

# Programming with Heterogeneous Structures: Manipulating XML data using **bondi** [Extended Abstract]

F.Y. Huang<sup>†</sup>   C.B. Jay<sup>‡</sup>   D.B. Skillicorn<sup>†</sup> (Contact author<sup>1</sup>)

<sup>†</sup> School of Computing, Queen's University

<sup>‡</sup>Department of Software Engineering, University of Technology, Sydney

## Abstract

Manipulating semistructured data, such as XML, does not fit well within conventional programming languages. A typical manipulation requires finding *all* occurrences of a structure whose context may be *different* in different places, and both aspects cause difficulty. These problems of data access are often addressed by special-purpose languages which then pass their results to other, general-purpose languages, creating the impedance mismatch problem. Alternative approaches add a few new means of data access to existing languages without being able to achieve the desired expressive power. We show how an existing programming language, **bondi**, can be used as is to handle XML data, since data access patterns, called signposts, are first-class expressions which can be passed as parameters to functions.

## 1 Introduction

Semistructured data, such as XML, pose a problem for general-purpose programming languages. A typical manipulation of such data consists of a *search* for all occurrences of a pattern, *extraction* of the occurrences and some part of their context, *changes* to these extracted data structures, and their *replacement* in the entire structure. General-purpose programming languages have trouble typing such programs because the target pattern can occur in different contexts; and they also have trouble expressing the implied universal quantifier in the search.

These difficulties led to the design of special-purpose query languages, emerging both from the database community and the structured text community. The problem with using a special-purpose language for manipulating XML is that it creates an interface between data extraction and data use. For example, in a typical web environment, the data itself is in a back-end system and the results of the data query/transformation must be passed to a front-end system for further processing. The existence of a boundary requires a common format, usually quite a low-level one such as a string, by which the back-end and front-end communicate. This requires extra programming effort and, potentially, performance penalties as well. This has been called the *impedance mismatch problem*.

There are obvious benefits to extending general-purpose programming languages so that they can handle XML manipulation in native mode. Doing so reduces or eliminates the impedance mismatch problem, since computations at the browser, front-end, and back-end can all be done in the same language environment. Because such languages are typed, there is less overhead at runtime (for dynamic type checking) and a reduced chance of catastrophic failure or unintended leakage of information. Also, because of the expressive power of such languages, programs may be smaller and more modular, making them cheaper and easier to build and maintain.

Extending general-purpose languages to include XML manipulation directly has proven difficult, although a number of languages have made some progress towards this goal. Here we show that

---

<sup>1</sup>skill@cs.queensu.ca, Phone: +1 613 533 6065

the `bondi`<sup>2</sup> language, in which data structures and patterns are treated as of equal importance to functions, allows XML manipulation to be expressed in a natural way, and without any extensions to the language.

The contribution of this paper is:

- to show that the ability to handle structures and patterns in the same way as other programming entities is the key to manipulating XML in an effective, but also properly typed, way;
- to show that this increase in expressiveness comes with greater simplicity, rather than greater complexity;

Rather than aggregate the features found in the existing wide variety of XML query and transformation languages, `bondi` treats structures and structure-matching patterns as first-class objects. Hence, control flow can be determined by data structures, not just values; and structures and patterns can be passed as parameters. This respect for the data creates new power for programming with heterogeneous structures in XML data.

Because `bondi` is a general-purpose language, XML applications can be seamlessly integrated into other applications, including web and web service applications.

## 2 Related Work

In the early years of XML, due to the mismatch between XML types and types in programming languages, special-purpose XML query languages such as Lorel [1], XML-QL [7] and XSLT [4] were invented to handle query and transformation of XML data. They are typically untyped, handling both tag names and element content on as low a level as strings. They have very limited expressive power, and are unable to describe complex substructures within XML data. In many settings, their results must be passed to other application programs developed in general-purpose programming languages. Using XPath [5], XSLT is able to search for hierarchical path patterns of arbitrary depth. The other languages are quite restricted for expressing hierarchical search patterns.

In recent years, attempts have been made to merge XML processing into general-purpose programming languages in two styles, object-oriented and functional. On the object-oriented side, the newest proposals are XJ [9] based on Java and  $C\omega$ [8] based on  $C\#$ . They both attempt to extend the base languages with datatypes matching the XML Schema types. On the functional side, representatives are XQuery[3], XDuce[10] and CDuce[2]. XQuery uses XML Schema-like syntax to define XML datatypes and uses XPath to express search patterns. XDuce and CDuce use regular-expression types. They all match XML Schema types very well. In these general-purpose programming languages of both flavors, XML elements are first-class items and can be manipulated freely, subject to the programming power of individual languages. However, none of them treat structures of XML elements, and the patterns to match these structures, as first-class citizens, losing the power to compute directly with them, e.g., passing them as parameters. Adding the ability to handle a rich set of hierarchical path patterns to such languages has, so far, broken their type systems. We will demonstrate some of the problems in Section 4 by comparing our `bondi` programs with CDuce programs for an XML processing task.

---

<sup>2</sup>`bondi` is pronounced Bond-eye and is named for a famous beach in Sydney.

### 3 Parameterizing Structures and Patterns

Programming (and maintenance) are simpler when programs are built so that as much of their behavior is captured by parameters as possible. Often this has a secondary benefit because the resulting program is simpler and easier to understand (many of the cases have become different parameter choices). Programming languages that support the passing of functions as parameters (higher-order functions) or use subtyping to pass objects of varying behavior are plentiful, but until now there has not been general account of how to pass around information about the structure of data within a typed programming language. The *pattern calculus* [12–14] achieves this within a very general framework. It supports program re-use through five distinct forms of polymorphism within a strongly-typed language. All such forms can simplify the task of manipulating XML data but two of these, path and pattern polymorphism by parameterizing data structures and structure-matching patterns, are particularly suited to describing XML access paths. The expressive power of the pattern calculus is demonstrated in the new, general-purpose programming language `bondi` [11]. This section introduces a sequence of examples requiring increasing degrees of parameterization; and encodes them in `bondi`.

#### 3.1 A Motivating Scenario

Suppose we have a data repository containing geographical information and we want to carry out the following operation: *Add 5% to the population of all of Canadian cities*. How could we express such an operation?

A sequence of expressions with increasing parameterization is: [more details in the full paper]

```
IncrementthePopulationsofCanadianCities(5)
updateThePopulationsofCanadianCities(incrementby, 5)
updateCanadianCities(Population, incrementby, 5)
updateCanadian(City, Population, incrementby, 5)
update(Countryname == "Canada", City, Population, incrementby, 5)
```

Now we see that the parameters “Canada”, *City* and *Population* are all related and it is the *connections* between them that define the real parameter of interest. So we could rewrite the code as:

```
update(Canadian_City_Population)(incrementby, 5)
```

which has a (complex) pattern parameter, or what we call *signposts*. Now if we want to search for more complicated structures within the geographical database, we don’t have to keep building more complicated functions; rather, the complexity is expressed in the choice of the signpost *parameter* of the standard *update* function.

#### 3.2 Realization in `bondi`

This subsection encodes the examples above in `bondi`; they have all been executed and also appear in the file “`xmldata.bon`” at the `bondi` web-site [11]. Language features will be explained as they are used without attempting a full introduction here.

Define a datatype of populations by

```
datatype popul = Pop of float;;          (* unit: thousand people *)
```

This declaration introduces both a new type `popul` and, a new term, its constructor `Pop : float -> popul`. We can define a function for updating populations by a pattern-matching function

```
let (atPopIncrementByFivePercent: popul -> popul) x =  
  match x with  
  | Pop z -> Pop (z * 1.05);;
```

When applied to a term of the form `Pop x` it returns `Pop (x*1.05)`. This function can be parameterized with respect to the action to be taken by defining

```
let (atPopApply: (float ->float) -> popul -> popul) f x =  
  match x with  
  | Pop z -> Pop (f z);;  
let incrementByFivePercent x = x*1.05;;  
let atPopIncrementByFivePercent = atPopApply incrementByFivePercent;;
```

Evaluation of the new version of `atPopIncrementByFivePercent` reduces to the old one by substituting for the variable `f`.

More generally, we can consider increasing populations stored in larger structures, e.g. lists, defined by

```
datatype list a =  
  | Nil  
  | Cons of a and list a;;
```

This example defines a data type `list` which takes one parameter, `a`, which is the type of the list elements. It has two constructors: `Nil` which builds an empty list and `Cons` which constructs a new list from an element and a (sub)list. (Syntax sugar `[x, y, z, ...]` is equivalent to `(Cons x (Cons y (Cons z ...)))` and `[]` to `Nil` for the empty list.)

The function

```
let (listMap: (a->b) -> list a -> list b) f x =  
  match x with  
  | Nil -> Nil  
  | Cons y z -> Cons (f y) (listMap f z);;
```

takes a function `f` for its first argument and then is defined by pattern-matching over the two list constructors. For example, `listMap incrementByFivePercent` acts on lists of floats and `listMap atPopIncrementByFivePercent` acts on lists of populations. This illustrates how `listMap` is polymorphic in the choice of types `a` and `b` that represent the list entries, i.e. `listMap` is *data polymorphic*.

Of course, populations may appear as data in all sorts of structures, not just lists. This situation can be handled using a mapping function that is parametric in the choice of *structure* type as well as in the choice of the *data* types, i.e.

```
map1: (a->b) -> c a -> c b
```

whose type includes a type variable `c` representing the type of a uniform structure, e.g. `list`. The definition of `map1` is more complex than its type suggests as it relies on the theory of data structures developed in [13].

Even `map1`, however, is not flexible enough for our purposes, since a typical database is not going to have type `(c popul)` since there is no reason to single out populations for special treatment while ignoring, say, city names and areas in heterogeneous data structures such as XML.

Instead, let us define a function that acts on populations wherever they occur, by

```
let (updatePops: (float->float) -> a -> a) f x =
  match x with
  | Pop z -> Pop (f z)
  | y z -> (updatePops f y) (updatePops f z)
  | z -> z;;
```

The first case is the same as `atPopApply` but the second and third cases cause the action to be propagated to all parts of the data structure. That is, the pattern `y z` matches against any compound data structure, and causes both parts of the compound to be updated, while the final case is used to terminate at atoms of data. For example, `updatePops incrementByFivePercent [Pop x1, Pop x2]` evaluates to `[Pop x1*1.05, Pop x2*1.05]` but also `updatePops incrementByFivePercent ([Pop x1], Pop x2)` evaluates to `([Pop x1*1.05], Pop x2*1.05)` even though the populations appear on different levels of the data structure with different access paths. Thus `updatePops` is *path polymorphic*.

Examining the program above, it is clear that the constructor `Pop` is playing a completely passive role, and so is ripe for parameterization. Define

```
let (update: lin(a->b) -> (a->a) -> c -> c) \P f x =
  match x with
  | P z -> P (f z)
  | y z -> (update P f y) (update P f z)
  | z -> z;;
```

so that `updatePop` is now given by `update Pop`. The program `update` arises naturally from our earlier examples, but has a number of unusual technical features. First some conventions: capitalized variables such as `P` are always free unless explicitly bound as in `\P` (to be thought of as  $\lambda P$ ). Thus, the pattern `P z` contains a free variable `P` and a binding variable `z`. Evaluation of `update Pop` will substitute `Pop` for `P` so that the pattern above becomes `Pop z`. That is, `update` is *pattern polymorphic* since it takes a parameter used to build patterns. An important feature of `P` is that it is strongly typed, carrying the type information `(a->b)` of the target data it matches.

Some care is required when substituting into patterns, so such variables are required to be *linear* as indicated by the linear type `lin(a->b)`. Linear terms are explained in detail in [12]. For this paper, we will pretend that all linear terms are constructors though there are important alternatives.

Similarly, we can define a function `check` that simply checks that some property holds for the argument of a given constructor, by:

```
let (check: lin(a->b) -> (a->bool) -> c -> bool) \P f x =
  match x with
```

```

| P z -> f z
| y z -> (check P f y) || (check P f z)
| z -> False;;

```

where `True`, `False` are two constant constructors of type `bool` as usual and `||` is logical-or.

Suppose now that the goal is to update the populations of only cities, while leaving other populations unchanged. For example, consider geographical data having datatypes of cities, provinces and countries given by the declarations

```

datatype cityname = CityName of string;;
datatype river = River of string;;
datatype city = City of cityname and popul and list river;;

datatype provname = ProvName of string;;
datatype province = Prov of provname and popul and list city;;

datatype countryname = CountryName of string;;
datatype country = Country of countryname and popul and list province;;

```

Applying `update Pop f` will act on both the city and country populations indiscriminately. However, the function `update City (update Pop f)` gives the desired behavior. Although correct, this is not quite satisfactory, since it requires two updates. More complicated access patterns typical of XML will then require three or more updates, and there is still the challenge of checking side-conditions, e.g. that the city is in Canada.

The solution is to construct an abstract type that represents all of the information about how to access the data items. In simple cases this will be given by a complete hierarchical path, but in general the access information will be partial. For example, it is not necessary to know everything above a city to update its population. Hence, let us call such partial descriptions of paths *signposts* since they guide the way to target data items.

For the purposes of this paper, let us consider three sorts of signposts. It will be easy to add more sorts as required. A *goal* is a constructor whose argument is the pattern of the target item of interest. A *stage* is a constructor which constructs a signposts from a leading pattern of the path and the rest of the path. A *detour* is a path that has a side-path with a side-condition to check before continuing on the main path. Thus we obtain

```

datatype signPosts
  at a b c =
  | Goal of lin(c -> b)
  at (a1,a2) (b1,b2) c =
  | Stage of lin (a1 -> b1) and signPosts a2 b2 c
  at (a1,a2) (b1,b2) (c1,c2) =
  | Detour of signPosts a1 b1 c1 and (c1->bool) and signPosts a2 b2 c2;;

```

For example, to search for all populations with the partial path (`...country-...-city-...-popul`), the compound pattern can be encoded as:

```
let popPath1 = Stage Country (Stage City (Goal Pop));;
```

and to search for all populations of Canadian cities, we use `Detour`:

```
let popPath2 = Stage Country (Detour (Goal Countryname)((==) "Canada")
                                     (Stage City (Goal Pop)))
```

Now `check` and `update` can be modified to act on signposts as follows.

```
let (checks: (signPosts a b c) -> (c->bool) -> d -> bool) s f x =
  match s with
  | Goal \P -> check P f x
  | Stage \P s1 -> check P (checks s1 f) x
  | Detour s1 f1 s2 -> (checks s1 f1 x) && (checks s2 f x);;
```

```
let (updates: (signPosts 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 s1 f1 s2 ->
      if (checks s1 f1 x) (* the detour *)
      then updates s2 f x
      else x;;
```

Notice that function `checks` invokes the simple version `check`, and `updates` invokes `update` for single patterns. They use pattern matching to explore the structure of a given path pattern, that is, a `signPosts`. If the path pattern is a single pattern, `check` and `update` are invoked directly. If the path pattern is a `Stage`, `check` and `update` are used to search for the preceding single pattern, then from the matching points the search for the rest of the path pattern continues.

If the path pattern is a `Detour`, the functions check whether the detour side path got a match and if the content of the match satisfies the carried boolean function. If so, the functions go back to the starting point and continues the search for the rest of the path pattern. If the detour does not get a match or the carried boolean function fails, the functions return false and unchanged data respectively. Now it is straightforward to increment all populations of all Canadian cities, if `data` is the data repository containing geographical information:

```
updates popPath2 incrementByFivePercent data;;
```

Note that this executes independently of the presence of provinces. With our approach, full knowledge of the data schema is not required. This is especially useful for applications using data in an analytical style.

### 3.3 Folding

Given `bondi`'s support for parameterization over data structures and data access patterns, we can design other general functions in much the same way as `map1`, `update` and `updates`.

A function `foldleft1` can be defined, similar to `map1`, as:

```
foldleft1: (a->b->a) -> a -> c b -> a
```

It traverses a uniform structure of type  $(c \ b)$ , applying a given function to the values of type  $b$  it finds to modify the given value of type  $a$ . For example, given a definition of the integer addition function `add`, the application `foldleft1 add 0 AListOfInt` produces the sum of all integers in a list of type `(list int)`.

In the same way that `update` handles target patterns appearing in heterogeneous data structures, a generalized `foldleftp` in `bondi` for arbitrary data structure can be defined, again simply using three-case pattern matching:

```
let (foldleftp: lin(a->b) -> (c->a->c) -> c -> d -> c) \P f x w =
  match w with
  | P z -> f x z
  | y z -> foldleftp P f (foldleftp P f x y) z
  | z -> x;;
```

A more sophisticated version that folds elements satisfying a path pattern (`signPosts`) looks like this:

```
let (foldlefts: (signPosts a b c) -> (d->c->d) -> d -> e -> d) s f x w =
  match s with
  | Goal \P -> foldleftp P f x w
  | Stage \P s1 -> foldleftp P (foldlefts s1 f) x w
  | Detour s1 f1 s2 ->
      if (checks s1 f1 w)
      then foldlefts s2 f x w
      else x;;
```

Many more subtle operations can be expressed as using `foldleftp` and `foldlefts`, for example selecting a set of elements satisfying a `signPosts`, and returning them within a new structure. Parameter `f` must then be a function that collects a matching element into the new structure. We call it an accumulating function.

Defining such highly-parametric functions in `bondi` simply by pattern matching over `signPosts` allows us to perform a large class of XML search and transformation operations within a general-purpose programming environment easily, especially when there is no precise knowledge about an XML data schema.

[A further section which shows how to describe a set of standard kinds of XML manipulations in `bondi` is given in the Appendix.]

## 4 Comparison with XML Processing in CDuce

CDuce is a general-purpose functional language with regular-expression types that match XML Schema types very well. For example, corresponding to the datatypes defined in Section 3 for geographical data, CDuce can define types like:

```
type Country = <country>[Name Population Province*] (* means unbounded repetition)
type Province = <prov>[Name Population City*]
type City = <city>[Name Population River*]
type Name = <name>[String]
```

```

type Population = <pop>[Int]
type River = <river>[String]

```

XML elements are first-class items and can be manipulated freely. We can write a CDuce version of `updatePop` with the same functionality as the `bondi` version in Section 3. Suppose we already have a CDuce function `f (Int->Int)` to update one integer:

```

let updatePop (f: (Int->Int)) (x: <_>[_*]) :<_>[_*] =
  let [ y ] =
    xtransform [ x ] with
      <pop>[(z & Int)] -> [ <pop>[(f z)] ]
  in y

```

This function traverses the whole structure of `x` using the macro iterative operator `xtransform`, matches any `<pop>[Int]` and update it using function `f`. However, when the task becomes a more general `update` function for any pattern and any `f`, like the `bondi` version in Section 3, CDuce reaches its limit. A CDuce version of `update` might look like:

```

let update (a: Atom) (f: T->T) (x: <_>[_*]): <_>[_*] =
  let [ y ] =
    xtransform [ x ] with
      w & <(b)>[ (z & T) ] ->
        if (a = b) then [ <(b)>[ (f z) ] ] else [ z ]
  in y;;

```

But this program is not well-typed because the type of contents of `<(a)>[...]` elements may not match the argument type of function `f`, so that the evaluation of `(f z)` could fail. The difficulty comes from the inability to treat patterns as first-class typed values and pass them as parameters; an ordinary variable does not carry enough type information.

Further, since patterns are not first-class items, there is no way to define pattern-parametric structures like `signPosts`. When searching for a complex pattern (Rivers in a Country whose Name is Canada), a CDuce program has to hardcode all the subpatterns to reflect their structural relation, and cannot treat them as a single parameter.

[Comparisons to other languages could be made, but CDuce is perhaps the best-developed typed language for XML processing.]

## 5 Conclusion

The strongly-typed general-purpose programming language `bondi` allows structures and patterns in data to be treated as first-class objects. This makes it straightforward to define data-access patterns, `signPosts`, that can describe, in a very general way, path patterns in semistructured data such as XML. These signposts can also be passed freely within programs, allowing a natural style for search, extraction, modification, and restructuring of complex data. The ability to manipulate XML in a general-purpose language with strong typing makes it easy to integrate back-end data-access programming with front-end user-interface programming within a single system. It thus represents the first steps to solving the impedance mismatch problem.

## References

- [1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
- [2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.
- [3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language - W3C Working Draft, October 2004.
- [4] J. Clark. XSL Transformation(XSLT): Version 1.0 - W3C Recommendation, November 1999.
- [5] J. Clark and S. DeRose. XML Path Language (XPath): Version 1.0 - W3C Recommendation, November 1999.
- [6] S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars. 1999. Communication to W3C, [www-db.research.bell-labs.com/user/simeon/xquery.ps](http://www-db.research.bell-labs.com/user/simeon/xquery.ps).
- [7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. *Computer Networks*, 31(11–16):1155–1169, 1999.
- [8] E. Meijer and W. Schulte and G. Bierman. Unifying Tables, Objects and Documents. In *Proc. DP-COOL 2003*, Uppsala, Sweden, August 2003.
- [9] M. Harren, B. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML Processing into Java, 2003. Technical Report rc23007, IBM Research.
- [10] H. Hosoya and B.C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.
- [11] C. B. Jay. bondi web-page, 2004. [www-staff.it.uts.edu.au/~cbj/bondi](http://www-staff.it.uts.edu.au/~cbj/bondi).
- [12] C. B. Jay. Higher-order patterns, 2004. [www-staff.it.uts.edu.au/~cbj/Publications/higher\\_order\\_patterns.pdf](http://www-staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf).
- [13] C. B. Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.
- [14] C. B. Jay. Unifiable subtyping, 2004. [www-staff.it.uts.edu.au/~cbj/Publications/unifiable\\_subtyping.pdf](http://www-staff.it.uts.edu.au/~cbj/Publications/unifiable_subtyping.pdf).

## A Using bondi for XML Processing

Several papers have summarized “standard” XML query operations, for example [6]. We believe that a large class of essential operations can be handled in `bondi` by simply defining a few pattern structures and highly-parametric functions.

First let us define two simple helper functions, to search in a list and insert into a list without duplication:

```

let (listSearch: list a -> a -> bool) x y
  match x with
  | Nil -> False
  | Cons z w -> (y==z) || (search w y);;

let (listInsert: list a -> a -> list a) x y =
  match x with
  | Nil -> Cons y Nil
  | Cons z w -> if (listSearch (Cons z w) y)
                  then (Cons z w)
                  else Cons y (Cons z w);;

```

In all the following subsections, we will use variable `data` to represent a data repository containing geographical information.

## A.1 Search, Filtering and Extraction

Extracting information from XML data based on a search pattern and a filter is the most common kind of XML query. For example, we may want a list of names of cities whose population is bigger than 300 (in units of thousands). This query will consist of three components: the pattern to search for: `...city...cityname`; the data filter: `population > 300`; and the way to construct the result. The search pattern is easy to describe by a `signPosts`, and the filter is a boolean function carried by a `Detour` pattern. When this detour is absent, the operation becomes “select the names of all cities”. Combining matching items into a final result is a `foldlefts` operation in `bondi`: how the matching items are accumulated is just a matter of the design of the accumulating function being passed as parameter. If the result is to be a list of city names, i.e., of type `list cityname`, the accumulating function is a slightly modified version of `listInsert`:

```

let (elemInsert: lin(a->b) -> list b -> a -> list b \P x y = listInsert x (P y));;
let namePath = Stage City (Detour (Goal Pop) ((>) 300) (Goal Cityname));;

let result = foldlefts namePath (elemInsert Cityname) [ ] data;;

```

## A.2 Extraction while Preserving Structural Relations

Suppose each city has some `River` elements that represent the rivers running through that city. If we want lists of rivers, to be grouped by the names of the countries the rivers run through, we need to apply folding three times. Let `(x, y)` be syntactic sugar for `(Pair x y)` where `Pair` is the constructor of type `pair`.

```

type resultType1 = list (countryname, list river);;      (* type synonym *)

let (acceptor: a -> a -> a) x y = y;;

let (collectRivers1: resultType1 -> a -> resultType1) x y =
  let riverList = (foldleftp River (elemInsert River) [ ] y)
  and name = (foldleftp Countryname acceptor "null" y)

```

```

in
  (listInsert x (Countryname name, riverList));;

let result = foldleftp Country collectRivers1 [ ] data;;

```

### A.3 Extraction with Flattening

In contrast to the previous subsection, the parts of the structure that are matched sometimes need to be restructured. One simple restructuring is flattening. For example, in the previous example, rather than grouping rivers with country names, we may want to pair up each river with the country it runs through, i.e. the result type will be `list (countryname, river)`. A slight change to the accumulating function will do.

```

type resultType2 = list (countryname, river);;      (* type synonym *)

let (collectRivers2: resultType2 -> a -> resultType2) x y =
  let (insertPair: resultType2 -> string -> resultType2) u v =
    (let name = (foldleftp Countryname acceptor "null" y)
     in (listInsert u (Countryname name, River v)) )
  in (foldleftp River insertPair x y);;

let result = foldleftp Country collectRivers2 [ ] data;;

```

### A.4 Extraction with Complex Restructuring

Sometimes the query result may have to be restructured in more complex ways than flattening. We may want to group each river with all the names of countries the river runs through, i.e.:

```

type pairentry = (river, list countryname);;      (* type synonym *)
datatype resultentry = ResultEntry of pairentry;;
type resultType3 = list resultentry;;           (* type synonym *)

```

Instead of simply inserting matches into the result, extra effort is needed to check the existence of the matches in the intermediate result before insertion:

```

let (updateResult:string -> resultType3 -> string -> resultType3) n x r =
  if check River ((==) r) x
  then let (insertCountry: pairentry -> pairentry) y =
        (match y with
         | (r1, nameList) ->
            (if r1 == r
             then (r1, elemInsert Countryname nameList n)
             else (r1, nameList) )
        in (update ResultEntry insertCountry x)
  else
    (listInsert x (ResultEntry (River r, [(Countryname n)])) );;

```

If the river name `r` has appeared in the intermediate result, `updateResult` finds the `ResultEntry` element with river name `r`, and inserts country name `n` into it; otherwise, it inserts a whole new `ResultEntry` into the result for river `r`.

```
let (collectRiver3: resultType3 -> a -> resultType3) x y =
  let name = foldleftp Countryname acceptor "null" y
  in (foldleftp River (updateResult name) x y);;

let result = foldleftp Country collectRivers3 [ ] data;;
```

Note that all of the accumulating functions presented so far hardcode the target patterns, `Country`, `Countryname`, `Cityname`, etc. Given `bondi`'s support for pattern parameters, it is not difficult to make all these patterns into parameters so that the accumulating functions, and hence the queries, are fully parametric. The algorithms for the queries can be formalized as library components of `bondi`, so that XML users can directly use them by simply supplying query parameters, just as in other query languages.

## A.5 Indexing and Sorting

Indexing and sorting are the strength of general-purpose programming languages. This is also the case for `bondi`.

When the order in which XML elements appear matters, it is easy for a `bondi` program to capture that order information from the traversal order. For example, function `foldleftp` always traverses from left to right. When we use it to extract XML elements as in the previous subsections, a counter variable accompanying the accumulating function can capture the order information of the extracted elements.

If we want the elements in an extraction result to be sorted, we do not need to modify the accumulating algorithms. Note that all accumulating functions presented so far use some version of insertion functions implemented using `listInsert`. A small change to `listInsert` gives a version that produces sorted results.