Pure pattern calculus

Barry Jay[1] and Delia Kesner[2]

[1] University of Technology, Sydney, cbj@it.uts.edu.au
[2] PPS, CNRS and Université Paris 7, kesner@pps.jussieu.fr

Abstract. The pure pattern calculus generalises the pure lambda-calculus by basing computation on pattern-matching instead of beta-reduction. The simplicity and power of the calculus derive from allowing any term to be a pattern. As well as supporting a uniform approach to functions, it supports a uniform approach to data structures which underpins two new forms of polymorphism. *Path polymorphism* supports searches or queries along all paths through an arbitrary data structure. *Pattern polymorphism* supports the dynamic creation and evaluation of patterns, so that queries can be customised in reaction to new information about the structures to be encountered. In combination, these features provide a natural account of tasks such as programming with XML paths.
As the variables used in matching can now be eliminated by reduction it is necessary to separate them from the binding variables used to control scope. Then standard techniques suffice to ensure that reduction progresses and to establish confluence of reduction.

1 Introduction

The lambda-calculus is a theory of functions which is powerful enough to model arbitrary computations. In its pure form every term is a function, so that function arguments are themselves functions. Such higher-order functions give a clean account of recursion as the application of the fixpoint function. Also data structures such as pairs, lists and trees can be modelled as higher-order functions that take as arguments functions that are to act on the data stored within the structure. Central to the expressive power of the lambda-calculus is that a single rule, beta-reduction, is used to describe the evaluation of an arbitrary function. This uniformity allows a single function to be applied in a variety of different situations, i.e. supports function polymorphism. Unfortunately, the description of data structures is not so uniform. Although the lambda-calculus supports functions that act uniformly on all pairs, or all lists, it cannot support operations that exploit characteristics common to all data structures. These include operations for searching, updating and aggregating that are at the heart of data processing but do not make sense with lambda-abstractions.

In a way, this is surprising because such operations can be specified quite simply. Every data structure is either an *atom* or a *compound*. For example, every list is either empty or is compounded from a head and a tail, every tree is either a leaf or a node. With this characterisation, one can define searching a data structure d as follows:

1. if d is the goal then return d;
2. else if d is a compound data structure then traverse its components;
3. else stop.

For example, consider the problem of listing all the points in a data structure. Each point is represented by terms of the form $\mathsf{Point}\ t$ where Point is a constructor used to represent points whose data is represented by t but the nature of the structure holding the points is not known. Let the syntax $[x_1, \ldots, x_n]$ be used for the list whose entries are given by $x_1, \ldots, x_n$ and let $s@t$ be the result of appending s to the front of t. Now the solution can be given by a pattern-matching function defined by *cases*

$$
\begin{array}{ll}
\mathsf{letrec\ listPoints} = & \\
\quad \mathsf{Point}\ y & \to [\mathsf{Point}\ y] \\
\mid\ x\ y & \to (\mathsf{listPoints}\ x)\ @\ (\mathsf{listPoints}\ y) \\
\mid\ y & \to [\,]
\end{array}
$$

The most interesting case of the three is the second, whose pattern $x\ y$ is able to match against an *arbitrary* compound data structure. For example, when $\mathsf{listPoints}$ is applied to a pair $\mathsf{Pair}\ s\ t$ of points s and t we get

$$
\begin{aligned}
\mathsf{listPoints}\ (\mathsf{Pair}\ s\ t) &= ((\mathsf{listPoints\ Pair})\ @\ (\mathsf{listPoints}\ s))\ @\ (\mathsf{listPoints}\ t) \\
&= ([\,]\ @\ [s])\ @\ [t] \\
&= [s, t]
\end{aligned}
$$

This uniform approach to compound data structures supports *path polymorphism* in which all paths through a data structure can be traversed.

Another example of path polymorphism is the function that updates point data within an arbitrary data structure. It is given by

$$
\begin{array}{ll}
\mathsf{letrec\ updatePoint} = & f \to \\
\quad \mathsf{Point}\ y & \to \mathsf{Point}\ (f\ y) \\
\mid z\ y & \to (\mathsf{updatePoint}\ f\ z)\ (\mathsf{updatePoint}\ f\ y) \\
\mid y & \to y
\end{array}
$$

Further generalisation is achieved by making Point a parameter to the *generic update function* defined by

$$
\begin{array}{ll}
\mathsf{letrec\ update} = & x \to f \to \\
\quad x\ y & \to_y x\ (f\ y) \\
\mid z\ y & \to\ (\mathsf{update}\ x\ f\ z)\ (\mathsf{update}\ x\ f\ y) \\
\mid y & \to\ y
\end{array}
$$

This time the two variables in the pattern $x\ y$ above behave quite differently as x is a *free variable* ready to be substituted by, say, Point while y is a binding variable, as usual. To distinguish these alternatives, the arrow in the case is decorated with a set of *binding variables*, in this case just y. Where no subscript is specified then all the free variables of the pattern are assumed bound.

The function update is *pattern polymorphic*, as it contains the free variable x in the pattern $x\ y$ whose instantiation can produce a variety of different update functions. For example, update Point reduces to updatePoint. Further, if update is applied to a case then the pattern must be reduced before matching can occur.

Such flexibility in the use of patterns leads to the following leitmotiv:

any term can be a pattern.

This complements the view in lambda-calculus that any term can be a function.

Hence, the *pure pattern calculus* has term syntax

$$t ::= x \mid \bullet \mid t\ t \mid t \rightarrow_\theta t$$

consisting of variables, a constructor, applications and cases $p \rightarrow_\theta s$ where θ is a set of variables, its *binding variables.* The sole reduction rule is motivated by the equation

$$(p \rightarrow_\theta s)\ u = \{p/u\}_\theta\ s \tag{1}$$

where $\{p/u\}_\theta$ is the *match* of p against u that produces either a substitution with domain θ or a failure.

A key challenge is to determine when a pattern is irreducible, and so is ready for matching. The difficulty arises from some pathological examples in which reduction of the pattern is blocked by binding variables of its own case. That resolved, the pure pattern calculus is fairly well behaved. In particular, every closed term of the form $(p \rightarrow_\theta s)\ u$ is reducible. Also, reduction is confluent.

The simplicity of the pure pattern calculus is best appreciated by comparing to previous approaches to pattern-matching. Popular functional programming languages such as Standard ML [SML], OCAML [Oca] and Haskell [Has] only support irreducible patterns which are either headed by a constructor or are a binding variable. Recent research has sought to augment the collection of patterns with new constructions [KPT96], reducible patterns [CK98] and free variables which do not bound occurrences in the body of the program [BCKL03], and patterns for compound data structures [Jay04c]. Only the last of these supports path polymorphism and none of them supports pattern polymorphic examples.

Moreover, the last of these underpinned the development of typed calculi supporting pattern polymorphism (e.g. [Jay04a]) which sought to allow more dynamic patterns (for us, polymorphism is about re-usability, which may be formalised by typing). These have been used to support the generic update, and its extension to handle arbitrary XML paths [HJS05a,HJS05b]. They have also been used to support an object model able to support central goals of object-orientation [Jay04b]. Again, they provide an account of *structure polymorphism* [JC94,JBM98,Jay04c] necessary to support operations such as mapping and folding in a uniform way, similar to *polytypic* or *generic functional programming* [Jan00,BdMH96,GJ03].

All these calculi attempted to control variable binding by restricting the class of patterns and their reduction. However, simplicity comes by treating binding separately from the pattern itself. It is expected that all of the applications above can be re-engineered in the new, simpler, framework.

The structure of the rest of the paper is as follows. Section 2 introduces the terms. Section 3 defines reduction. Section 4 considers some examples. Section 5 shows that matching does not get stuck. Section 6 proves reduction is confluent. Section 7 draws conclusions and considers further work.

Acknowledgements We would like to thank the anonymous referees and Eugenio Moggi for their constructive criticism.

2 Terms

Fix a countable alphabet of variables (meta-variables $\ldots x, y, z$). Let θ denote a finite set of variables. The term syntax of the *pure pattern calculus* is

$$\begin{array}{llll} Terms & t ::= & x & \text{(variable)} \\ & & \bullet & \text{(constructor)} \\ & & t\ t & \text{(application)} \\ & & t \rightarrow_\theta t & \text{(case)} \end{array}$$

The variables are available for binding, matching and substitution. The constructor is used to build data structures. In practice, other primitive constructors could be considered but we do not include them for simplicity. The application $r\ u$ applies the *function* r to its *argument* u. The case $p \rightarrow_\theta s$ is formed of a *pattern* p and a *body* s linked by the set θ of *binding variables*. Application is left-associative and case is right-associative. Application binds tighter than case. For example $x \rightarrow x\ y\ z \rightarrow_y y$ is equal to $x \rightarrow (((x\ y)\ z) \rightarrow_y y)$. Lambda-abstraction can be defined by setting $\lambda x.t$ to be $x \rightarrow_x t$.

Free variables of terms are defined by:

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(\bullet) &= \{\} \\ \mathsf{fv}(r\ u) &= \mathsf{fv}(r) \cup \mathsf{fv}(u) \\ \mathsf{fv}(p \rightarrow_\theta s) &= (\mathsf{fv}(p) \cup \mathsf{fv}(s)) \setminus \theta. \end{aligned}$$

Hence the binding variables of a case bind their free occurrences in both the pattern and body. A term is *closed* if it has no free variables.

The notation $p \rightarrow s$ stands for $p \rightarrow_{\mathsf{fv}(p)} s$. Hence programmers need never actually mention binding variables explicitly unless they require free variables in the pattern.

2.1 Matches

A *substitution* σ is a partial function from variables to terms. The notation $\{x_1/u_1, \ldots, x_n/u_n\}$ represents the substitution that maps x_i to u_i for $i = 1 \ldots n$ and $\{\}$ denotes the empty substitution. A *match* (meta-variable m) is either a *successful match*, given by a substitution, or a *failure*, denoted by none. The usual concepts and notation associated with substitutions will be defined for arbitrary matches.

The *domain* of σ is denoted $\mathsf{dom}(\sigma)$. The domain of none is the empty set. The set of *free variables* of σ is given by the union of the sets $\mathsf{fv}(\sigma x)$ where $x \in \mathsf{dom}(\sigma)$. Also, none has no free variables. Define the *variables* of m to be $\mathsf{var}(m) = \mathsf{dom}(m) \cup \mathsf{fv}(m)$.

The *application* of a substitution σ to a term is defined by

$$\begin{array}{lll} \sigma x & = \sigma x & \text{if } x \in \mathsf{dom}(\sigma) \\ \sigma x & = x & \text{if } x \notin \mathsf{dom}(\sigma) \\ \sigma \bullet & = \bullet & \\ \sigma(r\ u) & = (\sigma r)\ (\sigma u) & \\ \sigma(p \to_\theta s) & = \sigma p \to_\theta \sigma s & \text{if } \mathsf{var}(\sigma) \cap \theta = \{\} \end{array}$$

The restriction on the definition of $\sigma(p \to_\theta s)$ is necessary to avoid a *variable clash* which would change the semantics of the term. Variable clashes will be handled by α-conversion.

If matching fails in Equation (1) then none will be applied to the body of the case, which should be discarded. One possibility is to introduce a special error term, but match failure provides a natural branching mechanism which can be used to underpin the definitions of conditionals and pattern-matching functions. Hence, we define

$$\mathsf{none}\ t = x \to_x x.$$

Given matches m_1 and m_2 then $m_1 \uplus m_2$ is the match defined as follows. If m_1 and m_2 are substitutions σ_1 and σ_2 *whose domains are disjoint* then $\sigma_1 \uplus \sigma_2$ is defined by

$$(\sigma_1 \uplus \sigma_2)x = \begin{cases} \sigma_1 x & \text{if } x \in \mathsf{dom}(\sigma_1) \\ \sigma_2 x & \text{if } x \in \mathsf{dom}(\sigma_2) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In all other circumstances $m_1 \uplus m_2 = \mathsf{none}$. Disjoint domains will be used to ensure that matching is deterministic.

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 m_1 and m_2 are matches of which at least one is none then $m_2 \circ m_1$ is defined to be none.

The *check* m_θ of a match m on a set of variables θ is m if m is a substitution whose domain is exactly θ and is none otherwise.

2.2 Alpha conversion

Let θ be a set of variables and x and y be variables. Then $\{x/y\}\theta$ is defined to be the set obtained by replacing x by y in θ if $x \in \theta$ and $y \notin \theta$, and to be undefined otherwise.

Alpha conversion is the congruence relation generated by the following axiom

$$p \to_\theta s =_\alpha \{x/y\}p \to_{\{x/y\}\theta} \{x/y\}s \quad \text{if } y \notin \mathsf{fv}(p) \cup \mathsf{fv}(s).$$

For example, $x\ y \to_y x\ (f\ y) =_\alpha x\ z \to_z x\ (f\ z)$ if z is not free in f.

Lemma 1. *For every substitution σ and term t there is an α-equivalent term t' such that $\sigma t'$ is defined. If t_1 and t_2 are α-equivalent terms then* $\mathsf{fv}(t_1) = \mathsf{fv}(t_2)$ *and if $u_1 = \sigma t_1$ and $u_2 = \sigma t_2$ are both defined then $u_1 =_\alpha u_2$.*

Proof. The proofs are by straightforward inductions.

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

3 Reduction

Reduction proceeds in two stages: first generate a match and then apply it. The richness of the class of patterns introduces some complexity to the matching process, even in the handling of variables.

The most common situation is that the free variables of a pattern are binding variables and so ready to be matched, as in $(x \rightarrow_x x)\ \bullet$ or $(x\ y \rightarrow_{\{x,y\}} y)\ (\bullet\ \bullet)$ where x and y both bind to $\bullet$. In pattern polymorphic examples such as the generic update a pattern may contain a free variable that is awaiting substitution. Then matching of the pattern must be delayed until the value of the variable is known, as in $x \rightarrow (x\ y \rightarrow_y y)\ (\bullet\ \bullet)$. There is, however, a third possibility which is illustrated by the following closed term

$$t = ((x \rightarrow_{\{\}} x)\ x \rightarrow_x x)\ (\bullet\ \bullet). \tag{2}$$

The pattern p given by $(x \rightarrow_{\{\}} x)\ x$ contains a free variable x which cannot be replaced by substitution, as it is a binding variable of t. Hence there is no way that p can ever be reduced and so it is natural to treat it as a compound data structure in order to match against its parts. Then the match of $x \rightarrow_{\{\}} x$ against $\bullet$ will fail so that matching fails rather than gets stuck, as desired.

The difficulty with this approach is in determining that p is irreducible within t. Of course, this depends upon the status of x, so the notion of data structure must now be parametrised by a set of "fixed" variables φ and some means found of characterising the irreducible applications such as p. The easiest way we have found is to define the φ-data structures in terms of irreducibility of their parts, which in turn depends upon the data structures within them, in a virtuous cycle.

One can also consider matching of cases. Although feasible, it would require that matching be parametrised by yet another set of variables, representing those which are bound within the pattern itself, so is left for another occasion. In this paper, case matching will always fail.

Finally, there is a substantial literature concerning the appropriate treatment of non-linear patterns, i.e. those in which a binding variable x appears twice, such as $x\ x$. One approach would be to allow $x\ x$ to match with terms of the form $u\ u$. However, this may cause a loss of confluence, as in [FK03,Kah03], for reasons grounded in Klop's observation that the combination of untyped λ-calculus with *non left-linear* first-order rewriting systems breaks confluence [Klo80]. Since non-linear patterns cannot be banned (any term can be a pattern) our solution is to prevent $x\ x$ from matching anything, even a term of the form $u\ u$. This is handled implicitly by the requirement that the union of substitutions is only defined when their domains are disjoint.

3.1 Matching

The definitions of data structures, matchable forms, matching and reduction that follow are all mutually recursive. That is, the definition of reduction depends upon that of data structures which in turn depends on the irreducibility of some sub-terms.

Let φ be a set of variables. Define the *φ-data structures* (meta-variable d) by

$$
\begin{array}{lll}
d ::= & x & \text{if } x \in \varphi \\
 & \bullet & \\
 & (p \to_\theta s)\ u & \text{if } (p \to_\theta s)\ u \text{ is irreducible and all its free variables are in } \varphi \\
 & d\ u. &
\end{array}
$$

For example, $\bullet\ u$ is a φ-data structure for any term u. Also, $(x \to_{\{\}} x)\ x$ will prove to be irreducible, and so to be a $\{x\}$-data structure.

Define the *data structures* to be the $\{\}$-data structures. The *φ-matchable forms* are the φ-data structures and all cases. The *matchable forms* are the $\{\}$-matchable forms.

The *basic matching* $\{p /\!\!/ u\}_\theta$ of a term p (called the *pattern*) against a term u (called the *argument*) relative to a set θ of *binding variables* is the partial operation defined by the following equations

$$
\begin{array}{rcll}
\{x /\!\!/ u\}_\theta & = & \{x/u\} & \text{if } x \in \theta \\
\{\bullet /\!\!/ \bullet\}_\theta & = & \{\} & \\
\{q\ p /\!\!/ v\ u\}_\theta & = & \{q /\!\!/ v\}_\theta \uplus \{p /\!\!/ u\}_\theta & \text{if } q\ p \text{ is a } \theta\text{-matchable form} \\
 & & & \text{and } v\ u \text{ is a matchable form} \\
\{p \to_\psi s /\!\!/ u\}_\theta & = & \mathsf{none} & \\
\{p /\!\!/ u\}_\theta & = & \mathsf{none} & \text{if } p \text{ is a } \theta\text{-matchable form} \\
 & & & \text{and } u \text{ is a matchable form} \\
\{p /\!\!/ u\} & = & \text{undefined} & \text{otherwise}
\end{array}
$$

where the equations are to be applied in order. That is, matching is always defined if the pattern is a θ-matchable form and the argument is a matchable form, and match failure can only arise if rules for successful matching do not apply. A binding variable matches anything. The constructor matches itself. Matching of compound data structures is component-wise, using (disjoint) union. Note that the ordering of the equations can be avoided by expanding the definition into an induction on the structure of the pattern.

For example, evaluation of t in (2) will use the match $m = \{(x \to_{\{\}} x)\ x /\!\!/ \bullet\ \bullet\}_{\{x\}}$. Now $\{x /\!\!/ x\}_{\{\}}$ is undefined and so $(x \to_{\{\}} x)\ x$ turns out to be irreducible and thus is an $\{x\}$-data structure. Hence m matches $x \to_{\{\}} x$ against $\bullet$ but fails, and so m is none.

Let p and u be terms and let θ be a set of variables. Define the *matching $\{p/u\}_\theta$ of p against u with respect to binding variables θ* to be the check of $\{p /\!\!/ u\}_\theta$ on θ, where the check of a match is the function defined in Section 2. The check is necessary to ensure that reduction does not allow bound variables to become free. For example, $\{x/\bullet\}_{\{x,y\}} = \mathsf{none}$ since $\{x /\!\!/ \bullet\}_{\{x,y\}}$ is not defined on y.

The pure pattern calculus has a *match rule* given by

$$(p \rightarrow_\theta s)\ u \longmapsto \{p/u\}_\theta\ s. \tag{3}$$

That is, if matching of the pattern against the argument produces a substitution whose domain is the binding variables then apply this to the body. If the matching fails then return the identity function. Of course, if $\{p/u\}_\theta$ is undefined (e.g. because p or u needs to be reduced) then the match rule does not apply.

The *one-step reduction relation* $\longrightarrow$ is defined by applying the match rule to a sub-term. The *reduction relation* $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$. A term t is *irreducible* if there is no reduction of the form $t \longrightarrow t'$.

4 Examples

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 $x \rightarrow_x s$ or $x \rightarrow s$. Pattern-matching for these terms is exactly the β-reduction of the λ-calculus. For example, the *fixpoint* term

$$\mathsf{fix} = (x \rightarrow f \rightarrow f\ (x\ x\ f))\ (x \rightarrow f \rightarrow f\ (x\ x\ f))$$

can be used to define recursive functions. A term definition of the form $\mathsf{letrec}\ x = t$ will be interpreted as giving f the value $\mathsf{fix}\ (x \rightarrow t)$ in the usual way.

Branching constructs Let $\mathsf{False} = \bullet$ and $\mathsf{True} = \bullet\ \bullet$. Define conditionals by

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

where $x \notin \mathsf{fv}(s)$. Thus, $\mathsf{if}\ \mathsf{True}\ \mathsf{then}\ s\ \mathsf{else}\ r$ reduces to $(x \rightarrow s)\ r$ and then to s while $\mathsf{if}\ \mathsf{False}\ \mathsf{then}\ s\ \mathsf{else}\ r$ reduces to $(y \rightarrow y)\ r$ and then to r. More generally, the *extension* $p \rightarrow_\theta s \mid r$ extends the case $p \rightarrow_\theta s$ with a *default* r by

$$p \rightarrow_\theta s \mid r = x \rightarrow (p \rightarrow_\theta y \rightarrow s)\ x\ (r\ x)$$

where $x \notin \mathsf{fv}(p \rightarrow_\theta s) \cup \mathsf{fv}(r)$ and $y \notin \mathsf{fv}(s)$. When applied to some term u it reduces to $\{p/u\}_\theta(y \rightarrow s)\ (r\ u)$. Now if $\{p/u\}_\theta$ is some substitution σ then this reduces to $\sigma(y \rightarrow s)\ (r\ u) = (y \rightarrow \sigma s)\ (r\ u)$ and then to σs as desired. Alternatively, if $\{p/u\}_\theta = \mathsf{none}$ then the term reduces to $(\mathsf{none}\ (y \rightarrow s))\ (r\ u) = (z \rightarrow z)\ (r\ u)$ and then to $r\ u$ as desired.

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

$$\begin{array}{rl} & p_1 \rightarrow s_1 \\ \mid & p_2 \rightarrow s_2 \\ & \vdots \\ \mid & p_n \rightarrow s_n \end{array}$$

is $p_1 \to s_1 \mid (p_2 \to s_2 \mid (\ldots \mid p_n \to s_n))$. For example, the function $\mathsf{listPoints}$ in the introduction is defined in this way.

Arithmetic The natural numbers can be defined as data structures built from constructors $\mathsf{Zero} = \bullet$ and $\mathsf{Successor} = \bullet$. Then recursive functions can be defined using fix. This compares favourably with the representation of numbers as the iterators used to define the Church numerals.

Constructors In practice, it is natural to add a family of constructors c to the calculus, to represent truth values, numbers, lists etc. Each constructor is an atomic data structure that can be applied to an arbitrary number of arguments to produce compound data structures. The match rule $\{c /\!\!/ c\}_\theta = \{\}$ generalises that for $\bullet$.

Generic equality Now let us consider some novel programs. A generic equality is defined by

$$\mathsf{equal} = x \to (x \to_{\{\}} \mathsf{True} \mid y \to \mathsf{False})$$

where the first argument is used as the pattern for matching against the second. For example, $\mathsf{equal}\ (\bullet\ (\bullet\ \bullet))\ (\bullet\ (\bullet\ \bullet))$ reduces to True. This is a simple example of *pattern polymorphism* where the pattern is created dynamically.

The generic eliminator The generic eliminator is given by

$$\mathsf{elim} = x \to x\ y \to_y y$$

For example, $\mathsf{elim}\ \mathsf{Successor}$ reduces to $\mathsf{Successor}\ y \to y$. Again, suppose that the list constructors Nil and Cons are defined to be some data structures built from $\bullet$. Given $\mathsf{singleton} = x \to \mathsf{Cons}\ x\ \mathsf{Nil}$ then $\mathsf{elim}\ \mathsf{singleton}$ reduces to $\mathsf{Cons}\ y\ \mathsf{Nil} \to y$ by reduction of the pattern.

Generic updating Patterns of the form $x\ y$ are used to access data along arbitrary paths through a data structure, i.e. to support *path polymorphism*. Combining the use of pattern and path polymorphism yields the generic update function defined in the introduction. When applied to a constructor c, and 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, $\mathsf{update}\ c\ f\ ((c\ u)\ (c\ v))$ reduces to $(c\ (f\ u))\ (c\ (f\ v))$. In general, update can be applied to cases. For example, $\mathsf{update}\ \mathsf{singleton}\ f$ reduces to

$$\begin{array}{lll} & \mathsf{Cons}\ y\ \mathsf{Nil} & \to \mathsf{Cons}\ (f\ y)\ \mathsf{Nil} \\ \mid & z\ y & \to (\mathsf{update}\ \mathsf{singleton}\ f\ z)\ (\mathsf{update}\ \mathsf{singleton}\ f\ y) \\ \mid & y & \to y \end{array}$$

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

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

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 [HJS05a,HJS05b].

Wild-cards It is interesting to add a new constant denoted ? to the pure pattern calculus, the *wild-card*. It has no free variables and is unaffected by substitution. It is a data structure, is compatible with anything, and has the matching rule

$$\{? /\!\!/ u\}_\theta = \{\}.$$

That is, it behaves like a fresh binding variable in a pattern but like a constructor in a body. For example, the first and second projections from a pair can be encoded as $\mathsf{elim}\ (\mathsf{Pair}\ ?)$ and $\mathsf{elim}\ (x \to \mathsf{Pair}\ x\ ?)$.

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

$$\begin{array}{l} \mathsf{letrec\ entrypattern} = \\ \quad \mathsf{Succ}\ n \to x \to \mathsf{Cons}\ ?\ (\mathsf{entrypattern}\ n\ x) \\ \mid\ \mathsf{Zero} \quad\ \to x \to \mathsf{Cons}\ x\ ? \\ \\ \mathsf{entry} = n \to \mathsf{elim}\ (\mathsf{entrypattern}\ n) \end{array}$$

For example, $\mathsf{entry}\ (\mathsf{Succ}\ (\mathsf{Succ\ Zero}))$ reduces to $\mathsf{Cons}\ ?\ (\mathsf{Cons}\ ?\ (\mathsf{Cons}\ x\ ?)) \to x$ which recovers the second entry from a list. Note the standard approach, in which each occurrence of the wild-card represents a distinct binding variable, cannot support such recursion.

5 Progress

Lemma 2. *An irreducible term t is a $\mathsf{fv}(t)$-matchable form.*

Proof. The proof is by a straightforward induction on the structure of t. If t is of the form $(p \to_\theta s)\ u$ then it is either reducible or a $\mathsf{fv}(t)$-data structure.

Theorem 1. *Pattern-matching cannot get stuck. That is, a closed term of the form $(p \to_\theta s)\ u$ is always reducible.*

Proof. Assume that p and u are irreducible. Then p is a θ-matchable form and u is a matchable form by Lemma 2. Hence $\{p/u\}_\theta$ is defined and the whole term can be reduced.

Corollary 1. *The data structures are those terms headed by the constructor.*

6 Confluence

Confluence of reduction is established using the parallel reduction technique due to Tait and Martin-Löf [Bar84] which can be summarised in four steps: define a parallel reduction relation denoted $\gg$; prove that $\gg^*$ and $\longrightarrow^*$ are the same relation (Lemma 3); show that $\gg$ has the diamond property (Theorem 2); and use this to prove confluence. The *parallel reduction relation* is given in Figure 1.

$$\frac{}{t \gg t} \qquad \frac{r \gg r' \quad u \gg u'}{r\ u \gg r'\ u'} \qquad \frac{p \gg p' \quad s \gg s'}{p \rightarrow_\theta s \gg p' \rightarrow_\theta s'} \qquad \frac{p \gg p' \quad s \gg s' \quad u \gg u'}{(p \rightarrow_\theta s)\ u \gg \{p'/u'\}_\theta\ s'}$$

Fig. 1. Parallel reduction

Lemma 3. *Every one-step reduction is a parallel reduction. Also, every parallel reduction is a reduction. Hence the reflexive-transitive closure $\gg^*$ of $\gg$ is the reduction relation $\longrightarrow^*$.*

Proof. The proofs are by straightforward induction on the definitions.

The *parallel reduction relation* $\gg$ between matches is defined as follows. Given two substitutions σ and σ' then $\sigma \gg \sigma'$ if $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$ and $\sigma x \gg \sigma' x$ for every $x \in \mathsf{dom}(\sigma)$. Also $\mathsf{none} \gg \mathsf{none}$. Substitutions and none are not related.

Lemma 4. *If t is a term and m is a match then $\mathsf{fv}(mt) \subseteq \mathsf{fv}(m) \cup (\mathsf{fv}(t) \setminus \mathsf{dom}(m))$.*

Proof. If m is none then the result is immediate so assume that m is a substitution σ. The proof is by induction on the structure of t. If t is $p \rightarrow_\theta s$ where $\theta \cap \mathsf{var}(\sigma) = \{\}$ then

$$\begin{aligned} \mathsf{fv}(\sigma p \rightarrow_\theta \sigma s) &= (\mathsf{fv}(\sigma p) \cup \mathsf{fv}(\sigma s)) \setminus \theta \\ &\subseteq (\mathsf{fv}(\sigma) \cup ((\mathsf{fv}(p) \cup \mathsf{fv}(s)) \setminus \mathsf{dom}(\sigma))) \setminus \theta \quad \text{(by induction)} \\ &= \mathsf{fv}(\sigma) \cup (\mathsf{fv}(t) \setminus \mathsf{dom}(\sigma)). \end{aligned}$$

The other cases are straightforward.

Lemma 5. *If $m = \{p/u\}_\theta$ for some terms p and u and set of variables θ then $\mathsf{fv}(m) \subseteq \mathsf{fv}(u)$.*

Proof. If $m = \mathsf{none}$ then there is nothing to prove. Otherwise the proof is by a straightforward induction on the structure of p.

Lemma 6. *If $t \gg t'$ is a parallel reduction then $\mathsf{fv}(t') \subseteq \mathsf{fv}(t)$. Hence, if $m \gg m'$ is a parallel match reduction then $\mathsf{var}(m') \subseteq \mathsf{var}(m)$.*

Proof. By Lemma 3 it suffices to prove the result for the reduction rule (3). Then

$$\begin{aligned} \mathsf{fv}(t') &= \mathsf{fv}(\{p/u\}_\theta\ s) \\ &\subseteq \mathsf{fv}(\{p/u\}_\theta) \cup (\mathsf{fv}(s) \setminus \mathsf{dom}(\{p/u\}_\theta)) \quad \text{(Lemma 4)} \\ &\subseteq \mathsf{fv}(\{p/u\}_\theta) \cup (\mathsf{fv}(s) \setminus \theta) \\ &\subseteq \mathsf{fv}(u) \cup (\mathsf{fv}(p \rightarrow_\theta s)) \quad \text{(Lemma 5)} \\ &= \mathsf{fv}(t). \end{aligned}$$

Lemma 7. *Let m be a match and let θ be a set of variables such that $\mathsf{var}(m) \cap \theta = \{\}$. If p and u are terms such that $\{p /\!\!/ u\}_\theta$ is defined then so is $\{m\ p /\!\!/ m\ u\}_\theta$ and $\{m\ p /\!\!/ m\ u\}_\theta \circ m = m \circ \{p /\!\!/ u\}_\theta$. Hence*

$$\{m\ p/m\ u\}_\theta \circ m = m \circ \{p/u\}_\theta.$$

Proof. The second statement follows directly from the first. If m is none then both sides are none. So without loss of generality, assume that m is a substitution σ. The proof is by induction on the structure of p. If p is a variable $x \in \theta$ then both sides map x to σu and behave as σ on all other variables. If p is the constructor and u is too then both sides are m. If p and u are compatible applications then apply induction twice.

If $\{p /\!\!/ u\}_\theta = \mathsf{none}$ then $\{\sigma p /\!\!/ \sigma u\}_\theta = \mathsf{none}$ (since $\mathsf{dom}(\sigma) \cap \theta = \{\}$) and so both sides of the match equation are none.

Lemma 8. *If $p \gg p'$ and $u \gg u'$ are parallel reductions on terms and $\{p/u\}_\theta$ is defined then so is $\{p'/u'\}_\theta$ and $\{p/u\}_\theta \gg \{p'/u'\}_\theta$.*

Proof. The proof is by induction on the structure of p. If p is a variable then p' is the same variable and it must be in θ so that the result follows directly. If p is the constructor then u is the constructor if and only if u' is. If p is a case then both matches will fail. Otherwise p must be a θ-matchable form $p_1\ p_2$ and u must be a matchable form. If u is not an application then it must be the constructor or a case: either way, both matchings will fail. Alternatively, if $u = u_1\ u_2$ then Corollary 1 implies that u_1 is also a data structure and thus $u' = u_1'\ u_2'$ where $u_1 \gg u_1'$ and $u_2 \gg u_2'$. We then have two possibilities for p_1. If p_1 is a θ-data structure then $p' = p_1'\ p_2'$ where $p_1 \gg p_1'$ and $p_2 \gg p_2'$. Hence induction applies. Otherwise, p_1 is a case so that p is irreducible and both matches against it will fail.

Lemma 9. *If $m \gg m'$ and $t \gg t'$ are parallel reductions of matches and terms respectively then $m\ t \gg m'\ t'$.*

Proof. If m is none then m' is none and so the result is immediate. So assume that m and m' are substitutions σ and σ' respectively. The proof is by induction on the derivation of $t \gg t'$. The only non-trivial case is when $t = (p \rightarrow_\theta s)\ u \gg \{p'/u'\}_\theta\ s'$ where $p \gg p'$ and $u \gg u'$ and $s \gg s'$. Without loss of generality, assume $\mathsf{var}(\sigma) \cap \theta = \{\}$. By Lemma 6 and by the assumptions we have $\mathsf{var}(\sigma') \cap \theta \subseteq \mathsf{var}(\sigma) \cap \theta = \{\}$. Thus, $\sigma'(\{p'/u'\}_\theta s')$ is equal to $\{\sigma'\ p'/\sigma'\ u'\}_\theta(\sigma'\ s')$ by Lemma 7 and so $\sigma((p \rightarrow_\theta s)u) = (\sigma\ p \rightarrow_\theta \sigma s)\ (\sigma\ u) \gg \{\sigma' p'/\sigma'\ u'\}_\theta(\sigma'\ s') = \sigma'(\{p'/u'\}_\theta s')$.

Theorem 2. *The relation $\gg$ has the diamond property. That is, $t \gg t_1$ and $t \gg t_2$ then there is t_3 such that $t_1 \gg t_3$ and $t_2 \gg t_3$.*

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

$$(p_2 \rightarrow_\theta s_2)\ u_2\ \ll (p \rightarrow_\theta s)\ u \gg \{p_1/u_1\}_\theta s_1$$

where $p \gg p_1$ and $p \gg p_2$ and $s \gg s_1$ and $s \gg s_2$ and $u \gg u_1$ and $u \gg u_2$. By induction, there are terms p_3, s_3 and u_3 such that $p_1 \gg p_3$ and $p_2 \gg p_3$ and $s_1 \gg s_3$ and $s_2 \gg s_3$ and $u_1 \gg u_3$ and $u_2 \gg u_3$. Now $\{p_1/u_1\}_\theta \gg \{p_3/u_3\}_\theta$ by Lemma 8 and so $\{p_1/u_1\}_\theta s_1 \gg \{p_3/u_3\}_\theta s_3$ by Lemma 9. Hence, the diamond is completed by $\{p_3/u_3\}_\theta s_3$.

Again, suppose $(p \rightarrow_\theta s)\ u \gg \{p_2/u_2\}_\theta s_2$ and $(p \rightarrow_\theta s)\ u \gg \{p_1/u_1\}_\theta s_1$ where $p \gg p_1$ and $p \gg p_2$ and $s \gg s_1$ and $s \gg s_2$ and $u \gg u_1$ and $u \gg u_2$. By induction there are terms p_3, s_3 and u_3 such that $p_1 \gg p_3$ and $p_2 \gg p_3$ and $s_1 \gg s_3$ and $s_2 \gg s_3$ and $u_1 \gg u_3$ and $u_2 \gg u_3$. Now $\{p_1/u_1\}_\theta$ and $\{p_2/u_2\}_\theta$ both parallel reduce to $\{p_3/u_3\}_\theta$ by Lemma 8 and so Lemma 9 implies the diamond is completed by $\{p_3/u_3\}_\theta s_3$. The other cases are straightforward.

Corollary 2 (Confluence). *The reduction relation is confluent.*

Proof. Theorem 2 shows that $\gg$ has the diamond property and so the reflexive-transitive closure of $\gg$ is confluent. Now apply Lemma 3.

7 Conclusion and Further Work

Pattern-matching provides a natural mechanism for computing with data structures; its expressive power is determined by the nature of the patterns that are allowed. The pure pattern calculus maximises this expressive power by allowing any term to be a pattern. The resulting language supports patterns that are able to match with arbitrary compound data structures (path polymorphism), and patterns that can be assembled dynamically (using free variables to represent patterns) and simplified into a matchable form (pattern polymorphism). Such patterns will prove useful when manipulating remote data whose structure is only partially known, as illustrated by the example of updating.

There are a number of open questions concerning the pure pattern calculus itself, and its connections to rewriting, logic, type theory and category theory.

The matching process may extend to consider matching of cases as well as of data structures, provided the binding variables of cases are treated appropriately. We have not pursued this here as the complexity of the development was not justified by any new forms of polymorphism. However, it may prove useful in program analysis and transformation.

It is not yet clear what extensional equality should be for the pattern calculus, as earlier work on extensionally for pattern-matching [Kes97] does not take full account of data structures. For example, the η-equality rule $f = \lambda x.f\ x$ does *not* apply in our setting since a data structure is not a case.

Another issue concerns higher-order rewriting within a formalism with patterns [FK03]. It seems natural to extend such languages to capture the rich dynamics of the patterns presented here.

In the spirit of [KPT96] it would be interesting to explore a Curry-Howard interpretation for the pure pattern calculus in order to recognise, or develop, the corresponding logic. For example, matching against arbitrary compounds appears to model structural induction [Bur69] in a uniform way.

The calculus presented here uses a meta-level (or implicit) pattern-matching operation. One could also consider explicit pattern-matching, where the match equations become themselves rewriting rules which can then be interleaved with other reductions [CK99,For02,Kah03,Jay04c].

It is straightforward to provide simple types and indeed to support parametric polymorphism. Of more interest will be the addition of type specialisations [Jay04c] necessary to type the more complex examples. The calculus will then provide a clean foundation for a typed account of XML paths, as described in [HJS05a,HJS05b] and a platform upon which to model object-orientation, along the lines proposed in [Jay04b].

The denotational semantics of the pattern calculus also awaits exploration. It is not clear how to represent a case in a domain-theoretic setting. As a lambda-abstraction is an arrow in a category then perhaps a case is a *span* in a category or, rather, the internalisation of a span.

In conclusion, the pure pattern calculus provides a compact setting in which to handle both functions and data structures in a uniform manner, and so support new forms of polymorphism.

References

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. Revised Edition.

[BCKL03] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure Pattern Type Systems. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 250–261. ACM, 2003.

[BdMH96] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[Bur69] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 1969.

[CK98] Horatiu Cirstea and Claude Kirchner. ρ-calculus, the rewriting calculus. In *5th International Workshop on Constraints in Computational Logics (CCL)*, 1998.

[CK99] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 98–108. IEEE Computer Society Press, 1999.

[FK03] Julien Forest and Delia Kesner. Expression reduction systems with patterns. In Robert Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2003.

[For02] Julien Forest. A weak calculus with explicit operators for pattern matching and substitution. In Sophie Tison, editor, *13th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2378 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, 2002.

[GJ03] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11-12,2002, Dagstuhl, Germany*. Kluwer Academic Publishers, 2003.

[Has] The Haskell language. `http://www.haskell.org/`.

[HJS05a] Freeman Yufei Huang, C. Barry Jay, and David B. Skillicorn. Dealing with complex patterns in XML processing. Technical Report 2005-497, Queen's University School of Computing, 2005.

[HJS05b] Freeman Yufei Huang, C. Barry Jay, and David B. Skillicorn. Programming with heterogeneous structures: Manipulating XML data using bondi. Technical Report 2005-494, Queen's University School of Computing, 2005.

[Jan00] Patrick Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

[Jay04a] C. Barry Jay. Higher-order patterns. Available as `www-staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf`, 2004.

[Jay04b] C. Barry Jay. Methods as pattern-matching functions. In Sophia Drossopoulou, editor, *The 11th International Workshop on Foundations of Object-Oriented Languages*, 2004. Proc. available as `http://www.doc.ic.ac.uk/~scd/FOO.pdf`.

[Jay04c] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.

[JBM98] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[JC94] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer Verlag, 1994.

[Kah03] Wolfram Kahl. Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation. Technical Report 16, Software Quality Research Laboratory, McMaster Univ., 2003.

[Kes97] Delia Kesner. Reasoning about redundant patterns. *Journal of Functional and Logic Programming*, 1997(4), June 1997.

[Klo80] Jan-Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematical Centre Tracts 127, CWI, Amsterdam, 1980.

[KPT96] Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.

[Oca] The Objective Caml language. `http://caml.inria.fr/`.

[SML] Standard ML of New Jersey. `http://cm.bell-labs.com/cm/cs/what/smlnj/`.