Basic Pattern Matching Calculi: a Fresh View on Matching Failure

Wolfram Kahl

Department of Computing and Software, McMaster University
http://www.cas.mcmaster.ca/~kahl/

Abstract. We propose pattern matching calculi as a refinement of λ-calculus that integrates mechanisms appropriate for fine-grained modelling of non-strict pattern matching.
Compared with the functional rewriting strategy usually employed to define the operational semantics of pattern matching in non-strict functional programming languages like Haskell or Clean, our pattern matching calculi achieve the same effects using simpler and more local rules.
The main device is to embed into expressions the separate syntactic category of matchings; the resulting language naturally encompasses pattern guards and Boolean guards as special cases.
By allowing a confluent reduction system and a normalising strategy, these pattern matching calculi provide a new basis for operational semantics of non-strict programming languages and also for implementations.

1 Introduction

The operational semantics of functional programming languages is usually explained via special kinds of λ-calculi and term rewriting systems (TRSs). One way to look at the relation between these two approaches is to consider λ-calculi as *internalisations* of term rewriting systems: λ-abstraction internalises *applicative* TRSs (where each function is defined in a single "equation" the left-hand side of which is an application of the function symbol to only variables), and fixedpoint combinators internalise recursive function definitions.

In addition to these two features, modern functional programming languages support function definitions based on *pattern matching*. A *pattern* is an expression built only from variables and *constructors* — in the context of applicative TRSs, constructors are function symbols that never occur as head of a rule. In the functional programming context, constructors are introduced by datatype definitions, for example the list constructors "`[]`" (empty list) and "`_ : _`" ("cons", non-empty list construction from head and tail).

In the term rewriting view, a function is *defined by pattern matching* if it is defined by a group of rules, each having as left-hand side an application of the defined function symbol to patterns. Definitions using pattern matching are "processed sequentially"; for an example assume the following definition:

```
isEmptyList (x : xs) = False
isEmptyList ys       = True
```

The second line is not an equation valid for arbitrary `ys`, but a rule that is only considered if the left-hand side of the first rule gives rise to a mismatch — here, the only value of `ys` for which this is the case is the empty list `[]`.

Function definitions with patterns as arguments on the left-hand sides are typical of modern functional programming languages. In non-strict languages like Haskell, Clean, or Miranda, the operational semantics of pattern matching is quite complex; usually it is formulated as the *functional rewriting strategy*, which is a rather involved priority rewriting strategy [19, sect. 4.7.1].

In the case of, for example, Haskell, the operational semantics of pattern matching is defined via the special instance of pattern matching in case expressions. For `isEmptyList`, the above definition is considered as shorthand for:

```
isEmptyList zs = case zs of (x : xs) -> False
                            ys       -> True
```

Case expressions can be seen as an internalisation of pattern matching that is not quite analogous to the internalisation of function abstraction in λ-calculus; the important difference is that, in comparison with λ-abstractions, case expressions contain not only the abstracted pattern matchings, but also an additional application to an argument. To further complicate matters, Boolean *guards* and, more recently, *pattern guards* interfere with the "straightforward" pattern matching.

In this paper we present a new calculus that cleanly internalises pattern matching by drawing a clearer distinction between the aspects involved. For that purpose, we essentially liberate the case expression from its rigidly built-in application, generalising the special syntactic category of case alternatives into the new syntactic category of *matchings* that incorporates all aspects of pattern matching, as opposed to the (preserved) syntactical category of *expressions* that now is mostly concerned with pattern construction and function application.

This allows straightforward internalisation of pattern matching definitions without having to introduce new variables like `zs` for the case variant:

$$\texttt{isEmptyList} = \{\!|\, (x : xs) \mapsto \mathsf{False} \,\|\, ys \mapsto \mathsf{True} \,|\!\}$$

In addition, using the pattern matching calculus as basis of functional programming has advantages both for expressivity and reasoning about programs.

With respect to reasoning, the full internalisation of pattern matching eliminates the problem of all priority systems that what is written down as an unconditional equation only applies to certain patterns "left over" from higher-priority equations defining the same function. The usual justification for allowing this non-orthogonality is that otherwise the number of equations would explode. Our matching language allows direct transliteration of such prioritised definitions without additional cost, and even includes the means to factor out more commonalities than is possible in priority rewriting systems. The syntactical features necessary to achieve this turn out to be sufficient to include both Boolean guards and pattern guards as special cases. This gives the language a boost in expressivity over systems directly based on term rewriting, and at the same time keeps the system simple and uniform.

A noteworthy result is that by providing two variants of a simple rule concerned with results of matching failure, we obtain two interesting systems, both confluent and equipped with the same normalising strategy:

– The first mirrors exactly the definition of pattern matching in, e.g., Haskell, which corresponds to the functional rewrite strategy modified by treating matching against non-covered alternatives as a run-time error. It is well known that the functional strategy, considered as a term rewriting strategy, is not normalising, so there are certain terms that, translated into our first system, have a normal form that corresponds to such run-time errors.
– The second system is a refinement of the first in that it preserves all terminating reductions not ending in a run-time errors, and also has such "successful" reductions for some terms that reduce to run-time errors in the first system.

Similar mechanisms have been proposed in the literature, see Sect. 8; we feel that the setting of the pattern matching calculus helps to clarify the issues involved and provides an attractive environment for describing and analysing such alternative treatments of matching failure.

After presenting the abstract syntax, we show how the pattern matching calculus encodes λ-calculus and Haskell pattern matching including Boolean and pattern guards. Sect. 4 presents the reduction rules, which are applied to selected examples in Sect. 5. In Sect. 6 we summarise the mechanised confluence proof, and Sect. 7 is devoted to the normalising reduction strategy. Sect. 8 discusses related work. Details omitted here for reasons of space can be found in [10].

2 Abstract Syntax

The pattern matching calculus, from now on usually abbreviated PMC, has two major syntactic categories, namely *expressions* and *matchings*. These are defined by mutual recursion. When considering the analogy to functional programs, only expressions of the pattern matching calculus correspond to expressions of functional programming languages. Matchings can be seen as a generalisation of groups of case alternatives. Operationally, matchings can be "waiting for argument supply", or they can be *saturated*; saturated matchings can *succeed* and then *return* an expression, or they can *fail*. *Patterns* form a separate syntactic category that will be used to construct pattern matchings.

We now present the abstract syntax of the pattern matching calculus with some intuitive explanation of the intended meaning of the constructs.

As base sets, we use Var as the set of *variables*, and Constr as the set of *constructors*. For the purpose of our examples, numbers are assumed to be elements of Constr and are used only in zero-ary constructions (which are written without parentheses). Constructors will, as usual, be used to build both patterns and expressions. Indeed, one might consider Pat as a subset of Expr.

The following summarises the abstract syntax of PMC:

Pat	$::= \mathsf{Var}$	variable
	$\mid \mathsf{Constr}(\mathsf{Pat}, \ldots, \mathsf{Pat})$	constructor pattern
Expr	$::= \mathsf{Var}$	variable
	$\mid \mathsf{Constr}(\mathsf{Expr}, \ldots, \mathsf{Expr})$	constructor application
	$\mid \mathsf{Expr}\ \mathsf{Expr}$	function application
	$\mid \{\!\mid \mathsf{Match} \mid\!\}$	matching abstraction
	$\mid \oslash$	empty expression
Match	$::= \rceil\mathsf{Expr}\lceil$	expression matching
	$\mid \lightning$	failure
	$\mid \mathsf{Pat} \Mapsto \mathsf{Match}$	pattern matching
	$\mid \mathsf{Expr} \rhd \mathsf{Match}$	argument supply
	$\mid \mathsf{Match} \,\|\, \mathsf{Match}$	alternative

Patterns are built from variables and constructor applications. All variables occurring in a pattern are *free* in that pattern; for every pattern $p : \mathsf{Pat}$, we denote its set of free variables by $\mathsf{FV}(p)$. In the following, we silently restrict *all* patterns to be *linear*, i.e., not to contain more than one occurrence of any variable.

Expressions are the syntactic category that embodies the term construction aspects; besides variables, constructor application and function application, we also have the following special kinds of expressions:

– Every matching m gives rise to the *matching abstraction* (matching expression) $\{\!\mid m \mid\!\}$, which might be read "match m".
  If the matching m is *unsaturated*, i.e., "waiting for arguments", then $\{\!\mid m \mid\!\}$ abstracts m into a function.
  If m is a saturated matching, then it can either succeed or fail; if it succeeds, then $\{\!\mid m \mid\!\}$ reduces to the value "returned" by m; otherwise, $\{\!\mid m \mid\!\}$ is considered ill-defined.
– we call $\oslash$ the *empty expression*; it results from matching failures — according to the above, it could also be called the "ill-defined expression".
  We use the somewhat uncommitted name "empty expression" since we will consider two interpretations of $\oslash$:
  - It can be a "manifestly undefined" expression equivalent to non-termination — following the common view that divergence is semantically equivalent to run-time errors.
  - It can be a special "error" value propagating matching failure $\lightning$, considered as an "exception" through the syntactic category of expressions.

None of the expression constructors binds any variables; we overload the $\mathsf{FV}(\_)$ notation and denote for an expression $e : \mathsf{Expr}$ its set of free variables by $\mathsf{FV}(e)$.

For the purposes of pattern matching, constructor applications of the same constructor, but with different arities, are considered incompatible.

Matchings are the syntactic category embodying the pattern analysis aspects:

– For an expression $e : \mathsf{Expr}$, the *expression matching* $\rceil e \lceil$ always succeeds and returns e, so we propose to read it "*return* e".
– $\lightning$ is the matching that always fails.
– The *pattern matching* $p \mapsto m$ waits for supply of one argument more than m; this pattern matching can be understood as succeeding on instances of the (linear) pattern $p : \mathsf{Pat}$ and then continuing to behave as the resulting instance of the matching $m : \mathsf{Match}$. It roughly corresponds to a single case alternative in languages with case expressions.
– *argument supply* $a \rhd m$ is the matching-level incarnation of function application, with the argument on the left, and the matching it is supplied to on the right. It saturates the first argument m is waiting for.
  The inclusion of argument supply into the calculus is an important source of flexibility in the design of the reduction system.
– the *alternative* $m_1 \,\|\, m_2$ will in this paper be understood sequentially: it behaves like m_1 until this fails, and only then it behaves like m_2.

Pattern matching $p \mapsto m$ binds all variables occurring in p, so $\mathsf{FV}(p \mapsto m) = \mathsf{FV}(m) - \mathsf{FV}(p)$, letting $\mathsf{FV}(m)$ denote the set of free variables of a matching m. Pattern matching is the only variable binder in this calculus — taking this into account, the definitions of free variables, bound variables, and substitution are as usual. Note that there are no matching variables; variables can only occur as patterns or as expressions.

We will omit the parentheses in matchings of the shape $a \rhd (p \mapsto m)$ since there is only one way to parse $a \rhd p \mapsto m$ in PMC.

3 Examples

Even though we have not yet introduced PMC reduction, the explanations of the syntax of PMC in the previous section should allow the reader to understand the examples presented in this section. We first show the natural embedding of the untyped λ-calculus into PMC and then continue to give translations for Haskell function definitions first using pattern matching only, then together with Boolean guards and finally together with pattern guards,

It is easy to see that the pattern matching calculus includes the λ-*calculus.* Variables and function application are translated directly, and λ-abstraction is a matching abstraction over a pattern matching that has a single-variable pattern and a result matching that immediately returns the body:

$$\lambda\ v\ .\ e := \{\!| v \mapsto \rceil e \lceil |\!\}$$

In Sect. 5 we shall see that this embedding also preserves reducibility.

As an example for the translation of *Haskell programs* into PMC, we show one that also serves as an example for non-normalisation of the functional rewriting strategy; with this program and the additional definition `bot = bot`, the functional strategy loops (detected by some implementations) on evaluation of the expression `f bot (3:[])`, although "obviously" it "could" reduce to 2:

```
f (x:xs) []     = 1
f ys     (v:vs) = 2
```

For translation into PMC, we have to decide how we treat `bot`. We could translate it directly into an application of a fixedpoint combinator to the identity function; if we call the resulting expression $\perp$, then $\perp$ gives rise to cyclic reductions. In this case, we obtain for `f bot (3:[])` the following expression:

$$\{\!|\, ((x : xs) \Mapsto [] \Mapsto \upharpoonleft 1 \upharpoonright) \mathbin{\rule{0.2em}{1.6ex}} (ys \Mapsto (v : vs) \Mapsto \upharpoonleft 2 \upharpoonright) \,|\!\}\ \perp\ (3 : [])$$

A different possibility is to recognise that the above "definition" of `bot` has as goal to produce an undefined expression; if the empty expression $\oslash$ is understood as undefined, then we could use that.

We will investigate reduction of both possibilities below, in Sect. 5.

In several functional programming languages, *Boolean guards* may be added after the pattern part of a definition equation; the failure of such a guard has the same effect as pattern matching failure: if more definition equations are present, the next one is tried. For example:

```
g (x:xs) | x > 5 = 2
g ys             = 3
```

Translation into a `case`-expression turns such a guard into a match to the Boolean constructor `True` and a default branch that redirects mismatches to the next line of the definition. In PMC, we do not need to make the mismatch case explicit, but can directly translate from the Haskell formulation. The above function `g` therefore corresponds to the following PMC expression:

$$\{\!|\, ((x : xs) \Mapsto (x > 5) \rhd \mathsf{True} \Mapsto \upharpoonleft 2 \upharpoonright) \mathbin{\rule{0.2em}{1.6ex}} (ys \Mapsto \upharpoonleft 3 \upharpoonright) \,|\!\}$$

A generalisation of Boolean guards are *pattern guards* [6]; these incorporate not only the decision aspect of Boolean guards, but also the variable binding aspect of pattern matching. In PMC, both can be represented as *saturated patterns*, i.e., as pattern matchings that already have an argument supplied to them. For a pattern guard example, we use Peyton Jones' `clunky`:

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

We attempt analogous layout for the PMC expression corresponding to the function `clunky` (with appropriate conventions, we could omit more parentheses):

$$\begin{array}{l}\{\!|\, env \Mapsto v_1 \Mapsto v_2 \Mapsto ((lookup\ \ env\ \ v_1 \rhd Just(r_1) \Mapsto \\ \qquad\qquad\qquad\qquad lookup\ \ env\ \ v_2 \rhd Just(r_2) \Mapsto \upharpoonleft r_1 + r_2 \upharpoonright) \\ \qquad\qquad\qquad\quad \mathbin{\rule{0.2em}{1.6ex}} \upharpoonleft v_1 + v_2 \upharpoonright \qquad\qquad\qquad\qquad\qquad\qquad) \,|\!\}\end{array}$$

Irrefutable patterns, in Haskell indicated by the prefix "~", match lazily, i.e., matching is delayed until one of the component variables is needed. There are no special provisions for irrefutable patterns in PMC; they have to be translated in essentially the same way as in Haskell. For example, with *body* possibly containing occurrences of `x` and `xs`, the definition:

```
q ~(x:xs) = body
```

expands into the following shape according to the Haskell report:

```
q = \ v -> (\x -> \ xs -> body )(case v of (x:xs) -> x  )
                                (case v of (x:xs) -> xs )
```

In PMC, we can turn the two function applications into saturated patterns:

$$\begin{aligned} q = \{\!\!|\, v \Mapsto\ & \{\!\!|\, v \rhd (x : xs) \Mapsto \rceil x \lceil\ |\!\!\} \rhd x \ \Mapsto \\ & \{\!\!|\, v \rhd (x : xs) \Mapsto \rceil xs \lceil |\!\!\} \rhd xs \Mapsto \mathit{body}\, |\!\!\} \end{aligned}$$

4 Standard Reduction Rules

The intuitive explanations in Sect. 2 only provide guidance to one particular way of providing a semantics to PMC expressions and matchings. In this section, we provide a set of rules that implement the usual pattern matching semantics of non-strict languages by allowing corresponding reduction of PMC expressions as they arise from translating functional programs. In particular, we do not include extensionality rules.

Formally, we define two *redex reduction* relations: $\underset{\mathsf{E}}{\longrightarrow} : \mathsf{Expr} \leftrightarrow \mathsf{Expr}$ for expressions, and $\underset{\mathsf{M}}{\longrightarrow} : \mathsf{Match} \leftrightarrow \mathsf{Match}$ for matchings. These are the smallest relations including the rules listed in the sections 4.1 to 4.3. In 4.4 we shortly discuss the characteristics of the resulting rewriting system.

We use the following conventions for metavariables: v is a variable; a, a_1, a_2, …, b, e, e_1, e_2, …, f are expressions; k, n are natural numbers; c, d are constructors; m, m_1, m_2, … are matchings; p, p_1, p_2, …, q are patterns.

4.1 Failure and Returning

Failure is the (left) unit for $\mathbin{\rule{2pt}{8pt}}$; this enables discarding of failed alternatives and transfer of control to the next alternative:

$$\lightning \mathbin{\rule{2pt}{8pt}} m \quad \underset{\mathsf{M}}{\longrightarrow} \quad m \qquad\qquad (\lightning \mathbin{\rule{2pt}{8pt}})$$

A matching abstraction where all alternatives fail represents an ill-defined case — this motivates the introduction of the empty expression into our language:

$$\{\!\!|\, \lightning \,|\!\!\} \quad \underset{\mathsf{E}}{\longrightarrow} \quad \oslash \qquad\qquad (\{\!\!|\, \lightning \,|\!\!\})$$

Empty expressions are produced only by this rule; the rules ($\oslash$@) and ($\oslash \rhd c$) below only propagate them.

Expression matchings are left-zeros for $\pmb{|}$:

$$\rceil e \lceil \,\pmb{|}\, m \quad \xrightarrow[\mathsf{M}]{} \quad \rceil e \lceil \qquad\qquad (\rceil\lceil\pmb{|})$$

Matching abstractions built from expression matchings are equivalent to the contained expression:

$$\{\!|\, \rceil e \lceil \,|\!\} \quad \xrightarrow[\mathsf{E}]{} \quad e \qquad\qquad (\{\!|\rceil\lceil|\!\})$$

4.2 Application and Argument Supply

Application of a matching abstraction reduces to argument supply inside the abstraction:

$$\{\!|\, m \,|\!\}\ a \quad \xrightarrow[\mathsf{E}]{} \quad \{\!|\, a \rhd m \,|\!\} \qquad\qquad (\{\!|\ |\!\}@)$$

Argument supply to an expression matching reduces to function application inside the expression matching:

$$a \rhd \rceil e \lceil \quad \xrightarrow[\mathsf{M}]{} \quad \rceil e\ a \lceil \qquad\qquad (\rhd\rceil\lceil)$$

No matter which of our two interpretations of the empty expression we choose, it absorbs arguments when used as function in an application:

$$\oslash\ e \quad \xrightarrow[\mathsf{E}]{} \quad \oslash \qquad\qquad (\oslash @)$$

Analogously, failure absorbs argument supply:

$$e \rhd \lightning \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad\qquad (\rhd\lightning)$$

Argument supply distributes into alternatives:

$$e \rhd (m_1 \,\pmb{|}\, m_2) \quad \xrightarrow[\mathsf{M}]{} \quad (e \rhd m_1) \,\pmb{|}\, (e \rhd m_2) \qquad\qquad (\rhd\pmb{|})$$

4.3 Pattern Matching

Everything matches a variable pattern; this matching gives rise to substitution:

$$a \rhd v \mathrel{|\!\!\Rightarrow} m \quad \xrightarrow[\mathsf{M}]{} \quad m[v \backslash a] \qquad\qquad (\rhd v)$$

Matching constructors match, and the proviso can always be ensured via α-conversion (for this rule to make sense, linearity of patterns is important):

$$c(e_1, \dots, e_n) \rhd c(p_1, \dots, p_n) \mathrel{|\!\!\Rightarrow} m \quad \xrightarrow[\mathsf{M}]{} \quad e_1 \rhd p_1 \mathrel{|\!\!\Rightarrow} \cdots e_n \rhd p_n \mathrel{|\!\!\Rightarrow} m$$
$$\text{if } \mathsf{FV}(c(e_1, \dots, e_n)) \cap \mathsf{FV}(c(p_1, \dots, p_n)) = \{\} \qquad\qquad (c \rhd c)$$

Matching of different constructors fails:

$$d(e_1, \dots, e_k) \rhd c(p_1, \dots, p_n) \mathrel{|\!\!\Rightarrow} m \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad \text{if } c \neq d \text{ or } k \neq n \quad (d \rhd c)$$

For the case where an empty expression is matched against a constructor pattern, we consider two different right-hand sides:

- The calculus $\mathsf{PMC}_{\oslash}$ interprets the empty expression as equivalent to non-termination, so constructor pattern matchings are strict in the supplied argument:

$$\oslash \rhd c(p_1, \ldots, p_n) \mapsto\!\!\!\Rightarrow m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \rceil \oslash \lceil \qquad\qquad (\oslash \rhd c \to \oslash)$$

- The calculus $\mathsf{PMC}_{\lightning}$ interprets the empty expression as propagating the exception of matching failure, and "resurrects" that failure when matching against a constructor:

$$\oslash \rhd c(p_1, \ldots, p_n) \mapsto\!\!\!\Rightarrow m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lightning \qquad\qquad (\oslash \rhd c \to \lightning)$$

For statements that hold in both $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$, we let the rule name $(\oslash \rhd c)$ stand for the rule $(\oslash \rhd c \to \oslash)$ in $\mathsf{PMC}_{\oslash}$ and for $(\oslash \rhd c \to \lightning)$ in $\mathsf{PMC}_{\lightning}$.

4.4 Rewriting System Aspects

For a reduction rule (R), the one-step *redex* reduction relation defined by that rule is written $\xrightarrow[(R)]{}$; this will either have only expressions, or only matchings in its domain and range. Furthermore, we let $\xrightarrow[(R)]{\circ}$ be the one-step reduction relation closed under expression and matching construction.

Each of the rewriting systems $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$ formed by the reduction rules introduced in sections 4.1 to 4.3 consists of nine first-order term rewriting rules, two rule-schemata $(\oslash \rhd c)$ and $(d \rhd c)$ — parameterised by the constructors and the arities — that involve the binding constructor $\mapsto\!\!\!\Rightarrow$, but not any bound variables, the second-order rule $(\rhd v)$ involving substitution, and the second-order rule schema $(c \rhd c)$ for pattern matching that re-binds variables.

The substituting rule $(\rhd v)$ has almost the same syntactical characteristics as β-reduction, and can be directly reformulated as a CRS rule. (CRS stands for *combinatory reduction system* [11,17].)

The pattern matching rule schema $(c \rhd c)$ involves binders binding multiple variables, but its individual rules still could be reformulated as CRS rules.

The whole system is neither orthogonal nor does it have any other properties like weak orthogonality for which the literature provides confluence proofs; we describe a confluence proof in Sect. 6.

5 Reduction Examples

For the translation of λ-calculus into PMC it is easy to see that every β-reduction can be emulated by a three-step reduction sequence in PMC:

$$\begin{aligned}(\lambda\ v\ .\ e)\ a = \{\!|\, v \mapsto\!\!\!\Rightarrow \rceil e \lceil \,|\!\}\ a \ &\xrightarrow[(\{\!|\ |\!\}@)]{}\ \{\!|\, a \rhd v \mapsto\!\!\!\Rightarrow \rceil e \lceil \,|\!\} \ \xrightarrow[(\rhd v)]{\circ}\ \{\!|\, \rceil e \lceil [v \backslash a] \,|\!\} \\ = \{\!|\, \rceil e[v \backslash a] \lceil \,|\!\} \ &\xrightarrow[(\{\!|\rceil\lceil|\!\})]{}\ e[v \backslash a]\end{aligned}$$

By induction over λ-terms and PMC-reductions starting from translations of λ-terms one can show that such reductions can never lead to PMC expressions containing constructors, failure, $\oslash$, or alternatives, and can only use the four rules $(\{\!|\ |\!\}@)$, $(\rhd v)$, $(\{\!|\rceil\lceil|\!\})$, and $(\rhd\rceil\lceil)$. Of these, the first three make up the translation of β-reduction, and the last can only be applied to "undo" the effect of a "premature" application of $(\{\!|\ |\!\}@)$. In addition, for each PMC reduction sequence starting from the translation of a λ-term t, we can construct a corresponding β-reduction sequence starting from t showing that the only real difference between arbitrary *PMC* reduction sequences of translations of λ-terms and β-reduction sequences is that the *PMC* reduction sequence may contain steps corresponding to "unfinished" β-steps, and "premature" $(\{\!|\ |\!\}@)$ steps.

Therefore, no significant divergence is possible, and confluence of the standard PMC reduction rules, to be shown below, implies that this embedding of the untyped λ-calculus is faithful.

For the PMC translation of the Haskell expression `f bot (3:[])`, the normalising strategy we present below produces the following reduction sequence:

$$
\begin{aligned}
&\{\!|\,((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)\,|\!\}\ \bot\ (3 : [\,])\\
&\xrightarrow[(\{\!|\ |\!\}@)]{\circ}\ \{\!|\,\bot \rhd (((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\ (3 : [\,])\\
&\xrightarrow[(\{\!|\ |\!\}@)]{}\ \{\!|\,(3 : [\,]) \rhd \bot \rhd (((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\\
&\xrightarrow[(\rhd\mathbf{|})]{\circ}\ \{\!|\,(3 : [\,]) \rhd ((\bot \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (\bot \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}
\end{aligned}
$$

From here, reduction would loop on the vain attempt to evaluate the first occurrence of $\bot$. If we replace $\bot$ with the empty expression $\oslash$. then we obtain different behaviour according to which interpretation we choose for $\oslash$:

In $\mathsf{PMC}_{\oslash}$, the empty expression propagates:

$$
\begin{aligned}
&\{\!|\,(3 : [\,]) \rhd ((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\\
&\xrightarrow[(\oslash \rhd c \rightarrow \oslash)]{\circ}\ \{\!|\,(3 : [\,]) \rhd (\rceil \oslash \lceil \,\mathbf{|}\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\\
&\xrightarrow[(\rceil\lceil\mathbf{|})]{\circ}\ \{\!|\,(3 : [\,]) \rhd \rceil \oslash \lceil\,|\!\}\ \xrightarrow[(\rhd\rceil\lceil)]{\circ}\ \{\!|\,\rceil \oslash\ (3 : [\,])\lceil\,|\!\}\ \xrightarrow[(\{\!|\rceil\lceil|\!\})]{\circ}\ \oslash\ (3 : [\,])\ \xrightarrow[(\oslash@)]{}\ \oslash
\end{aligned}
$$

In $\mathsf{PMC}_{\oslash}$, the empty expression $\oslash$ is like a runtime error: it terminates reduction in an "abnormal" way, by propagating through all constructs like an uncaught exception. In $\mathsf{PMC}_{\lightning}$, however, this exception can be caught: matching the empty expression against list construction produces a failure, and the other alternative succeeds:

$$
\begin{aligned}
&\{\!|\,(3 : [\,]) \rhd ((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\mathbf{|}\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\\
&\xrightarrow[(\oslash \rhd c \rightarrow \lightning)]{\circ}\ \{\!|\,(3 : [\,]) \rhd (\lightning \,\mathbf{|}\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil))\,|\!\}\\
&\xrightarrow[(\lightning\mathbf{|})]{\circ}\ \{\!|\,(3 : [\,]) \rhd \oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil\,|\!\}\\
&\xrightarrow[(\rhd v)]{\circ}\ \{\!|\,(3 : [\,]) \rhd (v : vs) \Mapsto \rceil 2 \lceil\,|\!\}\ \xrightarrow[(c \rhd c)]{\circ}\ \{\!|\,3 \rhd v \Mapsto [\,] \rhd vs \Mapsto \rceil 2 \lceil\,|\!\}\\
&\xrightarrow[(\rhd v)]{\circ}\ \{\!|\,[\,] \rhd vs \Mapsto \rceil 2 \lceil\,|\!\}\ \xrightarrow[(\rhd v)]{\circ}\ \{\!|\,\rceil 2 \lceil\,|\!\}\ \xrightarrow[(\{\!|\rceil\lceil|\!\})]{\circ}\ 2
\end{aligned}
$$

It is not hard to see that reduction in PMC cannot arrive at the result 2 in the example with $\bot$, even if the second alternative can, for example after the third step of the original sequence, be reduced to $\rceil 2 \lceil$: If a pattern matching $p \mapsto\!\!\!\Rightarrow m$ has been supplied with an argument a, then in the resulting $a \rhd p \mapsto\!\!\!\Rightarrow m$, the matching m can be considered as *guarded* by the *pattern guard* "$a \rhd p$" (in abuse of syntax). An alternative can only be committed to if *all* its pattern guards *succeed*; discarding — ultimately via ($↯\Vert$) — an alternative with only non-variable patterns and arguments for all patterns only works if its *first* pattern guard can be determined as mismatching. In $\mathsf{PMC}_{\oslash}$, this can only be via $(d \rhd c)$; while in $\mathsf{PMC}_{↯}$, it could also be via $(\oslash \rhd c \rightarrow ↯)$.

6 Confluence and Formalisation

Just among the first-order rules, four critical pairs arise: where the matching delimiters $\Vert$ and $\{\!|\ |\!\}$ on the one hand are eliminated by failure $↯$ or expression matchings $\rceil e \lceil$, and on the other hand are traversed by argument supply. None of these critical pairs is resolved by single steps of simple parallel reduction. It is easy to see that a shortcut rule, such as $\{\!|\, a \rhd \rceil e \lceil \,|\!\} \rightarrow e\ a$, immediately gives rise to a new critical pair that would need to be resolved by a longer shortcut rule, in this case $\{\!|\, b \rhd a \rhd \rceil e \lceil \,|\!\} \rightarrow e\ a\ b$.

A more systematic approach than introducing an infinite number of such shortcut rules is to adopt Aczel's approach [1] to parallel reduction that also reduces redexes created "upwards" by parallel reduction steps. Confluence of PMC reduction can then be shown by establishing the diamond property for the parallel reduction relations.

Using a formalisation in Isabelle-2003/Isar/HOL [15], I have performed a machine-checked proof of this confluence result.[1] Since both de Bruijn indexing and translation into higher-order abstract syntax would have required considerable technical effort and would have resulted in proving properties less obviously related to the pattern matching calculus as presented here, I have chosen as basis for the formalisation the Isabelle-1999/HOL theory set used by Vestergaard and Brotherston in their confluence proof for λ-calculus [22]. This formalisation is based on *first-order abstract syntax* and makes all the issues involved in variable renaming explicit. Therefore, the formalisation includes the rules as given in Sect. 4 with the same side-conditions; only the formalisation of the substituting variable match rule ($\rhd v$) has an additional side-condition ensuring permissible substitution in analogy with the treatment of the β-rule in [22].

Vestergaard and Brotherston employed parallel reduction in the style of the Tait/Martin-Löf proof method, and used Takahashi's proof of the diamond property via complete developments. For PMC, we had to replace this by the Aczel-style extended parallel reduction relations, and a direct case analysis for the diamond property of these relations.

Due to the fact that we are employing two mutually recursive syntactic categories (in Isabelle, the argument list of constructors actually counts as a third

[1] The proof is available at URL: http://www.cas.mcmaster.ca/˜kahl/PMC/

category in that mutual recursion), and due to the number of constructors of the pattern matching calculus (twelve including the list constructors, as opposed to three in the λ-calculus), the number of constituent positions in these constructors (twelve — including those of list construction — versus three), and the number of reduction rules (thirteen versus one), there is considerable combinatorial blow-up in the length of both the formalisation and the necessary proofs.

7 Normalisation

Since PMC is intended to serve as operational semantics for *lazy* functional programming with pattern matching, we give a reduction strategy that reduces expressions and matchings to *strong head normal form* (SHNF), see, e.g., [19, Sect. 4.3] for an accessible definition. With the set of rules defined in Sect. 4, the following facts about SHNFs are easily seen:

- Variables, constructor applications, the empty expression $\oslash$, failure $\lightning$, expression matchings $\rceil e \lceil$, and pattern matchings $p \Mapsto m$ are already in SHNF.
- All rules that have an application $f\ a$ at their top level have a metavariable for a, and none of these rules has a metavariable for f, so $f\ a$ is in SHNF if f is in SHNF and $f\ a$ is not a redex.
- A matching abstraction $\{\!|\, m \,|\!\}$ is in SHNF if m is in SHNF, unless $\{\!|\, m \,|\!\}$ is a redex for one of the rules $(\{\!|\, \lightning \,|\!\})$ or $(\{\!|\, \rceil\lceil \,|\!\})$.
- Since all alternative rules have metavariables for m_2, an alternative $m_1 \mathbin{\pmb{|}} m_2$ is in SHNF if m_1 is in SHNF, unless $m_1 \mathbin{\pmb{|}} m_2$ itself is a redex.
- No rule for argument supply $a \rhd m$ has a metavariable for m, and all rules for argument supply $a \rhd m$ that have non-metavariable a have m of shape $c(p_1, \ldots, p_n) \Mapsto m'$. Therefore, if $a \rhd m$ is not a redex, it is in SHNF if m is in SHNF and, whenever m is of the shape $c(p_1, \ldots, p_n) \Mapsto m'$, a is in SHNF, too.

Due to the homogenous nature of its rule set, PMC therefore has a deterministic strategy for reduction of applications, matching abstractions, alternatives, and argument supply to SHNF:

- If an application $f\ a$ is a redex, reduce it; otherwise if f is not in SHNF, proceed into f.
- For a matching abstraction $\{\!|\, m \,|\!\}$, if m is not in SHNF, proceed into m, otherwise reduce $\{\!|\, m \,|\!\}$ if it is a redex.
- For an alternative $m_1 \mathbin{\pmb{|}} m_2$, if m_1 is not in SHNF, proceed into m_1, otherwise reduce $m_1 \mathbin{\pmb{|}} m_2$ if it is a redex.
- If an argument supply $a \rhd m$ is a redex, reduce it (this is essential for the case where m is of shape $m_1 \mathbin{\pmb{|}} m_2$, which is not necessarily in SHNF, and $(\rhd \pmb{|})$ has to be applied). Otherwise, if m is not in SHNF, proceed into m. If m is of the shape $c(p_1, \ldots, p_n) \Mapsto m'$, and a is not in SHNF, proceed into a.

Matching abstractions and alternatives are redexes only if the selected constituent is in SHNF — this simplified the formulation of the strategy for these cases.

This strategy induces a deterministic normalising strategy in the usual way.

8 Related Work

In Peyton Jones' book [18], the chapter 4 by Peyton Jones and Wadler introduces a "new built-in value FAIL, which is returned when a pattern-match fails" (p. 61). In addition, their "enriched λ-calculus" also contains an alternative constructor, for which FAIL is the identity, thus corresponding to our failure $\lightning$. However, FAIL only occurs in contexts where there is a right-most ERROR alternative (errors ERROR are distinguished from non-termination $\perp$), so there is no opportunity to discover that, in our terms, $\{\!|\, \mathsf{FAIL} \,|\!\} = \mathsf{ERROR}$. Also, errors always propagate; since ERROR corresponds to our empty expression $\oslash$, their error behaviour corresponds to our rule $(\oslash \rhd c \rightarrow \oslash)$. Wadler's chapter 5, one of the standard references for compilation of pattern matching, contains a section about optimisation of expressions containing alternative and FAIL, arguing along lines that would be simplified by our separation into matchings and expressions.

Similarly, Tullsen includes a primitive "failure combinator" that never succeeds among his "First Class Patterns" combinators extending Haskell [21]. He uses a purely semantic approach, with functions into a Maybe type or, more generally, into a $\mathsf{MonadPlus}$ as "patterns". In this way, he effectively embeds our two-sorted calculus into the single sort of Haskell expressions, with a fixed interpretation. However, since expressions are non-patterns, Tullsen's approach treats them as standard Haskell expressions and, therefore, does not have the option to consider "resurrecting failures" as in our rule $(\oslash \rhd c \rightarrow \lightning)$. Harrison *et al.* follow a similar approach for modelling Haskell's evaluation-on-demand in detail [9]; they consider "case branches `p -> e`" as separate syntactical units — such a case branch is a PMC matching $p \Mapsto e$ — and interpret them as functions into a Maybe type; the interpretation of case expressions translates failure into bottom, like in Tullsen's approach.

Erwig and Peyton Jones, together with their proposal of pattern guards, in [6] also proposed to use a `Fail` exception for allowing pattern matching failure as result of conventional Haskell expressions, and explicitly mention the possibility to catch this exception in the same *or in another* case expression. This is the only place in the literature where we encountered an approach somewhat corresponding to our rule $(\oslash \rhd c \rightarrow \lightning)$; we feel that our formalisation in the shape of $\mathsf{PMC}_{\lightning}$ can contribute significantly to the further exploration of this option.

Van Oostrom defined an untyped λ-calculus with patterns in [16], abstracting over (restricted) λ-terms. This calculus does not include mismatch rules and therefore requires complicated encoding of typical multi-pattern definitions.

Typed pattern calculi with less relation to lazy functional programming are investigated by Delia Kesner and others in, e.g., [3,8]. Patterns in these calculi can be understood as restricted to those cases that are the result of certain kinds of pattern compilation, and therefore need not include any concern for incomplete alternatives or failure propagation.

As explained in the introduction, pattern matching can be seen as an internalisation of term rewriting; PMC represents an internalisation of the functional rewriting strategy described for example in [19]. Internalisation of general, nondeterministic term rewriting has been studied by H. Cirstea, C. Kirchner and others as the rewriting calculus, also called ρ-calculus [4,5], and, most recently, in typed variants as “pure pattern type systems” [2]. The ρ-calculus is parameterised by a theory modulo which matching is performed; this can be used to deal with *views* [23]. Since the ρ-calculus allows arbitrary expressions as patterns, confluence holds only under restriction to *call-by-value* strategies. The ρ-calculus has unordered sets of alternatives that can also be empty; in [7] a distinction between matching failure and the empty alternative has been added for improving support for formulating rewriting strategies as ρ-calculus terms. Since matching failure in the ρ-calculus is an expression constant, it can occur as function or constructor argument, and proper propagation of failure must be enforced by call-by-value evaluation.

Maranget [13] describes “automata with failures” as used by several compilers. Translated into our formalism, this introduces a `default` matching that never matches, but transfers control to the closest enclosing alternative containing a wildcard (variable) pattern. This can be seen as closely related with our rule $(\oslash \rhd c \to \lightning)$; Maranget-1994 used this feature as a way of allowing backtracking during pattern matching. In [12], this is extended to labelled exceptions, and can also be understood as a way of implementing sharing between alternatives.

9 Conclusion and Outlook

The pattern matching calculus $\mathsf{PMC}_{\oslash}$ turns out to be a simple and elegant formalisation of the operational pattern matching semantics of current non-strict functional programming languages. $\mathsf{PMC}_{\oslash}$ is a confluent reduction system with a simple deterministic normalising strategy, and therefore does not require the complex priorisation mechanisms of the functional rewriting strategy or other pattern matching definitions.

In addition, we have shown how changing a single rule produces the new calculus $\mathsf{PMC}_{\lightning}$, which results in “more successful” evaluation, but is still confluent and normalising, Therefore, $\mathsf{PMC}_{\lightning}$ is a promising foundation for further exploration of the “failure as exception” approach proposed by Erwig and Peyton Jones, for turning it into a basis for programming language implementations, and for relating it with Maranget’s approach.

The technical report [10] shows, besides other details, also a simple polymorphic typing discipline, and the inclusion of an explicit fixedpoint combinator, which preserves confluence and normalisation.

The next step will be an investigation of theory and denotational semantics of both calculi: For $\mathsf{PMC}_{\oslash}$, the most natural approach will be essentially the Maybe semantics of pattern matching as proposed in [21,9]. For $\mathsf{PMC}_{\lightning}$, the semantic domain for expressions needs to include an alternative for failure, too, to represent the semantics of empty expressions $\oslash$.

We also envisage that pattern matching calculi would be a useful basis for an interactive program transformation and reasoning systems for Haskell, similar to what SPARKLE [14] is for CLEAN.

References

1. P. Aczel. *A general Church-Rosser theorem.* unpublished note, see [20], 1978.
2. G. Barthe *et al.*. *Pure Patterns Type Systems.* In: POPL 2003. ACM, 2003.
3. V. Breazu-Tannen, D. Kesner, L. Puel. *A Typed Pattern Calculus.* In: Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science, pp. 262–274, Montreal, Canada, 1993. IEEE Computer Society Press.
4. H. Cirstea, C. Kirchner. *Combining Higher-Order and First-Order Computation Using ρ-calculus: Towards a semantics of* ELAN. In D. Gabbay, M. de Rijke, eds., Frontiers of Combining Systems 2, Oct. 1998, pp. 95–120. Wiley, 1999.
5. H. Cirstea, C. Kirchner. *The rewriting calculus — Part I* and *II.* Logic Journal of the Interest Group in Pure and Applied Logics **9** 427–498, 2001.
6. M. Erwig, S. P. Jones. *Pattern Guards and Transformational Patterns.* In: Haskell Workshop 2000, ENTCS **41 no. 1**, pp. 12.1–12.27, 2001.
7. G. Faure, C. Kirchner. *Exceptions in the rewriting calculus.* In S. Tison, ed., RTA 2002, LNCS **2378**, pp. 66–82. Springer, 2002.
8. J. Forest, D. Kesner. *Expression Reduction Systems with Patterns.* In R. Nieuwenhuis, ed., RTA 2003, LNCS **2706**, pp. 107–122. Springer, 2003.
9. W. L. Harrison, T. Sheard, J. Hook. *Fine Control of Demand in Haskell.* In: Mathematics of Program Construction, MPC 2002, LNCS. Springer, 2002.
10. W. Kahl. *Basic Pattern Matching Calculi: Syntax, Reduction, Confluence, and Normalisation.* SQRL Rep. 16, Software Quality Res. Lab., McMaster Univ., 2003.
11. J. W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhD thesis.
12. F. Le Fessant, L. Maranget. *Optimizing Pattern Matching.* In X. Leroy, ed., ICFP 2001, pp. 26–37. ACM, 2001.
13. L. Maranget. *Two Techniques for Compiling Lazy Pattern Matching.* Technical Report RR 2385, INRIA, 1994.
14. M. de Mol, M. van Eekelen, R. Plasmeijer. *Theorem Proving for Functional Programmers —* Sparkle*: A Functional Theorem Prover.* In T. Arts, M. Mohnen, eds., IFL 2001, LNCS **2312**, pp. 55–71. Springer, 2001.
15. T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS **2283**. Springer, 2002.
16. V. van Oostrom. *Lambda Calculus with Patterns.* Technical Report IR 228, Vrije Universiteit, Amsterdam, 1990.
17. V. van Oostrom, F. van Raamsdonk. *Comparing combinatory reduction systems and higher-order rewrite systems.* In J. Heering, K. Meinke, B. Möller, T. Nipkow, eds., HOA '93, LNCS **816**, pp. 276–304. Springer, 1993.
18. S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
19. R. Plasmeijer, M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley, 1993.
20. F. van Raamsdonk. *Higher-order Rewriting.* In Terese, ed., Term Rewriting Systems, Chapt. 11, pp. 588–667. Cambridge Univ. Press, 2003.
21. M. Tullsen. *First Class Patterns.* In E. Pontelli, V. Santos Costa, eds., PADL 2000, LNCS **1753**, pp. 1–15. Springer, 2000.
22. R. Vestergaard, J. Brotherston. *A Formalised First-Order Confluence Proof for the λ-Calculus using One-Sorted Variable Names.* In A. Middeldorp, ed., RTA 2001, LNCS **2051**, pp. 306–321. Springer, 2001.
23. P. Wadler. *Views: A Way for Pattern Matching to Cohabit with Data Abstraction.* In S. Munchnik, ed., POPL 1987, pp. 307–313. ACM, 1987.