Native XML Processing in a Statically Typed Language

A Progress Report on the Xtatic Project

Benjamin C. Pierce

University of Pennsylvania

Overview

Xduce [Hosoya, Pierce, Vouillon]: a tiny experimental language with "native XML support" in the form of

- regular types
- regular pattern matching

Xtatic project: equip a full-blown OO language (C#) with these features

This talk: a status report on Xtatic, with some technical details on the core language and the current implementation.

Background

A missed opportunity

XML documents frequently come with precise and detailed schemas.

However, these schemas are not "understood by" the type systems of any of the programming languages that are commonly used to generate, manipulate, and analyze XML documents.

I.e., schemas can be used to validate values, but not to analyze programs.

Benefits of native static typechecking

- Obvious: many opportunities for silly errors in XML processing code (i.e., easy to produce documents not conforming to the expected schema). Any help the typechecker can give us would be welcome.
- More interesting: putting these rich types within the range of the typechecker's understanding gives it a substantial grip on the program's underlying logic, amplifying the "curious effectiveness" of static typing.

Regular Types

Key observations:

- XML schema languages are based on regular tree grammars (a simple extension of familiar regular expressions on strings)
- Core operations needed by typecheckers (membership and subtype testing, etc.) correspond to well-known operations on tree grammars

Regular types

`T`	::=	`String`	leaf
		`X`	type name
		`()`	empty sequence
		`T,T`	concatenation
		`l[T]`	tree labeled `l`
		`~[T]`	tree labeled anything
		`T|T`	alternation (union)

Fix global set of (mutually recursive) definitions `X = T`.

Recursive uses of variables only allowed in rightmost positions and under labels (to keep things regular).

Standard regex operators (`T?`, `T*`, etc.) definable

Regular Pattern Matching

Regular types suggest an elegant pattern-matching mechanism...

- statically typed "tree-grep"
- typechecker can do standard tests for exhaustiveness and irredundancy
- includes all of ML-style "algebraic pattern matching" as a special case
- experience in XDuce: very pleasant for programming

Regular Patterns

Regular patterns are just regular types decorated with variable bindings:

P	::=	`String`	leaf
		`X`	pattern name
		`()`	empty sequence
		`P,P`	concatenation
		`l[T]`	tree labeled `l`
		`~[T]`	tree labeled anything
		`P|P`	alternation
		`P as x`	binding

Linearity: In $\texttt{P}_1\texttt{|P}_2$, the sub-patterns $\texttt{P}_1$ and $\texttt{P}_2$ must bind the same set of variables. In $\texttt{P}_1\texttt{,P}_2$, disjoint sets.

XDuce

What we achieved:

- basic definitions (regular types and pattern matching)
- fundamental algorithms (subtyping, type-based pattern optimization)
- prototype implementation (good-quality front end + simple interpreter)

What we did not achieve:

- Full-blown language design / implementation
- Inter-operability with established libraries, mainstream programming idioms, and legacy code

Xtatic

Regular types for the masses...

Overall Goals

- lightweight, source-level extension of a mainstream, general purpose language with regular types and patterns
  - we are using C# as the host language; the same can easily be done with Java
- complete binary-level compatibility with existing host-language APIs and legacy code

  $\longrightarrow$ no changes to CLR abstract machine
- efficiently compiled
- "reasonably compatible" with existing standards (e.g. XML-Schema)

Specific Research Goals

- get inter-operability right...
  - in the data model (smooth intermingling of objects and XML)
  - in the type system (overloading, separate compilation, etc.)
  - in the run-time system
    - smooth transitions between statically and dynamically typed processing
    - fast re-validation when types get "lost"
    - DOM proxies
- deepen understanding of regular patterns
  - high-performance compilation
  - partial type inference
- develop formal foundations of the core language
  - subject reduction theorems and all that

Related Projects

- XQuery (W3C)
  - standard query language for XML
  - type system based on XML-Schema
  - no direct integration with objects
- XJ (IBM)
  - Java extension with native support for XML
  - emphasis on standards (XML-Schema, XPath, DOM, ...)
  - imperative programming idiom (dynamically validated)
- Xact (BRICS)
  - Java extension with novel "template filling" programming idiom for XML processing
  - global static analysis plays role of Xtatic's type system

More Related Projects

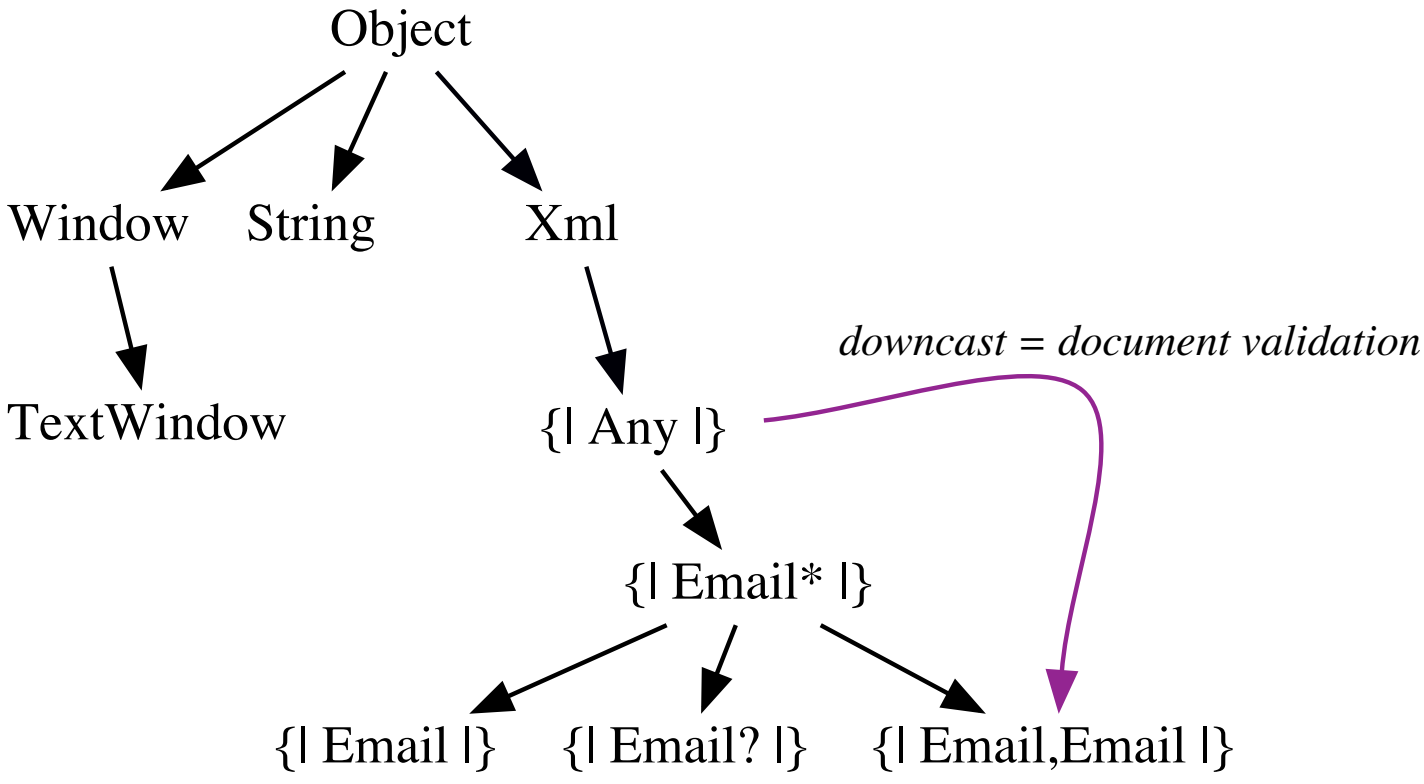
- CDuce
  - full-blown language design emphasizing statically typed XML processing
  - based on CLOS-like multi-method dispatch
  - ambitious semantic foundations and typechecking algorithms, generalizing XDuce type system with intersection and function types
- Xen (Microsoft)
  - "deep integration" of XML and C# type structures
  - little technical information available
  - Xobe (german)
    - Java-based design similar to Xtatic
    - few published details so far

Language Overview

Core Data Model and Type System

- Allow arbitrary host-language objects as tags in XML documents.
- Add regular types `[[T]]` to the class types from the host language.
- Treat XML structures as objects by introducing a special type `Xml` and making `[[T]] <: Xml` for every `T`.
- Allow (dynamically checked) downcasts from `Xml` to `[[T]]`, so that XML values can be manipulated by unmodified host-language programs and libraries (stored in generic data structures, etc.) and later passed back in to Xtatic modules.

Xtatic Subtype Hierarchy



Compilation scheme

After typechecking, Xtatic programs are transformed into pure host-language programs by:

- rewriting all XML types `[[T]]` to the single class type `Xml`
- rewriting all XML values as host-language data structures (objects of class `Xml`)
- compiling regular pattern matches into nested combinations of host-language `switch`es and conditionals [and, for good performance, method calls, arrays, exceptions, etc.]

(Same intuition as GJ)

Example

```
regtype Name  [[ <name>PCDATA</> ]]
regtype Tel   [[ <tel>PCDATA</>  ]]
regtype Email [[ <e-mail>PCDATA</> ]]

regtype Person    [[ <person> Name Tel? Email* </> ]]
regtype Addrbook  [[ <addrbook> Person* </> ]]
regtype Telbook   [[ (<entry> Name Tel </>)* ]]
```

Example

```
class TelbookMaker {
  void Main () {
    match (xtatic.io.fromFile("addressbook.xml")) {
      case [[ <addrbook> Person* as ps </> ]]:
        Telbook tbook = mkTelbook(ps);
        xtatic.io.toFile("telbook.xml", tbook);
      case [[ Any ]]:
        System.Console.writeln ("bad file");
    }
  }}
```

Example

```
Telbook mkTelbook ([[ Person* ]] ps) {
  match (ps) {
    case [[ <person>
               <name>PCDATA as n</>
               <tel>PCDATA as t</>
               Any
             </person>
             (Person* as rest) ]]:
      return [[ <entry> <name>n</> <tel>t</> </entry>
                mkTelbook(rest) ]];
    case [[ <person> Any </person> (Person* as rest) ]]:
      return mkTelbook(rest);
    case [[ ]]:  return [[ ]];
  }}
```

Core Language Design

FX: a core calculus for Xtatic

FX: a tiny core language combining objects and classes with regular expression types and patterns.

Based on Featherweight Java [Igarashi, Pierce, Wadler, 1999], a tiny core language retaining the key type structures and OO features of Java/C# (classes, methods, fields, inheritance, `this`) and nothing else.

FX types

Full language:

A	::=	C	"host type" (class name)
		[[T]]	XML type

regular types:

T	::=	X	type name
		()	empty sequence
		T,T	concatenation
		T|T	alternation
		<(A)> T </>	tree labeled with A object

FX values

objects:	`h`	::=	`new C(`$\overline{\texttt{a}}$`)`	object of class `C` with constructor args $\overline{\texttt{a}}$
trees:	`t`	::=	`<(a)>`$\overline{\texttt{t}}$`</>`	tree with label `a` and contents $\overline{\texttt{t}}$
full lang:	`a`	::=	`h`	host value
			`[[`$\overline{\texttt{t}}$`]]`	sequence of trees

(overbar stands for sequences)

Type membership

$$\text{instances}(\texttt{C}) = \begin{cases} \{[[\overline{\texttt{t}}]] \mid \text{for arbitrary } \overline{\texttt{t}}\} & \text{if } \texttt{C} = \texttt{Xml} \\ \{\texttt{new C}(\overline{\texttt{a}}) \mid \text{for arbitrary } \overline{\texttt{a}}\} & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl}
[\![\texttt{C}]\!] & = & \bigcup\{[\![\texttt{D}]\!] \mid \texttt{D} \text{ subclass of } \texttt{C}\} \\
[\![\texttt{()}]\!] & = & \{\ \langle\text{empty seq}\rangle\ \} \\
[\![\texttt{<(A)>T</>}]\!] & = & \{\ \texttt{<(a)>}\overline{\texttt{t}}\texttt{</>} \mid \texttt{a} \in [\![\texttt{A}]\!] \wedge \overline{\texttt{t}} \in [\![\texttt{T}]\!]\ \} \\
[\![\texttt{T}_1\ \ \texttt{T}_2]\!] & = & \{\ \overline{\texttt{t}}_1\ \ \overline{\texttt{t}}_2 \mid \overline{\texttt{t}}_1 \in [\![\texttt{T}_1]\!] \text{ and } \overline{\texttt{t}}_2 \in [\![\texttt{T}_2]\!]\ \} \\
[\![\texttt{T}_1 \,|\, \texttt{T}_2]\!] & = & [\![\texttt{T}_1]\!] \cup [\![\texttt{T}_2]\!] \\
[\![\texttt{X}]\!] & = & [\![\texttt{T}]\!] \quad \text{where } \texttt{X} = \texttt{T} \text{ in the global definitions}
\end{array}$$

Subtyping

$$A <: B \quad \text{iff} \quad [\![A]\!] \subseteq [\![B]\!]$$

Note that subtyping is generated "semantically" from the declared subclass relation. The simplicity of this construction is made possible by

- the nominal character of the host type system
- the absence of function types

Patterns

Q	::=	C	class pattern
		C as x	class pattern plus binder
		[[P]]	XML pattern
P	::=	X	type name
		()	empty sequence
		P,P	concatenation
		P|P	alternation
		<(Q)> P </>	tree
		P as x	XML pattern plus binder

Expressions

e	::=	x	value variable
		new C($\overline{e}$)	new object creation
		e.f	field access
		e.m($\overline{e}$)	method call
		<(e)>$\overline{e}$</>	tree
		[[$\overline{t}$]]	sequence of trees
		match (e) {$\overline{\text{case}}$ ($\overline{Q}$):$\overline{e}$}	pattern match

Adding Primitives

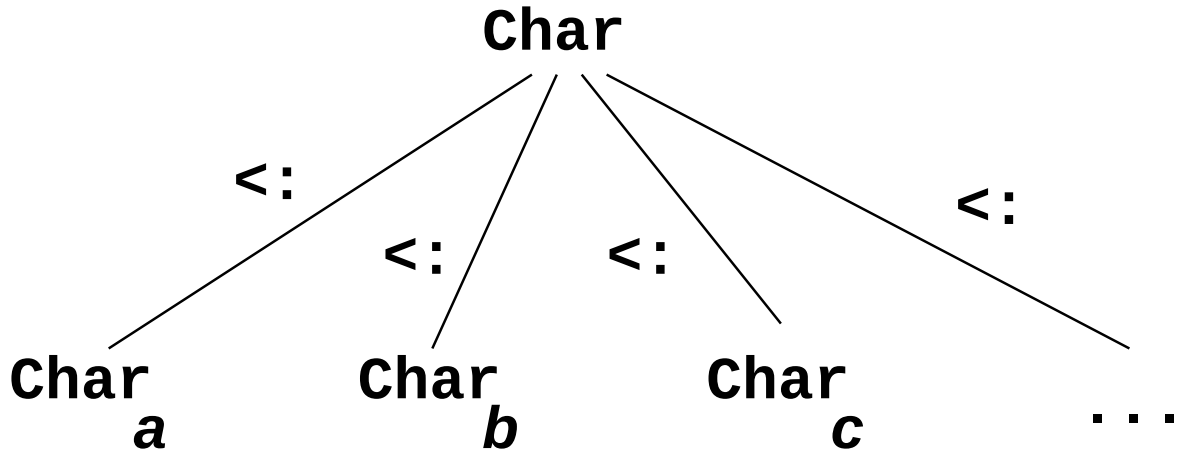
Naturally, it is easy to enrich this tiny language with primitive types such as `Char`.

Interestingly, though, we can simulate such primitive types using just the mechanisms that we already have.

(Doing so will suggest a nice way of representing XML structures...)

Representing `Char`

Introduce a new class `Char` and 256 subclasses $\texttt{Char}_a$, $\texttt{Char}_b$, etc.



Each subclass contains one object (written `new` $\texttt{Char}_a\texttt{()}$, `new` $\texttt{Char}_b\texttt{()}$, etc.), which stands for the corresponding character constant.

Representing `Char`

To recover the standard syntax of character literals, we can write `'a'` to stand for `new Char`$_a$`()` in values.
Then, by subsumption, we have `'a'` $\in$ `Char`, `'b'` $\in$ `Char`, etc.
I.e., `Char` is isomorphic to the standard primitive type of characters.

Representing `Char`

If, moreover, we let `'a'` stand for $\texttt{Char}_a$ in types, then we can recover the behavior of C#'s ordinary `switch` statement from regular pattern matching:

```
switch (c) {
 case ('a'):   ...
 case ('b'):   ...
 case ('c'):   ...
}
```

stands for

```
match (c) {
 case (Char_a):   ...
 case (Char_b):   ...
 case (Char_c):   ...
}
```

Representing PCDATA

Define

```
XChar = <(Char)/></>
```

I.e., an element of `XChar` is a trivial tree containing a single character

Now define

```
PCDATA = XChar*
```

I.e., PCDATA is an arbitrary length sequence of characters.

(We'll see in a bit why this is better than defining `PCDATA` as `<(String)></>`.)

Representing XML Tags

The same idea gives us a convenient representation of the tags in XML structures.

1. Introduce a new class `Tag`
2. Introduce an infinite collection of "singleton subclasses" — $\texttt{Tag}_{address}$, $\texttt{Tag}_{tel}$, $\texttt{Tag}_{email}$, ...
3. In values, write `<tag>...</>` as syntactic sugar for `<(new` $\texttt{Tag}_{tag}$`)>...</>`
4. In types, write `<tag>...</>` as syntactic sugar for `<(`$\texttt{Tag}_{tag}$`)>...</>`

Representing XML Tags

The class-membership test of our `match` statement now functions as a tag test

```
match (d) {
 case [[<person><name>PCDATA as n</></>]]:
 ... }
```

stands for

```
match (d) {
 case [[<(Tag_person)><(Tag_name)>PCDATA as n</></>):
 ... }
```

Representing XML Tags

This representation also supports dynamic investigation of XML documents with unknown (or partially known) schemas.

1. Provide primitive methods for converting back and forth between `Tag` and `String`
   E.g., equip class `Tag` with a method `toString` and a constructor that takes a `String` argument
2. Pattern-match against class `Tag` instead of $\texttt{Tag}_{xxx}$

Representing XML Tags

```
match (d) {
  case [[ ]]:
    System.Console.writeln ("empty sequence");
  case [[ <(Tag as tg)>Any</> ]]:
    System.Console.writeln ("singleton with tag"
                            + tg.toString());
  case [[ Any ]]:
    System.Console.writeln ("some other sequence");
}
```

Challenges

Polymorphism

- if patterns are just types decorated with variable binders...
- then parametric patterns are just polymorphism!
- parametric patterns (i.e., run-time pattern construction) seem to come up all over the place in practice
- unfortunately, they raise significant theoretical and algorithmic challenges!

(N.b.: What's hard is polymorphism over regular types — polymorphism over classes in the style of GJ is not problematic.)

Playing nice with existing standards

The XML world has some well established standards:

- XML-Schema: Standard schema language (replacement for DTDs)
- XPath: Standard notation for expressing "paths" to parts of documents.

Both are messy, complex beasts — we can't include all their features, verbatim.
However, they are the real world. Need to be able to interpret at least a large subset of each.

How close are we now?

XML-Schema

Disclaimer: Schema is full of bells and whistles — most of them we have not thought about very much!
Here's one interesting one, though...

Schema supports precise descriptions of string formats. (Not just "a phone number is a string" but "a phone number is a string of three digits, a hyphen, three more digits, etc.")

In Xtatic, we get the same effect "for free" from our definition of `PCDATA` as `XChar*`. I.e., instead of

```
regtype Tel   [[ <tel>PCDATA</>  ]]
```

we write

```
regtype Tel [[ <tel> AreaCode? Local </> ]] ;
regtype AreaCode [[ '+' '1' '-' Dgt Dgt Dgt '-' ]]
regtype Local [[ Dgt Dgt Dgt '-' Dgt Dgt Dgt Dgt ]]
regtype Dgt [[ '0'|'1'|'2'|'3'|'4'|
               '5'|'6'|'7'|'8'|'9' ]]
```

Conversely, regular pattern matching can be used to extract the parts of phone numbers:

```
match (tel) {
  case [[ tel[AreaCode ac, Local l] ]]:
    // process long distance number...
  case [[ tel[Local l] ]]:
    // process local number...
}
```

(Statically typed PERL, anyone?...)

XPath

XPath includes some features that we definitely do not want in Xtatic.

- "go to root of document; find any subtree labeled `tel`; go up to its parent; move to the right; ..."

However, at least two ideas from XPath look very useful.

- Find all occurrences of a given pattern.
- Find a given pattern at arbitrary depth in a tree.

The first requires a (straightforward?) change in the semantics of pattern matching. The second, though...

We can desugar a pattern of the form `//P` ("find `P` anywhere") as a regular pattern:

```
pattern PInside = Any P Any
                | Any <(Object)>PInside</> Any
```

where `Any` is the type of arbitrary sequences of XML trees:

```
regtype Any = ()  |  <(Object)>Any</> Any
```

Finishing Up

Status

- language design stable
- source-to-source compiler, emphasizing good pattern matching performance
  - not yet handling all of C#
- several small demos
- work proceeding on...
  - larger demos
  - polymorphism
  - even better (type-based) pattern compilation

Contributors

The Xtatic team:

Vladimir Gapeyev
Michael Levin
Benjamin Pierce
Alan Schmitt

Also, thanks to Eijiro Sumii for the ASP.NET work on the demo!

Any Questions?

http://www.cis.upenn.edu/~bcpierce/xtatic