Regular Expression Types for XML

Haruo Hosoya Jérôme Vouillon Benjamin C. Pierce

Department of Computer and Information Science
University of Pennsylvania
{hahosoya,vouillon,bcpierce}@saul.cis.upenn.edu

ABSTRACT

We propose *regular expression types* as a foundation for XML processing languages. Regular expression types are a natural generalization of Document Type Definitions (DTDs), describing structures in XML documents using regular expression operators (i.e., `*`, `?`, `|`, etc.) and supporting a simple but powerful notion of *subtyping*.

The decision problem for the subtype relation is EXPTIME-hard, but it can be checked quite efficiently in many cases of practical interest. The subtyping algorithm developed here is a variant of Aiken and Murphy's set-inclusion constraint solver, to which are added several optimizations and two new properties: (1) our algorithm is provably complete, and (2) it allows a useful "subtagging" relation between nodes with different labels in XML trees.

1. INTRODUCTION

The recent rush to adopt XML is due in part to the hope that the static typing provided by DTDs [22] (or more sophisticated mechanisms such as XML-Schema [23]) will improve the safety of data exchange and processing. However, although XML *documents* can be checked for conformance with DTDs, current XML processing languages offer no way of verifying that *programs* operating on these documents will always produce conforming outputs.

In this paper, we propose *regular expression types* as a foundation for statically typed processing of XML documents. Regular expression types are a natural generalization of DTDs, describing, as DTDs do, structures in XML documents using regular expression operators (i.e., `*`, `?`, `|`, etc.). Moreover, regular expression types support a simple but powerful notion of *subtyping*.

We have used regular expression types in the design of a domain-specific language called XDuce ("transduce") for XML processing [15, 14]. In the present paper, though, 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://www.cis.upenn.edu/~hahosoya/xduce.html`

for more information on the language as a whole.

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

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

corresponding to the following DTD:

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

In our syntax, type constructors of the form `label[...]` classify nodes ("elements" in XML jargon) with the label `label`, i.e., XML structures of the form `<label>...</label>`. Types may also involve the regular expression operators `*` (repetition) and `?` (optionality), as well as `|` (alternation), which we'll see examples of later. Here, the type `Addrbook` describes a label `addrbook` containing zero or more repetitions of nodes with label `person`. Each `person` contains a sequence of nodes labeled `name`, `addr`, and optionally `tel`, each of which contains a `String` (called `#PCDATA` in the DTD syntax). An instance of the type `Addrbook` is the following XML document:

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

In general, a type denotes a set of documents. Our notion of subtyping is simply inclusion between the sets denoted by two types. The flexibility provided by this form of subtyping turns out to be quite useful in programming. For example, this definition allows the following subtyping:

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

This subtyping gives a view of the type `Person*` as the union of sequences that have at least one `person` with `tel` and sequences that have no `persons` with `tel`. This inclusion can be used to trivially check the exhaustiveness of a `case` statement with two cases corresponding to the two clauses of the right-hand type. Notice that if we interpreted the `Person*` type as an ML-like type

```
{person: {name: String,
          addr: String,
          tel: String option}} list
```

the conventional subtyping of variant and record types would *not* yield an inclusion like the one above. We give further examples of the usefulness of this kind of flexible subtyping in Section 2.

The main difficulty that we must face in programming with regular expression types is that the decision problem for subtyping is algorithmically difficult: types are essentially tree automata [10] and thus the subtyping problem reduces to deciding language inclusion for tree automata, which is known to be EXPTIME-complete [20].[1]

We have developed an algorithm for subtype checking that works quite efficiently on examples from the XML processing domain. Our algorithm can be viewed as an adaptation of Aiken and Murphy's algorithm for set-inclusion constraint solving [1], with two important extensions. First, our algorithm is complete, as we prove in Section 4. While completeness is not critical in Aiken and Murphy's context of program analyses for optimization, it is crucial here for generating comprehensible error messages in case of typechecking failure. Second, we allow "subtagging" between labels, supporting a useful object-oriented programming idiom.

We have incorporated a number of optimization techniques in our implementation of the subtyping algorithm. These range from standard techniques such as hash consing to set-theoretic optimizations arising from our observations of the actual uses of subtyping in XML processing. Our algorithm runs at reasonable speed even on XDuce applications that involve quite large types, such as the complete DTD for HTML documents.

The contributions of this paper can be summarized as follows:

- We motivate the use of regular expression types, set-inclusion-based subtyping, and subtagging for the domain of XML processing.
- We formalize the connection of regular expression types to tree automata and develop a subtyping algorithm, giving soundness, completeness, and termination proofs.
- We outline several optimizations specialized to the domain of XML processing and present preliminary measurements of their practical effects.

[1]Strictly speaking, what is known is that our subtyping problem *without subtagging* is EXPTIME-complete. We conjecture that this is also the case with subtagging.

The paper is organized as follows. In the next section, we give some examples of programming with regular expression types. In Section 3, we give the connection of regular expression types to tree automata and the definition of subtyping. In Section 4, we present our subtyping algorithm and prove its correctness. Section 5 describes our implementation techniques and Section 6 discusses some performance measurements. We survey related work in Section 7 and conclude in Section 8.

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 `Tels`. 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 set of 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 for the formal definition.)

We now 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 relations:

```
    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 relations with the label `person` and combining these with the third relation, we obtain:

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

Finally, enclosing both sides by `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*)`. More interestingly, `T*` itself can also be derived using recursion. That is, `T*` is equal to a variable `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

Yet more interesting types can be built using recursion to express nesting of 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 `bad` 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 "equivalence classes" can be found in XML-Schema (Section 3.5 in [23]—so called even though it does not yield a symmetric relation between types).

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 subtyping 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 case for `fontstyle`, when both cases 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. In the XDuce language, the special label `~` denotes "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:

```
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]*),
       x:(person[Name,Addr,Tel]),
       _:(Person*)
     →  (* do some stuff *)
  | y:(person[Name,Addr]*)
     →  (* do other stuff *)
```

A pattern of the form `x:T` matches any value of type `T` and binds the variable `x` to this value. (Underscore is used as a "don't care" variable name, as in ML.) 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.

However, we can use subtyping also 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.

2.7 Semistructured Data

One of the main application domains for XML is representing and transmitting databases. Since this flexible representation allows for straightforward 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>
```

with 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 record 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 outputting the new type. 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 is helping 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 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 that all the `Person`s come before the `Person2`s:

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

Now, each element has either type `Person` or else type `Person2`, leading naturally to one-loop scans. 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, compared to the "tagged" sum types (as found, for example, in ML and Haskell).

3. DEFINITIONS

We 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, the subtyping algorithm uses a simpler representation to streamline both the implementation and its accompanying correctness proofs. We first give the syntax of each form. Next, we describe the translation from the external form to the internal form. Then we give the definition of subtyping in terms of the internal form.

3.1 External Form

We assume a countably infinite set of labels, ranged over by `l`, and a countably infinite set of variables, ranged over by `X`. Type expressions are then defined as follows.

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

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 sometimes regard E as a mapping from type variables to their bodies. We write $dom(E)$ for the set of defined variables.

Base types such as `String` and `Int` are actually treated as special labels: `String` is an abbreviation for `String[]` and `Int` for `Int[]`. (This means that, strictly speaking, values of these types are also labels: for example, the elements of `Int[]` include `1[]`, `2[]`, etc., where `1` and `2` are built-in sub-tags of `Int`.)

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

Since the decision problem for inclusion between context free languages is undecidable [13], 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 [13], so we adopt a simple syntactic condition, called *well-formedness*, that ensures regularity. Intuitively, well-formedness allows recursive uses of variables to occur only in tail positions. For example, we allow the following type definitions:

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

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

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

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

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

3.2 Internal Form

Transitions or *internal types* are defined by the following:

$$\begin{array}{lll}
A \; ::= & \epsilon & \text{leaf transition} \\
 & l(X, X) & \text{branch transition} \\
 & A \mid A & \text{alternation} \\
 & \emptyset & \text{empty transition}
\end{array}$$

$FV(A)$ is the set of states appearing in A. A *tree automaton* or *context* is a finite mapping Δ from states X_i to transitions A_i. Any state occurring in the transitions in the context must be one of its states: $FV(\Delta(X)) \subseteq dom(\Delta)$ for $X \in dom(\Delta)$.

The semantics of internal types are defined in terms of *ground types*, which are defined by the following grammar:

$$\begin{array}{lll}
t \; ::= & \epsilon & \text{leaf} \\
 & l(t, t) & \text{branch}
\end{array}$$

Intuitively, ϵ corresponds to the empty sequence; $l(t, t')$ corresponds to a sequence whose head is a label `l[T]` where t corresponds to `T` and t' corresponds to the remainder of the sequence. Ground types can be seen as abstractions of sets of concrete values. That is, each concrete value "directly" belongs to a ground type with the same structure. For example, the value

```
<name>Haruo</name> <addr>Tokyo</addr>
```

has external type

```
name[String[]], addr[String[]]
```

from which we can read off the ground type:

$$\mathtt{name(String(}\epsilon,\epsilon\mathtt{),\ addr(String(}\epsilon,\epsilon\mathtt{),\ }\epsilon\mathtt{)).}$$

A transition denotes a set of ground types. The denotation function $[\![A]\!]_\Delta$ is defined as the least solution of the following set of equations:

$$\begin{array}{rcl} [\![\epsilon]\!]_\Delta & = & \{\epsilon\} \\ [\![l(X,X')]\!]_\Delta & = & \{l'(t,t') \mid l' \prec l \wedge \ t \in [\![\Delta(X)]\!]_\Delta \wedge \\ & & \qquad\qquad\qquad t' \in [\![\Delta(X')]\!]_\Delta\} \\ [\![A \mid A']\!]_\Delta & = & [\![A]\!]_\Delta \cup [\![A']\!]_\Delta \\ [\![\emptyset]\!]_\Delta & = & \emptyset \end{array}$$

These rules are straightforward except that a branch transition $l(X, X')$ denotes not only branch ground types with label l but also all labels l' smaller than l in the sense of subtagging. (Viewed another way, the denotation function defines the set of trees accepted by a tree automaton.)

For later use in the subtyping algorithm, we define the *structural equivalence* $\equiv$ between transitions as the smallest relation containing the following:

$$\begin{array}{rcll} A \mid \emptyset & \equiv & A & \\ A \mid A & \equiv & A & \\ (A \mid B) \mid C & \equiv & A \mid (B \mid C) & \\ A \mid B & \equiv & B \mid A & \\ A \mid B & \equiv & A \mid C & \quad \text{if } B \equiv C \end{array}$$

These rules essentially treat a type as a *set* of (leaf or branch) transitions. We can easily prove that $[\![A]\!]_\Delta = [\![B]\!]_\Delta$ whenever $A \equiv B$.

In the remainder of the paper, we assume that a single context Δ is given once and for all and shared among all types. Therefore we simply write $[\![A]\!]$ to mean $[\![A]\!]_\Delta$. In examples and informal explanations, we sometimes identify a state and its associated transition. For instance, we write $l(B, C)$ for $l(X, Y)$ where $B = \Delta(X)$ and $C = \Delta(Y)$.

3.3 Translation

Given an external type $\mathtt{T}_0$, translation constructs a pair of an internal type and a context. The context associates each state X with an internal type corresponding to some external type T. For ease of formulation, we write such states as $X_\mathtt{T}$.

At each step, we compute an internal type corresponding to an external type using the function *ts* defined below.

$$\begin{array}{rcll} ts(\mathtt{()})\rho & = & \epsilon & \\ ts(\mathtt{l[T]})\rho & = & \mathtt{l}(X_\mathtt{T}, X_{\mathtt{()}}) & \\ ts(\mathtt{X})\rho & = & \emptyset & \mathtt{X} \in \rho \\ ts(\mathtt{X})\rho & = & ts(E(\mathtt{X}))(\rho \cup \{\mathtt{X}\}) & \mathtt{X} \notin \rho \\ ts(\mathtt{T}_1\mathtt{|T}_2)\rho & = & ts(\mathtt{T}_1)\rho \mid ts(\mathtt{T}_2)\rho & \\ & & & \\ ts(\mathtt{(),T})\rho & = & ts(\mathtt{T})\rho & \\ ts(\mathtt{l[T}_1\mathtt{],T}_2)\rho & = & \mathtt{l}(X_{\mathtt{T}_1}, X_{\mathtt{T}_2}) & \\ ts(\mathtt{X,T})\rho & = & \emptyset & (\mathtt{X,T}) \in \rho \\ ts(\mathtt{X,T})\rho & = & ts(E(\mathtt{X}),\mathtt{T})(\rho \cup \{(\mathtt{X,T})\}) & \\ & & & (\mathtt{X,T}) \notin \rho \\ ts(\mathtt{(T}_1\mathtt{,T}_2\mathtt{),T}_3)\rho & = & ts(\mathtt{T}_1\mathtt{,(T}_2\mathtt{,T}_3\mathtt{)})\rho & \\ ts(\mathtt{(T}_1\mathtt{|T}_2\mathtt{),T}_3)\rho & = & ts(\mathtt{T}_1\mathtt{,T}_2)\rho \mid ts(\mathtt{T}_1\mathtt{,T}_3)\rho & \end{array}$$

This function collects all labels appearing at the head of a sequence and turns them into branch transitions whose "car" state corresponds to the content type of the label and whose "cdr" state corresponds to the remainder of the sequence. These branch transitions contain states $X_\mathtt{T}$, for which the associated external type T will be the target of translation in the next step. The function *ts* also collects the leaf transition if the given type contains the empty sequence. We keep track of which types we have already seen, in the set ρ of types. When we encounter the same type for the second time, the result is the empty set. For example, suppose we have the type definition `type X = X` and we want to translate `(X,Int)`. The corresponding internal type is $ts(\mathtt{X,Int})\emptyset = ts(\mathtt{X,Int})\{(\mathtt{X,Int})\} = \emptyset$, which is reasonable since both X and therefore `(X,Int)` denote the empty set (because we are interested in the least solution of recursive type definitions). In the rule before the last, we turn a left associative concatenation to right associative, thus eventually revealing the head of a sequence.

The whole translation procedure takes as input an external type $\mathtt{T}_0$ and returns an internal type A_0 and a context Δ. Initially, Δ is set to the empty context, and we start with computing the internal type A_0 for the external type $\mathtt{T}_0$: $A_0 = ts(\mathtt{T}_0)\emptyset$. Some branch transitions in the result may contain states that are not yet in Δ. For such a state $X_\mathtt{T}$, we compute its internal type $A_{X_\mathtt{T}} = ts(\mathtt{T})\emptyset$ and add the mapping $X_\mathtt{T} \mapsto A_{X_\mathtt{T}}$ to Δ. We repeat this process until all transitions in A_0 and Δ contain states that are in Δ.

The intuition behind the termination of translation is that all external types encountered by the process (i.e., given to the function *ts*) have the form of a sequence $\mathtt{T}_1\mathtt{,..,T}_n$ where each $\mathtt{T}_i$ appears in a different place in the original type definitions; the set of such sequences is finite. A more formal discussion of this point can be found in the accompanying technical report [16].

3.4 Subtyping

The subtyping relation between two types is defined semantically: two internal types are in the *subtype* relation iff their denotations are in the *subset* relation:

$$A <: A' \text{ iff } [\![A]\!] \subseteq [\![A']\!].$$

4. SUBTYPING ALGORITHM

This section develops an algorithm for deciding the subtyping relation.

4.1 Highlights

Our algorithm is a top-down one similar in spirit to standard subtyping algorithms. We start with a pair of types, and, at each step, we generate one or more subgoals until we reach leaf goals involving only trivial checks.

The main difficulty of constructing such a top-down algorithm arises from the "untaggedness" of union types. Specifically, the most interesting case is when we have union types on the right-hand side:

$$l(A,B) \quad <: \quad l(C,D) \;\mid\; l(E,F)$$

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

$$\textsc{(Weak-Rec)}\quad \frac{l(A,B) <: l(C,D) \quad \text{or} \quad l(A,B) <: l(E,F)}{l(A,B) <: l(C,D) \mid l(E,F)}$$

However, this rule is too weak. For example, for checking $l((C \mid E), D) <: l(C, D) \mid l(E, D)$, neither premise holds. A more realistic example is the subtyping relation given in Section 2.6:

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

Aiken and Murphy [1] actually adopt a rule similar to WEAK-REC and argue that they rarely see cases where this rule is not enough. In our setting, however, such cases do arise in practice.

Another rule we might expect to see is one that distributes all unions over labels. For example, to verify $l((C \mid E), D) <: l(C, D) \mid l(E, D)$, we could transform the left-hand side to $l(C, D) \mid l(E, D)$ and 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, a simple set-theoretic observation leads to a solution to this difficulty. Let us consider a slightly more general case:

$$l(A, B) \quad <: \quad l(C_1, D_1) \mid l(C_2, D_2) \mid l(C_3, D_3)$$

First, in general, a cross product $X \times Y$ is equal to $(X \times \mathcal{T}) \cap (\mathcal{T} \times Y)$ where the maximal type $\mathcal{T}$ denotes the set of all ground types. Therefore the right-hand side of the subtyping relation can be rewritten as follows.

$$\begin{array}{ll} & (l(C_1, \mathcal{T}) \cap l(\mathcal{T}, D_1)) \\ \mid & (l(C_2, \mathcal{T}) \cap l(\mathcal{T}, D_2)) \\ \mid & (l(C_1, \mathcal{T}) \cap l(\mathcal{T}, D_1)) \end{array}$$

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

$$\begin{array}{l} (l(C_1, \mathcal{T}) \mid l(C_2, \mathcal{T}) \mid l(C_3, \mathcal{T})) \cap \\ (l(\mathcal{T}, D_1) \mid l(C_2, \mathcal{T}) \mid l(C_3, \mathcal{T})) \cap \\ (l(C_1, \mathcal{T}) \mid l(\mathcal{T}, D_2) \mid l(C_3, \mathcal{T})) \cap \\ (l(\mathcal{T}, D_1) \mid l(\mathcal{T}, D_2) \mid l(C_3, \mathcal{T})) \cap \\ (l(C_1, \mathcal{T}) \mid l(C_2, \mathcal{T}) \mid l(\mathcal{T}, D_3)) \ \cap \ \ldots \end{array}$$

In each clause of the conjunctive form, if C_i appears, then the corresponding argument D_i does not appear, and vice versa. Therefore each clause can be rewritten as

$$(|_{i \in I} \, l(C_i, \mathcal{T})) \ \mid \ \left(|_{i \in \overline{I}} \, l(\mathcal{T}, D_i)\right)$$

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

$$l(A, B) \quad <: \quad (|_{i \in I} \, l(C_i, \mathcal{T})) \ \mid \ \left(|_{i \in \overline{I}} \, l(\mathcal{T}, D_i)\right)$$

or equivalently,

$$l(A, B) \quad <: \quad l(|_{i \in I} \, C_i, \mathcal{T}) \ \mid \ l(\mathcal{T}, |_{i \in \overline{I}} \, D_i).$$

Write C for $|_{i \in I} \, C_i$ and D for $|_{i \in \overline{I}} \, D_i$.

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

$$A <: C \quad \text{or} \quad B <: D.$$

To see why, suppose $l(A, B) <: l(C, \mathcal{T}) \mid l(\mathcal{T}, D)$ but $A \not<: C$ and $B \not<: D$. We can find trees $t \in A \backslash C$ and $u \in B \backslash D$. This means that $l(t, u) \in l(A, B)$ but neither $l(t, u) \in l(C, \mathcal{T})$ or $l(t, u) \in l(\mathcal{T}, D)$, which contradicts the assumption.

This discussion can be further generalized to cases where the subtype relation to check has arbitrary number of clauses on the right-hand side.

There are two sources of computational inefficiency here. One is the exponential blow-up involved in considering all the subsets I of $\{1, \ldots, n\}$. The other is the backtracking incurred by checking whether one or the other of the two conditions above holds. Many of the optimizations described in Section 5.2 are intended to avoid dealing with these general cases, whenever possible.

4.2 Algorithm

The subtyping algorithm is defined by two forms of judgment $\Gamma \vdash A <: B \Rightarrow \Gamma'$ and $\Gamma \vdash^* A <: B \Rightarrow \Gamma'$, where Γ is a set of pairs of types of the form $C <: D$, called "assumptions." Both judgments should be read: "assuming that all relations $C <: D$ in Γ hold, the algorithm proves $A <: B$ and possibly returns additional pairs $E <: F$ in the output set Γ' that have been proved in this process."

As in standard algorithms for subtyping recursive types, we store the given pair of types at each step to the assumption set. Later on if we encounter the same pair, we stop going further, thus ensuring termination. We have to be careful not to check the assumption set immediately after storing the given pair, which would invalidly prove subtyping between any pair of types. This is why we have two different judgments.

The accumulated assumptions are eventually returned as the output set Γ', which is propagated as the input to other subtype checks, avoiding repeated checks of the same pair. Furthermore, it is reused not only in the process of a single subtype checking but also all the way through the compilation of the whole program, thus serving as a "cache" of all verified subtype relations. (See [3, 6, 12] for further discussion.)

We now present the rules for the subtyping algorithm. We write $(A <: B) \in_{\equiv} \Gamma$ for set membership up to structural equivalence: $(A' <: B') \in \Gamma$ for some $A' \equiv A$ and $B' \equiv B$.

(HYP)
$$\frac{(A <: B) \in_{\equiv} \Gamma}{\Gamma \vdash A <: B \Rightarrow \Gamma}$$

(ASSUM)
$$\frac{(A <: B) \notin_{\equiv} \Gamma \qquad \Gamma; A <: B \vdash^* A <: B \Rightarrow \Gamma'}{\Gamma \vdash A <: B \Rightarrow \Gamma'}$$

If the pair $A <: B$ of input types is already in the set Γ of assumptions, we immediately succeed (rule HYP). Otherwise, we add the pair to this set (rule ASSUM). These two rules ensure termination as well as avoidance of repeated checks of the same pair. In ASSUM, we switch from the judgment of the form $\Gamma \vdash A <: B \Rightarrow \Gamma'$ to $\Gamma \vdash^* A <: B \Rightarrow \Gamma'$, preventing the incorrect application of the rule HYP immediately following ASSUM. We keep using the judgment $\Gamma \vdash^* A <: B \Rightarrow \Gamma'$ in the subsequence rules, and switch back to the judgment $\Gamma \vdash A <: B \Rightarrow \Gamma'$ in the last rule REC below.

The remaining rules depend on the shapes of the input types. The first three are simple:

(EMPTY)
$$\frac{}{\Gamma \vdash^* \emptyset <: A \Rightarrow \Gamma}$$

(SPLIT)
$$\frac{\Gamma \vdash^* A <: B \Rightarrow \Gamma' \qquad \Gamma' \vdash^* A' <: B \Rightarrow \Gamma''}{\Gamma \vdash^* A \mid A' <: B \Rightarrow \Gamma''}$$

(LEAF)
$$\frac{A \equiv \epsilon \mid A'}{\Gamma \vdash^* \epsilon <: A \Rightarrow \Gamma}$$

If the left-hand side is the empty set, we simply return since the relation clearly holds (rule EMPTY). If the left-hand side can be split into the union of two sets A and A', we generate two subgoals for these sets (rule SPLIT). The intuition behind this rule is a set-theoretic property: $A \cup A' \subseteq B$ iff $A \subseteq B$ and $A' \subseteq B$. (There may be many ways to split the left-hand side into the union of two sets; the algorithm chooses one of them, non-deterministically.) If the left-hand side is a singleton set with a leaf transition, we check that the right-hand side also contains a leaf transition (rule LEAF). The algorithm can fail only at this rule.

In the remaining cases, the left-hand side has a single branch transition. First, define *card* as follows:

$$\begin{array}{lcll} card(A_1 \mid A_2) & = & card(A_1) + card(A_2) & \\ card(A) & = & 1 & \text{if } A = \epsilon \text{ or } l(X, X') \text{ or } \emptyset \end{array}$$

Now:

(PRUNE)

$$\frac{C \equiv l'(B, B') \mid C' \qquad card(C') < card(C) \qquad l \not\prec l' \qquad \Gamma \vdash^* l(A, A') <: C' \Rightarrow \Gamma'}{\Gamma \vdash^* l(A, A') <: C \Rightarrow \Gamma'}$$

(PRUNE-LEAF)

$$\frac{C \equiv \epsilon \mid C' \qquad card(C') < card(C) \qquad \Gamma \vdash^* l(A, A') <: C' \Rightarrow \Gamma'}{\Gamma \vdash^* l(A, A') <: C \Rightarrow \Gamma'}$$

(REC)

$$\frac{\begin{array}{c}\text{For all } 1 \le j \le n,\ l \prec l_j \\ A = \Delta(X) \quad A' = \Delta(X') \quad B_j = \Delta(Y_j) \quad B'_j = \Delta(Y'_j) \\ \text{For all } 1 \le i \le 2^n, \text{ either} \\ \Gamma_{i-1} \vdash A <: |_{j \in I^n_i} B_j \Rightarrow \Gamma_i \text{ or } \Gamma_{i-1} \vdash A' <: |_{j \in \overline{I^n_i}} B'_j \Rightarrow \Gamma_i\end{array}}{\Gamma_0 \vdash^* l(X, X') <: l_1(Y_1, Y'_1) \mid \ldots \mid l_n(Y_n, Y'_n) \Rightarrow \Gamma_{2^n}}$$

The rules PRUNE and PRUNE-LEAF remove from the right hand side a leaf transition ϵ and all branch transitions with labels l' that are not greater than l. The side condition $card(C') < card(C)$ ensures progress by guaranteeing that the algorithm proceeds with smaller transitions.

The rule REC handles the other cases: 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. This rule formalizes the intuition explained in Section 4.1. We index the subsets of $\{1, 2, \ldots, n\}$ in some arbitrary order from I^n_1 to $I^n_{2^n}$. We write $\overline{I^n_i}$ for the complement of such a subset. For each index i, we prove that *either* A is a subtype of the union $|_{j \in I^n_i} B_j$, *or* A' is a subtype of the union $|_{j \in \overline{I^n_i}} B'_j$.

In Figure 1, we show the proof tree for checking the subtyping relation $X <: Y$ where the context is:

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

In the proof tree, we write Γ for $\{X <: Y\}$. Notice that when $X <: Y$ is checked for the second time, HYP is used, thus ensuring termination. Also, checking $\Gamma \vdash^* l(X, \epsilon) <: l(Y, \epsilon)$ generates a complex form of premises, some of which fail, incurring backtracking.

This algorithm is sound and complete.

4.2.1 Theorem [Soundness]: If $\Gamma \vdash A <: B \Rightarrow \Gamma'$ and Γ is consistent (i.e., $C <: D$ for all $(C <: D) \in_{\equiv} \Gamma$), then $A <: B$ and Γ' is also consistent.

4.2.2 Theorem [Completeness]: If $A <: B$, then $\Gamma \vdash A <: B \Rightarrow \Gamma'$ for some Γ'.

The proofs of these theorems (and the termination property below) are given in our accompanying technical report [16].

The termination of the algorithm can be seen as follows. The process from EMPTY to PRUNE-LEAF obviously terminates because the sizes of the types always decrease. In REC, observe that X, X', $Y_1, \ldots, Y_n$, $Y'_1, \ldots, Y'_n$ are always some states of the tree automata of the input types. Since the set of states is finite, the set of *unions* of states, for which REC generate subgoals, is also finite. ASSUMP keeps track of all such unions of states that the algorithm encounters, and HYP ensures termination.

4.2.3 Theorem [Termination]: For all Γ, A, and B, the algorithm either proves $\Gamma \vdash A <: B \Rightarrow \Gamma'$ for some Γ' or fails.

5. IMPLEMENTATION

Our implementation of subtyping consists of the set of rules given in the previous section plus a number of optimization techniques specialized to the subtyping problems that arise in the XML domain. This section describes some of these techniques. The techniques are categorized into low-level, representational techniques and higher-level heuristics inspired by set-theoretic observations. Our implementation is written in O'Caml [17].

5.1 Low-level Optimizations

In order to make subtyping faster, it is crucial to minimize the number of internal types translated from the external types. Observing that semantically equal external types can be translated to the same internal type, we exploit the properties of alternation and comma operators. Since alternation is commutative, associative, and idempotent, an alternation of types is represented as a set of types. In addition, nested alternations are flattened so that `((R|S)|(T|U))` and `(U|(S|(T|R)))` can be recognized as equal, for example. Since the translation uses only union and equality operations on sets, a suitable representation is sorted lists, which allows us to perform these two operations in linear time. (We use a similar representation for internal types.) On the other hand, since the comma operator is associative, a sequence of types is simply represented as a list of types, with nested sequences flattened and the empty sequence removed so that `(((R,()),S),(T,U))` and `(R,(S,(T,U)))` can be recognized as equal. We also use *hash consing* to make equality tests faster.

We need to be a little careful about the representation of a set Γ of assumptions, described in Section 4. A hash table might appear a suitable representation for this, but it is not the case. Recall that in REC, we have the premises "$A <: \ldots$ or $A' <: \ldots$." For implementing this, when the first premise fails, what we have to pass to the second premise is the original set of assumptions, not what the first premise would return as an "output set of assumptions," which may contain wrong assumptions. Therefore, if we used a hash table for Γ, we would have to copy the whole hashtable every

$$
\dfrac{
\dfrac{\text{LEAF}}{\Gamma \vdash^* \epsilon <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Gamma}
\qquad
\dfrac{
\dfrac{
\left(\dfrac{\text{HYP}}{\Gamma \vdash X <: Y \Rightarrow \Gamma} \ \text{or}\ \dfrac{\textit{fail}}{\Gamma \vdash \epsilon <: \emptyset} \right)
\ \text{and}\
\left(\dfrac{\dfrac{\textit{fail}}{\vdots}}{\Gamma \vdash X <: \emptyset} \ \text{or}\ \dfrac{\text{LEAF}}{\Gamma \vdash \epsilon <: \epsilon \Rightarrow \Gamma} \right)
}{\Gamma \vdash^* l(X,\epsilon) <: l(Y,\epsilon) \Rightarrow \Gamma}
}{\Gamma \vdash^* l(X,\epsilon) <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Gamma}
}{
\dfrac{\Gamma \vdash^* \epsilon \mid l(X,\epsilon) <: \epsilon \mid l(Y,\epsilon) \Rightarrow \Gamma}{\vdash X <: Y \Rightarrow \Gamma}
}
$$

Figure 1: Example proof tree

time before checking the first premise, which would be very expensive. Instead, we use a functional representation of sets, where the required operations here (unlike the unions of types described above), are insertion and membership. Therefore 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 Optimizations

The rule REC is potentially costly as it may generate exponentially many subgoals and involve backtracking. Our heuristic rules are mainly intended to avoid the application of this rule as much as possible, or at least to simplify its applications.

5.2.1 Physical equality

In an implementation of subtyping for any type system, the most trivial optimization is checking physical equality before going deeply in the structure. In the presence of union types, this can be slightly generalized, using the fact that $A <: A \mid B$. In our implementation, we use the following rule

$$
\text{(TRIV)}\quad \frac{\Gamma \vdash B <: A \mid C \Rightarrow \Gamma'}{\Gamma \vdash A \mid B <: A \mid C \Rightarrow \Gamma'}
$$

which can be seen as a combination of SPLIT and $A <: A \mid B$. (We try this rule before HYP.) Notice that this rule simply takes the difference between two sets of transitions. Since a type is represented as a sorted list, this operation is cheap.

5.2.2 Empty type elimination

In this optimization, we eliminate all types denoting the empty set before starting the subtyping algorithm. Doing so allows the algorithm to short-cut some tests later, since it can assume that any type it encounters is not empty. In particular, the algorithm now does not have to generate two goals of the form $A <: \emptyset$ at each application of the rule REC. In such cases, it is fairly common that the right hand side is just a single branch transition: $l(A,B) <: l'(C,D)$. When this happens, we only need to check $A <: C$ and $B <: D$ with $l \prec l'$, which does not involve backtracking at all.

Empty type elimination can be highly tuned. In theory, identifying and eliminating empty states can be performed in linear time [10]. However, we use a simpler but potentially quadratic algorithm, which perform better in practice. (We refer to this optimization later by EMP.)

5.2.3 Merging similar transitions

In order to make the previous optimization more effective, we merge transitions on the right hand side when either the first or second arguments are the same:

$$
\text{(MERGE1)}\quad \frac{\Gamma \vdash^* l(A,B) <: l'(C,(D \mid E)) \Rightarrow \Gamma' \qquad l \prec l'}{\Gamma \vdash^* l(A,B) <: l'(C,D) \mid l'(C,E) \Rightarrow \Gamma'}
$$

$$
\text{(MERGE2)}\quad \frac{\Gamma \vdash^* l(A,B) <: l'((C \mid D),E) \Rightarrow \Gamma' \qquad l \prec l'}{\Gamma \vdash^* l(A,B) <: l'(C,E) \mid l'(D,E) \Rightarrow \Gamma'}
$$

(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. (If we "import" DTDs as regular expression types, labels of the same name are *required* to have the same content type.) Therefore we check MERGE1 first and then MERGE2.

5.2.4 Default case

Finally, if the type on the right hand side in REC has the form $B \mid C_1 \mid \ldots \mid C_n$ and B is larger than any C_i, then we only need to compare the left hand side with this largest type B:

$$
\text{(SUPER)}\quad \frac{1 \le i \le n \qquad \Gamma_i \vdash^* C_i <: B \Rightarrow \Gamma_{i+1} \qquad \Gamma_{n+1} \vdash^* A <: B \Rightarrow \Gamma'}{\Gamma_1 \vdash^* A <: B \mid C_1 \mid \ldots \mid C_n \Rightarrow \Gamma'}
$$

(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 so many subgoals in the course of searching for the largest type that the cost surpasses the gain. However, we have not found such a situation in practice so far. This is probably because most of the cases are handled by the previous optimizations, and because the alternative rule REC is often more expensive.

6. PERFORMANCE MEASUREMENT

We have incorporated the subtyping algorithm described in the previous sections in a prototype implementation of the XML processing language XDuce. XDuce is a simple first-order functional language; a typical XDuce program consists

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 1: Applications and Measurement Results

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 the wall-clock time that our algorithm takes to typecheck various application programs (subtype checks consume most of this time). The accompanying technical report shows additional experimental results on the separate effects of each high-level optimization [16]. The platform for our experiment is a Sun Enterprise 3000 (250MHz UltraSPARC) running SunOS 5.7.

Our test suite consists of three smallish (though non-trivial) applications written in XDuce:

Bookmarks is 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`).

Diff implements Chawathe's "tree diff" algorithm [8]. 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 was a tremendous help in finding silly mistakes.

The `HTML` type (more precisely, `XHTML`, which is an `XML` implementation of `HTML`) is considered to be one of the largest types for web documents. 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 1 shows the number of lines of XDuce in the whole program (counting 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), as described in Section 4. The optimizations are all turned on for this table. The column "subtyping" shows the time spent by the main algorithm SUB.

As the table indicates, the speed of type checking is acceptable for these applications. In particular, it is quite encouraging that it takes less than one second to type check programs involving the full `HTML` type.

For reference, Table 1 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. Notice that the number of assumptions is much smaller than the number of states. (If we used a classical algorithm based on determinization of tree automata, we would have to consider a far larger number of pairs of states. In fact, in an early stage, we did implement such an algorithm and it did not seem to be practical at all.)

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 the static type safety, but also all the other features provided by the host language. The cost is that XML values and their corresponding DTDs must somehow be "injected" into the value and type space of the host language; this usually involves adding more layers of tagging than were present in the original XML documents, which inhibits subtyping. The lack of subtyping (or availability of only restricted forms of subtyping) is not a serious problem for simple traversal of tree structures; it becomes a stumbling block, though, in tasks like the "database 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 [21] 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.)

Another piece of work along similar lines is the functional language XMλ for XML processing, proposed by Meijer and Shields [18]. Their type system is close to Haskell's, except that they incorporate so-called *Glushkov automata* in type checking, resulting in a more flexible type system. However, neither their type system nor Glushkov automata are described in detail in their paper, making comparison with our work difficult.

A closer relative to our type system is the type system for the query language YAT [9], which allows optional use of types similar to DTDs. The notion of subtyping between these types is somewhat weaker than ours (lacking, in particular, the distributivity laws used in our "database integration" example).

Types based on tree automata have also been proposed in a more abstract study of typechecking for a general form of "tree transformers" for XML by Milo, Suciu, and Vianu [19]. The types there are conceptually identical our regular expression types, (except for subtagging).

Regular expression types were originally motivated by an observation by Buneman and Pierce [7] that untagged union types corresponded naturally to forms of variation found in semistructured databases. The difference from the present paper is that they studied unordered record types instead of ordered sequences and did not treat recursive types.

Our subtyping algorithm is closely related to the algorithms for set-inclusion constraint solving developed by Aiken and others [1, 2]; in particular, our algorithm is very similar to the algorithm described in [1]. However, the goals of the analyses are somewhat different: we are interested in a type system for web programming, while they are interested in program analysis. This difference of domains has at least two implications. First, since we have to clearly explain any type errors to the user, it is important to devise a provably complete algorithm, while it is not so critical in their context of program analysis for optimization, where one can argue that incompleteness is tolerable if false answers happen only rarely (as in some other type systems [5, 4]). However, as we have shown in Section 4.1, such cases *would* arise in practice in our setting if we took Aiken and Murphy's algorithm directly. Second, the difference of the domain leads us to experiments with different inputs. Their inputs are collected from analysis on a whole program and can be huge. Tackling such a big job takes long to finish (e.g., more than one minute for a program of a few thousands of lines [11]). On the other hand, our inputs are types that the user writes, which we can assume are not nearly so large. In addition, since we want to use subtyping very casually in our type checker, it must be very quick.

Another paper by Aiken and Wimmers [2] describes a different decision procedure for set-constraint solving. Although it gives a complete algorithm, the use of intersections and negations appears rather critical in their algorithm; it is not obvious exactly how an efficient algorithm could be derived from it in the absence of these features.

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, 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).

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).

9. REFERENCES

[1] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture1991*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[2] A. Aiken and E. L. Wimmers. Solving systems of set constraints (extended abstract). In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

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

[4] Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur and A. Scedrov. Lambda prolog:

An extended logic programming language. In E. L. R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 754–755, Berlin, May 1988. Springer.

[5] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.

[6] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, Apr. 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309-338, 1998.

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

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

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

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

[11] M. Fahndrich and A. Aiken. Making set-constraint program analyses scale. Technical Report CSD-96-917, University of California, Berkeley, Sept. 1996.

[12] V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

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

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

[15] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, May 2000.

[16] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. Technical report, University of Pennsylvania, 2000.

[17] X. Leroy, J. Vouillon, D. Doligez, et al. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[18] E. Meijer and M. Shields. XMLambda: A Functional Programming Language for Constructing and Manipulating XML Documents. page 13. Submitted to USENIX 2000 Technical Conference.

[19] T. Milo, D. Suciu, and V. Vianu. Typechecking for xml transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

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

[21] M. Wallace and C. Ranciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34-9 of *ACM Sigplan Notices*, pages 148–159, N.Y., Sept. 27–29 1999. ACM Press.

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

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