Generic programming with XML

Barry Jay
University of Technology, Sydney

12th June, 2007

Pattern calculus

Pattern matching

- combines functions and data structures
- supports **5** forms of polymorphism (in type parameters, sub-typing, path, pattern and structure)
- supports all the usual programing styles (functional, imperative, object-oriented, relational, ...)
- requires new ideas about binding variables, constructors and typing
- is the subject of some seminars and an emerging monograph `www-staff.it.uts.edu.au/~cbj/draft-book/draft_chapters.pdf`
- is being implemented in **bondi**
- has a mailing list `pattern-calculus@ics.mq.edu.au`

XML

This talk will use pattern calculus to program with XML paths, to update

- an arbitrary data structure
- along an arbitrary XML path
- by an arbitrary function

Updating

Path and pattern polymorphism combine in the *generic update*

$$\text{update} : (X \to Y) \to (X \to X) \to Z \to Z.$$

For example, if f adds 2% to a floating point number and salary : float $\to$ salary is a salary constructor then

$$\text{update salary } f\ d$$

will update all salaries by 2% in a data structure d no matter where they are stored (in pairs, lists, trees, etc).

Some unusual patterns

The update program is given by

$$\begin{array}{l} \texttt{let update}\ x\ f = \\ \quad x\ \lambda z \rightarrow x\ (f\ z) \\ \mid\ \lambda y\ \lambda z \rightarrow \texttt{update}\ x\ f\ y\ (\texttt{update}\ x\ f\ z) \\ \mid \lambda z \rightarrow z. \end{array}$$

The *first case* has a pattern $x\ z$ in which x is free and z is bound. In `update salary` this reduces to the pattern `salary` λx. Free variables in patterns yield *pattern polymorphism*.
The *second case* has a pattern $\lambda y\ \lambda z$ made by applying one binding variable to another. It can match any *compound data structure* e.g. a pair or a non-empty list.
The *third case* will match any *atom*, e.g. the empty list.

The formalities

No, let's not.

See the draft book or the slides for technical details

The slogans

- Patterns are first class
- Special cases have special types

The technical tricks:

- binding variables = constructors = $\widehat{x}$ so that binders match themselves when reducing patterns
- separate binding from the patterns themselves:

$$\lambda x.s = \lambda x \rightarrow s = [x]\widehat{x} \rightarrow s$$

  so that reduction of patterns doesn't lose binders.
- combine cases $s : S$ and $r : R$ if S is a specialisation of R.

signPosts

Updating along an XML path is just like updating at a term, except that XML paths have more structure, so make an ADT for them.

```
datatype signPost
  at a b c =
  |Goal of c->b
  at (a1,a2) (b1,b2) c =
  |Stage of a1->b1 and signPost a2 b2 c
  |Detour of detourPath a1 b1 and signPost a2 b2 c


datatype detourPath
    at a b =
    | DetourGoal of a->b and a->bool
    at (a1,a2) (b1,b2) =
    | DetourStage of a1->b1 and detourPath a2 b2
```

These have since been described as Generalised ADTs.

Updates

```
let (checkd:(detourPath a b)->d->bool) p x =
  match p with
  | DetourGoal \P f -> check P f x
  | DetourStage \P p1 -> check P (checkd p1) x

let (updates:(signPost a b c)->(c->c)->d->d ) s f x =
  match s with
  | Goal \P -> update P f x
  | Stage \P s1 -> update P (updates s1 f) x
  | Detour dp1 s1 ->
          if (checkd dp1 x)  (* the detour *)
              then updates s1 f x
          else x
```

XPath, regular-expression style etc.

More complex patterns simply require more complex types (than signPosts), e.g.

```
datatype regexp
    at a b =
    | Single of a->b
    | Kstar of a->b
    at (a1,a2)(b1,b2)
    | Concat of regexp a1 b1 and regexp a2 b2
    | Altern of regexp a1 b1 and regexp a2 b2;;
```

encodes patterns of regular-expression style.

Conclusions

The challenge of programming with XML is pattern matching with

- a sophisticated approach to pattern matching
- a more sophisticated data type for representing paths.