Multi-polymorphic programming in bondi

Thomas Given-Wilson
NICTA[1]
Thomas.Given-Wilson@nicta.com.au

Freeman Huang
Queen's University, Canada
huang@cs.queensu.ca

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

Abstract

The **bondi** programming language is multi-polymorphic, in that it supports four polymorphic programming styles within a small core of computation, namely a typed pattern calculus. **bondi**'s expressive power is illustrated by considering the problem of assigning reviewers to a paper. As the context generalises from a committee to a committee with additional reviewers, to a conference, to a federation or confederation, the solution incorporates polymorphism familiar from the functional, generic functional, relational, path-based, and object-oriented programming styles, respectively. These experiments show that multi-polymorphic programming is both practical and desirable.

Keywords polymorphism, pattern matching, bondi, pattern calculus, generic programming

1. Introduction

The strength of each programming style emerges from the uniform treatment of its central concepts: functional programming treats functions; relational programming treats relations; path programming treats semi-structured data, such as XML; object-oriented programming treats objects. However, it is not so easy to combine these concepts within a single system: making all of these concepts equally fundamental produces a chimera, which undermines uniformity, while middleware must ignore all such concepts.

This paper shows that these concepts of functions, relations, paths, and objects, can be freely expressed and combined in a small programming language, **bondi** [1] based on *pattern calculus* [7, 9], in which computation evaluates pattern-matching functions. Since the proof is too big for the margin of the proceedings, we illustrate the approach and ideas through a motivating problem, implemented in **bondi** and supported by a sketch of the underlying theory presented in detail in "Pattern Calculus: Computing with Functions and Data Structures" [7]. This problem is to find reviewers for a paper submitted to a conference. The interest comes from considering how, and where the reviewers are represented, and deciding whether or not they are suitable for a paper.

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

The solutions are presented in **bondi**. Generally speaking, programs consist of let-declarations and abstract data type declarations, in the ML-style [12], and class declarations in the style of Java [2] or C# [3]. New notation will be explained when it appears for the first time.

The simplest solution for the reviewer-matching problem is given by

```
filterList (okToRev paper) committee        (1)
```

in which `committee` is a list of `Reviewer`, which in turn is an abstract data type to represent a reviewer. `okToRev: Paper -> Reviewer -> Bool` determines if a reviewer is suitable to review a paper, and `filterList` of type

```
(a -> Bool) -> List a -> List a
```

is the usual operation for filtering a list by some predicate. In Solution (1) the type variable `a` will be instantiated to type `Reviewer`.

More generally, the potential reviewers need not be in a single list. For example, suppose that the program committee forms one list and the past reviewers form another such list. These could be combined into a pair of lists, a list of lists, or a tree, etc. called `reviewers: b a` where `b` is a type variable representing a structure. In this case, a more general, *structure polymorphic* filter is required, given by `filter:(a -> Bool) -> b a -> List a` and the solution becomes

```
filter (okToRev paper) reviewers.           (2)
```

Solution (2) requires a dedicated data structure for holding reviewers, but in practice they will be scattered throughout a larger data structure representing the conference as a whole. Such a structure usually contains other types of data besides reviewers, such as conference name, papers, and events. Reviewers may also appear in different substructures such as committee member list and past reviewer repository. To traverse such a structure for reviewers, generalise the predicate `okToRev paper` which acts on reviewers only, to the function `okIsRev paper: a -> List Reviewer` which takes an argument `x` of any type `a`. If it is a suitable reviewer then the result is the singleton list containing `x` else the result is empty. Now the solution becomes

```
select (okIsRev paper) conference           (3)
```

where `select` has type

```
(all a.a -> Maybe b) -> c -> List b.
```

In our example, the type variable `b` is instantiated to the type of reviewers and `c` to that of the conference. The argument type `(all a. a -> Maybe b)` contains the quantified type variable `a` since `okIsRev paper` must be able to act on arguments of any type.

Now suppose that two conferences decide to federate, as represented by some data structure `confs`. Further, they agree to share reviewer data, so that a reviewer known to one conference can be

used by the other. However, this sharing is not to apply to members of the respective program committees, who already have enough to do.

The solution requires awareness of *where* the different lists of reviewers sit within the structure of the conference federation, so that it has the flexibility to choose lists according to their *role* within the conference. For this purpose *paths* are ideally suited since they combine properties of the *goals* (e.g. reviewers) with properties of their location within the structure, as will be represented by *stages*. The solution presented is a variation of *signposts* [5], or, more generally, a *pattern structure* [6]. The signpost employed is

```
Stage (getRevs paper) (Goal (okIsRev paper))
```

where `Stage (getRevs paper)` finds a list containing reviewers according to their role (relative to the given paper) and `Goal (okIsRev paper)` chooses suitable reviewers as before. Then the overall solution becomes

```
navigate (Stage(getRevs paper)
           (Goal(okIsRev paper))) confs          (4)
```

where `navigate: SignPost a -> b -> List a` is the pattern-matching function that navigates through the stages, to the goal.

If the two conferences are willing to create a central administration holding all their data then the above solution is fine. However, a more likely scenario is that they will choose a confederation, in which each conference maintains slightly different information about its reviewers. This can be handled using object-oriented classes. Create a super-class of reviewers, with sub-classes for specialised reviewers as required. Now `getRevs` and `okIsRev` must be adapted to handle classes, and the structure of the conference is also adapted. The object-oriented solution is thus

```
navigate (Stage(getRevs2 paper)
           (Goal(okIsRev2 paper))) confs.        (5)
```

Note that although `getRevs2` and `okIsRev2` are given as functions, they can be defined using methods which have special cases for different classes of reviewers.

These solutions illustrate four different sorts of polymorphism: the *data polymorphism* of `filterList`, the *structure polymorphism* of `filter`, the *path polymorphism* of `select` and of `navigate`, and the *inclusion polymorphism* of the reviewer classes.

While fragments of the expressive power displayed here can be found in other settings, no other calculus or typed programming language is able to support such breadth. For example: the structure polymorphism of generic Haskell [4] requires pre-processing; the path polymorphism of the "scrap your boilerplate" approach [10, 11] requires additional code within type classes; the path polymorphism of Stratego/XT [14] is untyped; Ocaml [13] does not support dynamic dispatch; and Java [2] cannot easily combine inclusion and data polymorphism.

Actually, the generality of the reviewer problem is not quite exhausted since there remains the problem of creating the superclass of reviewers, which may still be onerous. Pattern calculus (but not yet **bondi**) may be able to address this by using *dynamic* patterns (and *pattern polymorphism*) to exploit ontologies that can link together independent classes of reviewers. This leads in to the more general area of web services, as discussed in the conclusion.

The sections of the paper are: 1 Introduction; 2 Data polymorphism; 3 Structure polymorphism; 4 Path polymorphism; 5 Path matching; 6 Inclusion polymorphism; 7 Conclusions and future work.

2. Data polymorphism

Our running problem is to assign reviewers to a paper. For simplicity, we begin by assuming that all reviewers have the necessary expertise, so that the challenge is to exclude reviewers who have a conflict of interest, either because they are an author of the paper in question, or a close associate of such an author.

In this section, the potential reviewers are exactly the members of the program committee, who are represented by a list, so that the solution is to simply filter the list in the usual functional style.

The first polymorphic list function required is the `isMember` function

```
let rec (isMember : a -> List b -> Bool) x =
| Nil -> False
| Cons y ys -> equal(x,y) || (isMember x ys).
```

which checks for equality of its first argument with any entry of its list argument. `isMember` makes use of **bondi**'s generic equality

```
equal: a * b -> Bool
```

(as defined in Section **??**) that is able to compare arbitrary terms. Type safety can be strengthened by restricting the type to `a * a -> Bool` but later, more polymorphic examples, will require the greater flexibility of the generic `equal`.

Much of the notation should be familiar from ML and other functional programming languages: `let rec` introduces recursion; the vertical bar `|` separates cases of a pattern-matching function; `Nil` and `Cons` are the usual list constructors; and the double bar `||` is disjunction.

The second polymorphic list function is

```
disjoint: List a -> List b -> Bool
```

that is `True` if its arguments have no members in common. This can be easily implemented using the `isMember` function. The third polymorphic list function employed is

```
filterList: a -> Bool) -> List a -> List a.
```

Its first argument is a boolean function that serves as a predicate to filter the list members given by the second argument. The implementation is straightforward; the predicate is applied to list members one by one, and only those satisfying the predicate stay.

The specifics of the reviewing problem require the following abstract data types representing people, reviewers and papers, respectively.

```
datatype Person = Name of String
datatype Reviewer = Rev of Person and List Person
datatype Paper = Paper of String and List Person
```

A reviewer `Rev person assocs` is given by the person `person` and a list `assocs` of their associates. A paper `Paper title auths` provides a title `title` and a list `auths` of authors.

Conflicts are avoided using the function `okToRev` given by

```
let (okToRev : Paper -> Reviewer -> Bool) =
| Paper title auths ->
  | Rev person assocs ->
      disjoint (Cons person assocs) auths
```

Reviewer conflicts of interest are avoided using the function `okToRev` that is `True` if the authors of a paper and associates of a reviewer are disjoint. Thus the complete Solution (1) is given by

```
let findReviewers paper = filterList (okToRev paper).
```

A concrete example arises from considering the paper `paper` defined by

```
Paper "Calculus Revisited"
    [Name "B",Name "C",Name "N"]
```

and two reviewers

```
revA = Rev (Name "A") [Name "H"]
```

and

```
revB = Rev (Name "B") [Name "I",Name "J"].
```

Now `findReviewers paper [revA,revB]` evaluates to (the value of) `[revA]`.

bondi supports an interactive environment. Given the declarations above then the evaluation of `findReviewers` produces the following display:

```
~~ let okRevs = findReviewers paper [revA,revB];;
okRevs: List Reviewer
okRevs = [Rev (Name "A") [Name "H"]]
```

The `~~` is the **bondi** prompt and input is terminated by `;;`. The system reports the type and value of the resulting list.

3. Structure polymorphism

The limitation of the solution above is that the potential reviewers must be represented by a single list. In practice, it will often happen that reviewers are drawn from varying structures, containing past authors and reviewers, as well as the program committee. The challenge is thus to work with an arbitrary collection of reviewers, without first converting it into a list.

This is achieved by generalising the function `filterList` to a more generic

```
filter : (a -> Bool) -> b a -> List a
```

in which the collection is initially stored in a `b`-structure of `a`'s, not necessarily a list. This is done by using the structure polymorphic `foldleft` of type

```
(a -> b -> a) -> a -> c b -> a)
```

which in turn is defined in the standard prelude for **bondi**. Thus the `filter` function is defined by

```
let filter pred =
  foldleft (fun x y ->
             if (pred y)
             then Cons y x
             else x)
           []
```

that constructs a list of results that are `True` for the predicate `pred`.

While the program committee may be stored in a list, a larger pool of reviewers may be kept in a more efficient structure such as a tree. To represent a sample of such data structures the following data types are defined

```
datatype PairList a =
  PairList of List a and List a
datatype Tree a =
  Leaf of a | Node of Tree a and Tree a
datatype ListTree a =
  ListTree of List a and Tree a.
```

The `PairList a` is simply a pair of lists containing elements of type `a`. The `Tree` is a simple representation of a tree style data structure. The `ListTree a` combines both a list of `a`'s and a `Tree` of `a`'s. The general Solution (2) for filtering an arbitrary data structure of reviewers is now given by

```
let findReviewers paper = filter (okToRev paper).
```

Continuing from the previous example, define two new reviewers

```
revC = Rev (Name "C") [Name "K"]
```

and

```
revD = Rev (Name "D") [Name "L"].
```

Then both

```
findReviewers paper
  (PairList [revA,revB] [revC,revD])
```

and

```
findReviewers paper
  (ListTree [revA,revB] (Leaf revD))
```

evaluate to `[revD,revA]` as shown here.

```
~~ let okRevs1 =
     findReviewers paper
       (PairList [revA,revB] [revC,revD]);;
okRevs1: List Reviewer
okRevs1 = [ Rev (Name "D") ([Name "L"])
          , Rev (Name "A") ([Name "H"])]
~~ let okRevs2 =
     findReviewers paper
       (ListTree [revA,revB]
         (Node (Leaf revC) (Leaf revD)));;
okRevs2: List Reviewer
okRevs2 = [ Rev (Name "D") ([Name "L"])
          , Rev (Name "A") ([Name "H"])]
~~ okRevs1 == okRevs2;;
it: Bool
it = True
~~
```

Observe that the underlying structure is immaterial to the results as shown by comparing `okRevs1` and `okRevs2`.

4. Path polymorphism

A limitation of the structure polymorphic approach is that it requires a clear separation of the data of interest from the structure that contains it. However, a conference typically contains several sorts of data, concerning papers, authors, a program committee, a repository of past reviewers, etc. in which case it is infeasible to represent the data as a structure parameterised by reviewers. Now the challenge is to find reviewers within a larger conference structure with many types of elements and substructures such as the following:

```
datatype Committee =
  Comm of List Reviewer
datatype Reviewers =
  Revs of List Reviewer
datatype Conference =
  Conf of String and List Paper and
  Committee and Reviewers
```

In practice, the structure may be more complicated, and vary between conferences or over time.

To traverse a data structure without knowledge of the components requires a function that is *path polymorphic*. It should be able to walk through different *paths* to reach target elements that are located in different places, and be able to handle other types of elements encountered en route. Performing such a traversal and selecting relevant elements to collect can be done by the `select` function with the type

```
(all a.a -> List b) -> c -> List b
```

where the `a` must be for all types, `b` is the type of the desired results and `c` is the type of the data structure. Its definition is approximated by

```
let rec (select:
    (all a . a -> List b) -> c -> List b) =
  fun f ->
    | Ref (Ref _) -> []
    | Ref z -> select f z
    | z y -> append (f (z y))
             (append (select f z) (select f y))
    | y -> append (f y) []
;;
```

where `append` is the usual list append. The first two cases that include the constructor `Ref` are exploited in the inclusion polymorphism of Section 6. The `select` function cannot be applied to `okToRev` for typing reasons, so the latter must be generalised to

```
let (okIsRev : Paper -> a -> List Reviewer) paper =
| Rev name auths ->
    if (okToRev paper (Rev name auths))
    then [Rev name auths]
    else []
| z -> [].
```

Note that this function has cases of different types: the first has type `Reviewer -> List Reviewer` and the second has type `a -> List Reviewer`. This is typical of path polymorphic functions but makes it harder to ensure type safety. The resulting path polymorphic Solution (3) is given by

```
let findReviewers paper = select (okIsRev paper).
```

An example `conf1` of a conference can be defined as

```
Conf "Beta Programming" [paper]
  (Comm[revA,revB]) (Revs[revC,revD]).
```

Now `findReviewers paper conf1` evaluates to `[revA,revD]`.

5. Path matching

A limitation of the previous path polymorphic approach is that all reviewers in the conference are treated the same way, regardless of their role, but in practice it may be necessary to distinguish, say, committee members from other reviewers.

For example, suppose that two conferences are federated and agree to share reviewer lists, but not to ask members of the one program committee to review papers for the other conference. The difficulty is now to establish which collections of reviewers to collect, which must be done by distinguishing paths to the ordinary reviewer lists from those to program committees.

The necessary path descriptions can be given as *signposts* (a generalized datatype that contains not only data but also functions)

```
datatype SignPost b =
| Goal of all a.(a -> List b)
| Stage of all a.(a -> List c) and SignPost b.
```

They define how to select the final targets (the goals) and how to handle items encountered en route (the stages). The `Goal` signals that the end of the path has been reached and that the function held by the goal should now be used to select the results. A `Stage` provides a function that is used to collect possible paths to follow and a further `SignPost` to navigate the remainder of the path. The original signposts [5] were significantly more sophisticated. The situation can be further generalised to consider *pattern structures* [6].

To follow the signposts we define the `navigate` function that takes an instance of `SignPost` as one of its arguments:

```
let rec (navigate : SignPost b -> a -> List b) =
| Goal f -> select f
| Stage f sign ->
    fun z -> (navigate sign (select f z)).
```

For our conferences, the lists of reviewers are produced by a function

```
let (getRevs : Paper -> a -> List (List Reviewer))
  paper =
| Conf name papers (Comm committee) (Revs reviewers)
    ->  if (isMember paper papers)
        then [reviewers,committee]
        else [reviewers]
| x -> []
```

which returns the list of conference reviewers and perhaps the program committee, too. As with `okIsRev`, a default case that matches anything is also required to type match with `select`. Now the path matching Solution (4) is defined by

```
let findReviewers paper =
@navigate (Stage (getRevs paper)
            (Goal (okIsRev paper))).
```

A concrete example is given involving two new reviewers `revE` and `revF`

```
revE = Rev (Name "E") [Name "M"]
revF = Rev (Name "F") [].
```

and by defining a conference `conf2`

```
Conf "Information Theory" []
    (Comm [revE]) (Revs [revF]).
```

Then the evaluation of `findReviewers` proceeds as shown

```
~~ let okRevs = findReviewers paper (conf1,conf2);;
okRevs: List Reviewer
okRevs = [ Rev (Name "D") ([Name "L"])
         , Rev (Name "A") ([Name "H"])
         , Rev (Name "F") []]
~~
```

yielding three reviews, two from the first conference and one from the second. Observe that reviewer `revE` has been avoided due to being on the program committee of `conf2` (and not due to conflicts).

6. Inclusion polymorphism

Although the solution above is able to support various sorts of conference organisation it nevertheless requires that types such as `Reviewer` be fixed in advance. This requirement can be relaxed by introducing an object-oriented class of reviewers and allowing conference organisers to declare their own reviewer sub-classes. Such an approach also allows the suitability of reviewers to be expressed by a method which can be specialised for different sorts of reviewers.

This section creates a class of reviewers and a single sub-class of specialists with areas of expertise used in deciding suitability. Some of the earlier data types must now be reworked as classes. In particular, corresponding to the type `Person` is the class `PersonC`

```
class PersonC {
name : String;
get_name = { | () -> !this.name }
set_name = { fun n -> this.name = n } }.
```

It has one field `name` of type `String` and two standard methods, the get and set methods for the name. This declaration introduces a new abstract data type with constructor

```
PersonC: a -> ref String -> PersonC[a].
```

whose first argument (of type a) represents any additional fields the person may have, and whose second argument is an (assignable) reference to a name string. A PaperC class is defined similarly with fields title and authors for its fields.

Now reviewers are represented using a sub-class ReviewerC of PersonC declared by

```
class ReviewerC extends PersonC {
associates : List (PersonC [Top]);
get_associates = { | () -> !this.associates }
set_associates = { fun a -> this.associates = a }
suitable_for = { fun paper ->
  disjoint (paper.get_authors())
        (Cons (this:PersonC[Top]) !this.associates)
} }
```

Since the associates can be any sort of person, the field associates has type List (PersonC [Top]) where Top is the *top type*, a super-type of every type. The only non-trivial method in the class is suitable_for of type PaperC[b] -> Bool which is defined using the higher-order parametrically polymorphic function disjoint.

The sub-class declaration introduces another constructor

```
ReviewerC :  a -> ref (List PersonC[Top]) ->
    ReviewerC a
```

to collect the new fields. These are combined with the fields of the PersonC class so that the pattern for a reviewer is given by

```
PersonC (ReviewerC rest assocs) name :
    PersonC (ReviewerC a)
```

where rest is used to represent any additional fields associated with further sub-classes. As the depth of the sub-class hierarchy increases, so too will the size of the pattern and its type. To avoid this, we introduce the syntax ReviewerC[a] for the type above. Also, the paper uses x:ReviewerC[a] as syntax for the pattern PersonC (ReviewerC _ _) _ where _ matches anything.

The data types for the conference and substructures are recreated to use the classes for paper and reviewers.

```
datatype Committee2 =
  Comm2 of List ReviewerC[Top]
datatype Reviewers2 =
  Revs2 of List ReviewerC[Top]
datatype Conference2 =
  Conf2 of String and List PaperC[Top] and
  Committee2 and Reviewers2
```

The getRevs2 function can be defined similarly to getRevs with the new data types and classes. The same needs to be done for the okIsRev function, only now the matching must be on a class as shown here

```
let (okIsRev2: PaperC[b] -> a -> List a) paper =
| (x: ReviewerC[c]) ->
      if x.suitable_for(paper)
      then [x]
      else []
| z -> [].
```

The object-oriented solution (5) is thus given by

```
let findReviewers2 paper =
@navigate (Stage (getRevs2 paper)
    (Goal (okIsRev2 paper)))
```

Now suppose that a conference wishes to refine the representation of papers, reviewers and their suitability. Assume a sub-class KeywordPaperC of papers that have a field keywords representing relevant areas of research.

A sub-class of specialists is defined by

```
class SpecialistC extends ReviewerC {
areas : List String;
get_areas = { | () -> !this.areas }
set_areas = { fun a -> this.areas = a }
suitable_for  = {
| (x:KeywordPaperC[b]) ->
    super.suitable_for(x) &&
    (not (disjoint (x.get_keywords()) (!this.areas)))
| (z:PaperC[c]) -> False
} }
```

A specialist is a reviewer who has areas of expertise, represented by a list areas with its associated get and set methods. The method suitable_for is specialised to first check that it is suitable as before, represented by the boolean super.suitable_for(...) and also that there is at least one keyword among the areas of expertise, as represented by

```
(not (disjoint (kp.get\_keywords()) (!this.areas))).
```

The default case | (z:PaperC[c]) -> False expresses the intention that specialist reviewers refuse to review papers that do not specify keywords.

These additions to the class hierarchy require no modifications to the function findReviewers2, demonstrating *dynamic dispatch*. More precisely, evaluation of findReviewers2 will invoke, for each entry in a list of type ReviewerC[Top], the appropriate specialisation of the method suitable_for.

For illustration, define revCA,revCB,paperC etc. to represent the same data as revA,revB,paper in Section 5 but using the classes corresponding to the older data types. Also consider a keyword paper kpaperC whose keywords are given by ["types","calculus"], and a specialist speCG whose areas are given by ["formalism","types"].

```
let confs2 =
  ( Conf2 "Beta Programming" [paperC]
      (Comm2 [revCA,revCB])
      (Revs2 [revCC,revCD])
  , Conf2 "Information Theory"  [kpaperC]
      (Comm2 [revCE])
      (Revs2 [ (revCF:ReviewerC[Top])
             , (speCG:ReviewerC[Top]])).
```

Evaluating findReviewers2 paperC confs2 produces

```
[revCD,revCA,revCF]
```

since the specialist speCG is not interested in papers without keywords. By contrast, findReviewers2 kpaperC confs2 evaluated to [revCC,revCD,revCF, speCG,revCE], avoiding only the committee members of the conference kpaperC does not belong to.

This is the last of our solutions to the reviewing problem. It employs data polymorphism, path polymorphism, path matching and inclusion polymorphism in a natural manner.

7. Conclusions and future work

The difficulty of combining programming styles is well known. Perhaps the best example is the struggle to combine functional and object-oriented programming, but similar issues arise in combining objects with relational data, or more recently in trying to design web-services. This paper offers a new approach to the issues, through pattern-matching. The various styles are combined within solutions to the simple, but telling, reviewing problem, of searching unknown data structures for reviewers having the right roles, whose suitability is determined dynamically.

Future work may consider even greater generality by supporting confederations in which there are no shared super-classes of reviewers. Then the goal is to dynamically create the patterns necessary to interact with unfamiliar data structures, by passing ontologies as parameters. The basic principles are illustrated in the

Another question raised by this approach is its relation to logic. The Curry-Howard Isomorphism identifies computations with proofs, and types with propositions, as exploited in, say, [**?**]. This suggests that the new forms of polymorphism should correspond to new proof principals but it is not clear what they should be. One possibility is that path polymorphism corresponds to proof by structural induction, a generalisation of induction over the natural numbers. Generalising the Peano principal leads to the following (informal) proof rule

$$\frac{\forall c.P(c) \quad \forall x.\forall y.P(x) \wedge P(y) \Rightarrow P(x\ y)}{\forall x.P(x)}$$

in which c represents a constructor and $x\ y$ is a compound data structure. Some closer ties between pattern calculus and logic have been explored by examining the expressiveness of combinatory logic [8], however further relation to **bondi** and programming languages remains.

The examples in this paper demonstrate the desirability of multi-polymorphic programming. The implementations in **bondi** show that multi-polymorphic programing is practical, in the sense that the expected behaviour has been realised in a small programming language whose syntax is relatively familiar.

That done, one common question is whether the core of pattern calculus is fundamental to such a language. Each programming style tries to balance an emphasis on functions with an emphasis on data structures. At the extremes are pure λ-calculus, in which everything is a function, and Turing machines, in which everything is a structure on tape. In between are relational languages, with some choices of structure and functionality, and object-orientation, which try to package the two together. From this viewpoint, the challenge is to combine functions and data structures without strain. Pattern-matching functions provide a perfect combination of structure, as described by patterns, and functionality, as described by the bindings. On this view, the goal is to make pattern-matching as expressive as possible. Indeed, the intensionality afforded by considering the structure of an argument to a function proves to yield greater expressive power than the λ-calculus [8].

Putting aside such theoretical concerns, even the most skeptical reader can accept that the reviewing problem poses a challenge which cannot be met from within any previous programming style. Yet the issues it raises are typical of the challenges arising when working with large and evolving systems full of people, products and processes. Indeed, they are central to our ability to engage with systems about which we have only partial information, as arise in web services. The implementation of the typed pattern calculus in **bondi** is a step towards the general solution of such problems.

References

[1] bondi programming language. `http://bondi.it.uts.edu.au/`. URL `http://bondi.it.uts.edu.au/`.

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[3] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154916.

[4] R. Hinze and J. Jeuring. Generic Haskell: Applications. In *Summer School on Generic Programming*, 2002. URL `http://www.cs.uu.nl/~johanj/publications/ghpractice.pdf`.

[5] F. Huang, B. Jay, and D. Skillicorn. Programming with heterogeneous structures: Manipulating XML data using **bondi**. In *Twenty-Ninth Australasian Computer Science Conference (ACSC2006)*, volume 48(1), pages 287–296, 2006.

[6] F. Y. Huang. *Type-safe Computation with Heterogeneous Data*. PhD thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, August 2007.

[7] B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.

[8] B. Jay and T. Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.

[9] B. Jay and D. Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):34 pages, 2009.

[10] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *Proc. of POPL'07*. ACM Press, January 2007.

[11] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38 (3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and mplementation (TLDI 2003).

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[13] The Caml language. The caml language: Home, 2013. Retrieved 12 June 2013, from http://caml.inria.fr/.

[14] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004. URL `http://www.cs.uu.nl/research/techreps/UU-CS-2004-011.html`.