XDuce: A Typed XML Processing Language (Preliminary Report)

Haruo Hosoya[1] and Benjamin Pierce[2]

[1] Department of CIS, University of Pennsylvania
hahosoya@cis.upenn.edu
[2] 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 [XML] (or more sophisticated mechanisms such as XML-Schema [XS00]) 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. Appendices give a formal definition of the core language.

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

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`.

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(see Appendix A for the formal definition).

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:

$$\begin{array}{rl} \texttt{Name,Addr} & <: \texttt{Name,Addr,Tel?} \\ \texttt{Name,Addr,Tel} & <: \texttt{Name,Addr,Tel?} \end{array}$$

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:

$$\texttt{T,T,T} <: \texttt{T*}$$

Combining these relations, we obtain

$$\texttt{(Name,Addr),(Name,Addr,Tel),(Name,Addr)} <: \texttt{(Name,Addr,Tel?)*}.$$

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?)*]
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.

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

$$\texttt{(Name,Addr) | (Name,Tel)} = \texttt{Name,(Addr | Tel)}$$

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 adopting 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. See Section Appendix C for a more formal treatment.

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 [DFF$^+$] and Lorel [AQM$^+$97] and programming languages such as XSLT [XSL]. 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 [WR99] 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 [MS99]. 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 [CS98], 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).

Milo, Suciu, and Vianu have studied a typechecking problem for their general framework called *k-pebble tree transducers*, which can capture a wide range of query languages for XML [MSV00]. The types there are based on tree automata and conceptually identical to those of XDuce. Papakonstantinou and Vianu present a typechecking algorithm for their query language *loto-ql* by using extensions to DTDs [PV00]. One of their extensions is equivalent to tree automata. The type checking algorithms presented in both papers are *semantically complete*, while XDuce's is not (since such typechecking would be undecidable in general for Turing-complete languages).

The type system of XDuce was originally motivated by the observation by Buneman and Pierce [BP99] 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 [Cha99]. 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 [CDG^+], which are in general expensive. For instance, decision for subtyping is inclusion of tree automata, which is known to be EXPTIME-complete [Sei90]. We have addressed this problem and obtained an algorithm that runs efficiently in practice [HVP00]. 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 [HP00].

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

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 [XS00]. The combination of these features with regular expression types raises some subtle issues, which we are now seeking to solve.

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.

References

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BP99] Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Proceedings of the International Database Programming Languages Workshop*, September 1999. Also available as University of Pennsylvania Dept. of CIS technical report MS-CIS-99-09.

[CDG+] Hubert Common, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`.

[Cha99] Sudarshan 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., September 1999.

[CS98] Sophie Cluet and Jerome Simeon. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB)*, 1998.

[DFF+] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

[HP00] Haruo Hosoya and Benjamin Pierce. Tree automata and pattern matching. Available through `http://www.cis.upenn.edu/~hahosoya/papers/tapat-full.ps`, July 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[MS99] Erik Meijer and Mark Shields. XMλ: A functional programming language for constructing and manipulating XML documents. Submitted to USENIX 2000 Technical Conference, 1999.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, May 2000.

[Sei90] Hermut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[WR99] Malcolm Wallace and Colin 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., September 27–29 1999. ACM Press.

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

[XS00] XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

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

A Types

This section gives the formal definitions for types and subtyping.

A.1 Syntax

We assume a fixed set of base types, ranged over by $\mathtt{B}$, and a countably infinite set of labels, ranged over by $\mathtt{l}$. Each base type $\mathtt{B}$ denotes a set $V_{\mathtt{B}}$ of base values, ranged over by $\mathtt{b}_{\mathtt{B}}$. We assume that all $V_{\mathtt{B}}$ are disjoint.

Values are defined as follows.

$$\mathtt{v} ::= \mathtt{M}_1, \ldots, \mathtt{M}_n \quad \text{sequence } (n \geq 0)$$

$$\begin{array}{lll} \mathtt{M} ::= & \mathtt{b}_{\mathtt{B}} & \text{base element} \\ & \mathtt{l[v]} & \text{label element} \end{array}$$

We write $\mathtt{()}$ for the empty value (i.e., the empty sequence). We write $\mathtt{v,w}$ for the concatenation of values (i.e., sequences) $\mathtt{v}$ and $\mathtt{w}$.

A XDuce program consists of a set of type definitions, a set of function definitions, and an expression to be evaluated. Type definitions have the form

```
type X = T
```

where $\mathtt{X}$ ranges over variables and $\mathtt{T}$ ranges over type expressions. Type expressions are defined as follows.

$$\begin{array}{lll} \mathtt{T} ::= & \mathtt{X} & \text{variable} \\ & \mathtt{B} & \text{base type} \\ & \mathtt{()} & \text{empty sequence} \\ & \mathtt{l[T]} & \text{label} \\ & \mathtt{T,T} & \text{concatenation} \\ & \mathtt{T|T} & \text{union} \end{array}$$

We sometimes write $\mathtt{type}\ \overline{\mathtt{X}}\mathtt{=}\overline{\mathtt{T}}$ to abbreviate a set of type definitions $\mathtt{type}\ \mathtt{X}_1\mathtt{=T}_1, \ldots, \mathtt{type}\ \mathtt{X}_n\mathtt{=T}_n$. Since the set of type definitions is fixed for a given XDuce program, we lighten the notation below by assuming a fixed set of definitions $E = \mathtt{type}\ \overline{\mathtt{X}}\mathtt{=}\overline{\mathtt{T}}$.

A.2 Semantics of Types

The denotation of each type is a set of values, defined as follows.

First, we define a function $[\![\cdot]\!]$, which takes a type and an environment mapping type variables to sets of values, and returns a set of values.

$$\begin{aligned} [\![\mathtt{B}]\!]\rho &= V_{\mathtt{B}} \\ [\![\mathtt{()}]\!]\rho &= \{\, \mathtt{()} \,\} \\ [\![\mathtt{l[T]}]\!]\rho &= \{\, \mathtt{l[f]} \mid \mathtt{f} \in [\![\mathtt{T}]\!]\rho \,\} \\ [\![\mathtt{T}_1\mathtt{,T}_2]\!]\rho &= \{\, \mathtt{f,g} \mid \mathtt{f} \in [\![\mathtt{T}_1]\!]\rho \ \wedge\ \mathtt{g} \in [\![\mathtt{T}_2]\!]\rho \,\} \\ [\![\mathtt{T}_1\mathtt{|T}_2]\!]\rho &= [\![\mathtt{T}_1]\!]\rho \cup [\![\mathtt{T}_2]\!]\rho \\ [\![\mathtt{X}]\!]\rho &= \rho(\mathtt{X}) \end{aligned}$$

Let μ be the smallest mapping satisfying $\mu(\mathtt{X}_i) = [\![\mathtt{T}_i]\!]\mu$ for all the definitions `type` $\mathtt{X}_i$=$\mathtt{T}_i$ in E. Finally, write $[\![\mathtt{T}]\!]$ for $[\![\mathtt{T}]\!]\mu$.

A.3 Derived Forms

The regular expression type constructors are derivable as combinations of the above constructs. We represent the Kleene closure `T*` by a variable `X` that is recursively defined as follows (cf. lists as a datatype in ML).

```
type X = T,X | ()
```

The other regular expression constructors are defined as follows.

$$\mathtt{T+} \equiv \mathtt{T\ ,\ T*}$$
$$\mathtt{T?} \equiv \mathtt{T\ |\ ()}$$

A.4 Regularity

As we have defined them so far, types correspond to arbitrary context-free grammars—for example, we can write definitions like:

```
type X = Int,X,String | ()
```

Since the decision problem for inclusion between context free languages is undecidable [HU79], we need to impose an additional restriction to reduce the power of the system so that types correspond to regular tree languages. Deciding whether an arbitrary context-free grammar is regular is also undecidable [HU79], so we adopt a simple syntactic condition, called *well-formedness*, that ensures regularity. Intuitively, well-formedness allows recursive uses of variables to occur only in tail positions. For example, we allow the following type definitions:

```
type X = Int,Y
type Y = String,X | ()
```

More precisely, we define well-formedness in terms of a "right-linearity" judgment of the form $\sigma \vdash \mathtt{T} : rl(\mathtt{X})$, where σ is a set of variables. It should be read "`T` is right-linear in `X`, assuming that all bodies of variables in σ are right-linear in `X`." This judgment uses an auxiliary "disconnectedness" judgment of the form $\sigma \vdash \mathtt{T} : dc(\mathtt{X})$, read "`T` is disconnected from `X` (i.e., `X` does not occur in the top level of `T`), assuming that all bodies of variables in σ are disconnected from `X`."

These two judgments are defined by the following rules (where $\mathtt{X} \neq \mathtt{Y}$):

$$
\begin{array}{ll}
\sigma \vdash \mathtt{T} : rl(\mathtt{X}) & \text{for } \mathtt{T} = \mathtt{()} \text{ or } \mathtt{l[T]} \text{ or } \mathtt{X} \\
\sigma \vdash \mathtt{Y} : rl(\mathtt{X}) & \text{if } \mathtt{Y} \in \sigma \\
\sigma \vdash \mathtt{Y} : rl(\mathtt{X}) & \text{if } \mathtt{Y} \notin \sigma \text{ and } \sigma \cup \{\mathtt{Y}\} \vdash E(\mathtt{Y}) : rl(\mathtt{X}) \\
\sigma \vdash \mathtt{T|U} : rl(\mathtt{X}) & \text{if } \sigma \vdash \mathtt{T} : rl(\mathtt{X}) \text{ and } \sigma \vdash \mathtt{U} : rl(\mathtt{X}) \\
\sigma \vdash \mathtt{T,U} : rl(\mathtt{X}) & \text{if } \emptyset \vdash \mathtt{T} : dc(\mathtt{X}) \text{ and } \sigma \vdash \mathtt{U} : rl(\mathtt{X}) \\
\sigma \vdash \mathtt{T} : dc(\mathtt{X}) & \text{for } \mathtt{T} = \mathtt{()} \text{ or } \mathtt{l[T]} \\
\sigma \vdash \mathtt{Y} : dc(\mathtt{X}) & \text{if } \mathtt{Y} \in \sigma \\
\sigma \vdash \mathtt{Y} : dc(\mathtt{X}) & \text{if } \mathtt{Y} \notin \sigma \text{ and } \sigma \cup \{\mathtt{Y}\} \vdash E(\mathtt{Y}) : dc(\mathtt{X}) \\
\sigma \vdash \mathtt{T|U} : dc(\mathtt{X}) & \text{if } \sigma \vdash \mathtt{T} : dc(\mathtt{X}) \text{ and } \sigma \vdash \mathtt{U} : dc(\mathtt{X}) \\
\sigma \vdash \mathtt{T,U} : dc(\mathtt{X}) & \text{if } \sigma \vdash \mathtt{T} : dc(\mathtt{X}) \text{ and } \sigma \vdash \mathtt{U} : dc(\mathtt{X})
\end{array}
$$

The empty sequence, a label, and the variable `X` are right-linear in `X`. For variables `Y` other than `X`, we recursively check the right-linearity of the body of `Y`. To ensure termination, we keep track in σ of variables that have already been checked. For `(T|U)`, both `T` and `U` should be right-linear in `X`. For `(T,U)`, we check if `U` is right-linear in `X`, while `T` is disconnected from `X`. The disconnectedness judgment is defined similarly, except for the first rule, in which `X` is *not* disconnected from `X`. Now, the set of type definitions E is said to be *well-formed* if

$$\emptyset \vdash E(\mathtt{X}) : rl(\mathtt{X}) \text{ for all } \mathtt{X} \in dom(E)$$

A.5 Subtyping

We define the subtyping relation in the simplest possible way:

$$\mathtt{S} <: \mathtt{T} \text{ iff } [\![\mathtt{S}]\!] \subseteq [\![\mathtt{T}]\!].$$

A.1 Theorem: There is an algorithm to decide the subtyping relation.

The concrete algorithm and its soundness, completeness, and termination theorems are given in our another paper [HVP00].

B Terms

B.1 Syntax

At the top level, we have a set of declarations of mutually recursive functions, each of the following form.

```
fun f : S→T =
    p1 → e1
  | p2 → e2
  | ...
  | pn → en
```

Each function has type annotations for both the parameter and the result. The body consists of one or more pattern matching clauses. Each clause has a pattern and a term for the body. (We write $\texttt{fun f:S}\rightarrow\texttt{T = }\overline{\texttt{p}}\rightarrow\overline{\texttt{e}}$ to abbreviate the above form.) In addition to the fixed set E of type definitions, we fix a set F of function definitions for the remainder of the document.

The syntax of terms and patterns is:

$$
\begin{array}{rll}
\texttt{e} ::= & \texttt{x} & \text{variable} \\
 & \texttt{b}_\texttt{B} & \text{base value} \\
 & \texttt{()} & \text{empty sequence} \\
 & \texttt{l[e]} & \text{label} \\
 & \texttt{e,e} & \text{concatenation} \\
 & \texttt{f(e)} & \text{application}
\end{array}
$$

$$
\begin{array}{rll}
\texttt{p} ::= & \texttt{x:T} & \text{variable} \\
 & \texttt{()} & \text{empty sequence} \\
 & \texttt{l[p]} & \text{label} \\
 & \texttt{p,p} & \text{concatenation}
\end{array}
$$

B.2 Typing Rules

A context Γ is a mapping from variables to types, written $\texttt{x}_1\texttt{:T}_1\texttt{,}\ldots\texttt{,x}_n\texttt{:T}_n$. We have three judgments:

$$
\begin{array}{ll}
\Gamma \vdash \texttt{e} \in \texttt{T} & \texttt{e} \text{ has type } \texttt{T} \\
\vdash \texttt{p} \in \texttt{T} \Rightarrow \Gamma' & \texttt{p} \text{ accepts type } \texttt{T} \text{ and yields context } \Gamma' \\
\vdash \texttt{fun f:S}\rightarrow\texttt{T = }\overline{\texttt{p}}\rightarrow\overline{\texttt{e}} & \texttt{f} \text{ is well-typed}
\end{array}
$$

The typing relation for terms is:

$$\frac{\Gamma(\texttt{x}) = \texttt{T}}{\Gamma \vdash \texttt{x} \in \texttt{T}} \qquad \text{(TE-VAR)}$$

$$\Gamma \vdash \texttt{b}_\texttt{B} \in \texttt{B} \qquad \text{(TE-BASE)}$$

$$\Gamma \vdash \texttt{()} \in \texttt{()} \qquad \text{(TE-EMP)}$$

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{T}}{\Gamma \vdash \texttt{l[e]} \in \texttt{l[T]}} \qquad \text{(TE-LAB)}$$

$$\frac{\Gamma \vdash \texttt{e}_1 \in \texttt{T}_1 \qquad \Gamma \vdash \texttt{e}_2 \in \texttt{T}_2}{\Gamma \vdash \texttt{e}_1\texttt{,e}_2 \in \texttt{T}_1\texttt{,T}_2} \qquad \text{(TE-CAT)}$$

$$\frac{\begin{array}{c}\texttt{fun f:S}\rightarrow\texttt{T = }\ldots \in F \\ \Gamma \vdash \texttt{e} \in \texttt{S}'\end{array}}{\Gamma \vdash \texttt{f(e)} \in \texttt{T}} \qquad \text{(TE-APP)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{e} \in \mathtt{T}} \tag{TE-Sub}$$

For patterns:

$$\vdash \mathtt{x:T} \in \mathtt{T} \Rightarrow \mathtt{x:T} \tag{TP-Var}$$

$$\vdash \mathtt{()} \in \mathtt{()} \Rightarrow \cdot \tag{TP-Emp}$$

$$\frac{\vdash \mathtt{p} \in \mathtt{T} \Rightarrow \Gamma}{\vdash \mathtt{l[p]} \in \mathtt{l[T]} \Rightarrow \Gamma} \tag{TP-Lab}$$

$$\frac{\vdash \mathtt{p}_1 \in \mathtt{T}_1 \Rightarrow \Gamma_1 \qquad \vdash \mathtt{p}_2 \in \mathtt{T}_2 \Rightarrow \Gamma_2}{\vdash \mathtt{p}_1\mathtt{,p}_2 \in \mathtt{T}_1\mathtt{,T}_2 \Rightarrow \Gamma_1, \Gamma_2} \tag{TP-Cat}$$

For functions:

$$\frac{\begin{array}{c}\vdash \mathtt{p}_i \in \mathtt{S}_i \Rightarrow \Gamma_i \qquad \Gamma_i \vdash \mathtt{e}_i \in \mathtt{T}_i \\ \mathtt{S} <: \mathtt{S}_1 \,|\, \mathtt{..} \,|\, \mathtt{S}_n\end{array}}{\vdash \mathtt{fun\ f:S} \rightarrow \mathtt{T\ =\ }\overline{\mathtt{p}} \rightarrow \overline{\mathtt{e}}} \tag{TF}$$

The rule TF needs a little explanation: each pattern $\mathtt{p}_i$ accepts type $\mathtt{S}_i$ and yields the context Γ_i, under which the body $\mathtt{e}_i$ can be given the result type T (as annotated). Also, the parameter type S is required to be a subtype of the union of all $\mathtt{S}_i$'s. This subtype checks exhaustiveness of patterns.

C Operational Semantics

The operational semantics consists of evaluation relations for terms and pattern matching. The former is fairly standard, the latter a little unusual due to "ambiguity" in patterns.

There are two kinds of ambiguity. One appears when there are multiple clauses whose patterns match the input value. For example, in the function

```
fun is_single : Int*→Bool =
    x:Int   →  true
  | x:Int*  →  false
```

we have two possibilities when the input is a single integer.

The other source of ambiguity is when there are multiple ways in which a single clause can match a given value. For example, in the function

```
fun f : Int*→Int* =
  x:Int*, y:Int*  →  x
```

it is unclear how many integers `x` takes from the beginning of the input. We address the first source of ambiguity by adopting a "first match" policy, as in ML. For the second one, we adopt a "longest match" policy as in Emacs.

In fact, the longest-match policy turns out to be just a special case of the first-match policy. Recall that `Int*` is a derived form meaning a variable `X` defined as follows:

```
type X = Int,X | ()
```

The ordering of the branches in the definition is significant: `Int,X` comes before `()`. Now, in the above example, we first process the pattern `x:Int*`, traversing the input value and the definition of `X` in parallel and taking the first clause (`Int,X`) as often as possible. When the input value is exhausted and the second clause (`()`) is taken, we move on to the pattern `y:Int*`, where we can now only take the second clause because the remaining of the input value is the empty sequence.

To describe the first-match policy, we use a notion of *choice sequence*. During pattern matching, we remember the index of the branch we take at each choice point. A choice sequence is a sequence of such indices, listed according to the order of traversal—from left to right and from outer to inner. Finally, we take the *smallest* choice sequence in the dictionary order (written $\prec$).

The evaluation relations are defined with respect to an environments V—a mapping from variables to values. There are three judgments.

$V \vdash \mathtt{e} \Downarrow \mathtt{v}$	`e` evaluates to `v`
$\vdash \mathtt{p} \triangleright \mathtt{v} \Rightarrow V \,/\, \alpha$	`p` matches `v` and yields V and choice sequence α
$\vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha$	`T` matches `v` and yields choice sequence α

Evaluation of terms is defined as follows:

$$V \vdash \mathtt{x} \Downarrow V(\mathtt{x}) \tag{EE-Var}$$

$$V \vdash \mathtt{b_B} \Downarrow \mathtt{b_B} \tag{EE-Base}$$

$$V \vdash \mathtt{()} \Downarrow \mathtt{()} \tag{EE-Emp}$$

$$\frac{V \vdash \mathtt{e} \Downarrow \mathtt{v}}{V \vdash \mathtt{l[e]} \Downarrow \mathtt{l[v]}} \tag{EE-Lab}$$

$$\frac{V \vdash \mathtt{e}_1 \Downarrow \mathtt{v} \qquad V \vdash \mathtt{e}_2 \Downarrow \mathtt{w}}{V \vdash \mathtt{e}_1\mathtt{,e}_2 \Downarrow \mathtt{v,w}} \tag{EE-Cat}$$

$$\frac{\begin{array}{c} V \vdash \mathtt{e} \Downarrow \mathtt{v} \\ \mathtt{fun\ f:S}\to\mathtt{T} \;\mathtt{=}\; \overline{\mathtt{p}}\to\overline{\mathtt{e}} \in F \\ \vdash \mathtt{p}_i \triangleright \mathtt{v} \Rightarrow W \,/\, \alpha \\ \forall j,\beta.(\vdash \mathtt{p}_j \triangleright \mathtt{v} \Rightarrow U \,/\, \beta \implies (i,\alpha) \prec (j,\beta)) \\ W \vdash \mathtt{e}_i \Downarrow \mathtt{w} \end{array}}{V \vdash \mathtt{f(e)} \Downarrow \mathtt{w}} \tag{EE-App}$$

Pattern matching:

$$\frac{\vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha}{\vdash \mathtt{x{:}T} \triangleright \mathtt{v} \Rightarrow \mathtt{x} \mapsto \mathtt{v} \;/\; \alpha} \qquad \text{(EP-Var)}$$

$$\vdash \mathtt{()} \triangleright \mathtt{()} \Rightarrow \cdot \;/\; \cdot \qquad \text{(EP-Emp)}$$

$$\frac{\vdash \mathtt{p} \triangleright \mathtt{v} \Rightarrow V \;/\; \alpha}{\vdash \mathtt{l[p]} \triangleright \mathtt{l[v]} \Rightarrow V \;/\; \alpha} \qquad \text{(EP-Lab)}$$

$$\frac{\begin{array}{c}\vdash \mathtt{p} \triangleright \mathtt{v} \Rightarrow V \;/\; \alpha \\ \vdash \mathtt{q} \triangleright \mathtt{w} \Rightarrow W \;/\; \beta\end{array}}{\vdash \mathtt{p,q} \triangleright \mathtt{v,w} \Rightarrow V, W \;/\; \alpha, \beta} \qquad \text{(EP-Cat)}$$

Type matching:

$$\frac{\begin{array}{c}\mathtt{type\ X=T} \in E \\ \vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha\end{array}}{\vdash \mathtt{X} \triangleright \mathtt{v} \Rightarrow \alpha} \qquad \text{(ET-Var)}$$

$$\vdash \mathtt{B} \triangleright \mathtt{b_B} \Rightarrow \cdot \qquad \text{(ET-Base)}$$

$$\vdash \mathtt{()} \triangleright \mathtt{()} \Rightarrow \cdot \qquad \text{(ET-Emp)}$$

$$\frac{\vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha}{\vdash \mathtt{l[T]} \triangleright \mathtt{l[v]} \Rightarrow \alpha} \qquad \text{(ET-Lab)}$$

$$\frac{\begin{array}{c}\vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha \\ \vdash \mathtt{U} \triangleright \mathtt{w} \Rightarrow \beta\end{array}}{\vdash \mathtt{T,U} \triangleright \mathtt{v,w} \Rightarrow \alpha, \beta} \qquad \text{(ET-Cat)}$$

$$\frac{\vdash \mathtt{T} \triangleright \mathtt{v} \Rightarrow \alpha}{\vdash \mathtt{T|U} \triangleright \mathtt{v} \Rightarrow 1, \alpha} \qquad \text{(ET-Or1)}$$

$$\frac{\vdash \mathtt{U} \triangleright \mathtt{v} \Rightarrow \alpha}{\vdash \mathtt{T|U} \triangleright \mathtt{v} \Rightarrow 2, \alpha} \qquad \text{(ET-Or2)}$$

The third and fourth premises in EE-App say that we choose the match that yields the smallest choice sequence in the dictionary order. Also, notice that in EP-Cat and ET-Cat we concatenate the choice sequences left to right, and that in ET-Or1 and ET-Or2 we adjoin the present choice number to the front. These reflect our policy that the priority of choice is from left to right and from outer to inner.

We can prove the following type soundness: a well-typed term evaluates to a value inhabiting the expected type.

C.1 Theorem [Type Soundness]: Suppose $\vdash$ `fun f:S→T = p̄→ē` for all functions in F. Then, $\emptyset \vdash \mathtt{e} \Downarrow \mathtt{v}$ and $\emptyset \vdash \mathtt{e} \in \mathtt{T}$ imply $\mathtt{v} \in [\![\mathtt{T}]\!]$.

Proof: We prove the result by showing the following stronger statements:

- If $\vdash \mathtt{T} \triangleright \mathtt{v}$, then $\mathtt{v} \in \mathtt{T}$.
- If $\vdash \mathtt{p} \triangleright \mathtt{v} \Rightarrow V \,/\, \alpha$ and $\vdash \mathtt{p} \in \mathtt{T} \Rightarrow \Gamma$, then $\Gamma \vdash V$.
- If $V \vdash \mathtt{e} \Downarrow \mathtt{v}$ and $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ where $\Gamma \vdash V$, then $\mathtt{v} \in [\![\mathtt{T}]\!]$.

(Here, $\Gamma \vdash V$ means that $dom(\Gamma) = dom(V)$ and $V(\mathtt{x}) \in [\![\Gamma(\mathtt{x})]\!]$ for each $\mathtt{x} \in dom(\Gamma)$.) Each statement can be proved by induction on the derivation of the type matching, pattern matching, or term evaluation relation. For the last statement, we need the following inversion properties.

- $\Gamma \vdash \mathtt{x} \in \mathtt{T}$ implies $\Gamma(\mathtt{x})$ <: $\mathtt{T}$.
- $\Gamma \vdash \mathtt{b_B} \in \mathtt{T}$ implies $\mathtt{B}$ <: $\mathtt{T}$.
- $\Gamma \vdash \mathtt{()} \in \mathtt{T}$ implies `()` <: $\mathtt{T}$.
- $\Gamma \vdash \mathtt{l[e]} \in \mathtt{T}$ implies $\Gamma \vdash \mathtt{e} \in \mathtt{S}$ with `l[S]` <: $\mathtt{T}$ for some $\mathtt{S}$.
- $\Gamma \vdash \mathtt{e_1,e_2} \in \mathtt{T}$ implies $\Gamma \vdash \mathtt{e_1} \in \mathtt{S_1}$ and $\Gamma \vdash \mathtt{e_2} \in \mathtt{S_2}$ with $\mathtt{S_1,S_2}$ <: $\mathtt{T}$ for some $\mathtt{S_1}$ and $\mathtt{S_2}$.
- $\Gamma \vdash \mathtt{f(e)} \in \mathtt{T}$ implies $\Gamma \vdash \mathtt{e} \in \mathtt{S}$ where `fun f:S→T' = ...` $\in F$ and $\mathtt{T'}$ <: $\mathtt{T}$.

∎