Programming with Patterns

ACM SIGPLAN Developer Tracks on Functional Programming (DEFUN 2009)

Barry Jay
Centre for Quantum Computing and Intelligent Systems
University of Technology, Sydney

1:30 -5:00pm, 3rd September, 2009

The Pitch

- λ-calculus explores internal structure in an ad hoc manner
- but all data structures are either *atoms* or *compounds*.
- The pattern *xy* matches any compound in pattern calculus.
- Pattern calculus is strictly more expressive than λ-calculus,
- provides natural support for the main programming styles,
- and suggests some new ones (five sorts of polymorphism).
- Path polymorphism supports generic queries, e.g. `select`.
- Pattern polymorphism supports dynamic patt'ns (services).
- **bondi** implements pattern calculus.

The Back Story

Old start

The pattern calculus bases computation on pattern matching; as well as simplifying and unifying existing programming styles, it supports some new ones. In particular, a generic query can dynamically customise its pattern to suit the data structures it encounters. This is illustrated using examples in the **bondi** programming language. Then, after a glimpse at the pure pattern calculus, concluding remarks will speculate on the potential of this approach.

The Monograph

Pattern Calculus
Computing with Functions and Structures

Barry Jay

Springer monograph (March, 2009)
ISBN: 978-3-540-89184-0

"In this book the author will make you have second thoughts about the possibility and desirability of compiling away pattern matching ... It is amazing that the dynamic pattern calculus is syntactically almost as simple as the pure lambda-calculus, yet it is much more expressive." (Eugenio Moggi, U. di Genoa)

Updating Salaries

Problem Update all employee salaries within a company by some formula. The class of employees is known, but neither its sub-classes nor the company structure is known.
Solution Define a function

```
incrAllEmpSalary: (Float -> Float) -> a -> Command
```

Given a function `f:Float -> Float` for increasing salaries, and any data structure (presumably containing employee objects) of any type `a` it will increase all of the salaries. This exploits

Path Polymorphism

Let's take a look at the details in salaries.bon.

Points to Note (I)

- `incrAllEmpSalary` is a higher-order, polymorphic function, that
- takes proper account of the sub-class of managers, and
- finds employees within an arbitrary data structure
- while remaining strongly typed.

Reading Dates

Problem Provide a generic interface to dates that is able to act on a date pattern (a schema) and raw numerical information.
Solution Now there is no class to work with. Rather the pattern for the day-month-year information must itself be parametrised, in

```
~~let toDate = fun dateFormat ->
|{day,month,year} dateFormat day month year ->
    Date day month year;;
toDate: lin (Int->Int->Int->a) -> a -> Date
```

Here `dateFormat` is the parameter that describes the pattern holding the day, month and year integers. This exploits

Pattern Polymorphism

Let's take a look at the details in dates.bon.

Points to Note (II)

- Only *linear terms* can instantiate free variables in patterns, since otherwise binding symbols may be duplicated or lost.
- The type `Date` adds a case to the existing function `toString` which controls printing. This technique is central to the account of dynamic dispatch.
- The calculations necessary to handle the day-year format cannot be performed on the binding symbols `d`, `m` and `y` of the pattern. Hence a `view` is used to suspend pattern-matching while further calculations are performed upon the argument.

Collating Term Deposits

Problem Organise information about term deposits from multiple banks.
Solution This combines path and pattern polymorphism.
Let's take a look at the details in term_deposits.bon.

Points to Note (III)

- Ontological information is given by a parameter, `ont`.
- The banks do not have to change their repositories, or commit to some global standard, or a single service provider.
- The class `TD_service<a,b>` combines type parameters `a` and `b` with sub-typing.

Why Pattern Matching?

The need for programming styles is due to the tension between

- functions (searching, mapping, strategies) and
- data structures (arrays, records, trees).

This really hurts when trying to create powerful web services.

Pattern matching combines

- rich descriptions of structure (patterns)
- full functionality (matching).

Pattern calculus pushes patterns to their limits, e.g.

any term can be a pattern

so they can be discovered, combined and simplified on the fly.

Dynamic Pattern Calculus

$$
\begin{array}{lll}
t ::= & & \text{(term)} \\
 & x & \text{(variable symbol)} \\
 & \hat{x} & \text{(matchable symbol)} \\
 & t\ t & \text{(application)} \\
 & [\theta]\ t \to t & \text{(case).}
\end{array}
$$

Variable symbols can be instantiated; they *vary*.
Matchables symbols are used in pattern matching; they *match*.
θ is a sequence of symbols, the *binding symbols* of the case.
In $[\theta]\ p \to s$ any x in θ binds occurrences of $\hat{x}$ in p and of x in s.
If x is not bound then $\hat{x}$ is a *constructor*.
Symbols are related to channel *names* in pi-calculus.
Papers etc. can be found at
`http://www-staff.it.uts.edu.au/~cbj/patterns/`

A universal typed language?

bondi is an open source language

`http://www-staff.it.uts.edu.au/~cbj/bondi/`

It supports all the main styles.

- functional `let f x = x in f f`
- imperative `let x = Ref 3 in x = !x+1; !x`
- query `incrAllEmpSalary twopercent mycorp`
- object `class Employee extends Name`

The Expression Problem

Standard functional languages allow old data structures to support new functions. **bondi** also allows old functions to apply to new data structures. This solves the expression problem, and unifies method specialisation with function specialisation.

Smoother Sub-typing I

$$\frac{S < T}{P \to S < P \to T}$$

- Function result types are covariant (like other calculi but unlike Java).
- Function argument types are invariant (unlike other calculi, but like Java)
- Sub-typing and type parameters interact (unlike other calculi or Java).
- Now type inequalities $S < T$ must be solved (previously impossible).

Searching by Queries

Currently, Google searches are mainly by text (e.g. labels on images). To exploit data bases and repositories by generic queries requires:

- access to (views of) them (in the cloud?) AND
- users able to write formal queries (think SQL) OR
- a front end that converts natural language to formal queries

Web Services

- Once generic queries are supported then a spontaneous market is likely to develop in which service providers develop sophisticated queries for clients who need help.
- More concretely, applications for finding dates may be used immediately.

Softbots

Software negotiators need

- information (provided by generic queries) and
- strategies (provided by higher-order functions).

Similar remarks apply to automation of data mining.

Developing **bondi**

- **bondi** may eliminate the need for middleware.
- **bondi** may support typed inter-process communication
- **bondi** may expand to support a new model of concurrency.

Conclusions

- Pattern calculus is a new foundation for computing that eliminates the tension between functions and data structures.
- This unifies the main programming styles (and their polymorphism).
- It also supports new ones (generic queries, dynamic patterns).
- **bondi** shows how easy it is to implement.
- Data bases and repositories could form the next corpus.