Native XML Processing in Object Oriented Languages

Calling XMHell from PurgatOOry
“So the Essence of XML is this: the problem it solves is not hard, and it does not solve the problem well.”

[Siméon, Wadler – POPL’03]
The road to XML is paved with good intentions... 

- XML data is pervasive
  - need powerful tools to manipulate it

- XML has a rich data model
  - integrate it with the OO data model

- This talk is about the practical integration of the XML and OO data models

- This talk is not about
  - XML standards
    - Schema, Relax NG, ...
  - non-OO XML manipulation languages
    - XQuery, XDuce, CDuce, ...
Native XML manipulation in OO languages

- The evolution of XML integration
  *From Strings to Regular Types*

- Practical aspects of XML manipulation
  *Generation X: XJ, Xact, and Xtatic*

- Future challenges
  *Xen and the Art of Language Design?*
The Evolution of XML manipulation
A simple XML address book

<addrbk>
  <entry>
    <name>Pat</name>
    <tel>314-1593</tel>
    <email>Pat@pat.com</email>
  </entry>
  <entry>
    <name>Jo</name>
    <tel>271-8282</tel>
    <email>Jo@jo.com</email>
  </entry>
</addrbk>
<addrbk>
    <entry>
        <name>Pat</name>
        <tel>314-1593</tel>
        <email>Pat@pat.com</email>
    </entry>
    <entry>
        <name>Jo</name>
        <tel>271-8282</tel>
        <email>Jo@jo.com</email>
    </entry>
</addrbk>
The Stone Age
Strings
Strings

"<addrbk>
  <entry>
    <name>Pat</name>
    <tel>314-1593</tel>
    <email>Pat@pat.com</email>
  </entry>
  <entry>
    <name>Jo</name>
    <tel>271-8282</tel>
    <email>Jo@jo.com</email>
  </entry>
</addrbk>"

- Used widely...
  - CGI
  - Java servlets
- ...with difficulties
  - Tedious to write and maintain
  - Output might not be well formed
The Bronze Age
Concrete Data Structures
Concrete Data Structures

- DOM (Document Object Model) like JDOM
  - Provide a generic, standardized AST for XML values
  - Provide an API to manipulate it

- Advantages
  - Many parsers and pretty printers available
  - Generates well formed XML

- Annoyances
  - Little or no check of validity
  - Low-level API
  - Very concrete representation
    - White space may be significant and cannot be ignored
Address book in DOM

Element
  name = "addrbk"
  children
    Text
      data = " \n"
    Element
      name = "entry"
      children
        Element
          name = "name"
          children
            Text
              data = " Pat"
        Element
          name = "tel"
          children
            Text
              data = "314-1593"
        Element
          name = "email"
          children
            Text
              data = "Pat@pat.com"
The Middle Ages
Data Binding
Data Binding

XML language bindings are "software mechanisms that transform XML data into values that programmers can access and manipulate from within their language of choice."

[Simeoni et. al. – IEEE Internet Computing, 2003]

- Most XML documents follow a restricted model
- Many description systems: DTD, XML-Schema, Relax...
- Translate ("bind") XML types $S$ to classes $[S]$ and XML values $d$ satisfying $S$ to objects $[d]_S$ of class $[S]$
- Address book type:

\[
Addrbk = \langle addrbk \rangle \text{Entry}^* \rangle
\]

\[
\text{Entry} = \langle entry \rangle
\]

\[
\langle \text{name} \rangle \text{pcdata}\rangle, \langle \text{tel} \rangle \text{pcdata}\rangle, \langle \text{email} \rangle \text{pcdata}\rangle
\]

\[
\rangle \langle /entry \rangle
\]
Binding Structure

Reflect XML structure in the OO type system.

type Addrbk =
  <addrbk> Entry* </>

class Addrbk {
  List entries;
}

type Entry =
  <entry>
    <name>pcdata</>,
    <tel>pcdata</>,
    <email>pcdata</>
  </entry>

class Entry {
  Name name;
  Tel tel;
  Email email;
}

class Name { String value; }
class Tel { String value; }
class Email { String value; }
Binding Values

- Reflect XML Values as objects

```xml
<addrbk>
  <entry>
    <name>Pat</name>
    <tel>314-1593</tel>
    <email>Pat@pat.com</email>
  </entry>

  <entry>
    <name>Jo</name>
    <tel>271-8282</tel>
    <email>Jo@jo.com</email>
  </entry>
</addrbk>
```

```java
Addrbk ab = new Addrbk(
    new List(
        new Entry(
            new Name("Pat"),
            new Tel("314-1593"),
            new Email("Pat@pat.com")
        ),
        new List(
            new Entry(
                new Name("Jo"),
                new Tel("271-8282"),
                new Email("Jo@jo.com")
            ),
            EmptyList)
    )
);```

---

Alan Schmitt

Calling XMHell from PurgatOOry

16
**Data Binding**

**Advantages**

- Cleaner representation, easier to navigate
- Automatic generators (Castor, JAXB, Relaxer)
- Some statically checked constraints (OO type system)

**Annoyances**

- Application (or schema) specific
- Errors reported at the level of the host language
- Some features are tricky to reflect
  - Union (no union of classes)
  - Distributivity laws

\[
\text{\langle acq\rangle (\langle friend/\rangle \mid \langle work/\rangle) \text{\langle/acq\rangle} = } \\
(\text{\langle acq\rangle \langle friend/\rangle \text{\langle/acq\rangle}}) \mid (\text{\langle acq\rangle \langle work/\rangle \text{\langle/acq\rangle}})
\]
Enlightenment
The rise of Regular Types
Regular Types [Hosoya, Vouillon, Pierce – ICFP’00]

Do not reflect XML structure, add it as types!

- Regular expressions...

\[ T = () \mid T_1, T_2 \mid T_1 \mid T_2 \mid T^* \]

- ...containing trees...

\[ T = () \mid T_1, T_2 \mid T_1 \mid T_2 \mid T^* \mid <1>T</1> \]

- ...and recursive definitions (vertical recursion)

\[ T = () \mid T_1, T_2 \mid T_1 \mid T_2 \mid T^* \mid <1>T</1> \mid X \]

\[ E = \{ \text{type } X = T \} \]

    type Folder = <folder> Name, (Folder | File) * </>

    type File = <file> Name, Content </>

    type Name = <name> pcdata </>

    type Content = <content> pcdata </>

Technical note: This defines more than regular tree languages

⇒ restrict the position of variables inside an element
Regular Types as a language

- Types correspond to a language (a set of sequences of trees)
- Intuitive denotation of regular types

\[
\begin{align*}
\llbracket()\rrbracket &= \{()\} \\
\llbracket T_1, T_2 \rrbracket &= \{t_1, t_2 \mid t_1 \in \llbracket T_1 \rrbracket, \ t_2 \in \llbracket T_2 \rrbracket\} \\
\llbracket T_1 | T_2 \rrbracket &= \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket \\
\llbracket T^* \rrbracket &= \{t_1, \ldots, t_n \mid n \geq 0, \ \forall k \in [1..n].t_k \in \llbracket T \rrbracket\} \\
\llbracket \langle 1>T\langle/1> \rrbracket &= \{\langle 1>t\langle/1> \mid t \in \llbracket T \rrbracket\} \\
\llbracket X \rrbracket &= \llbracket T \rrbracket \quad \text{if } (\text{type } X = T) \in E
\end{align*}
\]

- Typing is set membership \( t : T \iff t \in \llbracket T \rrbracket \)
Types and Values

type Addrbk = <addrbk>(Friend | Colleague) * </addrbk>

type Friend = <entry> <acq><friend/></acq>, <name>pcdata</name>, <tel>pcdata</tel>,
              (<email>pcdata</email>)?, <addr>pcdata</addr> </entry>

type Colleague = <entry> <acq><work/></acq>, <name>pcdata</name>, <tel>pcdata</tel>,
                 <email>pcdata</email>, <dept>pcdata</dept> </entry>

<addrbk>
  <entry><acq><friend/></acq>, <name>Pat</name>, <tel>314-1593</tel>,
          <addr>42, Wallaby Way</addr> </entry>
  <entry><acq><work/></acq>, <name>Jo</name>, <tel>271-8282</tel>,
          <email>Jo@jo.com</email>, <dept>CIS</dept> </entry>
</addrbk>
Practical Aspects
of XML Manipulation
- Creation, exploration, and modification of XML values.
- Subtyping; interaction of regular types with OO types.
- Compilation and run-time representation.
Generation X

XJ  Bordawekar, Burke, Harren, Raghavachari, Sharkar, Shmueli
    ▶  IBM Research, Thomas J. Watson Research Center

Xobe  Kempa, Linnemann
    ▶  Universität zu Lübeck

Xact  Christensen, Kirkegaard, Møller, Schwartzbach
    ▶  BRICS

Xtatic  Gapeyev, Levin, Pierce, Schmitt, Sumii
    ▶  University of Pennsylvania
An overview...

<table>
<thead>
<tr>
<th></th>
<th>XJ</th>
<th>Xobe</th>
<th>Xact</th>
<th>Xtatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>C#</td>
</tr>
<tr>
<td>Exploration</td>
<td>XPath</td>
<td>XPath</td>
<td>XPath</td>
<td>Pattern Matching</td>
</tr>
<tr>
<td>Mutation</td>
<td>Imperative</td>
<td>Declarative</td>
<td>Declarative</td>
<td>Declarative</td>
</tr>
<tr>
<td>XML in Objects</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Objects in XML</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Objects as Labels</td>
</tr>
<tr>
<td>Subtyping</td>
<td>Nominal</td>
<td>Structural</td>
<td>?</td>
<td>Structural</td>
</tr>
<tr>
<td>Type Checking</td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td>XML at Runtime</td>
<td>DOM</td>
<td>DOM</td>
<td>Lazy List</td>
<td>Lazy List</td>
</tr>
</tbody>
</table>
► Creation, exploration, and modification of XML values.
► Subtyping; interaction of regular types with OO types.
► Compilation and run-time representation.
Creating XML

Most languages embed XML concrete syntax with some escaping mechanism (*pcdata*, *variables*):

```
[[Friend]] pat = [[<entry> <acq><friend/></>,<name>‘Pat’</>,
    <tel>‘314-1593’</>,<addr>‘42, Wallaby Way’</>
</entry>]]

[[Addrbk]] ad = [[<addrbk>pat</>]]
```
Creating XML: the Xact way

- XML templates: XML with named holes
- XML templates may be plugged into holes

![Diagram showing XML templates and plug operations]

[Schwartzbach – http://www.brics.dk/~ck/jaoo2003/]
Exploring trees using XPath

Where does my friend Pat live? 42, Wallaby Way

The XPath way: Giving directions and returning all results

//entry[acq/friend][name/text() = "Pat"]/addr/text()

1. Find all entry children anywhere

2. Consider those that have a <acq><friend/></> child

3. Consider those that also have a <name>Pat</> child

4. Look at what is in the <addr>…</> child

5. Return the text there

   <addrbk>
   <entry><acq><friend/></>, <name>Pat</>, <tel>314-1593</>,
   <addr>42, Wallaby Way</> </entry>
   <entry><acq><work/></>, <name>Jo</>, <tel>271-8282</>,
   <email>Jo@jo.com</>, <dept>CIS</> </entry>
   </addrbk>
Exploring trees using Patterns

Where does my friend Pat live? 42, Wallaby Way

The pattern matching way: giving a map [Hosoya, Pierce – POPL’01]

```xml
<addrbk>
  <entry><acq><friend/></>, <name>Pat</>, any,
    <addr>pcdata x</> </entry>
  any
</addrbk>

<addrbk>
  <entry><acq><friend/></>, <name>Pat</>, <tel>314-1593</>,
    <addr>42, Wallaby Way</> </entry>
  <entry><acq><work/></>, <name>Jo</>, <tel>271-8282</>,
    <email>Jo@jo.com</>, <dept>CIS</> </entry>
</addrbk>
```
Modifying XML in XJ

- **Imperative** assignment
  - Closer to OO style

- Substructure extraction using **XPath**

- Modification pointed by an **XPath** expression

'.addrbk/entry[name/text() = "Pat"]/addr/text()‘ = "4, Privet Drive"
Modifying XML in Xact

- **Declarative** approach (XML data is immutable)
  - Sharing of substructures, Concurrency, Static Analysis

- Extraction of substructures using XPath
  - To select a subtree

- Named holes may be created in a template
  - To select the context of a subtree

\[
\text{gapify}(a/b/c, g) = \text{some result}
\]
Modifying XML in Xtatic

- **Declarative** approach
- XML fragment extraction using **pattern matching**, followed by simple recombination

```scala
match (person) {
  case [[ <entry>Acq k, Name n, Tel t, any</entry> ]]:
    res = [[ <entry>k, n, t</entry> ]];
}
```
- Creation, exploration, and modification of XML values.
- **Subtyping; interaction of regular types with OO types.**
- Compilation and run-time representation.
A type is a type is a type... Subtyping

The essence of subtyping:

*If an operation is guaranteed to be safe on a value of the supertype, then it is safe on a value of the subtype.*
Subtyping for OO types

In the OO world, there already are two forms of subtyping:

**Structural** (OCaml):
- Subtyping of two classes depends on the presence and type of their fields and methods
- **Independent** of class hierarchy
- Rich (and complex)

**Nominal** (Java, C#):
- Subtyping is *declared* (*inheritance*)
- Class hierarchy checked to satisfy structural subtyping
  - Nominal subtyping implies structural subtyping
- Simplifies type checking
Subtyping for Regular Types

As in the OO world, two forms of subtyping can be considered:

**Structural** \( T^+ \sqsubseteq_S T^* \)

(A sequence of 1 or more Ts is a sequence of 0 or more Ts)

**Nominal** \( Km \sqsubseteq_S Distance \)

(A distance in km is a distance)

\[
\text{type } Distance = \langle \text{distance} \rangle Value, (<\text{km}/>)|(<\text{miles}/>) \rangle \\
\text{type } Km = \langle \text{distance} \rangle Value, <\text{km}/> \rangle \\
\text{type } Value = \langle \text{val} \rangle \text{int} \rangle
\]
Structural subtyping for Regular Types

- Each Regular Type is a language

- Subtyping is simply language inclusion
  \[ T \sqsubseteq_T T' \iff [T] \subseteq [T'] \]
  - Intuitive: \( t \in [T] \) and \( T \sqsubseteq_T T' \) implies \( t \in [T'] \)
  - Immediately satisfies many properties
    - **Distributivity** of union over sequences and trees
      \[
      [[<acq> (<friend/> | <work/>) </acq>] =
      [(<acq> <friend/> </acq>) | (<acq> <work/> </acq>)]
      \]
    - **Associativity** of sequence concatenation
Nominal Subtyping of Regular Types

Several approaches to nominal subtyping

- **Purely nominal**: every type declared has a name

- **Structural horizontally, Nominal vertically**
  - Language inclusion of regular expressions of labels
    \[ T = (\) \mid T_1, T_2 \mid T_1|T_2 \mid T^* \mid \mathcal{L} \]
  - Declare subtyping of elements by their label in \( \mathcal{L} \)
  - In Schema, labels are pairs (element, type name)

- Allows finer distinctions (**Mars Climate Orbiter**):
  \[ \text{miles} \neq \text{km} \implies <\text{height} :: \text{miles}>\text{int} \neq <\text{height} :: \text{km}>\text{int} \]

- Subtyping is **faster**

- Must still be structural: \( T \sqsubseteq_N T' \implies T \sqsubseteq_S T' \)

- Need to explicitly state all subtyping relations
Mixing XML and Objects

- Sequences are objects of class `XML`
  - May be used in collections

Most languages follow this approach.
Labels as Objects in Xtatic

- Labels are objects, Label types are classes

\[ T = () \mid T_1, T_2 \mid T_1 | T_2 \mid T^* \mid <(C)>T </> \]

- XML tags are singleton classes, subclass of Tag:
  \[ <\text{addrbk}>\cdots</> \equiv <(\text{Tag}_{\text{addrbk}})>\cdots</> \]

- Characters are singleton classes, subclass of Char:
  \[ \text{‘Pat}’ \equiv <(\text{Char}_P)/><(\text{Char}_a)/><(\text{Char}_t)/> \]

  Pattern matching used for string regular expressions
  ```
  regtype url_protocols [[ ‘http’ | ‘ftp’ | ‘https’ ]]
  regtype url [[ url_protocols , ‘://‘ , (url_char *) ]]
  ...
  case [[ url u, any rest ]]:
    res = [[ res , <a href = u>u</> ]]; p = rest;
  ```
The Class Struggle

Object
  ├── C
  │   ├── D1
  │   └── D2
  ├── String
  └── XML
      ├── Char
      │   ├── Char_p
      │   └── Char_a
      └── Tag
          ├── Tag_addrbk
          └── Tag_entry

R-Types
Mixing Structural and Nominal Subtyping

- **Structural** subtyping for sequences
- **Nominal** subtyping for labels
  - Use the class hierarchy

\[
\begin{align*}
\text{Miles} & 
\sqsubseteq_c 
\text{Km} 
\implies 
<\text{height}>(\text{Miles})/>(\text{Km})/\rangle \\
\text{but} \\
\text{Miles} & 
\sqsubseteq_c 
\text{Int} 
\implies 
<\text{height}>(\text{Miles})/>(\text{Int})/\rangle
\end{align*}
\]

- **Interesting theoretical construction** [Gapeyev, Pierce – Ecoop’03]
- Creation, exploration, and modification of XML values.
- Subtyping; interaction of regular types with OO types.
- **Compilation and run-time representation.**
Source to source translations

All these XML manipulation languages...

- Are language extensions
- Provide access to all language features
- Provide access to all libraries

either

- Write a full Java / C# compiler
- Write a source to source compiler
  - Translation of regular types and values
  - Type checking
  - Run-time representation
Faithful Data Binding (regular types as OO types)

- Translation $[\text{[]}]$ of types and values to target language
- **Exact correspondence** for typing and subtyping:
  \[
  v : _{\text{ext}} T \iff [v] : [T] \text{ and } T \subseteq _{\text{ext}} T' \iff [T] \subseteq [T']
  \]
- Uses existing typing/introspection infrastructure
- May still require type checking for the extension
  - Precise error localization and reporting
  - Type inference

but not there yet...

- May be **impossible** with structural subtyping
Heterogeneous vs Homogeneous translation

**Heterogeneous**  *Fitting square pegs into round holes*

- Approximates faithful data-binding
- Add *coercions* to regain lost subtyping relations
- Com**plex** to design
- Efficiency?

**Homogeneous**  *Where did my type go?*

- Simpler compilation: forget about regular types
- But... first need to typecheck them
- What to do when types are needed?
  - Method overloading → name mangling
  - Separate compilation → store types
  - Introspection (reflection) → type stamps
Type Checking

**XJ** [Haren et al – IBM RC23007]
- Usual type checking (regular types in the language)
- XPath expressions typed with XAEL [Fokoué – Unpublished]
- Imperative XML modifications typed *dynamically*

**Xact** [Kirkegaard, Møller, Schwartzbach – BRICS RS-03-19]
- Static validation on demand
  - Symbolic evaluation of XML transformations
  - Based on control flow graphs
- Guarantees satisfaction of a given DTD

**Xtatic** [Gapeyev, Pierce – Ecoop’03]
- Usual type-checking (regular types in the language)
  - Based on Xduce [Hosoya, Vouillon, Pierce – ICFP’00]
- Inference of types of bound variables in patterns
Xtatic: Type Inference in Patterns

static [[ Phbk ]] mkPhbk ([[ Addrbk ]] addr) {
    [[ PhPers* ]] res = [[ ]];
    [[ <addrbk> (Friend|Colleague)* pers</> ]] = addr;
    bool cont = true;
    while (cont) {
        match (pers) {
            case [[ <entry>Acq k, Name n, Tel t, any</entry>, any rest ]]:
                res = [[ res, <entry>k, n, t</entry> ]];
                pers = rest;
            case [[ ]]:
                cont = false;
        } }
    return [[ <addrbk>res</> ]]; }

Run-time representations

- XJ and Xobe use a DOM representation
  - Mutable doubly linked tree
  - Useful for XJ (imperative modification of XML)

- Xact and Xtatic use a custom representation
  - Immutable singly linked tree
    - Sharing of substructures
    - Lazy concatenation for efficiency
  - Xact: [Christensen, Kirkegaard, Møller – BRICS RS-03-29]
  - Xtatic: [Levin – ICFP’03], [Gapeyev, Levin, Pierce, Schmitt – MS-CIS-03-43]
To Infinity and Beyond
Boolean object types

- Needed for precise type inference of bound variables

\[
\text{case } [(\langle A \ x \rangle/ \mid \langle B \ x \rangle/)] : \ldots
\]

\(x\) should have type \(A \mid B\)

- Integrates nicely with an homogeneous compilation framework: only need to extend the typechecker.

- Current work extends FJ [Igarashi, Pierce, Wadler – OOPSLA’99] with union [Nagira, Igarashi – JSSST’03]
Filters

- Regular extension of pattern-matching clauses [Hosoya – PlanX’04]
- A clause is a pattern and an expression
- Example: transform every entry of an address book

```plaintext
static [[ Phbk ]] mkPhbk ([[ Addrbk ]] addr) {
    filter addr {
        <addrbk>
        ( <entry>Acq k, Name n, Tel t, any</entry> {<entry>k, n, t</entry>} )*
        </addrbk>
    }
}
```

- Similar to Cduce map or transform [Benzaken, Castagna, Frisch – ICFP’03]
- Integrates language features (loops) into pattern matching
Strategies of Pattern Matching

- **Greedy** [Frisch, Cardelli – PlanX’04]
  - Most common approach, simple to implement
  - Approximation of longest match

- **Lazy**
  - Very useful in practice (Find the first URL)
  - Recovered by stateful loops and first match policy
    ```
    while (cont) {
      match (curr) {
        case [[ url u, any rest ]]: curr = rest; ...
        case [[ one_char c, any rest ]]: curr = rest; ...
        case [[ ]]: cont = false
      }
    }
    ```
  - Interesting typing questions (Type of pcdata without any URL?)
Strategies of Pattern Matching

- **Multi**
  - Return all results
  - May bridge the gap between XPath and pattern matching

- **Deep**
  - Apply a transformation anywhere in the tree
    - Extension of filters with vertical recursion
  - Avoids boilerplate code
  - Challenging design and typing issues
Deeper Integration with OO

Types

▸ Mixing nominal and structural systems

▸ Integration of structural regular subtyping with languages that have structural OO subtyping (OCaml: CamlDuce?)

Sequences as objects

▸ XJ: sequences are Java lists
  ➤ sequence.size()

▸ Scala: For-Comprehensions
  ➤ List to list transformation
  ➤ for {val p <- persons; p.age > 20} yield p.name
  ➤ Defined using map, filter, and flatMap
  ➤ not restricted to lists
Xen and the Art of Language Design?

[Meijer, Schulte, Bierman – XML’03]

- Aims at a tight integration of OO, XML, and SQL (for C#)
- Includes Streams, Tuples, Union, Join Patterns (asynchronous programming)
- Map, Filter, and Fold on streams
- More details on the type system?
  - Aim at a seamless integration
    - No distinction between old and new types
  - What kind of subtyping integration?
    - Challenging issue
Take-home points

- **Regular types** are an expressive data model for XML
- **Type systems and subtyping integration** are crucial for a tight coupling of the two data models
- We need a better understanding of the relationship between nominal and structural subtyping
Do you want to know more?

**Xobe** [http://www.ifis.mu-luebeck.de/projects/XOBE/XOBE.html](http://www.ifis.mu-luebeck.de/projects/XOBE/XOBE.html) (in German)

**XJ** [http://www.google.com/search?hl=en&q=xj%20xml](http://www.google.com/search?hl=en&q=xj%20xml)

**Xact** [http://www.brics.dk/~amoeller/Xact/](http://www.brics.dk/~amoeller/Xact/)

**Xtatic** [http://www.cis.upenn.edu/~bcpierce/xtatic/](http://www.cis.upenn.edu/~bcpierce/xtatic/)