PETER THIEMANN

Avoiding Repeated Tests in Pattern Matching

Originalbeitrag erschienen in:
Patrick Cousot (Hrsg.): Static analysis : third international workshop; proceedings.
Berlin [u.a.]: Springer, 1993, S. 141 - 152

Avoiding Repeated Tests in Pattern Matching

Peter Thiemann*

Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany

1 Introduction

When programming in functional languages with pattern matching there are often situations where nested patterns are used to look ahead into the argument data structure. What we want is information about the structure below the top level data constructor in order to take the appropriate action. ML has the alias pattern *var* as *pattern* as a special syntax for cases where we also need values from the inside of a pattern.

The kind of nested patterns as just mentioned often leads to inefficiencies that are not obvious to the programmer. Consider the function `last` taken literally from Wikstrøm's SML library [11].

```
(* last : 'a list -> 'a *)
fun last []      = raise Last
  | last [x]     = x
  | last (x::xs) = last xs
```

Compilation of pattern matching transforms the definition to

```
fun last L = case L of
               nil    => raise Last
             | x :: L' => case L' of
                            nil     => x
                          | x'::L'' => last L'
```

Now we can make the following observation. Only the very first invocation of `last` can ever enter the `nil` branch of the first `case` and raise the exception. For all recursive invocations of `last` it is known that the argument is a non-empty list. An implementation of `last` that avoids repeating the test whether the argument list is empty can be given as follows.

```
fun last' x nil = x
  | last' _ (x::xs) = last' x xs

fun last nil = raise Last
  | last (x::xs) = last' x xs
```

The specialized function `last'` could be declared locally to `last`, but it is more advantageous to have `last'` declared globally since it can be used whenever it is known that the argument to `last` is a non-empty list.

* Email address: thiemann@informatik.uni-tuebingen.de

It is the purpose of this paper to give a simple and cheap approximate structure analysis (by abstract interpretation) that uncovers cases of repeated testing and generates specialized versions of the functions involved like `last'` above. Out intention is to use the analysis in a compiler. The techniques employed are connected to sharing analysis and partial evaluation. The analysis is applicable to a first order subset of ML.

It might be argued that such an analysis is applicable only to sloppy programming. However we feel that this is not the case. Consider the following fragment of an ML program to compute the next generation for Conway's game of life.

```
fun next_generation
    (x1::(xs as (x2::x3::_)))
    (y1::(ys as (y2::y3::_)))
    (z1::(zs as (z2::z3::_)))
  = fate y2 x1 x2 x3 y1 y3 z1 z2 z3:: next_generation xs ys zs
  | next_generation _ _ _ = [ 0 ]
```

Out of the nine constructor tests performed per recursive call, the outcome of six tests is known in advance since they have already been performed by the calling function. To expand the code by hand is tedious, error prone, and bad programming style since code for the same task is repeated in several places of the program. Our analysis identifies all of the repeated tests and produces specialized versions of `next_generation` without repeated tests.

The structure of the presentation is as follows. In Section 2 the syntax of a first order ML subset is defined along with an instrumented semantics that can express sharing properties. The next Section 3 defines a special environment structure for use in the abstract semantics. Section 4 introduces an analysis to find candidates for specialization by discovering argument patterns to function calls. The specialization process itself is detailed in Section 5 using arity raising. Finally we discuss related work in Section 6 and conclude in Section 7.

2 Syntax and semantics

We consider a first order ML subset with the syntax given by the grammar in Fig. 1.

The language is given an instrumented strict semantics. The instrumentation is chosen to provide enough detail to express sharing of structured values. Values are considered structured if an implementation allocates them on the heap (*eg.* constructed data, tuples, ...) so that they can be shared. Basic values like integers and nullary constructors are not considered structured. Each node of an object of a constructed data type carries a unique identification in the form of a non-empty sequence of numbers $m_0.m_1 \ldots m_n$, $n \geq 0$. When sharing occurs the object is formally copied but in a manner that shared nodes have a common (non-empty) prefix. Thus, we have unique nodes with unique access paths but are able to express sharing. A strict semantics with a store may be obtained by truncating all identifications to length 1 and considering m_0 as a store address. The strict standard semantics is recovered by projecting out the allocation information of the Mark component. The semantic domains are defined in Figure 2 where $\oplus$, $\otimes$, and $\multimap$ denote coalesced sum, smash product and strict function space construction. $\mathrm{Con}_{\perp}^{(k)}$ is the flat partial

$$
\begin{array}{lll}
prg & \to & dec\ exp \\
dec & \to & \texttt{fun}\ f\ v_1\ \ldots\ v_n = exp\ dec \\
 & | & \varepsilon \\
exp & \to & \texttt{case}\ exp\ \texttt{of}\ pat_1\ :\ exp_1 \mid \ldots \mid pat_m\ :\ exp_m \\
 & | & \texttt{let}\ v = exp\ \texttt{in}\ exp' \\
 & | & c(exp_1, \ldots, exp_k) \\
 & | & f(exp_1, \ldots, exp_n) \\
 & | & v \\
pat & \to & c(v_1, \ldots, v_k)
\end{array}
$$

where

$v \in \mathsf{Var}$	denumerable set of variables
$c \in \mathsf{Con}$	data constructors with arities $k(c)$
$f \in \mathsf{Fun}$	finite set of function symbols

Fig. 1. Syntax of the first-order language.

$$
\begin{array}{lll}
\mathsf{Int} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\}_\perp \\
\mathsf{Mark} & = & (\mathbb{N}^*)_\perp \\
\mathsf{Val0} & = & \mathsf{Int} \oplus \mathsf{Con}_\perp^{(0)} \oplus \underbrace{\mathsf{Con}_\perp^{(k)} \otimes \mathsf{Mark} \otimes \overbrace{\mathsf{Val0} \otimes \ldots \otimes \mathsf{Val0}}^{k}}_{\textbf{structured values}} \\
\mathsf{Val} & = & \mathbb{N}_\perp \otimes \mathsf{Val0} \\
\mathsf{Env} & = & \mathsf{Var} \to \mathsf{Val0} \\
\mathsf{FVal} & = & (\mathbb{N}_\perp \otimes \mathsf{Val0} \multimap \mathsf{Val}) \oplus \ldots \oplus (\mathbb{N}_\perp \otimes \overbrace{\mathsf{Val0} \otimes \ldots \otimes \mathsf{Val0}}^{n} \multimap \mathsf{Val}) \\
\mathsf{FEnv} & = & \mathsf{Fun} \to \mathsf{FVal}
\end{array}
$$

Fig. 2. Semantic domains.

order constructed from the constructor symbols of arity k. The instrumentation consists of supplying each structured value with a Mark component as discussed above and threading a generator of unique identifications (a natural number) through the whole execution. All functions take an unused identification as an additional argument and return an unused identification as part of the value. The semantic functions ensure that every identification is used at most once in an evaluation. The semantic functions are defined in Figure 3.

3 Representing environments

In our analyses we need a special environment structure which is able to transmit bindings. For example, if we match a list L against the pattern $x :: xs$ during the analysis the environment must keep track of the information that L is no longer a

$\mathcal{E}$: Exp $\to$ Mark $\to$ FEnv $\to$ Env $\to$ Val

$$\begin{aligned}
&\mathcal{E}[\![\, v \,]\!] m\psi\rho &&= \mathit{share}(m, \rho(v)) \\
&\mathcal{E}[\![\, c(e_1, \ldots, e_k) \,]\!] m_0\psi\rho &&= \textbf{let } (m_1, w_1) = \mathcal{E}[\![\, e_1 \,]\!] m_0\psi\rho \textbf{ in} \\
&&&\quad \ldots \\
&&&\quad \textbf{let } (m_k, w_k) = \mathcal{E}[\![\, e_k \,]\!] m_{k-1}\psi\rho \textbf{ in} \\
&&&\quad \mathit{fresh}\ (m_k, (c, m_k, w_1, \ldots, w_k)) \\
&\mathcal{E}[\![\, \texttt{let } v = e_1 \texttt{ in } e_2 \,]\!] m_0\psi\rho &&= \textbf{let } (m_1, w_1) = \mathcal{E}[\![\, e_1 \,]\!] m_0\psi\rho \textbf{ in} \\
&&&\quad \mathcal{E}[\![\, e_2 \,]\!] m_1\psi\rho[v \mapsto w_1] \\
&\mathcal{E}[\![\, f(e_1, \ldots, e_n) \,]\!] m_0\psi\rho &&= \textbf{let } (m_1, w_1) = \mathcal{E}[\![\, e_1 \,]\!] m_0\psi\rho \textbf{ in} \\
&&&\quad \ldots \\
&&&\quad \textbf{let } (m_n, w_n) = \mathcal{E}[\![\, e_n \,]\!] m_{n-1}\psi\rho \textbf{ in} \\
&&&\quad \psi(f)(m_n, w_1, \ldots, w_n) \\
&\mathcal{E}[\![\, \texttt{case } e_0 \texttt{ of} \ldots c_j(v_1, \ldots, v_k) : e_j \ldots \,]\!] m_0\psi\rho = \\
&&&\quad \textbf{let } (m_1, w) = \mathcal{E}[\![\, e_0 \,]\!] m_0\psi\rho \textbf{ in} \\
&&&\quad \textbf{if } w = (c_j, M, w_1, \ldots, w_k) \\
&&&\quad \textbf{then } \mathcal{E}[\![\, e_j \,]\!] m_1\psi\rho[v_i \mapsto w_i] \\
&&&\quad \textbf{elseif } w = (\ldots) \textbf{ then } \ldots \\
&&&\quad \textbf{else } \bot
\end{aligned}$$

$\mathcal{F}$: Dec $\to$ FEnv

$$\mathcal{F}[\![\, \{f(v_1, \ldots, v_{n_f}) = e_f \mid f \in \mathsf{Fun}\} \,]\!] = \\ \textbf{fix}\psi.\psi[f \mapsto \textbf{strict } \lambda(m, y_1, \ldots, y_{n_f}).\mathcal{E}[\![\, e_f \,]\!] m\psi\{v_i \mapsto y_i\} \mid f \in \mathsf{Fun}]$$

$\mathcal{P}$: Prg $\to$ Val

$$\mathcal{P}[\![\, d\ e \,]\!] = \mathcal{E}[\![\, e \,]\!] 1(\mathcal{F}[\![\, d \,]\!])\bot_{\mathsf{Env}}$$

where $m, m_i, m' \in \mathbb{N}$, $M \in$ Mark, $\psi \in$ FEnv, $\rho \in$ Env. The semantic **let** has a strict interpretation: **let** $v = \bot$ **in** e is equal to $\bot$ even if v does not occur in e. The projection **strict** is defined for domains D_1 and D_2 by

$$\textbf{strict}: (D_1 \to D_2) \to (D_1 \circ\!\!\to D_2)$$
$$\textbf{strict}\ f\ x = \begin{cases} \bot & \text{if } x = \bot \\ f\ x & \text{otherwise} \end{cases}$$

and the auxiliary functions *share* and *fresh* are defined by

$$\begin{aligned}
&\mathit{fresh} \in \mathsf{Val} \circ\!\!\to \mathsf{Val} \qquad \mathit{share}: \mathsf{Val} \circ\!\!\to \mathsf{Val} \\
&\mathit{fresh}\ (m, x) = (m + 1, x) \quad \mathit{share}(m, x) = \mathit{fresh}\ (m, \mathit{copy}\ x) \\
&\qquad \textbf{where } \mathit{copy}\ i = i \qquad\qquad i \in \mathsf{Int} \\
&\qquad\qquad \mathit{copy}\ c = c \qquad\qquad c \in \mathsf{Con}^{(0)} \\
&\qquad\qquad \mathit{copy}\ (c, M, w_1, \ldots, w_k) = \\
&\qquad\qquad\qquad (c, M.m, \mathit{copy}\ w_1, \ldots, \mathit{copy}\ w_k)
\end{aligned}$$

Fig. 3. Instrumented semantic functions.

totally unknown value, but that it is known to be a non-empty list with head x and tail xs. To achieve the transmission of bindings we represent an environment by

1. an equivalence relation on variables,
2. a mapping from equivalence classes of variables to right hand sides.

Right hand sides are defined by the grammar

$$\begin{array}{lll} \mathsf{Rhs} \longrightarrow 1 & & \text{the completely unknown value} \\ \quad \mid 0 & & \text{the contradictory value} \\ \quad \mid c(v_1, \ldots, v_k) & & \text{some constructor } c \text{ applied to representatives of} \\ & & \text{equivalence classes of variables.} \end{array}$$

Formally we define analysis environments by

$$\mathsf{Env}' = (\mathsf{Var} \longrightarrow \mathsf{Rhs}) \times \mathcal{P}(\mathsf{Var} \times \mathsf{Var}).$$

Each $\rho = (\rho_1, \rho_2) \in \mathsf{Env}'$ 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 $[v]_{\rho_2}$.

An environment stores annotated values. An annotated value $d \in \mathsf{AVal}$ is a tree whose nodes are decorated with a set of variables and a constructor symbol. If two variables appear at the same node, the variables are considered equivalent or — in other words — they are aliases for each other. Thus we take AVal as the greatest solution (wrt. set inclusion) of the equation

$$\mathsf{AVal} = \mathcal{P}\mathsf{Var} \times (\{0, 1\} + \mathsf{Con} \times \mathsf{AVal}^*)$$

where $\mathcal{P}$Var denotes the powerset of Var, * is formation of finite sequences, and $+$ is disjoint union.

Define an ordering $\leq$ on AVal by

$$\begin{array}{c} (S_1, 0) \leq (S_2, x) \Leftrightarrow S_1 \supseteq S_2 \\ (S_1, x) \leq (S_2, 1) \Leftrightarrow S_1 \supseteq S_2 \\ (S_1, c(d_1, \ldots, d_k)) \leq (S_2, c(d'_1, \ldots, d'_k)) \Leftrightarrow S_1 \supseteq S_2 \wedge \forall 1 \leq i \leq k : d_i \leq d'_i \end{array}$$

Proposition 1. $(\mathsf{AVal}, \leq)$ *forms a lattice.*

The least element of AVal is $(\mathsf{Var}, 0)$, the top element is $(\emptyset, 1)$. For example, the greatest lower bound operation $\sqcap$ on AVal is defined as

$$\begin{array}{ll} (S_1, 0) \sqcap (S_2, x) & = (S_1 \cup S_2, 0) \\ (S_1, x) \sqcap (S_2, 1) & = (S_1 \cup S_2, x) \\ (S_1, c(\ldots)) \sqcap (S_2, c'(\ldots)) & = (S_1 \cup S_2, 0) \quad \text{if } c \neq c' \\ (S_1, c(d_1, \ldots, d_k)) \sqcap (S_2, c(d'_1, \ldots, d'_k)) & = (S_1 \cup S_2, c(d_1 \sqcap s'_1, \ldots, d_k \sqcap d'_k)) \end{array}$$

There are two functions that manipulate environments namely *lookup* and *enter* to lookup and enter bindings in an environment. Both functions preserve the conditions 1.–3. above.

An enquiry to the environment yields an annotated value.

$$
\begin{array}{l}
\mathit{lookup}\colon \mathsf{Var} \to \mathsf{Env}' \to \mathsf{AVal} \\
\mathit{lookup}\ v\ \rho = ([v]_{\rho_2}, w) \\
\qquad \text{where } (\rho_1, \rho_2) = \rho \\
\qquad\qquad w = 1 \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \rho_1 v = 1 \\
\qquad\qquad w = c(\mathit{lookup}\ v_1\ \rho, \ldots, \mathit{lookup}\ v_k\ \rho) \text{ if } \rho_1 v = c(v_1, \ldots, v_k)
\end{array}
$$

Entering an annotated value into an environment does not cause aliasing of previously not aliased variables. Existing equivalence classes are enlarged as well as — possibly — some new classes are added.

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

In the second case for *enter* the variables $n_1, \ldots, n_k$ are fresh variables, *i.e.*, they do not appear anywhere else in the environment.

Symbolic evaluation of an expression to an annotated value is defined by the non-standard semantics $\mathcal{E}'$. Its type is

$$\mathcal{E}'\colon \mathsf{Exp} \to \mathsf{FEnv}' \to \mathsf{Env}' \to \mathsf{AVal}$$

so that it takes an expression e, a function environment $\psi \in \mathsf{FEnv}'$, an environment $\rho \in \mathsf{Env}'$, and yields an annotated value AVal that describes the shape of the result of evaluating e with values bound to the variables whose shapes are as described by ρ. A function environment FEnv′ is a mapping from function names Fun to functions over annotated values, *i.e.*, $\mathsf{FEnv}' = \mathsf{Fun} \to \mathsf{AVal}^n \to \mathsf{AVal}$. The greatest function environment $\psi_0 \in \mathsf{FEnv}'$ is $\psi_0(f)(d_1, \ldots, d_n) = (\emptyset, 1)$ for all functions.

$$
\begin{array}{ll}
\mathcal{E}'[\![\ v\ ]\!]\psi\rho & = \mathit{lookup}\ v\ \rho \\
\mathcal{E}'[\![\ c(e_1, \ldots, e_k)\ ]\!]\psi\rho & = (\emptyset, c(\mathcal{E}'[\![\ e_1\ ]\!]\psi\rho, \ldots, \mathcal{E}'[\![\ e_k\ ]\!]\psi\rho)) \\
\mathcal{E}'[\![\ f(e_1, \ldots, e_n)\ ]\!]\psi\rho & = \psi(f)(\mathcal{E}'[\![\ e_1\ ]\!]\psi\rho, \ldots, \mathcal{E}'[\![\ e_n\ ]\!]\psi\rho) \\
\mathcal{E}'[\![\ \texttt{let}\ v = e_1\ \texttt{in}\ e_2\ ]\!]\psi\rho & = \mathcal{E}'[\![\ e_2\ ]\!]\psi(\mathit{enter}\ v\ (\mathcal{E}'[\![\ e_1\ ]\!]\psi\rho)\ \rho) \\
\mathcal{E}'[\![\ \texttt{case}\ e_0\ \texttt{of} \ldots c_j(v_1, \ldots, v_k) : e_j \ldots]\!]\psi\rho = & \\
\quad \text{case } \mathcal{E}'[\![\ e_0\ ]\!]\psi\rho \text{ of} & \\
\quad (vs, c_j(d_1, \ldots, d_k)) : \mathcal{E}'[\![\ e_j\ ]\!]\psi(\mathit{enter}\ v_1\ d_1 \ldots(\mathit{enter}\ v_k\ d_k\ \rho)\ldots) & \\
\quad |(vs, 1) : \bigsqcup_{j=1}^{m} \mathcal{E}'[\![\ e_j\ ]\!]\psi(\mathit{enter}\ n_0\ c_j(\ldots(\{v_j\}, 1)\ldots))\rho) &
\end{array}
$$

Explanation: variables are looked up in the environment. Constructor applications create a new value which is completely unshared at the top, hence the $\emptyset$ at the top node. Function application is handled by a lookup through the function

environment ψ. 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 by means of $\mathcal{E}'$ the value of the `case` expression is the value of e_j. Otherwise all branches are entered with the environment changed to reflect the supposed structure of $\mathcal{E}'[\![\ e_0\ ]\!]$ and the least upper bound of the result is taken. Another possiblity at this place would be to safely approximate the outcome of the `case` expression by $(\emptyset, 1)$.

$\mathcal{E}$ is an abstract interpretation in the sense of Cousot and Cousot [4]. It is an abstraction of a concrete semantics that reveals sharing properties of the graph representation of the values. As outlined in previous work the standard semantics can be constructed as an abstraction thereof [9].

4 Finding specializable calls

The analysis function $\mathcal{C}$ finds specializable calls by employing the annotated value semantics of the preceding section 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. The type of $\mathcal{C}$ is

$$\mathcal{C}\colon \mathsf{Exp} \to \mathsf{FEnv}' \to \mathsf{Env}' \to \mathcal{P}(\mathsf{Fun} \times \mathsf{AVal}^*)$$

i.e., $\mathcal{C}$ takes an expression $e \in \mathsf{Exp}$ to analyze for calls with partially known arguments, a function environment $\psi \in \mathsf{FEnv}'$, and an environment $\rho \in \mathsf{Env}'$ the analyzed expression e. Its result is a set of function calls coded as tuples consisting of the name of the called function (Fun) and a list of argument shapes as annotated values AVal^*.

$$
\begin{array}{l}
\mathcal{C}[\![\ x\ ]\!]\psi\rho = \emptyset \\
\mathcal{C}[\![\ c(e_1,\ldots,e_k)\ ]\!]\psi\rho = \bigcup_{i=1}^{k} \mathcal{C}[\![\ e_i\ ]\!]\psi\rho \\
\mathcal{C}[\![\ f(e_1,\ldots,e_n)\ ]\!]\psi\rho = \bigcup_{i=1}^{n} \mathcal{C}[\![\ e_i\ ]\!]\psi\rho \cup \{(f, \ldots strip(\mathcal{E}'[\![\ e_i\ ]\!]\psi\rho)\ldots))\} \\
\mathcal{C}[\![\ \texttt{let}\ v = e_1\ \texttt{in}\ e_2\ ]\!]\psi\rho = \mathcal{C}[\![\ e_1\ ]\!]\psi\rho \cup \mathcal{C}[\![\ e_2\ ]\!]\psi(enter\ v\ (\mathcal{E}'[\![\ e_1\ ]\!]\psi\rho)\ \rho) \\
\mathcal{C}[\![\ \texttt{case}\ e_0\ \texttt{of}\ldots c_j(v_1,\ldots,v_k) : e_j \ldots\ ]\!]\psi\rho = \\
\quad \mathcal{C}[\![\ e_0\ ]\!]\psi\rho \cup \\
\quad \text{case}\ \mathcal{E}'[\![\ e_0\ ]\!]\psi\rho\ \text{of} \\
\qquad (vs, c_j(d_1,\ldots,d_k)) : \mathcal{C}[\![\ e_j\ ]\!]\psi(enter\ v_1\ d_1 \ldots (enter\ v_k\ d_k\ \rho)\ldots) \\
\qquad |(vs, 1) : \bigcup_{j=1}^{m} \mathcal{C}[\![\ e_j\ ]\!]\psi(enter\ n_0\ c_j(\ldots(\{v_j\}, 1)\ldots)\rho)
\end{array}
$$

Explanation: the equations for variables, constructor applications, and `let`-expressions only serve to collect call patterns from their subexpressions. 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 function arguments. 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 below. At a `case`-expression symbolic evaluation $\mathcal{E}'$ 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.

$$
\begin{array}{l}
\mathit{strip}\colon \mathsf{AVal} \to \mathsf{AVal} \\
\mathit{strip}\ (vs, 1) \qquad\qquad = (\emptyset, 1) \\
\mathit{strip}\ (vs, c(d_1, \ldots, d_k)) = (\emptyset, c(\mathit{strip}\ d_1, \ldots, \mathit{strip}\ d_k))
\end{array}
$$

The outcome of the call analysis is usable even if we take ψ_0, the greatest function environment, for ψ. So there is no need to do a fixpoint computation at all. However, information may be extracted from comparing the results of adjacent iteration steps. The information can be used to guide an unfolding mechanism [2], which in turn can uncover more specializable calls. Such a procedure can lead to non-termination of the analyzer, a well known phenomenon from partial evaluation.

4.1 Examples

As an example we analyze the set of calls and their associated argument shapes in the body of the function `last`. Initially nothing is known about the parameter L. Thus the initial environment is $\rho_0 = ([L \mapsto 1], \{(L, L)\})$ and $\psi = \psi_0$.

$$
\begin{array}{l}
C[\![\ \mathtt{case}\ L\ \mathtt{of}\ldots\ ]\!]\psi\ \rho_0\ = \\
\quad C[\![\ L\ ]\!]\psi\ \rho_0 \cup \qquad (*\ d = \mathcal{E}'[\![\ L\ ]\!]\psi\rho_0 = 1\ *) \\
\quad C[\![\ \mathtt{raise}\ \ldots\ ]\!]\psi(\mathit{enter}\ y_1\ (\{L\}, \mathtt{nil})\ \rho_0) \cup \\
\quad C[\![\ \mathtt{case}\ L'\ \mathtt{of}\ldots\ ]\!]\psi(\mathit{enter}\ y_2\ (\{L\}, \mathtt{cons}((\{x\}, 1), (\{L'\}, 1)))\ \rho_0) \\
\quad (*\ \rho_1 = \mathit{enter}\ldots\rho_0 = [L = y_2 = \mathtt{cons}(x, L'), x = 1, L' = 1]\ *) \\
= \emptyset \cup \emptyset \cup C[\![\ \mathtt{case}\ L'\ \mathtt{of}\ldots\ ]\!]\psi\rho_1 \\
= C[\![\ L'\ ]\!]\psi\ \rho_1 \cup \\
\quad \text{let } d = \mathcal{E}'[\![\ L'\ ]\!]\psi\ \rho_1 \text{ in} \qquad (*\ d = (\{L'\}, 1)\ *) \\
\quad C[\![\ x\ ]\!]\psi\ [y_2 = L = \mathtt{cons}(x, L'), x = 1, L' = \mathtt{nil}] \\
\quad \cup C[\![\ \mathtt{last}\ L'\ ]\!]\psi\ [y_2 = L = \mathtt{cons}(x, L'), x = 1, y_3 = L' = \mathtt{cons}(x', L''), x' = 1, L'' = 1] \\
= \emptyset \cup \emptyset \cup C[\![\ L'\ ]\!] \ldots \cup \{(f, \mathit{strip}(\mathcal{E}'[\![\ L'\ ]\!]\psi\ \rho_2))\} \\
= \{(f, (\emptyset, \mathtt{cons}((\emptyset, 1), (\emptyset, 1))))\}
\end{array}
$$

With this information a specialized version of the function `last` can be generated. We apply a technique from partial evaluation called arity raising where one parameter is replaced by many parameters. In the literature arity raising is applied to replacing an argument pair by two single arguments (cf. [8]) whereas we employ a conditional arity raising. Only if it is known that some argument is a constructor term with a certain top constructor we supply the arguments of the constructor in place of the term as arguments to the function.

In our example we have two choices. The argument $L' = \mathtt{cons}(x', L'')$ can either be replaced by the constructor arguments x' and L'' or by L' and L''. The first choice is the one shown in the introduction, which is generated almost verbatim by the specializer presented in the next section. The other choice could be even more advantageous, since the corresponding function even avoids accessing the list elements unless it is forced to do so. But more information is required to make that choice.

We are grateful to one of the referees for the following interesting example. Consider a function that merges two ascending lists of numbers.

```
fun merge (xs as xh::xt) (ys as yh::yt) =
        if xh <= yh then xh::merge xt ys else yh::merge xs yt
  | merge ...
```

At either recursive invocation of **merge** that **ys** or **xs**, respectively, are non-empty. Our analysis detects this fact and generates two mutually recursive auxiliary functions which completely avoid the redundant tests. The stripped output of the call analysis $\mathcal{C}$ on the body of merge is

$$\{(\texttt{merge}, 1 :: (1,1)), (\texttt{merge}, :: (1,1)\ 1)\}$$

and one of the generated functions is (transcribed in pattern matching notation)

```
fun merge_1_11 (xs as xh::xt) yh yt =
        if xh <= yh then xh::merge_1_11 xt yh yt
                    else yh::merge_11_1 xh xt yt
  | merge_1_11 nil yh yt = yh::yt
```

5 Arity raising and specialization

Suppose we are given a set of function definitions and the outcome of the call analysis $C_f = \mathcal{C}[\![\ e_f\ ]\!] \ldots$ on all definitions in $C = \bigcup_{f \in \mathsf{Fun}} C_f$. We will then select a set $C' \subseteq C$ with the requirement that for each $(f, p_1 \ldots p_n) \in C'$ there is some $p_i \neq 1$. The selection for C' must ensure that the specialized functions do not exhibit new call patterns, since new call patterns would cause another specialization phase, which could lead to a non-terminating process. Also if arbitrary call patterns are allowed we will end up in delaying unavoidable data constructions while only creating long argument lists. Only the construction of those poarts of a structure that are certainly decomposed or tested should be avoided or at least delayed. The analyses $\mathcal{T}$ and $\mathcal{R}$ below give information on the tested part of the arguments when given information on the shape of the arguments (a call pattern found by the $\mathcal{C}$-analysis above).

The definitions of $\mathcal{R}^{\langle x_1,\ldots,x_n\rangle}, \mathcal{T}^{\langle x_1,\ldots,x_n\rangle} : \mathsf{Exp} \to \mathsf{FEnv}' \to \mathsf{Env}' \to \mathsf{AVal}^n$ are as follows with $\overline{x} = \langle x_1, \ldots, x_n\rangle$ and the operations on AVal pointwise extended to AVal^n.

$$
\begin{array}{ll}
\mathcal{T}^{\overline{x}}[\![\ e\ ]\!]\psi\rho & = \langle \ldots, \mathit{lookup}\ x_i\ \rho, \ldots\rangle \sqcap \mathcal{R}^{\overline{x}}[\![\ e\ ]\!]\psi\rho \\
\\
\mathcal{R}^{\overline{x}}[\![\ v\ ]\!]\psi\rho & = \langle(\emptyset,1), \ldots, (\emptyset,1)\rangle \\
\mathcal{R}^{\overline{x}}[\![\ c(\ldots e_i \ldots)\ ]\!]\psi\rho & = \bigsqcap_i \mathcal{R}^{\overline{x}}[\![\ e_i\ ]\!]\psi\rho \\
\mathcal{R}^{\overline{x}}[\![\ f(\ldots e_i \ldots)\ ]\!]\psi\rho & = \bigsqcap_i \mathcal{R}^{\overline{x}}[\![\ e_i\ ]\!]\psi\rho \\
\mathcal{R}^{\overline{x}}[\![\ \texttt{let}\ v = e_1\ \texttt{in}\ e_2\ ]\!]\psi\rho & = \mathcal{R}^{\overline{x}}[\![\ e_1\ ]\!]\psi\rho \sqcap \mathcal{T}^{\overline{x}}[\![\ e_2\ ]\!]\psi(\mathit{enter}\ v\ (\mathcal{E}'[\![\ e_2\ ]\!]\psi\rho)\ \rho \\
\mathcal{R}^{\overline{x}}[\![\ \texttt{case}\ e_0\ \texttt{of} \ldots c_j(v_1,\ldots,v_k) : e_j \ldots]\!]\psi\rho = & \\
\qquad \texttt{case}\ \mathcal{E}'[\![\ e_0\ ]\!]\psi\rho\ \texttt{of} & \\
\qquad (vs, c_j(d_1,\ldots,d_k)) : \mathcal{T}^{\overline{x}}[\![\ e_j\ ]\!]\psi(\mathit{enter}\ v_1\ d_1 \ldots (\mathit{enter}\ v_k\ d_k\ \rho)\ldots) & \\
\qquad |(vs,1) : \bigsqcup_j \mathcal{T}^{\overline{x}}[\![\ e_j\ ]\!]\psi(\mathit{enter}\ n_0\ (vs, c_j(\ldots(\{v_j\},1)\ldots))\rho) &
\end{array}
$$

Explanation: the function $\mathcal{R}^{\overline{x}}$ only provides the control structure for the analysis. The function $\mathcal{T}^{\overline{x}}$ is called at every update of the environment. It keeps track of

bindings of the variables $\overline{x}$ and merges their current values with the recursive result from $\mathcal{R}$.

It remains to extract the tested part of the patterns from the result of $\mathcal{T}$. We define the closure operator *close* to yield an upper approximation to the tested part.

$$
\begin{array}{l}
close: \mathcal{P}\mathsf{Var} \to \mathsf{AVal} \to \mathsf{AVal} \\
close\ V\ (vs, 1) \qquad\qquad = (V \cap vs, 1) \\
close\ V\ (c(d_1, \ldots, d_k)) = \text{let } d_i' = (vs_1, x_i) = close\ V\ d_i \text{ for } 1 \le i \le k \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{if } \bigcup vs_i = \emptyset \text{ then } (V \cap vs, 1) \\
\qquad\qquad\qquad\qquad\quad \text{else } (V \cap vs, c(d_1', \ldots, d_k'))
\end{array}
$$

We only create specialized versions for tested part of the call patterns, *i.e.*

$$
\begin{array}{l}
C' = \{(f, p_1' \ldots p_n') \mid (f, p_1 \ldots p_n) \in C, \\
\qquad\qquad \langle p_1', \ldots, p_n' \rangle = (strip \circ close\ \mathrm{var}(e_f) \circ \mathcal{T}^{\langle x_1, \ldots, x_n \rangle}[\![\ e_f\ ]\!]\psi) \\
\qquad\qquad\qquad\qquad (enter\ x_1\ p_1 \ldots (enter\ x_n\ p_n\ \bot_{\mathsf{Env}'})) \sqcup \langle p_1, \ldots, p_n \rangle\}
\end{array}
$$

For the call patterns mentioned in C' we generate specialized functions as follows. First the function body, say e_f, is transformed in such a way that each intermediate value is bound to a variable. This is called transformation to sequential form in [5] and can be combined with common subexpression elimination [1]. Sequential form SExp is defined by the grammar

$$
\begin{array}{rl}
s \to & v \\
\mid & \texttt{let } v = c(v_1, \ldots, v_k) \texttt{ in } s \\
\mid & \texttt{let } v = f(v_1, \ldots, v_n) \texttt{ in } s \\
\mid & \texttt{case } v \texttt{ of } \ldots c_j(v_1, \ldots, v_k) : s_j \ldots
\end{array}
$$

where $v, v_i \in \mathsf{Var}$, $c, c_j \in \mathsf{Con}$, $f \in \mathsf{Fun}$, and $s, s_j \in \mathsf{SExp}$. We will also assume that all variables have unique names.

$\mathcal{S}: \mathsf{SExp} \to \mathsf{FEnv}' \to \mathsf{Env}' \to \mathsf{Env}' \to \mathcal{P}(\mathsf{Fun} \times \mathsf{AVal}^*) \to \mathsf{SExp}$ is the specialization function. It is applied to the sequentialized expression ($\mathcal{S}$ is described below). In the last step we remove unused variables from the resulting expression.

The arguments of $\mathcal{S}[\![\ e\ ]\!]\psi\rho\sigma P$ have the following meaning. The (sequentialized) expression e is the right hand side of a function definition. The function environment $\psi \in \mathsf{FEnv}'$ is needed to predict the outcome of `case` expressions. $\rho \in \mathsf{Env}'$ is the usual environment. $\sigma \in \mathsf{Env}'$ is another environment that keeps track of the arity raising process. If σ is defined on v it means that $\sigma(v)$ is the known part of the value bound to v. Because of our special environment structure we can deduce the variables that are bound to substructures of v's value from the entry for v. The functions that will have specialized versions are described by P as a set of function symbols with (a list of) call patterns.

Since the output of $\mathcal{S}$ is an expression we need to carefully distinguish our meta notation from generated program text. We make the distinction by underlining the generated program text.

$$\mathcal{S}[\![\ v\ ]\!]\psi\rho\sigma P = \begin{cases} \mathit{build}\ p \text{ if } p \equiv \mathit{lookup}\ v\ \sigma \\ \underline{v} \qquad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\ \texttt{let}\ v = c(v_1,\ldots,v_k)\ \texttt{in}\ s\ ]\!]\psi\rho\sigma P =$$
$$\quad \texttt{let}\ v = c(v_1,\ldots,v_k)\ \texttt{in}\ \mathcal{S}[\![\ s\ ]\!]\psi(\mathit{enter}\ v\ (\emptyset, c((\{v_1\},1),\ldots,(\{v_k\},1)))\ \rho)\sigma P$$

$$\mathcal{S}[\![\ \texttt{let}\ v = f(v_1,\ldots,v_n)\ \texttt{in}\ s\ ]\!]\psi\rho\sigma P =$$
$$\quad \text{choose } (f, p_1 \ldots p_n) \in P \text{ minimal where } \mathit{MATCH}(p_i, \mathit{lookup}\ v_i\ \rho) \text{ in}$$
$$\quad \underline{\texttt{let}\ v = f(\mathit{extract}(p_1,d_1)\_\ldots\_\mathit{extract}(p_n,d_n))\ \texttt{in}}\ \mathcal{S}[\![\ s\ ]\!]\psi(\mathit{enter}\ v\ (\emptyset,1)\ \rho)\ \sigma P$$

$$\mathcal{S}[\![\ \texttt{case}\ v\ \texttt{of}\ \ldots c_j(v_1,\ldots,v_k) : s_j \ldots\ ]\!]\psi\rho\sigma P =$$
$$\quad \text{case } \mathit{lookup}\ v\ \rho \text{ of}$$
$$\quad (vs, c_j(d_1,\ldots,d_k)) : \mathcal{S}[\![\ s_j\ ]\!]\psi(\mathit{enter}\ v_1\ d_1\ (\mathit{enter}\ldots\rho))\sigma P$$
$$\mid (vs, 1) : \underline{\texttt{case}\ v\ \texttt{of}}\ \ldots$$
$$\qquad \underline{c_j(v_1,\ldots,v_k)} : \mathcal{S}[\![\ s_j\ ]\!]\psi(\mathit{enter}\ v\ (vs, c_j(\ldots(\{v_j\},1)\ldots))\rho)\sigma P$$

We have used the auxiliary functions *build*: AVal → SExp to build concrete values from annotated values, *MATCH*: AVal × AVal → Boolean to dispatch **case** branches and *extract*: AVal × AVal → Var* to flatten argument expressions. They are defined as follows.

$$\begin{aligned}
&\mathit{build}\ (\{v,\ldots\},1) &&= \underline{v} \\
&\mathit{build}\ (vs, c((vs_1,x_1),\ldots,(vs_k,x_k))) &&= \text{let } v_i \in vs_i \quad v \in vs \text{ in} \\
&\qquad \mathit{build}\ (\{v_1\},x_1)\ \ldots\ \mathit{build}\ (\{v_k\},x_k)\ \underline{\texttt{let}\ v = c(v_1,\ldots,v_k)\ \texttt{in}} \\
&\mathit{MATCH}((S,1),d) &&= \text{true} \\
&\mathit{MATCH}((S,c(p_1,\ldots,p_k)),(S',c(d_1,\ldots,d_k))) &&= \textstyle\bigwedge_{i=1}^{k} \mathit{MATCH}(p_i,d_i) \\
&\mathit{MATCH}(p,d) &&= \text{false} \\
&\mathit{extract}((S,1),(\{v,\ldots\},d)) &&= \underline{v} \\
&\mathit{extract}((S,c(p_1,\ldots,p_k)),(S',c(d_1,\ldots,d_k)) &&= \mathit{extract}(p_1,d_1)_\ldots_\mathit{extract}(p_k,d_k) \\
&\mathit{extract}(p,d) &&= \text{abort}
\end{aligned}$$

Notice that $\mathit{extract}(p,d)$ is only called if $\mathit{MATCH}(p,d)$ is true. But that means that the abort case will not occur in the evaluation of *extract*.

6 Related work

Wadler (and others subsequently) worked on deforestation [10, 3, 6]. Deforestation is an algorithm to eliminate intermediate trees by symbolic composition. Although a different goal is pursued deforestation also avoids some tests by delaying the construction of results. In contrast, specialization with arity raising delays the construction of values that are passed as parameters.

Romanenko deals with arity raising in the context of partial evaluation and program specialization [8]. 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 single arguments. Here arity raising is conditional, since the top constructor of the argument which has to be decomposed must be known.

7 Conclusion and future work

We have presented an analysis that uncovers redundant tests caused by function declarations with pattern matching. Abstract interpretation yields function calls with pattern arguments and methods known from partial evaluation are employed to generate specialized functions that avoid the redundant tests. The analysis is simple and cheap enough to be incorporated into a compiler. It is shown with several examples that many interesting functions can be improved by the proposed technique.

Although demonstrated here in the context of strict functional languages, avoiding redundant test might prove even more beneficial for non-strict functional languages with lazy evaluation. In fact, all evaluation is driven by pattern matching in implementations like the the STG-machine [7] and avoiding a single constructor test really spares two tests: the test whether the argument closure is evaluated and the dispatch according to the constructor number.

Further directions of work include measurements with an implementation, the extension of the analysis to higher-order programs and the exploration of the connections to fusion and deforestation algorithms.

Acknowledgements The comments of the anonymous referees helped to improve the presentation of the paper. Special thanks to one of the referees for the `merge` example.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.
3. W.-N. Chin. Safe fusion of functional expressions. In *Proceedings Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, June 1992.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*. ACM, 1977.
5. C. K. Gomard and P. Sestoft. Globalization and live variables. In *Proc. PEPM '92*, pages 166–177, New Haven, June 1991. ACM. SIGPLAN Notices v26,9.
6. G. W. Hamilton and S. B. Jones. Extending deforestation for first order functional programs. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 134–145, Portree, Isle of Skye, Aug. 1992. Springer-Verlag, Berlin.
7. S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.
8. S. A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *ESOP 1990*, pages 341–360. Springer Verlag, 1990. LNCS 432.
9. P. Thiemann. A safety analysis for functional programs. In D. Schmidt, editor, *Proc. PEPM '93*, pages 133–144, Copenhagen, Denmark, June 1993. ACM.
10. P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Comput. Sci.*, 73(2):231–248, 1990.
11. Å. Wikström. *Functional Programming Using Standard ML.* Prentice Hall, 1987.