and the subsequent introduction of a specialized sort1 for sort with non-empty parameter list leads to the following program:

```
sort  [ ] = [ ]
sort  [x] = [x]
sort  (x1::x2::xs) = let (dx, dy) = deco xs in
                        merge (sort1 (x1, dx),
                               sort1 (x2, dy))
sort1 (x1, []) = [x1]
sort1 (x1, x2::xs) = let (dx, dy) = deco xs in
                        merge (sort1 (x1, dx),
                               sort1 (x2, dy))
```

According to our analysis, all functions in the above program are called with parameters of unknown shape. The analysis is not perfect: closer inspection reveals that sort1 always returns a non-empty list as well as merge produces non-empty lists when at least one of its inputs is non-empty. It is, however, not clear how a specializer could take advantage of that fact, since the head element of sort1 (x1, dx) might by any element of the list x1::dx.

Notice that the function merge itself makes for a nice example, too. We are grateful to a referee for pointing this out. Consider its implementation:

```
merge (xs as xh::xt, ys as yh::yt) =
        if xh <= yh then xh::merge (xt, ys)
                    else yh::merge (xs, yt)
merge ([], ys) = ys
merge (xs, []) = xs
```

In the first recursive call to merge the argument ys is known to be a non-empty list, while in the second call the argument xs is known to be a non-empty list. Applying our method yields three mutually recursive functions with essentially identical bodies (the branches dealing with empty lists have been omitted for brevity).

```
merge  (xs as xh::xt, ys as yh::yt) =
    if xh <= yh then xh::merge1 (xt, yh, yt)
                else yh::merge2 (xh, xt, yt)
merge1 (xs as xh::xt,       yh, yt) = ...
merge2 (      xh, xt, ys as yh::yt) = ...
```

When we apply these ideas in the context of the lazy functional programming language Haskell [Has92], our experiments with the Chalmers Haskell compiler (SPARC version 0.999.4) reveal a speedup of about 10% for the function merge alone. The functions have been transcribed literally but changed to using curried functions in place of argument tuples. This choice turns out to be more effective for a lazy language.

2.2 Binary Tree Traversal

The second example shows that redundant tests not only occur with lists but also with other algebraic datatypes. Consider making a list of the node labels of a binary tree in left-to-right order.

```
data IntTree = Empty | Node (IntTree, Int, IntTree)

inorder Empty = [ ]
inorder (Node (Empty, a, r)) = a::inorder r
inorder (Node (Node (l, a, r), b, r')) =
        inorder (Node (l, a, Node (r, b, r')))
```

Though the above definition can do its job using constant memory lots of tests are redundant due to the nested patterns and the data constructions in the argument position of the function calls. Our method generates the following specialized version of the function inorder.

```
inorder_111 (Empty, a, r) =
        a:: inorder r
inorder_111 (Node (l, a, r), b, r') =
        inorder_111 (l, a, Node (r, b, r'))
```

To come into effect the last equation for inorder must be changed as follows:

```
inorder (Node (Node (l, a, r), b, r')) =
        inorder_111 (l, a, Node (r, b, r'))
```

This innocuous change spares a constructor test at every `Empty` node of the tree. Since there are up to $n/2$ of them in a binary tree of size n the amount is considerable. A test with the Chalmers Haskell compiler reveals that the specialized version executes up to 24% faster where the input is a tree of depth 18.

3 Analysis

In the following we consider a strict higher order language with algebraic datatypes and a monomorphic type discipline. It is a typical example for a core language which arises in a compiler after removal of syntactic sugar. Its abstract syntax is defined in Fig. 1 assuming disjoint denumerable sets Var of variables, Kon of constant symbols, Con of data constructors for algebraic datatypes with arities $k(c)$, Fun of function symbols, and TN of type names including a set B of base types (*i.e.*, for integers). The operator $\_^*$ denotes zero or more repetitions of the respective syntactic entity. A program (*prg*) consists of some algebraic datatype declarations (*tdec*) followed by some function declarations (*dec*) and an expression (*exp*). An algebraic datatype declaration introduces a recursive sum-of-product type by listing all of its constructors and their types. Although types are not explicitly mentioned in the syntax we assume all expressions well-typed with a monomorphic typing discipline (see *i.e.*, [Mit90]). We will only make use of type information to make the distinction between base types, algebraic types, and function types. An expression is either a variable, a constant of some base type, the name of a defined function, an application, a (saturated) constructor application, a `let` expression, or a `case` expression, which performs a multi-way branch on values of an algebraic datatype. Call the decomposed variable v the *subject* and the expressions $exp_1, \ldots, exp_m$ the *branches* of a `case` expression. Notice that a `case` expression only matches flat patterns (a constructor applied to variables) and that patterns are assumed to be exhaustive, *i.e.* if the subject has type then there must be a branch for every constructor of t.

3.1 Instrumented Semantics

To make precise the operational notions of the language we give a denotational semantics which explicitly manipulates a heap to construct values of algebraic datatypes. A heap maps memory locations to contents which will be specified later on. The meaning of an expression is a transformation of the heap.

$$\begin{array}{rcl}
\mathsf{Heap} & = & \mathsf{Loc} \to \mathsf{Contents}_\perp \\
\mathsf{Loc} & = & \{ \text{ some unspecified infinite set of } \textit{locations} \ \}
\end{array}$$

Here and in the following $A_\perp$ denotes the partial order obtained by lifting A upon a new bottom element $\perp$, A^* denotes the set of finite sequences over A, and $\mathcal{P}(A)$ is the powerset of A.

To ease reading the semantic equations we will use a comprehension notation for heap transformations (cf. the monad of state transformers [Wad90a]). A heap transformation is a function of type $\mathsf{HST}(x) = \mathsf{Heap} \to x \times \mathsf{Heap}$. It accepts a heap and returns a result of type x paired with a (modified) heap. A comprehension $[e \mid q]$ consists of a body e and qualifier list q. An empty qualifier list maps the value e into a heap transformation which yields e and leaves the heap alone. The atomic qualifiers $v \leftarrow m$ and m create heap transformations which first execute m and then either bind the result to v in the remaining qualifier list and in e or (for m) they discard the result. Comprehension notation is defined as follows:

$$\begin{array}{ll}
[x \mid] & = \lambda h.(x, h) \\
[x \mid v \leftarrow m, ms] & = \lambda h.\mathbf{let}\ (v, h') = m\ h\ \mathbf{in}\ [x \mid ms]h' \\
[x \mid m, ms] & = \lambda h.\mathbf{let}\ (_, h') = m\ h\ \mathbf{in}\ [x \mid ms]h'
\end{array}$$

The remaining semantic domains are summarized in the table below. Let CSite be an infinite set of *program points*. We will assume every subexpression of a program to be uniquely marked with some $u \in \mathsf{CSite}$. Program points will be used to identify function call sites, data creation sites, and data consumption sites where constructed data is decomposed. Prefix is used to distinguish variable occurrences and creation sites within different function activations as manifested in PfxVar. Base provides a set of values for the interpretation of constants by the function $\kappa\colon \mathsf{Kon} \to \mathsf{Base}$. The symbol $\dot\cup$ denotes disjoint set union.

$$\begin{array}{rcl}
\mathsf{Prefix} & = & \mathsf{CSite}^* \\
\mathsf{PfxVar} & = & \mathsf{Prefix} \times (\mathsf{Var} \mathbin{\dot\cup} \mathsf{CSite}) \\
\mathsf{Base} & = & \{\text{some unspecified set of base values}\} \\
\mathsf{Val}^I & = & \mathsf{Loc} \mathbin{\dot\cup} \mathsf{Base} \\
\mathsf{Env}^I & = & \mathsf{Var} \to \mathsf{Val}^I_\perp \\
\mathsf{FEnv}^I & = & \mathsf{Fun} \to (\mathsf{Prefix} \times \mathsf{Val}^{I*})_\perp \to \mathsf{HST}(\mathsf{Val}^I)
\end{array}$$

Now we can characterize the contents of a heap cell being objects of type $\mathsf{Contents}$. Basically, a heap cell stores a closure with a variable number of arguments representing a partial application or a node of a constructed datatype, $(\mathsf{Con} \mathbin{\dot\cup} \mathsf{Fun}) \times \mathsf{Val}^{I*}$. Additionally it records the node's creation and its bindings to variables (PfxVar) and consumption sites ($\mathsf{PfxVar} \times \mathsf{CSite}$).

$$\begin{array}{rcl}
\mathsf{Contents} & = & \mathcal{P}(\mathsf{Prefix} \times \mathsf{CSite}) \times \mathcal{P}(\mathsf{PfxVar}) \\
& & \times\ (\mathsf{Con} \mathbin{\dot\cup} \mathsf{Fun}) \times \mathsf{Val}^{I*}
\end{array}$$

We need primitive operations on $\mathsf{HST}(x)$ to allocate a fresh storage cell in the heap (newloc), to obtain the contents of a location (get), to record a new binding in the heap (record), and to mark places in the heap when they have been examined (touch).

$$\begin{array}{lcl}
\multicolumn{3}{l}{\text{newloc}\colon \mathsf{Contents} \to \mathsf{HST}(\mathsf{Loc})} \\
\text{newloc}\ x\ h & = & (l, h[l \mapsto x]) \\
& & \mathbf{where}\ h(l) = \perp \\
\multicolumn{3}{l}{\text{get}\colon \mathsf{Loc} \to \mathsf{HST}((\mathsf{Con} \mathbin{\dot\cup} \mathsf{Fun}) \times \mathsf{Val}^{I*})} \\
\text{get}\ l\ h & = & ((c, ls), h) \\
& & \mathbf{where}\ (P, V, c, ls) = h(l) \\
\multicolumn{3}{l}{\text{record}\colon \mathsf{PfxVar} \to \mathsf{Val}^I \to \mathsf{HST}()} \\
\text{record}\ v\ l\ h & = & ((), h[l \mapsto (P, V \cup \{v\}, c, ls)]) \\
& & \mathbf{where}\ (P, V, c, ls) = h(l)\ \mathbf{if}\ l \in \mathsf{Loc} \\
& = & ((), h) \\
& & \mathbf{if}\ l \in \mathsf{Base} \\
\multicolumn{3}{l}{\text{touch}\colon \mathsf{Prefix} \times \mathsf{CSite} \to \mathsf{Loc} \to \mathsf{HST}()} \\
\text{touch}\ (\pi, z)\ l\ h & = & ((), h[l \mapsto (P \cup (\pi, z), V, c, ls)]) \\
& & \mathbf{where}\ (P, V, c, ls) = h(l)
\end{array}$$

The instrumented semantics function $\mathcal{E}^I\colon \mathsf{Exp} \to \mathsf{Prefix} \to \mathsf{FEnv}^I \to \mathsf{Env}^I \to \mathsf{HST}(\mathsf{Val}^I)$ is defined in Fig. 2 (with $u \in \mathsf{CSite}$). Explanation: Variables are looked up in the environment ρ, constants are interpreted by κ, and for a

$$
\begin{array}{lcll}
prg & \rightarrow & tdec^*\ dec^*\ exp & \text{(program)} \\
dec & \rightarrow & \texttt{fun}\ f\ v_1\ \dots\ v_n = exp & f \in \mathsf{Fun}, v_i \in \mathsf{Var}\ \text{(function declaration)} \\
tdec & \rightarrow & \texttt{data}\ t = \dots \mid c(t_1, \dots, t_k) \mid \dots & t, t_i \in \mathsf{TN}\ \text{(algebraic datatype declaration)} \\
exp & \rightarrow & v & v \in \mathsf{Var}\ \text{(variable)} \\
 & \mid & k & k \in \mathsf{Kon}\ \text{(constant symbol)} \\
 & \mid & f & f \in \mathsf{Fun}\ \text{(function symbol)} \\
 & \mid & (exp_1\ exp_2) & \text{(function application)} \\
 & \mid & c(exp_1, \dots, exp_k) & \text{(constructor application)} \\
 & \mid & \texttt{let}\ v = exp_1\ \texttt{in}\ exp_2 & (\texttt{let}\ \text{expression}) \\
 & \mid & \texttt{case}\ v\ \texttt{of}\ pat_1\ \texttt{=>}\ exp_1 \mid \dots \mid pat_m\ \texttt{=>}\ exp_m & (\texttt{case}\ \text{expression}) \\
pat & \rightarrow & c(v_1, \dots, v_k) & \text{(flat constructor pattern)}
\end{array}
$$

Figure 1: Syntax of the language.

$$
\begin{array}{rcl}
\mathcal{E}^I[\![v]\!]\pi\ \psi\ \rho & = & [\rho[\![v]\!]\ \mid\ ] \\
\mathcal{E}^I[\![k]\!]\pi\ \psi\ \rho & = & [\kappa[\![k]\!]\ \mid\ ] \\
\mathcal{E}^I[\![f^u]\!]\pi\ \psi\ \rho & = & [l\ \mid\ l \leftarrow \text{newloc}\ (\emptyset, \{(\pi, u)\}, f, \varepsilon)] \\
\mathcal{E}^I[\![c^u(e_1, \dots, e_k)]\!]\pi\ \psi\ \rho & = & [l\ \mid\ \dots, l_j \leftarrow \mathcal{E}^I[\![e_j]\!]\pi\psi\rho, \dots, l \leftarrow \text{newloc}(\emptyset, \{(\pi, u)\}, c, l_1 \dots l_k)] \\
\mathcal{E}^I[\![(e_1\ e_2)^u]\!]\pi\ \psi\ \rho & = & [l_0\ \mid\ \ l \leftarrow \mathcal{E}^I[\![e_1]\!]\pi\ \psi\ \rho, \\
 & & \qquad l' \leftarrow \mathcal{E}^I[\![e_2]\!]\pi\ \psi\ \rho, \\
 & & \qquad (f, l_1 \dots l_k) \leftarrow \text{get}\ l, \\
 & & \qquad l_0 \leftarrow \textbf{if}\ k+1 < n_f\ \ \textbf{then}\ \text{newloc}\ (\emptyset, \{(\pi, u)\}, f, l_1 \dots l_k l') \\
 & & \qquad\qquad\qquad\qquad \textbf{else}\ \psi(f)(\pi.u, l_1 \dots l_k l')] \\
\mathcal{E}^I[\![\texttt{let}\ v = e_1\ \texttt{in}\ e_2]\!]\pi\ \psi\ \rho & = & [l\ \mid\ l_0 \leftarrow \mathcal{E}^I[\![e_1]\!]\pi\ \psi\ \rho, \text{record}\ (\pi, v)\ l_0, l \leftarrow \mathcal{E}^I[\![e_2]\!]\pi\ \psi\ \rho[v \mapsto l_0]] \\
\mathcal{E}^I[\![\texttt{case}^u\ v_0\ \texttt{of} \dots & = & [l'\ \mid\ l_0 \leftarrow \mathcal{E}^I[\![v_0]\!]\pi\ \psi\ \rho, \text{touch}\ (\pi, u)\ l_0, \\
c_a(v_1, \dots, v_k)\ \texttt{=>}\ e_a \dots]\!]\pi\ \psi\ \rho & & \qquad (x, l_1 \dots l_k) \leftarrow \text{get}\ l_0, l' \leftarrow \textbf{case}\ x\ \textbf{of} \dots c_a : \\
 & & \qquad [l'\ \mid\ \dots, \text{record}\ (\pi, v_j)\ l_1, \dots, l' \leftarrow \mathcal{E}^I[\![e_a]\!]\pi\ \psi\ \rho[v_j \mapsto l_j]]]
\end{array}
$$

Figure 2: Instrumented semantics.

function symbol a closure without arguments is created. For a constructor application first the argument expressions are evaluated and then a new heap record is built from the constructor name, the arguments and creation site information. General function application first evaluates both subexpressions. It expects the first expression to evaluate to a closure, which is guaranteed by typing. Depending on whether or not the closure becomes a saturated application, either an augmented closure is generated or a function call is performed. Saturation of a closure is tested by comparing the number of arguments already in a closure for f with the number n_f of arguments of f. `let` expressions are handled in the obvious way. The evaluation of a `case` expression records the consumption of the subject (using the function "touch") and the bindings of the variables (using the function "record") and dispatches to the chosen branch depending on the constructor tag found. The semantics of a group of declarations is a function environment, *i.e.* $\mathcal{F}^I\colon dec^* \rightarrow \mathsf{FEnv}^I$. It is constructed as usual as the fixpoint of an environment constructing function.

$$
\begin{array}{l}
\mathcal{F}^I[\![\dots f(v_1, \dots, v_n) = e_f \dots]\!] = \\
\textbf{lfp}\ \lambda\psi.\psi[\dots, f \mapsto \textbf{strict}\lambda(\pi, l_1 \dots l_n). \\
\qquad [l\ \mid\ \text{record}\ (\pi, v_1)\ l_1, \dots, \text{record}\ (\pi, v_n)\ l_n, \\
\qquad\qquad \mathcal{E}^I[\![e_f]\!]\pi\ \psi\ [v_j \mapsto l_j]], \dots]
\end{array}
$$

(**lfp** computes the least fixpoint of its argument and **strict** f returns a strict version of the function f.) The outcome of the semantics is an environment ψ which binds function symbols to function which take a heap and returns pair consisting of a location and a modified heap.

The instrumented semantics computes what we call concrete values CVal which are obtained by unravelling the heap starting from a given location. A concrete value is a tree the nodes of which are labelled by a constructor or function symbol and the set of variables that are bound to it.

$$\mathsf{CVal} \quad = \quad \mathcal{P}(\mathsf{PfxVar}) \times (\mathsf{Base}\ \dot\cup\ (\mathsf{Con}\ \dot\cup\ \mathsf{Fun}) \times \mathsf{CVal}^*)$$

In order to create values in CVal from a location and a heap we need the unravelling function "mkCVal".

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{mkCVal}\colon \mathsf{Val}^I \times \mathsf{Heap} \rightarrow \mathsf{CVal}} \\
\text{mkCVal}(l, h) & = & (V, c, \text{mkCVal}(l_1, h) \dots \text{mkCVal}(l_k, h)) \\
 & & \textbf{where}\ (P, V, c, l_1 \dots, l_k) = h(l)\ \textbf{if}\ l \in \mathsf{Loc} \\
 & = & (\emptyset, l) \\
 & & \textbf{if}\ l \in \mathsf{Base}
\end{array}
$$

3.2 Symbolic Evaluation

3.2.1 Abstract Domain

Symbolic evaluation is used to determine that part of the shape of the value of an expression which can definitely be predicted. It deals with abstract values taken from

AVal which is the greatest solution of the equation

$$\mathsf{AVal} = \mathcal{P}(\mathsf{PfxVar}) \times (\{0,1\} \dot\cup \mathsf{Base} \dot\cup (\mathsf{Con} \dot\cup \mathsf{Fun}) \times \mathsf{AVal}^*)$$

An abstract value is a tree every node of which is decorated with a set of variables and either a constructor symbol or the name of a defined function representing constructed data or a closure, respectively. An ordering $\sqsubseteq$ on AVal is defined in Fig. 3. It makes 0 the smallest and 1 the greatest element of AVal. Constructed data as well as closures for the same function of different length ($l \neq l'$) are not comparable with $\sqsubseteq$. Their least upper bound in AVal is 1. However, if the top constructor is identical the comparison recurses on the corresponding arguments. Furthermore at every node of the smaller abstract value the set of variables must include the set of variables at the corresponding node of the larger value.

Proposition. $(\mathsf{AVal}, \sqsubseteq)$ forms a complete lattice.

The least element of AVal is $(\mathsf{Var}, 0)$, the top element is $(\emptyset, 1)$. Restricted to base types AVal is the well known lattice used for constant propagation [ASU86].

3.2.2 Environments

During symbolic execution we must keep track of some sharing information in order not to loose opportunities to expose calls with partially known arguments. A special environment structure keeps track of some definite sharing information. An environment is a pair of an equivalence relation on variables and a mapping from equivalence classes of variables to right hand sides. Right hand sides are defined by the grammar:

$\mathsf{Rhs} \to 1$	the completely unknown value,
$\mid 0$	the contradictory value,
$\mid c(v_1, \ldots, v_k)$	some constructor c applied to representatives of equivalence classes of variables,
$\mid f[v_1 \ldots v_l]$	a closure for f with $0 \leq l < n_f$ values.

Formally we define analysis environments by $\mathsf{Env}^S = (\mathsf{PfxVar} \to \mathsf{Rhs}) \times \mathcal{P}(\mathsf{PfxVar} \times \mathsf{PfxVar})$. Each $\rho' = (\rho_1, \rho_2) \in \mathsf{Env}^S$ is subject to the conditions

1. if $(v, v') \in \rho_2$ then $\rho_1 v = \rho_1 v'$,
2. if $\rho_1 v = c(v_1, \ldots, v_k)$ then $\{v_1, \ldots, v_k\} \subseteq \mathbf{dom}\ \rho_1$,
3. ρ_2 is an equivalence relation on $\mathbf{dom}\ \rho_1$, the domain of ρ_1.

We denote equivalence classes of ρ_2 by $[v]_{\rho_2}$. Let $\mathbf{dom}\ (\rho_1, \rho_2) = \mathbf{dom}\ \rho_2$.

Manipulation of environments is done by functions *lookup* and *enter* defined in Fig. 4. Both functions preserve the conditions 1.–3. above. We define an ordering on environments by setting $\rho \sqsubseteq \rho'$ iff $\forall v \in \mathsf{PfxVar}.\mathit{lookup}\ v\ \rho \sqsubseteq \mathit{lookup}\ v\ \rho'$. This makes Env^S a complete lattice with least element $(\emptyset, =)$ (*i.e.*, the empty mapping and equality on PfxVar) and *lookup* a continuous function.

In the second and third case for *enter* the variables $n_1, \ldots, n_k$ (n_l) are fresh variables, *i.e.*, they do not appear anywhere else in the environment or in d.

Proposition. *enter* is continuous.

Proof: First we show that for every $v \in \mathsf{PfxVar}$ and $d \in \mathsf{AVal}$ the function $\mathit{enter}\ v\ d$ is continuous in $\mathsf{Env}' \to \mathsf{Env}'$. In the definition of *enter* ρ_2' depends continuously on ρ since * (reflexive and transitve closure of a relation) is continuous in $(\mathcal{P}(\mathsf{PfxVar} \times \mathsf{PfxVar}), \supseteq)$ (with set intersection as least upper bound operation). An induction on $(vd, x) = d$ yields the claim: if $x \in \{0,1\}$ we are done, since updating the function ρ_1 is continuous in ρ. Otherwise ρ_1' depends continuously on ρ and *enter* is continuous on $d_1, \ldots, d_k$, by induction.

In a similar way it can be seen that $\mathit{enter}\ v\ d\ \rho$ depends continuously on $d \in \mathsf{AVal}$.

3.2.3 Evaluation

Symbolic evaluation of an expression to an abstract value is defined by the function $\mathcal{E}^S$ presented in Fig. 5. It takes an expression e, a function environment $\psi' \in \mathsf{FEnv}^S$, an environment $\rho' \in \mathsf{Env}^S$, and yields an annotated value AVal that describes the shape of the result of evaluating e with values bound to the variables the shapes of which are described by ρ'. A function environment FEnv^S is a mapping from function names Fun to functions over annotated values taking additionally a Prefix parameter and an environment, *i.e.*,

$$\mathsf{FEnv}^S = \mathsf{Fun} \to \mathsf{Prefix} \times \mathsf{AVal}^n \times \mathsf{Env}^S \to \mathsf{AVal}$$

Function environments are ordered pointwise such that the greatest function environment $\psi_0' \in \mathsf{FEnv}^S$ maps all function symbols $f \in \mathsf{Fun}$ to the function $(\pi, d_1 \ldots d_n, \rho) \mapsto (\emptyset, 1)$ which maps all arguments to the top value of the domain AVal and which is thus an abstraction of every function. The explanation for the semantics equations of $\mathcal{E}^S$ is as follows: variables are looked up in the environment, constants create an unshared value, and function symbols create unshared empty closures. Constructor applications create a new value which is completely unshared at the top, hence only its creation site is registered in the top node. Function application has two cases. It either creates an extended closure from the old function closure and the additional argument or it effects a function application which is handled via lookup in the function environment ψ. The value environment is passed on as an additional parameter in order to increase the amount of sharing which is detected. In principle it would suffice to pass on the equivalence relation on variables. The `let` expression opens a possibility for sharing in the variable v. There are two possibilities at a `case` expression. If the branch which is taken can be predicted to have the shape $c_a(\ldots)$ by means of $\mathcal{E}^S$ the value of the `case` expression is the value of e_a. Otherwise all branches are entered with the environment changed to reflect the supposed structure of $\mathcal{E}^S[\![e_0]\!]$ and the least upper bound of the result is taken. Another albeit less precise alternative at this place would be to safely approximate the outcome of the `case` expression by $(\emptyset, 1)$.

Define $\mathcal{F}^S \colon \mathsf{Dec} \to \mathsf{FEnv}^S$ analoguously to $\mathcal{F}^I$ as follows.

$$\begin{array}{l} \mathcal{F}^S[\![\ldots f(v_1, \ldots, v_n) = e_f \ldots]\!] = \\ \mathbf{gfp}\ \lambda\psi'.\psi'[\ldots, f \mapsto \lambda(\pi', y_1 \ldots y_n, \rho'). \\ \quad \mathcal{E}^S[\![e_f]\!]\pi'\ \psi'\ (\ldots \mathit{enter}\ (\pi', v_j)\ y_j \ldots \rho' \ldots), \ldots] \end{array}$$

$$
\begin{array}{rcl}
(S_1,0) \sqsubseteq (S_2,x) & \Leftrightarrow & S_1 \supseteq S_2 \\
(S_1,x) \sqsubseteq (S_2,1) & \Leftrightarrow & S_1 \supseteq S_2 \\
(S_1,c(d_1,\ldots,d_k)) \sqsubseteq (S_2,c(d'_1,\ldots,d'_k)) & \Leftrightarrow & S_1 \supseteq S_2 \wedge \forall 1 \le i \le k.d_i \sqsubseteq d'_i \\
(S_1,f[d_1\ldots d_l]) \sqsubseteq (S_2,f[d'_1\ldots d'_{l'}]) & \Leftrightarrow & S_1 \supseteq S_2 \wedge l = l' \wedge \forall 1 \le i \le l.d_i \sqsubseteq d'_i
\end{array}
$$

Figure 3: Ordering on abstract values.

$$
\begin{array}{rll}
\multicolumn{3}{l}{\mathit{lookup}\colon \mathsf{PfxVar} \to \mathsf{Env}^S \to \mathsf{AVal}^S} \\
\mathit{lookup}\ v\ \rho = & (\mathsf{PfxVar},0) & \textbf{if } v \notin \textbf{dom } \rho_1 \\
= & ([v]_{\rho_2}, w) & \\
\textbf{where} & (\rho_1,\rho_2) = \rho & \\
& w = 1 & \textbf{if } \rho_1 v = 1 \\
& w = c(\mathit{lookup}\ v_1\ \rho,\ldots,\mathit{lookup}\ v_k\ \rho) & \textbf{if } \rho_1 v = c(v_1,\ldots,v_k) \\
& w = f[\mathit{lookup}\ v_1\ \rho \ldots \mathit{lookup}\ v_l\ \rho] & \textbf{if } \rho_1 v = f[v_1 \ldots v_k]
\end{array}
$$

$$
\begin{array}{l}
\mathit{enter}\colon \mathsf{PfxVar} \to \mathsf{AVal}^S \to \mathsf{Env}^S \to \mathsf{Env}^S \\
\mathit{enter}\ v\ d\ \rho = \textbf{let}\quad (vs,x) = (\{v\},1) \sqcap d \\
\qquad\qquad\qquad\qquad (\rho_1,\rho_2) = \rho \\
\qquad\qquad\qquad\qquad \rho'_2 = (\rho_2 \cup \{(v,v') \mid v' \in vs\})^* \\
\qquad\qquad\quad \textbf{in}\quad \textbf{if } x \in \{0,1\} \textbf{ then} \\
\qquad\qquad\qquad\qquad (\rho_1[v' \mapsto x \mid v' \in vs]) \\
\qquad\qquad\qquad\quad \textbf{elseif } x = c(d_1,\ldots,d_k) \textbf{ then} \\
\qquad\qquad\qquad\qquad \textbf{let } \rho'_1 = \rho_1[v' \mapsto c(n_1,\ldots,n_k) \mid v' \in vs] \\
\qquad\qquad\qquad\qquad\quad \text{where the } n_i \text{ are fresh variables} \\
\qquad\qquad\qquad\qquad \textbf{in } \mathit{enter}\ n_1\ d_1\ (\ldots(\mathit{enter}\ n_k\ d_k\ (\rho'_1,\rho'_2))\ldots) \\
\qquad\qquad\qquad\qquad \textbf{else} \quad x = f[d_1\ldots d_l] \\
\qquad\qquad\qquad\qquad \textbf{let } \rho'_1 = \rho_1[v' \mapsto f(n_1,\ldots,n_l) \mid v' \in vs] \\
\qquad\qquad\qquad\qquad\quad \text{where the } n_i \text{ are fresh variables} \\
\qquad\qquad\qquad\qquad \textbf{in } \mathit{enter}\ n_1\ d_1\ (\ldots(\mathit{enter}\ n_l\ d_l\ (\rho'_1,\rho'_2))\ldots)
\end{array}
$$

Figure 4: Environment manipulation.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{E}^S\colon \mathsf{Exp} \to \mathsf{Prefix} \to \mathsf{FEnv}^S \to \mathsf{Env}^S \to \mathsf{AVal}} \\
\mathcal{E}^S[\![v]\!]\pi'\psi'\rho' & = & \mathit{lookup}\ (\pi',v)\ \rho' \\
\mathcal{E}^S[\![k]\!]\pi'\psi'\rho' & = & (\emptyset, \kappa(k)) \\
\mathcal{E}^S[\![f^u]\!]\pi'\psi'\rho' & = & (\{\pi'.u\}, f[\,]) \\
\mathcal{E}^S[\![c^u(e_1,\ldots,e_k)]\!]\pi'\psi'\rho' & = & (\{\pi'.u\}, c(\ldots\mathcal{E}^S[\![e_j]\!]\pi'\psi'\rho'\ldots)) \\
\mathcal{E}^S[\![(e_1\ e_2)^u]\!]\pi'\psi'\rho' & = & \textbf{case } \mathcal{E}^S[\![e_1]\!]\pi'\psi'\rho' \textbf{ of} \\
& & (vs,1) : (\emptyset,1) \\
& & |(vs,f[d_1\ldots d_l]) : \textbf{let } d = \mathcal{E}^S[\![e_2]\!]\pi'\psi'\rho' \textbf{ in } \textbf{if } l+1 < n_f \\
& & \qquad\qquad \textbf{then } (\{\pi'.u\}, f[d_1\ldots d_l d]) \\
& & \qquad\qquad \textbf{else } \psi'(f)(\pi'.u, d_1\ldots d_l d, \rho') \\
\mathcal{E}^S[\![\texttt{let } v = e_1 \texttt{ in } e_2]\!]\pi'\psi'\rho' & = & \mathcal{E}^S[\![e_2]\!]\pi'\psi'(\mathit{enter}\ (\pi',v)\ (\mathcal{E}^S[\![e_1]\!]\pi'\psi'\rho')\ \rho') \\
\multicolumn{3}{l}{\mathcal{E}^S[\![\texttt{case } v_0 \texttt{ of} \ldots c_a(v_1,\ldots,v_k) \texttt{ => } e_a\ldots]\!]\pi'\psi'\rho' =} \\
\multicolumn{3}{l}{\quad \textbf{case } \mathcal{E}^S[\![v_0]\!]\pi'\psi'\rho' \textbf{ of}} \\
\multicolumn{3}{l}{\quad (vs, c_a(d_1,\ldots,d_k)) : \mathcal{E}^S[\![e_a]\!]\pi'\psi'(\mathit{enter}\ (\pi',v_1)\ d_1 \ldots (\mathit{enter}\ (\pi',v_k)\ d_k\ \rho')\ldots)} \\
\multicolumn{3}{l}{\quad |(vs,1) : \bigsqcup_{j=1}^{m} \mathcal{E}^S[\![e_j]\!]\pi'\psi'(\mathit{enter}\ (\pi',v_0)\ (vs, c_j(\ldots(\{(\pi',v_j)\},1)\ldots))\rho')}
\end{array}
$$

Figure 5: Symbolic evaluation.

gfp computes the greatest fixpoint of its argument. $\mathcal{F}^S$ is well-defined since all operations on all right hand sides are continuous. In the following we use $\psi^\bullet = \mathcal{F}^S[\![decl]\!]$ where the set of declarations is clear from the context.

3.2.4 Correctness

Now the entities that $\mathcal{E}^S$ deals with need be connected to the entities that $\mathcal{E}^I$ understands. To this end we define a range of abstraction functions α^S which abstract prefixes, environments, function environments, and values. We will use the same symbol α^S for each of these mappings since there is no danger of confusion.

Let $\pi \in \mathsf{Prefix}$, $\rho \in \mathsf{Env}^I$, $h \in \mathsf{Heap}$, $\psi \in \mathsf{FEnv}^I$, and $(V, c, x_1 \dots x_k) \in \mathsf{CVal}$ where h must be valid for ρ. A heap h is *valid* for ρ if for all $v \in \mathbf{dom}\ \rho$ the unravelling $\mathrm{mkCVal}(\rho[\![v]\!], h)$ is defined.

$$\begin{aligned}
\alpha^S(\pi) &= \pi \\
\alpha^S(\rho, h) &= \mathit{enter}\ v\ \alpha^S(\mathrm{mkCVal}(\rho[\![v]\!], h) \dots (\emptyset, =) \\
&\quad \text{for all } v \in \mathbf{dom}\ \rho \\
\alpha^S(\psi) &= [f \mapsto \lambda(\pi', y_1 \dots y_n). \\
&\quad \textstyle\bigsqcup\{\alpha^S(\mathrm{mkCVal}(\psi(f)(\pi, l_1 \dots l_n)\ h)) \\
&\quad \mid (l_1 \dots l_n, h) \in \mathsf{Val}^{I^*} \times \mathsf{Heap} \text{ such that} \\
&\quad \alpha^S(\pi) = \pi' \text{ and } \alpha^S(\mathrm{mkCVal}(l_j, h)) \sqsubseteq y_j\} \\
&\quad \mid f \in \mathbf{dom}\ \psi] \\
\alpha^S(V, a, x_1 \dots x_k) & \\
&= (V, a, \alpha^S(x_1) \dots \alpha^S(x_k)) \qquad a \in \mathsf{Fun} \cup \mathsf{Con}
\end{aligned}$$

Proposition. Let $\pi, \pi' \in \mathsf{Prefix}$, $\psi \in \mathsf{FEnv}^I$, $\psi' \in \mathsf{FEnv}^S$, $\rho \in \mathsf{Env}^I$, $h \in \mathsf{Heap}$ valid for ρ, $\rho' \in \mathsf{Env}^S$ such that $\alpha^S(\pi) \sqsubseteq \pi'$, $\alpha^S(\psi) \sqsubseteq \psi'$, and $\alpha^S(\rho, h) \sqsubseteq \rho'$.

If $\mathcal{E}^I[\![e]\!]\pi\ \psi\ \rho\ h = (l', h')$ then $\alpha^S(\mathrm{mkCVal}(l', h')) \sqsubseteq \mathcal{E}^S[\![e]\!]\pi'\ \psi'\ \rho'$.

Proof: By induction on e using some auxiliary lemmas.

Lemma. Let $\alpha^S(\rho, h) \sqsubseteq \rho'$ and $\alpha^S(\mathrm{mkCVal}(l, h)) \sqsubseteq d$ (for suitable ρ, h, ρ', l, h, and d).

$$\alpha^S(\rho[v \mapsto l], h) \sqsubseteq \mathit{enter}\ v\ d\ \rho'$$

Proof: By continuity of *enter*:

$$\begin{aligned}
& \alpha^S(\rho[v \mapsto l], h) \\
= \;& \mathit{enter}\ v\ (\alpha^S(\mathrm{mkCVal}(l, h)))\ (\alpha^S(\rho, h)) \\
\sqsubseteq \;& \mathit{enter}\ v\ d\ (\alpha^S(\rho, h)) \\
\sqsubseteq \;& \mathit{enter}\ v\ d\ \rho'
\end{aligned}$$

We can also prove that $\mathcal{F}^S$ safely abstracts $\mathcal{F}^I$.

Proposition. For all $decl \in \mathsf{Dec}$ it holds $\alpha^S(\mathcal{F}^I[\![decl]\!]) \sqsubseteq \mathcal{F}^S[\![decl]\!]$.

Proof: By fixpoint induction.

Even though we will not need to compute the fixpoint (and in fact we cannot hope to do so) the result tells us that we can safely follow function calls in symbolic execution.

The main problem with symbolic evaluation is that it terminates strictly less often than real evaluation does: there are terminating programs the symbolic evaluation of which does not terminate. In order to obtain termination of symbolic execution we use a stack of active function calls with their argument patterns and check upon entry to a function whether the new call pattern is more specific than the call pattern of any pending call to the same function. If that is the case we crudely approximate the result by $(\emptyset, 1)$ and return immediately.

The check for a new function call being more specific as an already pending one is carried out as follows. First, the call patterns are compares componentwise using $\sqsubseteq$ disregarding the information on variable bindings. Components of base type are ignored in the comparison (which has the same effect as generalizing them to 1 first). Second, if a fixed number of invocations of a function f is pending we look for inductive arguments of f. If one of f's arguments has an ancestor which occurs in a pending call at the same position we regard this as evidence for an inductive argument and return $(\emptyset, 1)$. All necessary information for doing such a trace is present in the current environment in concert with the call strack. From the minimal prefix component of the binding information of the current argument we can determine the call which has effected the binding of that particular node to a variable. The pattern of that call is then used as a strating point to search for the particular node. We can even guarantee to find it at depth one if no function contains nested **case** expressions. Functions could be transformed into that form beforehand but it is simpler to add pseudo calls to the call stack for every encounter with a nested **case**. The call pattern of the pseudo call is just the symbolic value of the **case**'s subject variable. The third and final measure is to put an upper bound on the number of pending calls to a single function (or uses of a specific call site).

More advanced techniques are being used in online partial evaluators (*i.e.* [WCRS91]) but we will defer using them until practical experiences have been gathered.

3.3 Consumption Analysis

In order to correctly prune call patterns later on we must be able to determine which part of an argument to a function is certainly consumed. This is achieved by a different abstraction of the instrumented semantics $\mathcal{E}^I$. Its domains are extensions of the domains used for the symbolic evaluation above. We just sketch the definitions, here.

$$\begin{aligned}
\mathsf{Env}^X &= (\mathsf{PfxVar} \to \mathsf{Rhs} \times \mathsf{XInfo}) \\
&\quad \times \mathcal{P}(\mathsf{PfxVar} \times \mathsf{PfxVar}) \\
\mathsf{FEnv}^X &= \mathsf{Fun} \to \mathsf{Prefix} \times \mathsf{AVal}^{X*} \times \mathsf{Env}^X \to \mathsf{AVal}^X \\
\mathsf{AVal}^X &= \mathsf{XInfo} \times \mathcal{P}(\mathsf{PfxVar}) \\
&\quad \times (\{0,1\} \,\dot\cup\, (\mathsf{Con} \,\dot\cup\, \mathsf{Fun}) \times \mathsf{AVal}^{X*} \\
\mathsf{XVal} &= \mathsf{XInfo} \times \mathcal{P}(\mathsf{PfxVar}) \times (\mathsf{Con} \,\dot\cup\, \mathsf{Fun}) \times \mathsf{XVal}^*
\end{aligned}$$

By choosing different lattices for XInfo different degrees of knowledge about examination can be obtained. We will discuss that issue below.

Environments are extended to also register call sites which certainly lead to a test of the associated variable. Annotated values are extended in the same way. XVal is the examination information that we can obtain from a heap by the following function mkXVal (analoguous to CVal and mkCVal for the shape semantics).

$$\begin{aligned}
&\mathrm{mkXVal} \colon \mathsf{Loc} \times \mathsf{Heap} \to \mathsf{XVal} \\
&\mathrm{mkXVal}(l, h) = (P, V, c, \mathrm{mkXVal}(l_1, h) \dots \mathrm{mkXVal}(l_k, h)) \\
&\qquad \mathbf{where}\ (P, V, c, l_1 \dots, l_k) = h(l)
\end{aligned}$$

The order on AVal^X extends $\sqsubseteq$ on AVal^S by

$$(P, V, x) \sqsubseteq (P', V', x') \Leftrightarrow (V, x) \sqsubseteq (V', x') \wedge P \geq P'$$

which makes AVal^X also into a complete lattice. A reasonable and simple choice for XInfo is the two point lattice $\{0, 1\}$ with $0 \leq 1$. The value 1 denotes definitive examination while 0 means the converse. Another more exact but more expensive choice would be $\mathcal{P}(CSite)_\perp$ where the order is induced by set inclusion in CSite. Here, any element greater than $\emptyset$ denotes definitive examination effected at definitive program points, while $\emptyset$ denotes definitive examination where the exact program point is not known. In the following presentation we stick to the latter although that choice would not be advisable for an implementation.

In the resulting lattice AVal^X, the top element $(\perp, \emptyset, 1)$ is the least informative element. The least upper bound operation performs (lifted) set intersection on the XInfo component.

In order to obtain correct results for examinations which are mediated by function calls we introduce an additional parameter of type AVal^X to abstractions of functions. The abstraction of a function only provides the examination of the additional parameter. We add the new parameter in front of the argument list.

The analysis is parameterized by a variable v'. The goal is the approximation of the consumption of parts of the value of v'. In the definition of $\mathcal{E}^X$ in Fig. 6 we will at some places consider values in AVal^X as values in AVal^S by implicit application of the obvious projections.

The examination semantics of a declaration is given by $\mathcal{F}^X$ in Fig. 7. The correctness of the consumption analysis can be stated and proved analoguously to the case of symbolic evaluation.

The consupmtion analysis $\mathcal{E}_v^X$ returns the abstract value of v annotated with examination information. Variables, constants, and function symbols do not consume anything. In constructor applications the consumptions are collected from the subterms with $\sqcap$. For function application first the consumptions that occur in the subterms are determined and then, if there is a saturated application, the effect of a function call on the variable v' is computed. The only consumption takes place in the `case` expression which consumes the top constructor of v_0. This is recorded with a combination of *enter* and *lookup*. As usual, if the outcome of the `case` test is known only the result of the selected branch is taken, otherwise the results of the branches are merged using $\sqcup$.

Termination is an issue with $\mathcal{E}_v^X$ as well. First of all notice that we start $\mathcal{E}_v^X$ with an abstract value d bound to v. We can stop as soon as none of the original nodes of d is reachable from the current arguments of when all of the original nodes are annotated as examined. As in the case of $\mathcal{E}^S$ we need a call stack recording pending calls. We stop if there is no examination of v between two invocations of a function or if they examine the same node of v. From each of these cases we return the value $(\perp, \mathsf{PfxVar}, 0)$.

3.4 Call Pattern Detection

The computation of a complete set of call patterns is the goal of this section. The analysis itself is straightforward. However, some care must be taken to avoid nontermination. This is achieved through the consumption analysis of the previous section. Note that again we are not interested in the fixpoint semantics $\mathcal{F}^X$ but only in finite approximations. The fixpoint approximates what will eventually be consumed after a finite but unknown number of calls but we have to know what is consumed during a finite number of function calls, where the number does not depend on input data.

The analysis function $\mathcal{C}$ defined in Fig. 8 finds specializable calls by employing symbolic execution to predict the branch taken in a `case` expression and in order to find approximations to the set of concrete values that are passed as parameters. $\mathcal{C}$ takes an expression $e \in \mathsf{Exp}$ to analyze for calls with partially known arguments, a prefix π denoting the call history, a function environment $\psi' \in \mathsf{FEnv}^S$, and an environment $\rho' \in \mathsf{Env}^S$. Its result is a set of call patterns coded as tuples consisting of the name of the called function (Fun), an encoding of its call site, and a list of argument shapes as annotated values in AVal^*. Explanation: the equations for variables, constants, function names, constructor applications, and `let`-expressions only serve to collect call patterns from their subexpressions and to provide base cases. At a function application the call patterns of the subexpressions are collected and a new call pattern is constructed from the results of the symbolic evaluation of the additional argument if the application gets saturated. In order to be independant from the variables that are visible at a specific call site, we strip them from the annotated value with the function *strip* described in Fig. 9. At a `case` expression symbolic evaluation $\mathcal{E}^S$ is again used to predict the branch which is taken. If it is possible to predict the branch only the call patterns from that branch are extracted. Otherwise the call patterns are collected from all branches. Starting from a subset $E \subseteq \mathsf{Fun}$ (of externally called functions) and with $\psi^\bullet = \mathcal{F}_A^S[\![decl]\!]$ we can define a sequence C_i of sets with $C_i \in \mathcal{P}(\mathsf{Fun} \times \mathsf{Prefix} \times \mathsf{AVal}^*)$ as follows.

$$\begin{aligned} C_0 &= \{(f, f, (\emptyset, 1) \dots (\emptyset, 1)) \mid f \in E\} \\ C_{i+1} &= C_i \cup \bigcup \{\mathcal{C}[\![e_f]\!]\pi\ \psi^\bullet\ (enter\ (\pi, v_1)\ d_1 \dots (\emptyset, =) \dots) \\ &\qquad \mid (f, \pi, d_1 \dots d_n) \in C_i\} \end{aligned}$$

Unfortunately the sequence of the C_i may not become stationary and hence can lead to a non-terminating analysis, even for terminating subject programs. As an example consider the program

```
f xs = case xs of
         [ ]      => [ ]
       | n :: xtl => if n>0 then f ((n-1) :: xs)
                             else xs
```

which generates the call patterns $f(1), f(1 :: 1 :: 1), f(1 :: 1 :: 1 :: 1), \dots$ and so on.

The measure against the termination problem is to prune the call patterns. Only that part of a pattern which is certainly consumed is admitted. This solves the problem for the above function f since the argument of f is only tested at the topmost constructor, thus effectively pruning $f(1 :: 1 :: 1)$ to $f(1 :: 1)$ since the nested :: is never tested at all by f. This amounts to the definition in Fig 10. For every call pattern $(f, \pi, p_1 \dots, p_n)$ already accumulated in C_i we compute the set of call patterns $(g, \pi', q_1 \dots q_m)$ generated by symbolic execution of f's body e_f. Then the g-patterns are pruned by applying for each (non-trivial) argument pattern q_j the consumption analysis $\mathcal{E}_{v_j}^X$ which returns the consumed part r_j of q_j.

$$
\begin{array}{lcl}
\mathcal{E}^X_{v'} : \mathsf{Exp} \to \mathsf{Prefix} \to \mathsf{FEnv}^X \to \mathsf{Env}^X \to \mathsf{AVal}^X & & \\
\mathcal{E}^X_{v'}[\![v]\!]\pi'\psi'\rho' & = & (\bot, \mathsf{PfxVar}, 0) \\
\mathcal{E}^X_{v'}[\![k]\!]\pi'\psi'\rho' & = & (\bot, \mathsf{PfxVar}, 0) \\
\mathcal{E}^X_{v'}[\![f]\!]\pi'\psi'\rho' & = & (\bot, \mathsf{PfxVar}, 0) \\
\mathcal{E}^X_{v'}[\![c^u(e_1, \ldots, e_k)]\!]\pi'\psi'\rho' & = & \bigsqcap_{i=1}^k \mathcal{E}^X_{v'}[\![e_i]\!]\pi'\psi'\rho' \\
\mathcal{E}^X_{v'}[\![(e_1\ e_2)^u]\!]\pi'\psi'\rho' & = & \mathcal{E}^X_{v'}[\![e_1]\!]\pi'\psi'\rho' \sqcap \mathcal{E}^X_{v'}[\![e_2]\!]\pi'\psi'\rho' \sqcap \\
& & \mathbf{let}\ (\{v_0, \ldots\}, f[d_1 \ldots d_l]) = \mathcal{E}^S[\![e_1]\!]\pi'\psi^\bullet\rho' \\
& & \quad \rho'' = \mathit{enter}\ (\pi', v_0)\ (\{u\}, \emptyset, 1)\ \mathbf{in} \\
& & \mathit{lookup}\ v'\ \rho'' \sqcap\ \mathbf{if}\ l+1 < n_f \\
& & \qquad \mathbf{then}\ (\bot, \mathsf{PfxVar}, 0) \\
& & \qquad \mathbf{else}\ \psi'(f)(\pi'.u, (\mathit{lookup}\ v'\ \rho')d_1 \ldots d_l(\mathcal{E}^S[\![e_2]\!]\pi'\psi^\bullet\rho'), \rho') \\
\mathcal{E}^X_{v'}[\![\mathtt{let}\ v = e_1\ \mathtt{in}\ e_2]\!]\pi'\psi'\rho' & = & \mathcal{E}^X_{v'}[\![e_1]\!]\pi'\psi'\rho' \sqcap \mathcal{E}^X_{v'}[\![e_2]\!]\pi'\psi'(\mathit{enter}\ (\pi', v)\ (\mathcal{E}^S[\![e_1]\!]\pi'\psi^\bullet\rho')\ \rho')
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}^X_{v'}[\![\mathbf{case}\ v_0\ \mathbf{of} \ldots c_a(v_1, \ldots, v_k)\ \mathtt{=>}\ e_a \ldots]\!]\pi'\psi'\rho' = \\
\quad \mathbf{let}\ d_0 = \mathit{lookup}\ v_0\ \rho' \sqcap (\{\pi'\}, \emptyset, 1) \\
\qquad \rho'' = \mathit{enter}\ (\pi', v_0)\ d_0 \\
\quad \mathbf{in}\ \mathit{lookup}\ v'\ \rho'' \sqcap \\
\quad \mathbf{case}\ d_0\ \mathbf{of} \\
\quad (vs, c_a((V_1, d_1), \ldots, (V_k, d_k))) : \\
\qquad \mathcal{E}^X_{v'}[\![e_a]\!]\pi'\psi'\ (\mathit{enter}\ (\pi', v_0)\ (\{\pi'\}, \emptyset, c_a((\bot, \{(\pi', v_1)\} \cup V_1, d_1) \ldots (\bot, \{(\pi', v_k)\} \cup V_k, d_k)))\ \rho'') \\
\quad |(vs, 1) : \bigsqcup_{j=1}^m \mathcal{E}^X_{v'}[\![e_j]\!]\pi'\psi'(\mathit{enter}\ (\pi', v_0)\ (\{\pi'\}, vs, c_j(\ldots(\bot, \{(\pi', v_j)\}, 1)\ldots))\rho'')
\end{array}
$$

Figure 6: Consumption Analysis.

$$
\begin{array}{l}
\mathcal{F}^X : \mathsf{Dec} \to \mathsf{FEnv}^X \\
\mathcal{F}^X[\![\ldots f(v_1, \ldots, v_n) = e_f \ldots]\!] = \\
\mathbf{gfp}\ \lambda\psi'.\psi'[f \mapsto \lambda(\pi, y_0 y_1 \ldots y_n, \rho).\mathcal{E}^X_{v_0}[\![e_f]\!]\pi\psi'(\mathit{enter}\ (\pi, v_0)\ y_0\ (\mathit{enter}\ (\pi, v_1)\ y_1 \ldots \rho \ldots)) \\
\qquad\qquad |\ f \in \mathsf{Fun}\ \text{and}\ v_0 \notin \{v_1, \ldots, v_n\}]
\end{array}
$$

Figure 7: Consumption environment.

$$
\begin{array}{lcl}
\mathcal{C} : \mathsf{Exp} \to \mathsf{Prefix} \to \mathsf{FEnv}^S \to \mathsf{Env}^S \to \mathcal{P}(\mathsf{Fun} \times \mathsf{Prefix} \times \mathsf{AVal}^*) & & \\
\mathcal{C}[\![v]\!]\pi\psi'\rho' & = & \emptyset \\
\mathcal{C}[\![k]\!]\pi\psi'\rho' & = & \emptyset \\
\mathcal{C}[\![f]\!]\pi\psi'\rho' & = & \emptyset \\
\mathcal{C}[\![c(e_1, \ldots, e_k)]\!]\pi\psi'\rho' & = & \bigcup_{i=1}^k \mathcal{C}[\![e_i]\!]\pi\psi'\rho' \\
\mathcal{C}[\![(e_1\ e_2)]\!]\pi\psi'\rho' & = & \mathcal{C}[\![e_1]\!]\pi\psi'\rho' \cup \mathcal{C}[\![e_2]\!]\pi\psi'\rho' \cup \\
& & \mathbf{case}\ \mathcal{E}^S[\![e_1]\!]\pi\psi^\bullet\rho'\ \mathbf{of}\ (vs, 1) : \emptyset \\
& & \ |\ (vs, f[d_1 \ldots d_l]) :\ \mathbf{if}\ l+1 < n_f\ \mathbf{then}\ \emptyset \\
& & \qquad\qquad \mathbf{else}\ \{(f, \pi, \mathit{strip}\ (d_1 \ldots d_l(\mathcal{E}^S[\![e_i]\!]\pi\psi^\bullet\rho')))\} \\
\mathcal{C}[\![\mathtt{let}\ v = e_1\ \mathtt{in}\ e_2]\!]\pi\psi'\rho' & = & \mathcal{C}[\![e_1]\!]\pi\psi'\rho' \cup \mathcal{C}[\![e_2]\!]\pi\psi'(\mathit{enter}\ (\pi, v)\ (\mathcal{E}^S[\![e_1]\!]\pi\psi^\bullet\rho')\ \rho')
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\mathbf{case}\ v_0\ \mathbf{of} \ldots c_a(v_1, \ldots, v_k)\ \ \mathtt{=>}\ \ e_a \ldots]\!]\pi\psi'\rho' = \\
\quad \mathcal{C}[\![v_0]\!]\pi\psi'\rho' \cup \mathbf{case}\ \mathcal{E}^S[\![v_0]\!]\pi\psi^\bullet\rho'\ \mathbf{of} \\
\qquad (vs, c_a(d_1, \ldots, d_k)) : \mathcal{C}[\![e_j]\!]\pi\psi'(\mathit{enter}\ (\pi, v_1)\ d_1 \ldots (\mathit{enter}\ (\pi, v_k)\ d_k\ \rho') \ldots) \\
\qquad |(vs, 1) : \bigcup_{j=1}^m \mathcal{C}[\![e_j]\!]\psi(\mathit{enter}\ (\pi, v_0)\ (vs, c_j(\ldots(\{(\pi, v_j)\}, 1)\ldots)\rho)
\end{array}
$$

Figure 8: Detection of call patterns.

$$
\begin{array}{lcl}
\mathit{strip} : \mathsf{AVal} \to \mathsf{AVal} & & \\
\mathit{strip}\ (vs, 1) & = & (\emptyset, 1) \\
\mathit{strip}\ (vs, c(d_1, \ldots, d_k)) & = & (\emptyset, c(\mathit{strip}\ d_1, \ldots, \mathit{strip}\ d_k)) \\
\mathit{strip}\ (vs, f[d_1 \ldots d_l]) & = & (\emptyset, f[(\mathit{strip}\ d_1) \ldots (\mathit{strip}\ d_l)])
\end{array}
$$

Figure 9: Definition of *strip*.

$$
\begin{aligned}
C_{i+1} = C_i \cup \{(g, \pi', r_1 \ldots r_m) \;\; | \;& r_j = \mathcal{E}^X_{v_j}[\![e_g]\!]\pi'\psi^\bullet(enter\ (\pi', v_1)\ q_1\ \ldots (\emptyset, =) \ldots)), \\
& (g, \pi', q_1 \ldots q_m) \in \mathcal{C}[\![e_f]\!]\pi\psi^\bullet(enter\ (\pi, v_1)\ p_1\ \ldots (\emptyset, =) \ldots) \\
& (f, \pi, p_1 \ldots, p_n) \in C_i\}
\end{aligned}
$$

Figure 10: Pruning call patterns.

4 Synthesis

For reasons of space we can only give a brief outline of the specialization phase. For each call pattern $(f, \pi, p_1 \ldots p_n)$ a function $f^{p_1 \cdots p_n}$ is generated with m arguments where m is the sum of the sizes of the p_i. There will be an argument for every node which has unknown descendants in a pattern. This process is similar to arity raising [Rom90]. For instance, for the call pattern $(f, \pi, 1 :: 1)$ a function with three arguments is generated. Thus there are unique names for all known nodes of the arguments during specialization. Before actually processing the body e_f of f all function calls whose results can be predicted (using $\mathcal{E}^S$) and are known to be consumed (discovered by $\mathcal{E}^X$) are unfolded into e_f. Unfolding stops as soon as the creation sites of the consumed data are reached. In order to maximize the information available all `case` tests in the resulting expression are propagated as far outside as possible. Now the unique names for the argument nodes are propagated into e_f. Specialization proceeds by removing all `case` branches that are not selected by known data. For the remaining function calls the most specialized version available is chosen. This need not be uniquely determined. If more than one specialized version is applicable we can first try to select one by pruning the demanded pattern with $\mathcal{E}^X$. If there is still more than one version left we choose the one which uses the most number of known nodes. If there is still a choice at that point we choose an arbitrary version of the remaining ones. Finally, e_f is cleaned up by performing some static reductions on known data and by removing arguments which do not occur in e_f. If we already used f in other specializations before, we have to remove the arguments there as well. This process can give rise to removing arguments from other functions as well. It will terminate since there is only a finite number of functions and each has a finite number of arguments.

5 Related Work

The deforestation algorithm originating from Wadler [Wad90b] and further pursued by Chin [Chi92] and Hamilton and Jones [HJ92] is conceived to eliminate intermediate structured data (trees, list, etc.) by symbolic composition. Our method can sometimes avoid data construction but it can not achieve a deforestation effect since it only keeps track of definitely known parts of values while deforestation does not stop at unknown conditionals. Unlike deforestation it is also applicable to higher-order functions.

The concept of arity raising and its use in program specialization was introduced by Romanenko [Rom90]. He discusses the structure and principles of operation of an arity raiser in the context of a subset of pure Lisp. His arity raiser replaces a pair-valued argument by two separate arguments. In our approach arity raising is conditional, since the top constructor of the argument which has to be decomposed must be known. Of course flattening of pairs and tuples may also be integrated into the presented framework.

It should be noted that Romanenko's work is inspired by the concept of supercompilation introduced by Turchin [Tur86] in the 70s. Our method may be seen as an environment based special case of positive supercompilation, a term coined by Glück, Jones, Klimov, and Sørensen [GK93, SGJ94]. Supercompilation is a general mechanism to remove redundancy from programs by analyzing their execution histories and generating new programs by introducing suitable generalizations such that states recur. Positive supercompilation only propagates positive information through execution histories. However, both have only been investigated for first-order languages.

The present work generalizes a previous paper [Thi93] in several aspects: our earlier work only addresses a first-order subset of ML. The use of type information is not considered and hence constants of base type cannot be handled. Furthermore the previous analysis does not propagate information through function calls and it can not subject closures to arity raising, of course.

Performing arity raising on closures results in passing closure contents as parameters without packaging them together. The effect is similar to handling some closures in unboxed state as proposed by Leroy [Ler92]. As a matter of fact, as our transformed programs use tupling a lot, they should benefit from his unboxed tuples.

We were also made aware of the description of a higher-order arity raiser due to Steensgaard and Marquard [SM90]. Unfortunately we were unable to obtain it in time to discuss it here.

6 Conclusions and Future Work

We have demonstrated a practically feasible analysis and specialization method which can speed up certain functions considerably. By variation of parameters like limiting the level of unfolding and the level of recursion permitted for symbolic execution the analysis is fast enough to be included in a compiler. Even when only immediate recursive calls are considered some improvements can be achieved (cf. the function `merge`). Furthermore we feel that the analysis cannot be overcome by more rigorous programming. The duplication of code which happens due to the specialization cannot be avoided but should never be done by hand. Apart from being tedious and error prone, it is bad programming practice to duplicate code that essentially performs the same job.

The connection to deforestation deserves closer examination. It appears to be possible to derive the unfolding level mentioned above from the subject program. An extension of the analysis might take into account recursive knowledge about data structures, like the knowledge that all elements of a list have a certain form. However a fixpoint approximation would be needed for this and it is not clear how to construct a finite domain which can represent such information.

At the time of this writing an implementation in ML is almost completed. We hope that experiments with the implementation will support our claim that the technique is worthwhile including it into a compiler, both in terms of effectiveness and speed.

Acknowledgement

We would like to thank Arthur Steiner for his work on the implementation of the algorithms presented in this paper. We would also like to thank the referees who provided detailed and useful comments.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Aug85] Lennart Augustsson. Compiling pattern matching. In *Proc. Functional Programming Languages and Computer Architecture 1985*, pages 368–381. Springer Verlag, 1985. LNCS 201.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, June 1992.

[Fil93] Gilberto Filé, editor. *3rd International Workshop on Static Analysis*, Padova, Italia, September 1993. Springer Verlag. LNCS 724.

[GK93] Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Filé [Fil93], pages 112–123. LNCS 724.

[Has92] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R164, May 1992.

[HJ92] Geoffrey W. Hamilton and Simon B. Jones. Extending deforestation for first order functional programs. In Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors, *Proc. of the 1991 Glasgow Workshop on Functional Programming*, pages 134–145, Portree, Isle of Skye, August 1992. Springer-Verlag, Berlin.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

[Mit90] John C. Mitchell. *Type Systems for Programming Languages*, volume B of *Handbook of Theretical Computer Science*, chapter 8. Elsevier Science Publishers, Amsterdam, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Rom90] Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, pages 341–360, Copenhagen, Denmark, 1990. Springer Verlag. LNCS 432.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proc. 5th European Symposium on Programming*, pages 485–500, Edinburgh, UK, April 1994. Springer Verlag. LNCS 788.

[SM90] B. Steensgaard and M. Marquard. Parameter splitting in a higher-order functional language. DIKU Student Project 90-7-1, DIKU, University of Copenhagen, 1990 1990.

[Thi93] Peter Thiemann. Avoiding repeated tests in pattern matching. In Filé [Fil93], pages 141–152. LNCS 724.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wad90a] Philip L. Wadler. Comprehending monads. In *Proc. Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM.

[Wad90b] Philip L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, pages 165–191, Cambridge, MA, 1991. Springer Verlag. LNCS 523.