Programming with Heterogeneous Structures: Manipulating XML data Using bondi

F. Y. Huang
School of Computing
Queen's University
huang@cs.queensu.ca

C. B. Jay
Faculty of Information Technology
University of Technology, Sydney
cbj@it.uts.edu.au

D. B. Skillicorn
School of Computing
Queen's University
skill@cs.queensu.ca

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 matching a *structured* search pattern, whose context may be *different* in different places, and both aspects cause difficulty. If a special-purpose query language is used to manipulate XML, an interface to a more general programming environment is required, and this interface typically creates runtime overhead for type conversion. However, adding XML manipulation to a general-purpose programming language has proven difficult because of problems associated with expressiveness and typing.

We show an alternative approach that handles many kinds of patterns within an existing strongly-typed general-purpose programming language called bondi. The key ideas are to express complex search patterns as structures of simple patterns, pass these complex patterns as parameters to generic data-processing functions and traverse heterogeneous data structures by a generalized form of pattern matching. These ideas are made possible by the language's support for pattern calculus, whose typing on structures and patterns enables path and pattern polymorphism. With this approach, adding a new kind of pattern is just a matter of programming, not language design.

Keywords: Pattern Calculus, functional programming, heterogeneous data structure, XML processing

1 Introduction

When processing semistructured data such as XML, a basic operation is to locate data items by their position in a structured context, usually described by a pattern or sequence of patterns. In some situations, these patterns can be as simple as matching a single type of element, for example, in the search for all `population` elements in a geographical dataset, `population` is a simple pattern to match for target data items. In other situations, search patterns are more complex, but complex patterns can usually be decomposed into simpler ones. For example, in the search for the complex pattern `population of individual cities in Canada` includes searches for a `country` element with a `countryName` descendant element having the value `Canada`, and some `city` descendant elements which in turn have `population` descendant elements (we do not use attributes in our examples, because attributes can be transformed into elements easily).

There are two ways to compose simpler patterns into more complex ones. The first is *vertical* composition, as in the search for the population of cities of Canada. Such complex patterns match XML elements from different levels of a hierarchy. We call them vertical patterns. Location paths, expressed in the popular XPath (Clark & DeRose 1999) language, fall in this category. The second is *horizontal* composition, as in the search for cities having child elements for name, population, either timezone or continent, and zero or more rivers, i.e., the pattern `(cityName, population, timezone|continent, river*)`. Such complex patterns match XML elements from the same level of a hierarchy. We call them horizontal patterns.

Vertical and horizontal patterns can be combined into even more complex search patterns. For example, the pattern: contact phone numbers of city halls of Canadian cities having child elements for name, population and zero or more rivers, is a combination of vertical and horizontal patterns.

Semistructured data processing poses a problem for general-purpose programming languages. For example, typical processing of XML data consists of a *search* for all occurrences of a search pattern, *extraction* of the occurrences and some part of their context, *changes* to these extracted data structures, and their *replacement* in the entire data structure. General-purpose programming languages have trouble typing such programs because the target pattern can be complex in different ways and 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 XML query languages, emerging both from the database community and the structured text community. The problem with using a query 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, subject to security holes and runtime overhead. This has been called the *impedance mismatch problem* (Bancilhon & Maier 1988, Wadler 2004).

There is an obvious benefit 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, security of programs can be verified statically, reducing runtime overhead for dynamic type checking and the 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 programming languages to include XML manipulation directly has proven difficult, although a number of attempts have made some progress towards this goal. Such attempts usually end up with a new or extended language that can only handle specific kinds of hard-coded XML search patterns which cannot be passed as typed parameters, and cannot be further extended without changing the language.

In this paper we show that an existing general-purpose functional-programming language, bondi, in which structures and patterns are treated as of equal importance to data and functions, allows XML manipulation to be expressed in a natural and general way, and without any extensions to the language.

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

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

In this paper we show that:

- Existing approaches to XML processing only handle limited kinds of search patterns, with weaknesses in either type-safety, parameterizability, extensibility to other kinds of patterns, or all three;
- The ability to handle structures and structure-matching patterns in the same way as other programming entities is the key to manipulating XML in an effective, but also properly typed, way;
- This increase in expressiveness comes with greater simplicity, rather than greater complexity, due to more powerful parameterization;
- With such expressiveness, adding a new kind of pattern, either vertical or horizontal, to XML processing is just a programming task, rather than a language (re-)design task.

The rest of the paper is organized as follows. Section 2 reviews existing XML processing approaches, then the theory on which our approach is based. Section 3 introduces the use of patterns as first-class objects and the construction of complex patterns from simple ones, and shows how these patterns contribute to better type-safety and higher parameterization. Section 4 briefly shows how our approach extends to new kinds of complex patterns. Section 5 draws conclusion and discusses some open issues of our approach.

2 Related Work

2.1 XML Query Languages

In the early years of XML, special-purpose XML query languages such as Lorel (Abiteboul, Quass, McHugh, Widom & Wiener 1997), YATL (Cluet, Delobel, Siméon & Smaga 1998), XML-QL (Deutsch, Fernandez, Florescu, Levy & Suciu 1999), XQL (Robie, Lapp & Schach 1998) and XSLT (Clark 1999) were invented to handle query and transformation of XML data. They are typically untyped, handling both tag names and element content as strings.

These query languages have very limited programming power, unable to express sophisticated computations on XML data. In many settings, the queries and their results must be passed, at runtime, to other application programs for further processing. These transfers are usually in a low-level format such as strings, requiring extra programming effort and runtime overhead for parsing and type-checking. The type safety of XML manipulation programs then relies on type-checking at runtime by explicit checking code inserted by programmers at development time. The correctness and completeness of the checking code are not guaranteed.

XSLT uses XPath (Clark & DeRose 1999) expressions as search patterns. XPath is powerful at expressing a wide range of complex vertical patterns, but XSLT is limited in programming power, and incapable of sophisticated computation. The other languages are quite restricted both in expressing vertical patterns and in programming. None of them is able to express horizontal patterns systematically, although they can hardcode individual ones (e.g. sibling axes in XPath can represent simple horizontal patterns).

2.2 Native XML Processing

In recent years, attempts have been made to merge XML processing into general-purpose programming languages. Typical approaches use special types to represent XML data and special expressions for search patterns, in addition to regular programming language features. The most recent efforts include XJ (Harren, Raghavachari, Shmueli, Burke, Sarkar & Bordawekar 2004), XQuery (Boag, Chamberlin, Fernandez, Florescu, Robie & Simeon 2005) and Cω (Bierman, Meijer & Schulte 2004, Meijer, Schulte & Bierman 2003) focusing on vertical patterns, and XDuce(Hosoya & Pierce 2003) and CDuce(Benzaken, Castagna & Frisch 2003) focusing on horizontal patterns. In terms of programming style, XJ and Cω are object-oriented, while XQuery, XDuce and CDuce are functional.

XJ and XQuery enforce static typing against XML schemas rather than native types of the programming languages, and express search patterns using embedded strings; hence type mismatches still exist to some extent. Cω, XDuce and CDuce express XML data and search patterns fully in native mode with static typing, so that XML processing can be handled within a single language.

The inability to parameterize structures and patterns, and poor extensibility, are two common shortcomings in all these languages. First, these languages can only parameterize XML data items, not structures of these items, nor the patterns to match the structures, because the latter are not first-class entities. Traversal of heterogeneous XML structures has to rely on runtime type casts even if the XML data

are parsed into well-typed form, and patterns have to be hard-coded in programs. Second, these languages only allow specific kinds of patterns and cannot be extended to other kinds easily in a type-safe way. An extension to a new kind of pattern requires new features to be added to the language; the type system has to be modified; and so does the compiler.

XJ extends Java with XML data types and XPath expressions, capable of handling vertical patterns conforming to XPath 1.0 (Clark & DeRose 1999). It expresses XML element types as Java classes, and uses special embedded strings containing XPath expressions as search patterns. Static typing of these XML types and embedded pattern strings against XML schemas is enforced by a special type checker. Because the type checking is against XML schema types, not native Java types, XML data and pattern expressions are not fully type-safe in Java. Since search patterns are just strings, there is a potential to include patterns other than XPath expressions, but only in an untyped way (or at best typed against XML schemas, not Java). Also, the special type checking requires that schemas for XML data are always available and trustworthy, which is unrealistic in many situations.

XQuery is designed as a query language but is equipped with some basic functional-programming features. It is intended to be a language for XML processing analogous to SQL for relational data processing. It aggregates many features from older XML query languages and SQL, and its data model and type system fully conform to XML and XML Schema specifications. It is a superset of XPath 2.0 (Berglund, Boag, Chamberlin, Fernndez, Kay, Robie & Simon 2005), making XPath expressions native, and so it is fully capable of handling vertical patterns of the XPath form. On the other hand, XQuery has only very limited functional programming features. Except in user-interactive settings, its expressions are supposed to be embedded in host programs in other languages for processing of query results. In such situation, the impedance mismatch problem still exists, just as in XJ, since XQuery is only typed in terms of XML schemas. The mismatch between XML schema types and host-language types weakens the safety of XML processing programs. The only advantage over XJ is that, in the absence of XML schemas, XQuery expressions can still be type-checked to some extent based on the type information in the expressions themselves.

Cω is intended to extend C#, another general-purpose programming language, with native types that support both object-oriented, relational and semi-structured data models, so that it can unify the processing of all these kinds of data. It introduces three new kinds of types: stream, anonymous struct and choice, roughly equivalent to list, heterogeneous tuple and sum types in functional languages. It uses the notion of content class for expression of XML schemas. For example, suppose an XML schema for geographical data has a country element type, with name, population and zero or more provinces as child elements. It then can be encoded as a content class `Country` as:

```
class Country {
    struct{ string name; float population; Province* provs; };
    ...  // appropriate constructor
    void increasePopulation(float percentage){...}
    ...
}
```

which contains an anonymous struct holding `name`, `population` and a stream of `Province`. In turn, `Province` is another content class (declaration not shown here) for province element type, which may have children name, population and a stream of `City`, and so on. Suppose `canada` is an instance of `Country`. The pattern to get the population of Canada can then be expressed as `canada.population`. To accommodate XPath-style vertical patterns, Cω also introduces filter expressions such as `Country[name=="Canada"]`, and transitive query expressions such as `Country...population` for population data appearing at arbitrary depth below countries. For example, the following method returns a stream of populations of cities in a given country:

```
virtual float* getPopulation(Country c1) {
    foreach (p in c1...City.population) yield return p;
}
```

The expressiveness of Cω for patterns in XML processing is roughly equivalent to XPath 1.0 (Clark & DeRose 1999) without backward axes. In contrast to XJ and XQuery, XML data and pattern expressions in Cω are fully native, expressed by identifiers all having Cω native types. There is no impedance mismatch problem. However, the pattern to search, such as `c1...City.population` in the above method, has to be hardcoded in the program and cannot be passed to a method parameter in a typed manner, so that it is not possible to have a general method to search for user-defined target data, something like `get(somePatternType pattern, Country c1)`. Moreover, adding other kinds of patterns, for example XPath backward axes, self-nested structures, or horizontal patterns would require large changes to the language.

XDuce and CDuce are functional-programming languages with regular-expression types added to general-purpose functional language features. These two languages use regular expressions to denote XML element types, and to define horizontal patterns to match the elements. For example, the country element type above can be declared in CDuce as:

```
type Country = <country>[Name Population (Province)*]
type Province = <prov>[Name Population (City)*]
type City = <city>[Name Population ...]
type Name = <name>[String]
type Population = <pop>[Int]
```

Traditional Pattern matching can be used to locate all population items and make some update to them in a piece of XML data:

```
let updatePop (x:<_>[_*]) :<_>[_*] =
    let [ y ] =
        xtransform [ x ] with
            <pop>[(z & Int)] -> [ <pop>[(z*101/100)] ]
    in y
```

This CDuce function uses regular-expression type `<_>[_*]`, meaning an element with any tag name and any content, to constrain both the parameter and result, and regular-expression type `<pop>[Int]`, meaning an element with tag name "pop" and an integer as content, to match target items for update. It traverses the whole structure of a given piece of XML data `x` using the macro iterative operator `xtransform`, matches any population element and increases it by 1%.

In XDuce and CDuce programs, patterns are well-typed and handled natively. CDuce can even encode XPath-like vertical patterns with child axes (though not descendant axes). However, just as for Cω, search patterns such as `<pop>[Int]` in the above CDuce program are not first-class terms and cannot be referenced and passed as well-typed parameters. And new kinds of patterns are not easy to include without significant extensions to the languages.

2.3 bondi and Pattern Calculus

bondi (Jay 2004*a*) is a general-purpose functional programming language designed to allow many forms of genericity. Instead of aggregating features for XML data processing found in the existing wide variety of XML query and transformation languages, bondi has a very general extension to functional language features to achieve a higher degree of modularity and program re-use.

The extension is based on a sound theory, the *Pattern Calculus* (Jay 2004*c*, Jay 2004*b*, Jay 2004*d*), which:

- treats structures and patterns as first-class objects with equal importance to data and functions, allowing them to be referenced and passed as parameters, achieving parameterization of structures, access paths and search patterns;
- allows a generalized form of pattern matching, without requiring the pattern cases to be the same type.

Hence, in bondi, control flow can be determined by structures, not just datum values; and structures and structure-matching patterns are natively well typed, can be used as values, passed around as parameters, and matched in a general way.

In the same way that data and function parameterization make data and function polymorphism possible, the treatment of structures and patterns and the generalization of pattern matching in bondi make possible three new forms of polymorphism: structure polymorphism, path polymorphism and pattern polymorphism. They provide new expressive power and create the opportunity to represent XML processing in a well-typed, highly parametric and highly extensible way. The next section will explain these forms of polymorphism and how they can be used in XML processing.

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 that 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 data and functions as parameters (higher-order functions) or use subtyping to pass objects of varying behavior are plentiful, but until the Pattern Calculus (Jay 2004*c*, Jay 2004*b*, Jay 2004*d*) there has not been general account of how to pass around information about structures, and patterns to match these structures within a typed programming language. The Pattern Calculus, and its implementing language bondi, support all these kinds of parameter passing, achieving polymorphism on data, functions, subtypes, structures, paths and patterns within one typed programming language. The latter three, achieved by parameterizing structures and patterns, are particularly suited to describe XML access paths, and can greatly simplify programming for XML manipulation. This section explains these three new forms of polymorphism by introducing a sequence of XML processing examples requiring deep parameterization, and shows how simple and type-safe it is to design highly-parametric functions for XML data processing.

3.1 A Motivating Scenario

Suppose we have a data repository containing geographical information and we want to carry out the following operation: *Add 1% to the population of all of Canadian cities*. How could we express such an operation?

The first way is what might be called assembly language programming: a specific program that traverses the structure in the repository, finds all of the places where Canadian cities are present, and then finds their population elements and adds 1% to them. The problem is that if we decide to change the problem in any way we have to rewrite and recompile the program.

All high-level programming languages allow the amount by which the populations are to be incremented to be extracted and expressed as a parameter. So we might write something like:

$$IncrementPopsofCanadianCities(1\%)$$

This small change increases the generality of the program in the sense that we can make many different changes without rewriting or recompiling the program. The program is generic with respect to one argument.

Many programming languages also allow us to make the operation that is to be done to the populations of Canadian cities into a parameter as well. So we might write:

$$UpdatePopsofCanadianCities(incrementby, 1\%)$$

Now it is trivial to decrement the populations instead.

The next level of generality is to make the parts of the structure where the function is applied into a parameter as well. So we might write:

$$updateCanadianCities(Pops, incrementby, 1\%)$$

Now it is trivial to increment (or decrement) cities' *areas* instead of their populations. Most query languages, either for databases or for semistructured data, are powerful enough to allow this kind of programming, but many general-purpose languages have trouble because the contexts that define the regions where the function is to be applied are constructed in different ways and look different to the type system.

A further extension is to make the particular units within Canada that are being considered into a parameter. So we might write:

$$updateInCanada(City, Pops, incrementby, 1\%)$$

Now the program is generic in the pattern that describes *where* the increment is to be applied (cities above populations). It will work regardless of whether cities are immediately below countries, e.g., capitals such as Ottawa or Washington D.C., or accessed via intermediate layers such as states or provinces.

Now let us parameterize on the country too:

$$update(CountryName == \text{“}Canada\text{”}, City, Pops, incrementby, 1\%)$$

The code involves a side-condition to check on a related structure.

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

$$update(Canadian\_City\_Pop, incrementby, 1\%)$$

which has a (complex) pattern parameter. 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 a complex pattern *parameter* of the standard *update* function.

This example shows the many levels of need for genericity in processing semi-structured data. Most programming languages and query languages can satisfy some of these needs, but the following subsection will show that bondi is the first to handle them all in one language, and in a natural way.

3.2 Parameterizing 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 (Jay 2004*a*). Language features will be explained as they are used without attempting a full introduction here. As a convention, `a, b, c, d, ...` are used as variables for types and ..., w, x, y, z are variables for values.

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` of type `float->popul`. We can define a function for updating populations by pattern-matching:

```
let (atPopIncrementBy1Percent:popul->popul) x =
    match x with
    | Pop z -> Pop (z * 1.01);;
```

When applied to a term of the form `Pop x` it returns `Pop (x*1.01)`. 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 incrementBy1Percent x = x*1.01;;
let atPopIncrementBy1Percent = atPopApply incrementBy1Percent;;
```

Evaluation of the new version of `atPopIncrementBy1Percent` reduces to the old one by substituting for the variable `f`.

More generally, we can consider increasing populations stored in larger data 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. We use `[x, y, z, ...]` as syntax sugar for `(Cons x (Cons y (Cons z ...)))` and `[ ]` for the empty list `Nil`.

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` as its first argument and applies it to every element of the second argument, a list. `listMap` is defined by pattern-matching over the two list constructors. For example,

```
listMap incrementBy1Percent
```

acts on lists of floats and

```
listMap atPopIncrementBy1Percent
```

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., function

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

whose type includes a type variable `c` representing the structure, e.g., `list`. We say function `map1` is *structure polymorphic*. The definition of `map1` is more complex than its type suggests as it relies on the theory of data structures developed in (Jay 2004*c*).

Even `map1`, however, is not flexible enough for our purposes, since a typical database is not going to be as homogeneous as type `(c popul)`, having only one type of elements. There is no reason to single out populations while ignoring, say, city names and areas.

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

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

Note that the patterns of three matching cases are of different types. This generalized form of pattern matching is allowed by Pattern Calculus with a less-restricted typing requirement (Jay 2004*c*). The first case is the same as `atPop` 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 (e.g., `Cons s t`), and causes both parts of the compound (e.g., `Cons s` and `t`) to be updated, while the final case is used to terminate at atoms of data. They can match different type of structure in each recursive call. For example,

```
updatePops incrementBy1Percent [Pop x1, Pop x2]
```

evaluates to `[Pop x1*1.01,Pop x2*1.01]`; but

```
updatePops incrementBy1Percent ([Pop x1],Pop x2)
```

evaluates to `([Pop x1*1.01],Pop x2*1.01)` even though the populations appear on different levels of the data structure. Thus `updatePops` is *path polymorphic* since it can adapt to different data access paths.

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)->d->d) \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 function `updatePop` can now be defined 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 λP). 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.

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)`, meaning that the function of type `a->b` uses its argument exactly once. Linear terms are explained in detail in (Jay 2004*b*). For this paper, we will pretend that all linear terms are constructors though there are important alternatives. So for now `lin(a->b)` is the type of a constructor with an argument of type `a` for a data structure of type `b`. For example, `Pop` has type `lin(float->popul)`.

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

```
let (check:lin(a->b)->(a->bool)->d->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 XML geographical data conforming to the schema:

```
<xs:element name="cityname" type="xs:string"/>
<xs:element name="popul" type="xs:decimal"/>
    <!-- unit:  thousand people -->
<xs:element name="river" type="xs:string"/>

<xs:element name="city">
    <xs:complexType><xs:sequence>
        <xs:element ref="cityname"/>
        <xs:element ref="popul"/>
        <xs:element ref="river" minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>

<xs:element name="provname" type="xs:string"/>
<xs:element name="province">
    <xs:complexType><xs:sequence>
        <xs:element ref="provname"/>
        <xs:element ref="popul"/>
        <xs:element ref="city" minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>

<xs:element name="countryname" type="xs:string"/>
<xs:element name="country">
    <xs:complexType><xs:sequence>
        <xs:element ref="countryname"/>
        <xs:element ref="popul"/>
        <xs:element ref="province" minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>
```

In an implementation of our approach, a validating XML parser is needed to transform XML data into bondidata format for processing. Assuming such parsing, the above schema can be denoted as bondidata structures:

```
datatype cityname = CityName of string;;
datatype popul = Pop of float;;
(* unit: thousand people *)
datatype river = River of string;;
datatype city = City of cityname * popul * list river;;

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

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

Here `*` represents product type with the usual functional programming convention, and constructor `Pair of a and b` represents pairing data items. `(x, y)` is syntactic sugar for `Pair x y`, and tuple `(x, y, z, ...)` is nested pairs. For programming convenience, we always encode children of an XML element as nested pairs as in the above declarations, e.g. a city element for Kingston are encoded as:

```
City("Kingston",Pop 100.0,["St.Lawrence River"])
```

Now applying `update Pop f` to a piece of geographical data will act on all of the city, province 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 structure type for the complex patterns 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, and most information along the path down to that city is not interesting. Let us call such a partial description of a path a *signpost* since it guides the way to target data items.

For the purposes of encoding the motivating examples, let us consider three sorts of signposts. It will be easy to add more sorts as needed. A *goal* is a constructor whose argument is the singular pattern of the target item of interest. A *stage* is a constructor that constructs a signpost from a leading simple pattern of the path and the rest of the path. A *detour* is a path that has a side-path with a filtering condition to check before continuing on the main path. Thus we obtain a structure:

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

Here "`at`" indicates pattern matching with different forms of the type arguments. `signPost` takes three type arguments, the first two of which can be pairs of types. `detourPath` is a helper structure to represent a filtering condition in a `signPost`:

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

`signPost` and `detourPath` are similar in form to data structures such as `list`, but they are not data structures any more because they contain pattern type `lin(a->b)`, which is not data type. We call them *pattern structures*.

Note that in `signPost` an extra parameter type `c` is used to expose the content type of the final element, the goal, for programming convenience. It enables general programming of computation with such paths as parameters. 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 dpath = DetourGoal CountryName ((==) "Canada");;
let popPath2 = Stage Country
                    (Detour dpath (Stage City (Goal Pop)));;
```

Note that `(==)` is a boolean function of type `a->b->bool`, which takes two arguments, so that `((==) "Canada")` is a boolean function of type `a->bool`.

Now function `check` can be modified to act on `detourPath`, and `update` be modified to act on `signPost`, as follows.

```
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;;
```

Note that function `updates` ("s" stands for signpost) invokes the simple version `update` for singular patterns. It uses pattern matching to explore the structure of a given path pattern, that is, a `signPost`. If the path pattern is a singular pattern, `update` is invoked directly. If the path pattern is a `Stage`, `update` is used to search for the preceding singular pattern, then from the matching points the search for the rest of the path pattern continues. `checkd` also acts in a similar way.

If the path pattern given to function `updates` is a `Detour`, the function checks whether the detour path got a match and whether the content of the match satisfies the carried boolean function. If so, the function goes back to the starting point and continue the search for the rest of the main path pattern. If the detour does not get a match or the carried boolean function fails, the function returns unchanged data. Now it is straightforward to increment all populations of all Canadian cities, if `data` is the data repository containing geographical information:

```
updates popPath2 incrementBy1Percent data
```

Note that this executes independently of the presence of provinces.

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`. In this subsection we define functions for the folding operation, which is the basis of many common operations on heterogeneous data structures. We also show by an example the simplicity of using the folding functions in XML data processing.

A function `foldleft1` can be defined (Jay 2004*c*), similar to `map1`, as:

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

It traverses a homomorphic structure of type `(c b)` with all elements being only one type `b`, applying a given function to the values of all elements it finds to modify the given value of type `a`. For example, given a definition of 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 singular patterns appearing in various contexts of arbitrary heterogeneous data structures, a generalized `foldleftp` for heterogeneous structures can be defined, again simply using three-case pattern matching:

```
let (foldleftp:lin(a->b)->(e->a->e)->e->d->e) \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 complex path pattern (`signPost`) looks like this:

```
let (foldlefts:(signPost a b c)->(e->c->e)->e->d->e) 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 dp1 s1 ->
          if (checkd dp1 w)
               then foldlefts s1 f x w
          else x;;
```

Many essential XML processing operations can be expressed as using `foldleftp` and `foldlefts`. For example, extracting information from XML data based on a search pattern and a filter is the most common kind of XML query. It can be easily implemented by `foldlefts`.

Suppose we want a list of names of cities whose population is bigger than 300 (in units of thousands). This query consists 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 an instance of a `signPost`, and the filter is a boolean function carried by a `Detour` pattern:

```
let dpath = DetourGoal Pop ((>) 300.0);;
let namePath = Stage City (Detour dpath (Goal CityName));;
```

Collecting matching items into a final result is a `foldlefts` operation in bondi. An accumulating function will be given to the folding function as a parameter, to accumulate matching items. Users can use different accumulating functions for different ways of constructing the final result. If the result is to be a list of strings for city names, i.e., of type `list string`, the accumulating function can be as simple as:

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

Of course more sophisticated accumulating functions can be designed, for example to check for duplicates, or to construct results into a structure other than a flattened string list.

Given the pattern and accumulating function, the task is straightforward (again, `data` is the geographical data repository):

```
let nl = foldlefts namePath listInsert [ ] data;;
```

Many other essential XML processing operations, such as extraction while preserving or restructuring original structures, indexing and sorting, are basically folding operations as well and can be implemented in a similar way using the folding functions. More examples are available in one of our earlier reports (Huang, Jay & Skillicorn 2005*b*).

Designing highly-parametric general functions in bondi, such as `update`, `updates`, `foldleftp` and `foldlefts`, is as simple as pattern-matching several cases. Such simple functions allow us to perform a large class of XML search and transformation operations within a general-purpose programming environment easily. These functions can be formalized as library components of bondi. The only task left for users is to map complex search patterns into instances of appropriate pattern structures such as `signPost`, and this may also be automated.

4 More Complex Patterns

Besides designing new generic functions, another way to extend XML processing capability is to include more kinds of patterns. In the previous section we defined a pattern structure, `signPost`, which is able to express a large class of common vertical patterns. To handle more complex patterns, we can add more constructors to `signPost`, or even define new pattern structures as appropriate. In bondi this is a programming task, in contrast to other existing XML processing approaches where new kinds of patterns need new language features at best, and are impossible at worst.

We have experimented how to extend our approach to handle complex patterns in XPath style, vertical regular-expression style, and horizontal regular-expression style. These patterns have been considered individually in other existing approaches but have never appeared fully together in one language. Our extensions for these new patterns only need declarations of new pattern structures and changes to programs processing data using these structures. None of our extensions require any changes to the language bondi itself.

For example, we can declare a pattern structure to represent regular expressions:

```
datatype regexp
    at a b =
    | Single of lin(a->b)
    | Kstar of lin(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;;
```

and use this structure to encode patterns of horizontal regular-expression style. We can design functions for search, update and folding of target data matching such patterns.

Further details about handling complex patterns in XPath style, vertical regular-expression style and horizontal regular-expression style can be found in the report (Huang, Jay & Skillicorn 2005*a*).

5 Conclusion

The strongly-typed general-purpose programming language bondi, based on Pattern Calculus, treats structures and patterns as first-class objects, and allows a generalized form of pattern matching with less restricted typing rules. These increases in expressiveness create new forms of polymorphism, especially path polymorphism and pattern polymorphism. Path polymorphism enables traversal of data with heterogeneous structures, automatically adapting to different data-access paths on the fly in a well-typed manner. Pattern polymorphism allows data-access patterns to be passed as well-typed parameters, and be composed into complex pattern structures.

With the new expressive power, we have shown that we can define general programs using generalized pattern matching and parameterizing structures and patterns, implementing a large class of essential XML processing operations. Compared with those from other existing XML processing approaches, bondi programs are simpler and more modular due to higher parameterization and more freedom for pattern matching. These programs are also safer because static typing is enforced not only on data items and functions, but also on structures and patterns.

With the new expressive power, we have also shown that we can easily create new pattern structures or expand existing ones to handle new kinds of complex patterns in XML manipulation. These pattern structures can be treated as freely as data structures. They can be constructed, pattern-matched, traversed at runtime, and passed as values to parameters, making programming with them very flexible and simple. They carry all necessary type information, enabling static type verification for the programs that use them. Extensions to new kinds of patterns require only programming not, as in the other existing approaches, language design or revision. This makes our approach highly extensible, and applicable for a richer set of complex patterns than other XML query and transformation languages.

Given that our approach manipulates XML within one programming language with simplicity, strong type-safety and high extensibility, it is easy to integrate back-end data-access programming with front-end user-interface programming in a single system. The approach thus represents the first steps to solving the impedance mismatch problem.

Of course, there is still a long way to go to use this approach in practical applications. A lot of implementation effort is required and some issues are still open for further investigation.

- It is not expected that XML data users has to do the bondi programming. They will even not need to know about patterns and pattern structures. A library for common XML computations such as search, update and folding can be built. Pattern structure declarations and constructions could be automated, and XML- and XPath-style expressions could be adopted as syntax sugar.
- Currently only an interpreter for bondi is available. Given the high level of language abstraction and polymorphism, it is desirable to compile bondi programs into a format that can be optimized for performance.
- Examples given in this paper assume that XML data are transformed into bondi data structures in memory. When facing large-scale data, although bondi data structures could also be stored in external repositories, it is not yet clear how to optimize the data access for the performance of the repositories.

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. (1997), 'The Lorel query language for semistructured data', *Int. J. on Digital Libraries* **1**(1), 68–88.

Bancilhon, F. & Maier, D. (1988), Multi-language object-oriented systems: New answers to old database problems, *in* K. Fuchi & L. Kott, eds,

'Future Generation Computers II', Amsterdam, North-Holland.

Benzaken, V., Castagna, G. & Frisch, A. (2003), Cduce: an xml-centric general-purpose language, *in* 'Proc. of 2003 ACM SIGPLAN Int. Conf. on Functional Programming', ACM Press.

Berglund, A., Boag, S., Chamberlin, D., Fernndez, M., Kay, M., Robie, J. & Simon, J. (2005), 'Xml path language (xpath) 2.0 - w3c working draft'. `www.w3.org/TR/2005/WD-xpath20-20050211/`.

Bierman, G., Meijer, E. & Schulte, W. (2004), 'The essence of data access in cω'. `research.microsoft.com/~emeijer/Papers/popl.pdf`.

Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J. & Simeon, J. (2005), 'Xquery 1.0: An xml query language - w3c working draft'.

Clark, J. (1999), 'Xsl transformation(xslt): Version 1.0 - w3c recommendation'.

Clark, J. & DeRose, S. (1999), 'Xml path language (xpath): Version 1.0 - w3c recommendation'. `www.w3.org/TR/xpath`.

Cluet, S., Delobel, C., Siméon, J. & Smaga, K. (1998), Your mediators need data conversion!, *in* 'ACM SIGMOD International Conference on Management of Data', Seattle, Washington, USA, pp. 177–188.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A. & Suciu, D. (1999), 'A query language for XML', *Computer Networks* **31**(11–16), 1155–1169.

Harren, M., Raghavachari, B., Shmueli, O., Burke, M., Sarkar, V. & Bordawekar, R. (2004), XJ: Integration of XML processing into Java, *in* 'Proc. WWW2004', New York, NY, USA.

Hosoya, H. & Pierce, B. (2003), 'Xduce: A typed XML processing language', *ACM Transactions on Internet Technology* **3**(2), 117–148.

Huang, F. Y., Jay, C. B. & Skillicorn, D. B. (2005*a*), Dealing with complex patterns in XML processing, Technical Report 2005-497, School of Computing, Queen's University. `www.cs.queensu.ca/TechReports/Reports/2005-497.pdf`.

Huang, F. Y., Jay, C. B. & Skillicorn, D. B. (2005*b*), Programming with heterogeneous structure: Manipulating XML data using bondi, Technical Report 2005-494, School of Computing, Queen's University. `www.cs.queensu.ca/TechReports/Reports/2005-494.pdf`.

Jay, C. B. (2004*a*), 'bondi web-page'. `www-staff.it.uts.edu.au/~cbj/bondi`.

Jay, C. B. (2004*b*), 'Higher-order patterns'. `www-staff.it.uts.edu.au/~cbj/Publications/higherorderpatterns.pdf`.

Jay, C. B. (2004*c*), 'The pattern calculus', *ACM Trans. Program. Lang. Syst.* **26**(6), 911–937.

Jay, C. B. (2004*d*), 'Unifiable subtyping'. `www-staff.it.uts.edu.au/~cbj/Publications/unifablesubtyping.pdf`.

Meijer, E., Schulte, W. & Bierman, G. (2003), Unifying tables, objects and documents, *in* 'Proc. DP-COOL 2003', Uppsala, Sweden.

Robie, J., Lapp, J. & Schach, D. (1998), 'Xml query language (XQL)'. `www.w3.org/TandS/QL/QL98/pp/xql.html`.

Wadler, P. (2004), 'Links'. `homepages.inf.ed.ac.uk/wadler/papers/links/links-blurb.pdf`.