Dealing with Complex Patterns in XML Processing

F.Y. Huang C. B. Jay D.B. Skillicorn

June 2005
External Technical Report
ISSN-0836-0227-
2005-497

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared June 1, 2005

Abstract

The handling of search patterns for data access is critical to XML processing. Most efforts to integrate XML processing into general-purpose programming languages add different features to existing languages for different kinds of patterns, or even create completely new languages. The resulting languages only work for certain kinds of patterns and are hard to extend to others in a well typed manner. This poses burdens for both language designers and programmers, and restricts XML applications. In this paper we present an alternative approach that handles many kinds of patterns within the same framework. The key idea is to express complex patterns by pattern structures in one general-purpose programming language based on the pattern calculus. With this approach, adding a new kind of patterns is just a matter of programming, not language design; programs can be highly parametric over well-typed complex patterns.

Dealing with Complex Patterns in XML Processing

F.Y. Huang and C.B. Jay and D.B. Skillicorn

Abstract: The handling of search patterns for data access is critical to XML processing. Most efforts to integrate XML processing into general-purpose programming languages add different features to existing languages for different kinds of patterns, or even create completely new languages. The resulting languages only work for certain kinds of patterns and are hard to extend to others in a well typed manner. This poses burdens for both language designers and programmers, and restricts XML applications. In this paper we present an alternative approach that handles many kinds of patterns within the same framework. The key idea is to express complex patterns by pattern structures in one general-purpose programming language based on the pattern calculus. With this approach, adding a new kind of patterns is just a matter of programming, not language design; programs can be highly parametric over well-typed complex patterns.

1 Introduction

A basic operation in XML data processing is to locate data items by their position in a structured context, usually described by a pattern or sequence of patterns. In some situations search patterns can be as simple as matching a single type of element, for example in the search for all population items in a geographical dataset. In many other situations search patterns are more complex, but complex patterns can usually be decomposed into simpler ones. For example, the search for complex pattern "population of individual cities in Canada" includes searches for a 'country' element whose 'country-name' element has the value "Canada", with 'city' elements and 'population' elements under it.

There are two ways to compose simpler patterns into more complex ones, reflecting the two different kinds of relationships between elements in an XML structure. The first are *vertical* patterns, such as in the search for the population of cities of Canada. Such patterns match structural relationships of XML elements from different hierarchical levels. Location paths expressed in the popular XPath [8] language fall in this category.

The second are *horizontal* patterns, for example, in the search for cities having child elements for cityname, population, either timezone or continent, and zero or more running-through rivers, i.e., the pattern [cityname, population, timezone—continent, river*]. Such a pattern is to match the appearance of sibling elements on the same level of an XML hierarchy.

Vertical and horizontal patterns can be mixed into even more complex search patterns. For example, the search pattern "contact phone numbers of cityhalls of Canadian cities having child elements for cityname, population and zero or more running-through rivers" is a combination of vertical and horizontal patterns.

There are three approaches to expressing and executing such queries. The first uses untyped representations of XML data and low-level string-based programming. Each new query/transformation requires generating completely different code. The second uses special-purpose query languages. XML data is still viewed as untyped, but the language provides primitives to express many of the relationships between XML data elements. The third tries to integrate XML queries and transformations into a general-purpose programming language. XML data is represented by a typed data structure, and query and transformation programs are also typesafe. None of these approaches has been completely successful: the first is fully expressive, but too cumbersome for practical use; the others all express some kinds of queries naturally, but cannot (or cannot easily) express other kinds of queries.

XML query languages are special-purpose languages, similar to SQL in relational data processing. Such languages are usually untyped, descriptive, and have limited programming power. Because direct description of a pattern is always straightforward, it is not difficult to define (or extend) a query language to describe queries with new kinds of patterns as needed, although query languages so far focus primarily on vertical patterns. However, because there is no underlying paradigm relating the form of a query to the form of the implementation needed to handle it, implementation of new forms of queries requires designing new algorithms. Furthermore, type checking is limited, and static type checking impossible. Often the results of an XML query are used by other software systems. Conveying these results to other systems usually requires format conversion (often via low-level string representations) creating performance bottlenecks.

These problems suggest extending general-purpose programming languages so that XML data and XML queries can be handled in native mode. This removes the transformation bottleneck, allows static typing of programs, reduces runtime overhead and reduces the risk of catastrophic failure or unintended information leakage.

However, incorporating XML into a programming language is not an easy task. Typical efforts extend an existing general-purpose programming language, or create a new one, with special kinds of types for XML data. Careful design, formal proof of language soundness, and change to the compiler are required. As a result, adding a new kind of pattern to the language is expensive, both for language designers and programmers. It has also proven difficult to design queries that are expressive enough within the type systems of conventional functional or object-oriented languages.

In this paper we present an approach that can handle a rich set of patterns within a single language, in a well-typed manner. The key idea is to define *pattern structures*, which are used to express complex patterns and are first-class typed objects within the programming language bondi[1]. bondi supports the pattern calculus and treats structures and patterns as first-class objects.

We have shown, in an earlier report [16], that treating structures and patterns as first-class objects, the same as other programming entities, enables natural expression of search patterns as pattern structures, and such expression makes programming for XML data simple and well typed.

In this paper we show that:

- Existing approaches for XML processing only handle limited kinds of search patterns, with either poor extensibility to other kinds of patterns or poor type-safety.

- The expression of search patterns as pattern structures is general enough to accommodate all kinds of patterns in XML processing in a native and well-typed manner.

[1]bondi is pronounced Bond-eye and is the name of a famous beach in Sydney.

- The approach is fully extensible. Adding a new kind of pattern to XML processing is a programming task, rather than one of language design.

The organization of the rest of the paper is as follows. Section 2 reviews related work. Section 2.1 briefly reviews existing languages for XML processing, demonstrating their poor extensibility for patterns and poor type-safety; Section 2.2 reviews bondi the general-purpose language used for our approach, and pattern structures, which we introduced in earlier work. Section 3 presents the use of pattern structures for various kinds of search patterns, and demonstrates how easy it is to include a new kind of patterns into XML processing. Three important and interesting kinds of patterns are used for the demonstration: vertical patterns in the style of XPath in Section 3.1; vertical patterns in the style of regular expressions in Section 3.2; and horizontal patterns in the style of regular expressions in Section 3.3. We draw some conclusions in Section 4.

2 Related Work

2.1 Existing Approaches for XML Processing: Poor Extensibility and Poor Type Safety

In the early years of XML, untyped special-purpose XML query languages such as Lorel [1], YATL [9], XML-QL [10], XQL [17] and XSLT [7] were invented to handle query and transformation of XML data. They are more or less capable of handling vertical patterns, such as the patterns of XPath [8] style and patterns with self-nested elements. Expressing some simple cases of horizontal patterns is also possible, although it is not the focus of these languages. However, these languages are typically untyped, handling both tag names and element content on as low a level as strings. They have very limited programming power, and are unable to express sophisticated computations using XML data.

In many settings, the queries and their results must be passed to other application programs at runtime. These other programs are usually developed in general-purpose programming languages. Data is usually passed in a low-level format such as a string for flexibility and generality. Use of such low-level formats means that the programs on both ends must transform data; the data must be type-checked at runtime at the destination; and the input/output performance of most systems is orders of magnitude below memory copy. The extra costs of communication between front- and back-end processes has been called the *impedance mismatch problem.*

In recent years, attempts have been made to merge XML processing into general-purpose programming languages, but only with a focus on specific kinds of patterns that cannot be extended easily in a type-safe way. Typical approaches use special types to represent XML data and special expressions for specific search patterns in addition to regular programming language features. The most recent efforts include XJ [14], XQuery [4, 6] and Cω [11] focusing on vertical patterns, and XDuce[15] and CDuce[23] focusing on horizontal patterns.

XJ [14] is an extension of Java, a general-purpose programming language, with XML data types and XPath expressions, capable of handling vertical patterns conforming to XPath 1.0 [8]. It treats XML element types as regular Java classes, and uses special strings containing XPath expressions for search patterns. Static typing of these XML types and embedded pattern strings against XML schemas is enforced by a special type checker. Since this type checking is against XML schema types, not native Java types, pattern expressions are not necessarily safe in Java. Further, the special type checking requires that schemas for XML data are always available and trustworthy, which is unrealistic in many situations. Because search patterns are just strings, there is the potential to include patterns other than XPath expressions, but only in a untyped way, or at best typed against XML schemas, not Java.

XQuery [4, 6] is designed to be a language for XML processing analogous to SQL for relational database processing. It aggregates lots of features from older XML query languages and SQL, while its data model and type system fully conform to XML [5] and XML Schema [12] specifications. It is a superset of XPath 2.0 [3], making XPath expressions native, and so fully capable of handling vertical patterns of XPath form. XQuery is designed as a query language although equipped with some basic functional programming features. When XQuery is not used in a user-interactive setting, its expressions are supposed to be embedded in host programs in other languages to perform the queries, especially when sophisticated processing, beyond XQuery's limited power, is needed. Although XQuery is a strongly typed language and an implementation can optionally type-check pattern expressions against XML schemas, the impedance mismatch and static typing problem still exists, just as for XJ. 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 [11] 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 three kinds of data. It introduces three kinds of types: stream, anonymous struct and choice, roughly equivalent to list, heterogeneous tuple and sum types in functional languages. Vertical pattern `city/population` can be expressed as `city.population` where `city` is a class holding an anonymous struct and `population` is a component of that struct. To accommodate XPath-style patterns, Cω also introduces filter expressions such as `country[name=="Canada"]`, and transitive query expressions such as `country...city` for cities appearing at arbitrary depth below countries. So in terms of patterns in XML processing, its expressiveness is roughly equivalent to XPath 1.0 [8] without backward axes. In contrast to XJ and XQuery, pattern expressions in Cω are native, consisting of identifiers all having native types, and can be well typed in Cω itself. There is no impedance mismatching problem since everything can be handled in one language. However, if one needed to handle other kinds of patterns, for example XPath with backward axes, self-nested structures or horizontal patterns, new features would have to be added to the language; The type system would have to be modified; so would the compiler.

XDuce [15] and CDuce [23] are functional programming languages with regular expression types added to general-purpose functional language features. Regular expressions are used to define XML elements, and horizontal patterns to match the elements. XML data and patterns can be well typed natively and handled inside the one language. However, vertical patterns are not easy to express in these languages without significant extensions.

2.2 bondi and Pattern Structures

bondi[18] is a general-purpose functional programming language designed to allow many forms of genericity, not specifically for XML processing. Instead of aggregating features for XML data or XML data structures found in the existing wide variety of XML query and transformation languages, bondi has a very general extension to regular functional language features. The extension is based on a sound theory, the *pattern calculus* [19–21], which allows a very general form of pattern matching without requiring the pattern cases be the same type, and treats data structures and structure-matching patterns as first-class objects with equal importance to data and functions. Hence, in bondi, control flow can be determined by structures, not just values; and structures and patterns are natively well typed, can be passed around as parameters, and be used for programming using generalized pattern matching.

Treating data structures and structure-matching patterns as first-class objects in bondi creates new power for programming and opens the opportunity for us to represent complex patterns for XML processing in a well-typed way and to design highly parametric programs over these typed patterns. In an earlier report [16], we presented the idea of declaring a pattern structure called *signPosts* in bondi. This is used to express certain kind of complex vertical patterns in a well-typed way and pass them around as parameters for XML processing. The `signPosts` structure is capable of expressing a useful subset of XPath patterns, for example location path `country[name=="Canada"]//city//population`. Programming for XML processing with `signPosts` patterns is simple and highly modular. It is based on pattern-matching of the cases of `signPosts` constructors.

The following section will show that we can declare pattern structures in a similar way for *any* kind of complex patterns, and use them for XML processing, without any change to bondi.

3 Pattern Structures in bondi for Complex Patterns

We have shown in our earlier report [16] that a pattern structure, `signPosts`, can be declared in the general-purpose functional programming language bondi to express certain complex patterns and pass them around as parameters in a well-typed manner. The report also serves as a quick introduction to the bondi syntax.

In contrast to other existing XML processing approaches that only focus on some kinds of patterns and are hard to extend to others, our approach of using pattern structures is fully extensible to any pattern. In this section, we show that we can handle different kinds of patterns in a similar way to `signPosts`, using the same idea of declaring pattern structures in bondi and using them for XML processing. These extensions do not require any changes to the bondi language itself. Our demonstration focuses on patterns that have been considered in other existing languages for XML processing, including vertical patterns in XPath style, vertical patterns in regular-expression style and horizontal patterns in regular-expression style, but our approach is obviously applicable to any other kinds of patterns.

3.1 Vertical Patterns in XPath Style

The specification of a location path in XPath 1.0 [8] covers a wide range of vertical patterns. A location path expression consists of a sequence of location steps delimited by "/". Each step may contain three kinds of components:

Step ::= AxisName::NodeTest [Predicate]*

Each kind of component can be represented in bondi by a pattern structure, and so can the location steps and location path expressions. In this section, we show how to declare pattern structures for them and program with these structures. We only consider XML elements, leaving out attributes, namespaces, comments, processing instructions etc. which are either irrelevant to the issue or transformable to elements.

3.1.1 Declaring the Pattern Structures for XPath Patterns

Axes can be represented by constant constructors of type `axis`, as follows:

```
datatype axis = | Child
                | Descendant
                | DescendantOrSelf
                | Following
                | FollowingSibling
                | Parent
                | Ancestor
                | AncestorOrSelf
                | Preceding
                | PrecedingSibling
                | Self;;
```

A node test can be a test for a node name or a node type. Since we only consider XML elements, a node type test is not necessary. A node name test can be against the wildcard "*" or a specific (element) name:

```
datatype nametest a b = | NodeName of lin(a->b)
                        | Any;;
```

Notice the unusual `lin(a->b)`, which represents a "linear function type `a->b`". It is the form of types for singular high-order patterns in bondi. More details can be found in [16, 19]. When a structure declaration contains singular pattern types, it is no longer a data structure but a pattern structure.

The specification of predicate expressions in XPath 1.0 [8] is very general. We limit the cases we consider for clarity, by restricting each expression to contain at most one location path and no library function calls. So expressions like [sales/price $*$ inventory/quantity $>$ 350.0], [sales/price $<$ 9.0 | inventory/quantity $>$ 30] and [position() = 2] are excluded.

A typical predicate expression is of the form [sales/price $<$ 9.0]. This is the same as `detourPath` in the definition of `signPosts` [16], except that each stage of the path here is a (more expressive) `step` rather than a singular pattern of type `lin(a->b)`. Formally, we declare a recursive pattern structure:

```
datatype predicate (a1,a2) (b1,b2) =
    | PredGoal of step (a1,a2) (b1,b2) and (a1->bool)
    | PredStage of step a1 b1 and predicate a2 b2;;
```

where `step` is another pattern structure we will declare shortly. To represent a list of (zero or more) predicates in a well-typed way, we declare:

```
datatype predlist (a1,a2) (b1,b2) =
    | NilPred (* for empty list *)
    | ConsPred of predicate a1 b1 and predlist a2 b2;;
```

Now we are ready to declare the pattern structure for location steps:

```
datatype step (a1,a2) (b1,b2) =
    | Step of axis and nametest a1 b1 and predlist a2 b2;;
```

and that for location paths:

```
datatype locpath
    at a b c =
    | PathGoal of step (c,a) b
    at (a1,a2) (b1,b2) c =
    | PathStage of step a1 b1 and locpath a2 b2 c;;
```

In the declaration of `locpath`, an extra parameter type `c` is used to expose the content type of the element of the final step of a location path, enabling general programming of any computation with such a path as parameter. This is the same idea as in `signPosts` [16], except that a `locpath` uses `step` rather than singular pattern of type `lin(a->b)` for each stage of the path. This declaration does not handle absolute paths, but it is natural to keep track of current node in functional programming so relative paths are expressive enough in most cases.

The declarations for `predicate`, `predlist` and `step` are mutually recursive, because they are mutually recursive in XPath specification. Each step of a location path can contain a list of predicates, and a predicate in turn can contain a location path which is a sequence of steps, just like in the following XPath (abbreviated) location path for national population:
country[//city/cityname="Kingston"]/population

3.1.2 Expressing XPath Patterns

XML elements can be represented using datatypes in bondi and element names become constructor names. For example `Country`, `City`, `CityName` and `Pop` are constructors for the datatypes `country`, city, cityName and population. See Appendix A for declarations of these datatypes and their data. We choose to declare a datatype for an XML element so that its constructor takes one parameter, which is a tuple of XML elements representing children of the declared element. Given all these, the predicate [//city/cityname="Kingston"] in the location path for national population can be encoded as:

```
let step4city = Step Descendent (NodeName City) NilPred;;
let step4cname = Step Child (NodeName CityName) NilPred;;
let (isKingston:  string->bool) s = (s == "Kingston");;
let pred4cntry = PredStage step4city (PredGoal step4cname isKingston);;
```

and the whole location path for national population can be encoded as:

```
let step4cntry = Step Child (NodeName Country) (ConsPred pred4cntry NilPred);;
let step4pop = Step Child (NodeName Pop) NilPred;;
let poppath = PathStage step4cntry (PathGoal step4pop);;
```

3.1.3 Programming with XPath Patterns

With path patterns expressed using pattern structures, programming for XML processing is quite straightforward and modular, and we can get highly parametric programs with well-typed complex patterns as parameters. We demonstrate the design of a function updatepath that updates the content of every XML element that matches a given XPath pattern, such as updating the population of every city with name "kingston" under a country element. First we design some helper functions for clarity.

Function checkstep takes three parameters s, f and x. It verifies, in a piece of XML data x, whether there exists an element matching the pattern (a single location step) s, with the content of this element satisfying a boolean function f. Let us start from the simple case, considering only a descendant for the axis of a pattern step, and ignoring the possibility of a wildcard in the name test:

```
let (checkstep:  step (a1,a2) b -> (a1->bool) -> c -> bool) s f x =
    match s with
    | Step Descendant (NodeName N) pl ->
        match x with (
        | N z -> (checkpredlist pl z) && (f z)
        | y z -> checkstep s f y || checkstep s f z)
    | t -> false;;
```

Note the simplicity in the code for traversing the piece of XML data x (the inner pattern matching). Only two cases are needed for the pattern matching, and they can be of different types. bondi is the only language that allows such a general form of pattern matching. Also note that the pattern t in the final case is like a wildcard that can match anything, catching objects that do not match any preceding cases.

In function checkstep, another function checkpredlist is invoked, which we define as follows:

```
let (checkpred:  (predicate a b) -> c -> bool) p x =
    match p with
    | PredGoal s f -> checkstep s f x
    | PredStage s sp -> checkstep s (checkpred sp) x;;

let (checkpredlist:  (predlist a b) -> c -> bool) pl x =
    match pl with
    | NilPred -> true
    | ConsPred p spl -> (checkpred p x) && (checkpredlist spl x);;
```

These two functions evaluate predicate `p` and predicate list `pl` respectively against a given piece of XML data `x`. Note that the above three functions are mutual recursive. Using them, we can define update functions with path patterns easily. Function `updatestep` updates elements that match a simple pattern, a single location step. Again, for now we only consider a descendant for an axis in a step:

```
let (updatestep:  step (a1,a2) b -> (a1->a1) -> c -> c) s f x =
    match s with
    | Step Descendant (NodeName N) pl ->
        match x with (
        | N z -> if (checkpredlist pl z) then N (f z)
        | y z -> (updatestep s f y) (updatestep s f z) )
    | t -> x;;
```

Function `updatepath` updates elements that match a composite pattern, a location path:

```
let (updatepath:  (locpath a b c) -> (c->c) -> d -> d) lp f x =
    match lp with
    | PathGoal s -> updatestep s f x
    | PathStage s slp -> updatestep s (updatepath slp f) x;;
```

Similarly, we can define other functions that compute with `locpath`. For example, function `checkpath` verifies that there exists one element matching a given path pattern, and that the content of that element satisfies a given boolean function:

```
let (checkpath:  (locpath a b c) -> (c->bool) -> d -> bool) lp f x =
    match lp with
    | PathGoal s -> checkstep s f x
    | PathStage s slp -> checkstep s (checkpath slp f) x;;
```

3.1.4 Covering More Patterns

Adding more cases of patterns only incurs more programming work. For example, adding the other axis cases can be done in a modular way, by adding cases to the pattern matching in the program code. Let us consider child in addition to descendant as a demonstration. One more matching case needs to be added to `checkstep` and `updatestep` respectively; no changes to the other functions are needed. Recall that multiple child elements are represented as one tuple (nested pairs of type `binprod`, see Appendix A):

```
let (checkstep:  step (a1,a2) b -> (a1->bool) -> c -> bool) s f x =
    match s with
    | Step Descendant (NodeName N) pl ->
        ...  ...
    | Step Child (NodeName N) pl ->
        match x with (
        | N z -> (checkpredlist pl z) && (f z)
        | Pair y z -> checkstep s f y || checkstep s f z )
    | t -> false;;

let (updatestep:  step (a1,a2) b -> (a1->a1) -> c -> c) s f x =
    match s with
    | Step Descendant (NodeName N) pl ->
        ...  ...
    | Step Child (NodeName N) pl ->
        match x with (
        | N z -> if (checkpredlist pl z) then N (f z)
        | Pair y z -> Pair (updatestep s f y) (updatestep s f z) )
    | t -> x;;
```

Because of space limitations, we will not give an implementation for the full XPath specification, but will only demonstrate the most common and straightforward forms. For axes, we have only demonstrated with child and descendant; for node test, we have demonstrated with a test for a specific element name under the default namespace, leaving out wildcard and node type tests; for predicates, we consider only predicate expressions each containing at most one location path and no function calls. XPath patterns with other features can be handled in a similar way, but programming with them may require novel algorithms that are beyond the scope of this paper. There is a rich literature on XPath evaluation algorithms, for example [2, 13, 22], and we expect that most algorithms can be adapted to our approach, with a certain amount of programming work, but without the need to change bondi.

3.2 Vertical Patterns in Regular-Expression Style

In this subsection we demonstrate an extension to the location path declared in previous subsection, to accommodate vertical patterns of regular-expression style. This extension, although significant, can be done by making `locpath` more sophisticated and modifying relevant functions. Once again, only programming is required.

Although the XPath specification covers a wide range of vertical patterns, there remain many vertical patterns outside the specification. Among them, vertical patterns of regular-expression style are the most interesting. XPath expressions only include the case of path concatenation, not the cases of path alternation and Kleene star (alternation is only allowed for predicate expressions). In practice, there are situations where patterns composed by alternation and Kleene star are important. For example, a dealer may like to update both sales prices and purchase costs of products by certain percentage due to inflation, requiring a path pattern like “sales/price | supplier/cost”; a document analyzer may want to extract all titles of sections in papers, needing a path pattern like “section*/title”; sometimes the path to be nested is itself a composite pattern already. Other languages are not able to accommodate these without requiring significant changes to the languages' type systems and so compilers. In bondi, all that is required is changing the pattern structure `locpath`, adding a new helper structure `regpath`, and then updating the relevant programs accordingly.

The pattern `regpath` represents regular expressions of location steps, without exposing the content type of the final step. So it cannot be used for a complete path pattern but only for a partial one without a final step. Constructors `RegGoal`, `RegKstar`, `RegStage` and `RegAlter` represent single step, Kleene star of a partial path, path concatenation, and path alternation respectively:

```
datatype regpath
    at a b =
    | RegGoal of step a b
    | RegKstar of regpath a b
    at (a1,a2)(b1,b2)
    | RegStage of regpath a1 b1 and regpath a2 b2
    | RegAlter of regpath a1 b1 and regpath a2 b2
```

For a path pattern with a final step, we want the content type of the final step to be unique for programming convenience. Hence we do not include the case of path alternation (with two possible final steps), and the case of zero occurrence for Kleene star (with empty final step):

```
datatype locpath
    at a b c =
    | PathGoal of step (c,a) b
    | PathPlus of locpath a b c  (* representing r+; the final step can't be empty *)
    at (a1,a2) (b1,b2) c =
    | PathStage of regpath a1 b1 and locpath a2 b2 c;;
```

Programs that use `locpath` also need to be updated. For example, the function `checkpath` can be modified as:

```
let (checkpath:  (locpath a b c) -> (c->bool) -> d -> bool) lp f x =
    match lp with
    | PathGoal s -> checkstep s f x
    | PathPlus s -> checkstep s f x || checkstep s (checkpath lp f) x
    | PathStage (RegGoal s) slp -> checkstep s (checkpath slp f) x
    | PathStage (RegKStar r) slp ->
        checkpath slp f x || checkpath (PathStage r lp) f x
    | PathStage (RegStage r1 r2) slp ->
        checkpath (PathStage r1 (PathStage r2 slp))
    | PathStage (RegAlter r1 r2) slp ->
        checkpath (PathStage r1 slp) || checkpath (PathStage r2 slp);;
```

3.3 Horizontal Patterns in Regular-Expression Style

In contrast to vertical patterns that match hierarchical data structures, a horizontal pattern matches the appearance of sibling elements on the same hierarchical level under one parent element. For example, city[cityname, population, river*] is a horizontal pattern. New languages have been created to specifically handle XML processing with horizontal patterns of regular-expression style [15, 23]. Regular-expression types are introduced in those approaches, incurring a significant amount of work for type-system correctness and compiler implementation. These languages are not able to handle vertical patterns, and there is no easy way to extend them to do so.

In this subsection, we show that horizontal patterns can be handled using a pattern structure in bondi again without the need to change the language or create a new one. For demonstration, we consider four cases of horizontal regular expression: singular patterns, Kleene star of singular patterns, pattern concatenation, and pattern alternation.

```
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;;
```

Treating such regular expressions as horizontal patterns, we can design a program `localmatch` to check the existence of such patterns in a given tuple of XML elements and stop immediately at the first mismatch. Recall that multiple children of one element are represented as a tuple of elements, which is a nested pairs of type `binprod` in bondi.

```
let (localmatch:  regexp a b -> binprod(c,d) -> bool) r x =
    match r with
    | Single P -> (
        match x with
        | Pair (P y) z -> true
        | P z -> true
        | z -> false )
    | Kstar P -> (
        match x with
        | Pair (P y) z -> localmatch (Kstar P) z
        | z -> true )
    | Concat (Single P) r2 -> (
        match x with
        | Pair (P y) z -> localmatch r2 z
        | z -> false )
    | Concat (Kstar P) r2 -> (
        match x with
        | Pair (P y) z -> localmatch r z
        | z -> match r2 z )
    | Concat (Concat r3 r4) r2 -> localmatch (Concat r3 (Concat r4 r2))
    | Concat (Altern r3 r4) r2 ->
        localmatch (Concat r3 r2) || localmatch (Concat r4 r2))
    | Altern r1 r2 -> (localmatch r1 x) || (localmatch r2 x);;
```

In this function, pattern matching against data structures is simple, no more than three cases most of the time. Pattern matching against the pattern structure exhausts the cases of constructors. A more useful function `globalsearch` checks the existence of a regular horizontal pattern in an entire piece of arbitrary XML data. It can be defined using `localmatch`:

```
let (globalsearch :  regexp a b -> c -> bool) r x =
    match r with
    | Pair y z -> if (localmatch r (Pair y z)) then true
                  else (globalsearch r y) || (globalsearch r z)
    | y z -> if (localmatch r (Pair y z)) then true
             else (globalsearch r y) || (globalsearch r z)
    | z -> false;;
```

4 Conclusion

We have shown how to use pattern structures to represent complex patterns in XML data processing, enabling these patterns be treated as freely as data structures. They can be constructed, pattern matched, destroyed, traversed at runtime, and passed around as values, making programming with them very flexible and simple. They carry all necessary type information, enabling static type verification for the programs that use them.

We have also shown that this approach is applicable for a richer set of patterns than other XML query and transformation languages. In other languages, adding the ability to recognize new patterns required extending the type system, and hence altering the compiler. In our approach, adding the ability to recognize new patterns requires declaring new pattern structures and related helper functions, or perhaps simply modifying existing ones. Increasing expressive power is achieved by programming, not by language extension, one of the goals of generic programming.

This approach is made possible by the pattern calculus which elevates data structures and patterns to the level of first-order objects, the same as data values and functions. This gives more freedom to programming while preserving typing. What we have explored here, XML query and transformation, is just one of the many opportunities this new programming power opens to us.

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] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, , and V. Josifovsi. Streaming xpath with forward and backward axes. In *Proc. of International Conference on Data Engineering (ICDE)*, March 2003.

[3] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndez, Michael Kay, Jonathan Robie, and Jrme Simon. XML Path Language (XPath) 2.0 - W3C Working Draft, February 2005.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language - W3C Working Draft, February 2005.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition) - W3C Recommendation, February 2004.

[6] D. Chamberlin. XQuery: A query language for XML. *IBM Systems Journal*, 41(4), 2002.

[7] J. Clark. XSL Transformation(XSLT): Version 1.0 - W3C Recommendation, November 1999.

[8] J. Clark and S. DeRose. XML Path Language (XPath): Version 1.0 - W3C Recommendation, November 1999.

[9] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *ACM SIGMOD International Conference on Management of Data*, pages 177–188, Seattle, Washington, USA, 1998.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. *Computer Networks*, 31(11–16):1155–1169, 1999.

[11] E. Meijer and W. Schulte and G. Bierman. Unifying Tables, Objects and Documents. In *Proc. DP-COOL 2003*, Uppsala, Sweden, August 2003.

[12] David C. Fallside. XML Schema Part 0: Primer - W3C Recommendation, May 2001.

[13] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.

[14] M. Harren, B. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML Processing into Java. In *Proc. WWW2004*, New York, NY, USA, May 2004.

[15] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[16] F.Y. Huang, C.B. Jay, and D.B. Skillicorn. Programming with heterogeneous structure: Manipulating XML data using bondi. Technical Report 2005-494, School of Computing, Queen's University, March 2005. `http://www.cs.queensu.ca/TechReports/Reports/2005-494.pdf`.

[17] D. Schach J. Robie, J. Lapp. Xml query language (xql), 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[18] C. B. Jay. bondi web-page, 2004. `www-staff.it.uts.edu.au/~cbj/bondi`.

[19] C. B. Jay. Higher-order patterns, 2004. `www-staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf`.

[20] C. B. Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.

[21] C. B. Jay. Unifiable subtyping, 2004. `www-staff.it.uts.edu.au/~cbj/Publications/unifable_subtyping.pdf`.

[22] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proc. of Workshop on XML-Based Data Management at EDBT 2002, Prague, Czech Republic*, volume 2490 of *LNCS*, March 2002.

[23] V. Benzaken and G. Castagna and A. Frisch. Cduce: an XML-Centric General-Purpose Language. In *Proc. of 2003 ACM SIGPLAN Int. Conf. on Functional Programming.* ACM Press, 2003.

A Example XML Data in bondi Format

We show here a concrete example of XML data in bondi format. Some of the datatypes and data are used in the demonstration in Section 3.

Datatype definitions give some possible datatypes for geographical data. Precise knowledge of the data schema is not required at the time of programming or of data processing.

```
datatype population = Pop of float;; (* unit: thousands *)
datatype cityname = CityName of string;;
datatype river = River of string;;
datatype city = City of cityname * population * list river;;
datatype provname = ProvName of string;;
datatype province = Prov of provname * population * list city;;
datatype countryname = CountryName of string;;
datatype country = Country of countryname * population * list province;;
datatype world = World of list country;;
```

Note that constructor `City` takes only one parameter, which is a tuple of data of three types. Tuple is a shorthand form for nested pairs in bondi (also in many other functional languages): tuple `(a,b,c)` of type `x*y*z` is equivalent to `Pair (Pair a b) c`, where `x*y` is the infix form of binary product type `binprod (x,y)`, and `(a,b)` is the infix form of `Pair a b`:

```
datatype binprod (x,y) = Pair of x and y;;
```

The following are the concrete data for the datatypes declared above:

```
let r1 = River "Niagara River";;
let r2 = River "St.  Lawrence River";;
let r3 = River "Great Cataraqui River";;

let cityn1 = CityName "Montreal";;
let cityn2 = CityName "Kingston";;
let cityn3 = CityName "Niagara";;
let pop1 = Pop 3610.0;;
let pop2 = Pop 150.0;;
let pop3 = Pop 390.0;;
let montreal = City (cityn1, pop1, [r2]);;
let kingston = City (cityn2, pop2, [r2, r3]);;
let niagara = City (cityn3, pop3, [r1]);;

let pn1 = ProvName "Ontario";;
let pn2 = ProvName "Quebec";;
let pop5 = Pop 12390.0;;
let pop6 = Pop 7540.0;;
let ontario = Prov (pn1, pop5, [kingston, niagara]);;
let quebec = Prov (pn2, pop6, [montreal]);;

let cn1 = CountryName "Canada";;
let pop8 = Pop 31940.0;;
let canada = Country (cn1, pop8, [ontario, quebec]);;
let data = World [ canada ];;
```