A Lazy Abstract Machine Based on a Pattern Matching Calculus

Pedro Vasconcelos
LIACC
Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
Porto, Portugal
pbv@dcc.fc.up.pt

Rodrigo Marques
LIACC
Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
Porto, Portugal
rmarques@fc.up.pt

ABSTRACT

This paper presents the derivation of an abstract machine for λPMC, a small lazy functional language based on the pattern matching calculus of Kahl. The motivation for this work was the implementation of a web-based interpreter for a small Haskell-like language designed for teaching.

Classical presentations of pattern matching in functional languages define the operational semantics by translation into case expressions with simple patterns. The translation is non-local and some optimizations may be necessary to avoid duplicating expressions. By contrast, the translation of a language such as Haskell into λPMC is much more straightforward, preserving a one-to-one relation with the source language.

We define the syntax and two semantics for λPMC: a big-step lazy semantics in the style of Launchbury and a small-step abstract machine in the style of the Sestof's machines. Finally, we prove the correspondence between the two semantics and present some examples from a prototype implementation.

ACM Reference Format:
Pedro Vasconcelos and Rodrigo Marques. 2023. A Lazy Abstract Machine Based on a Pattern Matching Calculus. In *Proceedings of 35th Symposium on Implementation and Application of Functional Languages (IFL'2023)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACKNOWLEDGMENTS

This work was financially supported by the Artificial Intelligence and Computer Science Laboratory (LIACC) under the base research grant UIDB/00027/2020 of the Fundação para a Ciência e Tecnologia funded by national funds through the FCT/MCTES (PIDDAC).

1 INTRODUCTION

This paper presents the derivation of an abstract machine for λPMC, a small lazy functional language based on the pattern matching calculus of Kahl [4]. The motivation for this work was the implementation of a web-based step-by-interpreter for a small Haskell-like language designed for teaching (available at https://pbv.github.io/haskelite/). Hence, our objective is *not* a more efficient compilation technique for lazy languages; instead, we wanted an operational model where pattern matching can be more easily related to the source code written by the programmer.

IFL'2023, August 2023, Braga, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Classical presentations of pattern matching in functional languages define the operational semantics by translation into case expressions with simple patterns [3, 11]. For example, consider the following Haskell function that checks whether a list is "short":

```
isShort (x:y:ys) = False
isShort ys       = True
```

The translation into simple case expressions is:

```
isShort xs = case xs of
                (x:xs') -> case xs' of
                              (y:ys) -> False
                              [] -> True
                [] -> True
```

Nested patterns such as `(x:y:ys)` must be translated into nested case expressions with simple patterns and matches are made complete by introducing missing constructors.

By contrast, the translation of *isShort* into λPMC is straightforward:

$$isShort = \lambda((x : y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid ys \Rightarrow \lceil \mathsf{True} \rceil)$$

Compared to the version with case expressions, the λPMC translation preserves a closer relation to the original source program: each equation corresponds to one alternative in a *matching abstraction* and nested patterns are preserved.

Furthermore, translation into simple cases may require some optimizations to avoid duplicating expressions whenever patterns overlap. Consider the following example from [11] (where A and B are some unspecified expressions):

```
unwieldy [] [] = A
unwieldy xs ys = B xs ys
```

The translation into simple cases duplicates the sub expression $B\ xs\ ys$:

```
unwieldy xs ys = case xs of
                   [] -> case ys of
                          [] -> A
                          (y:ys') -> B xs ys
                   (x:xs') -> B xs ys
```

By contrast, the translation to λPMC preserves the single occurrence of the sub expression:

$$\begin{aligned} unwieldy = \lambda\,(&[\,] \Rightarrow [\,] \Rightarrow \lceil A \rceil \\ \mid\ & xs \Rightarrow ys \Rightarrow \lceil B\ xs\ ys \rceil) \end{aligned}$$

Furthermore, λPMC can easily handle other extensions to pattern matching, such as as-patterns, pattern guards, view patterns and lambda cases.

e		expressions	
e	$::=$	x	variable
	$\mid$	$e_1\ e_2$	function application
	$\mid$	λm	matching abstraction
	$\mid$	$c(e_1, \dots, e_n)$	constructor application
	$\mid$	$\text{let } \{x_i = e_i\} \text{ in } e'$	let(rec) bindings
m		matchings	
m	$::=$	$\lceil e \rceil$	return expression
	$\mid$	$\lightning$	failure
	$\mid$	$p \Rightarrow m$	match pattern
	$\mid$	$e \triangleright m$	argument supply
	$\mid$	$m_1 \mid m_2$	alternative
p		patterns	
p	$::=$	x	variable pattern
	$\mid$	$c(p_1, \dots, p_n)$	constructor pattern

Figure 1: Syntax of λPMC.

The contributions presented in this paper are:

(1) the definition of λPMC;
(2) a big-step operational semantics for λPMC;
(3) an abstract machine (i.e. a small-step semantics) for λPMC;
(4) a proof sketch of correctness of the abstract machine against the big-step semantics.

The remaining of this paper is structured as follows: Section 2 defines the syntax of λPMC and reviews the reduction rules of the Kahl's pattern matching calculus. Section 3 defines normal forms and presents a big-step semantics for language. Section 4 defines configurations and small-step reduction rules of an abstract machine for λPMC and presents some examples. Section 5 presents proof sketch for correctness with respect to the big-step semantics. Section 6 discusses related work. Finally, Section 7 highlights directions for further work.

2 SYNTAX

2.1 Expressions and matchings

Figure 1 defines λPMC, a small functional language with syntactical categories for *expressions*, *matchings* and *patterns*. Matchings and patterns are based on the PMC calculus of Kahl [4]. The only extension is the let ... in ... construct for defining (possibly) recursive local bindings.

Matchings can be a return expression $\lceil e \rceil$ (signaling a successful match), the matching failure $\lightning$, a pattern match $p \Rightarrow m$ (expecting a single argument that must match pattern p), an argument application $e \triangleright m$, or the choice $m_1 \mid m_2$ between two matchings. Note that patterns p can be nested; for example `(x:(y:ys))` is a valid pattern (using infix notation for the list constructor).

Lambda abstraction is subsumed by matching abstraction: $\lambda x.\, e$ is equivalent to $\lambda(x \Rightarrow \lceil e \rceil)$. Case expressions are also subsumed by matching abstraction; the expression

$$\text{case } e_0 \text{ of } \{p_1 \to e_1; \dots; p_n \to e_n\}$$

is equivalent to

$$\lambda(e_0 \triangleright (p_1 \Rightarrow \lceil e_1 \rceil \mid \dots \mid p_n \Rightarrow \lceil e_n \rceil))\ .$$

Note that matchings in λPMC allow both patterns and applications, so they may represent arbitrary expressions and not just lambda abstractions.

2.2 Reduction relations

Following [4] we review the pattern matching calculus as two *redex* reduction relations, namely, $\underset{\mathsf{E}}{\longrightarrow}$ between expressions and $\underset{\mathsf{M}}{\longrightarrow}$ between matchings. We leave the definition of an evaluation strategy and normal forms to Section 3, so this does not yet define a complete semantics.

We use the notation $m[e'/x]$ for the substitution of free occurrences of variable x for an expression e' in the matching m. The definitions of free occurrences and substitution are standard and are therefore omitted.

The first two rules state that failure is the left unit for $\mid$ while return is the left zero:

$$\lightning \mid m \underset{\mathsf{M}}{\longrightarrow} m \qquad (\lightning\ \mid)$$

$$\lceil e \rceil \mid m \underset{\mathsf{M}}{\longrightarrow} \lceil e \rceil \qquad (\lceil\rceil\ \mid)$$

The next rule states that matching abstraction built from return expressions reduce to the underlying expression:

$$\lambda \lceil e \rceil \underset{\mathsf{E}}{\longrightarrow} e \qquad (\lambda\ \lceil\rceil)$$

Unlike [4], we do not have a reduction rule for $\lambda\lightning$ because we do not have an "empty expression" corresponding to matching failure.

Application of a matching abstraction reduces to argument supply to the matching. Dually, argument supply to a return expression reduces to application of the expression.

$$(\lambda m)\ a \underset{\mathsf{E}}{\longrightarrow} \lambda(a \triangleright m) \qquad (\lambda @)$$

$$a \triangleright \lceil e \rceil \underset{\mathsf{M}}{\longrightarrow} \lceil e\ a \rceil \qquad (\triangleright\ \lceil\rceil)$$

The following rule propagates a matching failure through an argument supply.

$$e \triangleright \lightning \underset{\mathsf{M}}{\longrightarrow} \lightning \qquad (\triangleright \lightning)$$

Next, argument supply distributes through alternatives:

$$e \triangleright (m_1 \mid m_2) \underset{\mathsf{M}}{\longrightarrow} (e \triangleright m_1) \mid (e \triangleright m_2) \qquad (\triangleright\ \mid)$$

The following rules handle argument supply to patterns:

$$e \triangleright x \Rightarrow m \underset{\mathsf{M}}{\longrightarrow} m[e/x] \qquad (\triangleright x)$$

$$\begin{aligned} c(e_1, \dots, e_n) \triangleright c(p_1, \dots, p_n) \Rightarrow m &\underset{\mathsf{M}}{\longrightarrow} \\ e_1 \triangleright p_1 \Rightarrow \dots \Rightarrow e_n \triangleright p_n &\Rightarrow m \end{aligned} \qquad (c \triangleright c)$$

$$c'(e_1, \dots, e_k) \triangleright c(p_1, \dots, p_n) \Rightarrow m \underset{\mathsf{M}}{\longrightarrow} \lightning \qquad (c \triangleright c')$$

In the last rule we assume $c \neq c'$.

2.3 Examples

We will present some examples of the translation of Haskell definitions into λPMC. The emphasis is on illustrating the correspondence between the Haskell source and the translated notation.

Consider the following definition of the *zipWith* function that combines two lists using a functional argument:

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs'    ys'    = []
```

This can be translated directly to λPMC:

$$\begin{aligned} zipWith = \lambda\, (&f \Rightarrow (x : xs) \Rightarrow (y : ys) \Rightarrow \lceil f\ x\ y : zipWith\ f\ xs\ ys\rceil \\ \mid\ &f \Rightarrow xs' \Rightarrow ys' \Rightarrow \lceil [\,]\rceil) \end{aligned}$$

The function argument is handled identically in both branches, so we could factor it out:

$$\begin{aligned} zipWith = \lambda(f \Rightarrow\ &((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil f\ x\ y : zipWith\ f\ xs\ ys\rceil \\ &\mid xs' \Rightarrow ys' \Rightarrow \lceil [\,]\rceil)) \end{aligned}$$

Next, consider a function *nodups* which removes identical contiguous elements from a list; this illustrates the translation of *as-patterns* and *boolean guards*:

```
nodups (x:xs@(y:xs')) | x==y = nodups xs
nodups (x:xs)                = x:nodups xs
nodups [] = []
```

This can be translated to λPMC as a matching abstraction with three alternatives; notice the use of a pattern argument $(x == y)$ to encode the boolean guard [4]:

$$\begin{aligned} nodups = \lambda\, (&(x : xs) \Rightarrow xs \triangleright (y : xs') \Rightarrow \\ &\qquad (x == y) \triangleright \mathsf{True} \Rightarrow \lceil nodups\ xs\rceil \\ \mid\ &(x : xs) \Rightarrow \lceil x : nodups\ xs\rceil \\ \mid\ &[\,] \Rightarrow \lceil [\,]\rceil) \end{aligned}$$

The translation of boolean guards generalizes for arbitrary *pattern guards*; for example:

```
addLookup env v1 v2
  | Just r1 <- lookup env v1
  , Just r2 <- lookup env v2 = r1 + r2
```

translates as:

$$\begin{aligned} addLookup = \lambda\, (&env \Rightarrow v1 \Rightarrow v2 \Rightarrow \\ &\quad lookup\ env\ v1 \triangleright \mathsf{Just}(r1) \Rightarrow \\ &\quad lookup\ env\ v2 \triangleright \mathsf{Just}(r2) \Rightarrow \lceil r1 + r2\rceil) \end{aligned}$$

Pattern guards can also be used to encode *view patterns* [8, 14] that allow matching over abstract data types. Consider a Haskell data type for sequences implemented as a binary tree (also known as "join lists"):

```
data JList a = Empty | Single a | Join (JList a) (JList a)
```

To allow pattern matching but keep the internal representation opaque we define a *view type* for extracting the first element and a tail (a join list) and a *view function*:

```
data JListView a = Nil | Cons a (JList a)
view :: JList a -> JListView a
```

Using this view function, we can define a recursive *length* function without exposing the join list constructors:

```
length :: JList a -> Int
length (view -> Nil) = 0
length (view -> Cons x xs) = 1 + length xs
```

To match a variable `t` with a pattern `view -> patt` we need to evaluate `view t` and match the result against `patt`. This can be translated into λPMC in a straightforward way using pattern guards:

$$\begin{aligned} length = \lambda\, (&t \Rightarrow view\ t \triangleright \mathsf{Nil} \Rightarrow \lceil 0\rceil \\ \mid\ &t \Rightarrow view\ t \triangleright \mathsf{Cons}(h, xs) \Rightarrow \lceil 1 + length\ xs\rceil) \end{aligned}$$

The view function for join lists could be defined as follows; note the use of view patterns in the view function itself:

```
view Empty = Nil
view (Single a) = Cons a Empty
view (Join (view -> Cons h t) y) = Cons h (Join t y)
view (Join (view -> Nil) y) = view y
```

Again the translation in λPMC is straightforward:

$$\begin{aligned} view = \lambda\, (&\mathsf{Empty} \Rightarrow \lceil \mathsf{Nil}\rceil \\ \mid\ &\mathsf{Single}(x) \Rightarrow \lceil \mathsf{Cons}(x, \mathsf{Empty})\rceil \\ \mid\ &\mathsf{Join}(xs, ys) \Rightarrow view\ xs \triangleright \mathsf{Cons}(h, t) \Rightarrow \\ &\qquad\qquad \lceil \mathsf{Cons}(h, \mathsf{Join}(t, ys))\rceil \\ \mid\ &\mathsf{Join}(xs, ys) \Rightarrow view\ xs\ \triangleright \mathsf{Nil} \Rightarrow \lceil view\ ys\rceil) \end{aligned}$$

Finally, the GHC extensions for *LambdaCase* and *LambdaCases* [7] can be trivially encoded as abstraction matchings; for example

```
sign = \case x | x>0 -> 1
               | x==0 -> 0
               | otherwise -> -1
```

translates directly to:

$$\begin{aligned} sign = \lambda(x \Rightarrow\ &((x > 0) \triangleright \mathsf{True} \Rightarrow \lceil 1\rceil \\ &\mid (x == 0) \triangleright \mathsf{True} \Rightarrow \lceil 0\rceil \\ &\mid \mathsf{True} \triangleright \mathsf{True} \Rightarrow \lceil -1\rceil)) \end{aligned}$$

We could also avoid the trivial guard in the last alternative:

$$\begin{aligned} sign = \lambda(x \Rightarrow\ &((x > 0) \triangleright \mathsf{True} \Rightarrow \lceil 1\rceil \\ &\mid (x == 0) \triangleright \mathsf{True} \Rightarrow \lceil 0\rceil \\ &\mid \lceil -1\rceil)) \end{aligned}$$

3 NATURAL SEMANTICS

In this section we define a natural (i.e. big-step) semantics for evaluation of expressions and matchings. The semantics is based in Sestof's revision [13] of Launchbury's semantics for lazy evaluation [6].

3.1 Normalized syntax

As in Launchbury's semantics, we will restrict arguments of applications to be variables; complex arguments must be explicitly named using let bindings. This ensures that we can update a result after evaluation and properly implement call-by-need (i.e. lazy) evaluation.

Figure 2 defines the syntax for normalized λPMC. In the remaining of this paper we assume all expressions and matchings to be normalized.

3.2 Preliminary definitions

Let *heaps* Γ, Δ, Θ to be finite mappings from variables to (possibly unevaluated) expressions. The notation $\Gamma[y \mapsto e]$ in a conclusion extends a heap with an extra entry; dually, this is also used in hypotheses to extract an entry from a heap.

$$
\begin{array}{llll}
e & & \text{expressions} & \\
e & ::= & x & \text{variable} \\
 & | & e\ y & \text{function application} \\
 & | & \lambda m & \text{matching abstraction} \\
 & | & c(y_1, \ldots, y_n) & \text{constructor application} \\
 & | & \mathsf{let}\ \{x_i = e_i\}\ \mathsf{in}\ e' & \text{let(rec) bindings} \\
 & & & \\
m & & \text{matchings} & \\
m & ::= & \lceil e \rceil & \text{return expression} \\
 & | & \lightning & \text{failure} \\
 & | & p \Rightarrow m & \text{match pattern} \\
 & | & y \rhd m & \text{argument supply} \\
 & | & m_1 \mid m_2 & \text{alternative} \\
 & & & \\
p & & \text{patterns (as before)} &
\end{array}
$$

Figure 2: Normalized syntax for λPMC.

Because matchings can encode multi-argument functions and also argument applications, we have to define the results of evaluations accordingly. First we define a syntactical measure $\#m$ of the *arity* of a matching:

$$
\begin{array}{rcll}
\#\lceil e \rceil & = & \#\lightning & = 0 \\
\#(p \Rightarrow m) & = & 1 + \#m & \\
\#(y \rhd m) & = & \max(0, \#m - 1) & \\
\#(m_1 \mid m_2) & = & \#m_1 & = \#m_2 \\
 & & \text{(must have equal arity)} &
\end{array}
$$

The last condition generalizes the Haskell syntax rule that the number of patterns for each clause of a definition must be the same (Section 4.4.3.1 of [9]).

An expression is in weak head normal (whnf) w if it is either a matching abstraction of arity greater than zero or a constructor:

$$
\begin{array}{lll}
w & ::= & \lambda m \quad \text{such that } \#m > 0 \\
 & | & c(y_1, \ldots, y_n)
\end{array}
$$

If $\#m > 0$ then m expects at least one argument, i.e. behaves like a lambda abstraction. If $\#m = 0$ then m is *saturated* (i.e. fully applied) and therefore *not* a whnf.

This definition of weak normal forms implies that partially applied matchings will not be evaluated e.g. $\lambda(z \rhd x \Rightarrow y \Rightarrow \lceil e \rceil)$ is in whnf. This is similar to a partial application in an abstract machine with multi-argument functions e.g. the STG [3]. However, in our case this is done not for efficiency but rather to simplify evaluation by ensuring that we only ever evaluate saturated matchings.

We also define a weak normal form μ for matching results, namely, either a return expression or a failure:

$$\mu \quad ::= \quad \lceil e \rceil \quad | \quad \lightning$$

3.3 Evaluation rules

Evaluation is defined in Figures 3 and 4 by two mutually recursive judgments:

$\Gamma; L; e \Downarrow_{\mathsf{E}} \Delta; w$ Evaluating expression e from heap Γ yields heap Δ and result w;

$\Gamma; L; A; m \Downarrow_{\mathsf{M}} \Delta; \mu$ Evaluating matching m from heap Γ and argument stack A yields heap Δ and result μ.

$\boxed{\Gamma; L; e \Downarrow_{\mathsf{E}} \Delta; w}$ expression evaluation

$$\frac{}{\Gamma; L; w \Downarrow_{\mathsf{E}} \Gamma; w}\ \textsc{Whnf}_{\Downarrow}$$

$$\frac{\#m = 0 \quad \Gamma; L; [\,];\ m \Downarrow_{\mathsf{M}} \Delta; \lceil e \rceil \quad \Delta; L; e \Downarrow_{\mathsf{E}} \Theta;\ w}{\Gamma; L; \lambda m \Downarrow_{\mathsf{E}} \Theta; w}\ \textsc{Sat}_{\Downarrow}$$

$$\frac{\Gamma; L \cup \{y\}; e \Downarrow_{\mathsf{E}} \Delta; w}{\Gamma[y \mapsto e]; L; y \Downarrow_{\mathsf{E}} \Delta[y \mapsto w]; w}\ \textsc{Var}_{\Downarrow}$$

$$\frac{\Gamma; L; e \Downarrow_{\mathsf{E}} \Delta; \lambda m \quad \Delta; L; \lambda(y \rhd m) \Downarrow_{\mathsf{E}} \Theta; w}{\Gamma; L; (e\ y) \Downarrow_{\mathsf{E}} \Theta; w}\ \textsc{App}_{\Downarrow}$$

$$\frac{\Gamma[\{y_i \mapsto \widehat{e_i}\}]; L; \widehat{e'} \Downarrow_{\mathsf{E}} \Delta; w}{\Gamma; L; \mathsf{let}\ \{x_i = e_i\}\ \mathsf{in}\ e' \Downarrow_{\mathsf{E}} \Delta; w}\ \textsc{Let}_{\Downarrow}$$

where y_i are fresh

$$\widehat{e_i} = e_i[y_1/x_1, \ldots, y_n/x_n],$$
$$\widehat{e'} = e'[y_1/x_1, \ldots, y_n/x_n]$$

Figure 3: Expression evaluation

In the $\Downarrow_{\mathsf{M}}$ judgments the argument stack A is a sequence of variables representing the pending arguments to be applied to the matching expression.

In both judgments the set L keeps track of variables under evaluation and is used to ensure freshness of variables in the $\textsc{Let}_{\Downarrow}$ rule [13].

Definition 3.1 (Freshness condition). A variable y is fresh if it does not occur (free or bound) in L, Γ or $\mathsf{let}\ \{x_i = e_i\}$ in e.

Remarks about rules for expressions (Figure 3). Rule $\textsc{Whnf}_{\Downarrow}$ terminates evaluation immediately when we reach a whnf, namely a non-saturated matching abstraction or a constructor.

Rule $\textsc{Sat}_{\Downarrow}$ applies to saturated matching abstractions; if the matching evaluation succeeds then we proceed to evaluate the expression returned.

Rule $\textsc{Var}_{\Downarrow}$ forces evaluation of an expression in the heap; as in Launchbury and Sestof's semantics, we remove the entry from the heap while performing the evaluation ("black-holing") and update the heap with the result afterwards.

Rule $\textsc{App}_{\Downarrow}$ first evaluates the function to obtain a matching abstraction and then evaluates the argument application.

Rule $\textsc{Let}_{\Downarrow}$ is identical to the one by Sestof: it allocates new expressions in the heap (taking care of renaming) and continues evaluating the body of the let expression.

Remarks about rules for matchings (Figure 4). Rule $\textsc{Return}_{\Downarrow}$ terminates evaluation successfully when we reach a return expression. The notation $A \rhd e$ represents the nested applications of left over

$$\boxed{\Gamma; L; A; m \Downarrow_{\mathsf{M}} \Delta; \mu} \qquad \text{matching evaluation}$$

$$\frac{}{\Gamma; L; A; \lceil e \rceil \Downarrow_{\mathsf{M}} \Gamma; \lceil A \rhd e \rceil}\ \textsc{Return}_{\Downarrow}$$

$$\frac{}{\Gamma; L; A; \lightning \Downarrow_{\mathsf{M}} \Gamma; \lightning}\ \textsc{Fail}_{\Downarrow}$$

$$\frac{\Gamma; L; (y : A); m \Downarrow_{\mathsf{M}} \Delta; \mu}{\Gamma; L; A; y \rhd m \Downarrow_{\mathsf{M}} \Delta; \mu}\ \textsc{Arg}_{\Downarrow}$$

$$\frac{\Gamma; L; A; m[y/x] \Downarrow_{\mathsf{M}} \Delta; \mu}{\Gamma; L; (y : A); x \Rightarrow m \Downarrow_{\mathsf{M}} \Delta; \mu}\ \textsc{Bind}_{\Downarrow}$$

$$\frac{\begin{array}{l}\Gamma; L; y \Downarrow_{\mathsf{E}} \Delta; c(y_1, \ldots, y_n) \\ \Delta; L; A; y_1 \rhd p_1 \Rightarrow \ldots \Rightarrow y_n \rhd p_n \Rightarrow m \Downarrow_{\mathsf{M}} \Theta; \mu\end{array}}{\Gamma; L; (y : A); c(p_1, \ldots, p_n) \Rightarrow m \Downarrow_{\mathsf{M}} \Theta; \mu}\ \textsc{Cons1}_{\Downarrow}$$

$$\frac{\Gamma; L; y \Downarrow_{\mathsf{E}} \Delta; c'(y_1, \ldots, y_k) \qquad c \neq c' \vee n \neq k}{\Gamma; L; (y : A); c(p_1, \ldots, p_n) \Rightarrow m \Downarrow_{\mathsf{M}} \Delta; \lightning}\ \textsc{Cons2}_{\Downarrow}$$

$$\frac{\Gamma; L; A; m_1 \Downarrow_{\mathsf{M}} \Delta; \lceil e \rceil}{\Gamma; L; A; (m_1 \mid m_2) \Downarrow_{\mathsf{M}} \Delta; \lceil e \rceil}\ \textsc{Alt1}_{\Downarrow}$$

$$\frac{\Gamma; L; A; m_1 \Downarrow_{\mathsf{M}} \Delta; \lightning \qquad \Delta; L; A; m_2 \Downarrow_{\mathsf{M}} \Theta; \mu}{\Gamma; L; A; (m_1 \mid m_2) \Downarrow_{\mathsf{M}} \Theta; \mu}\ \textsc{Alt2}_{\Downarrow}$$

Figure 4: Basic matching evaluation

arguments on the stack A to the expression e. The definition is:

$$[\,] \rhd e = e$$
$$(y : ys) \rhd e = ys \rhd (e\ y)$$

Rule $\textsc{Fail}_{\Downarrow}$ terminates evaluation unsuccessfully when we reach the match failure $\lightning$.

Rule $\textsc{Arg}_{\Downarrow}$ pushes an argument onto the stack and carries on evaluation.

Rule $\textsc{Bind}_{\Downarrow}$ binds a variable pattern to an argument on the stack. This simply a renaming of the pattern variable x to the heap variable y.

Rules $\textsc{Cons1}_{\Downarrow}$ and $\textsc{Cons2}_{\Downarrow}$ handle the successful and unsuccessful matching of a constructor pattern; in the later case the matching evaluation returns $\lightning$. Note also that $\textsc{Cons1}_{\Downarrow}$ continues the matching of sub-patterns in left to right order.

Rules $\textsc{Alt1}_{\Downarrow}$ and $\textsc{Alt2}_{\Downarrow}$ handle progress and failure in alternative matchings. Note that in $\textsc{Alt2}_{\Downarrow}$ evaluation continues with the updated heap Δ because the effects of evaluation of the failed match are preserved when evaluating m_2. Note also that the argument stack is shared between m_1 and m_2 implementing rule ($\rhd$ |) of Section 2.2. This sharing justifies why we restrict argument supply to single variables: allowing arbitrary expressions as arguments could duplicate computations because the same expression could be evaluated in more than one alternative. By restricting to single variables we can ensure that results are shared.

3.4 Dealing with pattern guards and view patterns

The syntax of matchings in Figure 2 and evaluation rules of Figure 4 deal only with argument matches of the form $y \rhd m$. This is done to allow sharing required by lazy evaluation but begs the question of how to encode the boolean and pattern guards.

Consider again the *nodups* example of Section 2.3; at first sight, it may seem that we can use let to introduce a name for the boolean guard:

$$\begin{array}{l} nodups = \lambda\ ((x : xs) \Rightarrow xs \rhd (y : xs') \Rightarrow \\ \qquad\qquad \lceil \mathsf{let}\ b = (x == y)\ \mathsf{in}\ \lambda(b \rhd \mathsf{True} \Rightarrow \lceil nodups\ xs \rceil) \rceil \\ \qquad \mid \ldots) \end{array}$$

However, this does not work: if b evaluates to False, then $\lambda(b \rhd \mathsf{True} \Rightarrow \ldots)$ gets "stuck" as $\lambda\lightning$ and the subsequent equations are never attempted.

One solution would be to join the two alternatives for the boolean guard:

$$\begin{array}{l} nodups = \lambda\ ((x : xs) \Rightarrow xs \rhd (y : xs') \Rightarrow \\ \qquad\qquad \lceil \mathsf{let}\ b = (x == y)\ \mathsf{in}\ \lambda(b \rhd (\mathsf{True} \Rightarrow \lceil nodups\ xs \rceil \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{False} \Rightarrow \lceil x : nodups\ xs \rceil)) \rceil \\ \qquad \mid [\,] \Rightarrow \lceil [\,] \rceil) \end{array}$$

However, this encoding would not work if the equations had pattern guards with different expressions.

A better solution is to allow pattern guards explicitly in the normalized syntax for λPMC:

$$\begin{array}{llll} m & ::= & \ldots & \text{(as in Figure 2)} \\ & \mid & e \rhd c(\vec{p}) \Rightarrow m & \text{pattern guards} \end{array}$$

Now *nodups* can be written exactly as in Section 2.3:

$$\begin{array}{l} nodups = \lambda\ ((x : xs) \Rightarrow xs \rhd (y : xs') \Rightarrow \\ \qquad\qquad\qquad (x == y) \rhd \mathsf{True} \Rightarrow \lceil nodups\ xs \rceil \\ \qquad \mid (x : xs) \Rightarrow \lceil x : nodups\ xs \rceil \\ \qquad \mid [\,] \Rightarrow \lceil [\,] \rceil) \end{array}$$

The extra evaluation rules need to deal with pattern guards are analogous to $\textsc{Cons1}_{\Downarrow}$ and $\textsc{Cons2}_{\Downarrow}$:

$$\frac{\begin{array}{l}\Gamma; L; e \Downarrow_{\mathsf{E}} \Delta; c(y_1, \ldots, y_n) \\ \Delta; L; A; y_1 \rhd p_1 \Rightarrow \ldots \Rightarrow y_n \rhd p_n \Rightarrow m \Downarrow_{\mathsf{M}} \Theta; \mu\end{array}}{\Gamma; L; A; e \rhd c(p_1, \ldots, p_n) \Rightarrow m \Downarrow_{\mathsf{M}} \Theta; \mu}\ \textsc{Guard1}_{\Downarrow}$$

$$\frac{\Gamma; L; e \Downarrow_{\mathsf{E}} \Delta; c'(y_1, \ldots, y_k) \qquad c \neq c' \vee n \neq k}{\Gamma; L; A; e \rhd c(p_1, \ldots, p_n) \Rightarrow m \Downarrow_{\mathsf{M}} \Delta; \lightning}\ \textsc{Guard2}_{\Downarrow}$$

Note that when e is a single variable then these rules produce the same effect as the combination of $\textsc{Arg}_{\Downarrow}$ with $\textsc{Cons1}_{\Downarrow}$ or $\textsc{Cons2}_{\Downarrow}$.

As seen in Section 2.3, we can use pattern guards for encoding view patterns. Recall the *length* function over join lists:

$$\begin{array}{l} length = \lambda\ (t \Rightarrow view\ t \rhd \mathsf{Nil} \Rightarrow \lceil 0 \rceil \\ \qquad\qquad \mid t \Rightarrow view\ t \rhd \mathsf{Cons}(h, xs) \Rightarrow \lceil 1 + length\ xs \rceil) \end{array}$$

One issue with this encoding is that evaluation of the view function is not shared. By contrast, GHC will try to optimize calls to view functions in such cases [8]. Such as optimization can be expressed in λPMC by using a let to explicitly share the result. However, we lose the one-to-one relation with the source program:

$$\begin{aligned} length' = \lambda(t \Rightarrow & \lceil \text{let } v = view\ t \\ & \text{in } \lambda(v \triangleright (\mathsf{Nil} \Rightarrow \lceil 0 \rceil \\ & \qquad \mid \mathsf{Cons}(h, xs) \Rightarrow \lceil 1 + length'\ xs \rceil) \rceil)) \end{aligned}$$

4 ABSTRACT MACHINE

We will now transform the big-step semantics of Section 3 into a small-step semantics for an abstract machine, i.e. a transition function between configurations where each transition performs only bounded amount of work. The presentation follows the derivation done by Sestof [13].

4.1 Configurations

Because our big-step semantics has two mutually recursive judgments, we introduce a *control* component that keeps track of the current evaluation mode.

C			control
C	$::=$	$\mathsf{E}\ e$	evaluate expression
	$\mid$	$\mathsf{M}\ A\ m$	evaluate matching with arguments A

The next step is to make evaluation order explicit in a *stack*. A stack is a list of *continuations* κ:

κ			continuations
κ	$::=$	y	push argument
	$\mid$	$!y$	push update
	$\mid$	$\$$	end matching
	$\mid$	$?(A, m)$	push alternative
	$\mid$	$@(A, c(\vec{p}) \Rightarrow m)$	push pattern

A machine configuration is a triple (Γ, C, S) of heap, control, and return stack. The initial configuration for evaluating e is $(\{\}, \mathsf{E}\ e, [\,])$. Evaluation may terminate successfully in a configuration $(\Gamma, \mathsf{E}\ w, [\,])$, get "stuck" in a configuration $(\Gamma, \mathsf{M}\ A\ \lightning, \$: S)$ due to pattern matching failure, or diverge (i.e. non-termination).

4.2 Transitions

The transitions between configurations are given by rules in Figure 5.

Rules App1$_\Rightarrow$, Var$_\Rightarrow$, Update$_\Rightarrow$ and Let$_\Rightarrow$ are identical to the ones in the first version of Sestof's abstract machine [13].

Rule App2$_\Rightarrow$ and Sat$_\Rightarrow$ handle application and matching evaluations, respectively. Note that the side conditions on $\#m$ ensure at most one rule applies. As in the big-step semantics, rule Sat$_\Rightarrow$ switches from evaluating an expression to a matching, pushing a mark '\$' onto the return stack to allow checking when no pending alternatives are available (rule Return1B$_\Rightarrow$).

Rule Cons1$_\Rightarrow$ switches from evaluating a matching to an expression in order to perform a pattern match, pushing a continuation onto the return stack. Rule Cons2$_\Rightarrow$ and Fail$_\Rightarrow$ handle the successful and unsuccessful pattern match. Rules Alt1$_\Rightarrow$ and Alt2$_\Rightarrow$ deal with alternatives. Finally, rule Arg$_\Rightarrow$ push arguments on the local argument stack.

4.3 Examples

We now present some example programs in λPMC that illustrate the execution of the abstract machine. Both examples share an initial heap Γ with bindings for some constructors and the *tail* function:

$$\Gamma = [nil \mapsto \texttt{[]},\ u \mapsto \mathsf{Unit},\ tail \mapsto \lambda(x : xs) \Rightarrow \lceil xs \rceil]$$

Example 1. For a first example consider *isShort* function of Section 1 applied to a thunk that will return a list with a single (unit) element.

$$\begin{aligned} & \text{let } xs = \lambda(x \Rightarrow \lceil (x : nil) \rceil)\ u \\ & \text{in } \lambda((x : y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid ys \Rightarrow \lceil \mathsf{True} \rceil)\ xs \end{aligned} \tag{1}$$

Figure 6 shows the sequence of machine configurations starting from the initial heap Γ and an empty stack. Note that the thunk l_1 associated with the expression bound to xs has been evaluated as a side effect of pattern matching.

Example 2. For the second example consider the following Haskell function with two list arguments (which is a simplification of the *zipWith* function of Section 2.3):

```
f (x:xs) (y:ys) = False
f xs ys         = True
```

Like *zip* and *zipWith*, f is strict on the first list argument but not the second: if the first list is `[]` then the second list will not be evaluated. This means that an expression such as *zip xs (tail xs)* is well-defined even when *xs* is the empty list. We will see that the evaluation of the translation into λPMC preserves this strictness property.

The translation of `f [] (tail [])` into λPMC is:

$$\begin{aligned} & \text{let } zs = tail\ nil \\ & \text{in } (\lambda(x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)\ nil\ zs \end{aligned} \tag{2}$$

Figure 7 shows the execution of (2) form the initial heap Γ. Note that thunk l_1 for the second argument is unevaluated and evaluation succeeds with result True.

If we were to reverse the order of arguments, i.e. attempt to evaluate

$$(\lambda(x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)\ zs\ nil$$

then the evaluation would get stuck on *tail nil*.

5 SOUNDNESS

We are ready to state and prove the correspondence between the big-step semantics of Section 3 and the small-step semantics of Section 4.

The first result states that each big-step evaluation corresponds to a sequence of small-step transitions in the machine. Because the evaluation of expressions and matching are mutually recursive we must prove the result for both evaluations simultaneously. We use $\Rightarrow^*$ for the reflexive and transitive closure of the transition relation.

Theorem 5.1. *If*

$$\Gamma; L; e \Downarrow_E \Delta; w$$

then for all S

$$(\Gamma, E\ e, S) \Rightarrow^* (\Delta, E\ w, S)$$

	Heap	Control	RetStack	rule
	Γ	E $(e\ y)$	S	App1$_{\Rightarrow}$
$\Longrightarrow$	Γ	E e	$y : S$	
	Γ	E λm	$y : S$	App2$_{\Rightarrow}$
$\Longrightarrow$	Γ	E $\lambda(y \rhd m)$	S	
if $\#m > 0$				
	Γ	E λm	S	Sat$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $[\,]\ m$	$\$: S$	
if $\#m = 0$				
	$\Gamma[y \mapsto e]$	E y	S	Var$_{\Rightarrow}$
$\Longrightarrow$	Γ	E e	$!y : S$	
	Γ	E w	$!y : S$	Update$_{\Rightarrow}$
$\Longrightarrow$	$\Gamma[y \mapsto w]$	E w	S	
	Γ	E let $\{x_i = e_i\}$ in e'	S	Let$_{\Rightarrow}$
$\Longrightarrow$	$\Gamma[y_i \mapsto \widehat{e_i}]$	E $e'[y_1/x_1, \ldots, y_n/x_n]$	S	
where y_i are fresh and $\widehat{e_i} = e_i[y_1/x_1, \ldots, y_n/x_n]$				
	Γ	M $A\ \lceil e \rceil$	S	Return1A$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $[\,]\ \lceil A \rhd e \rceil$	S	
if $A \neq [\,]$				
	Γ	M $[\,]\ \lceil e \rceil$	$\$: S$	Return1B$_{\Rightarrow}$
$\Longrightarrow$	Γ	E e	S	
	Γ	M A ↯	S	Return1C$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $[\,]$ ↯	S	
if $A \neq [\,]$				
	Γ	M $[\,]\ \lceil e \rceil$	$(?(A', m)) : S$	Return2$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $[\,]\ \lceil e \rceil$	S	
	Γ	M $(y : A)\ (x \Rightarrow m)$	S	Bind$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $A\ m[y/x]$	S	
	Γ	M $(y : A)\ (c(\vec{p}) \Rightarrow m)$	S	Cons1$_{\Rightarrow}$
$\Longrightarrow$	Γ	E y	$@(A, c(\vec{p}) \Rightarrow m) : S$	
	Γ	E $c(y_1, \ldots, y_n)$	$@(A, c(\vec{p}) \Rightarrow m) : S$	Cons2$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $A\ (y_1 \rhd p_1 \Rightarrow \ldots y_n \rhd p_n \Rightarrow m)$	S	
if $\lvert\vec{p}\rvert = n$				
	Γ	E $c'(y_1, \ldots, y_n)$	$@(A, c(\vec{p}) \Rightarrow m) : S$	Fail$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $[\,]$ ↯	S	
if $c \neq c' \vee \lvert\vec{p}\rvert \neq n$				
	Γ	M $A\ (e \rhd c(\vec{p}) \Rightarrow m)$	S	Guard$_{\Rightarrow}$
$\Longrightarrow$	Γ	E e	$@(A, c(\vec{p}) \Rightarrow m) : S$	
	Γ	M $A\ (y \rhd m)$	S	Arg$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $(y : A)\ m$	S	
	Γ	M $A\ (m_1 \mid m_2)$	S	Alt1$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $A\ m_1$	$?(A, m_2) : S$	
	Γ	M $[\,]$ ↯	$?(A, m) : S$	Alt2$_{\Rightarrow}$
$\Longrightarrow$	Γ	M $A\ m$	S	

Figure 5: Abstract machine transition rules

Heap	Control	Top of Stack	Rule
Γ	E let $xs = \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u$ in $\lambda(\ldots)\ xs$	—	$(\textsc{Let}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{E}\ \lambda((x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil \mid ys \Rightarrow \lceil\mathsf{True}\rceil)\ l_1$	—	$(\textsc{App1}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{E}\ \lambda((x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil \mid ys \Rightarrow \lceil\mathsf{True}\rceil)$	l_1	$(\textsc{App2}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{E}\ \lambda(l_1 \rhd ((x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil \mid ys \Rightarrow \lceil\mathsf{True}\rceil))$	—	$(\textsc{Sat}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{M}\ [\,]\ l_1 \rhd ((x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil \mid ys \Rightarrow \lceil\mathsf{True}\rceil)$	$\$$	$(\textsc{Arg}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{M}\ [l_1]\ (x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil \mid ys \Rightarrow \lceil\mathsf{True}\rceil$	$\$$	$(\textsc{Alt1}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{M}\ [l_1]\ (x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Cons1}_{\Rightarrow})$
$\Gamma[l_1 \mapsto \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u]$	$\mathsf{E}\ l_1$	$@([\,], (x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil)$	$(\textsc{Var}_{\Rightarrow})$
Γ	$\mathsf{E}\ \lambda(x \Rightarrow \lceil(x : nil)\rceil)\ u$	$!l_1$	$(\textsc{App1}_{\Rightarrow})$
Γ	$\mathsf{E}\ \lambda(x \Rightarrow \lceil(x : nil)\rceil)$	u	$(\textsc{App2}_{\Rightarrow})$
Γ	$\mathsf{E}\ \lambda(u \rhd x \Rightarrow \lceil(x : nil)\rceil)$	$!l_1$	$(\textsc{Sat}_{\Rightarrow})$
Γ	$\mathsf{M}\ [\,]\ u \rhd x \Rightarrow \lceil(x : nil)\rceil$	$\$$	$(\textsc{Arg}_{\Rightarrow})$
Γ	$\mathsf{M}\ [u]\ x \Rightarrow \lceil(x : nil)\rceil$	$\$$	$(\textsc{Bind}_{\Rightarrow})$
Γ	$\mathsf{M}\ [\,]\ \lceil(u : nil)\rceil$	$\$$	$(\textsc{Return1B}_{\Rightarrow})$
Γ	$\mathsf{E}\ (u : nil)$	$!l_1$	$(\textsc{Update}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{E}\ (u : nil)$	$@([\,], (x : y : ys) \Rightarrow \lceil\mathsf{False}\rceil)$	$(\textsc{Cons2}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [\,]\ u \rhd x \Rightarrow nil \rhd (y : ys) \Rightarrow \lceil\mathsf{False}\rceil$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Arg}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [u]\ x \Rightarrow nil \rhd (y : ys) \Rightarrow \lceil\mathsf{False}\rceil$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Bind}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [\,]\ nil \rhd (y : ys) \Rightarrow \lceil\mathsf{False}\rceil$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Arg}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [nil]\ (y : ys) \Rightarrow \lceil\mathsf{False}\rceil$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Cons1}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{E}\ nil$	$@([\,], (y : ys) \Rightarrow \lceil\mathsf{False}\rceil)$	$(\textsc{Var}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{E}\ []$	$!nil$	$(\textsc{Update}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{E}\ []$	$@([\,], (y : ys) \Rightarrow \lceil\mathsf{False}\rceil)$	$(\textsc{Fail}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [\,]\ \lightning$	$?([l_1], ys \Rightarrow \lceil\mathsf{True}\rceil)$	$(\textsc{Alt2}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [l_1]\ ys \Rightarrow \lceil\mathsf{True}\rceil$	$\$$	$(\textsc{Bind}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{M}\ [\,]\ \lceil\mathsf{True}\rceil$	$\$$	$(\textsc{Return1B}_{\Rightarrow})$
$\Gamma[l_1 \mapsto (u : nil)]$	$\mathsf{E}\ \mathsf{True}$	—	

Figure 6: Execution of Example 1

If

$$\Gamma; L; A; m \Downarrow_M \Delta; \mu$$

then for all S

$$(\Gamma, M\ A\ m, S) \Rightarrow^* (\Delta, M\ [\,]\ \mu, S)$$

Proof. The proof is by induction on the height of the evaluation derivations of $\Downarrow_\mathsf{E}$ and $\Downarrow_\mathsf{M}$. We proceed by analysis on the last rule used.

Case $\textsc{App}_{\Downarrow}$. The evaluation rule is

$$\frac{\Gamma; L; e \Downarrow_\mathsf{E} \Delta; \lambda m \qquad \Delta; L; \lambda(y \rhd m) \Downarrow_\mathsf{E} \Theta; w}{\Gamma; L; (e\ y) \Downarrow_\mathsf{E} \Theta; w}$$

Let S be arbitrary and $S' = y : S$; applying the induction hypothesis we get

$$(\Gamma, \mathsf{E}\ e, S') \Rightarrow^* (\Delta, \mathsf{E}\ (\lambda m), S')$$
$$(\Delta, \mathsf{E}\ \lambda(y \rhd m), S) \Rightarrow^* (\Theta, \mathsf{E}\ e, S)$$

Because λm is in whnf we know that $\# m > 0$. We obtain the proof obligation as follows:

$$(\Gamma, \mathsf{E}\ (e\ y), S) \overset{\textsc{App1}}{\Rightarrow} (\Gamma, \mathsf{E}\ e, y : S) \Rightarrow^* (\Delta, \mathsf{E}\ (\lambda m), y : S)$$
$$\overset{\textsc{App2}}{\Rightarrow} (\Delta, \mathsf{E}\ \lambda(y \rhd m), S) \Rightarrow^* (\Theta, \mathsf{E}\ e, S)$$

Case $\textsc{Sat}_{\Downarrow}$. The evaluation rule is

$$\frac{\# m = 0 \qquad \Gamma; L; [\,];\ m \Downarrow_\mathsf{M} \Delta; \lceil e\rceil \qquad \Delta; L; e \Downarrow_\mathsf{E} \Theta;\ w}{\Gamma; L; \lambda m \Downarrow_\mathsf{E} \Theta; w}$$

Let S be arbitrary and $S' = \$: S$; the induction hypotheses give

$$(\Gamma, \mathsf{M}\ [\,]\ m, S') \Rightarrow^* (\Delta, \mathsf{M}\ [\,]\ \lceil e\rceil, S')$$
$$(\Delta, \mathsf{E}\ e, S) \Rightarrow^* (\Theta, \mathsf{E}\ w, S)$$

We obtain the proof obligation as

$$(\Gamma, \mathsf{E}\ \lambda m, S) \overset{\textsc{Sat}}{\Rightarrow} (\Gamma, \mathsf{M}\ [\,]\ m, \$: S) \Rightarrow^* (\Delta, \mathsf{M}\ [\,]\ \lceil e\rceil, \$: S)$$
$$\overset{\textsc{Return1B}}{\Rightarrow} (\Delta, \mathsf{E}\ e, S) \Rightarrow^* (\Theta, \mathsf{E}\ w, S)$$

Case $\textsc{Var}_{\Downarrow}$. The evaluation rule is

$$\frac{\Gamma; L \cup \{y\}; e \Downarrow_\mathsf{E} \Delta; w}{\Gamma[y \mapsto e]; L; y \Downarrow_\mathsf{E} \Delta[y \mapsto w]; w}$$

Let S be arbitrary and $S' = y : S$. Applying the induction hypothesis to the premise gives

$$(\Gamma, \mathsf{E}\ e, S') \Rightarrow^* (\Delta, \mathsf{E}\ w, S')$$

Heap	Control	Top of Stack	Rule
Γ	E let $zs = \mathit{tail}\ \mathit{nil}$ in $\lambda(\ldots)\ \mathit{nil}\ \mathit{xs}$	—	($\textsc{Let}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \lambda(\ldots)\ \mathit{nil}\ l1$	—	($\textsc{App1}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \lambda(\ldots)\ \mathit{nil}$	$l1$	($\textsc{App1}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \lambda\ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	nil	($\textsc{App2}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \lambda(\mathit{nil} \rhd\ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil\ )\ \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	$l1$	($\textsc{App2}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \lambda(l1 \rhd \mathit{nil} \rhd\ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil\ )\ \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	—	($\textsc{Sat}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\,]\ l1 \rhd \mathit{nil} \rhd\ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	\$	($\textsc{Arg}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [l1]\ \mathit{nil} \rhd\ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	\$	($\textsc{Arg}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\mathit{nil},\ l1]\ \ ((x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil \mid xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	\$	($\textsc{Alt1}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\mathit{nil},\ l1]\ (x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil$	$?([\mathit{nil},\ l1], xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	($\textsc{Cons1}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ \mathit{nil}$	$@([l1], (x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil)$	($\textsc{Var}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ [\,]$	$!\mathit{nil}$	($\textsc{Update}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{E}\ [\,]$	$@([l1], (x : xs) \Rightarrow (y : ys) \Rightarrow \lceil \mathsf{False} \rceil)$	($\textsc{Fail}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\,]\ \lightning$	$?([\mathit{nil},\ l1], xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil)$	($\textsc{Alt2}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\mathit{nil},\ l1]\ xs \Rightarrow ys \Rightarrow \lceil \mathsf{True} \rceil$	\$	($\textsc{Bind}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [l1]\ ys \Rightarrow \lceil \mathsf{True} \rceil$	\$	($\textsc{Bind}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	$\mathsf{M}\ [\,]\ \lceil \mathsf{True} \rceil$	\$	($\textsc{Return1B}_{\Rightarrow}$)
$\Gamma[l1 \mapsto \mathit{tail}\ \mathit{nil}]$	E True	—	

Figure 7: Execution of Example 2

We obtain the proof obligation as

$$(\Gamma[y \mapsto e], \mathsf{E}\ y, S) \overset{\textsc{Var}}{\Rightarrow} (\Gamma, \mathsf{E}\ e, S') \Rightarrow^{*} (\Delta, \mathsf{E}\ w, S')$$
$$\overset{\textsc{Update}}{\Rightarrow} (\Delta[y \mapsto w], \mathsf{E}\ w, S)$$

Case $\textsc{Return}_{\Downarrow}$. The evaluation rule is

$$\frac{}{\Gamma; L; A; \lceil e \rceil \Downarrow_{\mathsf{M}} \Gamma; \lceil A \rhd e \rceil}$$

The proof obligation follows by an application of $\textsc{Return1A}_{\Rightarrow}$:

$$(\Gamma, \mathsf{M}\ A\ \ \lceil e \rceil, S) \overset{\textsc{Return1A}}{\Rightarrow} (\Gamma, \mathsf{M}\ [\,]\ \ \lceil A \rhd e \rceil, S)$$

Case $\textsc{Fail}_{\Downarrow}$. The evaluation rule is

$$\frac{}{\Gamma; L; A; \lightning \Downarrow_{\mathsf{M}} \Gamma; \lightning}$$

The proof obligation follows trivially by either the empty sequence (if $A = [\,]$) or an application of $\textsc{Return1C}_{\Rightarrow}$.

Case $\textsc{Arg}_{\Downarrow}$. The evaluation rule is

$$\frac{\Gamma; L; (y : A); m \Downarrow_{\mathsf{M}} \Delta; \mu}{\Gamma; L; A; y \rhd m \Downarrow_{\mathsf{M}} \Delta; \mu}$$

The induction hypothesis gives

$$(\Gamma, \mathsf{M}\ (y : A)\ m, S) \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \mu, S)$$

Using $\textsc{Arg}_{\Rightarrow}$ gives the required proof obligation:

$$(\Gamma, \mathsf{M}\ A\ (y \rhd m), S) \overset{\textsc{Arg}}{\Rightarrow} (\Gamma, \mathsf{M}\ (y : A)\ m, S) \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \mu, S)$$

Case $\textsc{Alt1}_{\Downarrow}$. The evaluation rule is

$$\frac{\Gamma; L; A; m_1 \Downarrow_{\mathsf{M}} \Delta; \lceil e \rceil}{\Gamma; L; A; (m_1 \mid m_2) \Downarrow_{\mathsf{M}} \Delta; \lceil e \rceil}$$

Let S be arbitrary and $S' = (?(A, m_2)) : S$. Applying the induction hypothesis gives

$$(\Gamma, \mathsf{M}\ A\ m_1, S') \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \ \lceil e \rceil, S')$$

The proof obligation is

$$(\Gamma, \mathsf{M}\ A\ (m_1 \mid m_2), S) \overset{\textsc{Alt1}}{\Rightarrow} (\Gamma, \mathsf{M}\ A\ m_1, S') \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \ \lceil e \rceil, S')$$
$$\overset{\textsc{Return2}}{\Rightarrow} (\Delta, \mathsf{M}\ [\,]\ \ \lceil e \rceil, S)$$

Case $\textsc{Alt2}_{\Downarrow}$. The evaluation rule is

$$\frac{\Gamma; L; A; m_1 \Downarrow_{\mathsf{M}} \Delta; \lightning \qquad \Delta; L; A; m_2 \Downarrow_{\mathsf{M}} \Theta; \mu}{\Gamma; L; A; (m_1 \mid m_2) \Downarrow_{\mathsf{M}} \Theta; \mu}$$

Let S be arbitrary and $S' = (?(A, m_2)) : S$. The induction hypothesis gives

$$(\Gamma, \mathsf{M}\ A\ m_1, S') \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \lightning, S')$$
$$(\Delta, \mathsf{M}\ A\ m_2, S) \Rightarrow^{*} (\Theta, \mathsf{M}\ [\,]\ \mu, S)$$

We can obtain the proof obligation as follows:

$$(\Gamma, \mathsf{M}\ A\ (m_1 \mid m_2), S) \overset{\textsc{Alt1}}{\Rightarrow} (\Gamma, \mathsf{M}\ A\ m_1, S') \Rightarrow^{*} (\Delta, \mathsf{M}\ [\,]\ \lightning, S')$$
$$\overset{\textsc{Alt2}}{\Rightarrow} (\Delta, \mathsf{M}\ A\ m_2, S) \Rightarrow^{*} (\Theta, \mathsf{M}\ [\,]\ \mu, S)$$

□

The next result establishes that the small-step semantics derives only evaluations corresponding to the big-step semantics, provided we restrict ourselves to balanced evaluations.

Definition 5.2. A sequence of evaluation steps $(\Gamma, C, S) \Rightarrow^* (\Delta, C', S)$ is *balanced* if the initial and final stacks are identical and every intermediate stack is of the form $S' = \kappa_1 : \kappa_2 : \ldots : \kappa_n : S$, i.e. an extension of the initial stack.

Definition 5.3. The *trace* of a sequence of small-step evaluation steps $(\Gamma, C, S) \Rightarrow^* (\Delta, C', S')$ is the sequence of rules used in the evaluation.

Consider now the traces B of balanced expression evaluations $(\Gamma, \mathsf{E}\ e, S) \Rightarrow^* (\Delta, \mathsf{E}\ w, S)$ and B' for balanced matching evaluations $(\Gamma, \mathsf{M}\ A\ m, S) \Rightarrow^* (\Delta, \mathsf{M}\ [\,]\ \mu, S)$. Since specific rules introduce and eliminate stack continuations, we can see by inspection of Figure 5 that such traces must be generated by the following grammars:

$$\begin{aligned} B &::= \textsc{App1}\ B\ \textsc{App2}\ B \mid \textsc{Sat}\ B'\ \textsc{Return1B}\ B \mid \textsc{Var}\ B\ \textsc{Update} \\ &\quad \mid \textsc{Let}\ B \mid \varepsilon \end{aligned} \tag{3}$$

$$\begin{aligned} B' &::= \textsc{Alt1}\ B'\ \textsc{Alt2}\ B' \mid \textsc{Alt1}\ B'\ \textsc{Return2} \\ &\quad \mid \textsc{Cons1}\ B\ \textsc{Cons2}\ B' \mid \textsc{Cons1}\ B\ \textsc{Fail} \\ &\quad \mid \textsc{Guard}\ B\ \textsc{Cons2}\ B' \mid \textsc{Guard}\ B\ \textsc{Fail} \\ &\quad \mid \textsc{Arg}\ B' \mid \textsc{Bind}\ B' \mid \textsc{Return1A}\ B' \mid \textsc{Return1C}\ B' \mid \varepsilon \end{aligned} \tag{4}$$

The last step before we can state the second soundness result is to see that we can recover the set L of locations under evaluation used in the big-step semantics from the return stack of the small-step semantics.

Definition 5.4. Let $!\,S$ be the set of locations marked for updates in a stack S, i.e. $!\,S = \{y \;:\; (!y) \in S\}$.

Theorem 5.5. *If* $(\Gamma, E\,e, S) \Rightarrow^* (\Delta, E\,w, S)$ *is a balanced expression evaluation then* $\Gamma; !\,S; e \Downarrow_E \Delta; w$.

If $(\Gamma, M\,A\ m, S) \Rightarrow^* (\Delta, M\,[\,]\ \mu, S)$ *is a balanced matching evaluation then* $\Gamma; !\,S; A; m \Downarrow_M \Delta; \mu$

Proof. The proof is by induction on the derivation of balanced evaluations following the grammar (3) and (4). Each production of the grammar corresponds to one big-step evaluation rule for $\Downarrow_\mathsf{E}$ or $\Downarrow_\mathsf{M}$.

The base case corresponds to empty balanced evaluations; the start configurations can only of three forms:

$(\Gamma, \mathbf{E}\ w, S)$**:** the proof obligation is given by rule $\textsc{Whnf}_\Downarrow$.
$(\Gamma, \mathbf{M}\ [\,]\ \lceil e \rceil, S)$**:** the proof obligation is given by rule $\textsc{Return}_\Downarrow$.
$(\Gamma, \mathbf{M}\ [\,]\ \lightning, S)$ the proof obligation is given by rule $\textsc{Fail}_\Downarrow$.

The remaining cases are non-empty balanced evaluations; we present some cases in detail.

Case App1 B App2 B. The balanced evaluation must be

$$\begin{aligned} (\Gamma, \mathsf{E}\ (e\ y), S) &\overset{\textsc{App1}}{\Rightarrow} (\Gamma, \mathsf{E}\ e, y : S) \Rightarrow^* (\Gamma', \mathsf{E}\ (\lambda m), y : S) \\ &\overset{\textsc{App2}}{\Rightarrow} (\Gamma', \mathsf{E}\ \lambda(y \triangleright m), S) \Rightarrow^* (\Delta, \mathsf{E}\ w, S) \end{aligned}$$

Applying the induction hypothesis to the evaluation sequences above we get

$$\Gamma; !\,S; e \Downarrow_\mathsf{E} \Gamma'; \lambda m$$
$$\Gamma'; !\,S; \lambda(y \triangleright m) \Downarrow_\mathsf{E} \Delta; w$$

Applying rule $\textsc{App}_\Downarrow$ yields the proof obligation $\Gamma; !\,S; (e\ y); \Downarrow_\mathsf{E} \Delta; w$.

Case Sat B' Return1B B. The balanced evaluation must be

$$\begin{aligned} (\Gamma, \mathsf{E}\ \lambda m, S) &\overset{\textsc{Sat}}{\Rightarrow} (\Gamma, \mathsf{M}\ [\,]\ m, \$: S) \Rightarrow^* (\Gamma', \mathsf{M}\ [\,]\ \lceil e \rceil, \$: S) \\ &\overset{\textsc{Return1B}}{\Rightarrow} (\Gamma', \mathsf{E}\ e, S) \Rightarrow^* (\Delta, \mathsf{E}\ w, S) \end{aligned}$$

Note that $!\,(\$: S) = !\,S$. Applying the induction hypothesis to the evaluation sequences above we get

$$\Gamma; !\,S; [\,]; m \Downarrow_\mathsf{M} \Gamma'; \lceil e \rceil$$
$$\Gamma'; !\,S; e \Downarrow_\mathsf{E} \Delta; w$$

Applying rule $\textsc{Sat}_\Downarrow$ yields the proof obligation.

Case Alt1 B' Alt2 B'. The balanced evaluation must be

$$\begin{aligned} (\Gamma, \mathsf{M}\ A\ (m_1 \mid m_2), S) &\overset{\textsc{Alt1}}{\Rightarrow} (\Gamma, \mathsf{M}\ A\ m_1, S') \Rightarrow^* (\Gamma' \mathsf{M}\ [\,]\ \lightning, S') \\ &\overset{\textsc{Alt2}}{\Rightarrow} (\Gamma', \mathsf{M}\ A\ m_2, S) \Rightarrow^* (\Delta, \mathsf{M}\ [\,]\ \mu, S) \end{aligned}$$

where $S' = @(A, m_2) : S$. Note that $!\,S' = !\,S$. Applying the induction hypothesis to the balanced evaluations above we get

$$\Gamma; !\,S; A; m_1 \Downarrow_\mathsf{M} \Gamma'; \lightning$$
$$\Gamma'; !\,S; A; m_2 \Downarrow_\mathsf{M} \Delta; \mu$$

Applying rule $\textsc{Alt2}_\Downarrow$ yields the proof obligation. □

6 RELATED WORK

Chapter 4 of the classic textbook by Peyton Jones [11] defines the semantics of pattern matching using lambda abstractions with patterns $\lambda p.E$ together with a FAIL expression and a "fatbar" operator. This semantics is denotational and serves primarily as the basis for defining the correctness of compilation into case expressions presented in the subsequent chapter.

Kahl defines a pattern matching calculus and proved its confluence [4]. This work forms the basis for our λPMC language and the two semantics. The principle differences are: (1) our operational semantics deal with lazy evaluation explicitly and (2) we do not consider the "empty expression" corresponding to a pattern matching failure $\lambda\lightning$; there is simply no evaluation in such cases. Our definitions of normal forms for expressions and matchings are also novel.

Klop et al. [5] re-visit the lambda calculus with patterns of Peyton Jones and situates it in the context of more general combinatory reduction systems (CRS's) and higher-order rewriting systems.

Several researchers have looked at restricted forms of pattern calculi. Bucciarelli et al. [2] have looked at characterizing observability for pattern matching restricted to pairs using a type system based on intersection types. Inspired by this work, Alves et al. [1] have defined a type system based on non-idempotent intersection types to characterize bounds on the length of normalization sequences for such pattern calculi.

Wadler [14] introduced *view patterns* that generalize pattern matching to abstract data types; as seen in Section 2.3, these are easily translated into λPMC. McBride and McKinna [10] have looked at pattern matching in a dependently-typed setting (where matching refines not just values but also types) and generalized the notion of views to allow for user-defined induction principles.

7 CONCLUSION AND FURTHER WORK

This paper presented λPMC, a small lazy functional language based a pattern matching calculus together with two operational semantics: a big-step semantics in the style of Launchbury and a small-step abstract machine in the style of the Sestof's machines. The principal contribution of λPMC over previous presentations in the literature is that the translation of a source language such as Haskell into λPMC enjoys a closer relation: each equation in the source language corresponds to one alternative in a matching abstraction. Furthermore, we have shown that λPMC can seamlessly handle extensions such as pattern guards and view patterns.

A number of directions for extending this work are possible:

Where declarations. We have not considered the translation of `where` declarations as in Haskell. At first glance, it may appear that these can be translated in a similar way to pattern guards. For example, consider the following Haskell function:

```
foo x | y > 0  = e1
      | y == 0 = e2
      where y = bar x
```

where `e1` and `e2` are some arbitrary expressions. This can indeed be translated into λPMC as follows:

$$foo = \lambda(x \Rightarrow (bar\ x \rhd (y \Rightarrow \begin{array}{l}(y > 0 \rhd \mathsf{True} \Rightarrow \lceil e_1 \rceil \\ \mid y == 0 \rhd \mathsf{True} \Rightarrow \lceil e_2 \rceil)))\end{array}$$

However, this would not work if the binding was recursive. Moreover, the translation does not preserve the static semantics because `where` bindings could be used polymorphically and pattern argument cannot. A better approach might be to extend matchings with explicit `where` bindings (analogous to `let` bindings for expressions).

Binding patterns and irrefutable patterns. We have also not considered patterns on the left-hand side of (let) bindings. Again, it may appear that we could re-use matching arguments to translate a pattern binding

```
let p = e1 in e2
```

as

$$\lambda(e_1 \rhd (p \Rightarrow e_2))$$

However, this translation would not work if the binding is recursive (i.e. the pattern variables of p occur in e_1). Also, it would again not preserve the static semantics.

Moreover, it would not preserve the dynamic semantics either: The Haskell semantics specifies that pattern bindings introduce *irrefutable patterns* [9], i.e. any matching should delayed until the pattern variables are used in e_2. One approach for deadling with this is to employ the translation of irrefutable patterns into simple patterns as defined in the Haskell report [9]. Alternatively, we could add irrefutable patterns explicitly to λPMC. We leave the exploration of these approaches as further work.

Relation to lower-level machines. Because efficiency was not our primary concern, we have presented a machine that operates over expressions directly and employs substitutions (cf. rules $\textsc{Let}_{\Rightarrow}$ and $\textsc{Bind}_{\Rightarrow}$). Nonetheless, it should be straightforward to modify this to use offsets into environments (i.e. de Bruijn indices) as in the subsequent derivations of abstract machines in [13].

It should also be possible to add *unboxed types* similarly to what is done in the GHC Core and STG languages [3]. Another direction would be to explore whether the transformation based optimizations of Peyton Jones and Santos [12] carry over to our setting. Given that λPMC is equipped with a relatively simple operational model, it would be interesting to explore if we could prove some transformations to be optimizations at a more abstract level than the *de facto* implementation (the GHC compiler).

REFERENCES

[1] Sandra Alves, Delia Kesner, and Daniel Ventura. A Quantitative Understanding of Pattern Matching. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:36, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-158-0. doi: 10.4230/LIPIcs.TYPES.2019.3. URL https://drops.dagstuhl.de/opus/volltexte/2020/13067.

[2] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Observability for Pair Pattern Calculi. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 123–137, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-87-3. doi: 10.4230/LIPIcs.TLCA.2015.123. URL http://drops.dagstuhl.de/opus/volltexte/2015/5159.

[3] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2): 127–202, 1992. doi: 10.1017/S0956796800000319.

[4] Wolfram Kahl. Basic pattern matching calculi: a fresh view on matching failure. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming*, pages 276–290, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24754-8.

[5] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1):16–31, 2008. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2008.01.019. URL https://www.sciencedirect.com/science/article/pii/S0304397508000571. Calculi, Types and Applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca.

[6] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 144–154, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915607. doi: 10.1145/158511.158618. URL https://doi.org/10.1145/158511.158618.

[7] GHC maintainers. Ghc user guide, 2023. URL https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/lambda_case.html.

[8] GHC maintainers, 2023. URL https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/view_patterns.html.

[9] Simon Marlow. Haskell 2010 language report, 2023. URL https://www.haskell.org/onlinereport/haskell2010/.

[10] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi: 10.1017/S0956796803004829.

[11] Simon Peyton-Jones. *The implementation of functional languages.* Prentice-Hall, 1987.

[12] Simon Peyton Jones and Andre Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1), October 1997. URL https://www.microsoft.com/en-us/research/publication/a-transformation-based-optimiser-for-haskell/.

[13] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7:231–264, 05 1997. doi: 10.1017/S0956796897002712.

[14] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 307–313, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912152. doi: 10.1145/41625.41653. URL https://doi.org/10.1145/41625.41653.