XDuce: A Statically Typed XML Processing Language

HARUO HOSOYA
Kyoto University
hahosoya@kurims.kyoto-u.ac.jp
and
BENJAMIN C. PIERCE
University of Pennsylvania
bcpierce@cis.upenn.edu

XDuce is a statically typed programming language for XML processing. Its basic data values are XML documents, and its types (so-called *regular expression types*) directly correspond to document schemas. XDuce also provides a flexible form of *regular expression pattern matching*, integrating conditional branching, tag checking, and subtree extraction, as well as dynamic typechecking. We survey the principles of XDuce's design, develop examples illustrating its key features, describe its foundations in the theory of regular tree automata, and present a complete formal definition of its core, along with a proof of type safety.

Categories and Subject Descriptors: []: —

General Terms:

Additional Key Words and Phrases:

1. INTRODUCTION

XML is a simple, generic format for structured data that has been standardized by the World-Wide Web Consortium [Bray et al. 2000]. Data (or *documents*) in XML are ordered, labeled tree structures. The core XML standard imposes no restrictions on the labels that appear in a given context; instead, each document may be accompanied by a document type (or *schema*) describing its structure.[1]

[1]Several schema languages have been proposed to date. The original specification of XML defines a schema language called DTD (Document Type Definition) [Bray et al. 2000]. Other schema languages include XML-Schema [Fallside 2001], RELAX NG [Clark and Murata 2001], and DSD [Klarlund et al. 2000]. In this paper we use the term "schema" generically to refer to a structural

Authors' address: H. Hosoya, Research Institute for Mathematical Sciences, Kyoto University Oiwake-cho, Kitashirakawa, Sakyo-ku, Kyoto 606–8502, Japan. B. C. Pierce, Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd St., Philadelphia, PA 19104, USA.

The most common use of schemas by programs that manipulate XML is for *dynamic* typechecking. Applications can use *schema validators* both to ensure that their input documents actually conform to their expectations and to double-check their own outputs. A more ambitious approach that has recently drawn the attention of many researchers is *static* typechecking—using document schemas as the basis of compile-time analyses capable of ensuring that invalid documents can *never* arise as outputs or intermediate states of XML processing code [Cluet and Siméon 1998; Wallace and Runciman 1999; Sun Microsystems 2001; Asami 2000; Meijer and Shields 1999; Fernández et al. 2001; Murata 1997; Milo et al. 2000; Papakonstantinou and Vianu 2000; Hosoya and Pierce 2000, etc.].

In this paper we describe a statically typed XML processing language called XDuce (officially pronounced "transduce"). XDuce is a functional language whose primitive data structures represent XML documents and whose types—called *regular expression types*—correspond to document schemas. The motivating principle behind its design is that a simple, clean, and powerful type system for XML processing can be based directly on the theory of regular tree automata.

Tree automata are finite-state machines that accept trees, just as ordinary regular automata accept strings. The mathematical underpinnings of tree automata are well understood [Comon et al. 1999]—in particular, the problem of deciding whether the language accepted by one automaton is included in that accepted by another (which corresponds to the familiar operation of subtyping in programming languages) is known to be decidable, as are the intersection and difference of tree automata (which turn out to be very useful for pattern matching, as we describe below).

On the other hand, tree automata in full generality are quite powerful, and these worst-case complexity of these fundamental operations is correspondingly high (in particular, language inclusion checking can take exponential time in the size of the automata [Seidl 1990]). The most important choice in the XDuce design has been to accept this worst-case complexity, in return for a clean and powerful language design, rather than imposing language restrictions to reduce it. This choice entails significant work in the implementation to develop algorithms that are efficient enough for practical use; our results in this area are described in three companion articles [Hosoya et al. 2000; Hosoya and Pierce 2001; Hosoya 2003].

Another novel feature of XDuce is a powerful form of pattern matching derived directly from the type system, called *regular expression pattern matching*. Regular expression patterns combine conditional branching, tag checking, and subtree extraction. They are related to the pattern matching constructs found in many functional languages, but extend these constructs with the ability to write "recursive patterns" that precisely describe trees of arbitrary size; also, arbitrary type expressions may appear inside patterns, essentially incorporating dynamic type-analysis of tree structures into the pattern matching mechanism.

In our prototype implementation of XDuce, most of the sophistication lies in the algorithms for typechecking and for analysis of patterns; the back end is a simple (and not terribly fast) interpreter. We have used this prototype to develop a number of small XML-processing applications. During this development, we often

description in any of these formats.

found that the typechecker discovered subtle programming mistakes that would have been quite troublesome to find by hand. For example, in XHTML, a `table` tag is required to have at least one `tr` (table row) tag in it and, if the table is empty, the `table` tag itself must not exist at all. It is easy to write code that violates these rules, and the static type system was helpful in detecting such mistakes early. The source code of our implementation is publically available; interested readers are invited to visit the XDuce home page at `http://xduce.sourceforge.net`.

In earlier work [Hosoya et al. 2000; Hosoya and Pierce 2001; Hosoya 2003], we studied the consituent features of XDuce in isolation, concentrating on semantic and algorithmic issues. The focus of this article is on the language design as a whole. We illustrate the key features of its type system and pattern matching primitives and discuss the considerations motivating their design (Sections 2 to 4), present a formal definition of the complete core language and a proof of type soundness (Section 5), discuss a range of related work (Section 6), and sketch ongoing extensions (Section 7).

2. BASIC CONCEPTS

This section introduces the basic features of the XDuce language: values, types, pattern matching, and functions.

2.1 Values

Run-time values in XDuce are fragments of XML documents. These fragments can be built from scratch or by combining smaller fragments, or can result from destructing existing values using pattern matching.

For example, consider the following XML document.

```
<addrbook>
  <person> <name> Haruo Hosoya </name>
           <email> hahosoya@kyoto-u </email>
           <email> hahosoya@upenn </email> </person>
  <person> <name> Benjamin Pierce </name>
           <email> bcpierce@upenn </email>
           <tel> 123-456-789 </tel> </person>
</addrbook>
```

A node is written as a pair of an open tag `<label>` and its corresponding closing tag `</label>`. The children of the node appear between these tags, and the order of the children of a node is significant. The above document can thus be described as a tree where the root node is labeled `addrbook` and contains two nodes with label `person`. The first `person` contains a `name` and two `emails`; similarly the second `person` contains a `name`, an `email`, and a `tel`. Each `name`, `email`, or `tel` node contains a string.

To be precise, each value in XDuce is a *sequence* of *nodes*.[2] (I.e., the whole document above is actually a singleton sequence; the sequence of children of each node forms a single value.) XDuce provides several operations for constructing

[2]Supporting XML attributes in XDuce is on-going work. See Section 7 for a related discussion.

such sequences. For example, here is a fragment of program code that produces the document shown above.[3]

```
addrbook[
  person[name["Haruo Hosoya"],
         email["hahosoya@kyoto-u"],
         email["hahosoya@upenn"]],
  person[name["Benjamin Pierce"],
         email["bcpierce@upenn"],
         tel["123-456-789"]]]
```

An expression of the form `l[e]` constructs a singleton sequence of a label `l` containing a sequence resulted from evaluating the expression `e`. Comma is a binary operator that concatenates two sequences. Note that comma is associative—both `(e1,e2),e3` and `e1,(e2,e3)` produce the same sequence—we therefore drop parentheses from such expressions. String literals are written `"..."`. In addition to these operations, we also have a constructor `()` for creating the empty sequence.

2.2 Types

Types are descriptions of sets of structurally similar values. For example, the type

```
person[name[String], email[String], tel[String]]
```

describes values consisting of the single label `person` containing a sequence of `name`, `email`, and `tel` labels, each containing a string. Now let us slightly complicate this type.

```
person[name[String], email[String]*, tel[String]?]
```

The difference between this type and the previous one is that zero or more `email`s can follow after the `name` label, as indicated by the "`*`," and the `tel` label may be omitted, as indicated by the "`?`".

XDuce's types are called *regular expression types* because they closely resemble ordinary string regular expressions, the only difference being that they describe sequences of tree nodes, whereas string regular expressions describe sequences of characters. As "atoms," we have labeled types like `label[T]` (which denotes the set of sequences containing a single subtree labeled `label`), base types such as `String`, and the empty sequence type `()`. Types can be composed by concatenation (comma), zero-or-more-times repetition (`*`), one-or-more-times repetition (`+`), optionality (`?`), and alternation (`|`, also called union).

Type expressions can be given names in XDuce programs by type definitions. For example:[4][5]

[3]The XDuce implementation supports two ways of creating values: use sequence constructors or load an external XML document, e.g., from the file system. For the latter case, we perform a validation check of the incorporated document against an intended type.

[4]Again, the XDuce implementation supports two ways of declaring types: use type definitions in the XDuce native syntax or import existing DTDs from the external environment. Note that our types are more general than DTD, e.g., we allow the same label to have different contents depending the context. See Section 3.3 for a related discussion.

[5]Note that type definitions declare type names, not labels. Indeed, labels do not have to be declared at all; they can exist by themselves (just like Lisp's symbols).

```
type Person   = person[Name,Email*,Tel?]
type Name     = name[String]
type Email    = email[String]
type Tel      = tel[String]
```

That is, the type named Person is defined to be an abbreviation for the type person[Name,Email*,Tel?], which uses Name, Email, and Tel to refer to the types associated with these names. Type definitions are convenient for avoiding repetition of large type expressions in programs. More importantly, though, they may be (mutually) recursive; we will discuss this possibility further in Section 3.1.

2.3 Typechecking

XDuce uses types for various purposes. The most important is in checking that the values that may be consumed and produced by each function definition are consistent with its explicitly declared argument and result types. For example, consider the following function definition.

```
fun make_person (val nm as String)(val str as String) : Person =
  person[name[nm],
         (if looks_like_telnum(str) then tel[str] else email[str])]
```

The first line declares that the function make_person takes two parameters nm and str of type String and returns a value of type Person. (The val keyword in the function header is a signal that the following identifier is a bound variable.) In the body, we create a tree labeled person that contains two subtrees. The first is labeled name and contains the string nm. The second is labeled either tel or email, depending on whether the argument str looks like a telephone number or an email (according to some function looks_like_telnum defined elsewhere). The typechecking of this function proceeds as follows. From the type String of the variables nm and str, we can easily compute the type of each expression in the body. A labeled expression has a labeled type; a concatenation expression has the concatenation of the types of the subexpressions; an if expression has the union of the types of the then and else branches. As a result, the type of the whole body is this:

```
person[name[String], (email[String] | tel[String])]
```

Finally, we check that this type is a subtype of the annotated result type Person.

The next question, then, is when two types are in the subtype relation. The standard answer would involve giving a collection of subtyping laws corresponding to some intuitive notion of inclusion between sets of members. However, our types based on regular expressions yield many algebraic laws (including associativity of concatenation and union, commutativity of union, distributivity of union over concatenation or labelling) and all of these play crucial roles in XML processing as described in our previous paper [Hosoya et al. 2000]; enumerating all these laws as rules would make the specification rather complicated. We therefore adopt a more direct strategy: we first define which values belong to each type (the details are straightforward; see Section 5). We then say that one type is a subtype of another *exactly* when the former denotes a subset of the latter.[6]

This "semantic definition" yields a subtyping relation that is both intuitive and powerful. For example, consider again the type from the example just above and the `Person` type from the previous section. These types are syntactically quite different. But we can easily check that they fall in the subtype relation, since, in the first type, the sequence after the `name` is either one `email` (followed by no `tel`) or one `tel` (preceded by zero `emails`); both of these cases are also described by the second type.

The remaining question is how efficiently we can check subtyping. We know, from the theory of finite tree automata, that this decision problem takes exponential time in the general case. However, by choosing appropriate representations and applying a few domain-specific heuristics, we can obtain an algorithm whose speed is quite acceptable in practice. This algorithm is described in detail in [Hosoya et al. 2000].

In the example above, the subtype checker was invoked to verify that the actual type of a function's body is a subtype of the programmer-declared result type. Another use of subtyping is in checking the type of an argument to a function call against the parameter type given by the programmer. For example, if we have the function definition

```
fun print_fields (val fs as (Name|Tel|Email)*) : () =
  ...
```

we can apply it to an argument of the following type:

```
Name,Email*,Tel?
```

Note, again, that, though they are syntactically quite different, the argument type and the parameter type are in the subtype relation (the ordering constraint in the argument type is lost in the parameter type, yielding a strictly larger set).

2.4 Pattern matching

So far, we have focused on building values. We now turn our attention to decomposing existing values by pattern matching.

As a simple example, consider the following pattern match expression for creating a URL string from a value labeled with the protocol name. (The binary operator `^` is a string concatenation.)

```
match v with
    www[val s as String]    ->  "http://" ^ s
  | email[val s as String]  ->  "mailto:" ^ s
  | ftp[val s as String]    ->  "ftp://" ^ s
```

This pattern match branches depending on the top label (`www`, `email`, or `ftp`) of the input value and evaluates the corresponding body expression, which prepends

[6] We should note a somewhat special fact about XDuce's type system that makes this direct construction attractive. Since XDuce is a first-order language (functions cannot be passed as arguments to other functions), the type system does not need to deal with arrow types, unlike most functional languages. The absence of arrow types greatly simplifies the semantics of types in XDuce. Recently, however, Frisch, Castagna, and Benzaken have shown how to extend the semantic construction to include arrow types [Frisch et al. 2002].

the appropriate string to the variable `s`, which is bound to the content the label in the input value `v`.

In general, a pattern match expression takes an input value and a set of clauses of the form "*pattern* `->` *expression*." Given an input value, the pattern matcher finds the first clause whose pattern matches the value. It extracts the subtrees corresponding to bound variables in the pattern and then evaluates the corresponding body expression in an environment enriched with these bindings.

Patterns can be nested to test for the simultaneous presence of multiple labels and extract multiple subtress. For example, the pattern

```
person[name[val n as String], email[val e as String]]
```

matches a `person` label whose content is a `name` label followed by an `email` label. Also, the logical-or can be expressed by the union operator.

```
email[val s as String] | tel[val s as String]
```

Indeed, XDuce patterns have *exactly* the same form as type expressions, except that they may include variable binders of the form "`val x as` *pattern*" (which matches the input value against *pattern* as well as binding the variable `x` to the whole value). We demand that, for any input value, a pattern yields exactly one binding for each variable (we call this condition "linearity." See Section 5.4.2 for the precise definition.) Thus, a pattern like

```
email[val e as String] | tel[val t as String]
```

or

```
email[val e as String]*
```

is forbidden.

Since patterns are just types decorated with variable binders, we can even use patterns to perform dynamic typechecking. For example, the pattern

```
person[Name, Email+, Tel+]
```

matches the subset of elements of `Person` that contain a value of type `Name` followed by one or more values of type `Email` and then one or more values of type `Tel`. This capability is beyond the expressiveness of pattern matching facilities in conventional functional languages such as ML and Haskell.

XDuce's pattern matching has a "first-match" semantics. That is, a pattern match expression tries its clauses from top to bottom and fires the first matching one. This semantics is particularly useful to write default cases. For example, in the following pattern match expression

```
match v with
    person[name[val n as String], Email+, Tel+]  ->  ...
  | person[name[val n as String], Any]           ->  ...
```

the first clause matches when the input `person` value contains both `emails` and `tels`, and the second clause matches otherwise. (`Any` is a type that matches any values. See Section 3.2.) If such overlapping patterns were not permitted, we would have to rewrite the second pattern so as to negate the first one, which would be quite cumbersome.

What if none of the clauses match? XDuce performs a static exhaustiveness check so that such a failure can never arise. In Section 4, we will discuss in detail various static checks on patterns, including exhaustiveness (every value is matched by some clause), irredundancy (every clause can match some value, so that every clause body is reachable) and non-ambiguity (every pattern yields a unique binding for every value that it matches).

2.5 A Complete Example

Let us now look at a small but complete program. The task of this program is to create, from an address book document, a telephone book document by extracting just the entries with telephone numbers.

We first show the type definitions for input documents (partly repeated from above)

```
type Addrbook = addrbook[Person*]
type Person   = person[Name,Email*,Tel?]
type Name     = name[String]
type Email    = email[String]
type Tel      = tel[String]
```

and output documents.

```
type TelBook   = telbook[TelPerson*]
type TelPerson = person[Name,Tel]
```

The first thing we do is to load an address book document from a file and validate it against the type `Addrbook`. (We do not assume loaded documents to conform to any type.)

```
let val doc = load_xml("mybook.xml")
let val valid_doc = validate doc with Addrbook
```

We then extract the content of the top label `addrbook` and send it to the function `make_tel_book` (defined below). Finally, we enclose the result with the label `telbook` and save it to a file.

```
let val out_doc =
  match valid_doc with
    addrbook[val persons as Person*] ->
      telbook[make_tel_book(persons)]
save_xml("output.xml")(out_doc)
```

The function `make_tel_book` takes a value `ps` of type `Person*` and returns a value of type `TelPerson*`.

```
fun make_tel_book (val ps as Person*) : TelPerson* =
  match ps with
      person[name[val n as String], Email*, tel[val t as String]],
                                                  val rest as Person*
        ->  person[name[n], tel[t]], make_tel_book(rest)
    | person[name[val n as String], Email*], val rest as Person*
        ->  make_tel_book(rest)
    | ()
        ->  ()
```

The body of the function uses a pattern match to analyze `ps`. In the first case, the input sequence has a `person` label that contains a `tel` label; we pick out the `name` and `tel` components from the `person`, construct a new `person` label with them, and recursively call `make_tel_book` to process the remainder of the sequence. In the second case, the input sequence has a `person` label that does *not* contain a `tel` label; we simply ignore this `person` label and recursively call `make_tel_book`. In the last case, the input sequence is empty; we return the empty sequence itself.

3. MORE ON TYPES

This section gives some examples of more interesting uses of XDuce types.

3.1 Recursive Types

Like most programming languages, XDuce supports recursive types for describing arbitrarily nested structures. Consider the following definitions.

```
type Fld = Rcd*
type Rcd = name[String], folder[Fld]
         | name[String], url[String], (good[] | broken[])
```

(The form `label[]` is a short hand for `label[()]`, where `()` is the empty sequence type.) The mutually recursive types `Fld` ("folder") and `Rcd` ("record") define a simple template for storing structured lists of bookmarks, such 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 either a `good` or a `broken` tag indicating whether or not the link is broken.

We can write another pair of types

```
type GoodFld = GoodRcd*
type GoodRcd = name[String], folder[GoodFld]
             | name[String], url[String], good[]
```

which are identical to `Fld` and `Rcd` except that links are all good. Note that `GoodFld` is a subtype of `Fld` (it describes values with the same structure, but with stronger constraints).

3.2 Label Classes

The labeled types that we have seen so far have the form `l[T]` and describe singleton sequences labeled exactly with `l`. XDuce actually generalizes such types to allow more complex forms called "label class" that represent sets of possible labels. (This idea is also present in other XML type systems such as RELAX NG [Clark and Murata 2001].) The `l` in the form `l[T]` is a label class representing a singleton set.

The label class `~` represents the set of all labels. Using `~`, we can define a type `Any` that denotes the set of all values:

```
type Any = (~[Any] | Int | Float | String)*
```

That is, any value can be any repetition of either any label containing any value or any base value. (At the moment, XDuce supports only `Int`, `Float`, and `String` as base types. If we added more base types, we would have to modify the above definition of `Any` accordingly.)

We also allow a label class of the form (l_1|..|l_n), representing the choice between several labels. Such label classes are useful for describing a labeled type that has multiple possible labels, all with the same content type. For example, HTML headings may be labeled `h1` through `h6`, all with the content type `Inline`:

```
type Heading = (h1|h2|h3|h4|h5|h6)[Inline]
```

Finally, we allow "negation" label classes of the form ^(l_1|..|l_n), which represent the set of all labels except l_1, ..., and l_n. For example, we can use such label classes in the following way

```
match v with
  ^(h1|h2)[Inline]*, (h1|h2)[val c as Inline], Any  -> ...
```

where we extract the content of the first `h1` or `h2` label in the given value, ignoring all the other labels prior to this.

3.3 Union Types

Functionality similar to union types is provided in most schema languages for XML. However, some schema languages make restrictions on what types can be joined by a union, leading to significant differences in expressiveness. For example, XML Schema [Fallside 2001] and the XML Query Algebra [Fernández et al. 2001] require that the types to be unioned should have disjoint sets of top-level labels. On the other hand, XDuce and RELAX NG [Clark and Murata 2001] impose no restriction.[7]

The main advantage of having no restriction is that we can express dependencies between subtrees. Consider the following example (suggested by Murata). Suppose that we are designing a schema for LaTeX-like documents. We want to express a division into structures such as chapter, section, and subsection, with the following requirements. A chapter can contain only sections, a section can contain only subsections, and so on. Also, both chapters and sections can appear at the top level. One obvious way of implementing these division structures is to directly use labels `chapter`, `section`, etc., as in the following type definitions.

```
type Top = (Chapter | Section | Text)*
type Chapter = chapter[(Section | Text)*]
type Section = section[(Subsection | Text)*]
type Subsection = subsection[...]
```

(`Text` represents normal texts and is assumed to be defined somewhere else.)

However, in some cases, we may prefer to use the same label `div` to represent all divisions for programming convenience. To distinguish between different kinds of divisions, we add a field `kind` containing a discriminating tag:

```
type Top = (Chapter | Section | Text)*
type Chapter = div[kind[chapter[]],
                   (Section | Text)*]
```

[7]This difference corresponds to different kinds of tree automata. Types with the label-disjointness restriction correspond to *deterministic top-down tree automata*, whereas those with no restriction correspond to more general *nondeterministic (top-down) tree automata*. See [Comon et al. 1999] for more details.

```
type Section = div[kind[section[]],
                   (Subsection | Text)*]
type Subsection = div[kind[subsection[]], ...]
```

Notice that, in a `div` label appearing at the top level (in either `Chapter` or `Section`), the content of the `kind` label (either `chapter[]` or `section[]`) and the type coming after it (either `(Section|Text)*` or `(Subsection|Text)*`) are interdependent. That is, we cannot have both `chapter[]` in `kind` and a value of type `Subsection` at the same time, for example. Such a dependency cannot be expressed in type systems that adopt the label-disjointness restriction mentioned above.

When programming with this set of type definitions, we may wish to process any division uniformly, forgetting the dependency just discussed. For this, XDuce's subtyping facility is useful. The type `Top` (which has the dependency) is a subtype of the following type

```
type Top2 = div[kind[chapter[] | section[]],
                (Section | Subsection | Text)*]*
```

which collapses the dependency. This can be useful when we want to perform exactly the same operation on each top-level `div`, whether it is a `chapter` or a `section` (e.g., counting the maximum depth of divisions).

4. MORE ON PATTERN MATCHING

XDuce performs a number of static analyses on pattern matches: exhaustiveness, irredundancy, ambiguity checks, and local type inference. In this section, we illustrate these analyses by example.

4.1 Exhaustiveness and Irredundancy Checks

During typechecking, XDuce checks each pattern match expression for exhaustiveness—i.e., it makes sure that every possible input value should be matched by some clause. For example, suppose that the variable `p` has type `Person` (defined as `person[Name,Email*,Tel?]`) and consider the following pattern match:

```
match p with
    person[Name, Email+, Tel?]  ->  ...
  | person[Name, Email*, Tel]   ->  ...
```

The first clause matches `person` values with at least one `email` and the second matches `person` values with one `tel`. This pattern match is *not* exhaustive since it does not cover the case when the input `person` value contains neither an `email` nor a `tel`. Thus, the XDuce typechecker rejects this pattern match. In order to make it exhaustive, we could add the following clause:

```
  | person[Name]   ->  ...
```

A related check is for irredundancy of pattern matches. XDuce rejects a pattern match expression if it has a clause whose pattern will never be able to match any input value. For example, consider the following pattern match.

```
match p with
    person[Name, Email*, Tel?]  ->  ...
  | person[Name, Email+, Tel]   ->  ...
```

This pattern match is redundant, since all values that might be matched by the second clause are already covered by the first clause. Therefore we report an error for this pattern match.

It may appear that irredundancy checks are not particularly important since this sort of "a clause being completely covered by the preceding ones" happens rather infrequently. However, irredundancy checks are also valuable for detecting two other kinds of much sillier and more common mistakes. One is misspelling of labels, which usually makes a clause never match any input value. The other kind is misunderstanding of the structure among labels (e.g., switching the order of `Email` and `Tel` types in the above example), which also tends to make a clause redundant. In many cases, these two can also be detected by an exhaustiveness check since the values that are intended to be matched by the clause are actually not covered. However, an exhaustiveness check becomes useless when the pattern match contains a default case. For example, consider the following

```
match v with
    preson[Name, Email+, Tel]  ->  ...
  | Any                        ->  ...
```

where the label `preson` is a misspelling. An exhaustiveness check cannot find the error since the pattern match is vacuously exhaustive, whereas an irredundancy check can detect it since the first clause never matches.

In exhaustiveness and irredundancy checks, the theory of finite tree automata plays an important role. An exhaustiveness check is done by examining whether "the set of values in the input type is *included* in the set of values matched by each of the patterns." Since patterns are essentially the same as types, this is just a subtype check. Similarly, irredundancy is checked by examining whether "the set of values that are both in the input type *and* matched by a pattern is *included* in the set of values matched by the preceding patterns." Here, we rely on the fact that we may always calculate the intersection of two tree automata. A detailed discussion of the required algorithms can be found in a companion paper [Hosoya and Pierce 2001].

The XML Query Algebra [Fernández et al. 2001] provides "case expressions" for performing matching of input values against a series of patterns (similar to our patterns but somewhat simpler). However, XML Query Algebra supports neither exhaustiveness nor irredundancy checks. Exhaustiveness check would not make sense in their setting, since their case expressions are syntactically required to have a default case. On the other hand, irredundancy checks in the style of XDuce seem to make sense in their setting.

4.2 Ambiguity Checking

We say that a pattern is ambiguous if it yields multiple possible "parses" of some input value.

Given an input value and a pattern, pattern matching assigns each label in the value to a correspondingly labeled subpattern. This assignment is called a parse. For example, consider the following pattern match with the input type `(a[]|b[])*`.

```
match v with
  a[]*, (val x as b[]), (a[]|b[])*  ->  ...
```

The behavior of this pattern is to skip all the consecutive a labels from the beginning of the input sequence, bind the variable x to the first b label after these, and then ignore the remaining sequence of as and bs. Thus, the parse yielded by this pattern matching assigns all the skipped a labels to the leftmost a[] pattern, the first b label to the b[] pattern in the middle, the other a labels to the rightmost a[] pattern, and the other b labels to the rightmost b[] pattern. This pattern is unambiguous.

A pattern is ambiguous if it may yield multiple parses for some input value. For example, the following is ambiguous.

```
match v with
  (a[]|b[])*, (val x as b[]), (a[]|b[])*  ->  ...
```

Take the input value b[],b[]. There are two parses for this value. One assigns the first b to the leftmost b[] pattern and the second b to the middle b[] pattern. The other parse assigns the first b to the middle b[] pattern and the second b to the rightmost b[] pattern. Note that, in the pattern match shown first, the only possible parse for this input value is the second one. A more formal definition of ambiguity can be found in Section 5.4.6.

Usually, an ambiguous pattern signals a programming error. However, we have found that, in some cases, writing ambiguous patterns is reasonable. One typical case is when the application program knows, from implicit assumptions that cannot be expressed in types, that there is only one possible parse. For example, suppose that we have a simple book database of type Book* where

```
type Book = book[key[String], title[String], author[String]]
```

where we assume that there is only one book entry with the same key field value. We can extract a book with a specified key from this database by writing the following pattern match.

```
match db with
  Book*,
  book[key["Pierce2002"],
       title[val t as String],
       author[val a as String]],
  Book*  ->  ...
```

Note that the above assumption for keys guarantees that the entry yielded by this pattern match is unique.

Since writing a nonambiguous pattern is sometimes much more complicated than an ambigous one, requiring disambiguation even in situations that do not necessiate it can be a heavy burden for the user. (In the above pattern, we would only have to replace the first occurrence of Book with a type representing books with keys other than "Pierce2002". However, this could become more cumbersome if keys have a complex structure.) Therefore we decided to yield a warning for ambiguity rather than an error. In the case that the user writes an ambiguous pattern and ignores the warning, the semantics of pattern matching is to choose an arbitrary parse among multiple possibilities ("nondeterministic semantics").[8]

[8]Previously, XDuce used a first-match policy to resolve ambiguity even within a single pattern

4.3 Type Inference for Patterns

The type annotations on pattern variables are normally redundant. For example, in the following pattern match taking values of type `Person*`,

```
match ps with
    person[name[val n as String], Email*, Tel?], val rest as Person*
      -> ...
  | ...
```

the type (`String`) of the variable `n` and the type (`Person*`) of the variable `rest` can be deduced from the input type and the shape of the patterns. XDuce supports a mechanism that automatically infers such type annotations.

Our type inference scheme is *local* and *locally precise*. By local, we mean that the type of each pattern variable is inferred only from the input type and the pattern itself. (We do not consider long distance dependencies, e.g., constraints on pattern variables arising from the expressions in the bodies of the match branches.) By locally precise, we mean that, if we match all of the values from the input type against the pattern, the inferred type for a pattern variable precisely represents the set of values that the variable can be bound to. (Note, again, that this is a semantic definition: the specification of type inference depends on the dynamic semantics of pattern matching.) With type inference, the example above can be rewritten as follows.

```
match ps with
    person[name[val n], Email*, Tel?], val rest
      -> ...
  | ...
```

From the examples we have seen, it might appear that, whenever a pattern contains a binding of the form `(val x as T)`, the inferred type for `x` is `T` itself. It is not always the case, however—our type inference may compute a more precise type than `T`. Formally, the syntactic form `(val x)` is an abbreviation for `(val x as Any)`—that is, we continue to require type annotations on all pattern variables, but we allow them to be larger than necessary. The actual types of the variables are inferred by combining the types given by the programmer with the types discovered by propagating the input type through the pattern. For example, the pattern

```
match v with
    (val head as ~[Any]), val tail  ->  ...
```

binds `head` to the first labeled value in the input sequence and `tail` to the rest of the sequence. The types inferred for `head` and `tail` depend on the input type. For example, if the input type is `(Email|Tel)*`, then we infer `(Email|Tel)` for `head` and `(Email|Tel)*` for `tail`. If the input type is `(Email*,Tel)`, then we

clause [Hosoya and Pierce 2001]. However, we decided to throw this idea away, first because patterns behave in an quite unintuitive way once they become large, and second because guaranteeing first match semantics makes the implementation more complicated. See [Hosoya 2003] for more details.

infer (Email|Tel) for head and (Email*,Tel)? for tail.[9] This combination of declared and inferred structure is useful since it is often more concise to write a rough pattern (like the above ~[Any] pattern) than a precise one. In addition, if the input type is changed later on, we may not have to change the pattern, since the type inference will recompute appropriate types for the variables.

When a pattern match has multiple clauses, our type inference scheme takes the first-match semantics into account. That is, in inferring types for pattern variables for a given clause, the values that are already captured by the preceding clauses are excluded. For example, in the following pattern match with the input type Person,

```
match p with
   person[name[val n], Email*, tel[val t]]
     ->  ...
 | person[val c]
     ->  ...
```

we infer the type (Name,Email*) for the variable c, since persons with a tel are already taken by the first clause and only persons without a tel will reach the second clause. For handling this form of "exclusion," our type inference computes a difference between the set of input values and the set of values matched by the preceding patterns. Again, we exploit a closure property (closure under difference) of finite tree automata. See [Hosoya and Pierce 2001; Hosoya 2003] for details.

5. FORMAL DEFINITION OF THE CORE XDUCE LANGUAGE

This section presents a complete, formal definition of the core features of XDuce—types, patterns, terms, typechecking, pattern matching, and evaluation—and establishes basic soundness theorems. We assuming familiarity with basic notations and techniques from type systems and operational semantics (background on these topics may be found, for example, in [Pierce 2002]).

5.1 Labels and label classes

We assume given a (possibly infinite) set L of *labels*, ranged over by l. We then define *label classes* as follows:

L	::=	l	specific label
		~	wildcard label
		L|L	union
		L\L	difference

[9]The power of the type inference scheme has been improved from the one described in our previous paper [Hosoya and Pierce 2001]. In the previous one, we were able to infer precise types only for variables in tail positions. For example, in the pattern

```
match v with (val head as ~[Any]), val tail
```

we could infer a precise type for tail but not for head (we simply extracted the type ~[Any] directly from the pattern, which is less precise). This was due to a naiveness of the inference algorithm that we used at that time. However, since then, we have developed a new algorithm that overcomes this limitation and have incorporated it in the current XDuce [Hosoya 2003].

The semantics of label classes is defined by a denotation function $[\![\cdot]\!]$ mapping label classes to sets of labels.

$$\begin{array}{lcl}
[\![\mathtt{l}]\!] & = & \{\mathtt{l}\} \\
[\![\tilde{\ }]\!] & = & L \\
[\![\mathtt{L}_1 \,|\, \mathtt{L}_2]\!] & = & [\![\mathtt{L}_1]\!] \cup [\![\mathtt{L}_2]\!] \\
[\![\mathtt{L}_1 \backslash \mathtt{L}_2]\!] & = & [\![\mathtt{L}_1]\!] \setminus [\![\mathtt{L}_2]\!]
\end{array}$$

We write $\mathtt{l} \in \mathtt{L}$ for $\mathtt{l} \in [\![\mathtt{L}]\!]$.

5.2 Values

For brevity, we omit base values such as strings. (The changes required to add them are straightforward.) A *value* `v`, then, is just a sequence of labeled values, where a labeled value is a pair of a label and a value. We write `()` for the empty sequence, `l[v]` for a labeled value, and $\mathtt{v}_1\mathtt{,v}_2$ for the concatenation of two values.

5.3 Types

5.3.1 *Syntax.* We assume given a countably infinite set of type names, ranged over by `X`. *Types* are now defined as follows:

$$\begin{array}{lcll}
\mathtt{T} & ::= & \mathtt{X} & \text{type name} \\
 & & \mathtt{()} & \text{empty sequence} \\
 & & \mathtt{T,T} & \text{concatenation} \\
 & & \mathtt{L[T]} & \text{labeling} \\
 & & \mathtt{T|T} & \text{union} \\
 & & \mathtt{T*} & \text{repetition}
\end{array}$$

The interpretations of type names are given by a single, global set E of type definitions of the following form.

```
type X = T
```

The body of each definition may mention any of the defined type names (in particular, definitions may be recursive). We regard E as a mapping from type names to their bodies and write $E(\mathtt{X})$ for the right-hand side of the definition of `X` in E.

To ensure that types correspond to *regular* tree automata (rather than context-free grammars), we impose a syntactic restriction that disallows recursion "at the top level" of definitions. For a given type `T`, we define the set $S(\mathtt{T})$ of type names reachable from `T` at the top level as the smallest set satisfying the following.

$$S(\mathtt{T}) = \begin{cases}
S(E(\mathtt{X})) \cup \{\mathtt{X}\} & \text{if } \mathtt{T} = \mathtt{X} \\
S(\mathtt{T}_1) & \text{if } \mathtt{T} = \mathtt{T}_1\mathtt{*} \\
S(\mathtt{T}_1) \cup S(\mathtt{T}_2) & \text{if } \mathtt{T} = \mathtt{T}_1\mathtt{,T}_2 \text{ or } \mathtt{T} = \mathtt{T}_1\mathtt{|T}_2 \\
\emptyset & \text{otherwise.}
\end{cases}$$

We then require that the set E of type definitions satisfies

$$\mathtt{X} \notin S(E(\mathtt{X})) \text{ for all } \mathtt{X} \in dom(E).$$

The additional regular expression operators `?` and `+` are obtained as syntactic sugar:

$$\begin{array}{lcl}
\mathtt{T?} & \equiv & \mathtt{T|()} \\
\mathtt{T+} & \equiv & \mathtt{T,T*}
\end{array}$$

5.3.2 *Semantics.* The semantics of types is given by the relation $\mathtt{v} \in \mathtt{T}$, read "value v has type T"—the smallest relation closed under the following set of rules.

$$() \in () \tag{ET-Emp}$$

$$\frac{E(\mathtt{X}) = \mathtt{T} \qquad \mathtt{v} \in \mathtt{T}}{\mathtt{v} \in \mathtt{X}} \tag{ET-Var}$$

$$\frac{\mathtt{v} \in \mathtt{T} \qquad \mathtt{l} \in \mathtt{L}}{\mathtt{l[v]} \in \mathtt{L[T]}} \tag{ET-Lab}$$

$$\frac{\mathtt{v}_1 \in \mathtt{T}_1 \qquad \mathtt{v}_2 \in \mathtt{T}_2}{\mathtt{v}_1\mathtt{,v}_2 \in \mathtt{T}_1\mathtt{,T}_2} \tag{ET-Cat}$$

$$\frac{\mathtt{v} \in \mathtt{T}_1}{\mathtt{v} \in \mathtt{T}_1 \,|\, \mathtt{T}_2} \tag{ET-Or1}$$

$$\frac{\mathtt{v} \in \mathtt{T}_2}{\mathtt{v} \in \mathtt{T}_1 \,|\, \mathtt{T}_2} \tag{ET-Or2}$$

$$\frac{\mathtt{v}_i \in \mathtt{T} \quad \text{for each } i}{\mathtt{v}_1, \ldots, \mathtt{v}_n \in \mathtt{T*}} \tag{ET-Rep}$$

5.3.3 *Subtyping.* A type S is a *subtype* of another type T, written S <: T, iff $\mathtt{v} \in \mathtt{S}$ implies $\mathtt{v} \in \mathtt{T}$ for all v.

5.3.4 *Intersection and difference.* A type U is an *intersection* of types S and T, written by $\mathtt{S} \cap \mathtt{T} \Rightarrow \mathtt{U}$, iff $\mathtt{v} \in \mathtt{S}$ and $\mathtt{v} \in \mathtt{T}$ imply $\mathtt{v} \in \mathtt{U}$ and vice versa, for all v. (There can be more than one intersection of two given types, but all will describe the same set of values.) Similarly, a type U is a *difference* between types S and T, $\mathtt{S} \setminus \mathtt{T} \Rightarrow \mathtt{U}$, iff $\mathtt{v} \in \mathtt{S}$ and $\mathtt{v} \notin \mathtt{T}$ together imply $\mathtt{v} \in \mathtt{U}$ and vice versa, for all v.

5.4 Pattern language

5.4.1 *Syntax.* We assume a countably infinite set of pattern names, ranged over by Y, and a countably infinite set of variables, ranged over by x. Pattern expressions are now defined as follows.

P	::=	Y	pattern name
		val x as P	variable binder
		L[P]	label
		()	empty sequence
		P,P	concatenation
		P|P	choice
		P*	repetition

The bindings of pattern names to patterns are given by a fixed, global, mutually recursive set F of pattern definitions of the following form.

```
pat Y = P
```

For technical convenience, we assume that F includes all the type definitions in E, regarding the type expressions appearing in E as pattern expressions in the obvious way. Conversely, we assume that E includes all the pattern definitions in F with all the variable binders erased. We write $\mathit{tyof}(\mathtt{P})$ for the type obtained by erasing all variable binders from `P`. Pattern definitions must obey the same well-formedness restriction as type definitions.

We allow the same abbreviations for regular expression operators (`+` and `?`). Also, `val x` can be used to mean `val x as Any`, where we assume the following fixed type definition in E.[10]

```
type Any = ~[Any]*
```

5.4.2 *Linearity.* Let $\mathit{reach}(\mathtt{P})$ be the set of all variable bindings reachable from `P`—that is, the smallest set satisfying the following:

$$\mathit{reach}(\mathtt{P}) \quad = \quad BV(\mathtt{P}) \cup \bigcup_{\mathtt{Y} \in FN(\mathtt{P})} \mathit{reach}(F(\mathtt{Y})),$$

where $BV(\mathtt{P})$ is the set of variables bound in `P` and $FN(\mathtt{P})$ is the set of pattern names appearing in `P`. We say that a pattern `P` is *linear* iff, for any (reachable) subphrase $\mathtt{P}'$ of `P`, the following conditions hold.

—$\mathtt{x} \notin \mathit{reach}(\mathtt{P}_1)$ if $\mathtt{P}' = \mathtt{val\ x\ as\ P}_1$.
—$\mathit{reach}(\mathtt{P}_1) \cap \mathit{reach}(\mathtt{P}_2) = \emptyset$ if $\mathtt{P}' = \mathtt{P}_1\mathtt{,P}_2$.
—$\mathit{reach}(\mathtt{P}_1) = \mathit{reach}(\mathtt{P}_2)$ if $\mathtt{P}' = \mathtt{P}_1 \mathtt{|} \mathtt{P}_2$.
—$\mathit{reach}(\mathtt{P}_1) = \emptyset$ if $\mathtt{P}' = \mathtt{P}_1\mathtt{*}$.

In the following, we assume that all patterns are linear.

5.4.3 *Semantics.* We describe the semantics of patterns by first defining the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$, read "`v` is matched by `P`, yielding `V`," where an environment `V` is a finite mapping from variables to values (written $\mathtt{x}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,x}_n\mathtt{:v}_n$). The concatenation of environments binding distinct variables is written with a comma.

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{(val\ x\ as\ P)} \Rightarrow \mathtt{x:v,V}} \qquad \text{(EP-AS)}$$

$$\mathtt{()} \in \mathtt{()} \Rightarrow \emptyset \qquad \text{(EP-EMP)}$$

$$\frac{F(\mathtt{Y}) = \mathtt{P} \qquad \mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{Y} \Rightarrow \mathtt{V}} \qquad \text{(EP-VAR)}$$

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \qquad \mathtt{l} \in \mathtt{L}}{\mathtt{l[v]} \in \mathtt{L[P]} \Rightarrow \mathtt{V}} \qquad \text{(EP-LAB)}$$

[10] If we include base types in the formalization, we need to use the definition of `Any` given in Section 3.2.

$$\frac{\mathtt{v}_1 \in \mathtt{P}_1 \Rightarrow \mathtt{V}_1 \qquad \mathtt{v}_2 \in \mathtt{P}_2 \Rightarrow \mathtt{V}_2}{\mathtt{v}_1,\mathtt{v}_2 \in \mathtt{P}_1\mathtt{,}\mathtt{P}_2 \Rightarrow \mathtt{V}_1\mathtt{,}\mathtt{V}_2} \tag{EP-Cat}$$

$$\frac{\mathtt{v} \in \mathtt{P}_1 \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{P}_1\mathtt{|}\mathtt{P}_2 \Rightarrow \mathtt{V}} \tag{EP-Or1}$$

$$\frac{\mathtt{v} \in \mathtt{P}_2 \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{P}_1\mathtt{|}\mathtt{P}_2 \Rightarrow \mathtt{V}} \tag{EP-Or2}$$

$$\frac{\mathtt{v}_i \in \mathtt{P} \Rightarrow \mathtt{V}_i \quad \text{for each } i}{\mathtt{v}_1,\ldots,\mathtt{v}_n \in \mathtt{P*} \Rightarrow \mathtt{V}_1,\ldots,\mathtt{V}_n} \tag{EP-Rep}$$

Note that linearity ensures that environments that are concatenated in the conclusions of rules EP-As, EP-Cat, and EP-Rep have different domains (e.g., $\mathtt{x} \notin dom(\mathtt{V})$ always holds in rule EP-As).

5.4.4 *Exhaustiveness.* The definitions of exhaustiveness, irredundancy, non-ambiguity, and type inference for pattern-match expressions are all made with respect to an "input type" $\mathtt{T}$ describing the set of values that may be presented to the expression at run time.

A list $\mathtt{P}_1,\ldots,\mathtt{P}_n$ of patterns is *exhaustive* with respect to $\mathtt{T}$, written "$\mathtt{T} \rhd \mathtt{P}_1,\ldots,\mathtt{P}_n$: *exhaustive*," iff, for all $\mathtt{v}$, $\mathtt{v} \in \mathtt{T}$ implies $\mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{V}$ for some $\mathtt{P}_i$ and $\mathtt{V}$.

5.4.5 *Irredundancy.* A list $\mathtt{P}_1,\ldots,\mathtt{P}_n$ of patterns is *irredundant* with respect to $\mathtt{T}$, written "$\mathtt{T} \rhd \mathtt{P}_1,\ldots,\mathtt{P}_n$: *irredundant*," iff, for all $\mathtt{P}_i$, there is a value $\mathtt{v} \in \mathtt{T}$ such that $\mathtt{v} \notin \mathtt{P}_j$ for $1 \le j \le i-1$ and $\mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{V}$ for some $\mathtt{V}$.

5.4.6 *Non-ambiguity.* Our non-ambiguity is defined in terms of the *parsing* relation $\mathtt{v} \in^u \mathtt{P}$, which intuitively means that "$\mathtt{P}$ parses $\mathtt{v}$ uniquely." (We can view it as "there is a unique derivation for the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$.") The parsing relation is defined by the following set of rules.

$$\frac{\mathtt{v} \in^u \mathtt{P}}{\mathtt{v} \in^u \mathtt{(val\ x\ as\ P)}} \tag{EUP-As}$$

$$\mathtt{()} \in^u \mathtt{()} \tag{EUP-Emp}$$

$$\frac{F(\mathtt{Y}) = \mathtt{P} \qquad \mathtt{v} \in^u \mathtt{P}}{\mathtt{v} \in^u \mathtt{Y}} \tag{EUP-Var}$$

$$\frac{\mathtt{v} \in^u \mathtt{P} \qquad \mathtt{l} \in \mathtt{L}}{\mathtt{l[v]} \in^u \mathtt{L[P]}} \tag{EUP-Lab}$$

$$\frac{\begin{array}{c}\text{there are unique } \mathtt{v}_1,\mathtt{v}_2 \text{ s.t. } \mathtt{v} = \mathtt{v}_1\mathtt{,}\mathtt{v}_2 \\ \mathtt{v}_1 \in^u \mathtt{P}_1 \qquad \mathtt{v}_2 \in^u \mathtt{P}_2\end{array}}{\mathtt{v} \in^u \mathtt{P}_1\mathtt{,}\mathtt{P}_2} \tag{EUP-Cat}$$

$$\frac{\mathtt{v} \in^u \mathtt{P}_1 \qquad \mathtt{v} \notin \mathit{tyof}(\mathtt{P}_2)}{\mathtt{v} \in^u \mathtt{P}_1 \,|\, \mathtt{P}_2} \tag{EUP-Or1}$$

$$\frac{\mathtt{v} \notin \mathit{tyof}(\mathtt{P}_1) \qquad \mathtt{v} \in^u \mathtt{P}_2}{\mathtt{v} \in^u \mathtt{P}_1 \,|\, \mathtt{P}_2} \tag{EUP-Or2}$$

$$\frac{\begin{array}{c}\text{there are unique } \mathtt{v}_1, \ldots, \mathtt{v}_n \text{ s.t. } \mathtt{v} = \mathtt{v}_1\,\mathtt{,}\ldots\mathtt{,}\,\mathtt{v}_n \\ \mathtt{v}_i \in^u \mathtt{P} \quad \text{for each } i\end{array}}{\mathtt{v} \in^u \mathtt{P*}} \tag{EUP-Rep}$$

That is, these rules are similar to those for the matching relation (without environments) except that EUP-Cat and EUP-Rep ensure that the input sequence can be split uniquely at concatenation and repetition patterns, and that EUP-Or1 and EUP-Or2 ensure that the input can be matched exactly one of the choices.

Now, a pattern P is *non-ambiguous* with respect to a type T, written "$\mathtt{T} \triangleright \mathtt{P}$: *nonambiguous*," iff, for all $\mathtt{v} \in \mathtt{T}$ and $\mathtt{v} \in \mathit{tyof}(\mathtt{P})$, we have $\mathtt{v} \in^u \mathtt{P}$.

This definition of non-ambiguity is similar to *strong non-ambiguity* for string regular expressions [Sippu and Soisalon-Soininen 1988] except that we treat sequences of trees rather than strings, and that we consider non-ambiguity for a given restricted set of input values rather than for all input values.[11] Sippu and Soisalon-Soininen reduce the non-ambiguity problem for regular expressions to the LR(0) property for context-free grammars. We use a more direct algorithm based on product construction [Hosoya 2003]. (Discussions on various kinds of ambiguity for regular expressions and the relationship among them can be found in [Brüggemann-Klein 1993].)

5.4.7 *Pattern type inference.* The goal of pattern type inference is to compute the "range" of a pattern, defined as follows. A type environment Γ describes the range of a pattern P with respect to type T, written "$\mathtt{T} \triangleright \mathtt{P} \Rightarrow \Gamma$," iff, for all x and v, we have

> $\mathtt{v} \in \Gamma(\mathtt{x})$ iff there exists a value $\mathtt{u} \in \mathtt{T}$ such that $\mathtt{u} \in \mathtt{P} \Rightarrow \mathtt{V}$ for some V with $\mathtt{V}(\mathtt{x}) = \mathtt{v}$.

5.5 Term language

A *program* comprises a set of type definitions, a set of pattern definitions, a set of function definitions, and a term with which evaluation starts. Type and pattern definitions were described in the previous section. This section introduces functions and terms.

5.5.1 *Syntax.* We assume given a countably infinite set of function names, ranged over by f. The definitions of functions are given by a fixed, global, mutually recursive set G of function definitions of the following form.

```
fun f(P):T = e
```

[11] Since the design space for non-ambiguity is rather large and we have not yet explored it suffiently, we give here a tentative simple specification. This may change in a future version of XDuce.

(For brevity, we treat only one-argument functions here; the extension to multi-argument functions is straightforward.) Note that both the argument pattern (which provides the names and types of the bound variables) and the result type are given explicitly.

The syntax of terms, $\mathtt{e}$, is defined by the following grammar.

`e`	::=	`x`	variable
		`l[e]`	label
		`()`	empty sequence
		`e,e`	concatenation
		`f(e)`	application
		$\mathtt{match\ e\ with\ }\overline{\mathtt{P}}\mathtt{->}\overline{\mathtt{e}}$	pattern match

We write $\overline{\mathtt{P}}\mathtt{->}\overline{\mathtt{e}}$ as an abbreviation for the n-ary form $\mathtt{P}_1\mathtt{->e}_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \mathtt{P}_n\mathtt{->e}_n$. We also allow the following shorthands:

$$\begin{aligned}
\mathtt{let\ P=e}_1\mathtt{\ in\ e}_2 &\equiv \mathtt{match\ e}_1\mathtt{\ with\ P->e}_2\\
\mathtt{if\ e}_1\mathtt{\ then\ e}_2\mathtt{\ else\ e}_3 &\equiv \mathtt{match\ e}_1\mathtt{\ with\ True[]->e}_2\ \mathtt{|}\ \mathtt{False[]->\ e}_3\\
\mathtt{e}_1\mathtt{;e}_2 &\equiv \mathtt{let\ Any=e}_1\mathtt{\ in\ e}_2
\end{aligned}$$

For simplicity, we assume that the variables bound by patterns are all distinct. (Of course, we can always α-convert an arbitrary program so as to satisfy this condition.)

5.5.2 *Typing rules.* The typing relation $\Gamma \vdash \mathtt{e} \in \mathtt{T}$, pronounced "$\mathtt{e}$ has type $\mathtt{T}$ under environment Γ," is defined by the following rules.

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

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

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

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

$$\frac{\mathtt{fun\ f(P):T\ =\ e}_2 \in G \qquad \Gamma \vdash \mathtt{e}_1 \in \mathtt{U} \qquad \mathtt{U} <: \mathit{tyof}(\mathtt{P})}{\Gamma \vdash \mathtt{f(e}_1\mathtt{)} \in \mathtt{T}} \qquad \text{(TE-APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} \in \mathtt{R}\\ \mathtt{R} \triangleright \mathtt{P}_1,\ldots,\mathtt{P}_n : \mathit{exhaustive}\\ \mathtt{R} \triangleright \mathtt{P}_1,\ldots,\mathtt{P}_n : \mathit{irredundant}\\ \forall i.\ \left(\begin{array}{l}\mathtt{R} \setminus (\mathit{tyof}(\mathtt{P}_1)\mathtt{|}\ldots\mathtt{|}\mathit{tyof}(\mathtt{P}_{i-1})) \Rightarrow \mathtt{S}\\ \mathtt{S} \triangleright \mathtt{P}_i : \mathit{nonambiguous}\\ \mathtt{S} \triangleright \mathtt{P}_i \Rightarrow \Gamma_i\\ \Gamma,\Gamma_i \vdash \mathtt{e}_i \in \mathtt{T}_i\end{array}\right)\end{array}}{\Gamma \vdash \mathtt{match\ e\ with\ }\overline{\mathtt{P}}\mathtt{->}\overline{\mathtt{e}} \in \mathtt{T}_1\mathtt{|..|T}_n} \qquad \text{(TE-MATCH)}$$

As we discussed in Section 4.3, when performing type inference for a clause $\mathtt{P}_i$, we use the difference operation to exclude the values matched by the preceding patterns from the input type `R`. We also check the ambiguity of each clause with respect to the same difference type—i.e., our definition of ambiguity does not consider values that the pattern will never be used to match.

We then have a single rule for judging when the definition of a function `f` is well typed, written $\vdash$ `fun f(P):T = e`.

$$\frac{\begin{array}{c} \mathit{tyof}(\mathtt{P}) \triangleright \mathtt{P} \Rightarrow \Gamma \\ \Gamma \vdash \mathtt{e} \in \mathtt{S} \qquad \mathtt{S} <: \mathtt{T} \end{array}}{\vdash \mathtt{fun\ f(P):T\ =\ e}} \tag{TF}$$

5.5.3 *Evaluation rules.* The semantics of terms is defined by a "big step" evaluation relation $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v}$. The rules for the evaluation relation are all standard; the only interesting case is the rule for pattern matching which uses the semantics of patterns defined above.

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

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

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

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

$$\frac{\begin{array}{c} \mathtt{V} \vdash \mathtt{e}_1 \Downarrow \mathtt{v} \\ \mathtt{fun\ f(P):T\ =\ e}_2 \in G \\ \mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{W} \qquad \mathtt{W} \vdash \mathtt{e}_2 \Downarrow \mathtt{w} \end{array}}{\mathtt{V} \vdash \mathtt{f(e}_1\mathtt{)} \Downarrow \mathtt{w}} \tag{EE-App}$$

$$\frac{\begin{array}{c} \mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v} \\ \mathtt{v} \notin \mathtt{P}_1 \quad \ldots \quad \mathtt{v} \notin \mathtt{P}_{i-1} \qquad \mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W} \\ \mathtt{V,W} \vdash \mathtt{e}_i \Downarrow \mathtt{w} \end{array}}{\mathtt{V} \vdash \mathtt{match\ e\ with\ }\overline{\mathtt{P}}\mathtt{->}\overline{\mathtt{e}} \Downarrow \mathtt{w}} \tag{EE-Match}$$

Note that because of the assumption that all bound variables are distinct, all concatenated environments are ensured to have different domains.

5.6 Type soundness

We conclude this treatment of core XDuce by sketching a proof of type soundness. As usual, there are two parts to the proof: subject reduction (a well-typed term evaluates to a value inhabiting the expected type) and progress (a well-typed term does not get stuck).

5.1 Theorem [Subject Reduction]: Suppose $\vdash$ `fun f(P):T = e` for all function definitions in G. If $\emptyset \vdash \mathtt{e} \Downarrow \mathtt{w}$ and $\emptyset \vdash \mathtt{e} \in \mathtt{T}$, then $\mathtt{w} \in \mathtt{T}$.

PROOF. We prove the result by showing the following stronger statement:

If $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{w}$ and $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ with $\Gamma \vdash \mathtt{V}$, then $\mathtt{w} \in \mathtt{T}$.

Here, $\Gamma \vdash \mathtt{V}$ means that $dom(\Gamma) = dom(\mathtt{V})$ and $\mathtt{V}(\mathtt{x}) \in \Gamma(\mathtt{x})$ for each $\mathtt{x} \in dom(\Gamma)$. The proof proceeds by induction on the derivation on $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{w}$. We show only the interesting cases for function application and pattern matching. All the other cases can be proved by a straightforward use of the induction hypothesis.

Case: $\mathtt{e} = \mathtt{f(e_1)}$ $\quad \mathtt{V} \vdash \mathtt{e_1} \Downarrow \mathtt{v}$ $\quad$ `fun f(P):T = `$\mathtt{e_2} \in G$
$\mathtt{w} \in \mathtt{P} \Rightarrow \mathtt{W}$ $\quad \mathtt{W} \vdash \mathtt{e_2} \Downarrow \mathtt{w}$

Since $\Gamma \vdash \mathtt{e_1} \in \mathtt{U}$ by TE-APP, we obtain $\mathtt{v} \in \mathtt{U}$ by the induction hypothesis. Further, since $\mathtt{U} <: tyof(\mathtt{P})$ by TE-APP, we have $\mathtt{v} \in tyof(\mathtt{P})$ by the definition of subtyping. From $tyof(\mathtt{P}) \triangleright \mathtt{P} \Rightarrow \Gamma'$ by TF and the definition of type inference, $\Gamma' \vdash \mathtt{W}$. Finally, since $\Gamma' \vdash \mathtt{e_2} \in \mathtt{S}$ and $\mathtt{S} <: \mathtt{T}$ by TF, the use of the induction hypothesis and subtyping yields $\mathtt{w} \in \mathtt{T}$.

Case: $\mathtt{e} = \mathtt{match}\ \mathtt{e'}\ \mathtt{with}\ \overline{\mathtt{P}}\text{->}\overline{\mathtt{e}}$ $\quad \mathtt{V} \vdash \mathtt{e'} \Downarrow \mathtt{v}$
$\mathtt{v} \notin \mathtt{P_1} \quad \ldots \quad \mathtt{v} \notin \mathtt{P}_{i-1} \quad \mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W} \quad \mathtt{V},\mathtt{W} \vdash \mathtt{e}_i \Downarrow \mathtt{w}$

Since $\Gamma \vdash \mathtt{e'} \in \mathtt{R}$ by TE-MATCH, $\mathtt{v} \in \mathtt{R}$ by the induction hypothesis. Also, since $\mathtt{R} \setminus (tyof(\mathtt{P_1}) \mid \ldots \mid tyof(\mathtt{P}_{i-1})) \Rightarrow \mathtt{S}$ by TE-MATCH, the definition of difference implies $\mathtt{v} \in \mathtt{S}$. Further, $\mathtt{S} \triangleright \mathtt{P}_i \Rightarrow \Gamma_i$ by TE-MATCH, the definition of type inference yields $\Gamma_i \vdash \mathtt{W}$ and therefore $\Gamma, \Gamma_i \vdash \mathtt{V},\mathtt{W}$. Finally, since TE-MATCH gives $\Gamma, \Gamma_i \vdash \mathtt{e}_i \in \mathtt{T}_i$, we obtain $\mathtt{w} \in \mathtt{T}_i$ by the induction hypothesis. The result follows since $\mathtt{T}_i <: \mathtt{T_1} \mid \ldots \mid \mathtt{T}_n$. □

Since we have chosen a big-step semantics, we need to be a little careful about what it means for a term to get stuck. Naively, we might simply say "e is stuck if it is not the case that $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v}$ for any v." But this amounts to saying that e is stuck if there is no finite derivation of $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v}$, which is not quite what we want: a finite derivation may fail to exist either because e gets stuck or because it diverges. To precisely capture the notion that "e gets stuck in a finite number of steps," we define the *stuck evaluation* relation $\mathtt{V} \vdash \mathtt{e} \not\Downarrow$, inductively, as follows.

—$\mathtt{V} \vdash \mathtt{l[e]} \not\Downarrow$ if $\mathtt{V} \vdash \mathtt{e} \not\Downarrow$.

—$\mathtt{V} \vdash \mathtt{e_1,e_2} \not\Downarrow$ if either $\mathtt{V} \vdash \mathtt{e_1} \not\Downarrow$ or $\mathtt{V} \vdash \mathtt{e_2} \not\Downarrow$.

—$\mathtt{V} \vdash \mathtt{f(e_1)} \not\Downarrow$ if
  (1) $\mathtt{V} \vdash \mathtt{e_1} \not\Downarrow$, or
  (2) $\mathtt{V} \vdash \mathtt{e_1} \Downarrow \mathtt{v}$ and $\mathtt{v} \notin \mathtt{P}$, where `fun f(P):T = `$\mathtt{e_2} \in G$, or
  (3) $\mathtt{V} \vdash \mathtt{e_1} \Downarrow \mathtt{v}$ and $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{W}$ and $\mathtt{W} \vdash \mathtt{e_2} \not\Downarrow$, where `fun f(P):T = `$\mathtt{e_2} \in G$.

—$\mathtt{V} \vdash \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \overline{\mathtt{P}}\text{->}\overline{\mathtt{e}} \not\Downarrow$ if
  (1) $\mathtt{V} \vdash \mathtt{e} \not\Downarrow$, or
  (2) $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v}$ and $\mathtt{v} \in \mathtt{P}_i \not\Rightarrow \mathtt{W}$ for all i, or
  (3) $\mathtt{V} \vdash \mathtt{e} \Downarrow \mathtt{v}$ and for some i,

$$\mathtt{v} \notin \mathtt{P_1} \quad \ldots \quad \mathtt{v} \notin \mathtt{P}_{i-1} \qquad \mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W} \qquad \mathtt{V},\mathtt{W} \vdash \mathtt{e}_i \not\Downarrow.$$

Notice that the base cases are the second case for function applications and the second case for the `match` expression, where the input value does not match the pattern.

5.2 Theorem [Progress]: Suppose $\vdash$ `fun f(P):T = e` for all function definitions in G. Then $\emptyset \vdash \mathtt{e} \in \mathtt{T}$ implies not $\emptyset \vdash \mathtt{e} \not\Downarrow$.

PROOF. We obtain the result by proving the following stronger statement:

If $\mathtt{V} \vdash \mathtt{e} \not\Downarrow$, then there are no Γ and $\mathtt{T}$ such that $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Gamma \vdash \mathtt{V}$.

The proof of this statement goes by induction on the given derivation of $\mathtt{V} \vdash \mathtt{e} \not\Downarrow$. We show just the interesting cases; the rest proceed by straightforward use of the induction hypothesis.

Case: $\mathtt{e} = \mathtt{f(e_1)}$ $\quad$ $\mathtt{fun\ f(P):T' = e_2} \in G$
$\mathtt{V} \vdash \mathtt{e_1} \Downarrow \mathtt{v}$ $\quad$ $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{W}$ $\quad$ $\mathtt{W} \vdash \mathtt{e_2} \not\Downarrow$

Suppose, for a contradiction, that $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Gamma \vdash \mathtt{V}$ for some Γ, $\mathtt{T}$. Then for some $\mathtt{S}$ we have $\Gamma \vdash \mathtt{e_1} \in \mathtt{S}$ and $\mathtt{S} <: \mathit{tyof}(\mathtt{P})$, by TE-APP. By subject reduction, we have $\mathtt{v} \in \mathtt{S}$ and therefore $\mathtt{v} \in \mathit{tyof}(\mathtt{P})$. In addition, we have, by assumption, $\vdash \mathtt{fun\ f(P):T' = e_2}$, which implies $\mathit{tyof}(\mathtt{P}) \triangleright \mathtt{P} \Rightarrow \Gamma'$ and $\Gamma' \vdash \mathtt{e_2} \in \mathtt{T}'$. The former together with $\mathtt{v} \in \mathit{tyof}(\mathtt{P})$ and $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{W}$ implies $\Gamma' \vdash \mathtt{W}$. But, by the induction hypothesis, there are no Γ'' and $\mathtt{T}''$ such that $\Gamma'' \vdash \mathtt{e_2} \in \mathtt{T}''$ and $\Gamma'' \vdash \mathtt{W}$—a contradiction.

Case: $\mathtt{e} = \mathtt{match\ e'\ with\ \overline{P}\text{->}\overline{e}}$ $\quad$ $\mathtt{V} \vdash \mathtt{e'} \Downarrow \mathtt{v}$ $\quad$ $\forall i.\ \mathtt{v} \notin \mathtt{P}_i$

Suppose that $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Gamma \vdash \mathtt{V}$ for some Γ, $\mathtt{T}$. Then, from TE-MATCH, $\Gamma \vdash \mathtt{e'} \in \mathtt{R}$ and $\mathtt{R} \triangleright \mathtt{P}_1, \ldots, \mathtt{P}_n : \mathit{exhaustive}$. By subject reduction, $\mathtt{v} \in \mathtt{R}$. Together with the definition of exhaustiveness, this implies that $\mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W}$ for some i and $\mathtt{W}$, which contradicts the assumption.

Case: $\mathtt{e} = \mathtt{match\ e'\ with\ \overline{P}\text{->}\overline{e}}$ $\quad$ $\mathtt{V} \vdash \mathtt{e'} \Downarrow \mathtt{v}$
$\mathtt{v} \notin \mathtt{P}_1$ $\ \ldots\ $ $\mathtt{v} \notin \mathtt{P}_{i-1}$ $\quad$ $\mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W}$ $\quad$ $\mathtt{V}, \mathtt{W} \vdash \mathtt{e}_i \not\Downarrow$

Suppose that $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Gamma \vdash \mathtt{V}$ for some Γ, $\mathtt{T}$. Then, from TE-MATCH, $\Gamma \vdash \mathtt{e'} \in \mathtt{R}$. By subject reduction, $\mathtt{v} \in \mathtt{R}$. Since $\mathtt{v} \notin \mathtt{P}_j$ for $j = 1, \ldots, i-1$, we have $\mathtt{v} \in \mathtt{R} \setminus \mathit{tyof}(\mathtt{P}_1) \setminus \ldots \setminus \mathit{tyof}(\mathtt{P}_{i-1})$. TE-MATCH also tells us that $(\mathtt{R} \setminus \mathit{tyof}(\mathtt{P}_1) \setminus \ldots \setminus \mathit{tyof}(\mathtt{P}_{i-1})) \triangleright \mathtt{P}_i \Rightarrow \Gamma_i$. Together with $\mathtt{v} \in \mathtt{P}_i \Rightarrow \mathtt{W}$, we obtain $\Gamma' \vdash \mathtt{W}$; hence, $\Gamma, \Gamma' \vdash \mathtt{V}, \mathtt{W}$. Furthermore, from TE-MATCH, we have $\Gamma, \Gamma' \vdash \mathtt{e}_i \in \mathtt{T}$. But, by the induction hypothesis, there are no Γ'' and $\mathtt{T}''$ such that $\Gamma'' \vdash \mathtt{e}_i \in \mathtt{T}''$ and $\Gamma'' \vdash \mathtt{V}, \mathtt{W}$—a contradiction. □

6. RELATED WORK

Static typing of programs for XML processing has been approached from several different angles. One popular idea is to "embed" some schema language for XML into an existing typed language, translating document types into class hierarchies (in an object-oriented language) or algebraic datatypes (in a functional language). Examples of this approach include HaXML [Wallace and Runciman 1999], Relaxer [Asami 2000], and JAXB [Sun Microsystems 2001]. (There are a large number of software products implementing similar ideas; a comprehensive list can be found at [Bourret 2001].) The advantage of this approach is that it can be carried out entirely within an existing language. The cost is that XML values and their corresponding schemas must somehow be "injected" into the value and type spaces of the host language; this usually involves adding layers of "tagging" that were not present in the original XML documents, which inhibits subtyping and makes programming somewhat less flexible.

The XML processing language XMλ designed by Meijer and Shields has basically followed this approach but made a major improvement in flexibility by introducing *type-indexed rows* [Meijer and Shields 1999; Shields and Meijer 2001]. In their type system, a union type `T|U` in a DTD is represented by a sum type where each summand is tagged by its type (`T` or `U`) itself (whereas most systems in the mapping approach uses a fixed label determined by the order that `T` or `U` appears or by some name mangling based on the top-level tags of `T` or `U`). Thus, they can allow commutativity of union, as we do in XDuce. However, this mechanism does not validate other useful laws such as associativity nor distributivity (which XDuce does). On the other hand, they achieve a different direction of flexibility by row polymorphism. For example, they can write a polymorphic function of type $\forall \mathtt{X} \notin \{\mathtt{T},\mathtt{U}\}.\ (\mathtt{T|X}) \rightarrow (\mathtt{U|X})$ (e.g., a function that changes some specific labels but leaves the other), which conveys the typing constraint that the type unioned with `T` in the input is exactly the same as the type unioned with `U` in the output. Note that such typing constraints cannot be represented only by subtyping.

The query language YAT [Cluet and Siméon 1998; Kuper and Siméon 2001] includes a type system similar to regular expression types. Like XDuce, YAT offers a notion of subtyping for flexibility. However, they adopt a somewhat more restrictive subtyping relation, e.g., they do not allow `a[T|U]` to be a subtype of `a[T]|a[U]` (but do allow the other way). This choice was determined by their design goal of attaining efficient layout for large XML databases.

Since its initial publication, our work on XDuce has influenced proposals by other researchers. Fernandez, Siméon, and Wadler propose XML Query Algebra for the basis of XML query processing and optimization, and they use our regular expression types in their type system and our subtyping algorithm in their implementation [Fernández et al. 2001]. Frisch, Castagna, and Benzaken have made a significant extension to XDuce and yielded another language ℂDuce for XML processing [Frisch et al. 2002]. In particular, their type system treats higher-order functions as well as intersection and complementation type operators. Our work on regular expression types has also stimulated schema language designs. In particular, Clark's schema language TREX [Clark 2001] adopts a large part of our definition of types (and is now merged to an ISO standard RELAX NG [Clark and Murata 2001]).

At the theoretical level, there have been a number of proposals of typechecking algorithms for various XML processing languages. Milo, Suciu, and Vianu have studied a typechecking problem for a general framework called *k-pebble tree transducers*, which can capture a wide range of query languages for XML [Milo et al. 2000]. In a related vein, Papakonstantinou and Vianu present a typechecking algorithm for the query language *loto-ql* by using extensions to DTDs [Papakonstantinou and Vianu 2000]. Murata has developed a typechecking algorithm for his document transformation language with powerful pattern matching [Murata 1997]. Tozawa has pursued a typechecking technique for a subset of XSLT [Tozawa 2001].

Types used in the techniques in these papers are based on tree automata and are conceptually identical to those of XDuce. On the other hand, the type checking algorithms presented in these papers are "semantically complete" (that is, given a program, an input type, and an output type, the algorithm returns "yes" *exactly when* the documents produced by the program from the input type always have the output type), while XDuce's is not (since XDuce is Turing complete, this level of precision makes the problem undecidable).

Our investigation of regular expression types was originally motivated by an observation by Buneman and Pierce [Buneman and Pierce 1998] that untagged union types correspond naturally to forms of variation found in semistructured databases. The main differences from the present work are that they study unordered record types instead of ordered sequences and do not treat recursive types.

Pattern matching can be found in a wide variety of languages and in a variety of styles. One axis for categorization is how many bindings a pattern match yields. In the *all-matches* style, a pattern match yields a *set* of bindings corresponding to all possible matches. This style is often used in query languages [Deutsch et al. 1998; Abiteboul et al. 1997; Cluet and Siméon 1998; Cardelli and Ghelli 2000; Neven and Schwentick 2000; Fankhauser et al. 2001] and document processing languages [Clark 1999; Neumann and Seidl 1998; Murata 1997]. In the *single-match* style, a successful match yields just one binding. This style is the one commonly found in functional programming languages [Milner et al. 1990; Leroy et al. 1996; Peyton Jones et al. 1993, etc.], and is the one we have followed in XDuce. We are still experimenting with this aspect of the language, however, and hope to incorporate some form of all-match patterns in its successor, Xtatic.

Another axis for comparing pattern matching primitives is the expressiveness of the underlying "logic." Several papers have proposed extension of ML-like pattern matching with recursion [Fähndrich and Boyland 1997; Queinnec 1990] with essentially the same expressiveness as ours. Some query languages and document processing languages use pattern matching mechanisms based on tree automata [Neumann and Seidl 1998; Murata 1997] or monadic second-order logic (which has a strong connection to tree automata) [Neven and Schwentick 2000], and therefore they have a similar expressiveness to our pattern matching. TQL [Cardelli and Ghelli 2000] has a powerful pattern matching facility based on Ambient Logic [Cardelli and Gordon 2000]. Since Ambient Logic allows arbitrary logical operators (union, intersection, and complementation), this suggests that expressiveness can be similar to tree automata. However, exact comparison is not trivial since their underlying data model is unordered trees. On the other hand, pattern matching based on *regular path expressions*, popular in query languages for semistructured data [Deutsch et al. 1998; Abiteboul et al. 1997; Cluet and Siméon 1998], is less expressive than tree automata. For instance, these patterns cannot express patterns like "subtrees that contain *exactly* these labels." Both tree automata and regular path expressions can express extraction of data from an arbitrarily nested tree structure (although, with the single-match style, the usefulness of such deep matching is questionable; a related discussion can be found in our previous paper on pattern matching [Hosoya and Pierce 2001]).

Thiemann has proposed a technique to encode DTDs by Haskell's type classes and thereby statically ensure the validity of dynamically generated XML documents [Thiemann 2001]. His technique is implemented as a pure Haskell library, requiring no language extension. On the other hand, his proposal is limited to generating documents and provides no facility to deconstruct or analyze input XML documents.

Brabrand, Møller, and Schwartzbach have designed a domain-specific language `<bigwig>` for programming interactive Web services [Brabrand et al. 2002]. As a counterpart, they employ an interprocedural flow analysis for statically validating HTML documents produced by programs [Brabrand et al. 2001]. (Although their specific interest there is HTML, their framework is general enough to handle most of general XML DTDs.) Their analysis is, unlike ours, capable of checking programs with no type annotations. On the other hand, their language has no support for processing input documents.

7. CONCLUSIONS

XDuce is a typed programming language that takes XML documents (sequences of nodes) as primitive values. It provides constructors and deconstructors (pattern matching) for such sequences and uses regular expression types for describing their structure statically. The correspondence between types and finite tree automata gives the language a powerful mathematical foundation, leading to a simple, clean, and flexible design.

We regard XDuce as a good first step in the direction of "native programming language support" for XML. However, a number of significant issues remain to be considered before its innovations can be offered to mainstream XML programmers.

First, the XDuce type system needs to be extended to handle common features found in real-world schema languages. One is a support for XML attributes. Among different styles of treatments, we adopt RELAX NG's approach [Clark and Murata 2001], which symmetrically handles constraints on elements and attributes. Murata and the first author of the present paper have just figured out the imposed algorithmic issues [Hosoya and Murata 2002]. Another issue that has to be addressed is typing for unordered data, which are useful for representing records, for example. A clean solution is to introduce so-called shuffle (or interleave) operator as in RELAX NG. However, it is an open question whether we can test inclusion or compute intersection or difference with a reasonable efficiency.

Second, in writing programs more substantial than trivial XML transformations, we almost always need other kinds of data structures than sequences, such as hash tables and arrays. For this, we are now pursuing the direction of taking some existing popular programming language (such as Java or C#) and mixing its type system with regular expression types. In this way, we can avoid re-inventing existing language features and libraries, as well as easily invite people who have been working on XML with such languages.

Third, with such a mixed type system, we will need several more advanced typing features. One is parametric polymorphism. (Currently, neither Java or C# supports this, but since they are planning to do so, we will have to figure out how to deal with it.) Another is typing for imperative operations on XML data. Currently, XDuce disallows modification of values and this might be too rigid for many programmers. Technically, both parametric polymorphism and destructive operations are non-trivial.

We are now working on the design of a successor to XDuce, named Xtatic, which aims to address these issues in the context of a C# extension with regular expression types and pattern matching.

APPENDIX

REFERENCES

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

Asami, T. 2000. Relaxer. `http://www.asahi-net.or.jp/~dp8t-asm/java/tools/Relaxer/index.html`.

Bourret, R. 2001. XML data binding resources. `http://www.rpbourret.com/xml/XMLDataBinding.htm`.

Brabrand, C., Møller, A., and Schwartzbach, M. I. 2001. Static validation of dynamically generated HTML. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'01).*

Brabrand, C., Møller, A., and Schwartzbach, M. I. 2002. The `<bigwig>` project. *ACM Transactions on Internet Technology (TOIT).*

Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. 2000. Extensible markup language (XML^{TM}). `http://www.w3.org/XML/`.

Brüggemann-Klein, A. 1993. Regular expressions into finite automata. *Theoretical Computer Science 120*, 197–213.

Buneman, P. and Pierce, B. 1998. Union types for semistructured data. In *Internet Programming Languages.* Lecture Notes in Computer Science, vol. 1686. Springer-Verlag. Proceedings of the International Database Programming Languages Workshop.

Cardelli, L. and Ghelli, G. 2000. A query language for semistructured data based on the Ambient Logic. Manuscript.

Cardelli, L. and Gordon, A. D. 2000. Anytime, anywhere. Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages.* 365–377.

Clark, J. 1999. XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`.

Clark, J. 2001. TREX: Tree Regular Expressions for XML. `http://www.thaiopensource.com/trex/`.

Clark, J. and Murata, M. 2001. RELAX NG. `http://www.relaxng.org`.

Cluet, S. and Siméon, J. 1998. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB).*

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. 1999. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. 1998. XML-QL: A Query Language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

Fähndrich, M. and Boyland, J. 1997. Statically checkable pattern abstractions. In *Proceedings of the International Conference on Functional Programming (ICFP).* 75–84.

Fallside, D. C. 2001. XML Schema Part 0: Primer, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`.

Fankhauser, P., Fernández, M., Malhotra, A., Rys, M., Siméon, J., and Wadler, P. 2001. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`.

Fernández, M. F., Siméon, J., and Wadler, P. 2001. A semi-monad for semi-structured data. In *Proceedings of 8th International Conference on Database Theory (ICDT 2001)*, J. V. den Bussche and V. Vianu, Eds. Lecture Notes in Computer Science, vol. 1973. Springer, 263–300.

Frisch, A., Castagna, G., and Benzaken, V. 2002. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science.*

HOSOYA, H. 2003. Regular expression pattern matching — a simpler design. Tech. rep., RIMS, Kyoto University. To appear.

HOSOYA, H. AND MURATA, M. 2002. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*. 1–10.

HOSOYA, H. AND PIERCE, B. C. 2000. XDuce: A typed XML processing language (preliminary report). In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*. Lecture Notes in Computer Science, vol. 1997. 226–244.

HOSOYA, H. AND PIERCE, B. C. 2001. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 67–80.

HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. 2000. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*. 11–22. Full version under submission to TOPLAS.

KLARLUND, N., MØLLER, A., AND SCHWARTZBACH, M. I. 2000. DSD: A schema language for XML. `http://www.brics.dk/DSD/`.

KUPER, G. M. AND SIMÉON, J. 2001. Subsumption for XML types. In *International Conference on Database Theory (ICDT'2001)*. London.

LEROY, X., VOUILLON, J., DOLIGEZ, D., ET AL. 1996. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`.

MEIJER, E. AND SHIELDS, M. 1999. XMλ: A functional programming language for constructing and manipulating XML documents. Submitted to USENIX 2000 Technical Conference.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. The MIT Press.

MILO, T., SUCIU, D., AND VIANU, V. 2000. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 11–22.

MURATA, M. 1997. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*. Lecture Notes in Computer Science, vol. 1293. Springer-Verlag, 153–169.

NEUMANN, A. AND SEIDL, H. 1998. Locating matches of tree patterns in forests. In *18th FSTTCS*. LNCS, vol. 1530. 134–145.

NEVEN, F. AND SCHWENTICK, T. 2000. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 145–156.

PAPAKONSTANTINOU, Y. AND VIANU, V. 2000. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. Dallas, Texas, 35–46.

PEYTON JONES, S. L., HALL, C. V., HAMMOND, K., PARTAIN, W., AND WADLER, P. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*.

PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press.

QUEINNEC, C. 1990. Compilation of non-linear, second order patterns on s-expressions. In *Programming Language Implementation and Logic Programming, 2nd International Workshop (PLILP'90)*. Lecture Notes in Computer Science. Springer-Verlag, 340–357.

SEIDL, H. 1990. Deciding equivalence of finite tree automata. *SIAM Journal of Computing 19,* 3 (June), 424–437.

SHIELDS, M. AND MEIJER, E. 2001. Type-indexed rows. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. London.

SIPPU, S. AND SOISALON-SOININEN, E. 1988. Parsing theory. In *EATCS Monographs on Theoretical Computer Science*. Vol. 1. Springer.

SUN MICROSYSTEMS, I. 2001. The Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb`.

THIEMANN, P. 2001. A typed representation for html and xml documents in haskell. To appear in Journal of Functional Programming.

Tozawa, A. 2001. Towards static type inference for XSLT. In *Proceedings of ACM Symposium on Document Engineering*.

Wallace, M. and Runciman, C. 1999. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*. ACM Sigplan Notices, vol. 34-9. ACM Press, N.Y., 148–159.

ACKNOWLEDGMENTS

Our main collaborator in the XDuce project, Jérôme Vouillon, contributed a number of ideas, both in the design presented here and in the XDuce implementation. We are also grateful to the other XDuce team members (Peter Buneman, Vladimir Gapayev, Michael Levin, and Phil Wadler). We have learned a great deal from discussions 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 Japan Society for the Promotion of Science (Hosoya), the National Science Foundation under NSF Career grant CCR-9701826 (Pierce), and a gift from Microsoft.