Patterns as first-class citizens

C. Barry Jay, Delia Kesner

Patterns as first class citizens

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 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*

$$
\begin{aligned}
&\text{letrec listpoints} = \\
&\quad \text{point } y && \hookrightarrow [\text{point } y] \\
&\quad |\ x\ y && \hookrightarrow (\text{listpoints } x)\ @\ (\text{listpoints } y) \\
&\quad |\ y && \hookrightarrow [\,]
\end{aligned}
$$

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 listpoints is applied to a pair pair $s\ t$ of points s and t we get

$$
\begin{aligned}
\text{listpoints (pair } s\ t) &= ((\text{listpoints pair})\ @\ (\text{listpoints } s))\ @\ (\text{listpoints } t) \\
&= ([\,]\ @\ [s])\ @\ [t] \\
&= [s, t]
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{letrec updatepoint} = \ f \hookrightarrow \\
&\quad \text{point } y && \hookrightarrow \text{point } (f\ y) \\
&\quad |\ z\ y && \hookrightarrow (\text{updatepoint } f\ z)\ (\text{updatepoint } f\ y) \\
&\quad |\ y && \hookrightarrow y
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{letrec update} = \quad x \hookrightarrow f \hookrightarrow \\
&\quad x\ y && \hookrightarrow_x x\ (f\ y) \\
&\quad |\ z\ y && \hookrightarrow\ (\text{update } x\ f\ z)\ (\text{update } x\ f\ y) \\
&\quad |\ y && \hookrightarrow\ y
\end{aligned}
$$

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 the free variable x. 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

$$t ::= x \mid t\,t \mid [\theta]\,t \to t$$

consisting of variables, applications and cases $[\theta]\,p \to s$ where θ is a set of variables, its *binding variables*. In defining reduction one must first specify a set of variables γ which are to play the role of *constructors*. The sole reduction rule is motivated by the equation

$$([\theta]\,p \to s)\ u =_\gamma \{u/[\theta]\ p\}_\gamma\ \ s \tag{1}$$

where γ is disjoint from θ. Here $\{u/[\theta]\ p\}_\gamma$ is the *match* of p against u that produces either a substitution with domain θ or a failure.

It may be surprising to identify the constructors with a set of variables, but it proves convenient when reducing the pattern of a case $[\theta]\,p \to s$ since the variables in θ are considered to be constructors when reducing the pattern p. For example, $[x]\,([\{\ \}]\,x \to x)\,x \to x$ reduces to $[x]\,x \to x$ since x is constructor within the pattern $([\{\ \}]\,x \to x)\,x$.

The pure pattern calculus is well behaved. In particular, every closed term of the form $([\theta]\,p \to s)\,u$ 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. A general approach introducing λ-terms as patterns can be found in [vO90], where confluence is proved for terms verifying the *rigid pattern condition*, a technique which is also applied in the context of ρ-calculi [CK98,BCKL03]. Certification of meta-properties of pattern-matching calculi has been developed in [Kah03]. The patterns of our calculus also generalise those of the concurrent language Linda [Gel85] in which patterns are tuples containing a mix of free and binding occurences. Later research 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.

The last of these underpinned the development of typed calculi supporting pattern polymorphism (e.g. [Jay04a]) which allow free variables in patterns, and

their reduction but without allowing arbitrary terms as patterns. 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]. Finally, the "scrap-your-boilerplate" approach [syb06] supports some key cases of path polymorphism.

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. In Section 7 we present an alternative account of first-class patterns as proposed in [JK06]. Section 8 draws conclusions and considers further work.

2 Terms

Fix a countable alphabet of variables (meta-variables $\ldots x, y, z$). Let φ, θ and γ denote finite sets of variables and θ, γ denote the disjoint union of such sets.

Definition 1 (Terms). *The term syntax of the* pure pattern calculus *is given by the following grammar:*

$$
\begin{array}{llll}
t ::= & & \textit{(terms)} & \\
& x & \textit{(variable)} & | \\
& t\ t & \textit{(application)} & | \\
& [\theta]\, t \to t & \textit{(case)} &
\end{array}
$$

The notations $[x_1, \ldots, x_n]\, p \to s$ and $[\ ]\, p \to s$ will be used in place of $[\{x_1, \ldots, x_n\}]\, p \to s$ and $[\{\}]\, t \to s$ to avoid unnecessary brackets.

The variables are available for binding, matching and substitution. The application $r\, u$ applies the *function* r to its *argument* u. The case $[\theta]\, p \to s$ 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 $[x]\, x \to [y]\, x\ y\ z \to y$ is equal to $[x]\, x \to ([y]\, ((x\ y)\ z) \to y)$. Lambda-abstraction can be defined by setting $\lambda x.t$ to be $[x]\, x \to t$.

4

Definition 2 (Free and bound Variables). Free and bound *of terms are defined by:*

$$
\begin{aligned}
&\mathsf{fv}(x) &&:= \{x\} &&\mathsf{bv}(x) &&:= \{\} \\
&\mathsf{fv}(r\ u) &&:= \mathsf{fv}(r) \cup \mathsf{fv}(u) &&\mathsf{bv}(r\ u) &&:= \mathsf{bv}(r) \cup \mathsf{bv}(u) \\
&\mathsf{fv}([\theta]\, p \rightarrow s) := (\mathsf{fv}(p) \cup \mathsf{fv}(s)) \setminus \theta &&\mathsf{bv}([\theta]\, p \rightarrow s) := \mathsf{bv}(p) \cup \mathsf{bv}(s) \cup \theta.
\end{aligned}
$$

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 $p \hookrightarrow s$ stands for $[\mathsf{fv}(p)]\, p \rightarrow s$ (so that in particular a lambda-abstraction $\lambda x.t$ can also be written as $x \hookrightarrow t$). More generally, $p \hookrightarrow_\varphi s$ stands for $[\mathsf{fv}(p) \setminus \varphi]\, p \rightarrow s$.

Hence programmers need never actually mention binding variables explicitly unless they require free variables in the pattern.

2.1 Matches

A *substitution* (meta-variable σ) is a partial function from variables to terms. The notation $\{u_1/x_1, \ldots, u_n/x_n\}$ represents the finite substitution that maps x_i to u_i for $i = 1 \ldots n$ and $\{\}$ is the empty substitution.

A *match* (meta-variable μ) is either a *successful match*, given by a substitution σ, or a *failure*, written as `none`.

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 `none` is the empty set. If $\mathsf{dom}(\mu)$ and $\mathsf{dom}(\mu')$ are disjoint we write $\mu \# \mu'$. 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 μ to be $\mathsf{var}(\mu) = \mathsf{dom}(\mu) \cup \mathsf{fv}(\mu)$. We use the predicate μ `avoids` x to mean $x \notin \mathsf{var}(\mu)$. More generally, μ avoids θ if it avoids each variable in θ and avoids a term t if it avoids $\mathsf{fv}(t)$.

Definition 3 (Applying a substitution). *The* application *of a substitution σ to a term is defined by*

$$
\begin{aligned}
&\sigma x &&:= \sigma x &&\text{if } x \in \mathsf{dom}(\sigma) \\
&\sigma x &&:= x &&\text{if } x \notin \mathsf{dom}(\sigma) \\
&\sigma(r\ u) &&:= (\sigma r)\ (\sigma u) \\
&\sigma([\theta]\, p \rightarrow s) &&:= [\theta]\, \sigma p \rightarrow \sigma s &&\text{if } \sigma\ \texttt{avoids}\ \theta
\end{aligned}
$$

The restriction on the definition of $\sigma([\theta]\, p \rightarrow s)$ is necessary to avoid a *variable clash* which would change the semantics of the term. Thus for example $\{y/x\}([y]\, y \rightarrow x)$ would change the status of the free variable x in the body to a bound variable y while $\{y/x\}([x]\, x \rightarrow y)$ would change the bound variable x in the pattern to a free one. 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

$$\texttt{none } t := x \hookrightarrow x.$$

Definition 4 (Disjoint union of matches). *The* disjoint union $\mu_1 \uplus \mu_2$ *of matches* μ_1 *and* μ_2 *is defined as follows. If either of them is* none *or their domains have a non-empty intersection then their disjoint union is* none*. Otherwise, it is the substitution given by*

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

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 μ_1 and μ_2 are matches of which at least one is none then $\mu_2 \circ \mu_1$ is defined to be none.

Definition 5 (Check). *The* check μ_θ *of a match* μ *on a set of variables* θ *is* μ *if* μ *is a substitution whose domain is exactly* θ *and is* none *otherwise.*

Checks will be used to ensure that variables do not escape their scope in next section.

2.2 Alpha conversion

Let θ be a set of variables and x and y be variables. Then $\{y/x\}\theta$ 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

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

For example, $[y]\, x\ y \to x\ (t\ y) =_\alpha [z]\, x\ z \to x\ (t\ z)$ if z is not free in t.

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.

Note that although it is easy to avoid variable clashes by α-conversion, it is a little harder to decide equality of terms since there are n^2 ways of aligning two sets of n binding variables. If this is important then the definitions can be modified to employ lists of binding variables.

3   Reduction

In all well-known pattern-matching calculi reduction proceeds in two stages: first generate a match and then apply it. The difference between different languages resides in different notions of matching functions. As the patterns of our calculus contain free variables, the notion of matching has to be carefully defined in order to guarantee that substitutions generated by the match operation will only affect the set of binding variables of the case.

3.1   Matching

Basic matching is defined using two parameters φ and θ, the first being the set of constructors and the second one the set of binding variables. A key notion used in the definition of basic matching is the one of φ-matchable form, intuitively, those terms that are ready to be matched.

Definition 6 (Data Structures and Matchable Forms). *The φ-data structures (meta-variable d) and φ-matchable forms (meta-variable e) are given by the following grammar:*

$$d ::= x \ (x \in \varphi) \mid d \ t$$
$$e ::= d \mid [\theta] \, t \to t$$

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

Definition 7 (Basic Matching). *The* basic matching $\{\!\{u \triangleright_\theta \ p\}\!\}_\gamma$ *of a term p (called the* pattern*) against a term u (called the* argument*) relative to a set θ of* binding variables *and a disjoint set γ of* constructing variables *(or* constructors*) is the partial operation defined by applying the following equations in order*

$\{\!\{u \triangleright_\theta \ x\}\!\}_\gamma$	$:= \{u/x\}$	if $x \in \theta$
$\{\!\{x \triangleright_\theta \ x\}\!\}_\gamma$	$:= \{\}$	if $x \in \gamma$
$\{\!\{v \ u \triangleright_\theta \ q \ p\}\!\}_\gamma$	$:= \{\!\{v \triangleright_\theta \ q\}\!\}_\gamma \ \uplus \ \{\!\{u \triangleright_\theta \ p\}\!\}_\gamma$	if $q \ p$ is a γ, θ-matchable form and $v \ u$ is a γ-matchable form
$\{\!\{u \triangleright_\theta \ p\}\!\}_\gamma$	$:= \mathtt{none}$	if p is a γ, θ-matchable form and u is a γ-matchable form
$\{\!\{u \triangleright_\theta \ p\}\!\}_\gamma$	$:= \text{undefined}$	otherwise.

That is, matching is always defined if the pattern is a γ, θ-matchable form and the argument is a γ-matchable form, and match failure can only arise if rules for successful matching do not apply. A binding variable matches anything. A 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.

The use of disjoint unions when matching compound patterns means that matching against a compound such as $c \ x \ x$ can never succeed. Since non-linear

patterns cannot be banned (any term can be a pattern), the alternative would be to allow it to match with terms of the form $c\ u\ u$. However, this may cause a loss of confluence, as in [FK03,Kah03], for reasons grounded in Klop's observation [Klo80] that the combination of untyped λ-calculus with *non left-linear* first-order rewriting systems breaks confluence.

As defined, matching one case against another always fails. It should be possible to support this without too much effort, but it it not neccessary for the sorts of data manipulations that motivated this work.

Definition 8 (Matching). *Let p and u be terms and let θ and γ be disjoint sets of variables. Define the* matching $\{u/[\theta]\ p\}_\gamma$ *of p against u with respect to binding variables θ and constructors γ to be the check of $\{\!\{u \triangleright_\theta\ p\}\!\}_\gamma$ on θ, where the check of a match is the function in Definition 5.*

The check is necessary to ensure that reduction does not allow bound variables to become free. For example, $\{\!\{x \triangleright_y\ x\}\!\}_{\{x\}} = \{\}$ but $\{x/[y]\ x\}_{\{x\}} = \texttt{none}$ since the basic matching is not defined on y.

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

$$([\theta]\ p \to s)\ u >_\gamma \{u/[\theta]\ p\}_\gamma\ s \tag{2}$$

parametrised by the choice of constructors γ. 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 $\{u/[\theta]\ p\}_\gamma$ is undefined (e.g. because p or u needs to be reduced) then the match rule does not apply.

$$\frac{}{([\theta]\ p \to s)\ u \blacktriangleright_\gamma \{u/[\theta]\ p\}_\gamma\ s} \qquad \frac{r \blacktriangleright_\gamma r'}{r\ u \blacktriangleright_\gamma r'\ u} \qquad \frac{u \blacktriangleright_\gamma u'}{r\ u \blacktriangleright_\gamma r\ u'}$$

$$\frac{p \blacktriangleright_{\gamma,\theta} p'}{[\theta]\ p \to s \blacktriangleright_\gamma [\theta]\ p' \to s} \qquad \frac{s \blacktriangleright_\gamma s'}{[\theta]\ p \to s \blacktriangleright_\gamma [\theta]\ p \to s'}$$

Fig. 1. One-step reduction

The *one-step reduction relation* $\blacktriangleright_\gamma$ is defined by the rules in Figure 1. The *reduction relation* $\blacktriangleright_\gamma^*$ is the reflexive-transitive closure of $\blacktriangleright_\gamma$. A term t is γ-*irreducible* if there is no reduction of the form $t \blacktriangleright_\gamma t'$.

The key point is that the binding variables of a case become constructors when reducing the pattern. For example,

$$[x]\ ([\ ]\ x \to x)\ x \to x \blacktriangleright_{\{\}} [x]\ x \to x$$

since the binding variable x becomes a constructor when reducing $(([\ ]\ x \to x)\ x)$.

8

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 $[x]\,x \rightarrow s$
or $[\;]\,x \rightarrow s$. Pattern-matching for these terms with respect to any set of con-
structors is exactly the β-reduction of the λ-calculus. For example, the *fixpoint*
term

$$\mathsf{fix} := (x \hookrightarrow f \hookrightarrow f\ (x\ x\ f))\ (x \hookrightarrow f \hookrightarrow f\ (x\ x\ f))$$

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 $\mathsf{fix}\ (x \hookrightarrow t)$ in the usual way.

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 pair of a term p and a set of term variables γ, and
perform reduction relative to γ. Equivalently, one may define a program to be a
term of the form

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

where $\bullet \in \gamma$, with reduction relative to the empty set of variables.

Branching constructs Let true and false be constructors and define condition-
als by

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ r := (\mathsf{true} \hookrightarrow x \hookrightarrow s)\ b\ r$$

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

$$[\theta]\,p \rightarrow s \mid r := x \hookrightarrow ([\theta]\,p \rightarrow y \hookrightarrow s)\ x\ (r\ x)$$

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 $\{\!\{u \rhd_\theta\ p\}\!\}_\gamma\ (y \hookrightarrow s)\ (r\ u)$. Now if $\{\!\{u \rhd_\theta\ p\}\!\}_\gamma$ is some substitution σ
then this reduces to $\sigma(y \hookrightarrow s)\ (r\ u) = (y \hookrightarrow \sigma s)\ (r\ u)$ and then to σs as desired.
Alternatively, if $\{\!\{u \rhd_\theta\ p\}\!\}_\gamma = \mathtt{none}$ then the term reduces to $(\mathtt{none}\ (y \hookrightarrow s))\ (r\ u) = (z \hookrightarrow z)\ (r\ u)$ 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

$$\begin{aligned}
&[\theta_1]\,p_1 \rightarrow s_1 \\
\mid\ &[\theta_2]\,p_2 \rightarrow s_2 \\
&\vdots \\
\mid\ &[\theta_n]\,p_n \rightarrow s_n
\end{aligned}$$

is $[\theta_1]\,p_1 \rightarrow s_1 \mid ([\theta_2]\,p_2 \rightarrow s_2 \mid (\ldots \mid [\theta_n]\,p_n \rightarrow s_n))$. For example, the functions
$\mathsf{listpoints}$ and update in the introduction are defined in this way.

Arithmetic The natural numbers can be defined as data structures built from
constructors zero and $\mathsf{successor}$. 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.

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

$$\mathsf{equal} := x \hookrightarrow (x \hookrightarrow_x \mathsf{true} \mid y \hookrightarrow \mathsf{false})$$

where the first argument is used as the pattern for matching against the second. For example, $\mathsf{equal}\ (\mathsf{successor}\ \mathsf{zero})\ (\mathsf{successor}\ \mathsf{zero})$ 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

$$\mathsf{elim} := x \hookrightarrow x\ y \hookrightarrow_x y$$

For example, $\mathsf{elim}\ \mathsf{successor}$ reduces to $\mathsf{successor}\ y \hookrightarrow y$. Again, suppose that the list constructors nil and cons are given and define $\mathsf{singleton} := x \hookrightarrow \mathsf{cons}\ x\ \mathsf{nil}$. Then $\mathsf{elim}\ \mathsf{singleton}$ reduces to $\mathsf{cons}\ y\ \mathsf{nil} \hookrightarrow y$ 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

$$
\begin{array}{ll}
\mathsf{cons}\ y\ \mathsf{nil} & \hookrightarrow \mathsf{cons}\ (f\ y)\ \mathsf{nil} \\
\mid\ z\ y & \hookrightarrow (\mathsf{update}\ \mathsf{singleton}\ f\ z)\ (\mathsf{update}\ \mathsf{singleton}\ f\ y) \\
\mid\ y & \hookrightarrow y
\end{array}
$$

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

$$\{\!\!\{u \rhd_\theta\ ?\}\!\!\}_\gamma\ := \{\}$$

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

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

$$
\begin{aligned}
&\mathsf{letrec\ entrypattern} = \\
&\quad \mathsf{succ}\ n \hookrightarrow x \hookrightarrow \mathsf{cons}\ ?\ (\mathsf{entrypattern}\ n\ x) \\
&\quad |\ \mathsf{zero} \quad \hookrightarrow x \hookrightarrow \mathsf{cons}\ x\ ? \\
\\
&\mathsf{entry} = n \hookrightarrow \mathsf{elim}\ (\mathsf{entrypattern}\ n)
\end{aligned}
$$

For example, $\mathsf{entry}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))$ reduces to $\mathsf{cons}\ ?\ (\mathsf{cons}\ ?\ (\mathsf{cons}\ x\ ?)) \hookrightarrow x$ 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 Properties of reduction

This section establishes some basic properties of the reduction relation introduced in Figure 1.

Lemma 2. *If $\gamma \subseteq \varphi$, then $t \blacktriangleright_\gamma t'$ implies $t \blacktriangleright_\varphi t'$.*

Proof. The proof is by induction on the reduction relation. The interesting case is that of the match rule for which we use two facts :

- If u is a γ-matchable form, then it is also a φ-matchable form.
- If $\{\!\!\{ u \rhd_\theta\ p \}\!\!\}_\gamma$ is defined, then $\{\!\!\{ u \rhd_\theta\ p \}\!\!\}_\varphi$ is also defined.

Define contexts by the following grammar:

$$C ::= \square \mid C\ t \mid t\ C \mid [\theta]\ C \to t \mid [\theta]\ t \to C$$

where $\square$ is a constant.

The replacement of $\square$ by a term t in a context C is written $C[t]$.

A relation R is *closed under contexts* if $t\ R\ u$ implies $C[t]\ R\ C[u]$ for any context C.

Lemma 3. *The reduction relation $\blacktriangleright_\gamma$ is stable under substitution for variables not in γ and is closed under contexts.*

Proof. Stability of reduction under substitutions follows by a straightforward induction. Stability under contexts can be shown by induction. The interesting case consists in showing that $p \blacktriangleright_\gamma p'$ implies $[\theta]\ p \to s \blacktriangleright_\gamma [\theta]\ p' \to s$ for which we use Lemma 2 and definition of reduction.

Theorem 1. *Let t be a term whose free variables are all in some set γ. If t is γ-irreducible then t is a γ-matchable form. Hence, pattern-matching cannot get stuck.*

Proof. The proof is by induction on the structure of t. We only consider here t of the form $([\theta]\ p \to s)\ u$ as all the other cases are straightforward. Now u is γ-irreducible, and p is γ, θ-irreducible and so, by induction, u is γ-matchable and p is γ, θ-matchable. Hence the basic matching of p against u is defined and so t is γ-reducible, contradicting the assumption.

6 Confluence

Confluence of reduction is established using the simultaneous reduction technique due to Tait and Martin-Löf [Bar84] which can be summarised in four steps: define a simultaneous reduction relation denoted $\Rightarrow_\gamma$; prove that $\Rightarrow_\gamma^*$ and $\blacktriangleright_\gamma^*$ are the same relation (Lemma 4); show that $\Rightarrow_\gamma^*$ has the diamond property (Theorem 2); and use this to prove confluence.

Let γ be a set of variables. The *simultaneous γ-reduction relation* is given in Figure 2.

$$\frac{}{t \Rightarrow_\gamma t} \qquad\qquad \frac{r \Rightarrow_\gamma r' \quad u \Rightarrow_\gamma u'}{r\ u \Rightarrow_\gamma r'\ u'}$$

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

Fig. 2. Simultaneous γ-reduction

Lemma 4. *Every one-step γ-reduction is a simultaneous γ-reduction. Also, every simultaneous γ-reduction is a γ-reduction. Hence the reflexive-transitive closure $\Rightarrow_\gamma^*$ of $\Rightarrow_\gamma$ is the reduction relation $\blacktriangleright_\gamma^*$.*

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

The *simultaneous γ-reduction relation* $\Rightarrow_\gamma$ between matches is defined as follows. Given two substitutions σ and σ' then $\sigma \Rightarrow_\gamma \sigma'$ if $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$ and $\sigma x \Rightarrow_\gamma \sigma' x$ for every $x \in \mathsf{dom}(\sigma)$. We define also $\mathtt{none} \Rightarrow_\gamma \mathtt{none}$. Substitutions and $\mathtt{none}$ are not related.

Lemma 5. *If t is a term and μ is a match then $\mathsf{fv}(\mu\ t) \subseteq \mathsf{fv}(\mu) \cup (\mathsf{fv}(t) \setminus \mathsf{dom}(\mu))$.*

Proof. If μ is $\mathtt{none}$ then the result is immediate since $\mathsf{fv}(\mathtt{none}\ t)$ is exactly $\mathsf{fv}([x]\, x \to x)$ which is the empty set. 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 σ $\mathtt{avoids}$ θ then

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

The other cases are straightforward.

A simple example can be given by the term $t = x\ y\ w$ and the match $\{t/x, u/y, v/z\}$, for which we have $\mathsf{fv}(t\ u\ w) \subseteq \mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v) \cup \{w\}$.

Lemma 6. *If $\mu = \{\!\{u \rhd_\theta\ p\}\!\}_\gamma$ for some terms p and u and disjoint sets of variables γ and θ then $\mathsf{fv}(\mu) \subseteq \mathsf{fv}(u)$.*

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

For example, $\mu = \{\!\{t\ v\ w \rhd_{x,y}\ x\ y\ w\}\!\}_w = \{t/x, v/y\}$ we have $\mathsf{fv}(\mu) = \mathsf{fv}(t) \cup \mathsf{fv}(v) \subseteq \mathsf{fv}(t) \cup \mathsf{fv}(v) \cup \{w\} = \mathsf{fv}(t\ v\ w)$.

Exactly as in λ-calculus, reduction and simultaneous reduction in the pure pattern calculus preserve the set of free variables of terms but may lose some of them. A simple example of this can be given by the simultaneous reduction step $t = ([x, y]\ x\ y \to y)\ (z\ w) \Rrightarrow_{\{\}} w = t'$ which loses the free variable z. Formally,

Lemma 7. *If $t \Rrightarrow_\gamma t'$ is a simultaneous reduction then $\mathsf{fv}(t') \subseteq \mathsf{fv}(t)$. Hence, if $\mu \Rrightarrow_\gamma \mu'$ is a simultaneous reduction between matches, then $\mathsf{var}(\mu') \subseteq \mathsf{var}(\mu)$.*

Proof. By Lemma 4 it suffices to prove the result for the one-step reduction relation; we only show here the case of the reduction rule (2). Then

$$
\begin{aligned}
\mathsf{fv}(t') &= \mathsf{fv}(\{\!\{u \rhd_\theta\ p\}\!\}_\gamma\ s) \\
&\subseteq \mathsf{fv}(\{\!\{u \rhd_\theta\ p\}\!\}_\gamma) \cup (\mathsf{fv}(s) \setminus \mathsf{dom}(\{\!\{u \rhd_\theta\ p\}\!\}_\gamma)) \quad \text{(Lemma 5)} \\
&\subseteq \mathsf{fv}(\{\!\{u \rhd_\theta\ p\}\!\}_\gamma) \cup (\mathsf{fv}(s) \setminus \theta) \\
&\subseteq \mathsf{fv}(u) \cup (\mathsf{fv}([\theta]\ p \to s)) \quad\quad\quad\quad\quad \text{(Lemma 6)} \\
&= \mathsf{fv}(t).
\end{aligned}
$$

Generalising the Substitution Lemma of λ-calculus [Bar84] yields the following lemma.

Lemma 8. *Let μ be a match and let θ and γ be two disjoint sets of variables such that $\mu\ \mathtt{avoids}\ \theta$ and $\mathsf{dom}(\mu) \cap \gamma = \{\}$. If p and u are terms such that $\{\!\{u \rhd_\theta\ p\}\!\}_\gamma$ is defined then so is $\{\!\{\mu\ u \rhd_\theta\ \mu\ p\}\!\}_\gamma$ and $\{\!\{\mu\ u \rhd_\theta\ \mu\ p\}\!\}_\gamma \circ \mu = \mu \circ \{\!\{u \rhd_\theta\ p\}\!\}_\gamma$. Hence*

$$
\{\mu\ u/[\theta]\ \mu\ p\}_\gamma \circ \mu = \mu \circ \{u/[\theta]\ p\}_\gamma .
$$

Proof. The second statement follows directly from the first which we analyse in detail. If μ is $\mathtt{none}$ then both sides of the equation are $\mathtt{none}$. So without loss of generality, assume that μ 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 of the equation map x to σu and behave as σ on all other variables since $\mathsf{var}(\sigma) \cap \theta = \{\}$. If p is in γ and u is the same then both sides of the equation are μ. If p and u are compatible applications then apply the induction hypothesis twice. This requires that μ $\mathtt{avoids}\ \theta$.

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

As expected, basic matching is stable under reduction. This property can be specified by the following general statement concerning simultaneous reduction which includes by definition reduction.

Lemma 9. *If $p \Rightarrow_{\gamma,\theta} p'$ and $u \Rightarrow_\gamma u'$ are simultaneous reductions on terms and $\{\!\{u \rhd_\theta \ p\}\!\}_\gamma$ is defined then so is $\{\!\{u' \rhd_\theta \ p'\}\!\}_\gamma$ and $\{\!\{u \rhd_\theta \ p\}\!\}_\gamma \Rightarrow_\gamma \{\!\{u' \rhd_\theta \ p'\}\!\}_\gamma$.*

Proof. The proof is by induction on the structure of p. If p is a variable then p' is the same variable so that the result follows directly. If p is a case then both matches will fail. Otherwise p must be a γ,θ-matchable form $p_1 \ p_2$ and u must be a γ-matchable form. If u is not an application then it must be a constructor or a case: either way, both matchings will fail. Alternatively, if $u = u_1 \ u_2$ then Theorem 1 implies that u_1 is also a γ-data structure and thus $u' = u_1' \ u_2'$ where $u_1 \Rightarrow_\gamma u_1'$ and $u_2 \Rightarrow_\gamma u_2'$. Now p_1 is a γ,θ-data structure and so $p' = p_1' \ p_2'$ where $p_1 \Rightarrow_{\gamma,\theta} p_1'$ and $p_2 \Rightarrow_{\gamma,\theta} p_2'$. Hence the induction hypothesis applies.

Simultaneous reduction (and thus reduction) is stable under substitution:

Lemma 10. *If $\mu \Rightarrow_\gamma \mu'$ and $t \Rightarrow_\gamma t'$ are simultaneous reductions of matches and terms respectively and $\mathsf{dom}(\mu) \cap \gamma = \{\}$ then $\mu \ t \Rightarrow_\gamma \mu' \ t'$.*

Proof. If μ is **none** then μ' is **none** and so 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 $t \Rightarrow_\gamma t'$. The only non-trivial case is when $t = ([\theta] \ p \to s) \ u \Rightarrow_\gamma \{\!\{u' \rhd_\theta \ p'\}\!\}_\gamma \ s'$ where $p \Rightarrow_{\gamma,\theta} p'$ and $u \Rightarrow_\gamma u'$ and $s \Rightarrow_\gamma s'$. Without loss of generality, assume $\mathsf{var}(\sigma) \cap \theta = \{\}$ by α-conversion. Hence $\mathsf{var}(\sigma') \cap \theta \subseteq \mathsf{var}(\sigma) \cap \theta = \{\}$ by Lemma 7 and $\mathsf{dom}(m') \cap \gamma = \mathsf{dom}(m) \cap \gamma = \{\}$. Thus, $\sigma'(\{\!\{u' \rhd_\theta \ p'\}\!\}_\gamma \ s')$ is equal to $\{\!\{\sigma' \ u' \rhd_\theta \ \sigma' \ p'\}\!\}_\gamma \ (\sigma' \ s')$ by Lemma 8 and so $\sigma(([\theta] \ p \to s)u) = ([\theta] \ \sigma \ p \to \sigma s) \ (\sigma \ u) \Rightarrow_\gamma \{\!\{\sigma' \ u' \rhd_\theta \ \sigma' \ p'\}\!\}_\gamma \ (\sigma' \ s') = \sigma'(\{\!\{u' \rhd_\theta \ p'\}\!\}_\gamma \ s')$.

In order to get confluence of the reduction relation $\blacktriangleright_\gamma$ we first show that $\Rightarrow_\gamma$ has the diamond property, then we conclude by using the fact that $\Rightarrow_\gamma^*$ and $\blacktriangleright_\gamma^*$ are the same relation so that confluence of one implies confluence of the other one.

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

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

$$([\theta] \ p_2 \to s_2) \ u_2 \ {}_\gamma\!\!\Leftarrow ([\theta] \ p \to s) \ u \Rightarrow_\gamma \{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma \ s_1$$

where $p \Rightarrow_{\gamma,\theta} p_1$ and $p \Rightarrow_{\gamma,\theta} p_2$ and $s \Rightarrow_\gamma s_1$ and $s \Rightarrow_\gamma s_2$ and $u \Rightarrow_\gamma u_1$ and $u \Rightarrow_\gamma u_2$. By the induction hypothesis, there are terms p_3, s_3 and u_3 such that $p_1 \Rightarrow_{\gamma,\theta} p_3$ and $p_2 \Rightarrow_{\gamma,\theta} p_3$ and $s_1 \Rightarrow_\gamma s_3$ and $s_2 \Rightarrow_\gamma s_3$ and $u_1 \Rightarrow_\gamma u_3$ and $u_2 \Rightarrow_\gamma u_3$. Now $\{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma \Rightarrow_\gamma \{\!\{u_3 \rhd_\theta \ p_3\}\!\}_\gamma$ by Lemma 9 and so $\{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma \ s_1 \Rightarrow_\gamma \{\!\{u_3 \rhd_\theta \ p_3\}\!\}_\gamma \ s_3$ by Lemma 10 since $\mathsf{dom}(\{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma)$ does not contain γ by construction. Hence, the diamond is completed by $\{\!\{u_3 \rhd_\theta \ p_3\}\!\}_\gamma \ s_3$.

Again, suppose

$$\{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma \ s_1 \ _\gamma\!\Lleftarrow ([\theta]\, p \to s) \ u \Rrightarrow_\gamma \{\!\{u_2 \rhd_\theta \ p_2\}\!\}_\gamma \ s_2$$

where $p \Rrightarrow_{\gamma,\theta} p_1$ and $p \Rrightarrow_{\gamma,\theta} p_2$ and $s \Rrightarrow_\gamma s_1$ and $s \Rrightarrow_\gamma s_2$ and $u \Rrightarrow_\gamma u_1$ and $u \Rrightarrow_\gamma u_2$. By the induction hypothesis there are terms p_3, s_3 and u_3 such that $p_1 \Rrightarrow_{\gamma,\theta} p_3$ and $p_2 \Rrightarrow_{\gamma,\theta} p_3$ and $s_1 \Rrightarrow_\gamma s_3$ and $s_2 \Rrightarrow_\gamma s_3$ and $u_1 \Rrightarrow_\gamma u_3$ and $u_2 \Rrightarrow_\gamma u_3$. Now $\{\!\{u_1 \rhd_\theta \ p_1\}\!\}_\gamma$ and $\{\!\{u_2 \rhd_\theta \ p_2\}\!\}_\gamma$ both simultaneous reduce to $\{\!\{u_3 \rhd_\theta \ p_3\}\!\}_\gamma$ by Lemma 9 and so Lemma 10 implies the diamond is completed by $\{\!\{u_3 \rhd_\theta \ p_3\}\!\}_\gamma \ s_3$. The other cases are straightforward.

Corollary 1 (Confluence). *The reduction relation* $\blacktriangleright_\gamma$ *is confluent.*

Proof. Theorem 2 shows that $\Rrightarrow_\gamma$ has the diamond property and so the reflexive-transitive closure of $\Rrightarrow_\gamma$ is confluent (as can be spelled out by a straightforward induction). Now apply Lemma 4.

7 A second solution

This section presents an alternative account of first-class patterns [JK06]. From a syntactical point of view the main difference between this second solution and the one presented before is that the set of constructors does not increase dynamically during reduction as it is just defined once for all. So, one globally fixes a set of symbols to play the role of constructors. For simplicity we only consider here a single constructor $\bullet$.

Terms are now generated by the following grammar

$$
\begin{array}{lll}
t ::= & & \text{(terms)} \\
& x & \text{(variable)} \quad \mid \\
& \bullet & \text{(constructor)} \mid \\
& t\ t & \text{(application)} \mid \\
& [\theta]\, t \to t & \text{(case)}
\end{array}
$$

We add to Definition 2 the case $\mathsf{fv}(\bullet) := \{\}$ and to Definition 3 the case $\sigma\bullet := \bullet$.

Now, the most common situation is that the free variables of a pattern are binding variables and so ready to be matched, as in $(x \hookrightarrow x)\ \bullet$ or $(x\ y \hookrightarrow y)\ (\bullet\ \bullet)$ where x and y both bind to $\bullet$. In various examples that we gave a pattern may contain a free variable that is awaiting substitution. Then matching of the pattern must be delayed untile the value of the (free) variable is known, as in $x \hookrightarrow ([y]\, x\ y \to y)\ (\bullet\ \bullet)$. There is even a third possibility which is illustrated by the following *closed* term

$$t = ([x]\, ([\,]\, x \to x)\ x \to x)\ (\bullet\ \bullet) \tag{3}$$

The pattern p given by $([\,]\, x \to x)\ x$ 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 $([\,]\, x \to x)\, x$ against $(\bullet\ \bullet)$ will fail since the (subcase) match of $([\,]\, x \to x)$ against $\bullet$ will also fail. In this way, the matching does not get stuck.

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 needs to characterise the irreducible applications such as $([\,]\, x \to x)\, x$. We thus define the notion of data structure in terms of irreducibility of their parts, which in turn depends upon the data structures within them, in a virtuous, but well-founded cycle. Thus, data structures and reduction are *mutually recursive* definitions whose well-foundedness will be argued after their introduction.

Definition 9 (Data Structures and Matchable Forms). *The φ-data structures (meta-variable d) are defined as follows:*

- *The constant $\bullet$ is a φ-data structure.*
- *Every $x \in \varphi$ is a φ-data structure.*
- *If d is a φ-data structure and u a term, then $d\, u$ is a φ-data structure.*
- *If $([\theta]\, p \to s)\, u$ is an irreducible term and all its free variables are in φ, then the term is a φ-data structure.*

Thus for example, $\bullet\ \bullet$ is a φ-data structure. Also $([\,]\, x \to x)\, x$ will be proved to be irreducible, and so to be a $\{x\}$-data structure.

The φ-matchable forms are given by the grammar:

$$e ::= d \mid [\theta]\, t \to u$$

As before, define the data structures *(resp.* matchable forms*) to be the $\{\}$-data structures (resp. $\{\}$-matchable forms).*

Definition 10 (Basic Matching). *The* basic matching $\{\!\{u \rhd_\theta\ p\}\!\}$ *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 applying the following equations in order*

$$
\begin{aligned}
\{\!\{u \rhd_\theta\ x\}\!\} &:= \{u/x\} && \text{if } x \in \theta \\
\{\!\{\bullet \rhd_\theta\ \bullet\}\!\} &:= \{\} && \\
\{\!\{v\ u \rhd_\theta\ q\ p\}\!\} &:= \{\!\{v \rhd_\theta\ q\}\!\}\ \uplus\ \{\!\{u \rhd_\theta\ p\}\!\} && \text{if } q\,p \text{ is a } \theta\text{-matchable form} \\
&&& \text{and } v\,u \text{ is a matchable form} \\
\{\!\{u \rhd_\theta\ p\}\!\} &:= \mathtt{none} && \text{if } p \text{ is a } \theta\text{-matchable form} \\
&&& \text{and } u \text{ is a matchable form} \\
\{\!\{u \rhd_\theta\ p\}\!\} &:= \text{undefined} && \text{otherwise.}
\end{aligned}
$$

For example, evaluation of t in (3) will use the match

$$\sigma = \{\!\{(\bullet\ \bullet) \rhd_x\ ([\,]\, x \to x)\, x\}\!\}$$

Now, $\{\!\{x \rhd_{\{\}}\ x\}\!\}$ is undefined so that $([\,]\, x \to x)\, x$ turns out to be irreducible and thus it is a $\{x\}$-data structure. Hence the matching of $[\,]\, x \to x$ against $\bullet$ is calculated, and proves to be $\mathtt{none}$, and so σ is also $\mathtt{none}$ and t is reducible.

Definition 11 (Matching). *Let p and u be terms and let θ be a set of variables. Define the* matching $\{u/[\theta]\ p\}$ *of p against u with respect to binding variables θ to be the check of $\{\!\{u \triangleright_\theta\ p\}\!\}$ on θ, where now the check of a match m on a set of variables φ is m if m is a substitution whose domain is exactly φ and* none *otherwise.*

The new *match rule* is now given by

$$([\theta]\ p \rightarrow s)\ u > \{u/[\theta]\ p\}\ s \tag{4}$$

The new *one-step reduction relation* $\blacktriangleright$ is defined by the rules in Figure 3.

$$
\frac{}{([\theta]\ p \rightarrow s)\ u \blacktriangleright \{u/[\theta]\ p\}\ s} \qquad \frac{r \blacktriangleright r'}{r\ u \blacktriangleright r'\ u} \qquad \frac{u \blacktriangleright u'}{r\ u \blacktriangleright r\ u'}
$$

$$
\frac{p \blacktriangleright p'}{[\theta]\ p \rightarrow s \ \blacktriangleright\ [\theta]\ p' \rightarrow s} \qquad \frac{s \blacktriangleright s'}{[\theta]\ p \rightarrow s \ \blacktriangleright\ [\theta]\ p \rightarrow s'}
$$

Fig. 3. New one-step reduction

Now, remark that a term t is reducible if one of its subterms is reducible or if $t = ([\theta]\ p \rightarrow s)\ u$ and $\{u/[\theta]\ p\}$ is defined. The check $\{u/[\theta]\ p\}$ is defined if it is none or if it is a substitution $\{\!\{u \triangleright_\theta\ p\}\!\}$ whose domain is equal to θ. Now, $\{\!\{u \triangleright_\theta\ p\}\!\}$ is a substitution if one of the cases of the definition hold, in particular if p is a θ-matchable form (a case or a θ-data structure) and u is a matchable form (a case or a data structure). To check this on p and u we would eventually need to check if they are reducible or not, but they are subterms of the original term t.

Summing up, a simple induction on terms allows us to show that the property of being reducible is well-defined. Thus, Definitions 9, 10 and 11 together with Figure 3 give sound mutual recursive notions.

As expected, this formalism also gives a sound solution to the quest of a language with path and pattern polymorphism. More precisely,

Theorem 3 (Confluence). *For each γ the reduction relation $\blacktriangleright$ is confluent.*

We refer the reader to [JK06] for a detailed proof.

8   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.

Two related approaches were explored. The first solution identifies constructors with variables, and makes the notion of constructor context-dependent. This makes it easy to describe the data structures and matching, but now everything, including reduction, is parametrised by the choice of constructors. Such parametrised families of reduction relations are unfamiliar, and so there may be some surprises in store. The second solution fixes the constructors once and for all, so that there is only one reduction relation, but this one relation is defined by mutual recursion with that of the data structures, so that the overall account is actually more complicated. Overall, we prefer the first solution for its naturality: the difficulties of handling parametrised families of relations are probably less than those flowing from the mutual recursion; and the contextualisation of constructors is closer to programming practice.

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 yet 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.

[Gel85]   David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[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. To appear in ACSW'06.

[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.

[JK06]   Barry Jay and Delia Kesner. Pure pattern calculus. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (ed:P. Sestoft)*, pages 100–114, 2006. Available as `www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf`.

[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/`.

[syb06]   Scrap your boilerplate homepage. `http://www.cs.vu.nl/boilerplate/`, 2006.

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