Basic Pattern Matching Calculi: Syntax, Reduction, Confluence, and Normalisation

Wolfram Kahl
Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1
URL: http://www.cas.mcmaster.ca/~kahl/

October 12, 2003

Abstract

The pattern matching calculus is a refinement of λ-calculus that integrates mechanisms appropriate for fine-grained modelling of non-strict pattern matching.

In comparison with the functional rewriting strategy that is usually employed to define the operational semantics of pattern-matching in non-strict functional programming languages like Haskell or Clean, the pattern matching calculus allows simpler and more local definitions to achieve the same effects.

The main device of the calculus is to further emphasise the clear distinction between matching failure and undefinedness already discussed in the literature by embedding into expressions the separate syntactic category of matchings. This separation is also important to properly restrain the possible effects of the non-monotonicity that a naïve treatment of matching alternatives would exhibit. The language arising from that distinction turns out to naturally encompass the pattern guards of Peyton Jones and Erwig and conventional Boolean guards as special cases of the intermediate stages of matching reduction.

By allowing a confluent reduction system and a normalising strategy, the pattern matching calculus provides a new basis for operational semantics of non-strict programming languages and also for implementations.

McMaster University

SQRL Report No. 16

Contents

1 Introduction **3**

2 Abstract Syntax **5**

3 Examples **7**

3.1 λ-Calculus . . . 7

3.2 Simple Haskell Pattern Matching . . . 8

3.3 Boolean Guards . . . 8

3.4 Pattern Guards . . . 8

3.5 Irrefutable Patterns . . . 9

4 Standard Reduction Rules **9**

4.1 Failure and Returning . . . 10

4.2 Application and Argument Supply . . . 11

4.3 Pattern Matching . . . 11

4.4 Closure . . . 12

4.5 The Rewriting System PMC . . . 12

5 Reduction Examples **13**

5.1 λ-Calculus . . . 13

5.2 Simple Haskell Pattern Matching . . . 14

6 Confluence and Formalisation **17**

7 Normalisation **19**

8 Typing **21**

9 Recursion **22**

10 Related Work **23**

11 Conclusion and Outlook **25**

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: A function definable by a certain kind of term rewriting system, i.e., by a set of term rewriting rules, can be represented by a single expression in an appropriate λ-calculus.

For example, certain very simple functional programs can be explained as *applicative* TRSs: each function is defined in a single "equation" where the left-hand side is an application of the function symbol to only variables, like, for example, the Haskell function definition:

```
f x y = 2 * x + 3 * y
```

The mechanism that allows internalisation of applicative term rewriting systems is λ-abstraction: functions that can be defined by an applicative TRS can also be represented by a single λ-expression; the example function `f` from above corresponds to $\lambda\ x\ .\ \lambda\ y\ .\ 2 * x + 3 * y$.

Recursive function definitions are internalised using fixed-point combinators, either defined fixed-point combinators as in untyped λ-calculi, or primitive fixed-point combinators as in typed λ-calculi. For example, for the factorial function, the straightforward Haskell definition

```
factorial n = if n == 0 then 1 else n * factorial (n-1)
```

is turned into the following λ expression involving the fixed-point combinator Y:

$$Y\ (\lambda\ f\ .\ \lambda\ n\ .\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * f\ (n-1))$$

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 priority rewriting strategy with a complex definition.

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`, we would have:

```
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. (We discuss related work and provide references in Sect. 10.)

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.

This allows straightforward internalisation of pattern matching definitions without the ballast of having to introduce new variables like `zs` for the `case` variant (the syntax will be explained in detail in Sect. 2):

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

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

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 in pure term rewriting systems. Our matching language allows on the one hand direct transliteration of such prioritised definitions without additional cost, and on the other hand 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.

An interesting 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 runtime error, and also has such "successful" reductions for some terms that reduce to runtime errors in the first system.

Similar mechanisms have been proposed in the literature, see Sect. 10; 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 in the next section, in Sect. 3 we give a few examples that show how the pattern matching calculus can easily encode λ-calculus and Haskell pattern matching including Boolean and pattern guards. Sect. 4 contains the standard reduction rules, including the rule with the two variants. These rules are then applied to selected examples in Sect. 5 to demonstrate the matching mechanism at work. In Sect. 6 we summarise the confluence proof that has been implemented in a theorem prover, and Sect. 7 is devoted to the normalising reduction strategy. Finally, we show a simple polymorphic typing discipline in Sect. 8 and an extension of our calculi (and confluence results) with a fixed-point combinator in Sect. 9.

Discussion of related work is concentrated in Sect. 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 case alternatives, or groups of case alternatives. Matchings that directly correspond to (groups of) case alternatives expose patterns to be matched against arguments; we say such matchings are *waiting for argument supply*. Complete case expressions correspond to expressions formed from matchings that already have an argument supplied to their outermost patterns; matchings that have arguemnts supplied to all their open patterns are called *saturated*. Argument supply to patterns is separated from performing pattern matching itself; depending on the outcomes of the involved patterm matchings, 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.

We use the following base sets:

- Var is the set of *variables*, and
- Constr is the set of *constructors*.

For the purpose of our examples, all literals, like numbers and characters, 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	::=	Var	variable
		$\mid$ Constr(Pat, ..., Pat)	constructor pattern
Expr	::=	Var	variable
		$\mid$ Constr(Expr, ..., Expr)	constructor application
		$\mid$ Expr Expr	function application
		$\mid$ $\{\!\|$ Match $\|\!\}$	matching abstraction
		$\mid$ $\oslash$	empty expression
Match	::=	$\uparrow$Expr$\uparrow$	expression matching
		$\mid$ $\lightning$	failure
		$\mid$ Pat $\Mapsto$ Match	pattern matching
		$\mid$ Expr $\triangleright$ Match	argument supply
		$\mid$ Match $\mid$ 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 : Pat, we denote its set of free variables by 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) $\{\!| m |\!\}$, which might be read "match m".
  If the matching m is *unsaturated*, i.e., "waiting for arguments", then $\{\!| m |\!\}$ abstracts m into a function.
  If m is a saturated matching, then it can either succeed or fail; if it succeeds, then $\{\!| m |\!\}$ reduces to the value "returned" by m; otherwise, $\{\!| m |\!\}$ 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 shall 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 considered as an "exception" through the syntactic category of expressions.

None of the expression constructors binds any variables; we overload the FV(_) notation and denote for an expression e : Expr its set of free variables by 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 that embodies the pattern analysis aspects:

- For an expression e : Expr, the *expression matching* $\uparrow$Expr$\uparrow$ can be seen as the matching that 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* $a \triangleright m$ 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 and permits the design of the reduction system to implement separation of the concerns of on the one hand traversing the boundary between expressions and matchings and on the other hand matching patterns against the right arguments.
- the *alternative* $m_1 \mid m_2$ 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 $a \triangleright (p \Mapsto m)$ since there is only one way to parse $a \triangleright p \Mapsto m$ 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 patter guards,

3.1 λ-Calculus

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:

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

After introducing PMC reduction in Sect. 4, we shall see in 5.1 that this embedding also preserves reducibility.

3.2 Simple Haskell Pattern Matching

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:

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

bot = bot
```

With the above program, the functional strategy loops (detected by some implementations) on evaluation of the expression `f bot (3:[])`, although "obviously" it "could" reduce to `2`.

For translation into PMC, we have to decide how we treat `bot`. We could translate it directly into an application of a fixed-point combinator to the identity function; if we call the resulting expression $\bot$, then $\bot$ gives rise to cyclic reductions.

In this case, we obtain for `f bot (3:[])` the following expression:

$$\{\!|\, ((x : xs) \Mapsto [] \Mapsto \lceil 1 \rceil) \mid (ys \Mapsto (v : vs) \Mapsto \lceil 2 \rceil) \,|\!\} \triangleright \bot \triangleright (3 : [])$$

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.

3.3 Boolean Guards

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
```

Translated into a `case`-expression, such a guard will be a match to the Boolean constructor `True`, so we translate Boolean guards accordingly. The above function `g` therefore corresponds to the following PMC expression:

$$\{\!|\, ((x : xs) \Mapsto (x > 5) \triangleright \mathsf{True} \Mapsto \lceil 2 \rceil) \mid (ys \Mapsto \lceil 3 \rceil) \,|\!\}$$

3.4 Pattern Guards

A generalisation of Boolean guards are *pattern guards* [EJ01] that 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):

$$\{\!|\, env \Rightarrow v_1 \Rightarrow v_2 \Rightarrow ((lookup\ env\ v_1 \triangleright Just(r_1) \Rightarrow lookup\ env\ v_2 \triangleright Just(r_2) \Rightarrow \uparrow r_1 + r_2 \uparrow) \,|\, \uparrow v_1 + v_2 \uparrow)\ |\!\}$$

3.5 Irrefutable Patterns

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, assume that for some *body* that may contain occurrences of `x` and `xs`, the function `q` is defined as follows:

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

According to the Haskell report, this definition is expanded into the following shape:

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

In PMC, we would use essentially the same shape, only turning the two function applications into saturated patterns:

$$q = \{\!|\, v \Rightarrow \{\!|\, v \triangleright (x : xs) \Rightarrow \uparrow x \uparrow \,|\!\} \triangleright x \Rightarrow \{\!|\, v \triangleright (x : xs) \Rightarrow \uparrow xs \uparrow \,|\!\} \triangleright xs \Rightarrow body\ |\!\}$$

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. Not all laws that can be arrived at from this intuitive

understanding are useful or necessary in a standard reduction system.

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:

- $\underset{\mathsf{E}}{\longrightarrow}$: Expr $\leftrightarrow$ Expr for expressions, and
- $\underset{\mathsf{M}}{\longrightarrow}$: Match $\leftrightarrow$ Match for matchings.

These are the smallest relations including the rules listed in the sections 4.1 to 4.3. In 4.4 we explicitly give the definition of context closure for these relations, and in 4.5 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 |; this enables discarding of failed alternatives and transfer of control to the next alternative:

$$\lightning \,|\, m \quad \underset{\mathsf{M}}{\longrightarrow} \quad m \qquad (\lightning\,|)$$

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

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

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

Expression matchings are left-zeros for |:

$$\uparrow e \uparrow \,|\, m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \uparrow e \uparrow \qquad (\uparrow\uparrow\,|)$$

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

$$\{\!|\uparrow e \uparrow|\!\} \quad \underset{\mathsf{E}}{\longrightarrow} \quad e \qquad (\{\!|\uparrow\uparrow|\!\})$$

4.2 Application and Argument Supply

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

$$\{\!|\, m \,|\!\}\ a \quad \underset{\mathsf{E}}{\longrightarrow} \quad \{\!|\, a \triangleright m \,|\!\} \qquad (\{\!|\,|\!\}@)$$

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

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

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

$$\oslash\ e \quad \underset{\mathsf{E}}{\longrightarrow} \quad \oslash \qquad (\oslash@)$$

Analogously, failure absorbs argument supply:

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

Argument supply distributes into alternatives:

$$e \triangleright (m_1 \,|\, m_2) \quad \underset{\mathsf{M}}{\longrightarrow} \quad (e \triangleright m_1) \,|\, (e \triangleright m_2) \qquad (\triangleright|)$$

4.3 Pattern Matching

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

$$a \triangleright v \Mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad m[v \backslash a] \qquad (\triangleright v)$$

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

$$c(e_1, \ldots, e_n) \triangleright c(p_1, \ldots, p_n) \Mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad e_1 \triangleright p_1 \Mapsto \cdots e_n \triangleright p_n \Mapsto m$$
$$\text{if } \mathsf{FV}(c(e_1, \ldots, e_n)) \cap \mathsf{FV}(c(p_1, \ldots, p_n)) = \{\} \qquad (c \triangleright c)$$

Matching of different constructors fails:

$$d(e_1, \ldots, e_k) \triangleright c(p_1, \ldots, p_n) \Mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lightning \qquad \text{if } c \neq d \text{ or } k \neq n \qquad (d \triangleright c)$$

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

- With the first rule, corresponding to interpreting the empty expression as equivalent to non-termination, constructor pattern matchings are strict in the supplied argument:

  $$\oslash \triangleright c(p_1, \ldots, p_n) \Mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lceil \oslash \rceil \qquad (\oslash \triangleright c \to \oslash)$$

  The calculus including this rule will be denoted $\mathsf{PMC}_{\oslash}$.

- With the second rule, corresponding to interpreting the empty expression as propagating the exception of matching failure, that failure is "resurrected":

  $$\oslash \triangleright c(p_1, \ldots, p_n) \Mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lightning \qquad (\oslash \triangleright c \to \lightning)$$

  The calculus including this rule will be denoted $\mathsf{PMC}_{\lightning}$; in this calculus, it is not possible to give $\oslash$ the same semantics as expressions without normal form.

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

4.4 Closure

We use the following notation: for a reduction rule (R), we let $\underset{(R)}{\longrightarrow}$ denote the one-step *redex* reduction relation defined by that rule; this will either have only expressions, or only matchings in its domain and range. Furthermore, we let $\underset{(R)}{\multimap\!\!\rightarrow}$ be the one-step reduction relation closed under expression and matching construction. Analogously, $\underset{\mathsf{E}}{\multimap\!\!\rightarrow}$ and $\underset{\mathsf{M}}{\multimap\!\!\rightarrow}$ are the contextual closures of the redex reduction relations $\underset{\mathsf{E}}{\longrightarrow}$ and $\underset{\mathsf{M}}{\longrightarrow}$:

$$\frac{e \underset{\mathsf{E}}{\longrightarrow} e'}{e \underset{\mathsf{E}}{\multimap\!\!\rightarrow} e'} \qquad \frac{m \underset{\mathsf{M}}{\longrightarrow} m'}{m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m'}$$

$$\frac{e_i \underset{\mathsf{E}}{\multimap\!\!\rightarrow} e_i'}{c(e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots e_n) \underset{\mathsf{E}}{\multimap\!\!\rightarrow} c(e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots e_n)}$$

$$\frac{f \underset{\mathsf{E}}{\multimap\!\!\rightarrow} f'}{f\ a \underset{\mathsf{E}}{\multimap\!\!\rightarrow} f'\ a} \qquad \frac{a \underset{\mathsf{E}}{\multimap\!\!\rightarrow} a'}{f\ a \underset{\mathsf{E}}{\multimap\!\!\rightarrow} f\ a'}$$

$$\frac{m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m'}{\{\!|\, m \,|\!\} \underset{\mathsf{E}}{\multimap\!\!\rightarrow} \{\!|\, m' \,|\!\}} \qquad \frac{e \underset{\mathsf{E}}{\multimap\!\!\rightarrow} e'}{\lceil e \rceil \underset{\mathsf{M}}{\multimap\!\!\rightarrow} \lceil e' \rceil}$$

$$\frac{m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m'}{p \Mapsto m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} p \Mapsto m'}$$

$$\frac{a \underset{\mathsf{E}}{\multimap\!\!\rightarrow} a'}{a \triangleright m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} a' \triangleright m} \qquad \frac{m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m'}{a \triangleright m \underset{\mathsf{M}}{\multimap\!\!\rightarrow} a \triangleright m'}$$

$$\frac{m_1 \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m_1'}{m_1 \,|\, m_2 \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m_1' \,|\, m_2} \qquad \frac{m_2 \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m_2'}{m_1 \,|\, m_2 \underset{\mathsf{M}}{\multimap\!\!\rightarrow} m_1 \,|\, m_2'}$$

4.5 The Rewriting System PMC

We let PMC as a rewriting system denote the union of all reduction rules introduced in sections 4.1 to 4.3; this rewriting system consists of:

- nine first-order term rewriting rules,

- two rule-schemata $(\oslash \triangleright c)$ and $(d \triangleright c)$ — parameterised by the constructors and the arities — that involve the binding constructor $\Mapsto$, but not any bound variables,
- the second-order rule $(\triangleright v)$ involving substitution, and
- the second-order rule schema $(c \triangleright c)$ for pattern matching that re-binds variables.

The substituting rule $(\triangleright v)$ has almost the same syntactical characteristics as β-reduction, and can be directly reformulated as a CRS rule. (CRS stands for *combinatory reduction system* [Klo80, vOvR93].)

The pattern matching rule schema $(c \triangleright c)$ involves binders binding multiple variables, but its individual rules still could be reformulated as analogous 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; after some reduction examples in the next section we present a confluence proof in Sect. 6.

5 Reduction Examples

We now take a few of the examples from Sect. 3 and show how they behave under the reduction rules from Sect. 4.

5.1 λ-Calculus

For the translation of λ-calculus into PMC defined in 3.1, it is now easy to see that every β-reduction can be emulated by a three-step reduction sequence in PMC:

$$\begin{array}{rcl}
(\lambda\ v\ .\ e)\ a & = & \{\!|\, v \Mapsto \uparrow e \uparrow \,|\!\}\ a \\
& \xrightarrow[(\{\!|\,|\!\}@)]{} & \{\!|\, a \triangleright v \Mapsto \uparrow e \uparrow \,|\!\} \\
& \xrightarrow[(\triangleright v)]{} & \{\!|\, \uparrow e \uparrow [v \backslash a] \,|\!\} \\
& = & \{\!|\, \uparrow e[v \backslash a] \uparrow \,|\!\} \\
& \xrightarrow[(\{\!|\uparrow\uparrow|\!\})]{} & e[v \backslash a]
\end{array}$$

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 only can use the four rules $(\{\!|\,|\!\}@)$, $(\triangleright v)$, $(\{\!|\uparrow\uparrow|\!\})$, and $(\triangleright|\uparrow)$. 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 $(\{\!|\,|\!\}@)$, for example:

$$\begin{array}{rcl}
(\lambda\ v\ .\ e)\ a\ b & = & \{\!|\, v \Mapsto \uparrow e \uparrow \,|\!\}\ a\ b \\
& \xrightarrow[(\{\!|\,|\!\}@)]{} & \{\!|\, a \triangleright v \Mapsto \uparrow e \uparrow \,|\!\}\ b \\
& \xrightarrow[(\{\!|\,|\!\}@)]{} & \{\!|\, b \triangleright a \triangleright v \Mapsto \uparrow e \uparrow \,|\!\} \\
& \xrightarrow[(\triangleright v)]{} & \{\!|\, b \triangleright (\uparrow e \uparrow [v \backslash a]) \,|\!\} \\
& = & \{\!|\, b \triangleright \uparrow e[v \backslash a] \uparrow \,|\!\} \\
& \xrightarrow[(\triangleright|\uparrow)]{} & \{\!|\, \uparrow e[v \backslash a]\ b \uparrow \,|\!\} \\
& \xrightarrow[(\{\!|\uparrow\uparrow|\!\})]{} & e[v \backslash a]\ b
\end{array}$$

In addition, all PMC redexes in the translation of a λ-term are $(\{\!|\,|\!\}@)$ redexes which all correspond to β-redexes, so β-normal forms translate into PMC normal forms, and for each PMC reduction sequence s starting from the translation of a λ-term t, we can construct a corresponding β-reduction sequence b starting from t such that:

- all applications of the rules $(\triangleright v)$ and $(\{\!|\uparrow|\uparrow|\!\})$ in s can be associated with a unique β-step in b, and for each of those β-steps, there is also a unique application of $(\{\!|\,|\!\}@)$ in s associated with precisely that step;
- each application of $(\triangleright|\uparrow)$ in s is in one-to-one association with an application of $(\{\!|\,|\!\}@)$ in s *associated* with any β-step.

The only real difference between arbitrary *PMC* reduction sequences of translations of λ-terms and β-reduction sequences is therefore 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.

One of the uses of this embedding is the fact that the untyped λ-calculus provides fixed-point combinators that can now also be used in untyped PMC. An extension of the presented system with explicit fixed-point combinators that can then also be used in a typed PMC will be shown below, in Sect. 9.

5.2 Simple Haskell Pattern Matching

For the PMC translation of the Haskell expression `f bot (3:[])` from 3.2, the normalising strategy we present in Sect. 7 below will produce the following reduction sequence:

$$\{\!|((x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow)|\!\} \perp (3 : [])$$

$$\xrightarrow[(\{|\,|\}@)]{} \{\!|\perp \triangleright (((x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}\ (3 : [])$$

$$\xrightarrow[(\{|\,|\}@)]{} \{\!|(3 : []) \triangleright \perp \triangleright (((x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\triangleright|)]{} \{\!|(3 : []) \triangleright ((\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (\perp \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\perp)]{} \ldots$$

From here, reduction would loop on the vain attempt to evaluate the first occurrence of $\perp$.

If we replace $\perp$ 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:

$$\{\!|(3 : []) \triangleright ((\oslash \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (\oslash \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\oslash \triangleright c \to \oslash)]{} \{\!|(3 : []) \triangleright (\uparrow \oslash \uparrow | (\oslash \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\uparrow|)]{} \{\!|(3 : []) \triangleright \uparrow \oslash \uparrow|\!\}$$

$$\xrightarrow[(\triangleright \uparrow)]{} \{\!|\uparrow \oslash\ (3 : [])\uparrow|\!\}$$

$$\xrightarrow[(\{|\uparrow\uparrow|\})]{} \oslash\ (3 : [])$$

$$\xrightarrow[(\oslash@)]{} \oslash$$

It would be consistent with $\mathsf{PMC}_{\oslash}$ to make $\oslash$ explicitly undefined by adding a rule

$$\oslash \xrightarrow[\mathsf{E}]{} \oslash$$

With that additional rule, the normalising strategy would loop here in the same way as above for $\perp$.

As long as we do not add such a looping rule, $\oslash$ in $\mathsf{PMC}_{\oslash}$ 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:

$$\{\!|(3 : []) \triangleright ((\oslash \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | (\oslash \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\oslash \triangleright c \to \lightning)]{} \{\!|(3 : []) \triangleright (\lightning | (\oslash \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\lightning|)]{} \{\!|(3 : []) \triangleright \oslash \triangleright ys \Rightarrow (v : vs) \Rightarrow 1 2\uparrow|\!\}$$

$$\xrightarrow[(\triangleright v)]{} \{\!|(3 : []) \triangleright (v : vs) \Rightarrow 1 2\uparrow|\!\}$$

$$\xrightarrow[(c \triangleright c)]{} \{\!|3 \triangleright v \Rightarrow [] \triangleright vs \Rightarrow 1 2\uparrow|\!\}$$

$$\xrightarrow[(\triangleright v)]{} \{\!|[] \triangleright vs \Rightarrow 1 2\uparrow|\!\}$$

$$\xrightarrow[(\triangleright v)]{} \{\!|1 2\uparrow|\!\}$$

$$\xrightarrow[(\{|\uparrow\uparrow|\})]{} 2$$

We may explore different evaluation paths — after the third step of the original sequence, for example, we could also continue by reducing in the right alternative:

$$\xrightarrow[(\triangleright v)]{} \{\!|(3 : []) \triangleright ((\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow) | ((v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\triangleright|)]{} \{\!|((3 : []) \triangleright (\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow)) | ((3 : []) \triangleright (v : vs) \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(c \triangleright c)]{} \{\!|((3 : []) \triangleright (\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow)) | (3 \triangleright v \Rightarrow [] \triangleright vs \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\triangleright v)]{} \{\!|((3 : []) \triangleright (\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow)) | ([] \triangleright vs \Rightarrow 1 2\uparrow))|\!\}$$

$$\xrightarrow[(\triangleright v)]{} \{\!|((3 : []) \triangleright (\perp \triangleright (x : xs) \Rightarrow [] \Rightarrow 1 1\uparrow)) | 1 2\uparrow|\!\}$$

However, from here on we can only continue to reduce the left alternative and will come to the same conclusions as above.

It is not hard to see that reduction in PMC cannot arrive at the result 2 here, for the following reasons:

- If a pattern matching $p \Rightarrow m$ has been supplied with an argument a, then in the resulting $a \triangleright p \Rightarrow m$, the matching m can be considered as *guarded* by the *pattern guard* "$a \triangleright p$" — in abuse of syntax: the structure really is $a \triangleright (p \Rightarrow m)$.
- An alternative can only be committed to if *all* its pattern guards *succeed*.
- Discarding — ultimately via $(\lightning|)$ — 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 $(d \triangleright c)$; while in $\mathsf{PMC}_{\lightning}$, it could also be via $(\oslash \triangleright c \to \lightning)$.

6 Confluence and Formalisation

Just among the first-order rules, four critical pairs arise: where the matching delimiters $\mid$ and $\{\!|_|\!\}$ on the one hand are eliminated by failure $\lightning$ or expression matchings $\uparrow e \uparrow$, and on the other hand are traversed by argument supply. None of these critical pairs is resolved by single steps of simple parallel reduction. For example, for the rules $(\{\!|\ |\!\}@)$ and $(\{\!|\uparrow\uparrow|\!\})$, we have:

$$\begin{array}{ccc} \{\!|\uparrow e \uparrow|\!\}\ a & \xrightarrow{(\{\!|\ |\!\}@)} & \{\!|\ a \triangleright \uparrow e \uparrow\ |\!\} \\ (\{\!|\uparrow\uparrow|\!\})\downarrow & & \downarrow(\triangleright\uparrow\uparrow) \\ e\ a & \xleftarrow{(\{\!|\uparrow\uparrow|\!\})} & \{\!|\uparrow e\ a \uparrow|\!\} \end{array}$$

It is easy to see that a shortcut rule, such as $\{\!|\ a \triangleright \uparrow e \uparrow\ |\!\} \to e\ a$, immediately gives rise to a new critical pair that would need to be resolved by a longer shortcut rule, in this case $\{\!|\ b \triangleright a \triangleright \uparrow e \uparrow\ |\!\} \to e\ a\ b$.

A more systematic approach than introducing an infinite number of such shortcut rules is to adopt Aczel's approach [Acz78] to parallel reduction that also reduces redexes created "upwards" by parallel reduction steps.

Therefore, we define parallel reduction relations $\underset{\mathsf{E}}{\twoheadrightarrow}$ on expressions and $\underset{\mathsf{M}}{\twoheadrightarrow}$ on matchings by transforming each reduction rule into an inference rule for parallel reduction where the premises derive a redex by performing parallel reduction on the immediate constituents (the side condition of the matching rule $(c \triangleright c)$ becomes $\mathsf{FV}(c(e'_1, \ldots, e'_n)) \cap \mathsf{FV}(c(p_1, \ldots, p_n)) = \{\}$ for the corresponding rule here, and for the mismatch rule $(d \triangleright c)$ it is unchanged "$c \neq d$ or $k \neq n$"):

$$\frac{m_1 \underset{\mathsf{M}}{\twoheadrightarrow} \lightning \quad m_2 \underset{\mathsf{M}}{\twoheadrightarrow} m'_2}{m_1 \mid m_2 \underset{\mathsf{M}}{\twoheadrightarrow} m'_2} \qquad \frac{m \underset{\mathsf{M}}{\twoheadrightarrow} \lightning}{\{\!|\ m\ |\!\} \underset{\mathsf{E}}{\twoheadrightarrow} \oslash} \qquad \frac{m_1 \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow e' \uparrow}{m_1 \mid m_2 \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow e' \uparrow} \qquad \frac{m \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow e' \uparrow}{\{\!|\ m\ |\!\} \underset{\mathsf{E}}{\twoheadrightarrow} e'}$$

$$\frac{f \underset{\mathsf{E}}{\twoheadrightarrow} \{\!|\ m'\ |\!\} \quad a \underset{\mathsf{E}}{\twoheadrightarrow} a'}{f\ a \underset{\mathsf{E}}{\twoheadrightarrow} \{\!|\ a' \triangleright m'\ |\!\}} \qquad \frac{f \underset{\mathsf{E}}{\twoheadrightarrow} \oslash}{f\ a \underset{\mathsf{E}}{\twoheadrightarrow} \oslash} \qquad \frac{a \underset{\mathsf{E}}{\twoheadrightarrow} a' \quad m \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow e' \uparrow \quad (e'\ a') \underset{@}{\longrightarrow} r}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow r \uparrow} \qquad \frac{m \underset{\mathsf{M}}{\twoheadrightarrow} \lightning}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} \lightning}$$

$$\frac{a \underset{\mathsf{E}}{\twoheadrightarrow} a' \quad m \underset{\mathsf{M}}{\twoheadrightarrow} (m'_1 \mid m'_2)}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} (a' \triangleright m'_1) \mid (a' \triangleright m'_2)} \qquad \frac{a \underset{\mathsf{E}}{\twoheadrightarrow} a' \quad m \underset{\mathsf{M}}{\twoheadrightarrow} v \Rrightarrow m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} m'[v \backslash a']} \qquad \frac{a \underset{\mathsf{E}}{\twoheadrightarrow} \oslash \quad m \underset{\mathsf{M}}{\twoheadrightarrow} c(p_1, \ldots, p_n) \Rrightarrow m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} \boxed{\uparrow \oslash \uparrow \quad \text{resp.} \quad \lightning}}$$

$$\frac{a \underset{\mathsf{E}}{\twoheadrightarrow} c(e'_1, \ldots, e'_n) \quad m \underset{\mathsf{M}}{\twoheadrightarrow} c(p_1, \ldots, p_n) \Rrightarrow m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} e'_1 \triangleright p_1 \Rrightarrow \cdots \Rrightarrow e'_n \triangleright p_n \Rrightarrow m'} \quad \text{if } \mathsf{FV}(c(e'_1, \ldots, e'_n)) \cap \mathsf{FV}(c(p_1, \ldots, p_n)) = \{\}$$

$$\frac{a \underset{\mathsf{E}}{\twoheadrightarrow} d(e'_1, \ldots, e'_k) \quad m \underset{\mathsf{M}}{\twoheadrightarrow} c(p_1, \ldots, p_n) \Rrightarrow m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} \lightning} \quad \text{if } c \neq d \text{ or } k \neq n$$

From the box above, the appropriate alternative has to be chosen depending on whether $\mathsf{PMC}_{\oslash}$ is considered or $\mathsf{PMC}_{\lightning}$ — everything else in this section applies to both calculi equally.

In the above definition of the parallel reduction relations we used the auxiliary relation $\underset{@}{\longrightarrow}$ to reduce newly created application redexes; this is just the reflexive closure of the union of the two rules $(\{\!|\ |\!\}@)$ and $(\oslash@)$:

$$e \underset{@}{\longrightarrow} e \qquad \{\!|\ m\ |\!\}\ a \underset{@}{\longrightarrow} \{\!|\ a \triangleright m\ |\!\} \qquad \oslash\ e \underset{@}{\longrightarrow} \oslash$$

In addition, we need contextual closure as usual:

$$v \underset{\mathsf{E}}{\twoheadrightarrow} v \qquad \oslash \underset{\mathsf{E}}{\twoheadrightarrow} \oslash \qquad \lightning \underset{\mathsf{M}}{\twoheadrightarrow} \lightning$$

$$\frac{e_1 \underset{\mathsf{E}}{\twoheadrightarrow} e'_1 \quad \cdots \quad e_n \underset{\mathsf{E}}{\twoheadrightarrow} e'_n}{c(e_1, \ldots, e_n) \underset{\mathsf{E}}{\twoheadrightarrow} c(e'_1, \ldots, e'_n)} \qquad \frac{f \underset{\mathsf{E}}{\twoheadrightarrow} f' \quad a \underset{\mathsf{E}}{\twoheadrightarrow} a'}{f\ a \underset{\mathsf{E}}{\twoheadrightarrow} f'\ a'} \qquad \frac{a \underset{\mathsf{E}}{\twoheadrightarrow} a' \quad m \underset{\mathsf{M}}{\twoheadrightarrow} m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} a' \triangleright m'}$$

$$\frac{m \underset{\mathsf{M}}{\twoheadrightarrow} m'}{\{\!|\ m\ |\!\} \underset{\mathsf{E}}{\twoheadrightarrow} \{\!|\ m'\ |\!\}} \qquad \frac{e \underset{\mathsf{E}}{\twoheadrightarrow} e'}{\uparrow e \uparrow \underset{\mathsf{M}}{\twoheadrightarrow} \uparrow e' \uparrow} \qquad \frac{m \underset{\mathsf{M}}{\twoheadrightarrow} m'}{p \Rrightarrow m \underset{\mathsf{M}}{\twoheadrightarrow} p \Rrightarrow m'} \qquad \frac{m_1 \underset{\mathsf{M}}{\twoheadrightarrow} m'_1 \quad m_2 \underset{\mathsf{M}}{\twoheadrightarrow} m'_2}{m_1 \mid m_2 \underset{\mathsf{M}}{\twoheadrightarrow} m'_1 \mid m'_2}$$

Since, by construction, $\underset{@}{\longrightarrow} \subseteq \overset{*}{\underset{\mathsf{E}}{\longrightarrow}}$, and therewith also $\underset{\mathsf{E}}{\twoheadrightarrow} \subseteq \overset{*}{\underset{\mathsf{E}}{\longrightarrow}}$ and $\underset{\mathsf{M}}{\twoheadrightarrow} \subseteq \overset{*}{\underset{\mathsf{M}}{\longrightarrow}}$, confluence of PMC reduction can be shown by establishing the diamond property for the parallel reduction relations.

Using a formalisation in Isabelle-2003/Isar/HOL [NPW02], 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 [VB01]. 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 $(\triangleright v)$ has an additional side-condition ensuring permissible substitution in analogy with the treatment of the β-rule in [VB01].

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 $\underset{\mathsf{E}}{\twoheadrightarrow}$ and $\underset{\mathsf{M}}{\twoheadrightarrow}$ as defined above, 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 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.

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

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).

For completeness, let us first translate the definition of SHNFs from [PvE93, Sect. 4.3] into our setting.

For this purpose, we need to make explicit our use of *metavariables*, like e and m in the rule $(\lceil e \rceil \mid m)$:

$$\lceil e \rceil \mid m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lceil e \rceil$$

An *expression pattern*, respectively a *matching pattern* is an expression, respectively a matching, with potentially some sub-expressions and sub-matchings replaced by metavariables.

For example, $\lceil e \rceil \mid m$ is a matching pattern, and if c is a constructor, then $c(4, e_2)$ is an expression pattern. (Patterns as defined in Sect. 2 may be considered as a subset of expressions for the purposes here.)

Each rule r of Sect. 4 then is considered to consist of two patterns (either two expression patterns, or two matching patterns), the *left-hand side* of r and the *right-hand side* of r.

Definition 7.1 A rule *partially matches* a matching or expression t if its left-hand side partially matches t.

A non-variable matching pattern or expression pattern p *partially matches* a matching, respectively an expression, t, if firstly the top-level syntactic constructions of p and t are the same, and secondly, letting $p_1, \ldots, p_k$ be the immediate constituents of p and $t_1, \ldots, t_k$ the immediate constituents of t, if for each $i : \mathbb{N}$ with $1 \leq i \leq k$ for which p_i that is not a variable, p_i partially matches t_i, or there exists a rule that partially matches t_i.

A term is in *strong head normal form (SHNF)* if no rule partially matches this term. □

It is easy to see that a rule that matches an expression, respectively a matching, t, also partially matches t.

We illustrate these concepts with a few examples, assuming that 2, Branch etc. are constructors and x, insert etc. are variables.

In the following table, "arity" is the number of immediate constituents of the top-level syntactic construction.

matching/expression	**top-level construction**		**arity**	**immediate constituents**
$\{\!\mid x \Mapsto \lceil 2 + x \rceil \mid\!\} \; 3$	@		2	$\{\!\mid x \Mapsto \lceil 2 + x \rceil \mid\!\}, 3$
$\mathsf{Branch}(2 + 3, \mathsf{insert}\ 7\ t_1, t_2)$	$\mathsf{Branch}(_, _, _)$	(expr. constr.)	3	$2 + 3, (\mathsf{insert}\ 7\ t_1), t_2$
$x \Mapsto 2 + x$	$\Mapsto$		2	$x, (2 + x)$
$\mathsf{EmptyTree}$	$\mathsf{EmptyTree}$	(expr. constr.)	0	
$\lceil \{\!\mid 2 \triangleright \lceil 3 + x \rceil \mid\!\} \rceil$	$\lceil _ \rceil$		1	$\{\!\mid 2 \triangleright \lceil 3 + x \rceil \mid\!\}$

Partial matching of matchings or expressions:

	p	t	p **part. mat.** t	**Reason**
(1)	$\lceil e \rceil$	$\lceil 42 \rceil$	yes	p has no non-variable imm. const.
(2)	$\lceil e \rceil \mid m$	$\lceil 42 \rceil \mid \lightning$	yes	(1); m is variable
(3)	$\{\!\mid \lceil e \rceil \mid\!\}$	$\{\!\mid \lceil 42 \rceil \mid \lightning \mid\!\}$	yes	rule $(\lceil \mid \rceil)$ part. matches $\lceil 42 \rceil \mid \lightning$, see (2)
(4)	$e \triangleright \lightning$	$42 \triangleright (56 \triangleright \lightning)$	yes	rule $(\triangleright\lightning)$ (part.) matches $56 \triangleright \lightning$
(5)	$\{\!\mid \lceil e \rceil \mid\!\}$	$\{\!\mid 42 \triangleright 56 \triangleright \lightning \mid\!\}$	yes	rule $(\triangleright\lightning)$ part. matches $42 \triangleright 56 \triangleright \lightning$, see (4)
(6)	$\{\!\mid \lceil e \rceil \mid\!\}$	$\{\!\mid v \Mapsto \lceil 42 \rceil \mid\!\}$	no	no rule part. matches $v \Mapsto \lceil 42 \rceil$

With the set of rules defined in Sect. 4, this definition of SHNFs directly induces the following facts:

- Variable expressions, constructor applications, the empty expression $\oslash$, failure $\lightning$, expression matchings $\lceil e \rceil$, and pattern matchings $p \Mapsto m$ are already in SHNF.
- All rules that have an application $f\ a$ at their top level have a variable for a, and none of these rules has a variable for f, so $f\ a$ is in SHNF if f is in SHNF and $f\ a$ is not a redex.
- A matching abstraction $\{\!\mid m \mid\!\}$ is in SHNF if m is in SHNF unless $\{\!\mid m \mid\!\}$ is a redex for one of the rules $(\{\!\mid \lightning \mid\!\})$ or $(\{\!\mid \lceil _ \rceil \mid\!\})$.
- An alternative $m_1 \mid m_2$ is in SHNF if m_1 is in SHNF unless $m_1 \mid m_2$ is a redex for one of the rules $(\lightning \mid)$ or $(\lceil _ \rceil \mid)$, since all alternative rules have a variable for m_2.
- No rules for argument supply $a \triangleright m$ have a variable for m, and all rules for argument supply $a \triangleright m$ that have non-variable a have a constructor pattern matching for m. Therefore, if $a \triangleright m$ 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:

- For an application $f\ a$, if f is not in SHNF, proceed into f, otherwise reduce $f\ a$ if it is a redex.
- For a matching abstraction $\{\!\mid m \mid\!\}$, if m is not in SHNF, proceed into m, otherwise reduce $\{\!\mid m \mid\!\}$ if it is a redex.
- For an alternative $m_1 \mid m_2$, if m_1 is not in SHNF, proceed into m_1, otherwise reduce $m_1 \mid m_2$ if it is a redex.
- If an argument supply $a \triangleright m$ is a redex, reduce it (this is essential for the case where m is of shape $m_1 \mid m_2$, which is not necessarily in SHNF, and $(\triangleright \mid)$ 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.

Applications, 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 deterministic strategy for reduction to SHNF induces a deterministic normalising strategy in the usual way.

8 Typing

When we consider typing for PMC, the most obvious question is whether matchings should be assigned the same types as expressions or whether they should use different type constructors.

It turns out that using different type constructors runs into considerable technical problems, so we present a solution where the same types are used for both matchings and expressions. It is important to keep in mind that the semantics of matchings and expressions of the same type may still be elements of different interpretations of that type.

For a family $(\mathsf{TConstr}_k)_{k\in\mathbb{N}}$ of disjoint countable sets of type constructors of arity k and a countable set TVar of type variables, types are generated by the following grammar:

$$\mathsf{Type} ::= \mathsf{TVar} \mid \mathsf{TConstr}_k(\mathsf{Type}_1, \ldots, \mathsf{Type}_k) \mid \mathsf{Type} \to \mathsf{Type}$$

Let $\mathsf{TV}(\alpha)$ denote the set of type variables contained in the type α.

We then assume that the set of constructors is organised as a family of disjoint countable sets $\mathsf{Constr}_{\alpha_1\times\cdots\times\alpha_n\to\alpha}$ for all types $\alpha_1, \ldots, \alpha_n, \alpha$ where α is the application of some type constructor to only type variables, and $\mathsf{TV}(\alpha_1) \cup \ldots \cup \mathsf{TV}(\alpha_n) \subseteq \mathsf{TV}(\alpha)$.

For a pattern matching calculus with Church-style typing, there are two obvious possibilities:

i) adding the type annotation to the pattern matching constructor: $p : \alpha \Mapsto m$ for $\alpha : \mathsf{Type}$, and using contexts containing pattern type assignments $p : \alpha$ as in e.g. [CK99a], or

ii) using conventional contexts, i.e., partial functions from variables to types, and adding type annotations to variable patterns:

$$\begin{array}{rcll} \mathsf{Pat} & ::= & \mathsf{Var} : \mathsf{Type} & \text{variable pattern} \\ & \mid & \mathsf{Constr}(\mathsf{Pat}, \ldots, \mathsf{Pat}) & \text{constructor pattern} \end{array}$$

We use the second alternative to avoid the more complicated context interactions.

First we define a function $\mathsf{PContext}$ that assigns every pattern the context of all its variables:

$$\begin{array}{lcl} \mathsf{PContext}(v : \alpha) & = & \{v \mapsto \alpha\} \\ \mathsf{PContext}(c(p_1, \ldots, p_n) & = & \bigcup_{i:\{1,..,n\}} \mathsf{PContext}(p_i) \end{array}$$

Type judgements for patterns then need no context (we let $\alpha, \alpha_i, \beta, \beta_i$ stand for types and σ for type substitutions); for variable we have the axiom $\underset{\mathsf{P}}{\vdash} (v : \alpha) : \alpha$, and for constructor patterns the following rule:

$$\frac{c : \mathsf{Constr}_{\alpha_1\times\cdots\times\alpha_n\to\alpha} \quad \underset{\mathsf{P}}{\vdash} p_1 : \sigma\ \alpha_1 \quad \cdots \quad \underset{\mathsf{P}}{\vdash} p_n : \sigma\ \alpha_n}{\underset{\mathsf{P}}{\vdash} c(p_1, \ldots, p_n) : \sigma\ \alpha}$$

The typing rule for pattern matching then is:

$$\frac{\underset{\mathsf{P}}{\vdash} p : \alpha \qquad \Gamma, \mathsf{PContext}(p) \underset{\mathsf{M}}{\vdash} m : \beta}{\Gamma \underset{\mathsf{M}}{\vdash} (p \Mapsto m) : \alpha \to \beta}$$

For constructors with type variables in the result type that do not occur in their argument types, such as the direct-sum injection `Left :: a -> Either a b` or the empty list `[] :: [a]` in Haskell, this rule allows derivation of non-principal types, too. But it is easily seen that every pattern has a principal type.

Of the remaining typing rules, only the variable rule is affected by this decision:

$$\Gamma, v : \alpha \underset{\mathsf{E}}{\vdash} v : \alpha \qquad \underset{\mathsf{E}}{\vdash} \oslash : \alpha \qquad \underset{\mathsf{M}}{\vdash} \lightning : \alpha$$

$$\frac{c : \mathsf{Constr}_{\alpha_1\times\cdots\times\alpha_n\to\alpha} \quad \Gamma \underset{\mathsf{E}}{\vdash} e_1 : \sigma\ \alpha_1 \quad \cdots \Gamma \underset{\mathsf{E}}{\vdash} e_n : \sigma\ \alpha_n}{\Gamma \underset{\mathsf{E}}{\vdash} c(e_1, \ldots, e_n) : \sigma\ \alpha} \qquad \frac{\Gamma \underset{\mathsf{E}}{\vdash} e_1 : \alpha \to \beta \qquad \Gamma \underset{\mathsf{E}}{\vdash} e_2 : \alpha}{\Gamma \underset{\mathsf{E}}{\vdash} (e_1\ e_2) : \beta}$$

$$\frac{\Gamma \underset{\mathsf{M}}{\vdash} m : \alpha}{\Gamma \underset{\mathsf{E}}{\vdash} \{\!|\, m \,|\!\} : \alpha} \qquad \frac{\Gamma \underset{\mathsf{E}}{\vdash} e : \alpha \qquad \Gamma \underset{\mathsf{M}}{\vdash} m : \alpha \to \beta}{\Gamma \underset{\mathsf{M}}{\vdash} (e \triangleright m) : \beta} \qquad \frac{\Gamma \underset{\mathsf{E}}{\vdash} e : \alpha}{\Gamma \underset{\mathsf{M}}{\vdash} \lceil e \rceil : \alpha} \qquad \frac{\Gamma \underset{\mathsf{M}}{\vdash} m_1 : \alpha \qquad \Gamma \underset{\mathsf{M}}{\vdash} m_2 : \alpha}{\Gamma \underset{\mathsf{M}}{\vdash} (m_1 \mid m_2) : \alpha}$$

Principal types follow in the same way as in the corresponding typed λ-calculi.

Since the standard reduction rules in their principally typed variants all have principal types for their left-hand sides, we obtain the subject reduction property, and in addition, reduction in typed PMC is equal to the untyped variant, so it is still confluent.

Without fixed-point combinators, it is easy to see that the termination proof can be carried over from typed λ-calculus, so in that case we have a strongly normalising calculus.

9 Recursion

In the untyped system, it is not necessary to have a special fixed-point combinator since it is possible to define fixed-point combinators for example by translation from the untyped λ-calculus.

However, in preparation for typed PMC, it makes sense to introduce a *fixed-point combinator* as an atomic expression fix already into the untyped system. The fixed-point combinator reduces via the fixed-point equation:

$$\mathsf{fix}\ e \quad \underset{\mathsf{E}}{\longrightarrow} \quad e\ (\mathsf{fix}\ e) \qquad (\mathsf{fix}\ @)$$

For the confluence proof, this rule needs to be added to the definition of parallel reduction:

$$\frac{f \underset{\mathsf{E}}{\twoheadrightarrow} \mathsf{fix} \qquad a \underset{\mathsf{E}}{\twoheadrightarrow} a' \qquad (a'\ (\mathsf{fix}\ a')) \underset{@}{\longrightarrow} r}{f\ a \underset{\mathsf{E}}{\twoheadrightarrow} r}$$

Since the fixed-point combinator interacts with application, the relation $\underset{@}{\longrightarrow}$ needs to be extended, too; we add the fixed-point rule, two rules for consumption of applications created by (fix @), and a

closure rule to the definition of $\underset{@}{\longrightarrow}$:

$$\mathsf{fix}\ e \underset{@}{\longrightarrow} e\ (\mathsf{fix}\ e) \qquad \mathsf{fix}\ \oslash \underset{@}{\longrightarrow} \oslash \qquad \mathsf{fix}\ \{\!|\, m \,|\!\} \underset{@}{\longrightarrow} \{\!|\, (\mathsf{fix}\ \{\!|\, m \,|\!\}) \triangleright m \,|\!\} \qquad \frac{e \underset{@}{\longrightarrow} \mathsf{fix}\ f}{e \underset{@}{\longrightarrow} f\ (\mathsf{fix}\ f)}$$

These extensions are sufficient to show confluence of PMC with fixed-point combinator; all this is included in the Isabelle proof.

For pattern matching in the untyped system, the fixed-point combinator can be considered as a special constructor that cannot occur in patterns. In most typed systems, it could never be matched against a constructor pattern. It is probably most natural to have its matchings against a constructor pattern fail:

$$\mathsf{fix}\ \triangleright c(p_1, \ldots, p_n) \Rrightarrow m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lightning \qquad (\mathsf{fix}\ \triangleright c)$$

Adding this rule preserves confluence; for the proof, the corresponding rule needs to be added to the definition of parallel reduction:

$$\frac{a \underset{\mathsf{E}}{\twoheadrightarrow} \mathsf{fix} \qquad m \underset{\mathsf{M}}{\twoheadrightarrow} c(p_1, \ldots, p_n) \Rrightarrow m'}{a \triangleright m \underset{\mathsf{M}}{\twoheadrightarrow} \lightning}$$

For normalisation, the fixed-point rule (fix @), like all other rules involving application at top-level, has a variable for the argument, so the strategy is not affected by the introduction of (fix @) — adding the rule (fix $\triangleright$ c) does not change that, either, since it fits in well with the other pattern matching rules.

In the typed calculus, fixed-point combinators have the standard typing:

$$\underset{\mathsf{E}}{\vdash} \mathsf{fix} : (\alpha \to \alpha) \to \alpha$$

10 Related Work

In Peyton Jones' book [PJ87], 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 $\bot$), 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 epression $\oslash$, their error behaviour corresponds to our rule ($\oslash \triangleright c \to \oslash$).

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 [Tul00]. He uses a purely semantic approach, with functions

into a Maybe type or, more generally, into a MonadPlus as "patterns". In this way, he effectively embeds our two-sorted calculus into the single sort of Haskell expressions, with a fixed interpretation given by the definition of his pattern combinators. 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 ($\oslash \triangleright c \to \lightning$).

Harrison with his co-authors follow a similar approach in the course of modelling Haskell's evaluation-on-demand in detail [HSH02]; they consider "case branches `p -> e`" as separate syntactical units — such a case branch is a PMC matching $p \Rrightarrow e$ — and interpret them as functions into a Maybe type; the interpretation of whole case expressions translates failure into bottom, like in Tullsen's approach.

Erwig and Peyton Jones, together with their proposal of pattern guards, in [EJ01] also speculatively 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 ($\oslash \triangleright c \to \lightning$). However, the correspondence is not perfect: To make their examples work as presented, we would need more general mechanisms for "resurrecting failures" from $\oslash$, such as a rule $\uparrow \oslash \uparrow \quad \underset{\mathsf{E}}{\longrightarrow} \quad \lightning$. We feel that our initial formalisation in the shape of PMC$_{\lightning}$ can contribute significantly to the further exploration of this topic.

Van Oostrom defined an untyped λ-calculus with patterns in [vO90], 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 [BTKP93, Kes94, CK99a, For02, FK03]. 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 imcomplete 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 [PvE93]. Internalisation of general, non-deterministic term rewriting has been studied by H. Cirstea, C. Kirchner and others as the rewriting calculus, also called ρ-calculus [CK99b, CK01], and, most recently, in typed variants as "pure pattern type systems" [BCKL03]. The ρ-calculus is parameterised by a theory modulo which matching is performed; among other applications, this can be used to deal with *views* [Wad87]. Since the ρ-calculus allows arbitrary expressions as patterns, confluence holds only under restriction to *call-by-value* evaluation strategies. The ρ-calculus has unordered sets of alternatives that can also be empty; in [FK02] a distinction between matching failure on the one hand, and the empty alternative on the other hand has been added with the aim of better supporting the formulation of rewriting strategies as ρ-calculus terms. Since matching failure in the ρ-calculus is an expression constant (there is no second syntactic category of matchings), it can occur as function or constructor argument, and proper propagation of failure becomes a problem that is again solved by call-by-value evaluation.

Maranget [Mar94] 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 is used as a way of allowing backtracking during pattern matching. In [LFM01], this is extended to labelled exceptions, and can also be understood as a way of implementing sharing between alternatives.

11 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 functional programming languages. When term rewriting is used as basis for the operational semantics of functional programs, semantics needs to be described using the complex functional strategy which, considered as a pure term rewriting strategy, is not normalising, and gives deterministic meaning to term rewriting systems that need not be confluent. We feel that $\mathsf{PMC}_{\oslash}$ forms a more appropriate basis by providing a confluent and normalising reduction system.

In addition, we have shown how changing a single rule produces $\mathsf{PMC}_{\lightning}$, which is still confluent and normalising, but results in "more successful" evaluation.

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 [Tul00, HSH02]. For $\mathsf{PMC}_{\lightning}$, the semantic domain for expressions needs to include an alternative for failure, too (for the semantics of empty expressions $\oslash$).

We also plan to investigate how $\mathsf{PMC}_{\lightning}$ can be turned into a basis for programming language implementations, and we envisage that the pattern matching calculi would be a useful basis for an interactive program transformation and reasoning systems for Haskell, similar to what Sparkle [dMvEP01] is for Clean.

References

[Acz78] Peter Aczel. A general Church-Rosser theorem. unpublished note, summarised for example in [vR03], 1978.

[BCKL03] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure patterns type systems. In *Principles of Programming Languages, POPL 2003*. ACM, January 2003.

[BTKP93] Val Breazu-Tannen, Delia Kesner, and Laurence Puel. A typed pattern calculus. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 262–274, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[CK99a] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In *Proceedings, 14th Annual IEEE Symposium on Logic in Computer Science*, pages 98–108, Trento, Italy, July 1999. IEEE Computer Society Press.

[CK99b] Horatiu Cirstea and Claude Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2, Oct. 1998*, Research Studies, pages 95–120. Wiley, 1999.

[CK01] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[dMvEP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers — Sparkle: A functional theorem prover. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2001, Stockholm, Sweden, Sept. 2001*, volume 2312 of *LNCS*, pages 55–71. Springer, 2001.

[EJ01] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *Proc. Haskell Workshop 2000, Montreal*, volume 41 no. 1 of *Electronic Notes in Computer Science*, pages 12.1–12.27, 2001.

[FK02] Germain Faure and Claude Kirchner. Exceptions in the rewriting calculus. In Tison [Tis02], pages 66–82.

[FK03] Julien Forest and Delia Kesner. Expression reduction systems with patterns. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, RTA 2003*, volume 2706 of *LNCS*, pages 107–122. Springer, 2003.

[For02] Julien Forest. A weak calculus with explicit operators for pattern matching and substitution. In Tison [Tis02], pages 174–191.

[HSH02] William L. Harrison, Timothy Sheard, and James Hook. Fine control of demand in Haskell. In *Mathematics of Program Construction, MPC 2002*, LNCS. Springer, 2002.

[Kes94] Delia Kesner. Reasoning about layered, wildcard and product patterns. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *ALP '94*, volume 850 of *LNCS*, pages 253–268. Springer-Verlag, 1994.

[Klo80] Jan Willem Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhD thesis.

[LFM01] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In Xavier Leroy, editor, *ICFP 2001, International Conference on Functional Programming*, pages 26–37. ACM, September 2001.

[Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Technical Report RR 2385, INRIA, October 1994.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1993.

[Tis02] Sophie Tison, editor. *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark*, volume 2378 of *LNCS*. Springer, 2002.

[Tul00] Mark Tullsen. First class patterns. In Enrico Pontelli and Vítor Santos Costa, editors, *PADL 2000*, volume 1753 of *LNCS*, pages 1–15. Springer, 2000.

[VB01] René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names (barendregt was right after all ... almost). In Aart Middeldorp, editor, *Rewriting Techniques and Applications, RTA 2001*, volume 2051 of *LNCS*, pages 306–321. Springer, 2001.

[vO90] Vincent van Oostrom. Lambda calculus with patterns. Technical Report IR 228, Vrile Universiteit, Amsterdam, November 1990.

[vOvR93] Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *HOA '93*, volume 816 of *LNCS*, pages 276–304. Springer, 1993.

[vR03] Femke van Raamsdonk. Higher-order rewriting. In Terese, editor, *Term Rewriting Systems*, volume 55 of *Cambridge Tracts Theoret. Comput. Sci.*, chapter 11, pages 588–667. Cambridge Univ. Press, 2003.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *POPL 1987*, pages 307–313. ACM, January 1987.