Regular Expression Types for XML

HARUO HOSOYA
Kyoto University
hahosoya@kurims.kyoto-u.ac.jp
JÉRÔME VOUILLON
CNRS and Denis Diderot University
Jerome.Vouillon@pps.jussieu.fr
and
BENJAMIN C. PIERCE
University of Pennsylvania
bcpierce@cis.upenn.edu

We propose *regular expression types* as a foundation for statically typed XML processing languages. Regular expression types, like most schema languages for XML, introduce regular expression notations such as repetition (`*`), alternation (`|`), etc., to describe XML documents. The novelty of our type system is a semantic presentation of subtyping, as inclusion between the sets of documents denoted by two types. We give several examples illustrating the usefulness of this form of subtyping in XML processing.

The decision problem for the subtype relation reduces to the inclusion problem between tree automata, which is known to be EXPTIME-complete. To avoid this high complexity in typical cases, we develop a practical algorithm that, unlike classical algorithms based on determinization of tree automata, checks the inclusion relation by a top-down traversal of the original type expressions. The main advantage of this algorithm is that it can exploit the property that type expressions being compared often share portions of their representations. Our algorithm is a variant of Aiken and Murphy's set-inclusion constraint solver, to which are added several new implementation techniques, correctness proofs, and preliminary performance measurements on some small programs in the XML domain.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms: Languages, Theory

Additional Key Words and Phrases: Type systems, XML, subtyping

Authors' address: H. Hosoya, Research Institute for Mathematical Sciences, Kyoto University Oiwake-cho, Kitashirakawa, Sakyo-ku, Kyoto 606–8502, Japan. J. Vouillon, PPS, Université Denis Diderot, Case 7014, 2 Place Jussieu, F-75251 PARIS Cedex 05, France. B. C. Pierce, Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd St., Philadelphia, PA 19104, USA.

1. INTRODUCTION

XML [Bray et al. 2000] is an emerging standard format for tree-structured data. One of the reasons for its popularity is the existence of a number of schema languages, including DTDs [Bray et al. 2000], XML-Schema [Fallside 2001], DSD [Klarlund et al. 2000], and RELAX [Murata 2001], that can be used to define "types" (or "schemas") describing structural constraints on data and thereby improve the safety of data processing and exchange.

However, the use of types in mainstream XML processing technology is often limited to checking only *data*, not *programs*. Typically, an XML processing program first reads an XML document and checks that it conforms to a given type using a *validating parser*. The program then uses either a generic tree manipulation library such as DOM [DOM 2001] or a dedicated XML language such as XSLT [Clark 1999] or XML-QL [Deutsch et al. 1998]. Since these tools make no systematic connection between the program and the types of the documents it manipulates, they provide no compile-time guarantee that the documents produced by the program will always conform to an intended type.

In this article, we propose *regular expression types* as a foundation for statically typed processing of XML documents. Regular expression types capture (and generalize) the regular expression notations (`*`, `?`, `|`, etc.) commonly found in schema languages for XML, and support a natural semantic notion of subtyping.

We have used regular expression types in the design of a domain-specific language called XDuce ("transduce") for XML processing [Hosoya and Pierce 2000; 2001]. In the present article, however, our focus is on the structure of the types themselves, their role in describing transformations on XML documents, and the algorithmic problems they pose. Interested readers are invited to visit the XDuce home page

`http://xduce.sourceforge.net`

for more information on the language as a whole.

As a simple example of regular expression types, consider the definitions

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

corresponding to the following set of DTD declarations:

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

Type constructors of the form `label[...]` classify tree nodes with the tag `label` (i.e., XML structures of the form `<label>...</label>`). Thus, the inhabitants of the types `Name`, `Email`, and `Tel` are all strings with an appropriate identifying label. Types may also involve the regular expression operators `*` (repetition) and `?` (optional occurrence), as well as `|` (alternation). Thus, the type `Addrbook` describes a label `addrbook` whose content is zero or more repetitions of subtrees of type `Person`. Likewise, the type `Person` describes a label `person` whose content is a `Name` subtree, zero or more `Email` subtrees, and an optional `Tel` subtree. An instance of the type `Addrbook` is the following XML document:

```
<addrbook>
  <person> <name> Haruo Hosoya </name>
           <email> hahosoya@upenn </email>
           <email> haruo@u-tokyo </email> </person>
  <person> <name> Jerome Vouillon </name>
           <email> vouillon@upenn </email>
           <tel> 123-456-789 </tel> </person>
</addrbook>
```

We define subtyping between regular expression types in a semantic fashion. A type, in general, denotes a set of documents; subtyping is simply inclusion between the sets denoted by two types. For instance, consider again the `Person` type definition from above

```
type Person = person[Name,Email*,Tel?]
```

and the following variant:

```
type Person2 = person[(Name | Email | Tel)*]
```

Elements of the `Person` type can have one `name`, zero or more `emails`, and zero or one `tels` in this order, while the `Person2` type allows any number of such nodes in any order. Therefore `Person2` describes strictly more documents, which implies that `Person` is a subtype of `Person2`. Such subtype relations can be quite useful in programming. For example, suppose that we originally have a value of type `Person`. The above subtype relation allows us to process this value using code that does not care the ordering among the `name`, `email`, and `tel` nodes. (Such a situation might arise, for example, if we want to display the child nodes in a linear format, where we would naturally write a single loop over the sequence of child nodes with a case branch for each tag.)

Note that if we replaced `Person` with a more conventional type

```
person(name(string) × email(string) list × tel(string) option)
```

(using sum, product, and ML-like list and option types, plus unary covariant constructors `person`, `name`, `email`, and `tel`) and `Person2`, analogously, with

```
person((name(string) + email(string) + tel(string)) list)
```

the conventional subtyping of sum and product types would *not* yield the subtype relation above. In general, the subtype relation obtained from our definition is quite a bit more permissive than conventional subtyping. We give some further examples in Section 2.

Regular expression types are in one-to-one correspondence with *tree automata* [Common et al. 1999]—a finite-state machine model for accepting trees. It is easy to construct, from a given type, a tree automaton that accepts just the set of trees denoted by the type (Appendix A). Therefore the subtyping problem can be reduced to the inclusion problem between tree automata, which is known to be decidable. Unfortunately, in the worst case, this problem is EXPTIME-complete [Seidl 1990].

To address this high complexity, we develop an algorithm that runs efficiently for the cases that we expect to encounter most often in programming with regular expression types. Instead of beginning from classical algorithms for testing tree automata inclusion [Common et al. 1999], which work by (expensively) transforming their input automata to other completely separate automata before testing their properties, we take Aiken and Murphy's algorithm [Aiken and Murphy 1991] as the starting point. Like standard subtyping algorithms for other type systems, this algorithm works top-down—that is, a given pair of types, we check matching of the top-most type constructors, proceed to the subcomponents of the types, and repeat the same check recursively. The main advantage of this top-down algorithm is that it enables many simple optimizations. In particular, we can exploit reflexivity (`T <: T`) in order to decide subtype relations by looking at only a part of the whole input type expressions.

Our extensions to the work of Aiken and Murphy are threefold. First, we give proofs of soundness, completeness, and termination for (a reformulation of) their algorithm. Second, we incorporate several useful implementation techniques, including (1) sharing type expressions as much as possible by using a representation that exploits equivalence between types, and (2) caching intermediate results of the subtyping algorithm by using a *functional* data structure (for which the operations do not destructively update the data). We describe these techniques in detail in Section 5. Third, we describe some preliminary experiments with our algorithm (as implemented in the prototype XDuce interpreter), confirming its practicality on several small examples. (Aiken and Murphy's algorithm also addresses a more general problem: set-constraint solving. See Section 7 for further discussion of the relationship between their algorithm and ours.)

The contributions of this article can be summarized as follows:

—We motivate the use of regular expression types and set-inclusion-based subtyping for the domain of XML processing.

—We formalize the connection of regular expression types to tree automata and develop a practical subtyping algorithm, giving soundness, completeness, and termination proofs.

—We describe several implementation techniques for the subtyping algorithm and present preliminary measurements of their practical effects.

The article is organized as follows. In the next section, we give some examples of programming with regular expression types. In Section 3, we describe the connection of regular expression types to tree automata and give a precise definition of subtyping. In Section 4, we present our subtyping algorithm and prove its correctness. Section 5 describes our implementation techniques, and Section 6 presents the results of our preliminary experiments. We survey related work in Section 7 and conclude in Section 8. Appendix A presents an algorithm for translating regular expression types to tree automata, with its proof of correctness.

2. REGULAR EXPRESSION TYPES

We begin with a series of examples illustrating the application of regular expression types and subtyping to XML processing.

2.1 Values

Each type in our language denotes a set of sequences. Types like `String` and `tel[String]` denote singleton sequences; the type `Tel*` denotes sequences formed by repeating the singleton sequence `Tel` any finite number of times. So each element of the type `person[Tel*]` is a singleton sequence labeled with `person`, containing an arbitrary-length sequence of `Tel`s. If `S` and `T` are types, then the type `S,T` denotes all the sequences formed by concatenating a sequence from `S` and a sequence from `T`. The comma operator is associative: the types `(Name,Tel*),Addr` and `Name,(Tel*,Addr)` have exactly the same elements. As the unit element for the comma operator, we have the *empty* sequence type, written `()`. Thus, `Name,()` and `(),Name` are equivalent to `Name`.

2.2 Subtyping

The *subtype* relation between two types is simply inclusion between the sets of sequences that they denote. (See Section 3.1 for the formal definition.)

To illustrate the subtype relation, let us show the sequence of steps involved in verifying that the XML document given in the introduction actually has type `Addrbook`. First, from the intuition that `?` means "optional," we expect the following inclusions:

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

Notice that each right hand side describes a larger set of sequences than the left hand side. Similarly, `*` means "zero or more," so in particular it can be three:

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

Wrapping both sides of the first two inclusions with the label `person` and combining these with the third, we obtain:

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

Finally, enclosing both sides with the `addrbook` constructor, we obtain

```
   addrbook[
     person[Name,Addr],
     person[Name,Addr,Tel],
     person[Name,Addr]]
<: addrbook[(person[Name,Addr,Tel?])*]
=  Addrbook.
```

Since the XML document given in the introduction trivially has the type on the left hand side, it has also the type on the right hand side.

2.3 Regular Expression Types as Derived Forms

We have seen a rich variety of type constructors, but some of them can actually be derived as combinations of a smaller set of other constructors—concatenation, labeling, alternation, empty sequence, and recursive definition. For example, the optional type `T?` can be rewritten as `T|()`, using an alternation or *union type* and the empty sequence.

Other regular expression operators are also definable. `T+`, standing for one or more repetitions of `T`, can be rewritten as `(T,T*)`. Furthermore, `T*` itself can also be derived using recursion. That is, `T*` is equal to a type name `X` defined by the following equation:

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

Note the similarity to the definition of `list` as a datatype in ML.

2.4 Recursion

As in many type systems, we support 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 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. Intuitively, we expect that `GoodFld` should be a subtype of `Fld` because `GoodFld` allows fewer possibilities than `Fld`. Our type system validates this inclusion.

2.5 Subtagging

In XML processing, we sometimes encounter situations where we have to handle a large number of labels and it is convenient to organize them in a hierarchy, in the style of object-oriented languages. This leads us to support a notion of "subtagging" in our type system, allowing subtyping between types with different labels. This feature goes beyond the expressive power of DTDs, but a similar mechanism called "substitution groups" can be found in XML-Schema [Fallside 2001, Section 4.6].

The subtagging relation is a reflexive and transitive relation on labels. We declare subtagging explicitly with a set of global `subtag` forms. For example, the following declares that the tags `i` (italic) and `b` (bold) are subtags of `fontstyle`:

```
subtag i <: fontstyle
subtag b <: fontstyle
```

In the presence of these declarations, we have the subtype relations

```
i[T] <: fontstyle[T]
b[T] <: fontstyle[T]
```

for all `T`. These relations allow us to collapse two case branches for `i` and `b` into one for `fontstyle`, when both branches behave the same. This use of subtagging is similar to the common technique in object-oriented programming of defining an abstract class `fontstyle` with subclasses `i` and `b`.

Subtagging is also useful for other purposes. Our type system provides the special label `~`, denoting "any label." That is, for every label `l`, we have the following built-in subtagging relation:

```
subtag l <: ~
```

Thus, the type `~[T]` describes *any* labeled structure whose contents belong to type `T`. The label `~`, in turn, can be used to define a completely generic type `Any` as follows (assuming the base types in our language are `String`, `Int`, and `Float`):

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

That is, the values described by `Any` consist of zero or more repetitions of arbitrary labels (containing `Any`) and base types.

2.6 Pattern Matching

Regular expression types can enhance the pattern matching mechanisms found in mainstream functional languages. For example, in XDuce we can write the following pattern match:

```
  person[Name,Addr]*, (val x as person[Name,Addr,Tel]), Person*
   ->  (* do some stuff *)
| val y as person[Name,Addr]*
   ->  (* do other stuff *)
```

A pattern of the form `(val x as T)` matches any value of type `T` and binds the variable `x` to this value. In this example, the first case matches values containing at least one `person` with `tel`. In this case, the variable `x` is bound to the first `person` with a `tel`. The second case matches values containing no `person` with `tel`.

Notice that the first pattern—`person[Name,Addr]*`—in the first case matches a *variable length* sequence, something that is beyond the power of ML pattern matching. This makes standard techniques for exhaustiveness checking somewhat difficult to apply to such patterns. Fortunately, we can use subtyping for checking exhaustiveness of pattern matches. We assume that the type for the input value to the pattern match is available from the context. (In XDuce, this is ensured by type annotations on function headers.) In the example, suppose that the type of the input value is `Person*`. In order to show exhaustiveness of the pattern match with respect to this type, it is sufficient to show that every value of type `Person*` is accepted by the pattern. This leads to the following subtyping check

```
Person*  <:  (person[Name,Addr]*,
              person[Name,Addr,Tel],
              Person*)
            | person[Name,Addr]*
```

where the right hand side is calculated by taking the union of the types of the two pattern clauses. A more thorough study on regular expression pattern matching and static checking methods for it can be found in our separate paper [Hosoya and Pierce 2001].

2.7 Semistructured Data

One of the main application domains for XML is storing and transmitting databases. Since this flexible representation allows smooth evolution of database schemas and integration of databases with different schemas, it is often called a *semistructured* format in the database community. This view is especially useful for wrapper/mediator systems for the Web that integrate multiple independent data sources, which may themselves occasionally evolve. In this section, we present some different scenarios of database evolution and integration and show how our regular expression types ensure static safety in a flexible and robust way.

Suppose that we begin with the following XML database A

```
<addrbook>
  <person>
    <name>Haruo Hosoya</name>
    <addr>Tokyo</addr>
  </person>
  <person>
    <name>Jerome Vouillon</name>
    <addr>Paris</addr>
    <tel>123-456-789</tel>
  </person>
</addrbook>
```

belonging to the type `Addrbook`, defined as follows:

```
type Addrbook = addrbook[Person*]
type Person = person[Name,Addr,Tel?]
```

Now, suppose we upgrade this database so that some person records can contain arbitrarily many `tel`s. This process involves changes to types, databases, and programs. We change the types as follows:

```
type Person = person[Name,Addr,Tel*]
```

Notice that the new content type `(Name,Addr,Tel*)` of `person` is a supertype of the old type `(Name,Addr,Tel?)` and therefore the type `Addrbook` of the whole database becomes bigger as well. This means that our database, which had the old type, still conforms to the new type, without the need of restructuring. After adding some `tel` fields to our database, we arrive at the following database B:

```
<addrbook>
  <person>
    <name>Haruo Hosoya</name>
    <addr>Tokyo</addr>
    <tel>111-222-333</tel>
  </person>
  <person>
```

```
    <name>Jerome Vouillon</name>
    <addr>Paris</addr>
    <tel>123-456-789</tel>
    <tel>999-888-777</tel>
  </person>
</addrbook>
```

At each step in this process, the type of the database is `Addrbook`. The database can therefore smoothly evolve while preserving the robustness provided by type safety.

Upgrading the programs that operate on our database can be slightly trickier. Since the new type is a *supertype* of the old type rather than a subtype, all functions that *output* the old type can be treated as yielding the new type instead. On the other hand, functions that *input* the old type have to be modified so as to handle the additional cases. Some programmers may be happy with this, since the type system helps in isolating the part of the program requiring updates. Other programmers may feel that the types are preventing "forward compatibility" of old programs. For example, if we are interested in extracting specifically the `Name` field, then our program should work for the new database just as well as the old. But this sort of forward compatibility can easily be achieved, at the cost of writing the original program in a slightly more refined way: we maintain the convention that functions on `person`s should actually be able to handle inputs of type `(Name,Addr,Tel?,Any)`, simply ignoring the additional fields at the end. Now if, when the database's type is evolved, new fields are always added at the end, these old programs will work and typecheck without change.

XML also makes database integration easier than more structured formats such as relational databases. Again, regular expression types help ensure the type safety of integration steps. For example, consider integrating the previous database B with another database C with a slightly different type from B's:

```
type Addrbook2 = addrbook[Person2*]
type Person2 = person[Name,Addr,Email*]
type Email = email[String]
```

Data integration again involves changes to types, databases, and programs. We integrate databases by constructing a tree whose root has the label `addrbook` and whose content is the concatenation of the contents of the two databases. The natural type of this merged database is:

```
type Addrbook = addrbook[Person*,Person2*]
```

Suppose that we want to write a program to scan the whole sequence and extract the `name`s of all the `person`s. For writing such a program, the above type of the database is rather inconvenient, since it involves two occurrences of repetition, naturally leading to two separate loops for scanning the whole sequence. Obviously, it is better to roll these two loops into one. To do this, we can use a subtype inclusion that forgets the fact that all the `Person`s come before the `Person2`s:

```
Person*,Person2* <: (Person|Person2)*
```

Now we have *one* repetition, each of whose elements has either type `Person` or else type `Person2`, leading naturally to a one-loop scan. However, we can do better:

each step of the natural scan over this type involves two very similar cases, both of which just extract the Name field. We can use one more subtype inclusion (actually an equivalence, hence written =, using the fact that alternation distributes over labels and concatenations) to rewrite the type so that the common structure is exposed:

```
  person[Name,Addr,Tel*] | person[Name,Addr,Email*]
= person[(Name,Addr,Tel*) | (Name,Addr,Email*)]
= person[Name,Addr,(Tel*|Email*)]
```

After all this rewriting, the type AddrBook is now expressed in a form that leads naturally to scanning the Name fields with a single compact loop:

```
addrbook[ person[Name,Addr,(Tel*|Email*)]* ]
```

These distributive laws illustrate the flexibility of regular expression types. Note that the validity of these laws depends on the fact that our alternation operation is interpreted as an *untagged*, set-theoretic union—in contrast to the "tagged" sum types found in languages such as ML and Haskell.

3. DEFINITIONS

It will be useful to distinguish two forms of types: *external* and *internal*. The external form is one that the user actually reads and writes; all the examples in the previous sections are in this form. Internally, however, our subtyping algorithm uses the simpler internal representation to streamline both the implementation and its accompanying correctness proofs.

In this section, we define the syntax and semantics of each form as well as the subtype relations for each. For brevity, we omit base values (strings, numbers, etc.) and their types from our formalization; they are easily added.

3.1 External Form

We assume a countably infinite set of labels, ranged over by l. Values are defined as follows.

$$\mathtt{v} ::= \mathtt{l}_1\mathtt{[v]},..,\mathtt{l}_n\mathtt{[v]} \qquad (n \geq 0)$$

We write () for the empty sequence and v,w for the concatenation of sequences v and w.

We assume a countably infinite set of type names, ranged over by X. Type expressions are then defined as follows.

T ::=	()	empty sequence
	X	type variable
	l[T]	label
	T,T	concatenation
	T|T	union
	∅	empty set

The bindings of type variables 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 variables (in particular, definitions may be recursive). We regard E as a mapping from type variables to their bodies. The regular expression operators `*`, `+`, and `?` are derived forms as described in Section 2.3. We write $dom(E)$ for the set of defined variables.

We assume a global subtagging relation, a reflexive and transitive relation on labels, written $\prec$.

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

```
type X = a[],X,b[] | ()
```

Since the decision problem for inclusion between context free languages is undecidable [Hopcroft and Ullman 1979], 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 [Hopcroft and Ullman 1979], so we adopt a simple syntactic condition, called *well-formedness*, that ensures regularity. Intuitively, well-formedness allows unguarded (i.e., not enclosed by a label) recursive uses of variables, but restricts them to tail positions. For example, we allow the following type definitions:

```
type X = a[],Y
type Y = b[],X | ()
```

Formally, we define well-formedness in terms of a set of "non-tail variables" and an auxiliary set of "top-level variables." The set $toplevel(\mathtt{T})$ of top-level variables of a type `T` is the smallest set satisfying the following equations:

$$\begin{array}{lcll} toplevel(\mathtt{X}) & = & \{\mathtt{X}\} \cup toplevel(E(\mathtt{X})) & \\ toplevel(\mathtt{T}) & = & \emptyset & \text{if } \mathtt{T} = \emptyset,\ \mathtt{()},\ \text{or } \mathtt{l[T']} \\ toplevel(\mathtt{T|U}) & = & toplevel(\mathtt{T}) \cup toplevel(\mathtt{U}) & \\ toplevel(\mathtt{T,U}) & = & toplevel(\mathtt{T}) \cup toplevel(\mathtt{U}) & \end{array}$$

Likewise, the set $nontail(\mathtt{T})$ of non-tail variables of a type `T` is the smallest set satisfying the following equations:

$$\begin{array}{lcll} nontail(\mathtt{X}) & = & nontail(E(\mathtt{X})) & \\ nontail(\mathtt{T}) & = & \emptyset & \text{if } \mathtt{T} = \emptyset,\ \mathtt{()},\ \text{or } \mathtt{l[T']} \\ nontail(\mathtt{T|U}) & = & nontail(\mathtt{T}) \cup nontail(\mathtt{U}) & \\ nontail(\mathtt{T,U}) & = & toplevel(\mathtt{T}) \cup nontail(\mathtt{U}) & \end{array}$$

Now, the set E of type definitions is said to be *well-formed* if

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

From now on, we assume that a well-formed set E of type definitions is given once and for all.

The semantics of external types is given by the denotation function $[\![\mathtt{T}]\!]$, which is defined as the least solution of the following set of equations[1]:

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \emptyset \\ [\![\mathtt{()}]\!] & = & \{\mathtt{()}\} \\ [\![\mathtt{X}]\!] & = & [\![E(\mathtt{X})]\!] \\ [\![\mathtt{l[T]}]\!] & = & \{\mathtt{l'[v]} \mid \mathtt{l'} \prec \mathtt{l} \wedge \mathtt{v} \in [\![\mathtt{T}]\!]\} \\ [\![\mathtt{T,U}]\!] & = & \{\mathtt{v,w} \mid \mathtt{v} \in [\![\mathtt{T}]\!] \wedge \mathtt{w} \in [\![\mathtt{U}]\!]\} \\ [\![\mathtt{T} \mid \mathtt{U}]\!] & = & [\![\mathtt{T}]\!] \cup [\![\mathtt{U}]\!] \end{array}$$

These rules are straightforward except that a labeled type with label `l` denotes not only values with label `l` but also those with labels `l'` smaller than `l` in the sense of subtagging.

Subtyping is defined semantically: two internal types are in the subtype relation iff their denotations are in the subset relation:

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

3.2 Internal Form

In the external form, values are arbitrary-arity trees (i.e., each node can have an arbitrary number of children); in the internal form, we consider only binary trees.[2]

The labels l in the internal form are the same as labels in the external form. Internal (binary) *tree values* (or just *trees*) are defined as follows.

$$\begin{array}{rcll} t & ::= & \epsilon & \text{leaf} \\ & & l(t,t) & \text{label} \end{array}$$

There is a straightforward isomorphism between binary trees and sequences of arbitrary-arity trees. The leaf value ϵ corresponds to the empty sequence, while $l(t,t')$ corresponds to a sequence whose head is labeled `l`, with t corresponding to the content of `l` and t' to the remainder of the sequence. For example, from the arbitrary-arity tree

```
person[name[], email[]]
```

we can read off the binary tree

$$\mathtt{person}(\mathtt{name}(\epsilon, \mathtt{addr}(\epsilon,\epsilon)), \epsilon),$$

and vice versa.

To define the internal form of types, we begin as before by assuming a countably infinite set of (*internal*) *type states*, ranged over by X. A (*binary*) *tree automaton* M is a finite mapping from type states to (*internal*) *type expressions*, defined as

[1] A somewhat peculiar property of this type system is that one can define a type that is not syntactically $\emptyset$ but denotes the empty set, by using a "non-terminating" recursive type:

```
type X = a[X]
```

[2] Here, we consider automata accepting binary trees as the internal form of types. An alternative would be to use a more direct formalism, called hedge automata [Murata 2000], which accept arbitrary-arity trees.

follows:

$$
\begin{array}{rcll}
T & ::= & \emptyset & \text{empty set} \\
 & & \epsilon & \text{leaf} \\
 & & T \mid T & \text{union} \\
 & & l(X, X) & \text{label}
\end{array}
$$

(Note that this syntax ensures that type names can be used only under labels and labels cannot be nested.) There is a one-to-one correspondence between external and internal types, following the same intuition as for values. For example, the external type `person[name[],addr[]*]` corresponds to the internal type $\texttt{person}(X_1, X_0)$ where the states X_1 and X_0 are defined by the automaton M as follows.

$$
\begin{array}{l}
M(X_0) = \epsilon \\
M(X_1) = \texttt{name}(X_0, X_2) \\
M(X_2) = \texttt{addr}(X_0, X_2) \mid \epsilon
\end{array}
$$

The formal translation from external types to internal types is presented in full in Appendix A.

An internal type denotes a set of tree values. The denotation function $[\![T]\!]_M$ is defined as the least solution of the following set of equations:

$$
\begin{array}{lcl}
[\![\epsilon]\!]_M & = & \{\epsilon\} \\
[\![l(X, X')]\!]_M & = & \{l'(t, t') \mid l' \prec l \wedge t \in [\![M(X)]\!]_M \wedge t' \in [\![M(X')]\!]_M\} \\
[\![T \mid T']\!]_M & = & [\![T]\!]_M \cup [\![T']\!]_M \\
[\![\emptyset]\!]_M & = & \emptyset
\end{array}
$$

In the remainder of the article, we assume that a single context M is given once and for all and shared among all types. From now on we write simply $[\![T]\!]$ to mean $[\![T]\!]_M$.

Just as we did for the external form, we define subtyping between internal types as inclusion between the sets of trees denoted by the types.

$$T <: T' \;\text{ iff }\; [\![T]\!] \subseteq [\![T']\!].$$

Since types are now represented as tree automata, the subtyping problem can be reduced to the inclusion problem of regular tree languages.

3.2.1 Theorem [Complexity of Subtyping]: The decision problem for the (internal) subtype relation is EXPTIME-complete.

PROOF. We use Seidl's result of EXPTIME-completeness of tree automata inclusion [Seidl 1990]. Our formulation is different from his only in that we have subtagging. EXPTIME-hardness of our problem is trivial since any instance of Seidl's problem is an instance of ours (with no subtagging). For the other direction, we can transform our form of tree automata to his by a polynomial-time reduction. We transform each labeled type $l(X, X')$ to $\big|_{l' \prec l} l'(X, X')$. Comparing the resulting automata with no subtagging is equivalent to comparing the original automata with subtagging. Let L be the number of labels appearing in the two given tree automata and n be the number of labeled types appearing in them. The transformation increases the number of labeled types by $O(Ln)$. □

Although the worst-case complexity is quite high, the subtyping algorithm presented below appears to behave satisfactorily on practical examples.

4. SUBTYPING ALGORITHM

This section develops an algorithm for deciding the subtyping relation that is tuned to the domain of XML processing, and gives proofs of soundness, completeness, and termination.

4.1 Highlights

There is a classical algorithm for checking inclusion between tree automata. Given two automata M and M', it works as follows:

(1) Take the complement $\overline{M'}$ of M', by constructing a deterministic automaton from (the nondeterministic automaton) M' using a *subset* construction (where each state of the new automaton corresponds to a subset of the states of the original automaton), and then exchanging final and non-final states.
(2) Take the intersection of M and $\overline{M'}$ using a *product* construction.
(3) Test the emptiness of the result by checking reachability of final states.

(See [Common et al. 1999] for the details of each technique.) In an early implementation of XDuce, we actually used this algorithm. The problem quickly became apparent: the tree automata constructions that it uses will *always* realize any potential exponential blowup in the types involved. This is wasteful, since it is seldom necessary to explore all the states of the automata—in practice, most of the inputs to the subtype checker involve types with a large degree of sharing. For example, consider the inclusion from Section 2.7.

```
Person*,Person2* <: (Person|Person2)*
```

Since this inclusion holds whatever (and however big) the types `Person` and `Person2` may be, we should be able to check it without looking into their definitions.

In order to exploit this observation, we adopt, as a starting point, Aiken and Murphy's algorithm [Aiken and Murphy 1991]. Their algorithm works in a top-down manner—starting with a pair of types and, at each step, comparing the top-most type constructors and continuing recursively with corresponding subcomponents until we reach leaves that require only trivial checks. The main advantage of this top-down algorithm is that it enables numerous optimizations. In particular, we can use reflexivity (e.g., `Person <: Person` in the above example) to cut off large parts of the search. Also, we can cache intermediate results and use them for later comparisons.

This sort of top-down checking is quite standard in subtyping algorithms for conventional type systems. The differences from those algorithms arise from the presence of "untagged" union types in our setting, where two components of a union may have the same outermost label (unlike "tagged" or "disjoint" sum types, as in ML, where all components are required to have distinct tags). For example, we allow types like $l(R_1, S_1) \mid l(R_2, S_2)$.[3] To see the difficulties that this raises, suppose we want to check the following inclusion:

$$l(T, U) \texttt{ <: } l(R_1, S_1) \mid l(R_2, S_2)$$

[3]We use the internal form of types from now on. Also, we use in examples an informal notation that allows nested labels.

What subgoals should we generate? The first rule we might try is the following:

$$\text{(Weak-Rec)}\quad \frac{l(T,U) <: l(R_1,S_1) \quad \text{or} \quad l(T,U) <: l(R_2,S_2)}{l(T,U) <: l(R_1,S_1) \mid l(R_2,S_2)}$$

However, this rule is too weak. For example, if $T = R_1 \mid R_2$ and $U = S$, then neither premise holds, although we do have $l((R_1 \mid R_2), S) <: l(R_1,S) \mid l(R_2,S)$.

To avoid this counterexample, we might try first distributing all unions over labels. For example, to verify $l((R_1 \mid R_2),S) <: l(R_1,S) \mid l(R_2,S)$, we could transform the left-hand side to $l(R_1,S) \mid l(R_2,S)$ and then check whether each clause on the left appears on the right. However, this approach does not work for recursive types, where we would apply distributivity infinitely.

Fortunately, we can overcome the difficulty by reorganizing the clauses, using a simple set-theoretic observation. Let us consider a slightly more general case

$$l(T,U) <: l(R_1,S_1) \mid l(R_2,S_2) \mid l(R_3,S_3)$$

and consider a series of transformation of this relation. (This discussion can be further generalized to cases where the subtype relation to check has an arbitrary number of clauses on the right-hand side.) First, in general, two arguments to a labeled type $l(R,S)$ can be seen as a cross product $R \times S$, which in turn is equal to $(R \times \mathcal{T}) \cap (\mathcal{T} \times S)$, where the maximal type $\mathcal{T}$ denotes the set of all trees. Therefore the right-hand side of the subtype relation can be rewritten as follows.

$$(l(R_1,\mathcal{T}) \cap l(\mathcal{T},S_1)) \mid (l(R_2,\mathcal{T}) \cap l(\mathcal{T},S_2)) \mid (l(R_1,\mathcal{T}) \cap l(\mathcal{T},S_1))$$

Using distributivity of intersections over unions, we can turn this disjunctive form to the following conjunctive form.

$$\begin{array}{l}
(l(R_1,\mathcal{T}) \mid l(R_2,\mathcal{T}) \mid l(R_3,\mathcal{T})) \cap \\
(l(\mathcal{T},S_1) \mid l(R_2,\mathcal{T}) \mid l(R_3,\mathcal{T})) \cap \\
(l(R_1,\mathcal{T}) \mid l(\mathcal{T},S_2) \mid l(R_3,\mathcal{T})) \cap \\
(l(\mathcal{T},S_1) \mid l(\mathcal{T},S_2) \mid l(R_3,\mathcal{T})) \cap \\
(l(R_1,\mathcal{T}) \mid l(R_2,\mathcal{T}) \mid l(\mathcal{T},S_3)) \cap \\
\ldots
\end{array}$$

In each clause of the conjunctive form, if one argument R_i to l appears, then the other argument S_i does not appear, and vice versa. Therefore each clause can be rewritten as

$$\left(\Big|_{i\in I} l(R_i,\mathcal{T})\right) \mid \left(\Big|_{i\in \overline{I}} l(\mathcal{T},S_i)\right),$$

where I is a subset of $\{1,2,3\}$ and $\overline{I}$ is $\{1,2,3\} \setminus I$. Since the whole conjunctive form above is the intersection of such formulas for all subsets I of $\{1,2,3\}$, the original subtype relation reduces to checking, for each I, that

$$l(T,U) <: \left(\Big|_{i\in I} l(R_i,\mathcal{T})\right) \mid \left(\Big|_{i\in \overline{I}} l(\mathcal{T},S_i)\right),$$

or equivalently that

$$l(T,U) <: l(\Big|_{i\in I} R_i,\mathcal{T}) \mid l(\mathcal{T},\Big|_{i\in \overline{I}} S_i).$$

Write R_I for $\Big|_{i\in I} R_i$ and S_I for $\Big|_{i\in \overline{I}} S_i$.

Now, since each labeled type on the right has type $\mathcal{T}$ as one of its arguments, the situation becomes easier than the beginning: it suffices to test

$$T <: R_I \text{ or } U <: S_I.$$

To see why, suppose $l(T, U) <: l(R_I, \mathcal{T}) \mid l(\mathcal{T}, S_I)$ but $T \not<: R_I$ and $U \not<: S_I$. We can find trees $t \in T \setminus R_I$ and $u \in U \setminus S_I$. This means that $l(t, u) \in l(T, U)$ but neither $l(t, u) \in l(R_I, \mathcal{T})$ or $l(t, u) \in l(\mathcal{T}, S_I)$, which contradicts the assumption. (The other direction—if $T <: R_I$ or $U <: S_I$, then $l(T, U) <: l(R_I, \mathcal{T}) \mid l(\mathcal{T}, S_I)$—is obvious.)

Our subtyping algorithm is based on these intuitions. In addition, we incorporate several implementation techniques to further improve the efficiency. First, to make effective use of reflexivity checks, we recognize semantically equal types as often as possible and ensure that their representations are physically shared (i.e., we use a form of"hash consing" for types). Second, the subtyping algorithm may backtrack when checking the above formula "$T <: R_I$ or $U <: S_I$"; since, as we will see in Section 5.1, this requires reverting the cache that records intermediate results of subtype checking, we use functional data structures to represent the cache so as to make the reverting quick. Other implementation techniques capture some common special cases for untagged union types to avoid exploring an exponential number of subgoals (which is in general incurred by considering all the subsets I). In the following section, we describe the core of the algorithm, postponing the optimizations to Section 5.

4.2 Algorithm

The subtyping algorithm is defined by two judgments, $\Pi \vdash T <: U \Rightarrow \Pi'$ and $\Pi \vdash^* T <: U \Rightarrow \Pi'$, where Π is a set of pairs of types of the form $R <: S$, called "assumptions." Both judgments should be read: "assuming that all relations $R <: S$ in Π hold, the algorithm verifies $T <: U$ and yields, in the output set Π', both the old pairs in Π and the new pairs $T' <: U'$ that have been checked in the process." At each step of the subtyping algorithm, we store the given pair of types to the assumption set. Later on, when we encounter the same pair, we immediately return "yes," thus ensuring termination. Of course, we have to be careful not to check the assumption set immediately after storing the given pair, which would incorrectly establish subtyping between any pair of types. This is why we have two different judgments: we switch one to the other right after storing a pair and switch back whenever it is safe. The accumulated assumptions are eventually returned in the output set Π', which is propagated as the input to other subtype checks, avoiding repeated checks of the same pair. Furthermore, these pairs are reused not only in the process of checking a single inclusion, but all the way through the typechecking of the whole program, thus serving as a cache of all verified subtype relations. (Related discussions of assumptions can be found in [Amadio and Cardelli 1993; Brandt and Henglein 1997; Gapeyev et al. 2000].)

The subtyping algorithm defined below uses a *structural equivalence* $\equiv$ between types, which is defined as the smallest relation closed under the following rules:

$$
\begin{array}{rcll}
T \mid \emptyset & \equiv & T & \\
T \mid T & \equiv & T & \\
(T \mid U) \mid R & \equiv & T \mid (U \mid R) & \\
T \mid U & \equiv & U \mid T & \\
T \mid U & \equiv & T \mid R & \quad \text{if } U \equiv R
\end{array}
$$

These rules essentially treat a type as a *set* of leaf or labeled types. We write $\in_\equiv$ for set membership up to structural equivalence: $(T <: U) \in_\equiv \Pi$ iff $(T' <: U') \in \Pi$ for some $T' \equiv T$ and $U' \equiv U$.

We now give the rules for the subtyping algorithm. If the pair $T <: U$ of input types is already in the set Π of assumptions, we immediately succeed (rule HYP). Otherwise, we add the pair to this set (rule ASSUM).

$$
\text{(HYP)} \quad \frac{(T <: U) \in_\equiv \Pi}{\Pi \vdash T <: U \Rightarrow \Pi}
$$

$$
\text{(ASSUM)} \quad \frac{(T <: U) \notin_\equiv \Pi \qquad \Pi; T <: U \vdash^* T <: U \Rightarrow \Pi'}{\Pi \vdash T <: U \Rightarrow \Pi'}
$$

These two rules are for ensuring termination as well as avoidance of repeated checks of the same pair. In ASSUM, we switch from the judgment of the form $\Pi \vdash T <: U \Rightarrow \Pi'$ to $\Pi \vdash^* T <: U \Rightarrow \Pi'$, preventing the incorrect application of the rule HYP immediately following ASSUM. We keep using the judgment $\Pi \vdash^* T <: U \Rightarrow \Pi'$ in the subsequence rules, and switch back to the judgment $\Pi \vdash T <: U \Rightarrow \Pi'$ in the last rule REC below.

The remaining rules depend on the shapes of the input types. The first three handle the cases where the left-hand side is either an empty set type, a union type, or a leaf type.

$$
\text{(EMPTY)} \quad \frac{}{\Pi \vdash^* \emptyset <: T \Rightarrow \Pi}
$$

$$
\text{(SPLIT)} \quad \frac{\Pi \vdash^* T <: U \Rightarrow \Pi' \qquad \Pi' \vdash^* T' <: U \Rightarrow \Pi''}{\Pi \vdash^* T \mid T' <: U \Rightarrow \Pi''}
$$

$$
\text{(LEAF)} \quad \frac{T \equiv \epsilon \mid T'}{\Pi \vdash^* \epsilon <: T \Rightarrow \Pi}
$$

If the left-hand side is an empty set type, we simply return since the relation clearly holds (rule EMPTY). If the left-hand side is the union of two types T and T', we generate two subgoals for these types (rule SPLIT). The intuition behind this rule is the set-theoretic fact that $T \cup T' \subseteq U$ iff $T \subseteq U$ and $T' \subseteq U$. If the left-hand side is a leaf type, we check that the right-hand side also contains a leaf type (rule LEAF). The algorithm may fail only at this rule.

In the remaining cases, the left-hand side is a singleton labeled type. Define the function *card* as follows:

$$\begin{array}{lcll} card(T_1 \mid T_2) & = & card(T_1) + card(T_2) & \\ card(T) & = & 1 & \text{if } T = \epsilon \text{ or } l(X, X') \text{ or } \emptyset \end{array}$$

Now:

(PRUNE)

$$\frac{R \equiv l'(U, U') \mid R' \qquad card(R') < card(R) \qquad l \not\prec l' \qquad \Pi \vdash^* l(T, T') <: R' \Rightarrow \Pi'}{\Pi \vdash^* l(T, T') <: R \Rightarrow \Pi'}$$

(PRUNE-LEAF)

$$\frac{R \equiv \epsilon \mid R' \qquad card(R') < card(R) \qquad \Pi \vdash^* l(T, T') <: R' \Rightarrow \Pi'}{\Pi \vdash^* l(T, T') <: R \Rightarrow \Pi'}$$

(REC)

$$\frac{\begin{array}{c} \text{for all } 1 \leq j \leq n,\ l \prec l_j, \\ \text{for all } 1 \leq i \leq 2^n, \\ \text{either} \quad \Pi_{i-1} \vdash T <: \Big|_{j \in I_i^n} U_j \Rightarrow \Pi_i \quad \text{or} \quad \Pi_{i-1} \vdash T' <: \Big|_{j \in \overline{I_i^n}} U'_j \Rightarrow \Pi_i \\ \text{where } \ T = M(X) \quad T' = M(X') \\ U_j = M(Y_j) \quad U'_j = M(Y'_j) \end{array}}{\Pi_0 \vdash^* l(X, X') <: l_1(Y_1, Y'_1) \mid \ldots \mid l_n(Y_n, Y'_n) \Rightarrow \Pi_{2^n}}$$

The rules PRUNE and PRUNE-LEAF remove from the right hand side a leaf type ϵ and all types with labels l' that are not greater than l. The side condition $card(R') < card(R)$ guarantees that the algorithm proceeds with smaller types (i.e., it eliminates "stupid" ways of splitting R). The rule REC handles the other cases—i.e., where the right-hand side is the (possibly empty) union of types $l_j(Y_j, Y'_j)$, with all labels l_j greater than the label l—using the intuition explained in Section 4.1. We index the subsets of $\{1, \ldots, n\}$ in some arbitrary order from I_1^n to $I_{2^n}^n$. We write $\overline{I_i^n}$ for the complement $\{1, \ldots, n\} \setminus I_i^n$. For each index i, we prove that either T is a subtype of the union $\Big|_{j \in I_i^n} U_j$, or T' is a subtype of the union $\Big|_{j \in \overline{I_i^n}} U'_j$.

Figure 1 shows a proof-tree-like diagram that arises when we use the algorithm to check the subtype relation $X <: Y$, where we assume X and Y are defined in the global tree automaton as follows:

$$\begin{array}{lcl} M(X) & = & \epsilon \mid l(X, \epsilon) \\ M(Y) & = & \epsilon \mid l(Y, \epsilon) \end{array}$$

In the figure, we write Π for $\{X <: Y\}$. Notice that when $X <: Y$ is checked for the second time, HYP is used; thus the check terminates. Also, checking $\Pi \vdash^* l(X, \epsilon) <: l(Y, \epsilon)$ generates a complex form of premises, some of which fail; thus the check involves backtracking

$$
\dfrac{\dfrac{\dfrac{\textsc{Leaf}}{\Pi \vdash^* \epsilon <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Pi} \qquad \dfrac{\dfrac{\left(\begin{array}{c} \dfrac{\textsc{Hyp}}{\Pi \vdash X <: Y \Rightarrow \Pi} \\ or \\ \dfrac{fail}{\Pi \vdash \epsilon <: \emptyset} \end{array}\right) \text{ and } \left(\begin{array}{c} \dfrac{\dfrac{fail}{\;} \\ \vdots}{\Pi \vdash X <: \emptyset} \\ or \\ \dfrac{\textsc{Leaf}}{\Pi \vdash \epsilon <: \epsilon \Rightarrow \Pi} \end{array}\right)}{\Pi \vdash^* l(X,\epsilon) <: l(Y,\epsilon) \Rightarrow \Pi}}{\Pi \vdash^* l(X,\epsilon) <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Pi}}{\Pi \vdash^* \epsilon \mid l(X,\epsilon) <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Pi}}{\emptyset \vdash X <: Y \Rightarrow \Pi}
$$

Fig. 1. Example of the subtyping algorithm

Intuitively, we can check that the algorithm terminates by reasoning as follows. The process from EMPTY to PRUNE-LEAF obviously terminates because the cardinalities of the types always decrease. In REC, observe that X, X', $Y_1, \ldots, Y_n$, $Y_1', \ldots, Y_n'$ are always states of the tree automata of the input types. Since the set of such states is finite, the set of *unions* of such states, for which REC generates subgoals, is also finite (up to structural equivalence). ASSUMP keeps track of all such unions of states that the algorithm encounters, and HYP ensures termination. This argument is formalized in the next section.

4.3 Proofs

We start with some definitions used in the soundness proof. We first define the height of tree values as follows:

$$
\begin{aligned}
h(\epsilon) &= 1 \\
h(l(t,t')) &= 1 + \max(h(t), h(t'))
\end{aligned}
$$

We write $[\![T]\!]_n$ for $\{t \in [\![T]\!] \mid h(t) \leq n\}$. The sets $[\![T]\!]_n$ can be characterized by the following equations:

$$
\begin{aligned}
[\![T]\!]_0 &= \emptyset \\
[\![\emptyset]\!]_{n+1} &= \emptyset \\
[\![\epsilon]\!]_{n+1} &= \{\epsilon\} \\
[\![l(X,X')]\!]_{n+1} &= \{l'(t,t') \mid \; l' \prec l \;\wedge\; t \in [\![M(X)]\!]_n \;\wedge\; t' \in [\![M(X')]\!]_n\} \\
[\![T \mid T']\!]_{n+1} &= [\![T]\!]_{n+1} \cup [\![T']\!]_{n+1}
\end{aligned}
$$

We also define a family of subtype relations:

$$
T <:_n T' \text{ iff } [\![T]\!]_n \subseteq [\![T']\!]_n
$$

The following properties clearly hold:

—If $T <:_n T'$ for all n, then $T <: T'$.

—If $T <: T'$, then $T <:_n T'$ for all n.

—$T <:_0 T'$ for all types T and T'.

—If $T \equiv U$, then $[\![T]\!]_n = [\![U]\!]_n$ for all n.

Π is said to be *n-consistent* if $T <:_n U$ for all $(T <: U) \in_{\equiv} \Pi$ and *consistent* if Π is n-consistent for all n (that is, $T <: U$ for all $(T <: U) \in_{\equiv} \Pi$).

The proof of soundness uses the following lemma in the case of Rec.

4.3.1 Lemma: Let m be a natural number and A_i, B_i be sets $(1 \leq i \leq m)$. Define

$$\begin{aligned} I^m &= \{1, \ldots, m\} \\ C_m &= \bigcup_{i \in I^m} A_i \times B_i \\ D_m &= \bigcap_{I \subseteq I^m} (\bigcup_{j \in I} A_j \times \mathcal{T}) \cup (\bigcup_{j \in I^m \setminus I} \mathcal{T} \times B_j). \end{aligned}$$

Then, we have $C_m = D_m$ for all m.

Proof. By mathematical induction on m.

Case: $m = 0$

$C_m = D_m = \emptyset$.

Case: $m \geq 1$

First, D_m can be transformed as follows:

$$\begin{aligned} D_m &= \bigcap_{I \subseteq I^{m-1}} (\bigcup_{j \in I} A_j \times \mathcal{T}) \cup (\bigcup_{j \in I^{m-1} \setminus I} \mathcal{T} \times B_j) \cup (\mathcal{T} \times B_m) \\ &\cap \bigcap_{I \subseteq I^{m-1}} (\bigcup_{j \in I} A_j \times \mathcal{T}) \cup (A_m \times \mathcal{T}) \cup (\bigcup_{j \in I^{m-1} \setminus I} \mathcal{T} \times B_j) \\ &= \left(\bigcap_{I \subseteq I^{m-1}} (\bigcup_{j \in I} A_j \times \mathcal{T}) \cup (\bigcup_{j \in I^{m-1} \setminus I} \mathcal{T} \times B_j) \right) \cup ((\mathcal{T} \times B_m) \cap (A_m \times \mathcal{T})) \\ &= \left(\bigcap_{I \subseteq I^{m-1}} (\bigcup_{j \in I} A_j \times \mathcal{T}) \cup (\bigcup_{j \in I^{m-1} \setminus I} \mathcal{T} \times B_j) \right) \cup (A_m \times B_m) \end{aligned}$$

By using the induction hypothesis, the above formula equals to $C_{m-1} \cup (A_m \times B_m) = C_m$. □

4.3.2 Theorem [Soundness]: If $\Pi \vdash T <: U \Rightarrow \Pi'$ and Π is consistent, then $T <: U$ and Π' is consistent.

Proof. We prove stronger statements:

—Suppose $\Pi \vdash T <: U \Rightarrow \Pi'$. For all n, if Π is n-consistent, then $T <:_n U$ and Π' is n-consistent.

—Suppose $\Pi \vdash^* T <: U \Rightarrow \Pi'$. For all n, if Π is n-consistent, then $T <:_{n+1} U$ and Π' is n-consistent.

Then the assumption $\vdash T <: U \Rightarrow \Pi'$ entails that $T <:_n U$ and Π' is n-consistent for all n, which imply the result.

The proof proceeds by simultaneous induction on derivations of $\Pi \vdash T <: U \Rightarrow \Pi'$ and $\Pi \vdash^* T <: U \Rightarrow \Pi'$. (We elide the side conditions involving sizes in some of the rules, since they are irrelevant here.)

Case Hyp:

$$\frac{(T <: U) \in_{\equiv} \Pi}{\Pi \vdash T <: U \Rightarrow \Pi}$$

Immediate.

Case Assum:

$$\frac{\begin{array}{c}(T <: U) \notin_{\equiv} \Pi \\ \Pi; T <: U \vdash^* T <: U \Rightarrow \Pi'\end{array}}{\Pi \vdash T <: U \Rightarrow \Pi'}$$

Suppose that Π is n-consistent. We must show that $(\Pi; T <: U)$ is also n-consistent. From this, $T <:_n U$ will follow immediately and we can obtain the n-consistence of Π' using the the induction hypothesis.

We trivially have $T <:_0 U$. Also, since Π is n-consistent, we can easily show that Π is also i-consistent for all $0 \leq i \leq n$. Furthermore, by the induction hypothesis, for all $0 \leq i \leq n$, if $(\Pi; T <: U)$ is i-consistent, then $T <:_{i+1} U$. By an inner induction on i, it follows that $(\Pi; T <: U)$ is i-consistent for all $0 \leq i \leq n$, and in particular for $i = n$.

Case Split:

$$\frac{\Pi \vdash^* T <: U \Rightarrow \Pi' \qquad \Pi' \vdash^* T' <: U \Rightarrow \Pi''}{\Pi \vdash^* T \mid T' <: U \Rightarrow \Pi''}$$

Suppose Π is n-consistent. Using the induction hypothesis on the first premise, we find that $T <:_{n+1} U$ and Π' is n-consistent. Further using the induction hypothesis on the second premise, we obtain $T' <:_{n+1} U$, together with the n-consistency of Π''.

It remains to prove that $T \mid T' <:_{n+1} U$, that is, $[\![T]\!]_{n+1} \cup [\![T']\!]_{n+1} \subseteq [\![U]\!]_{n+1}$. This holds because, from $T <:_{n+1} U$ and $T' <:_{n+1} U$, we have $[\![T]\!]_{n+1} \subseteq [\![U]\!]_{n+1}$ and $[\![T']\!]_{n+1} \subseteq [\![U]\!]_{n+1}$.

Case Empty:

$$\frac{}{\Pi \vdash^* \emptyset <: T \Rightarrow \Pi}$$

$[\![\emptyset]\!]_{n+1} = \emptyset \subseteq [\![T]\!]_{n+1}$ for any T and n.

Case Leaf:

$$\frac{T \equiv \epsilon \mid T'}{\Pi \vdash^* \epsilon <: T \Rightarrow \Pi}$$

$[\![\epsilon]\!]_{n+1} = \{\epsilon\} \subseteq \{\epsilon\} \cup [\![T']\!]_{n+1} = [\![\epsilon \mid T']\!]_{n+1} = [\![T]\!]_{n+1}$, for any T' and n.

Case Prune**:**

$$\frac{R \equiv l'(U,U') \mid R' \qquad l \not\prec l' \qquad \Pi \vdash^* l(T,T') <: R' \Rightarrow \Pi'}{\Pi \vdash^* l(T,T') <: R \Rightarrow \Pi'}$$

Suppose that Π is n-consistent. By the induction hypothesis, $l(T,T') <:_{n+1} R'$ and Π' is n-consistent. Therefore $[\![l(T,T')]\!]_{n+1} \subseteq [\![R']\!]_{n+1}$. The result follows from this and $[\![R']\!]_{n+1} \subseteq [\![R']\!]_{n+1} \cup [\![l'(U,U')]\!]_{n+1} = [\![R]\!]_{n+1}$.

Case Prune-Leaf**:**

$$\frac{R \equiv \epsilon \mid R' \qquad \Pi \vdash^* l(T,T') <: R' \Rightarrow \Pi'}{\Pi \vdash^* l(T,T') <: R \Rightarrow \Pi'}$$

Similar.

Case Rec**:**

$$\frac{\begin{array}{c}\text{For all } 1 \le j \le m,\ l \prec l_j \\ \text{For all } 1 \le i \le 2^m, \text{ either} \\ \Pi_{i-1} \vdash T <: \Big|_{j \in I_i^m} U_j \Rightarrow \Pi_i \text{ or } \Pi_{i-1} \vdash T' <: \Big|_{j \in \overline{I_i^m}} U'_j \Rightarrow \Pi_i \\ \text{where } T = M(X) \qquad T' = M(X') \qquad U_j = M(Y_j) \qquad U'_j = M(Y'_j)\end{array}}{\Pi_0 \vdash^* l(X,X') <: l_1(Y_1,Y'_1) \mid \ldots \mid l_m(Y_m,Y'_m) \Rightarrow \Pi_{2^m}}$$

Suppose that Π is n-consistent. By assumption, Π_0 is n-consistent. By successive uses of the induction hypotheses, for all $1 \le i \le 2^m$, we have either $T <:_n \Big|_{j \in I_i^m} U_j$ or $T' <:_n \Big|_{j \in \overline{I_i^m}} U'_j$, where Π_i is n-consistent. We must show that

$$l(X,X') <:_{n+1} l_1(Y_1,Y'_1) \mid \ldots \mid l_m(Y_m,Y'_m)$$

that is,

$$[\![l(X,X')]\!]_{n+1} \subseteq [\![l_1(Y_1,Y'_1)]\!]_{n+1} \cup \ldots \cup [\![l_m(Y_m,Y'_m)]\!]_{n+1}.$$

For each j, since $l \prec l_j$, we have

$$\begin{array}{rcl} [\![l(Y_j,Y'_j)]\!]_{n+1} & = & \{l'(t,t') \mid l' \prec l \wedge t \in [\![U_j]\!]_n \wedge t' \in [\![U'_j]\!]_n\} \\ & \subseteq & \{l'(t,t') \mid l' \prec l_j \wedge t \in [\![U_j]\!]_n \wedge t' \in [\![U'_j]\!]_n\} \\ & \subseteq & [\![l_j(Y_j,Y'_j)]\!]_{n+1}. \end{array}$$

It is therefore sufficient to prove that

$$[\![l(X,X')]\!]_{n+1} \subseteq [\![l(Y_1,Y'_1)]\!]_{n+1} \cup \ldots \cup [\![l(Y_m,Y'_m)]\!]_{n+1}.$$

Notice that, by definition, we have

$$[\![l(X,X')]\!]_{n+1} = \{l'(t,t') \mid l' \prec l \wedge t \in [\![T]\!]_n \wedge t' \in [\![T']\!]_n\}$$

and this set is isomorphic to $\{l' \mid l' \prec l\} \times [\![T]\!]_n \times [\![T']\!]_n$; similarly, $[\![l(Y_i,Y'_i)]\!]_{n+1}$ is isomorphic to $\{l' \mid l' \prec l\} \times [\![U_i]\!]_n \times [\![U'_i]\!]_n$. Thus, the inclusion relation that we are working on proving can be rewritten as follows:

$$\{l' \mid l' \prec l\} \times [\![T]\!]_n \times [\![T']\!]_n \subseteq \bigcup_{1\leq j\leq m} (\{l' \mid l' \prec l\} \times [\![U_j]\!]_n \times [\![U'_j]\!]_n)$$

Since the label part $\{l' \mid l' \prec l\}$ of the products can be cancelled on both sides, we only consider the other parts. Hence,

$$[\![T]\!]_n \times [\![T']\!]_n \subseteq \bigcup_{1\leq j\leq m} ([\![U_j]\!]_n \times [\![U'_j]\!]_n).$$

Since $[\![U_j]\!]_n \times [\![U'_j]\!]_n = ([\![U_j]\!]_m \times \mathcal{T}) \cap (\mathcal{T} \times [\![U'_j]\!]_n)$, where $\mathcal{T}$ is the set of all trees, this inclusion is equivalent to

$$[\![T]\!]_n \times [\![T']\!]_n \subseteq \bigcup_{1\leq j\leq m} (([\![U_j]\!]_n \times \mathcal{T}) \cap (\mathcal{T} \times [\![U'_j]\!]_n))$$

which is in turn equivalent to

$$[\![T]\!]_n \times [\![T']\!]_n \ \subseteq\ \bigcap_{1\leq i\leq 2^m} (\bigcup_{j\in I_i^m} ([\![U_j]\!]_n \times \mathcal{T})\ \cup\ \bigcup_{j\in \overline{I_i^m}} (\mathcal{T} \times [\![U'_j]\!]_n))$$

using Lemma 4.3.1. What we need to show is now that for all $1 \leq i \leq 2^n$,

$$[\![T]\!]_n \times [\![T']\!]_n \subseteq \bigcup_{j\in I_i^m} ([\![U_j]\!]_n \times \mathcal{T}) \cup \bigcup_{j\in \overline{I_i^m}} (\mathcal{T} \times [\![U'_j]\!]_n).$$

This follows, for each i, from one of the following inclusions:

$$\begin{array}{rl} & [\![T]\!]_n \ \subseteq \bigcup_{j\in I_i^m} [\![U_j]\!]_n \\ \text{or} & [\![T']\!]_n \ \subseteq \bigcup_{j\in \overline{I_i^m}} [\![U'_j]\!]_n \end{array}$$

Each of these, in turn, follows from the definition of one of the inclusions

$$\begin{array}{rl} & T \mathrel{<:}_n \Big|_{j\in I_i^m} U_j \\ \text{or} & T' \mathrel{<:}_n \Big|_{j\in \overline{I_i^m}} U'_j, \end{array}$$

which we derived above using the induction hypotheses. □

4.3.3 Theorem [Termination]: For all Π, T, and U, the algorithm either proves $\Pi \vdash T \mathrel{<:} U \Rightarrow \Pi'$ for some Π' or else fails.

PROOF. We prove instead the following statement (from which the one above follows directly): for all Π, T, and U, the algorithm either proves $\Pi \vdash^* T \mathrel{<:} U \Rightarrow \Pi'$ for some $\Pi' \supseteq \Pi$ or else fails.

Let $\mathcal{X}$ be the set of states in the global automaton. The statement can be proved by induction on the lexicographic order of $(|\mathcal{X}| \cdot 2^{|\mathcal{X}|} - |\Pi|, \mathit{card}(T) + \mathit{card}(U))$. The crucial observations are that the rules SPLIT, PRUNE, and PRUNE-LEAF always decrease $\mathit{card}(T) + \mathit{card}(U)$, and that the rule REC is always followed either by HYP, which immediately terminates, or by ASSUM, which increases Π. Since an element of Π is always a pair of the body of a state and the union of the bodies of some states, the size of Π is bounded by $|\mathcal{X}| \cdot 2^{|\mathcal{X}|}$. □

4.3.4 Theorem [Completeness]: If $T \mathrel{<:} U$, then, for all Π, there is some Π' such that $\Pi \vdash T \mathrel{<:} U \Rightarrow \Pi'$.

PROOF. We prove a stronger statement: If $T <: U$, then $\Pi \vdash T <: U \Rightarrow \Pi'$ and $\Pi \vdash^* T <: U \Rightarrow \Pi'$. We show this by contradiction, that is, if $T <: U$ but $\Pi \not\vdash T <: U \Rightarrow \Pi'$ or $\Pi \not\vdash^* T <: U \Rightarrow \Pi'$, then there is a tree t such that $t \in [\![T]\!]$ and $t \notin [\![U]\!]$, contradicting $T <: U$.

The termination of the algorithm ensures that if $\Pi \not\vdash T <: U \Rightarrow \Pi'$ or not $\Pi \not\vdash^* T <: U \Rightarrow \Pi'$, then the algorithm fails by a finite number of steps. In other words, the algorithm yields a (finite) proof tree that shows the failure. Now, by induction on the height of this proof tree, we prove the statement in the previous paragraph.

Case ASSUM:

$$\frac{(T <: U) \notin_{\equiv} \Pi \qquad \Pi; T <: U \not\vdash^* T <: U \Rightarrow \Pi'}{\Pi \not\vdash T <: U \Rightarrow \Pi'}$$

The result follows by the induction hypothesis.

Case SPLIT:

$$\frac{\Pi \not\vdash^* T <: U \Rightarrow \Pi' \quad \text{or} \quad \Pi' \not\vdash^* T' <: U \Rightarrow \Pi''}{\Pi \not\vdash^* T \mid T' <: U \Rightarrow \Pi''}$$

By the induction hypothesis, there is t such that $t \in [\![T]\!]$ or $t \in [\![T']\!]$, and $t \notin [\![U]\!]$, from which the result follows.

Case PRUNE:

$$\frac{R \equiv l'(U, U') \mid R' \qquad card(R') < card(R) \qquad l \not\prec l' \qquad \Pi \not\vdash^* l(T, T') <: R' \Rightarrow \Pi'}{\Pi \not\vdash^* l(T, T') <: R \Rightarrow \Pi'}$$

By the induction hypothesis, there is $t \in [\![l(T, T')]\!]$ such that $t \notin [\![R']\!]$. By definition, $t = k(t_1, t_2)$ for some $k \prec l$ and t_1 and t_2 with $t_1 \in [\![T]\!]$ and $t_2 \in [\![T']\!]$. This implies that $t' = l(t_1, t_2) \in [\![l(T, T')]\!]$ and $t' \notin [\![R']\!]$. Since $l \not\prec l'$, we also have $t' \notin [\![l'(U, U')]\!]$. Therefore $t' \notin [\![l'(U, U') \mid R']\!] = [\![R]\!]$.

Case LEAF:

$$\frac{T \not\equiv \epsilon \mid T'}{\Pi \not\vdash^* \epsilon <: T \Rightarrow \Pi}$$

The result follows from $\epsilon \in [\![T]\!]$ and $\epsilon \notin [\![U]\!]$.

Case PRUNE-LEAF:

$$\frac{R \equiv \epsilon \mid R' \qquad card(R') < card(R) \qquad \Pi \not\vdash^* l(T, T') <: R' \Rightarrow \Pi'}{\Pi \not\vdash^* l(T, T') <: R \Rightarrow \Pi'}$$

where the premise is a stuck derivation. The result can be proved similarly to the previous case.

Case Rec**:**

$$\frac{\begin{array}{c}\text{For all } 1 \leq j \leq n,\ l \prec l_j \\ \text{For some } 1 \leq i \leq 2^m, \text{ both} \\ \Pi_{i-1} \not\vdash T <: \Big|_{j \in I_i^m} U_j \Rightarrow \Pi_i \text{ and } \Pi_{i-1} \not\vdash T' <: \Big|_{j \in \overline{I_i^m}} U'_j \Rightarrow \Pi_i \\ \text{where } T = M(X) \qquad T' = M(X') \qquad U_j = M(Y_j) \qquad U'_j = M(Y'_j)\end{array}}{\Pi_0 \not\vdash^* l(X, X') <: l_1(Y_1, Y'_1) \mid \ldots \mid l_n(Y_n, Y'_n) \Rightarrow \Pi_{2^n}}$$

By the induction hypothesis, there are

—$t \in [\![T]\!]$ such that $t \notin [\![\Big|_{j \in I_i^n} U_j]\!] (= \bigcup_{j \in I_i^n} [\![U_j]\!])$, and

—$t' \in [\![T']\!]$ such that $t' \notin [\![\Big|_{j \in \overline{I_i^n}} U'_j]\!] (= \bigcup_{j \in \overline{I_i^n}} [\![U'_j]\!])$.

Then we have $l(t, t') \in [\![l(X, X')]\!]$. Suppose we had $l(t, t') \in [\![l_1(Y_1, Y'_1) \mid \ldots \mid l_n(Y_n, Y'_n)]\!]$. Then there would be some j such that $l(t, t') \in [\![l_j(Y_j, Y'_j)]\!]$, which implies $t \in [\![U_j]\!]$ and $t' \in [\![U'_j]\!]$. This contradicts either $t \notin \bigcup_{j \in I_i^n} [\![U_j]\!]$ or $t' \notin \bigcup_{j \in \overline{I_i^n}} [\![U'_j]\!]$. $\square$

5. IMPLEMENTATION

Our subtype checker implementation embodies of the rules in the previous section, plus a number of optimizations specialized to the subtyping problems that arise in practice in the XML domain. This section describes several of these techniques. They are categorized into low-level, representational techniques and higher-level heuristics inspired by set-theoretic observations.

5.1 Low-level Techniques

In order for the subtyping algorithm to be able to exploit fast reflexivity checks (as mentioned in 4.1), it is crucial to share the physical representations of internal types as much as possible. To this end, we recognize some semantically equal external types and translate them to the same internal type. The equalities that we exploit are commutativity, associativity, and idempotency of the union operator, and associativity of the concatenation operator. In order to make the recognition of equal types quick, we represent external types in such a way that equal types are *structurally* equal. We represent the union of types as a set of types, with nested unions flattened; thus, `((R|S)|(T|U))` and `(U|(S|(T|R)))` can be recognized as equal, for example. Since we use only union and equality for the operations on such sets, a suitable representation is a sorted list, which allows us to perform these two operations in linear time. (We sort the types in the list by their hash values, which are computed from the structures of the types.) We represent the concatenation of types simply by a list of types, with nested concatenations flattened and the empty sequence removed; thus, `(((R,()),S),(T,U))` and `(R,(S,(T,U)))` can be recognized as equal, for example. To further improve the speed of equality tests, we use *hash consing* (which associates each type expression with its integer hash value, so that equality can be quickly checked in most cases by comparing their hash values.)

In the internal form of types, we use cyclic structures to represent recursion. Compared to explicit use of recursion variables and table lookups, this representation not only makes the dereferencing slightly faster, but simplifies the implementation.

We need to be a little careful about the representation of the sets of assumptions Π described in Section 4.2. A hash table might initially appear to be a suitable representation, but it is not. To see why, recall that in REC, we have the premises "$T <: \ldots$ or $T' <: \ldots$." Suppose that the first premise fails. Then we try the second premise, to which we need to pass the original set of assumptions. However, if we represent the set of assumptions by a hash table, then the hash table available after the first premise may contain some assumptions that have been added during the processing of the first premise. Since those assumptions may be wrong, we have to remove them from the hash table. Or, we could copy the whole table before trying the first premise. Either would be very expensive. Instead, we use a functional representation of sets. Since the required operations here are insertion and membership testing (unlike the unions of types described above), we use *balanced binary trees.* Operations for balanced binary trees take logarithmic time in the size of the set, as opposed to constant time for hash tables; but this does not appear to cause any problems in practice.

5.2 High-level Techniques

5.2.1 *Literal equality.* In the implementation of a subtype checker for any type system, the most trivial optimization is, before going deeply into the structures, checking if the given types are literally equal (we use a physical equality check in the implementation). In the presence of union types, this can be slightly generalized, using the fact that $T <: T \mid U$. In our implementation, we use the following rule

$$\text{(TRIV)}\quad \frac{\Pi \vdash U <: T \mid R \Rightarrow \Pi'}{\Pi \vdash T \mid U <: T \mid R \Rightarrow \Pi'}$$

which can be seen as a combination of SPLIT and $T <: T \mid U$. (We try this rule before HYP.) (Notice that this rule simply takes the difference between two sets of types. Since we represent the union of types as a sorted list, the difference can be calculated by a linear scan.)

5.2.2 *Empty type elimination.* In this optimization (which we call EMP), before starting the subtyping algorithm, we perform a preprocessing step that eliminates all the types denoting the empty set. Since the subtyping algorithm can now assume that any type it encounters is not empty, some tests can be short cut. For example, two subgoals of the form $T <: \emptyset$ in the rule REC need not be generated. In particular, in the special case of REC where the right hand side is just a single labeled type—$l(T, U) <: l'(R, S)$—we only need to check $T <: R$ and $U <: S$ with $l \prec l'$, which involves no backtracking at all.

The preprocessing of empty type elimination can be highly tuned. In theory, identifying and eliminating empty states can be performed in linear time [Common et al. 1999]. However, we use a simpler but potentially quadratic algorithm, which seems to perform better in practice. (See Section 6.)

5.2.3 *Merging labeled types.* In order to make the previous optimization more effective, we merge types on the right hand side when either the first or second arguments are the same:

$$\frac{\Pi \vdash^* l(T,U) <: l'(R,(S \mid E)) \Rightarrow \Pi' \qquad l \prec l'}{\Pi \vdash^* l(T,U) <: l'(R,S) \mid l'(R,E) \Rightarrow \Pi'} \quad \text{(Merge1)}$$

$$\frac{\Pi \vdash^* l(T,U) <: l'((R \mid S),E) \Rightarrow \Pi' \qquad l \prec l'}{\Pi \vdash^* l(T,U) <: l'(R,E) \mid l'(S,E) \Rightarrow \Pi'} \quad \text{(Merge2)}$$

(We try these rules before SINGLE.) In our experience, the first case is more common than the second case. This is because, in the external form of types, labels of the same name often have the same content type. (When we "import" existing DTDs and interpret them as regular expression types, labels of the same name are even *required* to have the same content type.) Therefore we check MERGE1 first and then MERGE2.

5.2.4 *Default case.* If the type on the right hand side in REC has the form $U \mid R_1 \mid \ldots \mid R_n$ and U is larger than any C_i, then we only need to compare the left hand side with this largest type U:

$$\frac{\begin{array}{c}1 \leq i \leq n \qquad \Pi_i \vdash^* R_i <: U \Rightarrow \Pi_{i+1} \\ \Pi_{n+1} \vdash^* T <: U \Rightarrow \Pi'\end{array}}{\Pi_1 \vdash^* T <: U \mid R_1 \mid \ldots \mid R_n \Rightarrow \Pi'} \quad \text{(Super)}$$

(We try this rule before REC after MERGE1/2.) This typically happens when the programmer writes a "default" case in a pattern match, which is given a type (e.g. `Any`) that covers all the other cases.

In principle, this optimization can generate, in the course of searching for the largest type, so many subgoals that the cost surpasses the gain. However, we have not found such a situation in practice so far (see Section 6). This is probably because most of the cases are handled by the previous optimizations, and because the general rule REC (which we would have to use if not using SUPER) is often more expensive.

5.2.5 *Type* `Any`. Although the type `Any` can be encoded using the special label `~` as described in Section 2.5, it is useful to represent `Any` as a special state in the internal form, so that we can use the following rule:

$$\frac{}{\Pi \vdash^* T <: \texttt{Any} \Rightarrow \Pi} \quad \text{(Any)}$$

(We try this rule before SPLIT.) `Any` is used quite heavily in our programs, especially when we do not want to specify precise types in patterns. Without this optimization, we would have to traverse the whole structure of the left hand side type T, where every step just compares a subphrase of T with `Any`.

Using these heuristics together, almost all uses of the rule REC are eliminated in our example programs (see the following section).

Application	# of lines		time (sec)		subtype alg. internals	
	XDuce	DTD	total	subtyping	states	assumptions
Bookmarks	310	1197	0.48	0.018	756	133
Diff	355	—	0.039	0.014	276	165
Html2Latex (strict)	307	989	0.58	0.22	783	345
Html2Latex (transitional)	312	1197	0.88	0.36	946	433
Html2Latex (frameset)	323	1226	0.87	0.34	975	454

Table I. Applications and Measurement Results

6. PRELIMINARY EXPERIMENTS

We have incorporated the subtyping algorithm described above in a prototype implementation of the XDuce language. XDuce is a simple first-order functional language; a typical program consists of a collection of type declarations and recursive functions that use pattern matching to analyze input values. XDuce can parse ordinary DTDs, interpreting them as as regular expression type declarations. Some of our applications use this feature to incorporate fairly large DTDs from real-world XML applications.

In this section, we present the results of some preliminary performance measurements of our implementation. In the experiments, we are interested in (1) the wall-clock time that our algorithm takes to typecheck various application programs (subtype checks consume most of this time, and (2) the separate effects of each high-level optimization. The platform for our experiment was a Sun Enterprise 3000 (250MHz UltraSPARC) running SunOS 5.7.

Our test suite consists of three modest, but non-trivial, applications written in XDuce:

Bookmarks. A simple database query. It takes as input a Netscape bookmarks file of type `Bookmarks`, which is a subset of the (much larger) type `HTML`. It extracts a particular 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 type of the result is the full `HTML` type.

Html2Latex. Takes an HTML file (of type `HTML`) and converts it into LaTeX (a value of type `String`) by interpreting some of the markup commands.

Diff. An implementation of Chawathe's "tree diff" algorithm [Chawathe 1999]. It takes a pair of XML files of type `Xml`, which is the type of all XML documents, and returns a tree with annotations indicating whether each subtree has been retained, inserted, deleted, or changed between the two inputs.

The first two applications are written as straightforward traversals of the input tree by several simple recursive functions. The third one is more complex. Its 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, using types to ensure that the annotations and the actual trees are never confused. In the course of writing these applications, our type checker proved to be a tremendous help in finding silly mistakes.

The `HTML` type (more precisely, `XHTML`, which is an `XML` implementation of `HTML`) currently one of the largest types for documents found on the. This makes it an excellent benchmark case for our implementation. There are actually three versions of `XHTML`: XHTML-strict, XHTML-transitional, and XHTML-frameset; accordingly, our Html2Latex application comes in three versions. The first is smallest and the third is slightly larger than the second.

The first group of columns in Table I shows the number of lines in XDuce code (counting functions and types written in XDuce syntax, but not external DTDs), and the number of lines in external DTDs (if used). The difference between the three versions of Html2Latex is mainly in the number of lines of DTDs. The column "total" in the table shows the total time spent by the subtyping algorithm during the type checking of the whole program. It includes conversion from the external form to the internal form (INT), empty type elimination (EMP), and the main subtyping algorithm (SUB). The optimizations are all turned on for this table. The column "subtyping" shows the time spent by the main algorithm SUB. The table gives two more columns: "states" and "assumptions." The "states" column indicates the number of states of the internal form stored in the system, and the "assumps" indicates the number of pairs stored in the set of assumptions.

As the table indicates, the speed of type checking is acceptable for these applications. In particular, it takes less than one second to type check programs involving the full `HTML` type. Also, the space consumed by the subtyping algorithm indicated by the number of assumptions is relatively small, despite its potential of blow-up. These results are quite encouraging, though what will happen for bigger applications is not clear from these small experiments.

Notice that for the applications that we use, the ratio of the "subtyping" column to the "total" column is relatively small, even though subtyping takes exponential time in the worst case. This ratio would not be so small if we did not use any of our high-level optimizations, as we will discuss next.

We ran a further set of measurements to evaluate the effects of each high-level optimization. The measurement strategy is as follows. We start with a subtype checker that implements the set of rules in Section 4.2 and the low-level techniques described in Section 5.1 (None). We then turn on the high-level optimizations one by one and measure the performance. We add high-level optimizations in this order: Emp, Triv, Merge1, Merge2, Super and Any.

The graphs in Figure 2 show how the behavior of our algorithm varies as we add each optimization. We only consider the "transitional" version of Html2Latex since the graphs look alike for other versions. In addition, we give a graph for a program where we need just one subtype check: `Html <: Xml`. This check defeats the optimizations Triv and Any and requires a deep comparisons of the two types. Each graph provides two kinds of information. The first is given by bars indicating the transition of the time spent by the algorithm. Each bar in the graphs has three components corresponding to each phase of the subtyping procedure. We are mainly interested in SUB. The others are only presented so as to show the ratio of SUB in the total time. The times, except for SUB, should not depend on the optimization

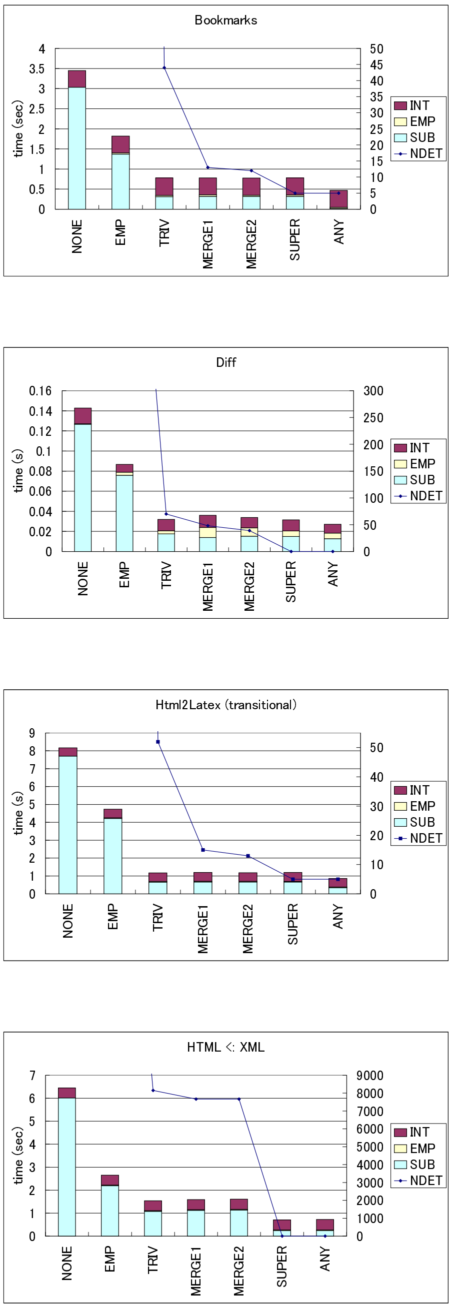

Fig. 2. The Effects of High-level Optimizations

switches but one can see noises in the graphs (especially in Diff). We suspect that this is due to garbage collection since this occurs consistently over several runs. The second piece of information is given by a line indicating the number of times the rule REC is used (NDET). (For NONE and EMP, these numbers are sometimes too big to fit on the graph.)

We first notice that EMP and TRIV are always effective. (For Diff, unfortunately, the total time becomes bigger with MERGE1 because of the "GC noise.") ANY is effective in Bookmarks; so is SUPER in HTML <: XML. The effects of MERGE1 and MERGE2 are unclear in these graphs.

The number of times the rule REC is used almost always decreases each time an optimization is added; by using all the optimizations, this rule can almost be avoided. However, even though REC rule is usually avoided, the savings in wall-clock time is less. This indicates that, in some cases, the cost of the optimization and that of blindly using REC rule may be comparable.

Also, notice that the cost of empty type elimination (EMP) is very low.

Since the experiments we have run so far have all been quite small, we hesitate to conclude too much from them; still, a few observations emerge fairly clearly. It is obvious that simple optimizations such as EMP, TRIV and ANY are effective. Also, although we cannot see the effectiveness of MERGE1, MERGE2, and SUPER in terms of wall-clock time, they at least seem to have no serious negative effects and, moreover, they do quite often avoid REC rule.

7. RELATED WORK

Static typing of programs for XML processing has been approached from several different angles. One popular idea is to embed a type system for XML in an existing typed language. The advantage is that we can enjoy not only static type safety, but also all the other features provided by the host 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 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 integration" that we discussed in Section 2.7, where ordering-forgetting subtyping and distributivity were critically needed.

A recent example of the embedding approach is Wallace and Runciman's proposal to use Haskell as a host language [Wallace and Runciman 1999] 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. A difference in the other direction is that our type system does not currently support higher-order functions or parametric polymorphism. (We are working on both of these extensions, but they are nontrivial.)

Meijer and Shields propose the typed functional language XMλ for XML processing [Meijer and Shields 1999; Shields and Meijer 2001]. Although their type system stems from Haskell's, they attain additional flexibility required in XML processing by incorporating, instead of subtyping, extensible records and variants based on row polymorphism.

The query language YAT [Cluet and Siméon 1998] has a type system similar to regular expression types. They also provide a notion of subtyping that can make programs flexible against changes of types. However, their subtyping is somewhat weaker than ours: they can handle the first example ("database evolution") in Section 2.7, but since their subtyping lacks distributive laws, they cannot treat the second example ("database integration") given in the same section.

Since its initial publication, our work on regular expression types has influenced proposals by other researchers. In particular, 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.].

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 [Milo et al. 2000]. The types there are based on tree automata and conceptually identical to regular expression types. Papakonstantinou and Vianu present a typechecking algorithm for their query language *loto-ql*, where the algorithm uses extensions to DTDs [Papakonstantinou and Vianu 2000]. One of their extensions is equivalent to tree automata.

Although schema languages for XML do not treat static verification of programs, the type structures in these languages and regular expression types are worth discussing. Since schema languages are too many to enumerate, we consider the following representative ones: DTD [Bray et al. 2000], XML-Schema [Fallside 2001], DSD (Document Structure Description) [Klarlund et al. 2000], RELAX [Murata 2001], and SOX (Schema for Object-Oriented XML) [Davidson et al. 1999]. As our regular expression types do, all of these use regular expressions or similar mechanisms for describing sequences. Both our regular expression types and RELAX are equivalent to tree automata (more precisely, nondeterministic tree automata; see [Common et al. 1999] for a detailed classification of tree automata), and therefore these have the same expressiveness. The other schema languages are more restrictive. Some of the above schema languages consider subtyping. XML-Schema has a "restriction" subtyping (which yields a subtype by decreasing the number of choices in an existing type), and both XML-Schema and SOX have an "extension" subtyping (which yields a subtype by adding new fields to the tail, similarly to inheritance in object-oriented languages). Both forms of subtyping are subsumed by ours. Kuper and Siméon define a schema language similar to regular expression types; they provide a subtype relation that is weaker than ours but allows efficient storage layout for XML databases [Kuper and Siméon 2001].

Several researchers have developed type systems for query languages for tree-like data that are similar to XML except that they have no ordering among children nodes (some of the formalisms also allow cycles). Their unorderedness makes these type systems quite different from those for XML. In the type system studied by Buneman, Davidson, Fernandez, and Suciu [Buneman et al. 1997], types are graph structures and their conformance and subtype relations are defined in terms of graph simulation (which is is weaker than the inclusion relation). The query language Lorel uses a somewhat similar type system called DataGuide [Goldman and Widom

1997].

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 difference from the present work is that they study unordered record types instead of ordered sequences and do not treat recursive types.

Our subtyping algorithm can be seen as an extension to subtyping algorithms for conventional type systems. In particular, the technique used in our algorithm for keeping track of a set of "assumptions" to detect termination (Section 4.2) can be found in many subtyping algorithms for simple recursive types [Amadio and Cardelli 1993; Brandt and Henglein 1997; Gapeyev et al. 2000]. The ingredient that is rarely found in classical algorithms is the complex rule shown Section 4.2 for the "untagged" union type operator.

As we discussed above, our subtyping algorithm can be viewed as a variant of Aiken and Murphy's algorithm [Aiken and Murphy 1991]. They actually treat a more general problem called set-constraint solving. In their setting, types can contain free variables and the goal is to obtain a substitution for the variables that satisfies the given set-constraints, if they are satisfiable. In our setting, on the other hand, types do not contain free variables and the goal is to know only whether the constraints are satisfiable. Thus, we can solve our subtyping problem by using their algorithm and removing the rules related to free variables. They do not, however, prove correctness in their paper—in particular, they have no completeness proof since their algorithm is indeed incomplete. (Incompleteness means that the algorithm may answer "no" even if the constraints are solvable.) What we have proved is that their algorithm is actually complete if we restrict types to contain no free variables.

In their framework for *refinement types* for ML, Freeman and Pfenning adopt regular tree languages to express more precise types than ML datatypes and use tests for inclusion between tree automata in type checking [Freeman and Pfenning 1991]. Davies addresses the efficiency issue of the inclusion test in this context [Davies 2000].

Damm [Damm 1994] considers a type system with union, intersection, and recursive types where the subtype relation is defined in a semantic way. He presents a decision procedure for subtype checking that first encodes types into regular tree expressions and then uses existing set-constraint solving algorithms. Furthermore, he treats higher-order functions, which we have not handled yet. On the other hand, he does not address the efficiency of the subtyping algorithm.

8. CONCLUSIONS

We have proposed regular expression types for XML processing, arguing that set-inclusion-based subtyping and subtagging yield useful expressive power in this domain. We developed an algorithm for subtyping that can exploit sharing among type expressions, giving soundness, completeness, and termination proofs. By incorporating several optimization techniques, our algorithm runs at acceptable speeds on several applications involving fairly large types, such as the complete DTD for HTML documents.

Our work on type systems for XML processing has just begun. In the future, we hope to incorporate other standard features from functional programming, such as higher-order functions and parametric polymorphism. The combination of these features with regular expression types raises some subtle issues. For function types, we have not found a sensible semantics of types that yields a complete algorithm. For polymorphism, inference of type arguments at type applications is not obvious (there is no unique minimal solution in general).

APPENDIX

A. TRANSLATION OF TYPES

This section presents an algorithm for translating from the external to the internal form of types and proves its soundness, completeness, and termination. (A related algorithm can be found in [Hornung 1996]).

A.1 Algorithm

We first show the translation of values, which is straightforward.

$$\begin{aligned} ts(\texttt{()}) &= \epsilon \\ ts(\texttt{l[v}_1\texttt{],v}_2) &= \texttt{l}(ts(\texttt{v}_1), ts(\texttt{v}_2)) \end{aligned}$$

We now consider the translation of types. Let us first illustrate it by an example. Consider the following external type:

```
a[]*, d[]
```

By expanding the abbreviation `a[]*`, this type is equivalent to `(X, d[])` where:

```
type X = (a[], X) | ()
```

Now, we want to compute the internal type corresponding to `(X, d[])`. For this, we transform it in such a way that all head labels are revealed. We first expand `X` to its definition and then use distributivity of the union operator, associativity of the concatenation operator, and neutrality of the empty sequence[4]:

$$\begin{aligned} & \texttt{(X, d[])} \\ \Longrightarrow\ & \texttt{(((a[], X) | ()), d[])} \\ \Longrightarrow\ & \texttt{(((a[], X), d[]) | ((), d[]))} \\ \Longrightarrow\ & \texttt{((a[], (X, d[])) | d[])} \end{aligned}$$

Since the head labels `a` and `d` are revealed now, this type can be translated to an internal type:

$$\texttt{a}(X_0, X_1) \mid \texttt{d}(X_0, X_0)$$

Since the content of `a` and both the content and the remainder of `d` are all the empty sequence, we can share the internal types corresponding to these by translating them all as the single state X_0 and associating it with ϵ:

$$M(X_0) = \epsilon$$

[4]This technique of revealing head labels is similar to *derivatives* [Brzozowski 1964], which are often used in construction of string automata from regular expressions.

The remainder of `a`, i.e., the type `(X, d[])`, is translated as the state X_1. To associate with this state, we need an internal type corresponding to `(X, d[])`. By transforming it exactly in the same way as above, we obtain `((a[], (X, d[])) | d[])`. This type is then translated to the internal type $\mathtt{a}(X_0, X_1) \mid \mathtt{d}(X_0, X_0)$. Here, note that we can reuse the states X_0 and X_1 since we have already translated the empty sequence to X_0 and the type `(X, d[])` to X_1. Thus, we obtain

$$M(X_1) \;=\; \mathtt{a}(X_0, X_1) \mid \mathtt{d}(X_0, X_0).$$

Before presenting the general translation algorithm, it is convenient to make a small restriction on the syntax of types. In the given type definitions, we make sure that all recursive uses of type names must be preceded by non-nullable type, where a nullable type is one whose denotation contains the empty sequence. For example, both of the type definitions of `X` and `Y` in the following are rejected.

```
type X = X
type Y = a[]*,Y | ()
```

Semantically, these type definitions make sense since we can interpret `X` to be the the empty set and `Y` to be the set of sequences of zero or more `a[]`s. Nonetheless, we reject such type definitions, since handling these would make our formulation complicated, while these have no practical interest. (There are more straightforward ways of writing both the empty set type and the type of sequences of zero or more `a[]`s.)

Formally, we define a function *nohead* that takes a type and a set of type names and answers `true` if and only if all recursive uses of type names in the type are preceded by non-nullable types:

$$\begin{array}{rcl}
\mathit{nohead}(\emptyset)\sigma & = & \mathsf{true} \\
\mathit{nohead}(\mathtt{()})\sigma & = & \mathsf{true} \\
\mathit{nohead}(\mathtt{l[T]})\sigma & = & \mathsf{true} \\
\mathit{nohead}(\mathtt{T}_1 \,|\, \mathtt{T}_2)\sigma & = & \mathit{nohead}(\mathtt{T}_1)\sigma \wedge \mathit{nohead}(\mathtt{T}_2)\sigma \\
\mathit{nohead}(\mathtt{X})\sigma & = & \begin{cases} \mathit{nohead}(E(\mathtt{X}))(\sigma \cup \{\mathtt{X}\}) & (\mathtt{X} \notin \sigma) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathit{nohead}(\emptyset\mathtt{,T})\sigma & = & \mathsf{true} \\
\mathit{nohead}(\mathtt{()\,,T})\sigma & = & \mathit{nohead}(\mathtt{T})\sigma \\
\mathit{nohead}(\mathtt{l[T}_1\mathtt{]\,,T}_2)\sigma & = & \mathsf{true} \\
\mathit{nohead}(\mathtt{(T}_1\mathtt{,T}_2\mathtt{)\,,T}_3)\sigma & = & \mathit{nohead}(\mathtt{T}_1\mathtt{,(T}_2\mathtt{,T}_3\mathtt{)})\sigma \\
\mathit{nohead}(\mathtt{(T}_1 \,|\, \mathtt{T}_2\mathtt{)\,,T}_3)\sigma & = & \mathit{nohead}(\mathtt{T}_1\mathtt{,T}_2)\sigma \wedge \mathit{nohead}(\mathtt{T}_1\mathtt{,T}_3)\sigma \\
\mathit{nohead}(\mathtt{X,T})\sigma & = & \begin{cases} \mathit{nohead}(E(\mathtt{X}),\mathtt{T})(\sigma \cup \{\mathtt{X}\}) & (\mathtt{X} \notin \sigma) \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{array}$$

Then, we require:

$$\mathit{nohead}(E(\mathtt{X}))\emptyset = \mathsf{true} \qquad\qquad \forall \mathtt{X} \in \mathit{dom}(E)$$

The main translation procedure takes an external type $\mathtt{T}_0$ and a set E of type definitions, and computes an internal type T_0 and a tree automaton M. The procedure works by double loops where the outer loop constructs the automaton and the inner loop computes an internal type expression, which will be associated with a state in the automaton. Each state corresponds to an external type expression from which the internal type associated with the state is computed; to make this correspondance explicit, we write each state in the form $X_{\mathtt{T}}$.

The translation function ts ("the inner loop") takes an external type and returns an internal type. The rules below define the function ts.

$$
\begin{array}{rcll}
ts(\emptyset) & = & \emptyset & \text{TR-EMP1} \\
ts(\mathtt{()}) & = & \epsilon & \text{TR-EPS1} \\
ts(\mathtt{l[T]}) & = & \mathtt{l}(X_{\mathtt{T}}, X_{\mathtt{()}}) & \text{TR-LAB1} \\
ts(\mathtt{T}_1\,\mathtt{|}\,\mathtt{T}_2) & = & ts(\mathtt{T}_1) \mid ts(\mathtt{T}_2) & \text{TR-OR1} \\
ts(\mathtt{X}) & = & ts(E(\mathtt{X})) & \text{TR-NM1} \\
ts(\emptyset\mathtt{,T}) & = & \emptyset & \text{TR-EMP2} \\
ts(\mathtt{(),T}) & = & ts(\mathtt{T}) & \text{TR-EPS2} \\
ts(\mathtt{l[T}_1\mathtt{],T}_2) & = & \mathtt{l}(X_{\mathtt{T}_1}, X_{\mathtt{T}_2}) & \text{TR-LAB2} \\
ts(\mathtt{(T}_1\mathtt{,T}_2\mathtt{),T}_3) & = & ts(\mathtt{T}_1\mathtt{,(T}_2\mathtt{,T}_3\mathtt{)}) & \text{TR-ASSOC} \\
ts(\mathtt{(T}_1\,\mathtt{|}\,\mathtt{T}_2\mathtt{),T}_3) & = & ts(\mathtt{T}_1\mathtt{,T}_2) \mid ts(\mathtt{T}_1\mathtt{,T}_3) & \text{TR-OR2} \\
ts(\mathtt{X,T}) & = & ts(E(\mathtt{X}),\mathtt{T}) & \text{TR-NM2}
\end{array}
$$

These rules simply generalize the operations that we have seen in the example. Rule TR-ASSOC uses associativity and rule TR-OR2 uses distributivity. Also, rule TR-EPS2 uses the neutrality of `()`. Rules TR-NM1 and TR-NM2 unfold the type name `X` to their definitions. (Since the syntactic restriction described above ensures that any type name never recurs without non-nullable types in front, rules TR-NM1 and TR-NM2 never unfold the same type name more than once. This ensures that the function ts terminates.) By using all these rules, the head of a sequence is eventually revealed, and turned (by rules TR-LAB1 and TR-LAB2) into an internal label type whose "car" state corresponds to the content type of the label and whose "cdr" state corresponds to the remainder of the sequence. The empty sequence is turned into the leaf transition (by rules TR-EPS1 and TR-LAB2).

The outer loop proceeds as follows. The translation begins with setting M to the empty automaton and computing T_0 from $\mathtt{T}_0$ by using the above function ts. The resulting internal type T_0 may contain some states that are not yet in the domain of M. For each such state $X_{\mathtt{T}}$, we calculate an internal type T from the external type `T` by the ts function and add the mapping $X_{\mathtt{T}} \mapsto T$ to M. We repeat this process until the states appearing in T_0 and M are all in the domain of M. Thus, the result of the whole translation satisfies the following.

$$
\begin{array}{rcll}
T_0 & = & ts(\mathtt{T}_0) & \\
M(X_{\mathtt{T}}) & = & ts(\mathtt{T}) & \forall X_{\mathtt{T}} \in dom(M) \\
dom(M) & \supseteq & St(T_0) \cup St(M). &
\end{array}
$$

A.1.1 Theorem [Soundness and Completeness]: Suppose that a given tree automaton M satisfies $M(X_{\mathtt{T}}) = ts(\mathtt{T})$ for all $X_{\mathtt{T}} \in dom(M)$ and $dom(M) \supseteq St(M)$. If $ts(\mathtt{v}) = t$ and $ts(\mathtt{T}) = T$, then $\mathtt{v} \in [\![\mathtt{T}]\!]$ implies $t \in [\![T]\!]$ and vice versa.

PROOF. By lexicographic induction on the pair of the height $h(t)$ of the tree t and the size of the derivation of $ts(\mathtt{T}) = T$, with the case analysis on the rule used in the derivation. $\square$

A.2 Termination

In a classical algorithm dealing with recursive types, the type expressions encountered by the algorithm are always subphrases of the original type expression. By using the fact that such subphrases are finite in number, termination of such an algorithm can easily be proved. However, the types encountered by our translation algorithm are not necessarily subphrases of the original type, since thealgorithm may restructure types from their original form. For example, consider the following type definitions:

```
type X = a[],X | ()
type Y = X,b[]
```

Let us see the steps involved in the translation of the type `Y`. We first unfold `Y`, which yields `(X,b[])`. The next step unfolds `X`, and yields `(a[],X | ()),b[]`. Note that this type does not appear in the above type definitions. (But note that this type is the *concatenation* of some subphrases appearing there.) Nonetheless, the number of types encountered by the algorithm is finite. Let us continue the translation. We expand the type `(a[],X | ()),b[]` by distributivity, and then proceed to each of the clauses `((a[],X),b[])` and `((),b[])`. The second clause ends up with `b[]`. On the other hand, the first clause transforms to `(a[],(X,b[]))` by associativity and finally leads to `(X,b[])`. Since we have already seen this type, we can stop here. (Notice that the type name `X` was once unfolded before, but `X` was followed by the same type `b[]` at that unfolding as it is now. This is ensured by the well-formedness of the type definitions, which restricts unguarded, recursive uses of type names to be in tail positions.)

For our proof of termination, we devise a special transition system where each transition represents the fact that, if the algorithm encounters the first type expression in one step, then it will encounter the second in a later step. We concentrate on what the algorithm encounters and abstract away what it returns. The reason is that outputs of the translation are never used by the translation itself and therefore termination only depends on inputs to the translation.

The terms over which the transition function is defined are type expressions augmented with several kinds of information that were hidden in the description of the algorithm. First, we separate concatenations introduced by the algorithm from those already in the original type definitions, writing the former with semicolons. Second, we associate each type with a list of type names that records a history of which type names have been unfolded before the algorithm gets to this type. Formally, a *type sequence* K (i.e., a term to transit) has the form $\mathtt{T}_1{}^{\tau_1};\ldots;\mathtt{T}_n{}^{\tau_n}$, where each type is superscripted with a *name list* of the form $\mathtt{X}_1;\ldots;\mathtt{X}_m$. We write $\cdot$ for the empty type sequence and $\emptyset$ for the empty name list. Intuitively, a type sequence of the form $\mathtt{T}_1{}^{\tau_1};\ldots;\mathtt{T}_n{}^{\tau_n}$ corresponds to the type $(\mathtt{T}_1,(\mathtt{..},\mathtt{T}_n))$. In each name list τ_i, the first name was unfolded most recently. The empty type sequence $\cdot$ corresponds to the empty sequence type. We write $\tau \supseteq \tau'$ when $\tau = \tau'';\tau'$ for some τ''. Also, we write $\mathtt{X} \in \tau$ when $\mathtt{X}$ is an element of τ.

We now define the transition relation of the form $K_1 \Rightarrow K_2$, read "if the algorithm encounters the type corresponding to K_1, then it also encounters the type corresponding to K_2." The transition relation is the smallest relation including the following rules.

$$\begin{aligned}
\emptyset; K &\Rightarrow \cdot \\
\mathtt{()}^{\tau}; K &\Rightarrow K \\
\mathtt{Y}^{\tau}; K &\Rightarrow E(\mathtt{Y})^{(\mathtt{Y};\tau)}; K \\
\mathtt{l[T]}^{\tau}; K &\Rightarrow \mathtt{T}^{\emptyset} \\
\mathtt{l[T]}^{\tau}; K &\Rightarrow K \\
(\mathtt{T}_1\mathtt{,T}_2)^{\tau}; K &\Rightarrow \mathtt{T}_1{}^{\tau}; \mathtt{T}_2{}^{\tau}; K \\
(\mathtt{T}_1\mathtt{|T}_2)^{\tau}; K &\Rightarrow \mathtt{T}_1{}^{\tau}; K \\
(\mathtt{T}_1\mathtt{|T}_2)^{\tau}; K &\Rightarrow \mathtt{T}_2{}^{\tau}; K
\end{aligned}$$

In the first rule (corresponding to the algorithm rules TR-EMP1 and TR-EMP2), if we encounter a sequence beginning with the empty sequence, then we also encounter the sequence obtained by removing the empty sequence. Likewise in the second rule (corresponding to the algorithm rules TR-NM1 and TR-NM2), if we encounter a sequence beginning with a type name, then we also encounter a sequence obtained by unfolding the type name. (Note that we record the unfolded type name in the type name list.) Further, in the combination of the third and the fourth rules (corresponding to TR-LAB1 and TR-LAB2), if we encounter a sequence beginning with a label, then we also encounter both its content and its remainder. (For the content, we reset the type name list to the empty, since, when we get to the content T, we freshly consider the transitions from this.) The other rules are analogous.

Our goal is then to show that the set of type sequences reachable from the starting type sequence is finite. The key observation in the proof is that the transition function defined as above preserves the invariant that all $\mathtt{T}_i$s in a type sequence $\mathtt{T}_1{}^{\tau_1}; \ldots; \mathtt{T}_n{}^{\tau_n}$ appear in *different places* in the original type definitions; the type sequences satisfying this invariant are finite in number. To see why this is so, let us consider the above example again. We start with the type sequence $\mathtt{Y}^{\emptyset}$ and transitions go as follows.

$$\mathtt{Y}^{\emptyset} \Rightarrow \mathtt{(X,b[])}^{\mathtt{Y}} \Rightarrow \mathtt{X}^{\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}} \Rightarrow \mathtt{((a[],X)|())}^{\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}}$$

The last type sequence then branches into two chains of transitions: first

$$\begin{aligned}
&\Rightarrow \mathtt{(a[],X)}^{\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}} \Rightarrow \mathtt{a[]}^{\mathtt{X};\mathtt{Y}}; \mathtt{X}^{\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}} \Rightarrow \mathtt{X}^{\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}} \\
&\Rightarrow \mathtt{((a[],X)|())}^{\mathtt{X};\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}}
\end{aligned}$$

(since the last type sequence here is the same as the beginning of the chain, continuing the transition will only yield the same type sequences, modulo differences in type name lists.), and second

$$\Rightarrow \mathtt{()}^{\mathtt{X};\mathtt{Y}}; \mathtt{b[]}^{\mathtt{Y}} \Rightarrow \mathtt{b[]}^{\mathtt{Y}}.$$

Notice that each type sequence appearing in all the above transitions is a sequence of types from different places in the original type definitions.

In order to prove the above-mentioned invariant, we will also need several other invariants. First, a type appearing earlier in any type sequence has an equal or longer type name list than any other type appearing later (since unfolding occurs only in the first type in the sequence). Second, in any type sequence, only the right-most type that has X in its type name list may contain X in tail positions; any other type with X in its type name list cannot contain X at all. Third, in order for induction to work, the condition "appearing different places" has to be slightly strengthened: each type in the type sequence must not be a subphrase of any other type in the sequence "in the same level."

To formalize these invariants, we make a few notational preparations. First, we assume that a unique id is implicitly added to each type expression appearing in the given type definitions or the type $\mathtt{T}_0$ that we want to translate, and that type expressions referred to by the same metavariable name always have the same id. We define the *same-level subphrase* relation, written $\sqsubseteq$, as the smallest reflexive and transitive relation including the following:

$$
\begin{array}{l}
\mathtt{T}_1 \sqsubseteq (\mathtt{T}_1,\mathtt{T}_2)\\
\mathtt{T}_2 \sqsubseteq (\mathtt{T}_1,\mathtt{T}_2)\\
\mathtt{T}_1 \sqsubseteq (\mathtt{T}_1|\mathtt{T}_2)\\
\mathtt{T}_2 \sqsubseteq (\mathtt{T}_1|\mathtt{T}_2)
\end{array}
$$

We write $\Sigma_{\mathtt{X}}$ for the all unguarded (not enclosed by a label) subphrases in the body of $\mathtt{X}$, that is, $\{\mathtt{T} \mid \mathtt{T} \sqsubseteq E(\mathtt{X})\}$. Also, we write Σ for *all* the subphrases (including guarded ones) appearing in the type definitions E and the type $\mathtt{T}_0$; we write $\Sigma_\emptyset$ for all the guarded subphrases in the type definitions, that is, $\Sigma \setminus \bigcup_{\mathtt{X}\in dom(E)} \Sigma_{\mathtt{X}}$.

All the properties of type sequences that we have discussed can be summarized by the following definition of invariants.

A.2.1 Definition [Invariants]: Let $K = \mathtt{T}_1{}^{\tau_1};\ldots;\mathtt{T}_n{}^{\tau_n}$. We call the following the *invariants* for K.

(1) $\mathtt{T}_i \in \Sigma_\emptyset$ if $\tau_i = \emptyset$, and $\mathtt{T}_i \in \Sigma_{\mathtt{Y}}$ if $\tau_i = \mathtt{Y};\tau_i'$.
(2) $\tau_1 \supseteq \ldots \supseteq \tau_n$
(3) For all $\mathtt{X}$, if $\mathtt{X} \in \tau_m$ and $\mathtt{X} \notin \tau_{m+1}$ (or $m = n$), then $\mathtt{X} \notin nontail(\mathtt{T}_m)$ and $\mathtt{X} \notin toplevel(\mathtt{T}_i)$ for all $1 \le i < m$.
(4) $\mathtt{T}_i \not\sqsubseteq \mathtt{T}_j$ for all $i \neq j$.

Here, $nontail(\mathtt{T})$ is the set of non-tail variables of a type $\mathtt{T}$ and $toplevel(\mathtt{T})$ is the set of top-level variables of a type $\mathtt{T}$. The definitions of these are given in Section 3.1.

The main theorem is that transition preserves the invariants for type sequences.

A.2.2 Theorem: If K satisfies the invariants and $K \Rightarrow K'$, then K' satisfies the invariants.

Since Σ is finite, condition 4 ensures that the set of type sequences that satisfy the invariants is finite. Furthermore, we begin the translation with a singleton sequence $\mathtt{T}_0{}^\emptyset$, which trivially satisfies the invariants. Therefore the theorem implies that the translation terminates.

Proof of Theorem A.2.2: Conditions 1 and 2 can easily be checked; the others require a little more work. Let us begin with condition 3.

Case: $()^\tau;K \Rightarrow K$
or $\mathtt{l[T]}^\tau;K \Rightarrow K$

Condition 3 for the lhs immediately implies condition 3 for the rhs.

Case: $\mathtt{l[T]}^\tau;K \Rightarrow \mathtt{T}^\emptyset$

The result holds trivially, since there is no $\mathtt{X}$ that satisfies the condition.

Case: $\mathtt{Y}^{\tau};K \Rightarrow E(\mathtt{Y})^{(\mathtt{Y};\tau)};K$

Let $\tau_0 = \mathtt{Y};\tau$ and $K = \mathtt{U}_1{}^{\tau_1};\ldots;\mathtt{U}_n{}^{\tau_n}$. From condition 2, we have two cases on X.

Subcase: $\mathtt{X} \in \tau_0 \quad \mathtt{X} \notin \tau_1$
or $n = 0$

If $\mathtt{X} \neq \mathtt{Y}$, then we have $\mathtt{X} \notin \mathit{nontail}(\mathtt{Y})$ from condition 3 for the lhs. Therefore $\mathtt{X} \notin \mathit{nontail}(E(\mathtt{Y}))$ by the definition of *nontail*. If $\mathtt{X} = \mathtt{Y}$, then, from well-formedness of E, we immediately obtain $\mathtt{Y} \notin \mathit{nontail}(E(\mathtt{Y}))$.

Subcase: $\mathtt{X} \in \tau_m \quad \mathtt{X} \notin \tau_{m+1} \quad 1 \leq m \leq n$
or $n = m$

If $\mathtt{X} \neq \mathtt{Y}$, then we have $\mathtt{X} \notin \mathit{toplevel}(\mathtt{Y})$ from condition 3 for the lhs. Therefore $\mathtt{X} \notin \mathit{toplevel}(E(\mathtt{Y}))$ by the definition of *toplevel*. If $\mathtt{X} = \mathtt{Y}$, then condition 3 requires $\mathtt{Y} \notin \mathit{toplevel}(\mathtt{Y})$. But this is impossible from the definition of *toplevel*.

Case: $(\mathtt{T}_1,\mathtt{T}_2)^{\tau};K \Rightarrow \mathtt{T}_1{}^{\tau};\mathtt{T}_2{}^{\tau};K$

Let $K = \mathtt{U}_1{}^{\tau_1};\ldots;\mathtt{U}_n{}^{\tau_n}$. From condition 2, we have two cases on X.

Subcase: $\mathtt{X} \in \tau_0 \quad \mathtt{X} \notin \tau_1$
or $n = 0$

Since $\mathtt{X} \notin \mathit{nontail}((\mathtt{T}_1,\mathtt{T}_2))$ from condition 3 for the lhs, $\mathtt{X} \notin \mathit{toplevel}(\mathtt{T}_1)$ and $\mathtt{X} \notin \mathit{nontail}(\mathtt{T}_2)$ follow by the definitions of *nontail* and *toplevel*.

Subcase: $\mathtt{X} \in \tau_m \quad \mathtt{X} \notin \tau_{m+1} \quad 1 \leq m \leq n$
or $n = m$

Conditions $\mathtt{X} \notin \mathit{nontail}(\mathtt{U}_m)$ and $\mathtt{X} \notin \mathit{toplevel}(\mathtt{U}_i)$ for $1 \leq i < m$ immediately hold. Since $\mathtt{X} \notin \mathit{toplevel}((\mathtt{T}_1,\mathtt{T}_2))$ from condition 3 for the lhs, conditions $\mathtt{X} \notin \mathit{toplevel}(\mathtt{T}_1)$ and $\mathtt{X} \notin \mathit{toplevel}(\mathtt{T}_2)$ follow from the definition of *toplevel*.

Case: $(\mathtt{T}_1|\mathtt{T}_2)^{\tau};K \Rightarrow \mathtt{T}_1{}^{\tau};K$
or $(\mathtt{T}_1|\mathtt{T}_2)^{\tau};K \Rightarrow \mathtt{T}_2{}^{\tau};K$

Similar to the previous case.

Finally, we prove condition 4. All the cases are obvious except for

$$\mathtt{Y}^{\tau};K \quad \Rightarrow \quad E(\mathtt{Y})^{(\mathtt{Y};\tau)};K.$$

Let $K = \mathtt{U}_1{}^{\tau_1};\ldots;\mathtt{U}_n{}^{\tau_n}$. The case $\mathtt{Y} \notin \tau_i$ with $i \geq 1$ never happens, because otherwise, condition 3 would require $\mathtt{Y} \notin \mathit{toplevel}(\mathtt{Y})$, which is impossible from the definition of *toplevel*. Therefore the result follows from condition 1. ∎

REFERENCES

Aiken, A. and Murphy, B. R. 1991. Implementing regular tree expressions. In *Proceedings of Functional Programming and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag.

Amadio, R. M. and Cardelli, L. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems 15,* 4, 575–631. Preliminary version in POPL '91 (pp. 104–118); also DEC Systems Research Center Research Report number 62, August 1990.

Brandt, M. and Henglein, F. 1997. Coinductive axiomatization of recursive type equality and subtyping. In *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, R. Hindley, Ed. Lecture Notes in Computer Science (LNCS), vol. 1210. Springer-Verlag, 63–81.

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

BRZOZOWSKI, J. 1964. Derivatives of regular expressions. *J. ACM 11*, 481–494.

BUNEMAN, P., DAVIDSON, S., FERNANDEZ, M., AND SUCIU, D. 1997. Adding structure to unstructured data. In *International Conference on Database Theory*. Springer LNCS 1, 336–351.

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.

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

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

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

COMMON, 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`.

DAMM, F. M. 1994. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software*, M. Hagiya and J. C. Mitchell, Eds. Lecture Notes in Computer Science, vol. 789. Springer-Verlag, 687–706.

DAVIDSON, A., FUCHS, M., HEDIN, M., JAIN, M., KOISTINEN, J., LLOYD, C., MALONEY, M., AND SCHWARZHOF, K. 1999. Schema for object-oriented xml. `http://www.w3.org/TR/NOTE-SOX/`.

DAVIES, R. 2000. Tree automata inclusion. Personal communication.

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`.

DOM 2001. Document object model (DOM). `http://www.w3.org/DOM/`.

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

FERNÁNDEZ, M., SIMÉON, J., AND WADLER, P. An algebra for XML query. `http://www.cs.bell-labs.com/~wadler/topics/xml.html#xalgebra` .

FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*. ACM Press.

GAPEYEV, V., LEVIN, M., AND PIERCE, B. 2000. Recursive subtyping revealed. In *Proceedings of the International Conference on Functional Programming (ICFP)*. 221–232.

GOLDMAN, R. AND WIDOM, J. 1997. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 436–445.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

HORNUNG, T. 1996. Labelled trees and their recognition. In *Publ. Math. Debrecen*. Number 3–4 in 48. 309–316.

HOSOYA, H. AND PIERCE, B. C. 2000. XDuce: A typed XML processing language. 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.

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, UK.

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

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. 2000. Hedge automata: a formal model for XML schemata. `http://www.xml.gr.jp/relax/hedge_nice.html`.

Murata, M. 2001. RELAX (REgular LAnguage description for XML). `http://www.xml.gr.jp/relax/`.

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.

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.

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 collaborators in the XDuce project are Peter Buneman 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. Comments from the ICFP referees helped improve the presentation significantly.

This work was supported by the Japan Society for the Promotion of Science (Hosoya), the University of Pennsylvania's Institute for Research in Cognitive Science (Vouillon), and the National Science Foundation under NSF Career grant CCR-9701826 (Pierce).