

Pattern Calculus

Talk given to FPSyd

Barry Jay
University of Technology, Sydney
cbj@it.uts.edu.au

22nd October, 2009

Pattern calculus is a new foundation for computation based on pattern matching. This generalisation from substitution to matching combines the pure functionality of lambda-calculus with the ability to describe arbitrary data structures by patterns. In turn, this leads to new forms of program re-use, or polymorphism, such as **path polymorphism**, which may traverse all paths through a structure, and **pattern polymorphism**, in which patterns may be computed dynamically.

The talk will provide an overview of the speaker's recent monograph on pattern calculus

<http://www.springer.com/computer/foundations/book/978-3-540-89184-0> covering pure pattern calculus, typed pattern calculus, and its implementation in the bondi programming language <http://bondi.it.uts.edu.au/> here in Sydney.

Functions = Actions = Reason

but

Structure = Being = Experience

Knowledge comes from experience as well as reason.

Functions cannot provide a uniform treatment of data structures or the queries that can be made of them.

Different programming styles combine functions and data structures in various ways, but always with some compromise.

Pattern matching

Pattern matching combines functions and data structures without compromising either.

The pattern describes the data structure whose matching yields a substitution.

$$(p \rightarrow s)u \longrightarrow \{u/p\}s$$

Traditional pattern matching restricts the patterns to limit the expressive power to that of λ -calculus.

Dynamic pattern calculus allows *any* term to be a pattern, so that they are first class.

Five Sorts of Polymorphism in **bondi**

- parametric polymorphism (type variables, as in ML, Haskell, Generic Java)
- subtype polymorphism (as in Java)
- structure polymorphism (generic mapping, etc)
- path polymorphism (select, update, etc)
- pattern polymorphism (eliminators, etc)

The first two involve types, while the third concerns algebraic data type declarations: the last two are typeless, and will be considered here.

The Calculi of the Talk

λ -calculus	$t ::= x \mid t t \mid \lambda x.t$
compound calculus	$t ::= x \mid c \mid t t \mid \lambda x.t \mid \text{car } t \mid \text{cdr } t \mid \dots$
static patterns	$t ::= x \mid c \mid t t \mid p \rightarrow t$
dynamic patterns	$t ::= x \mid \hat{x} \mid t t \mid [\theta] t \rightarrow t$
SK -logic	$M ::= S \mid K \mid M M$
SF -logic	$M ::= S \mid F \mid M M$

Static pattern calculus supports path polymorphism.

Dynamic pattern calculus supports pattern polymorphism.

$$t ::= x \mid t t \mid \lambda x.t$$
$$(\lambda x.s)u \longrightarrow \{u/x\}s$$

where

$$\begin{aligned}\{u/x\}x &= u \\ \{u/x\}y &= y \text{ if } y \neq x \\ \{u/x\}(s t) &= (\{u/x\}s) (\{u/x\}t) \\ \{u/x\}(\lambda y.s) &= \lambda y.\{u/x\}s \text{ if } x \notin \text{fv}(s) \cup \{y\}\end{aligned}$$

Compound Calculus

Compound calculus is Lisp-like; both functions and data structures are handled uniformly.

$$t ::= x \mid t t \mid \lambda x. t \mid \text{car } t \mid \text{cdr } t \mid \text{pair? } t \mid t \text{ eqcons } t$$
$$(\lambda x. s) u \longrightarrow \{u/x\}s$$
$$\text{car}(u v) \longrightarrow u \text{ if } u v \text{ is a compound}$$
$$\text{cdr}(u v) \longrightarrow v \text{ if } u v \text{ is a compound}$$
$$\text{pair?}(u v) \longrightarrow \text{true if } u v \text{ is a compound}$$
$$\text{pair? } c \longrightarrow \text{false if } c \text{ is an atom}$$
$$c \text{ eqcons } c \longrightarrow \text{true}$$
$$s \text{ eqcons } t \longrightarrow \text{false otherwise, if } s \text{ and } t \text{ are matchable forms}$$

A *data structure* is a term whose head is a constructor c .

A *matchable form* is either a data structure or an abstraction.

A *compound* is a matchable form that is an application.

An *atom* is a matchable form that is not a compound.

Static Pattern Calculus

$$\begin{aligned} p & ::= x \mid c \mid p p \\ t & ::= x \mid c \mid t t \mid p \rightarrow s \\ (p \rightarrow s)u & \longrightarrow \{u/p\}s \end{aligned}$$

Traditional pattern matching requires that non-trivial patterns be headed by a constructor. Here, the pattern

$$x y$$

will match against an arbitrary compound.

$$\begin{aligned} \{u/x\} & = \text{Some}\{u/x\} \\ \{c/c\} & = \text{Some}\{\} \\ \{u v/p q\} & = \{u/p\} \uplus \{v/q\} \\ \{u/p\} & = \text{None otherwise, if } u \text{ is a matchable form} \\ \{u/p\} & = \text{undefined, otherwise.} \end{aligned}$$

The *matchable forms* are the data structures and cases $p \rightarrow s$.

Path Polymorphism in **bondi**

Let's look at some examples.

The standard account is that constructors are constants introduced by data type declarations.

What is their status in an untyped setting?

The traditional view is that they are λ -abstractions, e.g.

$$\text{pair} = \lambda x.\lambda y.\lambda f.f\ x\ y$$

Here, they are atoms used to seed data structures, but how are they introduced, controlled?

What is the status of binding symbols in a pattern?

Free Variables in Patterns

Dynamic patterns may contain *free* as well as *binding* variables.
Consider the *generic eliminator*

$\text{elim0} = x \rightarrow x \ y \rightarrow y$ (where is x bound, where free?)

Then

$$\begin{array}{lcl} \text{elim0 Leaf} & \longrightarrow & \text{Leaf } y \rightarrow y \\ \text{elim0 Leaf (Leaf 3)} & \longrightarrow & (\text{Leaf } y \rightarrow y) (\text{Leaf 3}) \\ & \longrightarrow & 3 \end{array}$$

In the pattern $x \ y$ above, the intent is that x is free (to become Leaf) while y is bound, but the syntax above does not distinguish them!

Explicit Binding Symbols

$$[\theta] p \rightarrow s$$

Let θ be the sequence of binding symbols. For example

$$\text{elim1} = [x] x \rightarrow [y] x y \rightarrow y \quad ?$$

This is good enough for **bondi**. Let's consider some examples.

A Pathological Example

This approach does easily not support confluence and progress.

Consider

$$\text{pathology} = [x]p \rightarrow x \text{ where } p = ([x \rightarrow x]) x .$$

Here p is not matchable, but it cannot reduce either, as its free variable x will *never* take a value, will never actually vary.

Matchable Symbols

Rename the variables as *symbols*.

A symbol x may appear as either a *variable symbol* (called x),
or as a *matchable symbol* (called \hat{x}).

Matchable symbols behave as constructors for patterns.

Identify constructors and matchable symbols!

$$\begin{aligned} \text{identity} &= [x] \hat{x} \rightarrow x \\ \text{elim} &= [x] \hat{x} \rightarrow [y] x \hat{y} \rightarrow y \\ \text{pathology} &= [x] ([] \hat{x} \rightarrow \hat{x}) \hat{x} \rightarrow x \\ &\longrightarrow \text{identity} . \end{aligned}$$

$$\begin{array}{l} t ::= x \mid \hat{x} \mid t t \mid [\theta] t \rightarrow t \\ ([\theta] p \rightarrow s) u \longrightarrow \{u/[\theta]p\}s \end{array}$$

No separate class of constructors.

No separate class of patterns.

Patterns are first class.

Pattern calculus can be encoded in λ -calculus, but only by converting terms to syntax trees. More precisely, there is no mapping from dynamic pattern calculus to lambda calculus that preserves equality and application.

This is not easy to see, since matchable symbols are strange, and the rules for matching have a lot of internal structure, so shift to combinatory logic.

Combinatory logic

$$M, N ::= A \mid M N$$

where A is a class of *operators*. Typical is

$$A ::= S \mid K$$

with reduction rules

$$\begin{aligned} SMNX &= MX(NX) \\ KXY &= X. \end{aligned}$$

The identity I can be represented by SKX for any X since

$$SKXY = KY(XY) = Y$$

Now consider a factorisation operator F with the reduction rules

$$\begin{aligned}FAMN &= M \\ F(PQ)MN &= NPQ \text{ if } PQ \text{ is a compound}\end{aligned}$$

The compounds are partially applied operators of the form SM , SMN , FM and FMN . Now F *cannot* be defined in terms of S and K since it can distinguish SKK and SKS .

Conclusions

Functions and structures should be equal partners in computation.

Pattern calculus supports this: patterns describe structures while matching supports functionality.

New forms of polymorphism, in path and pattern.

The whole calculus has a natural type system (and subtyped and classed, not shown) which supports three more sorts of polymorphism.

It is implemented in **bondi**, which supports all the main programming styles.

It is written up in my book <http://www.springer.com/computer/foundations/book/978-3-540-89184-0>.