XACT
Type-Safe XML Transformations in Java

Copyright © 2009 Anders Møller <amoeller@cs.au.dk>

What is XACT?

XACT is an API for writing XML transformations in Java

Building on Java, we get

- the strength of a general-purpose programming language
- a rich and well-known standard library
- platform independence

XACT adds the following features:

- **XML data** as first-class Java values, with high-level operations for data manipulation
- efficient **runtime** model
- **compile-time validation** of transformed XML documents

XML templates

XML data is represented as ***templates***

- well-formed XML fragments
- contain **gaps** (named / Java expressions)
- **first-class values**
- **immutable**

```
XML x =
  [[ <h:html>
       <h:head>
         <h:title><[TITLE]></h:title>
       </h:head>
       <h:body bgcolor={color}>
         <h:h1><[TITLE]></h:h1>
         <[NAME]>
       </h:body>
     </h:html> ]];
```

Operations on XML templates

- **`parseTemplate`** – constructs template from constant string (syntactic sugar: `[[ … ]]`)
- **`parseDocument`** – imports XML data
- **`toTemplate`/`toDocument`** – exports XML data
- **`plug`** – inserts templates or strings into gaps
- **`get`** – selects subtemplates (using **XPath**)
- **`gapify`** – converts subtrees to gaps
- **`validate`** – runtime check of validity (like type cast)
- **`analyze`** – *compile-time* check for validity
- …

Operations on XML templates

- `append`
- `prepend`
- `has`
- `remove`
- `set`
- …

XACT unifies the *template* approach and the *DOM* approach

Example: PhoneList

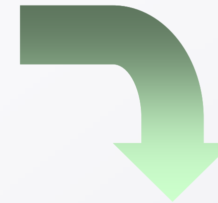
```
<cardlist xmlns="http://businesscard.org">

  <card>
    <name>John Doe</name>
    <email>john.doe@widget.inc</email>
    <phone>(202) 555-1414</phone>
  </card>

  <card>
    <name>Zacharias Doe</name>
    <email>zach@notmail.com</email>
  </card>

  <card>
    <name>Jack Doe</name>
    <email>jack@mailorder.edu</email>
    <email>jack@geemail.com</email>
    <phone>(202) 456-1414</phone>
  </card>

</cardlist>
```



My Phone List

- **John Doe**, phone: (202) 555-1414
- **Jack Doe**, phone: (202) 456-1414

The following solution isn't the simplest possible, but it shows a lot of XACT features…

Example: PhoneList (1/4)

```
import java.io.*;
import dk.brics.xact.*;

public class PhoneList {
  static {
    XML.getNamespaceMap().put("h", "http://www.w3.org/1999/xhtml");
    XML.getNamespaceMap().put("b", "http://businesscard.org");
    XML.getNamespaceMap().put("s", "http://www.w3.org/2001/XMLSchema");
    XML.loadXMLSchema("file:xhtml1-transitional.dtd");
    XML.loadXMLSchema("file:bcard.xsd");
  }

  ...
}
```

Namespaces and schemas are specified declaratively

Example: PhoneList (2/4)

```
public static void main(String[] args)
     throws XMLException, IOException {
  Phonelist pp = new Phonelist();
  pp.setDefaultWrapper("white");
  XML cardlist = XML.parseDocument(new URL("file:bcard.xml"))
                    .validate("b:cardlist");
  XML xhtml = pp.transform(cardlist);
  xhtml = xhtml.analyze("h:html");
  System.out.println(xhtml.toDocument());
}
```

Example: PhoneList (3/4)

```
XML wrapper;

private void setDefaultWrapper(String color) {
  wrapper = [[<h:html>
                <h:head>
                  <h:title><[TITLE]></h:title>
                </h:head>
                <h:body bgcolor={color}>
                  <h:h1><[TITLE]></h:h1>
                  <[MAIN]>
                </h:body>
              </h:html>]];
}
```

Example: PhoneList (4/4)

```
public XML transform(XML cardlist) {
  return wrapper.plug("TITLE", "My Phone List")
                .plug("MAIN", makeList(cardlist));
}

public XML makeList(XML cardlist) {
  XML r = [[<h:ul><[CARDS]></h:ul>]]);
  for (Element c : cardlist.getElements("b:card[b:phone]")) {
    r = r.plug("CARDS", [[
     <h:li>
       <h:b><{ c.getString("b:name") }></h:b>,
       phone: <{ c.getString("b:phone") }>
     </h:li>
     <[CARDS]>
    ]]);
  }
  return r.close();
}
```

Example: PhoneList (4/4)

```
public XML transform(XML cardlist) {
  return wrapper.plug("TITLE", "My Phone List")
                .plug("MAIN", makeList(cardlist));
}

public XML m
  XML r =
  for (Ele                                          :phone]")) {
    r = r.
     <h:li>
       <h:b
       phone
     </h:li>
     <[CARDS]
    ]]);
  }
  return r.close();
}
```

XACT analysis:

VALID!

Catching errors with the program analyzer

```
...
XML r = [[      <[CARDS]>      ]]);
...
```

```
*** Validation error
Source: element
  {http://www.w3.org/1999/xhtml}body at
  PhoneList line 41 column 31
Schema: file:xhtml1-transitional.dtd line
  913 column 26
Error: invalid child:
  {http://www.w3.org/1999/xhtml}li
```

Typed gaps

```
<h:html>
  <h:head>
    <h:title><[s:string TITLE]></h:title>
  </h:head>
  <h:body bgcolor={color}>
    <h:h1><[s:string TITLE]></h:h1>
    <[h:Flow MAIN]>
  </h:body>
</h:html>
```

A ***typed*** gap can only be plugged with a valid value

Optional type annotations

- Declares a variable holding *any* XML template:

```
XML foo;
```

- **Annotated** type:

```
@Type("S") XML foo;
```

  – the value of `foo` must have schema type *S*

- Annotated type **with gaps**:

```
@Type("S[T1 g1,...,Tn gn]") XML foo;
```

  – the value of `foo` must have schema type *S* if every gap g_i is plugged with a value of type T_i

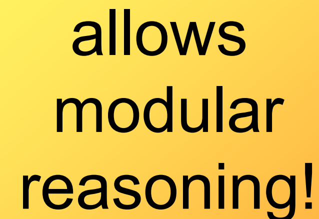
Example: PhoneList2

```
public @Type("h:html[s:string TITLE, h:Flow MAIN]") XML wrapper;
```

Example: PhoneList2

```
public @Type("h:html") XML transform(@Type("b:cardlist") XML cardlist) {
  return wrapper.plug("TITLE", "My Phone List")
                .plug("MAIN", makeList(cardlist));
}
```

Annotated XML types are implicit `validate` & `analyze` instructions



Runtime representation of XML templates

- Obtaining reasonable **runtime efficiency** of the XML template operations is not trivial (because of immutability)

- ... but it's possible ☺

The XACT program analyzer

- Checks expressions marked with `analyze`
- Checks `plug` operations for **gap types**
- Checks assignments and method input/output for **type annotations**

  – exact answers are impossible (Rice's theorem), we settle for **conservative approximations**!

The main challenges

1. Extract the **control-flow** from the Java program
2. Define a suitable **abstraction** of XML templates
3. Define **data-flow equations** modeling the operations

– *immutability of XML templates is crucial here!*

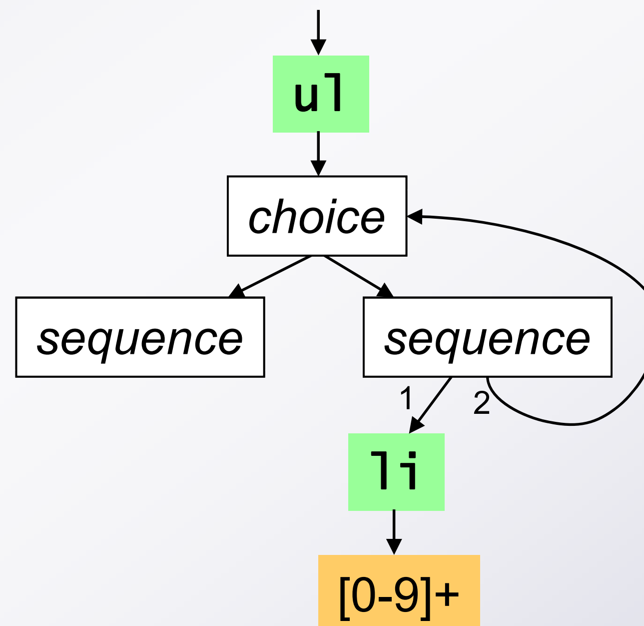
A suitable abstraction

For each XML expression, construct an **XML graph** that represents the possible values

XML graphs

- An XML graph represents **a set of XML templates**
  (like an "XML tree" with loops, choices, named gaps, and regexp text)

- *Example:* `ul` lists with zero or more `li` items that each contain an integer as text



From XACT programs to XML graphs

- **Constant** XML template → XML graph
- **XML Schema** type → XML graph
- Model all XML template **operations** and XPath expressions on XML graphs
- Check **inclusion** between XML graph and XML Schema type

– we omit the (many!) technical details...

- http://www.brics.dk/schematools/

Summary

- XACT is a Java API for writing XML transformations
- Key ideas:
  - **immutable XML templates**
  - **XPath for navigation**
  - **DOM-like operations**
  - **optional XML Schema type annotations**
  - **static program analysis for transformation validity**
- http://www.brics.dk/Xact/