Regular Expression Types for XML

Haruo Hosoya
Benjamin C. Pierce
Jérôme Vouillon

University of Pennsylvania

(with Peter Buneman and Phil Wadler)

XML

- a standard format for exchange of structured data
- huge corporate and* academic buy-in

*hence?

XML “Types”

A major selling point for XML is that documents come with descriptions of their structure

- DTDs (old standard)
- XML-Schema (proposed new standard)
- Relax, etc. (other similar proposals)
- Growing literature on refinements and extensions (e.g. keys)

But...

XML “Types”

A major selling point for XML is that documents come with descriptions of their structure

- DTDs (old standard)
- XML-Schema (proposed new standard)
- Relax, etc. (other similar proposals)
- Growing literature on refinements and extensions (e.g. keys)

But... these “types” are not actually used for typechecking programs!

The standard approach

A typical XML processing program...

- reads an XML tree
- verifies (during parsing) that it matches some DTD (or Schema)
- ignores the DTD from this point on, treating the input as a generic labeled tree (e.g., a DOM structure)
- constructs new trees that may or may not conform to an intended DTD

The standard approach

A typical XML processing program...

- reads an XML tree
- verifies (during parsing) that it matches some DTD (or Schema)
- ignores the DTD from this point on, treating the input as a generic labeled tree (e.g., a DOM structure)
- constructs new trees that may or may not conform to an intended DTD

We can do better!

A Better Approach

There have been a number of proposals for using the existing features of popular statically typed languages to represent XML structures.

E.g.,

- Data-binding for Java [Sun]
- HaXML [Wallace and Runciman, ICFP99]

This is good!

Static type safety $\longrightarrow$ increased robustness

- A well-typed program cannot fail because an intermediate stage generates a structure that another stage is not expecting
- A well-typed program will always produce output that conforms to the expected DTD

But...

Union has lost its flexibility

The problem with such embeddings is that they introduce spurious structure, mapping the non-disjoint union operator of DTDs to the disjoint union of the host-language type system.

XML	Java/C#	ML
`S|T`	class `SorT`	`datatype SorT =`
	class `S` extends `SorT`	`left of S`
	class `T` extends `SorT`	`| right of T`

Consequences:

- An element of `S` cannot simply be viewed as an element of `S|T` — an explicit coercion is required
- If `S` and `T` have fields in common, there is no way to access them directly in an element of `S|T` — we must first perform a tag-test

Consequences:

- An element of `S` cannot simply be viewed as an element of `S|T` — an explicit coercion is required
- If `S` and `T` have fields in common, there is no way to access them directly in an element of `S|T` — we must first perform a tag-test

We can do <u>even</u> better!

Regular expression types

Our proposal: Use DTDs* themselves as types.

*suitably cleaned up and generalized.

XDuce

- An experimental programming language for writing "recursive tree transducers" for XML structures
- Static typechecking based on regular expression types
- Flexible subtyping relation supporting common patterns of evolution and data integration
- Generalization of ML-style pattern matching with regular expression patterns
- Similar "feel" to XSLT

Outline

- Introduction ✔
- Regular expression types
- Regular expression patterns
- Current work: Xtatic

Regular Expression Types

An Example (in XDuce notation)

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

val mybook = addrbook[person[name["Haruo Hosoya"],
                             email["hahosoya@upenn"],
                             email["haruo@u-tokyo"]],
                      person[name["Jerome Vouillon"],
                             email["vouillon@upenn"],
                             tel["215-123-4567"]]]
```

Semantics of Types

Each type denotes a set of sequences.

E.g.:

$$
\begin{array}{rcll}
[\![\texttt{Email*}]\!] & = & \{ & () \\
 & & & \texttt{email}[\ldots] \\
 & & & \texttt{email}[\ldots], \texttt{email}[\ldots] \\
 & & & \text{etc.} \quad \}
\end{array}
$$

Comma denotes concatenation of sequences:

$$[\![\texttt{Name},\texttt{Email*},\texttt{Tel}]\!] \quad = \quad \left\{ \begin{array}{l} \texttt{name}[\ldots], \texttt{tel}[\ldots] \\ \texttt{name}[\ldots], \texttt{email}[\ldots], \texttt{tel}[\ldots] \\ \texttt{name}[\ldots], \texttt{email}[\ldots], \texttt{email}[\ldots], \texttt{tel}[\ldots] \\ \text{etc.} \quad \end{array} \right\}$$

Subtyping (Examples)

$?$ means "optional"

$$\texttt{Name} \quad <: \quad \texttt{Name}, \texttt{Email}?$$

$$\texttt{Name}, \texttt{Email} \quad <: \quad \texttt{Name}, \texttt{Email}?$$

$*$ means "zero or more"

$$\texttt{Email}, \texttt{Email}, \texttt{Email} \quad <: \quad \texttt{Email}*$$

$$\texttt{Email}, \texttt{Email}* \quad <: \quad \texttt{Email}*$$

$|$ means "or"

$$\texttt{Email} \quad <: \quad \texttt{Email}|\texttt{Tel}$$

$$\texttt{Tel} \quad <: \quad \texttt{Email}|\texttt{Tel}$$

"forget ordering" subtyping

$$\texttt{Email}*, \texttt{Tel}* \quad <: \quad (\texttt{Email}|\texttt{Tel})*$$

Subtyping (Examples)

$?$ means "optional"

$$\texttt{Name} \;<:\; \texttt{Name},\texttt{Email}?$$

$$\texttt{Name},\texttt{Email} \;<:\; \texttt{Name},\texttt{Email}?$$

$*$ means "zero or more"

$$\texttt{Email},\texttt{Email},\texttt{Email} \;<:\; \texttt{Email}*$$

$$\texttt{Email},\texttt{Email}* \;<:\; \texttt{Email}*$$

$|$ means "or" Note that $|$ is a non-disjoint union!

$$\texttt{Email} \;<:\; \texttt{Email}|\texttt{Tel}$$

$$\texttt{Tel} \;<:\; \texttt{Email}|\texttt{Tel}$$

"forget ordering" subtyping

$$\texttt{Email}*,\texttt{Tel}* \;<:\; (\texttt{Email}|\texttt{Tel})*$$

Subtyping Recursive Types

Recursive types describe arbitrarily deep tree structures:

```
type BinTree  = Leaf[String]
              | node[BinTree,BinTree]

type UBinTree = Leaf[String]
              | node[BinTree,UBinTree?]
```

```
BinTree  <:  UBinTree
```

Example: Data Integration

Challenge: Suppose we are given two data sources that have almost but not exactly the same types (e.g., two databases that have evolved separately from a common origin)

Task:

1. Integrate (combine) the two sources
2. Write a simple program that performs some operation on the common fields

Original data sources:

```
src1 ∈ (Name,Email)*
src2 ∈ (Name,Tel)*
```

Integration = concatenation

```
src1,src2 ∈ (Name,Email)*,(Name,Tel)*
```

Next, we want to do something with the common part (i.e., the `Name` fields).

Since there are two $*$'s, we need to use two separate loops, right?

No! Just use subtyping between regular expression types to factor out the common part.

Step 1: Use "forget ordering" subtyping to merge $*$'s:

$$
\begin{array}{ll}
 & (\mathtt{Name},\mathtt{Email})*,(\mathtt{Name},\mathtt{Tel})* \\
<: & ((\mathtt{Name},\mathtt{Email})|(\mathtt{Name},\mathtt{Tel}))*
\end{array}
$$

Step 2: Distribute union over concatenation to factor out `Name`:

$$
\begin{array}{ll}
 & ((\mathtt{Name},\mathtt{Email})|(\mathtt{Name},\mathtt{Tel}))* \\
<: & (\mathtt{Name},(\mathtt{Email}|\mathtt{Tel}))*
\end{array}
$$

(Distributivity is the most important advantage of real union types over disjoint unions a la ML or Java.)

Step 3: Write a single loop to extract the `Names`.

```
fun names : (Name, (Email|Tel))* -> Name* =
    name[n], (Email,Tel), rest
      -> n, names(rest)
  | ()
      -> ()
```

Subtyping

[insert additional slides]

Efficiency of Subtyping

The algorithm we've sketched gives us a decision procedure for the full "semantic" subtyping relation.

How fast is it?

Efficiency of Subtyping

The algorithm we've sketched gives us a decision procedure for the full "semantic" subtyping relation.

How fast is it?

In the worst case, not very fast: subtyping between regular expression types is essentially inclusion-testing for regular tree languages, which is exptime-complete.

However, we can go a <u>long</u> way with heuristics...

Subtyping Heuristics

- low-level tricks (hash-consing and memoization)
- special treatment of empty types
- a variety of "set-theoretic" optimizations to avoid using the general union rule

cf. [Hosoya, Pierce, & Vouillon, ICFP 2000]

Some Preliminary Measurements

	Bookmark	Html2Latex
Size (code)	310 lines	312 lines
Size (types)	1242 lines	1217 lines
# of subtype checks	61	123
Type checking time	0.48 secs	0.88 secs

on a 300Mhz Ultrasparc

Regular Expression Patterns

Example

Recall the address book types from earlier:

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

A simple pattern match:

```
match p with
    person[name[n], Email*, tel[t]]
      ->  (* do some stuff involving n and t *)
  | person[p]
      ->  (* do other stuff *)
```

Note how the type `Email*` is used in the first pattern to match a variable-length sequence of `email` nodes.

A complete XDuce function

```
fun tels : Person* -> (Name,Tel)* =
    person[name[n], Email*, tel[t]], rest
                        ->  name[n], tel[t], tels(rest)
  | person[p], rest     ->  tels(rest)
  | ()                  ->  ()
```

- header is explicitly annoted with both argument and result types
- other type annotations (in particular, on pattern variables) are omitted

A More Interesting Example

Using regular expression patterns, we can extract the subcomponents of an HTML table with a single `match`...

```
match t with
    table[cap as Caption?,
          col as (Col*|Colgroup*),
          hd as Thead,
          ft as Tfoot?,
          bd as (Tbody+|Tr+)]
      -> ...
```

Issues

- Type inference for pattern variables
  - complicated by...
    - recursive types and patterns
    - complex control flows arising from "first-match" policy

  [Cf. Hosoya & Pierce , POPL 2001]

- Optimization of patterns to avoid type membership testing at run time whenever possible

Status

XDuce Status

- Prototype implementation
  - Interpreted runtime
  - Fairly fast typechecker
- Several small (but nontrivial) applications
  - Web browser bookmark formatter
  - Html2latex translator
  - Tree-diff [Chawathe, VLDB '99]
  - Prototype XML-Schema validator [Renneberg '00]

Ongoing work: the Xtatic project

A new language design and implementation based on XDuce

- Emphasis on inter-operability with a mainstream "host language" (Java or C#)
- Compiling to a common runtime system (JVM or MS .net common runtime)

Xtatic: New design issues

- unordered record types (based on “interleave” operator for regular trees)
- integration of “horizontal” (XSLT-, ML-, or XDuce-style) and “vertical” (DB query language) patterns
- object types (and/or higher-order functions)
- polymorphism?

Finishing up...

Related work

Other XML languages with static type systems:

- XMλ [Meijer & Shields, 2000]
- YAT [Cluet & Simeon, WebDB1998]

Similar aims. Limited support for subtyping (e.g., no distributivity laws).

Related work

Typechecking for XML query languages [Milo, Suciu, and Vianu, PODS 2000, Papakonstantinou & Vianu, PODS 2000]

- Complexity studies of typechecking problems
- Similar notions of types and subtyping
- Applications / implementation issues not considered

Related work

"Union Types for Semi-structured Data" [Buneman & Pierce, DBPL 99]

- Powerful subtyping (including similar distributive laws)
- Unordered records (rather than XML's ordered sequences)
- No recursive types

Related work

XML Algebra [Fernandez, Simeon, and Wadler, 2000]

- Proposed core for XML query languages
- Draft W3C standard
- Type system based directly on XDuce

Related work

"Set inclusion constraints" [Aiken&Murphy, FPCA91, Aiken&Wimmers, LICS92]

- Closely related algorithmic problem
- See our ICFP 2000 paper for a detailed comparison

If you want to play...

The XDuce home page

www.cis.upenn.edu/~hahosoya/xduce

contains papers, talks, and our prototype implementation.