JFP **19** (2): 191–225, 2009. © 2009 Cambridge University Press
doi:10.1017/S0956796808007144 Printed in the United Kingdom

First-class patterns

BARRY JAY
University of Technology, Sydney, Australia
(*e-mail:* `cbj@it.uts.edu.au`)

DELIA KESNER
PPS, CNRS and Université Paris Diderot, Paris, France
(*e-mail:* `kesner@pps.jussieu.fr`)

Abstract

Pure pattern calculus supports pattern-matching functions in which patterns are first-class citizens that can be passed as parameters, evaluated and returned as results. This new expressive power supports two new forms of polymorphism. Path polymorphism allows recursive functions to traverse arbitrary data structures. Pattern polymorphism allows patterns to be treated as parameters which may be collected from various sources or generated from training data. A general framework for pattern calculi is developed. It supports a proof of confluence that is parameterised by the nature of the matching algorithm, suitable for the pure pattern calculus and all other known pattern calculi.

1 Introduction

Patterns play many roles in computation. Pattern recognition is a central concern of artificial intelligence, data mining and image processing. Design patterns provide a framework for producing software. Modern functional programming uses a pattern to query structures. Compared to the ongoing challenges of pattern recognition, pattern matching may seem quite dull, merely a concise and readable means of describing some existing functions. However, it has a deeper significance. For example, computation can be based upon rewriting theory, where pattern matching is used to match the redex of a rule against a term. This paper introduces yet another approach where patterns are ideal for describing *data structures* in a uniform manner.

For example, given the constructors Nil and Cons to represent the data structures of lists, where Nil is used for the empty list and Cons x y for the list with head x and tail y, one can define the length of a list by the following pattern-matching function

$$\begin{array}{l} \mathsf{length} = \\ \quad \mathsf{Nil} \qquad\quad \to \mathsf{Zero} \\ \mid \mathsf{Cons}\ x\ y \to \mathsf{Successor}\ (\mathsf{length}\ y). \end{array}$$

The syntax above employs the *implicit binding grammar* described in Section 4 but should be intelligible to most readers. In brief, the definition of length is given by a recursive, pattern-matching definition built from two cases, one for the pattern Nil and one for the pattern Cons $x\ y$. Also, Zero and Successor are the constructors used to build unary natural numbers. This programming style has proved to be both popular and durable: it is fundamental to functional languages, such as OCaml, Haskell and SML (available respectively at `http://caml.inria.fr/`, `http://www.haskell.org/`, `http://www.smlnj.org/`), as well as to proof assistants, such as Coq (available at `http://coq.inria.fr`) and and Isabelle (available at http://isabelle.in.tum.de/).

Various interpretations have been offered for pattern matching. One approach is to reduce pattern matching to pure λ-calculus, where each constructor is encoded by a λ-abstraction. These encodings are far from obvious, so that one needs sophisticated machinery to provide and manipulate them, as in Böhm *et al.* (1994).

A more direct approach is to generalise the λ-calculus, so that substitution is generalised to matching, as in the *λ-calculus with patterns* (van Oostrom, 1990; Klop *et al*., 2008). Now every term is either a variable, an application or a *case* $p \to s$. The encodings of constructors can be used directly in patterns, which must then be allowed to include cases. However, as explained in Section 5.1, if no condition is imposed, different reduction paths may yield incompatible results, so that *confluence* is lost. It is recovered by requiring that the patterns satisfy the *rigid pattern condition* (RPC). Unfortunately, however, the Church encodings of elementary constructors such as Cons do not satisfy the RPC so that one cannot encode the familiar pattern-matching functions in this way.

However, none of these approaches have taken full advantage of the expressive power of pattern matching, since the emphasis has always been on patterns headed by constructors (or their encodings) which makes patterns easy to understand but limits program reuse. For example, length cannot be used to count the leaves of a binary tree. To do so, one can add more cases for binary trees, but each new data structure will then require more cases.

This paper adopts a different approach. Instead of making the constructors disappear, it embraces constructors as *atoms* from which data structures are built as *compounds* (the *constructed terms* of (Jay, 2004)). For example every canonical list is the atom Nil or a compound of the form Cons $h\ t$. Every canonical tree is either a leaf or a node. In this manner, one can compute the size of an arbitrary data structure, i.e. the number of atoms it contains, using

$$\begin{array}{lcl} \mathsf{size} & = & \\ \quad y\ z & \to & \mathsf{size}\ y + \mathsf{size}\ z \\ \mid x & \to & 1. \end{array}$$

Now two cases suffice – one for compounds and one for atoms – and the size can be calculated for lists or trees or for data structures not yet introduced.

This use of constructors is subtly different from previous uses. For example, *higher order rewriting* (Klop, 2008) is based on a class of *function symbols* in which the distinction between *defined symbols*, which are subject to reduction, and *constructor symbols*, which do not have any associated reduction rule, does not appear in the computational machinery but only at the meta level. By contrast, the notion of a constructor plays a central role when defining reduction of the pure pattern calculus, and so constructors must be distinguished within the syntax, too.

Constructors also arise naturally in type declarations, e.g. in the *calculus of inductive constructions* (Pfenning & Paulin-Mohring, 1989). These are used to support pattern-matching functions for the declared type, so that the focus is on finding sets of patterns that cover a type, rather than *all* data types, as size does.

A related issue is the usual habit of applying constructors to n-tuples of arguments instead of using n applications. In this case, the size function above would require a case for each arity, instead of just two cases, one for atoms and another one for compounds.

Syntactically, all that has changed is to allow a pattern of the form $y\ z$ in which the symbol y appears at the head of the pattern, in an *active* position. This makes no sense in traditional functional programming in which active positions are restricted to constructors. Moreover, patterns like $y\ z$ do not satisfy the RPC in the sense above, so that confluence is at risk. However, confluence is maintained by restricting successful matching of the pattern $y\ z$ to compounds.

This new expressive power supports *path polymorphism*, so called because recursive functions can now traverse arbitrary paths through data structures. As well as size above, examples include *generic queries* able to select or update terms from within an arbitrary structure. A simple example of this is the function $\mathsf{updatePoint}$ which uses its first argument to update every point within an arbitrary structure. It is given by

$$\begin{array}{lcl} \mathsf{updatePoint} & = & f \to \\ \quad \mathsf{Point}\ w & \to & \mathsf{Point}\ (f\ w) \\ \mid y\ z & \to & (\mathsf{updatePoint}\ f\ y)\ (\mathsf{updatePoint}\ f\ z) \\ \mid x & \to & x. \end{array}$$

The first case handles the points, while the other two support path polymorphism.

The patterns above are all *static*, but even greater expressive power can be gained by allowing *dynamic* patterns which can be evaluated, used as arguments and returned as results. A trivial example is the generic eliminator elim which arises as follows. Each constructor c has an eliminator, given by

$$c\ y \to y.$$

The generic eliminator elim parametrises this construction with respect to c. That is, in the case above c is replaced by a variable x that is bound *outside* the case. Now there is nothing to require that x be replaced by a constructor: useful examples exist in which x is replaced by a case, which can then be applied to y (see Example 3.9). However, when patterns are so dynamic, reduction may eliminate binding occurrences, so the bindings must be made explicit. Hence a *case*

$$[\theta]\ p \to s$$

is then given by a collection of *binding symbols* θ, a pattern p and a *body* s. For example, using the syntax of the *explicit binding grammar* of Section 5, the generic eliminator elim is given by

$$\mathsf{elim} = [x]\, x \to ([y]\, x\ y \to y).$$

The eliminator for the constructor c is now obtained by simply applying elim to c. Even more interestingly, elim can be applied to the case

$$\mathsf{singleton} = [z]\, z \to \mathsf{Cons}\ z\ \mathsf{Nil},$$

thus yielding a function which extracts the unique element of a singleton list. Similarly, one can define a generic update function which can update points or other structures. It is given by

$$\begin{array}{rll} \mathsf{cs\text{-}update} = & [x]\ x \to [f]\ f \to & \\ [w] & x\ w & \to x\,(f\ w) \\ \mid [y,z] & y\ z & \to (\mathsf{cs\text{-}update}\ x\ f\ y)\ (\mathsf{cs\text{-}update}\ x\ f\ z) \\ \mid [y] & y & \to y. \end{array}$$

Note that the variable x of the pattern $x\ w$ does not appear in the binding sequence $[w]$, as it is a free variable available for substitution. Even after making the bindings explicit, the presence of free variables in patterns generates significant technical hurdles, as noted in the *original pure pattern calculus* (Jay & Kesner, 2006a). Since the same symbol x is used for both its free and binding occurrences, reduction is at risk of becoming stuck, waiting for the value of a symbol that will never be given. However, progress can be ensured by using a context to keep track of the binding symbols. The resulting reduction is confluent but *context-sensitive* (Jay & Kesner, 2006a).

The notation used here is to allow each symbol x to appear as either a *variable symbol* x or a *matchable symbol* $\widehat{x}$. Reduction is still confluent and guarantees progress but now it is also context-free. In the resulting *matchable binding grammar* (Section 2.1) the generic update is given by

$$\begin{array}{rll} \mathsf{update} = & [x]\ \widehat{x} \to [f]\ \widehat{f} \to & \\ [w] & x\ \widehat{w} & \to x\,(f\ w) \\ \mid [y,z] & \widehat{y}\ \widehat{z} & \to (\mathsf{update}\ f\ y)\ (\mathsf{update}\ f\ z) \\ \mid [y] & \widehat{y} & \to y. \end{array}$$

The generic update is representative of a large class of novel programs. It can be generalised to handle XML paths (Huang *et al.*, 2006b) or support analysis of the syntax trees of programs (Huang *et al.*, 2006a) in the style of Stratego (Visser, 2004). Perhaps more significant in the long term is that this *pattern polymorphism* unites, for the first time, pattern generation (in the sense of data mining) and pattern consumption (matching) within a single, small calculus.

In its own terms, the pure pattern calculus has met its goal of using pattern matching to provide a better account of data structures, able to support both path and pattern polymorphisms. However, it is natural to wonder how it relates to existing calculi and whether it can be improved further. Much of the structure and content of the paper is devoted to the former question while postponing the latter question to the conclusion, by which point the concepts necessary for the discussion will be fully developed.

Pattern matching appears in so many guises that it is not realistic to attempt exhaustive comparisons. Our first restriction is to consider calculi rather than programming languages, e.g. variants of λ-calculus rather than variants of programming languages, such as the 'scrap-your-boilerplate' extensions of Haskell, begun in Lämmel and Peyton Jones (2003). Our second restriction is to consider calculi whose reduction is confluent. This excludes various concurrent systems, such as those underpinning Linda (Gelernter, 1985), Klaim (De Nicola *et al.*, 1998) and muKlaim (Gorla & Pugliese, 2003). It also excludes pattern calculi intended to give general accounts of, say, rewriting, where confluence is of secondary importance. That done, there are still many different alternatives, according to the nature and representation of binding symbols, patterns, the matching operation and the reducibility of patterns and pattern-matching functions as combinations of cases. With some minor caveats, all the known variations can be captured by the general framework for describing pattern calculi which is proposed in the paper.

Section 2 introduces this general framework and considers general properties necessary to establish confluence of reduction. The oldest result of this kind achieves confluence for *greedy matching* (defined in Section 2.3) by requiring that patterns satisfy the RPC (van Oostrom, 1990). However, path and pattern polymorphisms employ patterns that are not rigid, so a new approach was required for the confluence of the pure pattern calculus (Jay & Kesner, 2006a). The latter approach can be made abstract enough to include both the earlier results as corollaries (Cirstea & Faure, 2007), though it was only illustrated by a simplified version of the original pure pattern calculus, as discussed in Section 5.1. This paper recasts the more general proof to handle matchable symbols and further generalises its premises to a single property, our *rigid matching condition* (RMC).

Section 3 defines the pure pattern calculus with matchable symbols, establishes its confluence and provides many examples of pattern-matching functions. In this calculus, patterns are first-class citizens, able to be passed as parameters, evaluated and returned as results. This allows patterns to be assembled from several sources and computed by applying primitive or even general recursive functions.

Section 4 considers pattern calculi whose patterns are *closed*, in the sense that they are protected from substitution by their enclosing case. In each calculus, confluence can be established by applying a general confluence theorem. Section 5 discusses *open pattern calculi*, where patterns may have free variables. Examples are the original pure pattern calculus, the *context-sensitive* pure pattern calculus (Jay & Kesner, 2006b) and the open ρ-calculus (Barthe *et al.*, 2003). Confluence of the original pure pattern calculus follows from a general result. Such results do not apply directly to calculi whose reduction is context-sensitive, but the context-sensitive pure pattern calculus is isomorphic to the pure pattern calculus with matchable symbols and so inherits its confluence. Confluence of the open ρ-calculus does not yet follow from our general result.

Section 6 draws conclusions and considers future developments.

2 Pattern calculi

This section provides a general framework for pattern calculi that will support the rest of the paper. Section 2.1 introduces a grammar of terms (and patterns). Sections 2.2 and 2.3 introduce matching and reduction. Section 2.4 defines properties sufficient to ensure confluence of reduction.

2.1 Patterns

Fix a countable alphabet of *symbols* (meta-variables $f, g, \ldots, w, x, y, z, \ldots$). Lists of distinct symbols are represented by the meta-variables φ, θ and γ.

Terms (meta-variables p, q, r, s, t, u, v) are given by the *matchable binding grammar*:

$$\begin{array}{llll} t ::= & & \text{(term)} & \\ & x & \text{(variable symbol)} & \mid \\ & \widehat{x} & \text{(matchable symbol)} & \mid \\ & t\ t & \text{(application)} & \mid \\ & [\theta]\, t \rightarrow t & \text{(case).} & \end{array}$$

The variable symbols, or *variables*, are available for substitution. The matchable symbols, or *matchables*, are available for matching. The application $r\ u$ applies the *function* r to its *argument* u. The case $[\theta]\, p \rightarrow s$ is formed of a *pattern* p and a *body* s linked by the list θ of *binding symbols*. These lists can be relaxed to sets, but some operations (such as α-equality) become harder. When θ is empty, then $[\theta]\, p \rightarrow s$ may be written $[\,]\, p \rightarrow s$. In the simpler examples, this notation may appear heavy, but it removes all ambiguity while keeping the algorithms simple. It also provides a convenient framework in which to discuss lighter alternatives. Note that the calculus does not require a separate alphabet of constructors. Rather, the role of constructors is played by matchable symbols $\widehat{x}$, where x does not appear as a binding symbol.

Application is left-associative, and case formation is right-associative. Application binds tighter than case. For example $[x]\, \widehat{x} \rightarrow [y]\, \widehat{x}\ \widehat{y}\ z \rightarrow y$ is equal to $[x]\, \widehat{x} \rightarrow ([y]\, ((\widehat{x}\ \widehat{y})\ z) \rightarrow y)$. Lambda-abstraction can be defined by setting $\lambda x.t$ to be $[x]\, \widehat{x} \rightarrow t$.

Free variable symbols and *free matchable symbols* of terms are now defined by

$$\begin{array}{lcllcl} \mathsf{fv}(x) & = & \{x\} & \mathsf{fm}(x) & = & \{\} \\ \mathsf{fv}(\widehat{x}) & = & \{\} & \mathsf{fm}(\widehat{x}) & = & \{x\} \\ \mathsf{fv}(r\ u) & = & \mathsf{fv}(r) \cup \mathsf{fv}(u) & \mathsf{fm}(r\ u) & = & \mathsf{fm}(r) \cup \mathsf{fm}(u) \\ \mathsf{fv}([\theta]\, p \rightarrow s) & = & (\mathsf{fv}(s) \setminus \theta) \cup \mathsf{fv}(p) & \mathsf{fm}([\theta]\, p \rightarrow s) & = & (\mathsf{fm}(p) \setminus \theta) \cup \mathsf{fm}(s). \end{array}$$

Bound symbols of terms are defined by

$$\begin{array}{lcl} \mathsf{bs}(x) & = & \{\} \\ \mathsf{bs}(\widehat{x}) & = & \{\} \\ \mathsf{bs}(r\ u) & = & \mathsf{bs}(r) \cup \mathsf{bs}(u) \\ \mathsf{bs}([\theta]\, p \rightarrow s) & = & \mathsf{bs}(p) \cup \theta \cup \mathsf{bs}(s). \end{array}$$

Hence a binding symbol $x \in \theta$ of a case $[\theta]\, p \rightarrow s$ binds the free variable occurences of x in s and its free matchable occurences in p. A term is *closed* if it has no free variables. Note that free matchables are allowed in closed terms, as they may play the role of constructors.

The pattern p in the case expression $[\theta]\, p \to s$ is said to be *linear* if every $x \in \theta$ occurs exactly once as a free matchable symbol of p. Thus for example $[x, y]\, \widehat{x}\ \widehat{y} \to x$ is linear, but $[x]\, \widehat{z}\ \widehat{x}\ \widehat{x} \to x$ and $[x]\, \widehat{z} \to x$ are not. Typically, successful pattern matching will require the pattern to be linear.

Renaming of a binding symbol x by a fresh binding symbol y in a case $[\theta]\, p \to s$ will replace x by y in θ, the matchable $\widehat{x}$ by $\widehat{y}$ in p and the variable x by y in s. More precisely, given a term t, symbols x and y define the *renamings* $\{y/x\}t$ and $\{\widehat{y}/\widehat{x}\}t$ as follows:

$$\begin{array}{lcll}
\{y/x\}x & = & y & \\
\{y/x\}z & = & z & \text{if } z \neq x \\
\{y/x\}\widehat{z} & = & \widehat{z} & \\
\{y/x\}(r\ u) & = & \{y/x\}r\ \{y/x\}u & \\
\{y/x\}([\theta]\, p \to s) & = & [\theta]\, \{y/x\}p \to \{y/x\}s & \text{if } x, y \notin \theta;
\end{array}$$

$$\begin{array}{lcll}
\{\widehat{y}/\widehat{x}\}z & = & z & \\
\{\widehat{y}/\widehat{x}\}\widehat{x} & = & \widehat{y} & \\
\{\widehat{y}/\widehat{x}\}\widehat{z} & = & \widehat{z} & \text{if } z \neq x \\
\{\widehat{y}/\widehat{x}\}(r\ u) & = & \{\widehat{y}/\widehat{x}\}r\ \{\widehat{y}/\widehat{x}\}u & \\
\{\widehat{y}/\widehat{x}\}([\theta]\, p \to s) & = & [\theta]\, \{\widehat{y}/\widehat{x}\}p \to \{\widehat{y}/\widehat{x}\}s & \text{if } x, y \notin \theta.
\end{array}$$

These renaming operations are partial operations on term syntax that will become total when applied to α-equivalence classes.

Alpha conversion is the congruence generated by the following axiom:

$$[\theta]\, p \to s \quad =_\alpha \quad [\{y/x\}\theta]\, \{\widehat{y}/\widehat{x}\}p \to \{y/x\}s \quad \text{if } x \in \theta \text{ and } y \text{ is fresh.}$$

For example, $[y]\, y\ \widehat{w}\ \widehat{y} \to y\ \widehat{y}\ x =_\alpha [z]\, y\ \widehat{w}\ \widehat{z} \to z\ \widehat{y}\ x$, assuming that w, x, y and z are distinct symbols. When it is convenient, we may, without loss of generality, assume that bound symbols are disjoint from free variable symbols and free matchable symbols.

2.2 Matches

A *substitution* (meta-variable σ) is a function from symbols to terms such that there are only finitely many symbols x such that $\sigma x \neq x$. The *domain* of σ is the (finite) set of symbols that are not mapped to themselves: $\mathsf{dom}(\sigma) = \{x \mid \sigma x \neq x\}$. When $\mathsf{dom}(\sigma) = \{x_1, \ldots, x_n\}$, the notation $\{u_1/x_1, \ldots,\ u_n/x_n\}$ represents the substitution that maps x_i to u_i for $i = 1, \ldots, n$. The notation $\{\}$ is used for the empty substitution. *Restricting* the domain of a substitution σ to a sequence or set θ is written $\sigma|_\theta$.

A *match* (meta-variable μ) is either a *successful match*, given by a substitution σ, or a *failure*, written as `fail`, or a *waiting* match, written as `wait`. The successful matches and failure are called the *decided* matches.

The usual concepts and notation associated with substitutions will be defined for arbitrary matches. The *domain* of μ is written $\mathsf{dom}(\mu)$. The domain of `fail` is the empty set, and that of wait is undefined. The set of *free variables* (respectively, *free matchables*) of σ is given by the union of the sets $\mathsf{fv}(\sigma x)$ (respectively, $\mathsf{fm}(\sigma x)$), where $x \in \mathsf{dom}(\sigma)$. Also, fail has no free variables or free matchables, and those of wait are undefined. Define the *symbols* of μ to be $\mathsf{sym}(\mu) = \mathsf{dom}(\mu) \cup \mathsf{fv}(\mu) \cup \mathsf{fm}(\mu)$. We use the predicate x avoids μ to mean $x \notin \mathsf{sym}(\mu)$. More generally, θ avoids μ if each symbol in θ avoids μ. Thus in particular, when θ avoids a match, this match must be a decided one.

The *application of a substitution* σ to the variables of a term is given by

$$\begin{array}{lcll}
\sigma x & = & \sigma(x) & \text{if } x \in \mathsf{dom}(\sigma) \\
\sigma x & = & x & \text{if } x \notin \mathsf{dom}(\sigma) \\
\sigma \widehat{x} & = & \widehat{x} & \\
\sigma(r\ u) & = & (\sigma r)\ (\sigma u) & \\
\sigma([\theta]\, p \to s) & = & [\theta]\, \sigma p \to \sigma s & \text{if } \theta \ \texttt{avoids}\ \sigma.
\end{array}$$

The restriction on the definition of $\sigma([\theta]\, p \to s)$ is necessary to avoid a *variable clash* which must not change the semantics of the term. For example, if the application of $\{y/x\}$ to $[y]\,\widehat{y} \to x$ yielded $[y]\,\widehat{y} \to y$, then it would change the status of the free variable x in the body to a binding symbol y. Dually, the application to $\{\widehat{y}/x\}$ to $[y]\, x \to y$ cannot be $[y]\,\widehat{y} \to y$. Variable clashes will be handled by α-conversion.

Lemma 2.1
For every substitution σ and term t there is an α-equivalent term t' such that $\sigma t'$ is defined. If t and t' are α-equivalent terms, then $\mathsf{fv}(t) = \mathsf{fv}(t')$ and $\mathsf{fm}(t) = \mathsf{fm}(t')$, and if $u = \sigma t$ and $u' = \sigma t'$ are both defined, then $u =_\alpha u'$.

Proof
The proofs are by straightforward inductions. □

From now on, a *term* is an α-equivalence class in the matchable binding grammar.

When defining matching, it will prove convenient to consider how to apply a substitution to the matchable symbols of a term, even though this will not happen in reduction. Given a substitution σ and a term t define $\widehat{\sigma} t$ as follows:

$$\begin{array}{lcll}
\widehat{\sigma} x & = & x & \\
\widehat{\sigma}\, \widehat{x} & = & \sigma(x) & \text{if } x \in \mathsf{dom}(\sigma) \\
\widehat{\sigma}\, \widehat{x} & = & \widehat{x} & \text{if } x \notin \mathsf{dom}(\sigma) \\
\widehat{\sigma}(r\ u) & = & (\widehat{\sigma} r)\ (\widehat{\sigma} u) & \\
\widehat{\sigma}([\theta]\, p \to s) & = & [\theta]\, \widehat{\sigma} p \to \widehat{\sigma} s & \text{if } \theta \ \texttt{avoids}\ \sigma.
\end{array}$$

From now on, when talking about application of substitutions we usually mean substitutions of variables; otherwise the distinction will be made explicit or will be clear from the context and the notation.

The *application of a match* μ to a term is defined as follows: If μ is a substitution, then the application of the match to a term is obtained by applying the substitution to variables of the term as explained above. If μ is wait, then $\mu\ t$ is undefined. If μ is fail we define

$$\texttt{fail}\ t = [x]\,\widehat{x} \to x.$$

We will see in Section 3.2 how match failure provides a natural branching mechanism which can be used to underpin the definitions of conditionals and pattern-matching functions, much as in pure λ-calculus. However, other semantics for `fail` t are acceptable for pattern calculi in which branching is handled separately.

The *composition* $\sigma_2 \circ \sigma_1$ of two substitutions σ_1 and σ_2 is defined by $(\sigma_2 \circ \sigma_1)x = \sigma_2(\sigma_1 x)$. Further, if μ_1 and μ_2 are matches of which at least one is `fail`, then $\mu_2 \circ \mu_1$ is defined to be `fail`. Otherwise, if μ_1 and μ_2 are matches of which at least one is `wait`, then $\mu_2 \circ \mu_1$ is defined to be `wait`. Thus, in particular, `fail` $\circ$ `wait` is `fail`.

The *disjoint union* $\mu_1 \uplus \mu_2$ of *matches* μ_1 and μ_2 is defined as follows: If either of them is `fail` or their domains have a non-empty intersection, then their disjoint union is `fail`. Otherwise, if either of them is `wait`, then so is their disjoint union. Otherwise, it is the substitution given by

$$(\mu_1 \uplus \mu_2)x = \begin{cases} \mu_1 x & \textit{if } x \in \mathsf{dom}(\mu_1) \\ \mu_2 x & \textit{if } x \in \mathsf{dom}(\mu_2) \\ x & \textit{otherwise.} \end{cases}$$

Disjoint domains will be used to ensure that matching is deterministic.

The *check* μ_θ of a match μ on a set of symbols θ is `fail` if μ is a substitution whose domain is not θ and is μ otherwise. Thus, given a set θ, μ_θ is a decided match if and only if μ is. Checks will be used to ensure that variables do not escape their scope during reduction.

Lemma 2.2
If t is a term and μ is a decided match, then $\mathsf{fv}(\mu t) \subseteq \mathsf{fv}(\mu) \cup (\mathsf{fv}(t) \setminus \mathsf{dom}(\mu))$.

Proof
If μ is `fail`, then the result is immediate, so assume that μ is a substitution σ. The proof is by induction on the structure of t. If t is $[\theta]\, p \to s$, where θ `avoids` σ, then

$$\begin{array}{rcll} \mathsf{fv}([\theta]\, \sigma p \to \sigma s) & = & \mathsf{fv}(\sigma p) \cup (\mathsf{fv}(\sigma s)) \setminus \theta) & \\ & \subseteq & \mathsf{fv}(\sigma) \cup (\mathsf{fv}(p) \setminus \mathsf{dom}(\sigma)) \cup (\mathsf{fv}(\sigma) \setminus \theta) & \\ & & \quad \cup\ (\mathsf{fv}(s) \setminus \mathsf{dom}(\sigma) \setminus \theta) & \text{(by induction)} \\ & = & \mathsf{fv}(\sigma) \cup (\mathsf{fv}(p) \setminus \mathsf{dom}(\sigma)) \cup (\mathsf{fv}(s) \setminus \mathsf{dom}(\sigma) \setminus \theta) & \\ & = & \mathsf{fv}(\sigma) \cup (((\mathsf{fv}(s) \setminus \theta) \cup \mathsf{fv}(p)) \setminus\ \mathsf{dom}(\sigma)) & \\ & = & \mathsf{fv}(\sigma) \cup (\mathsf{fv}(t) \setminus \mathsf{dom}(\sigma)). & \end{array}$$

The other cases are straightforward. □

2.3 Reduction

Reduction is based on a (meta-)level operation $\{u/[\theta]\ p\}$ which *matches* the term p against the term u relative to a sequence of binding symbols θ. For the terms of the matchable binding grammar, if the matching operation yields a substitution σ, then it should satisfy $\mathsf{dom}(\sigma) = \theta$ and $\widehat{\sigma} p = u$. It is necessary to check that the domain of σ is θ to ensure that all binding variables get a value. For example, given the application $t = ([x]\ \widehat{y} \to x)\ \widehat{y}$, then even though the empty substitution maps $\widehat{y}$ to itself, t must not be allowed to reduce to x, since this would increase free variables.

$$\frac{}{([\theta]\, p \to s)\ u \twoheadrightarrow \{u/[\theta]\ p\}s}\ \text{(Start)} \qquad \frac{t \twoheadrightarrow t'}{C[t] \twoheadrightarrow C[t']}\ \text{(Ctx)}$$

Fig. 1. Reduction for context-free pattern calculi.

Now, *reduction* is driven by the rule

$$([\theta]\, p \to s)\ u \twoheadrightarrow \{u/[\theta]\ p\}s,$$

where its right-hand side must be a term resulting from the application of a *defined* match $\{u/[\theta]\ p\}$ to s.

Assuming that every binding symbol in θ is a free matchable of p it is obvious that there is at most one substitution σ such that $\widehat{\sigma}p = u$. In this case, a very simple approach is to define a *greedy matching* which assigns to $\{u/[\theta]\ p\}$ such a substitution, if it exists, and is `wait` otherwise. Thus in particular the greedy matching cannot fail. However, this will break confluence of reduction, as can be seen from the following example.

Example 2.1
Let $t = ([x,y]\, \widehat{x}\, \widehat{y} \to\ z)\ (([w]\, \widehat{w} \to\ w)\, f)$ and remark that $([w]\, \widehat{w} \to\ w)\, f = \{[w]\, \widehat{w} \to w/\widehat{x}, f/\widehat{y}\}\ (\widehat{x}\ \widehat{y})$. Assuming that matching is greedy, the following reduction steps from t lead to two different normal forms:

$$\begin{array}{lll} t & \twoheadrightarrow & ([x,y]\, \widehat{x}\, \widehat{y} \to\ z)\ f \\ t & \twoheadrightarrow & z. \end{array}$$

Fortunately, confluence can be recovered by restricting the match in various ways, as will be considered in Section 2.4.

For now, we assume that the matching operation is given but delay discussion of various definitions until Section 4, which discusses different pattern calculi. Hence, the next step is to consider the circumstances under which reduction is to be performed within terms. The main emphasis will be on *context-free* reduction, though Section 5 also considers pattern calculi in which reduction is *context-sensitive*.

Contexts (meta-variable C) are given by the following grammar:

$$C ::= \Box \mid C\ t \mid t\ C \mid [\theta]\, C \to t \mid [\theta]\, t \to C,$$

where $\Box$ is a distinguished constant.

The replacement of $\Box$ by a term t in a context C is written $C[t]$ and may provoke capture of symbols. Although $C[t]$ can always be generated by the matchable binding grammar defined in Section 2.1, this will not be the case for later grammars used to describe related work, as they impose restrictions on their patterns.

The ‘context-free reduction relation’ $\twoheadrightarrow$ given in Figure 1 is generated by the rule (Start) and closed by the rule (Ctx). In particular, if the match of the pattern against the argument produces a substitution, then apply this to the body. If the match is `fail`, then return the identity function. Of course, if the match is `wait` (e.g. because the pattern or argument needs to be evaluated), then the rule (Start) does not apply.

$$\frac{}{t \Rrightarrow t} \qquad \frac{r \Rrightarrow r' \quad u \Rrightarrow u'}{r\ u \Rrightarrow r'\ u'}$$

$$\frac{p \Rrightarrow p' \quad s \Rrightarrow s'}{[\theta]\, p \to s \Rrightarrow [\theta]\, p' \to s'} \qquad \frac{p \Rrightarrow p' \quad s \Rrightarrow s' \quad u \Rrightarrow u'}{([\theta]\, p \to s)\ u \Rrightarrow \{u'/[\theta]\ p'\}s'}$$

Fig. 2. Simultaneous reduction for context-free pattern calculi.

The relation $\rightarrow^*$ is the reflexive–transitive closure of $\rightarrow$. A term t is *irreducible* or in *normal form* if there is no reduction of the form $t \rightarrow t'$.

2.4 Confluence

This section presents a general proof of confluence for the context-free reduction relation $\rightarrow$, provided that matching satisfies the RMC, which imposes constraints upon matching that are satisfied by most confluent pattern calculi. Historically, one way to guarantee confluence is by requiring that patterns be rigid, as described by the RPC (van Oostrom, 1990; Klop *et al.*, 2008). However, this condition turns out to be too restrictive if one considers more expressive calculi such as the pure pattern calculus. The pure pattern calculus will take a different approach: any term may be a pattern, but matching of applications is restricted to avoid troublesome reductions, as in Example 2.1.

Confluence of reduction is here established using the technique due to Tait and Martin-Löf (Brendregt, 1984) which can be summarised in three steps: define a simultaneous reduction relation denoted $\Rrightarrow$; prove that $\Rrightarrow^*$ and $\rightarrow^*$ are the same relation (Lemma 2.3); prove that $\Rrightarrow$ has the diamond property (Lemma 2.4) and so is confluent; and infer that $\rightarrow$ is confluent.

The *simultaneous reduction relation* is given in Figure 2. It is a natural generalisation of simultaneous reduction relation for the λ-calculus. Exactly as in the definition of the reduction relation $\rightarrow$, the right-hand side of the last rule in Figure 2 needs to be a term given by the application of a *decided* math $\{u'/[\theta]\ p'\}$ to s'.

Lemma 2.3

The relations $\Rrightarrow^*$ and $\rightarrow^*$ are the same.

Proof

It is sufficient to prove $\rightarrow\ \subseteq\ \Rrightarrow\ \subseteq\ \rightarrow^*$. The inclusion $\rightarrow\ \subseteq\ \Rrightarrow$ trivially holds by reflexivity of $\Rrightarrow$. To show $\Rrightarrow\ \subseteq\ \rightarrow^*$ we reason by induction on $\Rrightarrow$. The interesting case is $([\theta]\, p \to s)\ u \Rrightarrow \{u'/[\theta]\ p'\}s'$, where $p \Rrightarrow p'$ and $s \Rrightarrow s'$ and $u \Rrightarrow u'$. By the induction hypothesis we have $p \rightarrow^* p'$ and $s \rightarrow^* s'$ and $u \rightarrow^* u'$ so that $([\theta]\, p \to s)\ u \rightarrow^* ([\theta]\, p' \to s')\ u' \rightarrow \{u'/[\theta]\ p'\}s'$. □

Given substitutions σ and σ' we write $\sigma \Rrightarrow \sigma'$ if and only if $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$ and $\sigma x \Rrightarrow \sigma' x$ for every $x \in \mathsf{dom}(\sigma)$. We extend this notion to matches only by $\mathtt{fail} \Rrightarrow \mathtt{fail}$ so that $\Rrightarrow$ does not hold if one of the matches is $\mathtt{wait}$.

To establish confluence, it is enough to show that matching has the following property:

Rigid matching condition (RMC): A matching algorithm $\{u/[\theta]\ p\}$ satisfies the *RMC* if for all simultaneous reductions $u \Rrightarrow u'$ and $p \Rrightarrow p'$ and $s \Rrightarrow s'$ of terms; if $\{u/[\theta]\ p\}$ is decided, then $\{u/[\theta]\ p\}s \Rrightarrow \{u'/[\theta]\ p'\}s'$.

Note that the RMC is not quite a consequence of confluence, since it involves simultaneous reduction in place of the context-free reduction relation.

Lemma 2.4
If the *RMC* holds, then the relation $\Rrightarrow$ has the diamond property. That is if $t \Rrightarrow t_1$ and $t \Rrightarrow t_2$, then there is a term t_3 such that $t_1 \Rrightarrow t_3$ and $t_2 \Rrightarrow t_3$.

Proof
The proof is by induction on the definition of simultaneous reduction. Suppose

$$([\theta]\, p_2 \to s_2)\ u_2 \ \Lleftarrow ([\theta]\, p \to s)\ u \Rrightarrow \{u_1/[\theta]\ p_1\}s_1,$$

where $p \Rrightarrow p_1$ and $p \Rrightarrow p_2$ and $s \Rrightarrow s_1$ and $s \Rrightarrow s_2$ and $u \Rrightarrow u_1$ and $u \Rrightarrow u_2$. By the induction hypothesis, there are terms p_3, s_3 and u_3 such that $p_1 \Rrightarrow p_3$ and $p_2 \Rrightarrow p_3$ and $s_1 \Rrightarrow s_3$ and $s_2 \Rrightarrow s_3$ and $u_1 \Rrightarrow u_3$ and $u_2 \Rrightarrow u_3$. Now, we close the diagram using $([\theta]\, p_2 \to s_2)u_2 \Rrightarrow \{u_3/[\theta]\ p_3\}s_3$, which holds by definition, and $\{u_1/[\theta]\ p_1\}s_1 \Rrightarrow \{u_3/[\theta]\ p_3\}s_3$, which holds by the RMC.

The other cases are straightforward. □

Theorem 2.5
If the *RMC* holds, then the reduction relation $\rightarrow$ is confluent.

Proof
The reduction relation $\Rrightarrow$ has the diamond property by Lemma 2.4, so that $\Rrightarrow$ is confluent. We conclude, since $\Rrightarrow^* = \rightarrow^*$ by Lemma 2.3. □

Proving the RMC is a bit convoluted, as it involves six terms. However, it is a consequence of two simpler properties P1 and P2 abstracted from lemmas appearing in the confluence proof of the pure pattern calculus (Jay & Kesner, 2006a; see Section 3).

Property 1 (P1): If θ $\mathtt{avoids}$ σ and $\{u/[\theta]\ p\}$ is decided, then $\{\sigma u/[\theta]\ \sigma p\}$ is decided and equal to $(\sigma \circ \{u/[\theta]\ p\})|_\theta$.

Property 2 (P2): If $u \Rrightarrow u'$ and $p \Rrightarrow p'$ and $\{u/[\theta]\ p\}$ is decided, then $\{u'/[\theta]\ p'\}$ is decided and $\{u/[\theta]\ p\} \Rrightarrow \{u'/[\theta]\ p'\}$.

The first property asserts that match generation and substitution commute. The second property asserts that match generation and simultaneous reduction commute.

Lemma 2.6
Assume that *P1* holds. Let σ be a substitution, and let θ be a sequence of symbols such that θ `avoids` σ. If p and u are terms such that $\{u/[\theta]\ p\}$ is decided, then so is $\{\sigma u/[\theta]\ \sigma p\}$ and $\{\sigma u/[\theta]\ \sigma p\} \circ \sigma = \sigma \circ \{u/[\theta]\ p\}$.

Proof
If $\{u/[\theta]\ p\} = \texttt{fail}$, then $\{\sigma u/[\theta]\ \sigma p\} = \texttt{fail}$ by *P1*, and the result follows. Thus, without loss of generality, assume that $\{u/[\theta]\ p\}$ is a substitution σ' satisfying $\mathsf{dom}(\sigma') = \theta$.

If $x \in \theta$, then

$$\begin{array}{rcll} (\{\sigma u/[\theta]\ \sigma p\} \circ \sigma)(x) & = & ((\sigma \circ \sigma')|_\theta \circ \sigma)(x) & \text{(by } P1\text{)} \\ & = & (\sigma \circ \sigma')|_\theta(x) & \text{(by } \theta \ \texttt{avoids}\ \sigma\text{)} \\ & = & (\sigma(\sigma'(x)) & \\ & = & (\sigma \circ \sigma')(x). & \end{array}$$

If $x \notin \theta$, then

$$\begin{array}{rcll} (\{\sigma u/[\theta]\ \sigma p\} \circ \sigma)(x) & = & \sigma(x) & (\text{by } \mathsf{dom}(\{\sigma u/[\theta]\ \sigma p\}) = \theta \ \texttt{avoids}\ \sigma) \\ & = & (\sigma \circ \sigma')(x) & (\text{by } \mathsf{dom}(\sigma') = \theta). \end{array}$$

□

Lemma 2.7
Assume that *P1* holds. If $\mu \Rrightarrow \mu'$ are matches and $t \Rrightarrow t'$ are terms, then $\mu t \Rrightarrow \mu' t'$.

Proof
If μ is `fail`, then μ' is `fail`, and the result is immediate. So assume that μ and μ' are substitutions σ and σ' respectively. The proof is by induction on the derivation of $t \Rrightarrow t'$. The only non-trivial case is when $t = ([\theta]\, p \to s)\ u \Rrightarrow \{u'/[\theta]\ p'\}s' = t'$, where $p \Rrightarrow p'$ and $u \Rrightarrow u'$ and $s \Rrightarrow s'$. Without loss of generality, assume $\mathsf{sym}(\sigma) \cap \theta = \{\}$ and $\mathsf{sym}(\sigma') \cap \theta = \{\}$. By Lemma 2.6 $\sigma'(\{u'/[\theta]\ p'\}s')$ is equal to $\{\sigma' u'/[\theta]\ \sigma' p'\}(\sigma'\ s')$. By the induction hypothesis we have $\sigma p \Rrightarrow \sigma' p'$, $\sigma u \Rrightarrow \sigma' u'$ and $\sigma s \Rrightarrow \sigma' s'$. Then, we conclude $\sigma t = ([\theta]\, \sigma\ p \to \sigma s)\ (\sigma\ u) \Rrightarrow \{\sigma' u'/[\theta]\ \sigma' p'\}(\sigma'\ s') = \sigma' t'$. □

Theorem 2.8
P1 and *P2* imply the *RMC*.

Proof
Suppose $u \Rrightarrow u'$ and $p \Rrightarrow p'$ and $s \Rrightarrow s'$ and $\{u/[\theta]\ p\}$ is decided. P2 gives $\{u/[\theta]\ p\} \Rrightarrow \{u'/[\theta]\ p'\}$ so that P2 and Lemma 2.7 give $\{u/[\theta]\ p\}s \Rrightarrow \{u'/[\theta]\ p'\}s'$ as desired. □

Corollary 2.9
A pattern calculus expressed in the general framework of Sections 2.1 and 2.2 that satisfies *P1* and *P2* is confluent.

Another way of satisfying the RMC is to allow matching to be as generous as possible while restricting the patterns. This is achieved using the following condition, adapted from that of van Oostrom (1990; see also Section 4.3):

Closed rigid pattern condition (CRPC): A term p is said to satisfy the CRPC if it is closed, and for all substitutions σ_1 and terms q, if $\widehat{\sigma_1} p \Rrightarrow q$, then $q = \widehat{\sigma_2} p$ for some σ_2 such that $\sigma_1 \Rrightarrow \sigma_2$. A term t is said to be *rigid* if and only if all its patterns satisfy the CRPC.

The following theorem generalises the one in Cirstea and Faure (2007) to context-free calculi expressed within the matchable binding grammar.

Theorem 2.10
Reduction of rigid terms using greedy matching satisfies *P1* and *P2*.

Proof
First of all note that the use of the greedy matching implies that matches can only be decided substitutions or `wait`. For P1, suppose that θ avoids σ and $\{u/[\theta]\ p\}$ is decided, so that it is a substitution σ_1 such that $\widehat{\sigma_1} p = u$. Let $\sigma_2 = (\sigma \circ \{u/[\theta]\ p\})|_\theta$. Now

$$\begin{array}{rcll} \widehat{\sigma_2}(\sigma p) & = & \widehat{\sigma_2} p & \text{(since rigidity implies } p \text{ is closed)} \\ & = & \sigma(\widehat{\sigma_1}\ p) & \\ & = & \sigma u & \text{(by definition of matching).} \end{array}$$

Hence, since matching is greedy, $\sigma_2 = \{\sigma u/[\theta]\ \sigma p\}$ as required.

For P2, suppose that $u \Rrightarrow u'$ and $p \Rrightarrow p'$ and $\{u/[\theta]\ p\}$ is decided, and so is some substitution σ_1. Then by rigidity, $p = \{\}\ p \Rrightarrow p'$ implies $p' = \sigma' p$, where $\{\}\ \Rrightarrow \sigma'$ so that $\sigma' = \{\}$ and $p' = p$. Now $\widehat{\sigma_1} p = u \Rrightarrow u'$, and so the CRPC implies that $u' = \widehat{\sigma_2} p$ for some σ_2 such that $\sigma_1 \Rrightarrow \sigma_2$. Hence $\sigma_1 = \{u/[\theta]\ p\} \Rrightarrow \{u'/[\theta]\ p\} = \sigma_2$ as required.
□

Now Theorems 2.8 and 2.5 yield the following corollary.

Corollary 2.11
Reduction of rigid terms using greedy matching is confluent.

Unfortunately, as explained in Section 4.3, the interesting examples of path and pattern polymorphisms are not rigid. The following section will show another approach, in which the RMC is satisfied by restricting matching, not the patterns.

3 Pure pattern calculus

This section considers the *pure pattern calculus with matchable symbols*, notation which is introduced in (Jay, 2009) to define context-free reduction relations on terms with dynamic patterns. Two of our earlier versions will be considered in Section 5, namely *the original pure pattern calculus* and *the context-sensitive pattern calculus.* We will show the pure pattern calculus with matchable symbols to be equivalent to the context-sensitive pattern calculus in Section 5.2.

3.1 Data structures

Having established the general syntax which allows any term to be a pattern, let us return to the challenge of matching them and, in particular, to the challenge of matching applications.

Confluence will be broken if the following reductions are allowed:

$$\begin{array}{l}([x,y]\,\widehat{x}\;\widehat{y} \to\; y)\;(([w]\,\widehat{w} \to\; \widehat{z_1}\;\widehat{z_2})\;\widehat{z_1}) \boldsymbol{\rightarrow} ([x,y]\,\widehat{x}\;\widehat{y} \to\; y)\;(\widehat{z_1}\;\widehat{z_2}) \boldsymbol{\rightarrow} \widehat{z_2}\\ ([x,y]\,\widehat{x}\;\widehat{y} \to\; y)\;(([w]\,\widehat{w} \to\; \widehat{z_1}\;\widehat{z_2})\;\widehat{z_1}) \boldsymbol{\rightarrow} \widehat{z_1}.\end{array}$$

The problem arises if the pattern $\widehat{x}\;\widehat{y}$ is able to match with applications that may still be reduced. The problem could be handled by requiring irreducibility of arguments, but then the embedding of λ-calculus would not preserve arbitrary reduction. Rather, it is enough to require that arguments be sufficiently reduced to allow matching to be decided: either the pattern is a binding symbol or the argument is a *matchable form*, as defined below. As similar problems may arise when the pattern is reducible, both the pattern and the arguments must be matchable forms.

Data structures (meta-variable d) and *matchable forms* (meta-variable m) are defined via the grammar

$$\begin{array}{rcl} d & ::= & \widehat{x} \mid d\;t \\ m & ::= & d \mid [\theta]\,p \to s. \end{array}$$

An application $d\;t$, which is a data structure, is a *compound*. All other matchable forms, i.e. the matchable symbols and cases, are *atoms*. Note that the application $\widehat{x}\;u$ is a compound no matter whether $\widehat{x}$ is to be a constructor or a binding symbol. Also, data structure may contain arbitrary terms as arguments. In particular, it will not be necessary to reduce data structures to normal form before matching against them.

The *matching* $\{u/[\theta]\;p\}$ of a term p against a term u relative to a sequence of binding symbols θ is now obtained by the check on θ of the *compound matching* $\{\!\{u \triangleright [\theta]\;p\}\!\}$ which is defined by applying the following equations in order:

$$\begin{array}{lcll}
\{\!\{u \triangleright [\theta]\;\widehat{x}\}\!\} & = & \{u/x\} & \text{if } x \in \theta \\
\{\!\{\widehat{x} \triangleright [\theta]\;\widehat{x}\}\!\} & = & \{\} & \text{if } x \notin \theta \\
\{\!\{u\;v \triangleright [\theta]\;p\;q\}\!\} & = & \{\!\{u \triangleright [\theta]\;p\}\!\} \uplus \{\!\{v \triangleright [\theta]\;q\}\!\} & \text{if } u\;v \text{ and } p\;q \text{ are} \\
 & & & \quad\text{matchable forms} \\
\{\!\{u \triangleright [\theta]\;p\}\!\} & = & \texttt{fail} \quad \text{otherwise} & \text{if } u \text{ and } p \text{ are} \\
 & & & \quad\text{matchable forms} \\
\{\!\{u \triangleright [\theta]\;p\}\!\} & = & \texttt{wait} \quad \text{otherwise.} &
\end{array}$$

The main point is that applications can be matched only if they are both compounds, so that matching is stable under reduction. The use of disjoint unions when matching data structures means that matching against a compound such as $\widehat{z}\;\widehat{x}\;\widehat{x}$ can never succeed. Indeed, it would be enough to use a pattern of the form $\widehat{z}\;\widehat{x}\;\widehat{y}$ and check equality of x and y afterwards. The restriction to linear patterns is also common when establishing confluence of rewriting systems (Klop, 1980) or of pattern calculi based on rewriting (Forest & Kesner, 2003; Kahl, 2004), but this is related to orthogonality of pairs of rules or cases, while our framework only considers one case at a time.

The two last equations of compound matching yield `fail` or `wait`. Definite failure arises when both pattern and argument are matchable, and none of the earlier equations for successful matching applies. Otherwise matching must wait. As defined, matching one case against another always fails. Successful case matching is not necessary for the path and pattern polymorphic examples that motivated this work, though it is supported by some other calculi discussed later. Note that the ordering of the equations can be avoided by expanding the definition into an induction on the structure of the pattern.

The resulting *context-free reduction relation for the pure pattern calculus* is written $\rightarrow_{PPC}$.

Theorem 3.1
The reduction relation $\rightarrow_{PPC}$ is confluent.

Proof
Properties P1 and P2 can be shown to hold for the pure pattern calculus, by induction on the structure of the patterns involved. Now apply Corollary 2.9. □

3.2 Examples

This section presents examples of terms in the pure pattern calculus (augmented with wild cards in Example 3.10). As well as conveying the general flavour of the approach, they will serve to illustrate the path and pattern polymorphisms of the pure pattern calculus and provide benchmarks for later comparison with other pattern calculi. Names starting with capital letters such as Nil and Pair always represent constructors. In the current setting, using the matchable binding grammar, they are free matchable symbols: in later grammars they may be members of a separate alphabet of constructors.

Example 3.1 (*λ-calculus*)
There is a simple embedding of the pure λ-calculus into the pure pattern calculus obtained by identifying the λ-abstraction $\lambda x.s$ with $[x]\,\widehat{x} \rightarrow s$. Pattern matching for these terms will be exactly the β-reduction of the λ-calculus. For example, the *fixpoint* term

$$\mathsf{fix} = ([x]\,\widehat{x} \rightarrow [f]\,\widehat{f} \rightarrow f\ (x\ x\ f))\ ([x]\,\widehat{x} \rightarrow [f]\,\widehat{f} \rightarrow f\ (x\ x\ f))$$

can be used to define recursive functions.

Example 3.2 (*branching constructs*)
Let True and False be constructors and define conditionals by

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ r = ([\,]\,\mathsf{True} \rightarrow [x]\,\widehat{x} \rightarrow s)\ b\ r,$$

where $x \notin \mathsf{fv}(s)$. Thus, if True then s else r reduces to $([x]\,\widehat{x} \rightarrow s)\ r$ and then to s, while if False then s else r reduces to $([y]\,\widehat{y} \rightarrow y)\ r$ and then to r. Note how the interpretation of `fail` $([x]\,\widehat{x} \rightarrow s)$ as the identity term contributes here.

More generally, define the *extension* of a default r by a special case $[\theta]\,p \rightarrow s$ by

$$[\theta]\,p \rightarrow s \mid r = [x]\,\widehat{x} \rightarrow ([\theta]\,p \rightarrow [y]\,\widehat{y} \rightarrow s)\ x\ (r\ x),$$

where $x \notin \mathsf{fv}([\theta]\,p \rightarrow s) \cup \mathsf{fv}(r)$ and $y \notin \mathsf{fv}(s)$. When applied to some term u it reduces to $\{\!\{u \rhd [\theta]\ p\}\!\}([y]\,\widehat{y} \rightarrow s)\ (r\ u)$. Now if $\{\!\{u \rhd [\theta]\ p\}\!\}$ is some substitution σ, then this reduces to $(\sigma([y]\,\widehat{y} \rightarrow s))\ (r\ u) = ([y]\,\widehat{y} \rightarrow \sigma s)\ (r\ u)$ and then to σs as desired. Alternatively, if $\{\!\{u \triangleright [\theta]\ p\}\!\} = \texttt{fail}$, then the term reduces to $(\texttt{fail}\ ([y]\,\widehat{y} \rightarrow s))\ (r\ u) = ([z]\,\widehat{z} \rightarrow z)\ (r\ u)$ and then to $r\ u$ as desired.

Extensions can be iterated to produce pattern-matching functions out of a sequence of many cases. Make | right-associative so that

$$\begin{array}{l} \ \ [\theta_1]\,p_1 \rightarrow s_1 \\ |\ [\theta_2]\,p_2 \rightarrow s_2 \\ \quad\vdots \\ |\ [\theta_n]\,p_n \rightarrow s_n \end{array}$$

is $[\theta_1]\,p_1 \rightarrow s_1 \mid ([\theta_2]\,p_2 \rightarrow s_2 \mid (\ldots \mid [\theta_n]\,p_n \rightarrow s_n))$.

Example 3.3 (*constructors*)
It is common to add to the λ-calculus a collection γ of term constants to play the role of constructors for data structures. Here we can define a *program* to consist of a closed term p whose free matchables (constructors) γ play the role of constructors. One then defines a program by the expression

$$[\gamma]\,p \rightarrow \bullet,$$

where $\bullet$ is some closed irreducible term, say $[x]\,\widehat{x} \rightarrow x$.

Example 3.4 (*structural induction*)
Recursive functions were informally introduced in Section 1 by equations of the form $F = \{F/f\}t$, where $F \notin \mathsf{fv}(t)$. Of course, these can be defined using fix by setting $F = \mathsf{fix}\ ([f]\,\widehat{f} \rightarrow t)$, so that F reduces to $\{F/f\}t$.

The natural numbers can be defined as data structures built from constructors Zero and $\mathsf{Successor}$, and addition can be defined by

$$\begin{array}{l} \mathsf{plusNat} = \\ \quad\ [\,]\ \ \mathsf{Zero} \qquad\quad \rightarrow [y]\ \widehat{y} \rightarrow y \\ \ \ \mid [x]\ \mathsf{Successor}\ \widehat{x} \rightarrow [y]\ \widehat{y} \rightarrow \mathsf{Successor}\ (\mathsf{plusNat}\ x\ y). \end{array}$$

The lists can be defined as data structures built from constructors Nil and Cons. Then the length of a list is given by

$$\begin{array}{l} \mathsf{length} = \\ \quad\ [\,]\quad\ \ \mathsf{Nil} \qquad\ \rightarrow \mathsf{Zero} \\ \ \ \mid [x,y]\ \mathsf{Cons}\ \widehat{x}\ \widehat{y} \rightarrow \mathsf{Successor}\ (\mathsf{length}\ y). \end{array}$$

For example, $\mathsf{length}\ (\mathsf{Cons}\ u\ \mathsf{Nil})$ first computes $\{\mathsf{Cons}\ u\ \mathsf{Nil}/[\,]\ \mathsf{Nil}\}$ which is $\texttt{fail}$ and so computes $\{\mathsf{Cons}\ u\ \mathsf{Nil}/[x,y]\ \mathsf{Cons}\ \widehat{x}\ \widehat{y}\}$ which is $\{u/x, \mathsf{Nil}/y\}$. More interestingly, let $u = ([x]\,\widehat{x} \rightarrow \mathsf{Nil})\ v$ for some term v, and consider $\mathsf{length}\ u$. If matching is attempted immediately, then $\{u/[\,]\ \mathsf{Nil}\}$ is $\texttt{wait}$ so that reduction does not occur. Rather, u must first be reduced to Nil at which point the matching succeeds.

Other common list functions are

$$\mathsf{singleton} = [x]\,\widehat{x} \rightarrow \mathsf{Cons}\ x\ \mathsf{Nil}$$

for creating singleton lists and

$$
\begin{array}{lll}
\mathsf{append} & = & \\
\quad \mathsf{Nil} & \rightarrow [y]\ \widehat{y} \rightarrow y & \\
\mid \mathsf{Cons}\ \widehat{x_1}\ \widehat{x_2} & \rightarrow [y]\ \widehat{y} \rightarrow \mathsf{Cons}\ x_1\ (\mathsf{append}\ x_2\ y) &
\end{array}
$$

for appending lists.

The function which specifies an operation computing all the suffix lists of a list is given by

$$
\begin{array}{llll}
\mathsf{suffixlist} = & & & \\
\quad [\,] & \mathsf{Nil} & \rightarrow & \mathsf{Cons\ Nil\ Nil} \\
\mid [x, y] & \mathsf{Cons}\ \widehat{x}\ \widehat{y} & \rightarrow & \mathsf{Cons}\ (\mathsf{Cons}\ x\ y)\ (\mathsf{suffixlist}\ y.
\end{array}
$$

Structural induction can be also used generically on arbitrary data structures by means of a path polymorphic function. Thus, for example, we can define a function that counts the number of atoms of an arbitrary data structure, using

$$
\begin{array}{lll}
\mathsf{size} & = & \\
\quad [y, z]\ \widehat{y}\ \widehat{z} & \rightarrow & \mathsf{plusNat}\ (\mathsf{size}\ y)\ (\mathsf{size}\ z) \\
\mid [x] \quad \widehat{x} & \rightarrow & \mathsf{Successor\ Zero}.
\end{array}
$$

Patterns of the form $\widehat{y}\ \widehat{z}$ are used to access data along arbitrary paths through a data structure, i.e. to support *path polymorphism.* The pattern $\widehat{x}$ denotes any other possible argument which is not a compound.

Example 3.5 (*update of arbitrary data structures*)
A typical example of path polymorphism can be given by a function that updates point data within an arbitrary data structure. Let Point be some constructor. Then define updatePoint by

$$
\begin{array}{lll}
\mathsf{updatePoint} & = & [f]\ \widehat{f} \rightarrow \\
\quad [w] \quad \mathsf{Point}\ \widehat{w} & \rightarrow & \mathsf{Point}\ (f\ w) \\
\mid [y, z]\ \widehat{y}\ \widehat{z} & \rightarrow & (\mathsf{updatePoint}\ f\ y)\ (\mathsf{updatePoint}\ f\ z) \\
\mid [x] \quad \widehat{x} & \rightarrow & x.
\end{array}
$$

The pattern Point $\widehat{w}$ denotes a data structure headed by the free matchable Point which is playing the role of a constructor, since Point is not in the binding set $[w]$. This function turns to be an application of the forthcoming pattern polymorphic function update (Example 3.9) to the constructor Point.

In the same style we can define a path polymorphic function that applies the same transformation f to *every* component of an arbitrary data structure:

$$
\begin{array}{lll}
\mathsf{apply2all} & = & [f]\ \widehat{f} \rightarrow \\
\quad [y, z]\ \widehat{y}\ \widehat{z} & \rightarrow & f\ ((\mathsf{apply2all}\ f\ y)\ (\mathsf{apply2all}\ f\ z)) \\
\mid [x] \quad \widehat{x} & \rightarrow & f\ x.
\end{array}
$$

Example 3.6 (*selecting components of arbitrary data structure*)
Another typical example of path polymorphism can be given by a function that selects the components of an arbitrary data structure satisfying some property. This

can be given by

$$\begin{array}{rcl} \mathsf{select} & = & [f]\,\widehat{f} \rightarrow \\ [y,z]\ \widehat{y}\ \widehat{z} & \rightarrow & \mathsf{if}\ (f\ (y\ z)) \\ & & \mathsf{then\ singleton}\ (y\ z) \\ & & \mathsf{else\ append}\ (\mathsf{select}\ f\ y)\ (\mathsf{select}\ f\ z) \\ \mid [x]\quad \widehat{x} & \rightarrow & \mathsf{if}\ (f\ x)\ \mathsf{then}\ (\mathsf{singleton}\ x)\ \mathsf{else\ Nil}. \end{array}$$

Note that this function does not find components of components that satisfy the property, though this could be also be specified.

Example 3.7 (*generic equality*)
A typical example of pattern polymorphism is the generic equality defined by

$$\mathsf{equal} = [x]\,\widehat{x} \rightarrow \quad \begin{array}{l} (\quad [\,]\ \ x \rightarrow \mathsf{True} \\ \quad\mid [y]\ \widehat{y} \rightarrow \mathsf{False}), \end{array}$$

where the first argument is used as the pattern for matching against the second. For example, equal (Successor Zero) (Successor Zero) reduces to True. Note that the function equal yields False when applied to identical abstractions and so is not expressive enough to support Klop's counterexemple to confluence of pattern calculi, where equality is used on arbitrary terms.

Example 3.8 (*the generic eliminator*)
The generic eliminator is another typical example of pattern polymorphic function given by

$$\mathsf{elim} = [x]\,\widehat{x} \rightarrow ([y]\,x\ \widehat{y} \rightarrow y).$$

For example, elim Successor reduces to $[y]$ Successor $\widehat{y} \rightarrow y$, and elim singleton reduces to $[y]$ Cons $\widehat{y}$ Nil $\rightarrow y$ by reduction of the pattern singleton $\widehat{y}$.

Example 3.9 (*generic updating*)
Combining the use of pattern and path polymorphism yields the generic update function

$$\begin{array}{lll} \mathsf{update} = & & [x]\ \widehat{x} \rightarrow [f]\ \widehat{f} \rightarrow \\ \quad [w] & x\ \widehat{w} & \rightarrow x\ (f\ w) \\ \mid [y,z] & \widehat{y}\ \widehat{z} & \rightarrow (\mathsf{update}\ x\ f\ y)\ (\mathsf{update}\ x\ f\ z) \\ \mid [g] & \widehat{g} & \rightarrow g. \end{array}$$

When applied to a constructor c, a function f and a data structure d it replaces sub-terms of d of the form $c\ t$ by $c\ (f\ t)$. For example, update $c\ f\ ((c\ u)\ (c\ v))$ reduces to $(c\ (f\ u))\ (c\ (f\ v))$, and update Point reduces to updatePoint (Example 3.5). Also, update singleton f reduces to a pattern-matching function whose first case is

$$[w]\ \mathsf{Cons}\ \widehat{w}\ \mathsf{Nil}\ \rightarrow \mathsf{Cons}\ (f\ w)\ \mathsf{Nil}.$$

Also, updating can be iterated to give finer control. For example, given the constructors Salary, Employee and Department and a function f, the program

$$\mathsf{update\ Department\ (update\ Employee\ (update\ Salary}\ f))$$

updates departmental employee salaries. Note that it is not necessary to know how employees are represented within departments for this to work, so that a new level of abstraction arises, similar to that which XML is intended to support. The full range of XML paths can be handled by defining an appropriate abstract data type, similar to that of *signposts* given in (Huang *et al*., 2006a, 2006b).

Example 3.10 (*wild cards*)
It is interesting to add a new constant denoted? to the matchable binding grammar, the *wild card*. It has no free variable or matchable symbol and is unaffected by substitution. It is a data structure, is compatible with anything and has the matching rule

$$\{\!\{u \triangleright [\theta]\ ?\}\!\} = \{\}$$

for any θ and u. That is it behaves like a fresh binding variable in a pattern but like a constructor in a body. The matching algorithm $\{u/[\theta]\ p\}$ generated by adding the new rule for wild cards to those of the pure pattern calculus satisfies the *RMC*, even if for some pattern p containing wild cards, $\widehat{\sigma} p \neq u$, when $\{u/[\theta]\ p\} = \sigma$.

For example, the second and first projections from a pair (built using a constructor Pair) can be encoded as $\mathsf{elim}\ (\mathsf{Pair}\ ?)$ and $\mathsf{elim}\ ([x]\,\widehat{x} \to \mathsf{Pair}\ x\ ?)$.

The following example uses recursion in the pattern. Define the function for the extracting list entries by

$$\begin{array}{lll}
\mathsf{let\ entrypattern} & = & \\
\quad [z]\ \mathsf{Succ}\ \widehat{z} & \to [x]\,\widehat{x} \to \mathsf{Cons}\ ?\ (\mathsf{entrypattern}\ z\ x) & \\
\mid\ [\,]\ \ \mathsf{Zero} & \to [x]\,\widehat{x} \to \mathsf{Cons}\ x\ ? & \\
 & & \\
\mathsf{entry} = & [z]\,\widehat{z} \to \mathsf{elim}\ (\mathsf{entrypattern}\ z). &
\end{array}$$

For example, $\mathsf{entry}\ (\mathsf{Succ}\ (\mathsf{Succ}\ \mathsf{Zero}))$ reduces (in many steps) to the function $[y]\,\mathsf{Cons}\ ?\ (\mathsf{Cons}\ ?\ (\mathsf{Cons}\ \widehat{y}\ ?)) \to y$ which recovers the third entry from a list. Note that standard approaches cannot support such examples, since their wild cards are not first-class terms (see Section 4.2).

4 Closed patterns

The next task is to consider the relationship between the pure pattern calculus with matchable symbols and other pattern calculi in the literature, including earlier versions of the pure pattern calculus.

Before continuing, note that the comparisons will focus on the treatment of a single case, rather than how cases are combined into pattern-matching functions such as those in Section 3.2. At least three techniques have been used in the literature.

One is to create *sets* of cases, such as $\{p_i \to s_i\}$ whose application to a term u can reduce to $\{u/p_i\}s_i$ if the latter term is defined. This is a natural approach if one is using pattern matching to combine λ-calculus and rewriting, as in CRS (Klop, 1980). To achieve confluence in this setting requires additional restrictions, such as *orthogonality* of the patterns, of a kind familiar from rewriting theory (Terese, 2003).

A second technique is to create a *list* of cases, $[p_i \to s_i]$ whose application to a term u reduces to $\{u/p_i\}s_i$, where p_i is the *first* pattern to match against u. For this to work, it is necessary to formalise the notion of *match failure*, so that one may pass over the cases whose patterns cannot match as in Kahl (2004).

The third technique is a variation of the second. Now the list of cases is represented by a single (but nested) case, so that there is no need to add new term forms. This approach is adopted by all variants of the pure pattern calculus. This technique has the benefit of keeping the term grammar compact, and as in the second technique, it does not introduce more orthogonality issues.

This section considers calculi in which binding is *implicit*, in the sense that all free variables of patterns are bound in the enclosing case. Since substitutions cannot affect patterns within cases, these are *closed patterns*.

When binding is implicit, the notation can be much lighter than in the matchable binding grammar in Section 2.1. Lightest of all is to identify the patterns and terms, as contemplated in Section 4.3. However, some care is still required. In general, reduction may eliminate free variables, but if pattern reduction loses a free variable, then it may lose a binding too. Hence, it is necessary to restrict the patterns, or their reduction, to avoid loss of bindings.

The usual way of doing this is to describe a separate class of patterns guaranteeing stability of bindings. It is also usual to add an alphabet of *constructors*. The resulting *implicit binding grammar* then requires four syntactic classes, of symbols, constructors (meta-variable c), patterns (meta-variables p, q) and terms (meta-variables r, s, t, u, v):

$$\begin{array}{lcl} p & := & x \mid c \mid p\ p \mid \dots \\ t & ::= & x \mid c \mid t\ t \mid p \to t \mid \dots. \end{array}$$

The syntactic machinery is as expected, on the understanding that the free variables of a case $p \to s$ are given by those of s that are not free in p. Without free variables inside patterns, such cases cannot be used to express pattern polymorphism.

In each calculus, matching of a pattern p against a term u will be described by a match $\{\!\{u \triangleright p\}\!\}$. There is no need to specify or check the binding symbols, since they are exactly the free variables of the pattern, and so they are all in the domain of the generated substitution. Hence the rule (Start) of Figure 1 becomes

$$\text{(Start)} \qquad (p \to s)\ u \rightarrow \{\!\{u \triangleright p\}\!\}s.$$

The translation from this syntax to the one of Section 2.1 is straightforward: a case $p \to s$ translates to $[\theta]\,\widehat{p} \to s$, where θ is a sequence containing the free variables of p in some order determined by p and $\widehat{p}$ replaces each free variable x of p by $\widehat{x}$. Constructors are translated to fresh free matchable symbols. Note that each pattern is translated to a closed term. Conversely, when importing concepts from the matchable binding grammar to the implicit binding grammar, free matchable symbols will typically be replaced by constructors in the definitions and equations.

In many calculi, patterns are irreducible by definition, and so one may restrict consideration to the *restricted contexts* (meta-variable B) given by the grammar

$$B ::= \square \mid B\ t \mid t\ B \mid p \to B,$$

where p is a meta-variable ranging over the set of patterns of the language under consideration.

This section considers various examples of pattern calculi with implicit bindings. Section 4.1 focuses on algebraic patterns as used in first-order term rewriting systems (Baader & Nipkow, 1998). Section 4.2 introduces more sophisticated syntactic patterns which appear for example in well-known functional languages. Section 4.3 presents the λ-calculus with patterns, which considers matching on abstractions. Section 4.4 introduces a simple language able to model path polymorphism. All of the calculi mentioned above enjoy P1 and P2 of Section 2.4 and so are confluent.

4.1 Algebraic patterns

Algebraic pattern calculi (e.g. Peyton Jones, 1987; Kahl, 2004) can be understood as calculi containing a minimal form of pattern matching in which non-trivial patterns are headed by a constructor. That is patterns (p) are built from variables (for binding) and constructors, using the grammar

$$\begin{array}{lcl} p & ::= & x \mid d \\ d & ::= & c \mid d\ p \text{ if } \mathsf{fv}(d) \cap \mathsf{fv}(p) = \{\}. \end{array}$$

The side condition guarantees linearity of patterns. Note that patterns are in *normal form*, as they cannot be reduced. The equations for *algebraic matching* are

$$\begin{array}{lcl} \{\!\{u \triangleright x\}\!\} & = & \{u/x\} \\ \{\!\{c \triangleright c\}\!\} & = & \{\} \\ \{\!\{u\ v \triangleright p\ q\}\!\} & = & \{\!\{u \triangleright p\}\!\} \cup \{\!\{v \triangleright q\}\!\}. \end{array}$$

This matching can be viewed as the restriction of the compound matching to algebraic patterns, on the understanding that the free matchable symbols become constructors. Since reduction cannot take place inside algebraic patterns, which are already in normal form, it is sufficient to consider restricted contexts.

A typical example which can be expressed in this framework is the length function (Example 3.4).

It is well known that if all the patterns appearing in cases are linear, then confluence holds.

Theorem 4.1
The algebraic calculus is confluent.

Proof
It is sufficient to verify P1 and P2. This can be done using the definition of algebraic matching and reasoning by induction on patterns. □

4.2 Pattern operations

When focusing on the convenience of programming, it is natural to add some operations (meta-variable o) on patterns (SML, `http://www.smlnj.org/`; Haskell, `http://www.haskell.org/`; OCaml, `http://caml.inria.fr/`) so that patterns are no longer a special case of terms. Now the patterns are given by

$$\begin{array}{lcl} p & ::= & x \mid q \\ q & ::= & c \mid o \mid q\ p \text{ if } \mathsf{fv}(q) \cap \mathsf{fv}(p) = \{\}. \end{array}$$

As in Section 4.1, the side condition guarantees linearity. Binding variables and contexts are just as for the algebraic case. Note also that patterns are still in normal form, but matching is now given by augmenting the algebraic matching algorithm to deal with the pattern operators. We will consider three examples here: the *wild card* ? which can match anything; as which allows two patterns to match a single argument; and # which matches an arbitrary case. Their matching is given by

$$\begin{array}{rcl} \{\!\{u \rhd ?\}\!\} & = & \{\} \\ \{\!\{u \rhd \mathsf{as}\ p\ q\}\!\} & = & \{\!\{u \rhd p\}\!\} \uplus \{\!\{u \rhd q\}\!\} \\ \{\!\{p \to s \rhd \#x\}\!\} & = & \{p \to s/x\}. \end{array}$$

The reduction relation is generated exactly as in Section 4.1. As in the previous section, confluence follows from P1 and P2.

For example, pattern operators can simplify the presentation of the function `suffixlist` in Example 3.4 to get

$$\begin{array}{lcl} \mathsf{suffixlist} = & & \\ \quad \mathsf{Nil} & \to & \mathsf{Cons\ Nil\ Nil} \\ \mid\ \mathsf{as}\ x\ (\mathsf{Cons}\ ?\ y) & \to & \mathsf{Cons}\ x\ (\mathsf{suffixlist}\ y). \end{array}$$

In general, pattern operators may increase the expressive power of the calculus, but these three examples do not: wild cards can be represented by fresh (binding) variables; matching a pattern as $p\ q$ against a term u can be handled by two successive matchings of p and q against the same term u ; and (closed) cases can be recognised as those matchable forms which are not equal to themselves, using the equality equal of data structures given in the Example 3.7 of Section 3.2.

4.3 The λ-calculus with patterns

The λ-calculus with patterns (van Oostrom, 1990; Klop *et al.*, 2008) generalises the λ-calculus to support pattern matching, with terms given by the grammar

$$t ::= x \mid t\ t \mid t \to t.$$

Note that there is no alphabet of constructors. Instead, the encodings of constructors as cases can be used directly in patterns. Hence, matching on cases becomes central. Now any term can appear as a pattern but only as a closed pattern immune to substitution and so is not first class, in the sense of the pure pattern calculus. The reduction relation forbids reduction of patterns in the original calculus (van Oostrom, 1990) but allows it in the revised version (Klop *et al.*, 2008). Either way, confluence is still at risk, as shown in Example 2.1, which can be rewritten in the implicit binding grammar as follows.

Example 4.1
If $t = (x\ y \to z)\ ((w \to w)\ f)$, then

$$\begin{array}{lcl} t & \twoheadrightarrow & (x\ y \to z)\ f \\ t & \twoheadrightarrow & z \end{array}$$

shows that t has two different normal forms.

One way of achieving confluence is to keep the greedy matching while restricting the patterns by the CRPC. In the current grammar, patterns are always closed, so the CRPC can be re-expressed in terms closer to those of its authors (Definition 4.19 in van Oostrom, 1990; Definition 4.23 in Klop *et al.*, 2008) by

Rigid pattern condition (RPC): The pattern p is said to satisfy the RPC if, for all substitutions σ_1 and patterns q, such that $\sigma_1 p \Rrightarrow q$, then $q = \sigma_2 p$ for some σ_2 such that $\sigma_1 \Rrightarrow \sigma_2$. The term t is said to be *rigid* if and only if all its patterns satisfy the RPC.

Theorem 4.2 (*Klop et al., 2008*)
The λ-calculus with patterns is confluent on rigid terms.

Proof
Since all patterns are closed, the result follows directly from Corollary 2.11. □

Of course, the challenge is now to identify a non-trivial set of patterns that satisfy the RPC. It is clear that the RPC excludes patterns with *active* variables, where a variable symbol x is said to be *active* in a term t if t contains a sub-term of the form $x\ v$ where x is free in t. Thus for example x and y are both active in $y\ (x\ z)$. The RPC excludes also non-linear patterns, but it does not force patterns to be in normal form. Thus, for instance, let $\Delta = x \to x\ x$ and $\Omega = \Delta\ \Delta$. A term such as $\Omega \to t$ is rigid, even if Ω is reducible (to itself).

The first attempt (van Oostrom, 1990) to define a decidable set of patterns satisfying the RPC naively excludes patterns which are still reducible, imposes linearity and forbids active variables:

$$\Pi = \{p \text{ in normal form } \mid p \text{ is linear } \textit{and} \text{ has no active variables}\}.$$

Define a term to be a Π-term if all its patterns are in the set Π.

While it is clear that counter-example 4.1 is ruled out by condition Π (since x is active in the pattern $x\ y$), not every Π-term is confluent, as the following example shows.

Example 4.2
Let $I = x \to x$ and $t_1 = ((I \to x)\ y) \to z$ and $t_2 = (I \to z)\ I$. Let $t = t_1\ t_2$ and note that all patterns in t are in Π. Then, t reduces to two different normal forms:

$$\begin{array}{lll} t & \to & t_1\ z \\ t & \to & z. \end{array}$$

Thus, condition Π is not sufficient to guarantee confluence or the *RPC*.

To repair this problem, the set Π is restricted further in Klop *et al.* (2008) as follows: A 'λ-term' is a term in the image of the pure λ-calculus, i.e. whose patterns are always variables. Now define

$$\Pi^+ = \{p \text{ is a } \lambda\text{-term in normal form } \mid p \text{ is linear } \textit{and} \text{ has no active variables}\}.$$

Define a term to be a Π^+-term if all its patterns are in the set Π^+.

The previous counter-example is now ruled out, since the pattern $(I \to x)\ y$ is not a λ-term, as I is not a variable. Since patterns in Π^+ enjoy the RPC we thus obtain the following corollary.

Corollary 4.3 (*Klop* et al.*, 2008*)
The λ-calculus with patterns is confluent on Π^+-terms.

A λ-calculus with patterns that is confluent is unable to express path polymorphism (which requires active variables) or pattern polymorphism (which requires free variables). On the other hand, it does work with some λ-terms, such as the first and second projections on pairs. Encode the terms `pair`, $\mathsf{projection}_1$ and $\mathsf{projection}_2$ for pairing and projection as is usually done in λ-calculus (Barendregt, 1984) by

$$\begin{array}{lcl} \mathtt{pair} & = & k \to k\ x\ y \\ \mathsf{projection}_1 & = & (k \to k\ x\ y) \to x \\ \mathsf{projection}_2 & = & k \to k\ x\ y) \to y. \end{array}$$

Then

$$\begin{array}{l} \mathsf{projection}_1\ (\mathtt{pair}\ t\ u) = ((k \to k\ x\ y) \to x)\ (k \to k\ t\ u) \twoheadrightarrow t \\ \mathsf{projection}_2\ (\mathtt{pair}\ t\ u) = ((k \to k\ x\ y) \to y)\ (k \to k\ t\ u) \twoheadrightarrow u \end{array}$$

by using the greedy matching.

Unfortunately, however, this approach does not extend to recursive data types such as lists, at least, using encodings in the style of Church. For example, encode Nil as the term $\lambda z.\lambda f.z$ and Cons as $\lambda x'.\lambda y'.\lambda z.\lambda f.f\ x'\ (y'\ z\ f)$. Now when length (as defined in Section 1) is applied to some Cons $h\ t$, then the first case Nil $\to$ Zero fails, and the second case Cons $x\ y \to$ Succesor (length y) will be applied to Cons $h\ t$. The abstraction Cons may match itself, while x and y are bound to h and t respectively. However, things are not so simple. After all, Cons is an abstraction, so that Cons $h\ t$ may be reduced to $\lambda z.\lambda f.f\ h\ (t\ z\ f)$. To match this, one must also reduce the pattern Cons $x\ y$ to $\lambda z.\lambda f.f\ x\ (y\ z\ f)$. Even this may not be enough, however, since $t\ z\ f$ will reduce if t is a λ-abstraction, but $y\ z\ f$ cannot reduce in the pattern. Since different reduction paths yield incompatible results, reduction is not *confluent*. Of course, the pattern $\lambda z.\lambda f.f\ x\ (y\ z\ f)$ is not rigid, since y is an active variable in the pattern. Moreover, even the pattern Cons $x\ y$ is not rigid, since σ_1 (Cons $x\ y$) reduces to $\lambda z.\lambda f.f\ (\sigma_1\ x)\ ((\sigma_1\ y)\ z\ f)$ which cannot be written as $\sigma_2\ p$ with $\sigma_1 \Rrightarrow \sigma_2$. This shows that the λ-calculus with patterns cannot perform matching on the usual encoding of lists, much less path or pattern polymorphism. This is not to deny the existence of some other encoding. Thus for example, one can encode all n-ary constructors in the same manner, so that Cons is encoded by $\lambda x.\lambda y.\lambda z.z\ x\ y$, but then many distinct constructors become identified.

4.4 Arbitrary compounds as patterns

The first use of pattern matching for path polymorphism occurred in Jay (2004), where it was used to compute things such as the size of a data structure and the addition of two such things. Similar ideas appear in the 'scrap-your-boilerplate' approach (Lämmel & Peyton Jones, 2003). We present here a simple calculus allowing path polymorphism in the spirit of the pattern calculi introduced in Section 2.

In contrast to algebraic patterns which are necessarily headed by constructors, patterns are now given by a more flexible grammar,

$$p, q ::= x \mid c \mid p\ q \text{ if } \mathsf{fv}(p) \cap \mathsf{fv}(q) = \{\}.$$

The patterns of this calculus can be headed by any matchable symbol, whether bound or not. Note that in any case patterns are in normal form and linear. Thus, for example, $x\ y$ is a pattern. Such patterns were not allowed in algebraic calculi (Section 4.1) or λ-calculus with patterns (Section 4.3), since they may contain *active* variables. All free variables in patterns are assumed to be binding, so that, as in all the previous subsections, there is no need to check matches. The pattern-matching operation places a side condition on the rule for applications, by requiring the argument be a compound, not merely an application. Thus, we can adapt the notions of *data structure* and *matchable form* introduced in Section 3.1 for constructors:

$$\begin{array}{rcl} d & ::= & c \mid d\ t \\ m & ::= & d \mid p \rightarrow s. \end{array}$$

The complete algorithm for matching is given by

$$\begin{array}{lcll} \{\!\{u \rhd x\}\!\} & = & \{u/x\} & \\ \{\!\{c \rhd c\}\!\} & = & \{\} & \\ \{\!\{u\ v \rhd p\ q\}\!\} & = & \{\!\{u \rhd p\}\!\} \cup \{\!\{v \rhd q\}\!\} & \text{if } u\ v \text{ is a matchable form} \\ \{\!\{u \rhd p\}\!\} & = & \texttt{fail} \quad \text{otherwise} & \text{if } u \text{ is a matchable form} \\ \{\!\{u \rhd p\}\!\} & = & \texttt{wait} \quad \text{otherwise.} & \end{array}$$

Once again, this can be viewed as the restriction of compound matching to the current syntax. Also, patterns are once again inert, so that the reduction relation can be defined using restricted contexts.

The properties P1 and P2 are easily verified, so that reduction is confluent.

A typical example of path polymorphic function that can be expressed in this calculus is the update of points (Example 3.5) but not the generic update (Example 3.9).

5 Open patterns

When patterns are allowed to contain free variables as well as binding symbols, i.e. when patterns are *open*, then it is necessary to distinguish the binding symbols explicitly, here handled by using the *explicit binding grammar*:

$$\begin{array}{llll} t ::= & & \text{(term)} & \\ & x & \text{(variable)} & \mid \\ & c & \text{(constructor)} & \mid \\ & t\ t & \text{(application)} & \mid \\ & [\theta]\, t \rightarrow t & \text{(case).} & \end{array}$$

Now a free variable in a pattern may or may not be free in its case, according to whether it is a binding symbol or not.

Free and *bound variables* are defined by

$$
\begin{array}{lcllcl}
\mathsf{fv}(x) & = & \{x\} & \mathsf{bv}(x) & = & \{\} \\
\mathsf{fv}(c) & = & \{\} & \mathsf{bv}(c) & = & \{\} \\
\mathsf{fv}(r\ u) & = & \mathsf{fv}(r) \cup \mathsf{fv}(u) & \mathsf{bv}(r\ u) & = & \mathsf{bv}(r) \cup \mathsf{bv}(u) \\
\mathsf{fv}([\theta]\, p \to s) & = & (\mathsf{fv}(p) \cup \mathsf{fv}(s)) \setminus \theta & \mathsf{bv}([\theta]\, p \to s) & = & \mathsf{bv}(p) \cup \mathsf{bv}(s) \cup \theta.
\end{array}
$$

The definitions of symbol renaming, α-conversion, substitution application and the like are defined in the obvious manner (Jay & Kesner, 2006b), given that constructors are unaffected by any of these. For example $\{y/x\}c = c$.

Section 5.1 describes the original pure pattern calculus. Although its reduction is context-free, the identification of variables and matchables exposes it to certain pathologies, which are handled by non-trivial notions of (mutually recursive) matchable forms and reduction steps. Section 5.2 avoids these pathologies in a different manner, by making reduction in the pure pattern calculus *context-sensitive*. Section 5.3 uses the same syntax to describe the open ρ-calculus which uses rigid patterns, rather than compounds, to ensure confluence of matching.

5.1 The original pure pattern calculus

The original pure pattern calculus (Jay & Kesner, 2006a) has terms given by the explicit binding grammar (plus a sole constructor $\bullet$ which need not distract us here). Now, any term can be a pattern, but the *identification* of variables and matchables causes difficulties when reducing patterns.

These difficulties are also present in the version of the pure pattern calculus described in Cirstea and Faure (2007). Consider the example $[x]\ (([]\ x \to x)\ x) \to x$. Its pattern $([]\ x \to x)\ x$ cannot reduce, as x is a free variable and so is waiting for some substitution. On the other hand, this x will never be instantiated, as it is being used for matching, so that reduction is in danger of being blocked.

In the matchable binding grammar, this pattern would be written as $([]\ \widehat{x} \to \widehat{x})\ \widehat{x}$ which reduces to $\widehat{x}$.

The solution adopted in the original calculus (Jay & Kesner, 2006a) was to accept the pattern $([]\ x \to x)\ x)$ as a matchable form, since it cannot be reduced. To be more precise, the definitions of matchable forms and reduction were mutually recursive. Although unambiguous and technically correct, this approach is hard to reason about – or to implement. The pattern-matching operation of this calculus is quite involved, but reduction is always context-free.

Another way of handling these difficulties is to keep track of the variables that are actually matchables, so that reduction becomes context-sensitive. Both this calculus and the pure pattern calculus with matchable symbols are perfectly able to handle the pathological example. This is the approach that will be discussed in more detail in Section 5.2.

We take this opportunity to remark that although the technique in Cirstea and Faure (2007) can be applied to the original pure pattern calculus, they chose to consider a simplified version of this calculus that combines the context-free reduction relation in Figure 1 with a simple definition of matchable forms and the pattern-matching algorithm. This account is of course more elegant than the one originally used in Jay and Kesner (2006a) but cannot reduce the pathological example.

5.2 Context-sensitive pure pattern calculus

In the context-sensitive pattern calculus, reduction of patterns requires that all the definitions of data structures, matchables, matching and reduction be parameterised by a sequence φ of *eventually binding* symbols. Our starting point will be that φ is empty, but reduction of the pattern p of a case $[\theta]\ p \to s$ will include θ among the eventually binding symbols. The only change from the earlier context-sensitive pure pattern calculus (Jay & Kesner, 2006b) is that the calculus now contains constructors, in that each symbol x is used as both a variable x and a *constructor* c_x. Recall that the matchable symbol $\widehat{x}$ in the context-free setting played two roles, of constructor and of a binding symbol in a pattern. Here the binding symbols in a pattern are given by variables, leaving their role as constructors still to be handled. The *constructor names* of t are given by the set $\mathtt{cn}(t)$ of symbols x such that c_x appears in t.

Let φ be a sequence of symbols. The *φ-data structures* (meta-variable d) and *φ-matchable forms* (meta-variable m) are given by the following grammar:

$$\begin{array}{lcl} d & ::= & x\ (x \in \varphi) \mid c_x \mid d\ t \\ m & ::= & d \mid [\theta]\, t \to t, \end{array}$$

where t can be an arbitrary term. Define the *data structures* (respectively *matchable forms*) to be the $\{\}$-data structures (respectively $\{\}$-matchable forms).

Let p and u be terms, and let θ and φ be disjoint sequences of symbols. Define the *matching* $\langle u/[\theta]\ p\rangle_\varphi$ of p against u with respect to binding symbols θ and eventually binding symbols φ to be the check for $\langle\!\langle u \triangleright [\theta]\ p\rangle\!\rangle_\varphi$ on θ, where the *context-sensitive matching* $\langle\!\langle u \triangleright [\theta]\ p\rangle\!\rangle_\varphi$ is the partial operation defined by applying the following equations in order:

$$\begin{array}{lcll}
\langle\!\langle u \triangleright [\theta]\ x\rangle\!\rangle_\varphi & = & \{u/x\} & \text{if } x \in \theta \\
\langle\!\langle c_x \triangleright [\theta]\ c_x\rangle\!\rangle_\varphi & = & \{\} & \\
\langle\!\langle x \triangleright [\theta]\ x\rangle\!\rangle_\varphi & = & \{\} & \text{if } x \in \varphi \\
\langle\!\langle v\ u \triangleright [\theta]\ q\ p\rangle\!\rangle_\varphi & = & \langle\!\langle v \triangleright [\theta]\ q\rangle\!\rangle_\varphi & \\
 & & \uplus\ \langle\!\langle u \triangleright [\theta]\ p\rangle\!\rangle_\varphi & \text{if } q\ p \text{ is a } (\varphi,\theta)\text{-matchable form} \\
 & & & \text{and } v\ u \text{ is a } \varphi\text{-matchable form} \\
\langle\!\langle u \triangleright [\theta]\ p\rangle\!\rangle_\varphi & = & \mathtt{fail} \quad \text{otherwise} & \text{if } p \text{ is a } (\varphi,\theta)\text{-matchable form} \\
 & & & \text{and } u \text{ is a } \varphi\text{-matchable form} \\
\langle\!\langle u \triangleright [\theta]\ p\rangle\!\rangle_\varphi & = & \mathtt{wait} & \text{otherwise.}
\end{array}$$

This matching is similar to compound matching, except that there is an extra rule, allowing variables to match themselves if they are eventually binding symbols.

The *context-sensitive φ-reduction relation* $\overset{\varphi}{\rightarrow}$ is given in Figure 3. It differs from the rules for context-free reduction in that the binding symbols θ of a case $[\theta]\, p \to s$ are added to the eventually binding symbols φ when reducing the pattern p. The *context-sensitive reduction relation* $\overset{\{\}}{\rightarrow}$ is defined by setting φ to be empty.

$$\frac{}{([\theta]\,p \to s)\ u \xrightarrow{\varphi} \langle u/[\theta]\ p\rangle_{\varphi}\ s} \qquad \frac{r \xrightarrow{\varphi} r'}{r\ u \xrightarrow{\varphi} r'\ u} \qquad \frac{u \xrightarrow{\varphi} u'}{r\ u \xrightarrow{\varphi} r\ u'}$$

$$\frac{p \xrightarrow{\varphi,\theta} p'}{[\theta]\,p \to s \xrightarrow{\varphi} [\theta]\,p' \to s} \qquad \frac{s \xrightarrow{\varphi} s'}{[\theta]\,p \to s \xrightarrow{\varphi} [\theta]\,p \to s'}$$

Fig. 3. The context-sensitive reduction relation.

The properties of reduction are easily established once the calculus is shown to be isomorphic to the pure pattern calculus with matchable symbols. Let CS-terms be the terms of the former calculus and CF-terms be the terms of the latter.

We translate CS-terms to CF-terms by $\mathtt{T}(t) = \mathtt{T}_{\{\}}(t)$, where $\mathtt{T}_{\varphi}(t)$ is defined by induction as follows:

$$\begin{array}{lcll}
\mathtt{T}_{\varphi}(x) & = & x & \text{if } x \notin \varphi \\
\mathtt{T}_{\varphi}(x) & = & \widehat{x} & \text{if } x \in \varphi \\
\mathtt{T}_{\varphi}(c_x) & = & \widehat{x} & \\
\mathtt{T}_{\varphi}(r\ u) & = & \mathtt{T}_{\varphi}(r)\ \mathtt{T}_{\varphi}(u) & \\
\mathtt{T}_{\varphi}([\theta]\,p \to s) & = & [\theta]\,\mathtt{T}_{\varphi,\theta}(p) \to \mathtt{T}_{\varphi}(s) & \text{if } \theta \cap (\mathtt{cn}(p), \varphi) = \{\}.
\end{array}$$

Note that eventually binding symbols and constructors are both mapped to matchables, while other symbols remain variables. Also, translation of a case adds its binding symbols to the eventually binding symbols used when translating its pattern. Matches are translated by $\mathtt{T}_{\varphi}(\sigma)\ x = \mathtt{T}_{\varphi}(\sigma x)$ with $\mathtt{T}_{\varphi}(\mu) = \mu$ otherwise.

Conversely, we translate CF-terms to CS-terms by $\mathtt{W}(t) = \mathtt{W}_{\{\}}(t)$, where $\mathtt{W}_{\varphi}(t)$ is defined by induction as follows:

$$\begin{array}{lcll}
\mathtt{W}_{\varphi}(x) & = & x & \\
\mathtt{W}_{\varphi}(\widehat{x}) & = & c_x & \text{if } x \notin \varphi \\
\mathtt{W}_{\varphi}(\widehat{x}) & = & x & \text{if } x \in \varphi \\
\mathtt{W}_{\varphi}(r\ u) & = & \mathtt{W}_{\varphi}(r)\ \mathtt{W}_{\varphi}(u) & \\
\mathtt{W}_{\varphi}([\theta]\,p \to s) & = & [\theta]\,\mathtt{W}_{\varphi,\theta}(p) \to \mathtt{W}_{\varphi}(s) & \text{if } \theta \cap (\mathsf{fv}(p), \varphi) = \{\}.
\end{array}$$

Note that matchables are translated to variables if they are eventually binding symbols and to constructors otherwise. Matches are translated by $\mathtt{W}_{\varphi}(\sigma)\ x = \mathtt{W}_{\varphi}(\sigma x)$ with $\mathtt{W}_{\varphi}(\mu) = \mu$ otherwise.

Note that the translations set up an exact correspondence between the matchable symbols on the one hand and the eventually binding symbols and constructors of the context-sensitive calculus on the other hand: the rule of compound matching that matches a matchable with itself exactly corresponds to the two equations of the context-sensitive matching that consider eventually binding symbols and constructors. This makes it easy to show that matching and reduction are preserved.

Lemma 5.1
Let t be a CS-term, and let φ be a sequence of symbols such that $\varphi \cap \mathtt{cn}(t) = \{\}$. Then $\mathtt{W}_\varphi(\mathtt{T}_\varphi(t)) = t$. Thus in particular $\mathtt{W}(\mathtt{T}(t)) = t$. Conversely, let t be a CF-term, and let φ be a sequence of symbols such that $\varphi \cap \mathsf{fv}(t) = \{\}$. Then $\mathtt{T}_\varphi(\mathtt{W}_\varphi(t)) = t$. Thus in particular $\mathtt{T}(\mathtt{W}(t)) = t$.

Proof
Let t be a CS-term. If $t = c_x$, then $\mathtt{W}_\varphi(\mathtt{T}_\varphi(c_x)) = \mathtt{W}_\varphi(\widehat{x}) = c_x$, since $x \notin \varphi$ by hypothesis. If $t = x$, suppose $x \in \varphi$. Then $\mathtt{W}_\varphi(\mathtt{T}_\varphi(x)) = \mathtt{W}_\varphi(\widehat{x}) = x$. Otherwise $x \notin \varphi$ and $\mathtt{W}_\varphi(\mathtt{T}_\varphi(x)) = \mathtt{W}_\varphi(x) = x$. If t is an application, then apply induction twice. If $t = [\theta]\, p \to s$, then $\mathtt{W}_\varphi(\mathtt{T}_\varphi(t)) = [\theta]\, \mathtt{W}_{\varphi,\theta}(\mathtt{T}_{\varphi,\theta}(p)) \to \mathtt{W}_\varphi(\mathtt{T}_\varphi(s))$. Now the induction hypothesis gives the result, since $\mathtt{cn}(p) \cap \varphi = \{\}$ by hypothesis and $\mathtt{cn}(p) \cap \theta = \{\}$ by definition.

Conversely, let t be a CF-term. If $t = x$, then $\mathtt{T}_\varphi(\mathtt{W}_\varphi(x)) = \mathtt{T}_\varphi(x) = x$, since $x \notin \varphi$ by hypothesis. If $t = \widehat{x}$, suppose $x \notin \varphi$. Then $\mathtt{T}_\varphi(\mathtt{W}_\varphi(\widehat{x})) = \mathtt{T}_\varphi(c_x) = \widehat{x}$. Otherwise, $x \in \varphi$ so that $\mathtt{T}_\varphi(\mathtt{W}_\varphi(\widehat{x})) = \mathtt{T}_\varphi(x) = \widehat{x}$. If t is an application, then apply induction twice. If $t = [\theta]\, p \to s$, then $\mathtt{T}_\varphi(\mathtt{W}_\varphi(t)) = [\theta]\, \mathtt{T}_{\varphi,\theta}(\mathtt{W}_{\varphi,\theta}(p)) \to \mathtt{T}_\varphi(\mathtt{W}_\varphi(s))$. The induction hypothesis gives the result, since $\mathsf{fv}(p) \cap \varphi = \{\}$ by hypothesis and $\mathsf{fv}(p) \cap \theta = \{\}$ by definition. □

Lemma 5.2
Let φ be a sequence of symbols. If t is a φ-matchable form in CS, then $\mathtt{T}_\varphi(t)$ is a CF-matchable form. Conversely, if t is a CF-matchable form, then $\mathtt{W}_\varphi(t)$ is a φ-matchable form in CS.

Proof
The proofs are by straightforward inductions on the structure of t. □

Lemma 5.3
Let μ be a decided CS-match. If $\mathsf{dom}(\mu) \cap \varphi = \{\}$, then $\mathtt{T}_\varphi(\mu t) = \mathtt{T}_\varphi(\mu)\mathtt{T}_\varphi(t)$.

Proof
If $\mu = \mathtt{fail}$, then $\mathtt{T}_\varphi(\mu t) = [x]\, \widehat{x} \to x = \mathtt{fail}\ \mathtt{T}_\varphi(t) = \mathtt{T}_\varphi(\mathtt{fail})\mathtt{T}_\varphi(t)$. If μ is a substitution σ, then the proof is by a straightforward induction on the structure of t. □

Lemma 5.4
Let $t = ([\theta]\, p \to s)\ u$ be a CS-term, and let φ be a sequence of symbols such that $\varphi \cap \mathtt{cn}(t) = \{\}$. If $\sigma = \langle\!\langle u \triangleright [\theta]\ p \rangle\!\rangle_\varphi$, then $\mathtt{T}_\varphi(\sigma) = \{\!\{\mathtt{T}_\varphi(u) \triangleright [\theta]\ \mathtt{T}_{\varphi,\theta}(p)\}\!\}$. If $\langle\!\langle u \triangleright [\theta]\ p \rangle\!\rangle_\varphi = \mathtt{fail}$, then $\{\!\{\mathtt{T}_\varphi(u) \triangleright [\theta]\ \mathtt{T}_{\varphi,\theta}(p)\}\!\} = \mathtt{fail}$. Thus, if $\mu = \langle u/[\theta]\ p\rangle_\varphi$, then $\mathtt{T}_\varphi(\mu) = \{\mathtt{T}_\varphi(u)/[\theta]\ \mathtt{T}_{\varphi,\theta}(p)\}$.

Proof
The proof is by a long, but straightforward, induction on the definition of context-sensitive matching using Lemma 5.2. □

Lemma 5.5
Let μ be a decided math. If $\varphi \cap \mathsf{dom}(\mu) = \{\}$, then $\mathtt{W}_\varphi(\mu s) = \mathtt{W}_\varphi(\mu)\mathtt{W}_\varphi(s)$.

Proof
If μ is $\mathtt{fail}$, then $\mathtt{W}_\varphi(\mathtt{fail})\mathtt{W}_\varphi(s) = \mathtt{fail}\ \mathtt{W}_\varphi(s) = [x]\, x \to x = \mathtt{W}_\varphi(\mathtt{fail}\ s)$. If μ is a substitution σ, the proof is by a straightforward induction on the structure of t.
□

Lemma 5.6
Let $t = ([\theta]\, p \to s)\ u$ be a CF-term, and let φ be a sequence of symbols such that $\varphi \cap \mathsf{fv}(t) = \{\}$. If $\sigma = \{\!\{u \triangleright [\theta]\ p\}\!\}$, then $\mathtt{W}_\varphi(\sigma) = \langle\!\langle \mathtt{W}_\varphi(u) \triangleright [\theta]\ \mathtt{W}_{\varphi,\theta}(p) \rangle\!\rangle_\varphi$. If $\{\!\{u \triangleright [\theta]\ p\}\!\} = \mathtt{fail}$, then $\langle\!\langle \mathtt{W}_\varphi(u) \triangleright [\theta]\ \mathtt{W}_{\varphi,\theta}(p) \rangle\!\rangle_\varphi = \mathtt{fail}$. Thus, if $\mu = \{u/[\theta]\ p\}$, then $\mathtt{W}_\varphi(\mu) = \langle \mathtt{W}_\varphi(u)/[\theta]\ \mathtt{W}_{\varphi,\theta}(p) \rangle_\varphi$.

Proof
The proof is by a long, but straightforward, induction on the definition of context-sensitive matching using Lemma 5.2. □

The last step of this section consists in relating reduction in CF and CS.

Theorem 5.7
Let t be a CS-term, and let φ be a sequence of symbols such that $\varphi \cap \mathtt{cn}(t) = \{\}$. If $t \overset{\varphi}{\to} t'$, then $\mathtt{T}_\varphi(t) \to \mathtt{T}_\varphi(t')$.

Proof
The proof is by induction on the structure of t. If $t = ([\theta]\, p \to s)\ u \overset{\varphi}{\to} \langle u/[\theta]\ p\rangle_\varphi\ \ s = t'$, then

$$\begin{array}{rcll}
\mathtt{T}_\varphi(([\theta]\, p \to s)\ u) & = & ([\theta]\, \mathtt{T}_{\varphi,\theta}(p) \to \mathtt{T}_\varphi(s))\ \mathtt{T}_\varphi(u) & \\
& \to & \{\mathtt{T}_\varphi(u)/[\theta]\ \mathtt{T}_{\varphi,\theta}(p)\}\ \mathtt{T}_\varphi(s) & \\
& = & \mathtt{T}_\varphi(\langle u/[\theta]\ p\rangle_\varphi\ )\ \mathtt{T}_\varphi(s) & \text{(by Lemma 5.4)} \\
& = & \mathtt{T}_\varphi(\langle u/[\theta]\ p\rangle_\varphi\ \ s) & \text{(by Lemma 5.3).}
\end{array}$$

All the other cases are straightforward. □

Theorem 5.8
Let t be a CF-term, and let φ be a sequence of symbols such that $\varphi \cap \mathsf{fv}(t) = \{\}$. If $t \to t'$, then $\mathtt{W}_\varphi(t) \overset{\varphi}{\to} \mathtt{W}_\varphi(t')$.

Proof
The proof is by induction on the structure of t. If $t = ([\theta]\, p \to s)\ u \to \{u/[\theta]\ p\}\ s = t'$, then

$$\begin{array}{rcll}
\mathtt{W}_\varphi(([\theta]\, p \to s)\ u) & = & ([\theta]\, \mathtt{W}_{\varphi,\theta}(p) \to \mathtt{W}_\varphi(s))\ \mathtt{W}_\varphi(u) & \\
& \overset{\varphi}{\to} & \langle \mathtt{W}_\varphi(u)/[\theta]\ \mathtt{W}_{\varphi,\theta}(p)\rangle_\varphi\ \ \mathtt{W}_\varphi(s) & \\
& = & \mathtt{W}_\varphi(\{u/[\theta]\ p\})\ \mathtt{W}_\varphi(s) & \text{(by Lemma 5.6)} \\
& = & \mathtt{W}_\varphi(\{u/[\theta]\ p\}s) & \text{(by Lemma 5.5).}
\end{array}$$

All the other cases are straightforward. □

When φ is empty, then the premises of the last two theorems are automatically satisfied. Hence we have the following corollary.

Corollary 5.9
The translations `W` and `T` form an isomorphism between the CS-terms and the CF-terms that preserves matching, substitution and reduction.

Corollary 5.10
The reduction relation $\overset{\{\}}{\rightarrow}$ is confluent.

Proof
Since the isomorphic relation $\rightarrow$ is confluent by Theorem 3.1. $\square$

Corollary 5.11
The reduction relation of the context-free pure pattern calculus in Jay and Kesner (2006b) is confluent.

Proof
It is the sub-calculus of the CS-calculus described here, obtained by forbidding constructors. Such terms are closed under reduction. $\square$

5.3 *Open ρ-calculus*

The ρ-calculus (Cirstea and Kirchner 2001) (or *rewriting calculus*) was introduced to specify a large class of pattern calculi dealing with matching operations over rich theories (such as associative–commutative theories). Such specification is especially interesting to make the operational semantics independent of the particular matching mechanism used to evaluate programs. Also, such specifications allow non-deterministic languages to be modelled in a very natural way. Nevertheless, there is considerable interest in establishing confluence of ρ-calculi.

The first versions of the ρ-calculus appearing in the literature use closed patterns so that they can be understood by means of the calculi in Section 4. Since then, a more refined version of the ρ-calculus was proposed in Barthe *et al.* (2003) to study properties of ρ-calculi in *type theory*. However, an untyped variant of it can be expressed using the explicit binding grammar. It supports open patterns, constructors and successful matching of cases. Interestingly, matching is context-sensitive, but reduction is context-free.

Confluence is obtained by *extending* the RPC, which generalises the RPC in Section 4.3 to handle open patterns. It is not yet clear if the extended RPC implies the RMC.

Syntactically, this ρ-calculus is closest in spirit to the pure pattern calculus, since it allows free variables in patterns and pattern reduction. The biggest differences from the pure pattern calculi is that it allows successful matching on open patterns as well as closed patterns and uses rigid patterns to achieve confluence, rather than matchable forms.

Though confluent, its rigid patterns do not include most of the challenging examples considered earlier. In particular, it can support neither the Church-style encoding of `Cons` (for the same reasons as given in Section 4.3) nor the patterns used in the path and pattern polymorphic examples in Section 3.2, except perhaps equality.

6 Conclusions

The pure pattern calculus provides a simple and expressive account of pattern matching which can be understood in a slightly more general framework. It is simple because it has only four term forms, without requiring a separate class of patterns. Indeed, patterns are now first-class citizens, so that they can be evaluated, used as arguments and returned as results. This approach provides a rich expressive formalism because it can support two new forms of polymorphism: path polymorphic functions such as updatePoint, which allow matching with arbitrary data structures, and pattern polymorphic functions such as the generic eliminator elim and generic update update, which can treat any term as a pattern.

The paper provides a general framework for discussing context-free pattern-calculi in the literature based on open patterns as well as closed patterns. In particular, it gives formal tools to provide a general proof of confluence for the reduction relation of pattern calculi whose reduction is context-free and whose matching operation satisfies the RPC.

The RPC (with greedy matching) implies P1 and P2, which in turn imply the RMC, which implies confluence of context-free reduction. The properties P1 and P2 are satisfied by all well-known confluent pattern calculi in which successful matching is limited to closed patterns. The only other calculus we are aware of is the open ρ-calculus: the establishment of its confluence in the framework may require further generalisation of our approach.

Concerning expressivity, the pure pattern calculi are the only known calculi able to support path and pattern polymorphisms. Some calculi support examples not in the pure calculi, but these do not seem very significant. For example, the λ-calculus with patterns is able to express the Church-style encoding of Pair but not of Cons. Also, the open ρ-calculus is able to match free variables with themselves, but no particular examples have been offered to motivate this level of generality.

There may be some potential for further generalisation of matching in the pure pattern calculus. In particular, one could add case matching and perhaps some matching of open patterns, without requiring that patterns be rigid. This could support matching when constructors such as Cons are encoded in the Church style. However, when data structures are fundamental, it is not clear how important this might be. Perhaps case matching has other benefits e.g. in program analysis.

The power of the pure pattern calculi derives from its identification of data structures as being separate from, and equally important to, the abstractions (or cases) which are so central to the λ-calculus. This balance between functionality and structure is the source of new forms of flexibility in programming.

The pure pattern calculus with matchable symbols introduced here is specified by means of one particular syntax allowing each symbol x to be used as a variable or as a matchable. It allows matchable forms to be easily described while keeping reduction in a context-free setting. This calculus is formally equivalent to the context-sensitive pure pattern calculus but more expressive than the original pure pattern calculus, since it can match more patterns.

Some work has been done on providing type systems for pattern calculi (Jay, 2004, 2009) and implementing them in the programming language 'bondi' (available at `www-staff.it.uts.edu.au/~cbj/bondi`). Also, it would be interesting to understand the logical system behind the pure pattern calculus by means of the Curry–Howard approach. Again, definition of different reduction strategies for the pure pattern calculus seems to be pertinent to the treatment of infinite data. This could consider explicit pattern matching, in contrast to the implicit (meta-level) pattern matching used in this paper.

Acknowledgments

We would like to thank Thibaut Balabonski, Germain Faure, Thomas Given-Wilson, Olivier Laurent, Simon Peyton Jones, Eugenio Moggi, Alexandre Miquel and Matthew Roberts for stimulating discussions and constructive suggestions.

References

Baader, F. & Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge: Cambridge University Press.

Barendregt, H. (1984) *The Lambda Calculus: Its Syntax and Semantics*, vol. 103, Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland.

Barthe, G., Cirstea, H., Kirchner, C. & Liquori, L. (2003) Pure pattern type systems. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*. Greg Morrisett (ed), New Orleans, USA, New York: ACM Press, pp. 250–261.

Bezem, M., Klop, J. W. & De Vrijer, R. (eds) (2003) *Term Rewriting Systems – Terese*, vol. 55, Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press.

Böhm, C., Piperno, A. & Guerrini, S. (1994) Lambda-definition of function(al)s by normal forms. In *Proceedings of the 5th European Symposium on Programming (ESOP)*. Donald Sannella (ed), Edinburgh, vol. 788, Lecture Notes in Computer Science, Berlin: Springer, pp. 135–149.

Cirstea, H. & Faure, G. (2007) Confluence of pattern-based lambda-calculi. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA)*. Franz Baader (ed), Paris, vol. 4533, Lecture Notes in Computer Science, Berlin: Springer, pp. 78–92.

Cirstea, H. & Kirchner, C. (2001) The rewriting calculus — Part I and II, *Logic Journal of the IGPL*, **9**(3), 427–498.

De Nicola, R., Ferrari, G. L. & Pugliese, R. (1998) KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. Software Engng* **24**(5), 315–330.

Forest, J. & Kesner, D. (2003) Expression reduction systems with patterns. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA)*. Robert Nieuwenhuis (ed), Valencia, Spain, vol. 2706, Lecture Notes in Computer Science, Berlin: Springer, pp. 107–122.

Gelernter, D. (1985) Generative communication in Linda. **7**(1), 80–112.

Gorla, D. & Pugliese, R. (2003) Resource access and mobility control with dynamic privileges acquisition. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*. Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow & Gerhard J. Woeginger (eds), Eindhoven, The Netherlands, vol. 2719, Lecture Notes in Computer Science, Berlin: Springer, pp. 119–132.

Huang, F. Y., Jay, C. B. & Skillicorn, D. B. (2006a) Adaptiveness in well-typed java bytecode verification. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON)*. Kelly Lyons & Christian Couturier (eds), Toronto, Canada, New York: ACM Press, pp. 248–262.

Huang, F. Y., Jay, C. B. & Skillicorn, D. B. (2006b) Programming with heterogeneous structures: Manipulating XML data using **bondi.** *29th Australasian Computer Science Conference (ACSC'06)*. Gill Dobbie & Vladimir Estivill-Castro (eds). ACM, pp. 287–296.

Jay, B. (2009) *Pattern Calculus: Computing with Functions and Structures*. Berlin: Springer.

Jay, B & Kesner, D. (2006) Pure pattern calculus. In *Proceedings of the 15th European Symposium on Programming (ESOP)*. Peter Sestoft (ed), Vienna, Austria, vol. 3924, Lecture Notes in Computer Science, Berlin: Springer, pp. 100–114.

Jay, C. B. (2004) The pattern calculus. *ACM Trans. Prog. Lang. Sys.* **26**(6), 911–937.

Jay, C. B. & Kesner, D. (2006) Patterns as first-class citizens. Available at `http://hal.archives-ouvertes.fr/hal-00229331/fr/`.

Kahl, W. (2004) Basic pattern matching calculi: A fresh view on matching failure. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS)*. Yukiyoshi Kameyama, Peter J. Stuckey (eds), Nara, Japan, vol. 2998, Lecture Notes in Computer Science, Berlin: Springer, pp. 276–290.

Klop, J.-W. (1980) *Combinatory Reduction Systems*. Amsterdam: Mathematical Centre Tracts, CWI.

Klop, J.-W., van Oostrom, V. & de Vrijer, R. (2008) Lambda calculus with patterns. *Theoret. Comp. Sci.* **398**(1–3), 16–31.

Lämmel, R. & Peyton-Jones, S. (2003) Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*. Peter Lee (ed), New Orleans, USA, vol. 38, no. 3, SIGPLAN Notices, pp. 26–37.

Paulson, L. C. (1994) *Isabelle: A Generic Theorem-Prover*, vol. 828, Lecture Notes in Computer Science. Springer.

Peyton Jones, S. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall.

Pfenning, F. & Paulin-Mohring, C. (1989) Inductively defined types in the calculus of constructions. In *Proceedings of the 5th international conference on Mathematical Foundations of Programming Semantics (MFPS)*. Michael G. Main, Austin Melton, Michael W. Mislove & David A. Schmidt (eds), New Orleans, USA, vol. 442, Lecture Notes in Computer Science, Berlin: Springer, pp. 209–228.

Van Oostrom, V. (1990) *Lambda Calculus with Patterns*. Amsterdam: Vrije Universiteit.

Visser, E. (2004) Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation: Revised Papers*, vol. 3016, Lecture Notes in Computer Science, Berlin: Springer pp. 216–238.