XDuce: A Typed XML Processing Language

(Preliminary Report)

Haruo Hosoya
Department of CIS
University of Pennsylvania
hahosoya@cis.upenn.edu

Benjamin C. Pierce
Department of CIS
University of Pennsylvania
bcpierce@cis.upenn.edu

1. INTRODUCTION

Among the reasons for the popularity of XML is the hope that the static typing provided by DTDs [13] (or more sophisticated mechanisms such as XML-Schema [14]) will improve the safety of data exchange and processing. But, in order to make the best use of such typing mechanisms, we need to go beyond types for *documents* and exploit type information in static checking of *programs* for XML processing.

In this paper, we present a preliminary design for a statically typed programming language, *XDuce* (pronounced "transduce"). XDuce is a tree transformation language, similar in spirit to mainstream functional languages but specialized to the domain of XML processing. Its novel features are *regular expression types* and a corresponding mechanism for *regular expression pattern matching*. Regular expression types are a natural generalization of DTDs, describing, as DTDs do, structures in XML documents using regular expression operators (i.e., *, ?, |, etc.). Moreover, regular expression types support a simple but powerful notion of *subtyping*, yielding a substantial degree of flexibility in programming. Regular expression pattern matching is similar to ML pattern matching except that regular expression types can be embedded in patterns, which allows even more flexible matching.

In this preliminary report, we show by example the role of these features in writing robust and flexible programs for XML processing. After discussing the relationship of our work to other work, we briefly sketch some larger applications that we have written in XDuce, and close with remarks on future work. A formal definition of the core language can be found in the full version of this paper [6].

2. PROGRAMMING IN XDUCE

We develop a series of examples of programming in XDuce, using regular expression types and regular expression pattern matching.

2.1 Regular Expression Types

2.1.1 Values and Types

XDuce's values are XML documents. A XDuce program may read in an XML document as a value and write out a value as an XML document. Even values for intermediate results during the execution of the program have a one-to-one correspondance to XML documents (besides some trivial differences).

As concrete syntax, the user has two choices: XML syntax or XDuce's native syntax. We can either write the following XDuce value (we assign it to the variable `mybook` for later explanation)

```
val mybook = addrbook[
               name["Haruo Hosoya"],
               addr["Tokyo"],
               name["ABC"],
               addr["Def"],
               tel["123-456-789"],
               name["Benjamin Pierce"],
               addr["Philadelphia"]]
```

in the native syntax, or the following corresponding document in standard XML syntax:

```
<addrbook>
  <name>Haruo Hosoya</name>
  <addr>Tokyo</addr>
  <name>ABC</name>
  <addr>Def</addr>
  <tel>123-456-789</tel>
  <name>Benjamin Pierce</name>
  <addr>Philadelphia</addr>
</addrbook>
```

XDuce provides term constructors of the form `label[...]`, where ... is a sequence of other values. This corresponds to `<label>...</label>` in XML notation. We enclose strings in double-quotes, unlike XML.

Observe the sequence contained in `addrbook`. It is natural to impose a structure on the seven children so that they can be regarded as three "entries," each of which consists of fields tagged `name`, `addr` and optional `tel`. We can capture this structure by defining the following regular expression types.

```
type Addrbook = addrbook[(Name,Addr,Tel?)*]
type Name = name[String]
type Addr = addr[String]
type Tel = tel[String]
```

These XDuce definitions roughly correspond to the following DTD:

```
<!ELEMENT addrbook (name,addr,tel?)*>
<!ELEMENT name #PCDATA>
<!ELEMENT addr #PCDATA>
<!ELEMENT tel #PCDATA>
```

(Just as XDuce can read standard XML documents, we also provide a construct to import DTDs as regular expression types.) Type constructors `label[...]` have the same form as the term constructors that they classify. In addition, types may be formed using the regular expression operators `*` for repetition, `|` for alternation, and `?` for optional elements. (We will show examples of alternations later.) For instance, the type `(Name,Addr,Tel?)*` stands for zero or more repetitions of the sequence of a `Name`, an `Addr`, and an optional `Tel`.

The notion of *subtyping* will play a crucial role in the calculation that justifies assigning the type `Addrbook` to the value `mybook`.

2.1.2 Subtyping

Before showing the subtyping relation, we need to clearly state this: the elements of every type in XDuce are *sequences*. For example, the type `Tel*` contains the following sequences.

`()`	the empty sequence
`tel["123"]`	sequence with one `Tel`
`tel["123"],tel["234"]`	sequence with two `Tel`'s
...	

In the type language, comma is the type constructor for concatenation of sequences. For example, the type `(Name,Tel*, Addr)` contains

```
name["abc"],addr["ABC"]
name["abc"],tel["123"],addr["ABC"]
name["abc"],tel["123"],tel["234"],addr["ABC"]
...
```

i.e., sequences with one `Name` value, followed by zero or more `Tel` values, then followed by one `Addr` value. The comma operator on types is associative: the types `((Name,Tel*),Addr)` and `(Name,(Tel*,Addr))` have exactly the same set of elements.

The *subtype* relation between two types is simply inclusion of the sets denoted by types.

We now show the sequence of steps involved in verifying that `mybook` has type `Addrbook`. First, from the intuition that `?` means "optional," we would expect the following relations:

```
    Name,Addr     <:  Name,Addr,Tel?
    Name,Addr,Tel <:  Name,Addr,Tel?
```

Notice that each right hand side has more possibilities than the left hand side. Similarly, `*` means "zero or more" intuitively, so in particular it could be three:

```
    T,T,T  <:  T*
```

Combining these relations, we obtain

```
        (Name,Addr),(Name,Addr,Tel),(Name,Addr)
    <:  (Name,Addr,Tel?)*.
```

Since comma is associative, we can get rid of parentheses:

```
        Name,Addr,Name,Addr,Tel,Name,Addr
    =   (Name,Addr),(Name,Addr,Tel),(Name,Addr)
```

(Here, we mean by `T = U` that both `T <: U` and `U <: T`.) Finally, combining these two relations and enclosing both sides by `addrbook` constructor, we obtain

```
        addrbook[Name,Addr,Name,Addr,Tel,Name,Addr]
    <:  addrbook[(Name,Addr,Tel?)*]
```
$\overset{\text{def}}{=}$ `Addrbook.`

Since the `mybook` value trivially has the type on the left hand side, it has also the type on the right hand side.

2.1.3 Union Types

XDuce also provides a union (or alternation) type constructor `|`. For example, we write `(Name|Tel)` to mean "either `Name` or `Tel`"; the basic subtyping relations for union types are the following.

```
    Name  <:  Name | Tel
    Tel   <:  Name | Tel
```

Notice that each right hand side offers more possibilities, and so describes a larger set of sequences.

Union types substantially increase our flexibility in programming. In particular, union types yield two interesting relations: "forget ordering" subtyping and "distributivity." These are the distinguishing points in union types, as opposed to conventional tagged sum types (as found, say, in ML). To illustrate these, let us consider the following scenario of a "database evolution."

Suppose we begin with a trivial database consisting of just a list of names, with type `Name*`. At some point, this database is copied to two different sites and maintained and evolved separately. At one site, address information is added to each name and the type of the database becomes `(Name,Addr)*`, while at the other telephone numbers are added and it becomes `(Name,Tel)*`.

Now, suppose we want to re-integrate these databases—that is, combine the copies `src1`, whose type is `(Name,Addr)*`, and `src2` of type `(Name,Tel)*` by concatenating them: `src1, src2`. The type of this merged database is, of course, `(Name, Addr)*, (Name,Tel)*`.

Next, suppose we want to do something with our new database that involves extracting the common part (i.e., the name) from each record. Since we have two repetitions in the type, we might expect to need two loops in the program. (We do not show such a program explicitly, but it is easy to write.) However, we can do better by making the two loops into one, using the following "forget ordering" subtype relation:

```
        (Name,Addr)*, (Name,Tel)*
    <:  ((Name,Addr) | (Name,Tel))*
```

The intuition behind this relation is that the ordering information of the left hand side is forgotten on the right hand side. That is, on the left hand side, any (Name,Tel) pairs must occur after any (Name,Addr) pairs, while on the right hand side, these pairs can appear in any order.

Finally, since we have two alternatives joined by | in the new type, we might expect to need two branches in our inner loop, to extract the Name field from each of them. But we don't: we can use the following distributive subtyping law

```
  (Name,Addr) | (Name,Tel)  =  Name,(Addr | Tel)
```

to reorganize the type so that the Name field can be accessed directly.

2.2 Regular Expression Pattern Matching

Our term language is based on a powerful form of pattern matching. Our pattern matching is similar in spirit to ML's (or Haskell's, etc.), but somewhat more powerful, since it includes the use of regular expression types to dynamically match values of those types. Our patterns also require a different treatment of the usual checks for exhaustiveness and ambiguity of patterns.

The body of a XDuce program is a series of function definitions. As an example, the following function converts an address book into a telephone list.

```
fun mkTelList : (Name,Addr,Tel?)* → (Name,Tel)* =
    name[n:String], addr[a:String],
              tel[t:String], rest:(Name,Addr,Tel?)*
      → name[n], tel[t], mkTelList(rest)
  | name[n:String], addr[a:String],
                          rest:(Name,Addr,Tel?)*
      → mkTelList(rest)
  | ()
      → ()
```

This function takes a value of type (Name,Addr,Tel?)* and returns a value of type (Name,Tel)*. The body is a pattern match that breaks up the possibilities on the input values into three cases. The first case matches when the input value is a sequence beginning with name, addr, and tel labels, followed by some further sequence of type (Name,Addr,Tel?)*. In this case, we pick out the name and tel elements and prepend them to the result of calling mkTelList recursively on the remainder. The second case matches when we cannot find tel after addr, and simply calls mkTelList recursively. The third case matches the empty sequence, and returns the empty sequence.

As another example, consider the following function firstTriple, which takes out the first entry with a tel element.

```
fun firstTriple : (Name,Addr,Tel?)* →
                                (Name,Addr,Tel)? =
    ps:(Name,Addr)*, t:(Name,Addr,Tel),
                  rest:(Name,Addr,Tel?)*  → t
  | whole:(Name,Addr,Tel?)*  → ()
```

The function firstTriple has a pattern matching with two cases. In the first case, we skip all "pair" entries (i.e., (Name,Addr)) from the beginning and then pick out the first "triple" entry (i.e., (Name,Addr,Tel)) if such an entry exists. The second case matches otherwise and returns the empty.

The second example is more interesting in that the use of regular expression types is more critical there than in the first example. In the first case in the first example, the pattern matcher will walk over the first three elements of label name, addr, and tel, and then try to match the rest value against the pattern rest:(Name,Addr,Tel?)*. However, any value should already have this type. Therefore such a matching would not be meaningful. This is not true in the first case in the second example. When the pattern matcher looks at the first pattern ps:(Name,Addr)* in the first case, there is no hint about how many entries are "pairs." Therefore the matcher must walk through the input value to find where the chain of pairs ends. This matching for a *variable* length sequence is beyond ML pattern matching.

In these examples, pattern matchings are exhaustive. That is, all the values of type (Name,Addr,Tel?)* are covered by these patterns. In order to check exhaustiveness, we again use subtyping. For instance, in the first example, we check the following subtype relation

```
    (Name,Addr,Tel?)*
 <:  name[String], addr[String],  tel[String],
                                  (Name,Addr,Tel?)*
    | name[String], addr[String], (Name,Addr,Tel?)*
    | ()
```

where the left hand side is the parameter type on the annotation and the right hand side is the type constructed from the patterns (i.e., the union of the patterns with all the term variables n, a, etc. removed). (Although in these examples subtyping of the other way around also holds, we do not check this since allowing this sometimes makes programming easier. Such a situation typically occurs when a pattern contains variables whose type information is useless in the body.)

Our pattern matchings can have two kinds of ambiguity. The first ambiguity occurs when multiple patterns match the same input value. For example, the patterns in firstTriple function above are ambiguous since any value that matches the first pattern also matches the second pattern. In such a case, we simply take the first matching pattern ("first matching policy"). The second ambiguity occurs when a single pattern can have multiple ways for variable bindings. This is intrinsic in regular expression pattern matching. For example, suppose we replace the first case in firstTriple with the following:

```
es:(Name,Addr,Tel?)*, t:(Name,Addr,Tel),
                   rest:(Name,Addr,Tel?)*  → t
```

since we now skip both pair and triple entries at the beginning using the pattern es:(Name,Addr,Tel?)*, it is not clear which triple entry the variable t is bound to. We resolve this ambiguity by adpoting a "longest match" policy where patterns appearing earlier have higher priority. In this example, the first (Name,Addr,Tel?)* matches as a long sequence as possible and therefore t is bound to the last triple entry.

Another possible approach to resolving this ambiguity issue would be to simply disallow ambiguity. However, when we want to write a "default case" in a pattern matching, this restriction would force to write a somewhat cumbersome pattern that captures the "negation" of the other cases.

2.3 More Complex Example: Folder Manipulation

Up to now, the types that we have seen looked like regular expressions on strings. More interesting programs involve regular expressions on *trees*. Consider the following program.

```
type Folder = Record*
type Record = name[String], folder[Folder]
            | name[String], url[String], exists[Bool]

fun tidyFolder : Folder→Folder =
    record:Record, folder:Folder
       → tidyRecord(record), tidyFolder(folder)
  | () → ()

fun tidyRecord : Record→Record? =
    name[nm:String], folder[fl:Folder]
       → name[nm], folder[tidyFolder(fl)]
  | name[nm:String], url[s:String], exists[false[]]
       → ()
  | name[nm:String], url[s:String], exists[true[]]
       → name[nm], url[s], exists[true[]]
```

The mutually recursive types Folder and Record define a simple template for storing structured lists of bookmarks, as might be found in a web browser: a folder is a list of records, while a record is either a named folder or a named URL plus a boolean indicating whether the link is good or broken. The functions tidyFolder and tidyRecord traverse a bookmark list recursively, preserving leaves with good links and dropping ones with bad links.

3. RELATED WORK

Mainstream XML-specific languages can be divided into query languages such as XML-QL [5] and Lorel [1] and programming languages such as XSLT [15]. In general, when one is interested in rather simple information extraction from XML databases, programs in programming languages are less succinct than the same programs in a suitable query language. On the other hand, programming languages tend to be more suitable for writing complicated transformations like conversion to a display format. XDuce is categorized as a programming language.

Static typing of programs for XML processing has been approached from several different angles. One popular approach is to embed a type system for XML in an existing typed language. The advantage is that we can enjoy not only the static safety and typechecking, but also all the other features provided by the host language. The cost is that XML values and their corresponding DTDs must be some how "injected" into the value and type space of the host language; this usually involves adding more layers of tagging than were present in the original XML documents, which inhibits subtyping. The lack of subtyping (or availability of only restricted forms of subtyping) is not a serious problem for simple traversal of tree structures; it becomes a stumbling block, though, in tasks like the "database evolution" that we discussed in Section 2, where forget-ordering subtyping and distributivity were critically needed.

A recent example of the embedding approach is Wallace and Runciman proposal to use Haskell as a host language [12] for XML processing. The only thing they add to Haskell is a mapping from DTDs into Haskell datatypes. This allows their programs to make use of other mechanisms standard in functional programming languages, such as higher-order functions, parametric polymorphism, and pattern matching. However, they do not have any notion of subtyping. Moreover, pattern matching in XDuce is more powerful than Haskell's in some cases. For instance, as shown in Section 2.2, we can concisely write patterns that skip a variable length sequence by using regular expression types. A difference in the other direction is that XDuce does not currently support higher-order functions or parametric polymorphism. (We are working on both of these extensions.)

Another piece of work along similar lines is the functional language XMλ for XML processing, proposed by Meijer and Shields [8]. Their type system is not described in detail in this paper, but seems to be close to Haskell's, except that they incorporate *Glushkov automata* in type checking, resulting in a more flexible type system.

A closer relative to XDuce is the query language YAT [11], which allows optional use of types similar to DTDs. The notion of subtyping between these types is somewhat weaker than ours (lacking, in particular, the distributivity laws used in the "database evolution" example in Section 2.1).

Types based on tree automata have also been proposed in a more abstract study of typechecking for a general form of "tree transformers" for XML by Milo, Suciu, and Vianu [9]. The types there are conceptually identical to those of XDuce.

The type system of XDuce was originally motivated by the observation by Buneman and Pierce [2] that untagged union types corresponded naturally to forms of variation found in semistructured databases.

4. CONCLUSIONS AND FUTURE WORK

We have presented several examples of XDuce programming and shown how we can write flexible and robust programs for processing XML by combining regular expression types and regular expression pattern matching.

We consider XDuce suitable for applications involving rather complicated tree transformation. Moreover, for such applications, our static typing mechanism would help in reducing development periods.

In this view, we have built a prototype implementation of XDuce and used it to develop some small but non-trivial applications:

Bookmarks can be viewed as a simple database query. It takes as input an Netscape bookmarks file of type `Bookmarks`, which is a subset of the (much larger) type HTML. It extracts a certain folder named "Public", formats it as a free-standing document, adds a table of contents at the front, and inserts links between the contents and the body. The result has type the full `HTML` type. (Total: 224 lines)

Html2Latex takes an HTML file (of type `HTML`) and converts it into LaTeX format (of type `String`). (264 lines)

Diff implements the "tree diff" algorithm proposed by Chawathe [3]. It takes a pair of XML files of generic `Xml` type and returns a tree with annotations indicating whether each subtree has been retained, inserted, deleted, or changed between the two inputs. (300 lines)

The first two applications are written in the way that traverses the input tree by several simple recursive functions. The third one is more complex. The first phase is a dynamic programming algorithm, where regular expression types are used for representing the internal data structures; the second phase combines two input trees and inserts annotations at each node, where types ensure that the annotations and the actual trees are never confused. In the course of writing these applications, our type checker gave us tremendous help in finding silly mistakes.

The implementation of XDuce raises many algorithmic issues. The primary source of complication is that types and patterns in XDuce are essentially tree automata and therefore we need to use operations on tree automata [4], which are in general expensive. For instance, decision for subtyping is inclusion of tree automata, which is known to be EXPTIME-complete [10]. We have addressed this problem and obtained an algorithm that runs efficiently in practice [7]. In particular, the HTML type[1] used in the above applications is generally considered to be one of the largest XML DTDs, yet type checking of our programs involving it takes a fraction of a second on stock hardware. As other implementation issues, we are working on a type inference algorithm to eliminate spurious type annotations in patterns, and a pattern compilation scheme to improve run-time efficiency.

XDuce's language design is far from finished. We plan to add standard features from functional programming, such as higher-order functions and parametric polymorphism. We also consider a support for object-oriented features found in XML-Schema specifications [14]. The combination of these features with regular expression types raises some subtle issues, which we are now seeking to solve.

[1]More precisely, we use XHTML, which is an XML implementation of HTML.

Our prototype implementation is written in O'Caml (6500 lines excluding libraries such as the XML parser). Interested readers are invited to visit our home page:

`http://www.cis.upenn.edu/~hahosoya/xduce.html`

5. ACKNOWLEDGMENTS

Our main collaborators in the XDuce project are Peter Buneman, Jérôme Vouillon, and Phil Wadler. We have also learned a great deal from discussions with Nils Klarlund and Volker Renneberg, with the DB Group and the PL Club at Penn, and with members of Professor Yonezawa's group at Tokyo.

This work was supported by the University of Pennsylvania and by an NSF Career grant, CCR-9701826. Haruo Hosoya is supported by Japan Society for the Promotion of Science.

6. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] P. Buneman and B. Pierce. Union types for semistructured data. In *Proceedings of the International Database Programming Languages Workshop*, Sept. 1999. Also available as University of Pennsylvania Dept. of CIS technical report MS-CIS-99-09.

[3] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., Sept. 1999.

[4] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`.

[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

[6] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. `http://www.cis.upenn.edu/~hahosoya/xduce.html`, 2000. Full version.

[7] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. `http://www.cis.upenn.edu/~hahosoya/xduce.html`, 2000.

[8] E. Meijer and M. Shields. Xmlambda: A functional programming language for constructing and manipulating xml documents. In *Submitted to USENIX 2000 Technical Conference*, page 13 pages.

[9] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *PODS 00*, May 2000. To appear.

[10] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[11] S. C. J. Simeon. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB)*, 1998.

[12] M. Wallace and C. Ranciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 148–159, N.Y., Sept. 27–29 1999. ACM Press.

[13] Extensible markup language (XML$^{\text{TM}}$). `http://www.w3.org/XML/`.

[14] XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

[15] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`.