XMλ
A Functional Language for Constructing and Manipulating XML Documents[*]

Erik Meijer
University of Utrecht
mailto:erik@cs.uu.nl
http://www.cs.uu.nl/~erik

Mark Shields
Oregon Graduate Institute
mailto:mbs@cse.ogi.edu
http://www.cse.ogi.edu/~mbs

Abstract

XML has been widely adopted as a standard language for describing *static* documents and data. However, many application domains require XML, and it's cousin HTML, to be filtered and generated *dynamically*, and each such domain has adopted a language for the tasks at hand. These languages are often ill-suited, unsafe, and interact poorly with each other.

In this paper we present XMλ, a small functional language which has XML documents as its basic data types. It is expressly designed for the task of generating and filtering XML fragments. The language is *statically typed*, which guarantees every document it generates at run-time will conform to its DTD, but also uses *type inference* to avoid the need for many tedious type annotations. The language is also *higher-order* and *polymorphic*, which allows many common programming patterns to be captured in a small highly reusable library. Furthermore, the language uses *pattern-matching* so that XML fragments may be deconstructed into their components just as easily as they are constructed.

We present the language by a series of worked examples. A formal definition and an implementation are in preparation.

[*]Submitted to *USENIX Annual Technical Conference*, 2000

1 Introduction

XML is a language for specifying both the *structure* and *content* of documents, and has become a standard for representing domain-specific metadata. Examples include

- XHTML, for describing HTML documents [31];
- MathML, for describing mathematical expressions [14];
- XSL, for describing style sheets for other XML documents [33];
- SVG, for describing graphical diagrams [21]; and
- SOAP, for describing parameter information for remote procedure calls [5].

More applications for XML arise as the standard gains even wider acceptance.

Key to the success of XML is its notion of *document conformance.* A document's content may be checked against its *Document Type Definition* (DTD), and ill-formed documents rejected.

Though XML is well suited to describing static documents, it lacks any support for *dynamic* documents, *i.e.*, documents which are produced on-the-fly. Unfortunately, for many application domains this is the *modus operandi.* Most Internet applications use SQL or XQL to perform queries on a relational or structured database in response to a client request. This information is then typically processed by server-side CGI Perl scripts, Active

Server Pages (ASP) [30], or Java Server Pages (JSP) [17] to generate XML or HTML from the query results. Separate languages such as XSL or CSS[12] are then used to apply layout, font, color and other graphical features to the resulting document.

This need to mix static documents and dynamic behavior has resulted in a proliferation of languages which are often ill-suited for manipulating fragments of XML:

- Languages such as Perl [27] represent XML fragments as raw strings. Documents are constructed by concatenating strings, and deconstructed by elaborate regular expressions. This is clearly too low level.
- By contrast, many other languages represent XML fully abstractly using DOM [3] . However, now programs become a tangle of calls to a DOM library, making it difficult to understand what's going on. Thus this representation seems too abstract.
- In all of these languages, the connection between a generated document and its DTD is lost. It is left to the programmer to ensure that every generated document is conforming. In a large application using many languages this is typically infeasible

Why not use just one language with XML documents as its basic data type?

In this paper, we try to systematically apply the techniques of modern typed functional programming language design [10, 23, 8] to the field of XML document processing. The resulting language, XMλ, stems from these four key connections:

DTDs	are	XMλ types
Document fragments	are	XMλ values
Document conformance	is	type-correctness
Dynamic documents	are	XMλ programs

We don't suggest XMλ should replace XML, but rather build upon it.

Another way documents become dynamic involves embedded scripts which are activated on the client-side in response to user interaction. Again, this has resulted in a proliferation of languages, such as Java Script, Java Applets and VBScript. And again, the results are often unsatisfactory. We also believe XMλ would be appropriate in this setting if extended by constructs for *staged computation* [22, 20]. Such an extension would allow both server-side and client-side actions to be coded in the same type-safe language, without there being any confusion as to what code is executed when and by whom. However, in this paper we shall concentrate on the core features of XMλ, and defer such extensions to future work.

1.1 Overview

We start our informal introduction of XMλ by giving a few examples of constructing dynamic documents (Section 2) using expressions and higher-order functions. Then in section 3 we show how to transform and take apart documents using pattern matching and discriminators. Section 3.2 describes a spam filter in XMλ, our main example. In section 4 we introduce polymorphism and extensible sums and discuss the subtleties associated with these advanced features. We close with a short overview of related and future work.

2 Constructing XML

In this section we'll introduce the key features of XMλ for constructing documents dynamically.

2.1 XML

Consider a simple DTD describing e-mail messages:

```
<!DOCTYPE Message[
<!ENTITY  % Recipients "(To|Bcc)">
<!ELEMENT Message (From,%Recipients;,
                   Subject?,Body)>
<!ELEMENT From    (#PCDATA)>
<!ELEMENT Bcc     (#PCDATA)>
<!ELEMENT To      (#PCDATA)>
<!ELEMENT Subject (#PCDATA)>
<!ELEMENT Body    (P*)>
<!ELEMENT P       (#PCDATA)>
]>
```

The `Message` element declaration describes an XML document tree with root `<Message>`...`</Message>`

whose children consist of a `From` element, followed by zero or more `To` or `Bcc` elements, followed by an optional `Subject`, and finally a `Body`. The only primitive data type we consider is `#PCDATA`, which stands for "parsed character data", *i.e.*, character strings without embedded elements.

Though the bodies of element declarations resemble arbitrary regular expressions, the XML standard restricts them to be *1-non-ambiguous.* This ensures a document may be checked for conformance using a deterministic Glushkov automaton (Section 4) [2]. The `Message` DTD is 1-non-ambiguous. An example *ambiguous* DTD is

```
<!ELEMENT Foo (Bar*,Bar?)>
<!ELEMENT Bar (#PCDATA)>
```

Should the last `Bar` in a sequence within a `Foo` belong to `Bar*` or `Bar?`...?

A well-formed document which conforms to the `Message` DTD is:

```
<Message>
  <From>Erik</From>
  <To>Mark</To>
  <Bcc>Erik</Bcc>
  <Subject>XMLambda</Subject>
  <Body>
    <P>
      The Glushkov automaton for the
      Email DTD is indeed deterministic.
    </P>
    <P>
      Unfortunately, there is not enough
      space in this message to include
      the proof.
    </P>
  </Body>
</Message>
```

Note how strings of text are entered directly, tags are escaped, elements are well-nested, and that the sequence of elements within the top-most `Message`

```
From, To, Bcc, Subject, Body
```

is valid for the regular expression (after expanding the definition for `Recipients`)

```
(From,(To|Bcc)*,Subject?,Body)
```

2.2 DTDs in XMλ

Our first task is to represent DTDs as types within XMλ. There is a trend toward using XML syntax for everything, even for DTD declarations as in XML-Schema [32], or for specifying document templates such as in XSL. We think it's not a good idea to write programs which manipulate or define abstract syntax in abstract syntax as well. Instead, the syntax (and semantics!) of XMλ is modeled after modern functional languages such as SML and Haskell.

The `Message` DTD appears in XMλ as a number of *type* and *element* declarations:

```
TYPE Recipients = {To|Bcc}*
ELEMENT Message = (From, Recipients,
                   Subject?,Body)
ELEMENT From    = String
ELEMENT Bcc     = String
ELEMENT To      = String
ELEMENT Subject = String
ELEMENT Body    = P*
ELEMENT P       = String
```

Here `Recipients` is declared as a type abbreviation: each occurrence of it within other types is simply replaced by the type `{To|Bcc}*`. For technical reasons (see Section 4.4) we delimit sums using braces rather than brackets.

`Message` is declared as an element with the given body type. An element declaration simultaneously introduces both its tag and type, written identically. The other elements are declared similarly. Types themselves resemble regular expressions, just as in XML. The primitive type `String` corresponds with XML's `#PCDATA`.

2.3 Documents in XMλ

Documents in XMλ are written essentially as they are in XML. That is, since most of a document is text we retain the feature of XML that text is primitive and tags must be escaped. This is in contrast to most other programming languages where program terms are taken as primitive and it is the strings which must be escaped [16].

Hence the example `Message` we gave above is a valid

XMλ value, and it is well-typed with type `Message`.

2.4 Escaped expressions

In XML *entities* can be used as a very simple abstraction mechanism. For instance, we can define two entities

```
<!ENTITY me  "Erik">
<!ENTITY him "Mark">
```

and then refer to `Erik` (or `Mark`) using the escape `&me;` (or `&him;`). However, entities cannot be parametrized, and their values cannot computed dynamically. Hence XML entities are purely an abbreviation device.

XMλ uses a technique similar to that of JSP and ASP whereby an arbitrary expression `e` may be embedded within a document by escaping it as `<%= e %>`. By binding the variables `me` and `him` to values of type `String`, we can construct the same document as we had before:

```
me :: String
me = "Erik"

him :: String
him = "Mark"

<Message>
  <From><%= me %></From>
  <To><%= him %></To>
  <Bcc><%= me %></Bcc>
  <Subject>XMLambda</Subject>
  <Body>
    <P>The Gluskov ...</P>
    <P>Unfortunately, ... </P>
  </Body>
</Message>
```

Notice how within elements strings are unescaped, but within expressions (which includes the program's top level) strings must be put inside quotation marks:

type	`String`
Elements	`Hello World!`
Expressions	`"Hello World!"`

In XMλ strings are not the only document fragments we may construct within an embedded expression. Indeed *any document fragment whatsoever* may be constructed within an embedded expression, provided it is well-typed. This includes sequences, tuples, optionals, sums, and even other elements.

For example, inside elements we may construct a sequence of elements by simple juxtaposition. The 1-non-ambiguity constraint ensures that we always know where the sequence starts and ends. Inside expressions, we must enumerate the elements within an explicit list:

type	`P*`
Elements	`<P>...</P> ... <P>...</P>`
Expressions	`[ <P>...</P>, ..., <P>...</P> ]`

By naming the list of paragraphs which constitutes the body of our example message as `paras`:

```
paras :: P*
paras =
  [ <P>The Glushkov automaton
    for the Email DTD is indeed
    deterministic.</P>
  , <P>Unfortunately, there is
    not enough space in this
    message to include the
    proof.</P>
  ]
```

we may shorten the message even more:

```
<Message>
  <From><%= me %></From>
  <To><%= him %></To>
  <Bcc><%= me %></Bcc>
  <Subject>XMLambda</Subject>
  <Body><%= paras %></Body>
</Message>
```

Notice how the type of `paras`, `P*`, is consistent with its use within the `Message` element. We will explore this further in Section 2.6.

Tuples are similar to sequences. Inside elements, we may construct a tuple of elements by simple juxtaposition. Inside expressions, we must enumerate the elements in an explicit tuple using parenthesis and commas.

Inside elements, we can either write or omit an optional element. Again, the 1-non-ambiguity constraint ensures we know whether an element was left out. Inside an expression, we must explicitly tag the value of options using the discriminators `Just` or `Nothing`:

type	`Subject?`
Elements	`<Subject>...</Subject>` (empty)

Expressions	`Just <Subject>...</Subject>`
	`Nothing`

Inside elements, we can write any member of a sum. The 1-non-ambiguity constraint ensures that we know which of the alternatives was chosen. Inside an expression, we must explicitly *inject* the value into a sum:

type	`{To|Bcc}`

Elements	`<To>...</To>`
	`<Bcc>...</Bcc>`

Expressions	`Inj <To>...</To>`
	`Inj <Bcc>...</Bcc>`

More details about XMλ sum types will be given in Section 4.4.

One thing we can't do is construct an ill-bracketed element within an expression, even if the final result could conceivably yield a well bracketed element. For example:

```
illegal = <Subject>XMLambda

<Message>
  ...
  <%= illegal %> </Subject>
  ...
</Message>
```

is rejected. Even if such a program were syntactically valid, what possible type could we give to the "psuedo-element" `<Subject>XMLambda`?

So far have only dealt with embedded *expressions*. What about embedded side-effecting *commands*? Firstly, since documents themselves are just values, and hence free of side effects, we need not leave the expression world in order to create them. This is in contrast to, for example, ASP, in which documents are created as a side effect by calls to procedures (cf JSP's `<%`, `%>` brackets). Secondly, the field of functional programming has developed the technique of *monadic programming*[24] which allows commands to be treated as first-class values. In a future paper we will show how the same technique may be applied in the context of XMλ programs.

2.5 A little more sophistication

Escaped expressions may be any expression, not just the simple kind of constant expression we used in the previous examples. Indeed we have the power of a higher-order functional language at our disposal.

Instead of defining `paras` as a list of paragraphs, consider its slightly more primitive representation as a list of string fragments, concatenated together using the explicit string concatenation operator `(++)`:

```
rawParas :: String*
rawparas =
  [ "The Glushkov automaton for the " ++
    "message DTD is indeed deterministic."
  , "Unfortunately, there is " ++
    "insufficient space in this "++
    "message to include the proof."
  ]
```

In order to use `rawParas` to define the body of our message we must somehow convert each entry from a `String` to a `P`. Thankfully the library provides such a function in the form of `map`, which in this case we will use at type `(String -> P) -> (String* -> P*)`. That is, given a function f from `String` to `P`, `map` f yields a function from `String*` to `P*`. So we're done if we can build a function to embed any string inside a `P`. This is easily done using the *anonymous* function

```
\s -> <P><%= s %></P>
```

Here `\` signals that we are constructing a function taking an argument `s` and yielding a paragraph `<P><%= s %><P>`.

Putting this all together we have:

```
<Message>
  <From><%= me %></From>
  <To><%= him %></To>
  <Bcc><%= me %></Bcc>
  <Subject>XMLambda</Subject>
  <Body>
    <%=
      map (\s -> <P><%= s %></P>)
          rawParas
    %>
    </Body>
  </Message>
```

This example shows the great power which comes from treating XML documents as ordinary values. In particular, we may define functions to dynamically create document fragments which may themselves be created dynamically. Notice how the expression using `map` which appeared within an escape again contained a document `<P>...</P>` which itself contained an embedded expression `<%= s %>`! There is (within practical bounds) no static limit to this nesting of elements and expressions.

Contrast this with ASP and JSP where there is no inherent notion of document once we are in the VisualBasic (ASP) or Java (JSP) world. Once you escape, you're out in the cold.

2.6 Well-typedness

Most approaches to dynamic document generation provide no guarantee at all that generated documents confirm to their DTD. Usually, the value embedded expressions are just inserted as strings.

In ASP for instance we can write the following script (using JavaScript syntax) that injects an expression of the wrong type into the document, and will even generate syntactically invalid HTML:

```
  <Message>
    ...
    <Subject>
     <%
       Response.Write("<H1>Strange<H2>");
       Response.Write("indeed</H1></H2>");
     %>
    </Subject>
    ...
  </Message>
```

The ASP processor will happily run this program and send the resulting "HTML" to the client.

XML entities also lack the protection of a strong type system. As an entity just abbreviates a piece of string, it is possible to construct documents whose non-well-formedness will only be discovered dynamically when parsing the document:

```
  <!ENTITY me "Erik<From>">
  <!ENTITY him "</From>Mark">

  <Message>
    <From>&me;</From>
    <To>&him;</To>
    ...
  </Message>
```

One of the most important lessons that we have learned from software engineering is that catching errors early is extremely important [1]. Strong static typing is a very effective means to catch errors early, and viewed in this light the fact that almost all popular scripting languages [9] are untyped is rather disappointing.

In XMλ, the type-checker will reject any attempts to build a non-conforming document, even if it is constructed dynamically. For example:

```
  <Message>
    <From><%= me %></From>
    <To><%= him %></To>
    <Bcc><%= me %></Bcc>
    <Subject><%= paras %></Subject>
    <Body><%= paras %> </Body>
  </Message>
```

is rejected as ill-typed:

```
Type error: The term
  <%= paras %>
of type
  P*
is used in an incompatible context of type
  String
```

Our type-checker takes advantage of the 1-non-ambiguity constraint when type-checking elements with embedded expressions using a Glushkov automaton. Unlike in XML, where such an automaton would transition based on *tags*, our automaton will instead transition based on *types*.

Suppose we explicitly construct the list of recipients as follows (since we are explicitly building a sequence of a sum-type, we must use `Inj`):

```
  recipients :: {To | Bcc}*
  recipients =
    [ Inj <Bcc>Erik</Bcc>
    , Inj <To>Mark</To>
    ]
```

and that we have a function `mkSubject` which explicitly constructs a `Subject` value from a string:

```
mkSubject :: String -> Subject?
mkSubject = \s ->
  Just (<Subject><%= s %></Subject>)
```

Now, given the expression:

```
<Message>
  <From>Erik</From>
<!-- start of {To|Bcc}* -->
  <To>Scotty</To>
  <%= recipients %>
  <To>Bill</To>
<!-- end of {To|Bcc}* -->
  <%= mkSubject "XMLambda" %>
  ...
</Message>
```

the XMλ type-checker runs the Glushkov automaton for `Message` over the body, checking that each child has an appropriate type. In this case, the automaton wants to recognize an element of type `From`, then zero or more elements of type `{To | Bcc}`, possibly a `Subject` and finally a `Body`. When the automaton encounters `<%= recipients %>` it recognizes zero or more elements of type `{To | Bcc}` because the expression `recipients` has type `{To | Bcc}*`. *I.e.*, when the value of `recipients` is spliced in, it is guaranteed to consist of zero or more `To` or `Bcc` elements. Similarly, the automaton knowns when the recipients list has finished, because the expression `<%= mkSubject "XMLambda" %>` has type `Subject?`:

```
<Message>
  <From>Erik</From>
<!-- start of (To|Bcc)* -->
  <To>Scotty</To>
<!-- start expansion -->
  <Bcc>Erik</Bcc>
  <To>Mark</To>
<!-- end of expansion -->
  <To>Bill</To>
<!-- end of (To|Bcc)* -->
  <Subject>XMLambda</Subject>
  ...
</Message>
```

So in principle, our type-checker uses the standard technique for checking validity of XML documents, but extended to take care of embedded expressions. However, as we will see shortly in Section 4, the introduction of polymorphic functions over documents makes life a little more complicated than this.

The idea of static type checking has further applications for XML too. In many case, server-side dynamically generated documents will themselves contain embedded client-side scripts. For example the document may contain a form with embedded scripts for entry validation. In this case it is even more important to guarantee that the resulting document is correct, as most browsers are very savvy towards bad HTML, but much less clever when executing bogus script. The full XMλ language will use the notion of typed staged computation to provide solid guarantees about the type correctness of dynamically generated script fragments.

3 Patterns and Discriminators

Besides generating documents from scratch, many applications create new documents by inspecting and transforming other documents. In XMλ we use pattern matching and sum discrimination to dissect documents.

3.1 Patterns

Consider a function `getPara` to match a `P` element and return its body as a `String`. In XMλ it may be written using a *pattern* instead of just a variable in the function argument:

```
getPara :: P -> String
getPara = \<P><%= p %></P> -> p
```

We say that the variable `p` is *bound* by the function `getPara`. The expression `getPara <P>Hello World!</P>` then evaluates to the string `Hello World!`.

To get the body of a `Message` we can define a similar function which matches on a `Message` and returns the list of all the paragraphs in its body. The novelty here is that we now use a *nested* pattern to match inside the one of the children of an element:

```
getBody :: Message -> P*
getBody = \<Message>
```

```
                <%= from :: From %>
                <%= to :: Recipient* %>
                <%= subject :: Subject %>
                <Body><%= body %></Body>
              <Message> -> body
```

All four variables from, to, subject and body are bound by this pattern. For technical reasons we must give the XMλ type checker some hints as to the types of the first three variables, otherwise the pattern would be ambiguous.

3.2 Example: spam filter

It's time to put some of the pieces together in a more substantial example. We often find messages like this in our mailbox of a morning:

```
<Message>
  <From>mohwo@hotmail.com</From>
  <To>Mark</To>
  <Subject>
    We need home workers like you!
  </Subject>
  <Body>
    <P>
      Are you sick of earning a piddling
      amount of cash as a research student?
    </P>
    <P>
      Think you're being exploited as
      glorified child labor?
    </P>
    <P>
      Then get rich fast from the
      comfort of your own home! We need
      honest and reliable research students
      like you to stuff and mail envelopes.
      It's so simple even a PhD could manage
      it! And it sure beats flipping burgers.
    </p>
  </Body>
</Message>
```

Let's get rid of them. The standard XMλ library already contains (in this case):

```
  map :: (P -> String) -> P* -> String*
  concat :: String* -> String
  toLowerCase :: String -> String
  words :: String -> String*
  contains :: String* -> String* -> Bool
  filter :: (Message -> Bool) ->
            Message* -> Message*
  not :: Bool -> Bool
```

The words function breaks a String into the list of words within it, contains ws ws' yields True if any word in ws appears within ws', and filter p as discards every element a of list as for which p a yields False.

By combining these library functions and XMλ's pattern matching facilities we can easily construct a function to scan our mailbox for messages containing suspicious words, which are most likely spam. We first need a function to extract and normalise the words within a message's Body:

```
  getContents :: Message -> String*
  getContents =
    ( words
    . toLowerCase
    . concat
    . map getPara
    . getBody
    )
```

The "." here denotes function composition: (f . g) x yields f (g x). Defining complicated functions by the composition of simpler functions is a typical idiom in functional programming. It is very similar in spirit to the Unix style of writing complex shell scripts by piping together simpler scripts. Indeed, you may like to think of "." as a back-to-front "|".

Given function getContents, we can now easily define the isSpam test, and use it to filter away all spam messages from our mailbox:

```
  suspiciousWords :: String*
  suspiciousWords =
    [ "money", "rich", ...]

  isSpam :: Message -> Bool
  isSpam = \message ->
     contains suspiciousWords
       (getWords message)

  killSpam :: Message* -> Message*
  killSpam = filter (not . isSpam)
```

Imagine how long-winded this code would look if we had used XML syntax for XMλ's functions as well!

3.3 Discriminators

Pattern matchers on elements may be combined in order to discriminate amongst particular values. For example:

```
isSelf :: To -> Bool
isSelf = { \<To>Erik</To> -> True
         , \otherwise -> False
         }
```

Here two functions on `To` elements are being combined. The first yields `True` when applied to the value `<To>Erik</To>`. The other always yields `False`. Their combination is a function which tests for a self addressed `To` element.

In a discriminating function `{f, g, ...}`, the functions `f`, `g`, etc, are tried in turn for a match against the argument, and the body of the first matching function is evaluated. All the functions must have the same type.

We may also discriminate between alternatives of a sum-type. Suppose we want to discard all the `Bcc` recipients of a message. We first define the `isBcc` function by combining two functions which pattern match on `Bcc` and `To` elements respectively into a single function that accepts an argument of type `{Bcc | To}`:

```
isBcc :: {Bcc | To} -> Bool
isBcc =
  { \<Bcc><%= s %></Bcc> -> True
  | \<To><%= s %></To> -> False
  }
```

Depending on the actual *type* of the argument to function `isBcc`, one of the two alternatives is chosen. For example:

```
isBcc <Bcc>Erik</Bcc>
```

evaluates to `True`. If we were to try calling `isBcc` with an argument of type other than `To` or `Bcc`, the type-checker would complain.

The desired function that discard all the `Bcc` recipients is then simple:

```
stripBCC :: Recipients -> Recipients
stripBCC = filter (not . isBCC)
```

Unlike for `{f, g, ...}`, each function `f`, `g`, etc in `{f | g | ...}` must have the same result type but a *distinct* argument type.

4 Polymorphism

So far in our examples we have been assuming some of our library functions may be reused at many different types. For example, we have silently used the function `map` at the two types:

```
map :: (String -> P) -> String* -> P*
map :: (P -> String) -> P* -> String
```

In practice we give functions like `map` only one type, namely its *most general* type, which we describe using *polymorphism*:

```
map :: forall a:Type, b:Type.
       (a -> b) -> a* -> b*
```

Here `a` and `b` are *type variables* of *kind* `Type` (see section 4.4), and `forall a:Type, b:Type` tells us that the function `map` has the type `(a -> b) -> a* -> b*` for arbitrary types `a` and `b`. Notice how both of the types `map` was used at are *instances* of this general type.

Some other common library functions which exploit polymorphism include:

```
filter   :: forall a:Type.
            (a -> Bool) -> a* -> a*
contains :: forall a:Type.
            a* -> a* -> Bool
```

Polymorphism is essential to the utility of XMλ, as without it users would be required to rewrite the same functions each time they wished to manipulate new element types. Polymorphism in XMλ has much the same utility as *inheritance* in object-oriented languages: both are a tremendous aid to reuse.

In a similar vein, we can also parameterize element and type declarations over types. This allows us to define generic elements such as:

```
ELEMENT ListPair a b = (a*, b*)
```

Thus just as we may reuse functions at many types, we may also reuse elements at many types. However this cannot be done naïvely!

4.1 Regular expression inference?

So far all of the type annotations we have given in our examples have been completely optional. Though they aid the readability of a program, XMλ's type checker is just as comfortable inferring them. However, when we allow parameterized elements such as `ListPair`, we can run into severe type-checking problems. For example, what type should be assigned to:

```
<ListPair>
  <Subject>Strange</Subject>
  <Subject>Problem</Subject>
  <%= "Indeed" %>
</ListPair>
```

There are many possibilities:

- `ListPair Subject String`: interpreting the document as a 2-element list of type `Subject*` and a 1-element list of type `String`.
- `ListPair (Subject, Subject, String) a`: interpreting the document as a 1-element list of type `(Subject, Subject, Int)*` followed by an empty list of arbitrary type `a`.
- `ListPair a (Subject, Subject, String)`: interpreting the document as an empty list of arbitrary type `a` followed by a 1-element list of type `(Subject, Subject, String)*`.

but none of these is obviously the best.

Hence we need to provide the type-checker with enough clues to do its work. One solution is to construct polymorphic elements using explicit expressions. For the element type `ListPair` this means that we would write:

```
<ListPair>
  <%= [ <Subject>Strange</Subject>
      , <Subject>Problem</Strange>
      ]
  %>
  <%= "Indeed" %>
</ListPair>
```

Alternatively, we can tell the type-checker at which (monomorphic, non-ambiguous instance, see Section 4.2) type we intend to use the element. In our case we can write:

```
<ListPair Subject String>
  <Subject>Strange</Strange>
  <Subject>Problem</Strange>
  <%= "Indeed" %>
</ListPair>
```

Of course we can abbreviate this last element using a type synonym and instead write:

```
TYPE S = ListPair Subject String
<S>
  <Subject>Strange</Strange>
  <Subject>Problem</Strange>
  <%= "Indeed" %>
</S>
```

4.2 Ambiguity

As we have seen before, a very important condition on element types is their non-ambiguity. A related problem with polymorphic elements is that we can determine an element body type is 1-non-ambiguous only when its type is sufficiently monomorphic.

Let's look again at our example polymorphic element `ListPair`:

```
ELEMENT ListPair a b = (a*, b*)
```

and define a function which removes the head of both lists in the pair:

```
tails = \<ListPair>
           <%= as :: a* %> <%= bs :: b* %>
         </ListPair> ->
  <ListPair>
    <%= tail as %> <%= tail bs %>
  </ListPair>
```

Here `tail :: forall a:Type. a* -> a*` is the polymorphic function that returns all but the very first element of a sequence. However, we cannot give `tails` the fully polymorphic type:

```
tails :: forall a:Type b:Type.
         ListPair a b -> ListPair a b
```

since many instances of `ListPair a b`, such as `ListPair String String`, are ambiguous.

4.3 Qualified types

This situation is reminiscent of the classical polymorphic equality problem in languages such as Miranda and SML. It is impossible to naïvely define a truly *polymorphic* equality function `equals :: forall a:Type . a -> a -> Bool` which determines for *any* type `a` whether two values of type `a` are equivalent. For example, how could we define equality on function types, or for user-defined types? A very elegant way out is to restrict the polymorphism to types which admit equality. In Standard ML, type-variables which range over types admitting equality are written as `''a`, `''b`, etc. The ML type for the equality function then becomes:

```
equals :: ''a -> ''a -> Bool
```

Haskell type classes [26, 7] generalize this idea by allowing arbitrary *type predicates* to constrain the polymorphism of type variables. In Haskell, we write the type of `equals` as:

```
equals :: Eq a => a -> a -> Bool
```

which indicates that the function `equals` can only be instantiated at types `a` for which the predicate `Eq a` holds.

We happily reuse this approach and introduce the type predicate `Deterministic a`, which restricts `a` to a 1-non-ambiguous type. Now we can give the correct type to our `tails` function, namely:

```
tails :: forall a:Type, b:Type.
  Deterministic (a*,b*) =>
    ListPair a b -> ListPair a b
```

Any attempt to use `tails` at an ambiguous type will be caught by the type checker. The precise details about the typing rules and satifiability of the `Deterministic` predicate is outside the scope of this paper.

4.4 Extensible Sums

Recall the function `isBcc` defined earlier:

```
isBcc :: {Bcc | To} -> Bool
isBcc =
  { \<Bcc><%= s %></Bcc> -> True
  | \<To><%= s %></To> -> False
  }
```

This function is too specific; it would much nicer if we could make it polymorphic on “the rest of the sum”, *i.e.*, we would like to write:

```
isAnyBcc =
  { \<Bcc><%= s %></Bcc> -> True
  | otherwise -> False
  }
```

The question is what type do we give `isBcc`? Again, the functional programming community provides the solution [4, 18, 29].

We have already been using type constants such as `Int`, `Bool`, and `String`, type variables such as `a` and `b`, and type expressions such as `Int -> Bool` and `a*`. Type expressions are similar to program expressions, so it is natural to ask “what is the *type* of a type expressions?” We call these meta-types *kinds*. For example, the kind of the type `Int` is `Type`, the option type constructor `?` and the list type constructor `*` have kind `Type -> Type` while the function space constructor `->` has kind `Type -> Type -> Type`. Note that `* -> ?` is not a valid type - we cannot assign it a kind!

To deal with *extensible sums* in XMλ, we introduce a new kind `Row` of extensible rows, together with a constant `Empty` of kind `Row`, a binary sum-extension operator `|` of kind `Type -> Row -> Row`, and the sum type constructor `{_}` of kind `Row -> Type`. For example, we have:

```
Int : Type
Int | Empty : Row
{ Int | Empty } : Type
```

Often we abbreviate sum types by omitting the trailing `| Empty`.

We've already seen type variables which range over all `Type`s. Now to express the type of `isAnyBcc` we use a type variable ranging over all `Row`s:

```
isAnyBcc :: forall r:Row .
            (...) => {Bcc | r} -> Bool
```

We are almost there: the problem is that we cannot instantiate `r` to `Int | Empty` as the argument type to `isAnyBcc` would then be `{Int | Int | Empty}`, which is ambiguous. Thankfully though we already have a solution to this problem: we simply extend our `Deterministic` type predicate to reject ambiguous sum types:

```
isAnyBcc :: forall r:Row .
  Deterministic { Bcc | r } =>
    { Bcc | r } -> Bool
isBcc =
  { \<Bcc><%= s %></Bcc> -> True
  | otherwise -> False
  }
```

5 Future and related work

Though expressive, our language so far is very limited and not really suited for real-world applications.

- First of all we have to extend XMλ to support full XML, in particular element attributes. We believe that implicit parameters [11] are a very elegant mechanism for dealing with attributes.
- As we have remarked before we will use staged computation for type-safe separation server-size and client-side computation.
- To interact with the real world, full XMλ will support monads.
- An important topic for future research are generic document transformations. The approach of Hinze [6] seems very promising in this respect.
- For programming in-the-large we need name spaces and modules.
- We have some interesting ideas on specifying layout by using constraints over the real line.

There are a few other XML-related functional programming languages and combinator libraries abound. Wallace and Runciman [28] propose an interesting set of combinators to transform XML documents represented as Haskell data types. Since the expression language of XMλ is very close to Haskell, it is straightforward to implement these combinators in XMLamda as well. Other functional programming toolkits for processing XML are LAML [16], *fxp* [15] and Tony [13]. Wadler [25] mentions several yet unpublished language proposals. Another attempt to well-typed generation of dynamic documents is given by Sandholm and Schwartzbach [19]. They only deal with monomorphic HTML documents however.

Acknowledgements

This work was partially supported by the Dutch Science Foundation (NWO) under contract 600-60-110-10 (which allowed Mark to visit Erik) and by the US Air Force under contract F19628-96-C-0161 and by the Department of Defense (which allowed Erik to visit Mark).

References

[1] B. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[2] A. Brüggemann-Klein and D. Wood. Unambiguous regular expressions and SGML document grammars. Technical Report 337, Computer Science Department, University of Western Ontario, London, Ontario, Canada, Nov. 1992.

[3] Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1, 1 October 1998.

[4] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report NOTTCS-TR-96-3, University of Nottingham, Languages and Programming Group, Department of Computer Science, Nottingham NG7 2RD, UK, Nov. 1996.

[5] H. W. Group. SOAP: Simple object access protocol. http://msdn.microsoft.com/workshop/-xml/general/SOAP_V09.asp, 1999.

[6] R. Hinze. A new approach to generic functional programming. In *POPL'00*, 2000.

[7] M. P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

[8] S. P. Jones and J. H. (eds). Report on the Language Haskell'98. htpp://www.haskell.org/-report, February 1999.

[9] R. Khare, editor. *Scripting Languages: Automating the Web*, volume 2 of *World Wide Web Journal*. O'Reilly&Associates, Inc., Spring 1997.

[10] P. J. Landin. The next 700 programming languages. *CACM*, 9(3):157–164, March 1966.

[11] J. R. Lewis, , M. B. S. E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *POPL'00*, 2000.

[12] H. W. Lie and B. Bos. *Cascading Style Sheets (2n Edition)*. Addison-Wesley, 1999.

[13] C. Lindig. Tony - a xml parser and pretty printer. http://www.cs.tu-bs.de/softech/-people/lindig/software/tony.html, 1999.

[14] Mathematical markup language (mathml) 1.01 specification. http://www.w3.org/TR/REC-MathML, 7 July 1999.

[15] A. Neumann. fxp – Processing Structured Documents in SML. In *1st Scottish Functional Programming Workshop*, 1999. http://www.informatik.uni-trier.de/-neumann/Papers/sfp99.pdf.gz.

[16] K. Nørmark. Programming world wide web pages in scheme. http://www.cs.auc.dk/ normark/scheme.

[17] E. Peligrí-Llopart and L. Cable. Java server pages specification. http://java.sun.com/-products/jsp/index.html, 1999.

[18] D. Rémy. Type inference for records in a natural extension of ML. Technical Report MS-CIS-90-73, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1990.

[19] A. Sandholm and M. I. Schwartzbach. A type system for dynamic web documents. In *POPL'00*, 2000.

[20] M. Shields, T. Sheard, and S. P. Jones. Dynamic typing as staged type inference. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, San Diego, California, 19–21 Jan. 1998.

[21] Scalable vector graphics (svg) 1.0 specification. http://www.w3.org/1999/08/WD-SVG-19990812, 12 August 1999.

[22] W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. *Lecture Notes in Computer Science*, 1443, 1998.

[23] R. Tennent. *Pprinciples of Programming Languages*. Prentice Hall, 1981.

[24] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[25] P. Wadler. The next 700 markup languages (invited talk). In *USENIX Workshop on Domain Specific Languages*, October 1999.

[26] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[27] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl (2n Edition)*. O'Reilly&Associates, Inc., 1996.

[28] M. Wallace and C. Runciman. Haskell and xml: Generic combinators or type-based translation? In *ICFP*, 1999.

[29] M. Wand. Complete type inference for simple objects. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 37–44. IEEE, June 1987. Corrigendum in *Third Annual Symposium on Logic in Computer Science*, page 132, 1988.

[30] R. Wodaski. Asp technology feature overview. http://msdn.microsoft.com/workshop/-server/asp/aspfeat.asp, 27 August 1998.

[31] XHTML-1.0: The extensible hypertext markup language, a reformulation of html 4.0 in xml 1.0. http://www.w3.org/TR/xhtml1, 24 November 1999.

[32] XML Schema Part 1: Structures. http:-//www.w3.org/TR/xmlschema-1, 5 November 1999.

[33] Extensible stylesheet language (xsl) specification. http://www.w3.org/TR/WD-xsl, 21 April 1999.