xml transformations

stratego & xml

Martin Bravenboer

martin@mbravenboer.org

Institute of Information and Computing Sciences

University Utrecht

The Netherlands

contents: xml transformations

- introduction to xml
- generation of xml
- transformation of xml
- xml schema languages
- transformation of xml, postscript

contents: distributed services

- introduction to services
- aterm services
- xml-rpc

introduction to xml

xml: principles

syntax for labeled (at nodes) trees:

- platform and language independent
- application independent: documents and data

generic tools, languages and libraries:

- parsers, pretty-printers
- transformation and query languages
- schema languages and validators

xml: structure (1)

- elements
  - name (namespace)
  - attributes
    - name (namespace)
    - value: string, int, tokens
- character data

running example

```
<results>
  <lecture>
    <date>2003-01-06</date>
    <result> <nr>9512721</nr> <sufficient/>  </result>
    <result> <nr>9609911</nr> <sufficient/>  </result>
    <result> <nr>0241679</nr> <good/>        </result>
  </lecture>

  <lecture>
    <date>2003-01-10</date>
    <result> <nr>9704337</nr> <good/>        </result>
    <result> <nr>9708928</nr> <sufficient/>  </result>
  </lecture>
</results>
```

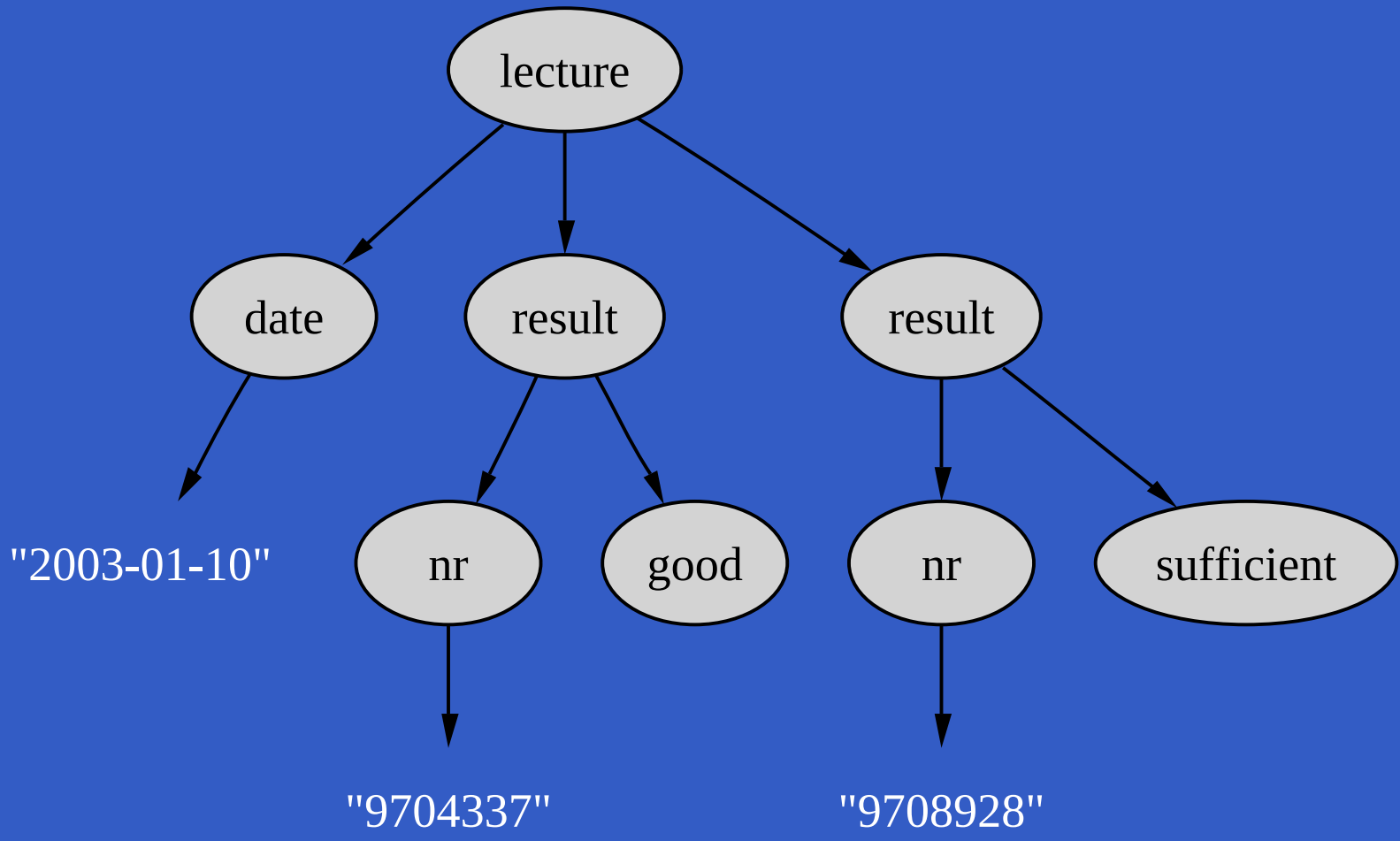
xml: syntax

```
<lecture>
  <date>2003-01-10</date>

  <result>
    <nr>9704337</nr>
    <good/>
  </result>

  <result>
    <nr>9708928</nr>
    <sufficient/>
  </result>
</lecture>
```

xml: tree view



xml: application

exchange of structured, tree-like data between *software components*

- in different languages
- on different platforms
- at different locations
- in different locales

note the similar application of aterms

the evolution of a language?

xml is often bashed for being a verbose, overhyped piece of crap.

why?

the evolution of a language?

Descartes

$2x$

the evolution of a language?

Church

$$\lambda x.2x$$

the evolution of a language?

McCarthy

```
(lambda (x) (* 2 x))
```

the evolution of a language?

W3C

```
<lambda-term>
  <var-list>
    <var>x</var>
  </var-list>
  <application>
    <const>*</const>
    <args>
      <const>2</const>
      <var>x</var>
    </args>
  </application>
</lambda-term>
```

the evolution of a language?

CWI

```
lambda-term(
  var-list( [ var("x") ] )
, application(
    const("*")
  , args( [ const(2), var("x") ] )
  )
)
```

is xml verbose?

yes, but

- compare xml to *abstract syntax*, not *concrete*
- do not write xml

unfortunately

- used as a concrete syntax
- use of concrete syntax *in* xml

  xpath, xml schema regexps, css, svg path data
- use of ‘structure’ in attribute values

  namespace prefix, tokens

generation of xml

generation of xml: syntax

xml is just syntax: need a syntax definition to generate xml with *concrete syntax* for xml.

```
"<" QName Attribute* "/>"
   -> Element {cons("EmptyElement")}

"<" QName Attribute* ">" Content*  "</" QName ">"
   -> Element {cons("Element")}

QName "=" AttValue
   -> Attribute {cons("Attribute")}
```

xml in stratego

```
"%>" Document "<%" -> StrategoTerm {cons("ToTerm")}
"%>" Content  "<%" -> StrategoTerm {cons("ToTerm"), prefer}
"%>" Content* "<%" -> StrategoTerm {cons("ToTerm")}

"<%" Strategy                  "%>" -> Content    {cons("FromApp")}
"<%" Strategy "::" "content"   "%>" -> Content    {cons("FromApp")}
"<%" Strategy "::" "*"         "%>" -> Content*   {cons("FromApp")}
"<%" Strategy "::" "content*"  "%>" -> Content*   {cons("FromApp")}

"<%" Strategy                  "%>" -> Attribute  {cons("FromApp")}
"<%" Strategy "::" "*"         "%>" -> Attribute* {cons("FromApp")}
"<%" Strategy                  "%>" -> AttValue   {cons("FromApp")}

"<%" Strategy "::" "cdata"     "%>" -> CharData   {cons("FromApp")}
```

hello world

```
!%>
  <?xml version="1.0"?>

  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>Hello world</title>
    </head>
    <body>
      <h1>Hello world!</h1>
      <p>
        This page is generated by
        <a href="http://www.stratego-language.org">Stratego</a>
        at <% create-time :: cdata %>
      </p>
    </body>
  </html>
<%
```

transformation of xml

how to transform?

Can we transform xml with concrete syntax? No!

xml syntax contains many irrelevant details:

- attribute order
- default attributes
- namespace prefix
- optional content
- comments, processing instructions

what to transform?

- *xml* - xml syntax
- *xml-info* - xml document
- *data* - data of an xml document

running example

```
<results>
  <lecture>
    <date>2003-01-06</date>
    <result> <nr>9512721</nr> <sufficient/>  </result>
    <result> <nr>9609911</nr> <sufficient/>  </result>
    <result> <nr>0241679</nr> <good/>        </result>
  </lecture>

  <lecture>
    <date>2003-01-10</date>
    <result> <nr>9704337</nr> <good/>        </result>
    <result> <nr>9708928</nr> <sufficient/>  </result>
  </lecture>
</results>
```

running example: xml

```
Document(
  Prologue(None, [], None)
, Element(QName(None, "results"), [],
  [ Element(QName(None, "lecture"), [],
    [ Element(QName(None, "date"), []
      , [ Text("2003-01-06") ]
      , QName(None, "date")
      )
    , Element(QName(None, "result"), []
      , [ Element(QName(None, "nr"), []
          , [ Text("9512721") ]
          , QName(None, "nr"))
        , EmptyElement(QName(None, "sufficient"), [])
        ]
      , QName(None, "result")
      )
  ...
```

running example: xml-info

```
Document(
  Element(Name(None, "results"), []
  , [ Element(Name(None, "lecture"), []
      , [ Element(Name(None, "date"), []
          , [ Text("2003-01-06") ])
        , Element(Name(None, "result"), []
          , [ Element(Name(None, "nr"), []
              , [ Text("9512721") ])
            , Element(Name(None, "sufficient"), [], [])
            ])
        , Element(Name(None, "result"), []
          , [ Element(Name(None, "nr"), []
              , [ Text("9609911") ])
            , Element(Name(None, "sufficient"), [], [])
            ])
    ...
```

signature of xml-info

```
module xml-info
imports option

signature
  constructors
    Document  : Element -> Document

    Attribute : Name * String -> Attribute
    Element   : Name * List(Attribute) * List(Content) -> Element

    Name      : Option(Namespace) * String -> Name
    Namespace : String -> Namespace

    // Element -> Content
    Element   : Name * List(Attribute) * List(Content) -> Content
    Text      : String -> Content
```

running example: data

```
results(
  lecture(
    date("2003-01-06")
  , result(nr("9512721"), sufficient)
  , result(nr("9609911"), sufficient)
  , result(nr("0241679"), good)
  )
, lecture(
    date("2003-01-10")
  , result(nr("9704337"), good)
  , result(nr("9708928"), sufficient)
  )
)
```

running example: data signature?

```
module results-data

signature
  constructors
    results      : Lecture * Lecture * ... -> Results
    lecture      : Date * Result * Result * ... -> Lecture
    date         : String -> Date
    result       : Number * Mark -> Result
    nr           : String -> Number
    good         : Mark
    sufficient   : Mark
    insufficient : Mark
```

xml-info → data

```
xml-info2data =
    ?Document(<id>)
  ; topdown-wannos( try(Implode) )

Implode:
  Element(Name(_, s), [atts*], children) -> s#(children){atts*}

Implode:
  Text(s) -> s

Implode:
  Attribute(Name(_, s), value) -> (s, value)

topdown-wannos(s) =
  rec x(
    topdown(s; id{map(x)} )
  )
```

xml ∼ aterm

How similar are xml and aterm?

- node labeled trees
- regular tree languages and grammars
- ‘extensible’
- exchange of data: used in very similar ways

xml ∼ aterm

aterm *instances* are more structured

- aterm: list, tuples
- aterm: string, int, real, blob
- xml: attributes not structured

running example: data

```
results(
  [ lecture(
      date("2003-01-06")
    , [ result(nr(9512721), sufficient)
      , result(nr(9609911), sufficient)
      , result(nr(241679),  good)
      ]
    )
  , lecture(
      date("2003-01-10")
    , [ result(nr(9704337), good)
      , result(nr(9708928), sufficient)
      ]
    )
  ]
)
```

xml ∼ aterm

different conventions

- xml: namespaces
- aterm: annotations not used for non-structural information
- xml: interleaving, mixed-content
- stratego: optional data with `Option`

structuring xml

xml documents can be structured by creating an *interpretation* against a *schema*.

xml schema languages

why schema languages?

xml: create your own markup language

- schemas define:
  - $\Rightarrow$ your own markup language
  - $\Rightarrow$ a set of permissible xml documents

    like context-free grammars define a set of strings
- schema language: language for defining schemas
- validation of instance documents

running example: DTD

```
<!ELEMENT results (lecture*)>
<!ELEMENT lecture (date, result*)>
<!ELEMENT date    (#PCDATA)>

<!ELEMENT result (nr, (good | sufficient | insufficient))>
<!ELEMENT nr (#PCDATA)>

<!ELEMENT good         EMPTY>
<!ELEMENT sufficient   EMPTY>
<!ELEMENT insufficient EMPTY>
```

running example: W3C XML Schema

```
<xs:schema ...>
  <xs:element name="results">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="lecture"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="lecture">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="date"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="result"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

running example: W3C XML Schema

```
  <xs:element name="result">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="nr"/>
        <xs:choice>
          <xs:element name="good" type="empty"/>
          <xs:element name="sufficient" type="empty"/>
          <xs:element name="insufficient" type="empty"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="date" type="xs:date"/>
  <xs:element name="nr"   type="xs:int"/>
  <xs:complexType name="empty"/>
</xs:schema>
```

running example: RELAX NG

```
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"
start = Results

Results = element results {Lecture*}
Lecture = element lecture {Date, Result*}
Result  = element result  {Number, Mark}

Date    = element date {xsd:date}
Number  = element nr   {xsd:int }

Mark =
    element good         {empty}
  | element sufficient   {empty}
  | element insufficient {empty}
```

running example: RELAX NG

```
<grammar ... >
  <start>
    <ref name="Results"/>
  </start>

  <define name="Results">
    <element name="results">
      <zeroOrMore> <ref name="Lecture"/> </zeroOrMore>
    </element>
  </define>

  <define name="Lecture">
    <element name="lecture">
      <ref name="Date"/>
      <zeroOrMore> <ref name="Result"/>  </zeroOrMore>
    </element>
  </define>
```

running example: RELAX NG

```
  <define name="Mark">
    <choice>
      <element name="good">         <empty/>  </element>
      <element name="sufficient">   <empty/>  </element>
      <element name="insufficient"> <empty/>  </element>
    </choice>
  </define>

  <define name="Date">
    <element name="date">  <data type="date"/> </element>
  </define>

  <define name="Number">
    <element name="nr">  <data type="int"/>  </element>
  </define>
</grammar>
```

xml interpretation

regular trees

regular tree grammar :

- set N of non-terminals
- set T of terminals
- set P of productions $n \rightarrow t\ r$ where
  - $n \in N$
  - $t \in T$
  - r regexp over N

regular tree language :

set of trees generated by an rtg

rtg for running example

```
regular tree grammar
  start Results

  productions
    Results -> results (Lecture*)
    Lecture -> lecture (Date, Result*)
    Date    -> date    (String)
    Result  -> result  (Number, Mark)
    Number  -> nr      (Int)
    Mark    -> good          (E)
    Mark    -> sufficient    (E)
    Mark    -> insufficient (E)
```

xml schema $\Leftrightarrow$ rtg $\Leftrightarrow$ sig

DTD, W3C XML Schema and RELAX NG allow you to define a language in some subclass of the regular tree languages.

Stratego signatures $\sim$ regular tree grammars

interpretation of xml

- introduce lists and tuples
- introduce `None` and `Some`
- parse strings to ints, reals

composition:

1. compile *schema* to *rtg*
2. parse *xml* to *xml-info*
3. interpet xml-info against rtg
4. implode *interpretation* to *aterm*

generating signatures

```
module results
imports list-cons option
signature
  constructors
    results       : List(Lecture) -> Results
    lecture       : Date * List(Result) -> Lecture
    date          : String -> Date
    result        : Number * Mark -> Result
    nr            : Int -> Number
    good          : Mark
    sufficient    : Mark
    insufficient  : Mark
```

transformation of xml, postscript

now we can transform!

transforming xml: xml $\Rightarrow$ aterms

native data structure of Stratego

compare to:

- xml databinding (JAXB, Castor)
- HaXml's Dtd2Haskell

XSLT, XQuery and XDuce: xml is native data structure.

running example to xhtml (1)

```
io-lecture-results2xhtml =
  xtc-io-wrap(
    xtc-xml-interpret(!"lecture-results.artg")
  ; xtc-io-transform(lecture-results2xhtml)
  ; xtc-pp-xml
  )

lecture-results2xhtml :
  results(lectures) ->
    %>
      <?xml version="1.0"?>
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" ... >

      <html xmlns="http://www.w3.org/1999/xhtml">
        ....
      </html>
    <%
```

running example to xhtml (2)

```
<table border="1" cellpadding="5">
  <thead>
    <tr>
      <th>Student</th>
      <% !lectures; map(!%>
        <th>
          <% ?lecture(date(<id>), _) :: cdata %>
        </th>
      <% ) :: content* %>
    </tr>
  </thead>
   ...
```

distributed services

Why go distributed?

exchange of structured, tree-like data between software components

- Why not consider *distributed* components?
- xml web services

  SOAP, XML-RPC, RDF
- exchange formats are useful in distributed environments

ATermService

- is accessible at some URL using HTTP
- takes an ATerm input in the body of a HTTP POST request
- returns an ATerm output in a HTTP response

$\Rightarrow$ Can be implemented and accessed in any language.

calculator, server-side

```
strategies

  cgi-calculator =
    cgi-service-wrap(
      service-just-post(calculator-service)
    )

  calculator-service =
    cgi-aterm-io(calc)

strategies

  calc = innermost(Simplify)

  Simplify : Plus(Int(x), Int(y)) -> Int(<add> (x, y))
  Simplify : Mul( Int(x), Int(y)) -> Int(<mul> (x, y))
```

calculator, client-side

```
module inline-calc-client
imports http-client Exp

strategies

  io-inline-calc-client =
    <http-transform(!URL("http://127.0.0.1/cgi-bin/calculator"))>
      Plus(Int(1), Int(2))
```

or:

```
  io-calc-client =
    xtc-io-wrap(
      xtc-http-transform(!URL("http://127.0.0.1/cgi-bin/calculator"))
    )
```

calculator, request

```
POST /cgi-bin/calculator HTTP/1.1
User-Agent: curl/7.10.3 (i686-pc-linux-gnu) libcurl/7.10.3 OpenSSL/0.9.6a
Host: localhost:8080
Pragma: no-cache
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Content-Length: 21
Content-Type: application/x-www-form-urlencoded

Plus(Int(1), Int(2))
```

Content-Type should be changed

calculator, response

```
HTTP/1.1 200 "OK"
Date: Thu, 20 Feb 2003 22:44:19 GMT
Server: Apache/2.0.39 (Unix)
Content-Length: 7
Content-Type: text/plain; charset=ISO-8859-1

Int(3)
```

dynamic xhtml

```
cgi-hello-xhtml =
  cgi-service-wrap(service-just-get(hello-xhtml-service))

hello-xhtml-service =
  cgi-xml-output(!HTML(),
    !%>
      <?xml version="1.0"?>
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" ...>

      <html xmlns="http://www.w3.org/1999/xhtml">
        ...
      </html>
    <%
  )
```

aterm database interface

1. database server as ATermService:
   - Java ATermServices
   - servlet container
   - JDBC drivers
   - returns ResultSets in aterm format
   - generic!
2. Stratego service transforms data to presentation

xml-rpc: http call

```
POST /RPC2 HTTP/1.1
User-Agent: curl/7.10.3 (i686-pc-linux-gnu)
Host: betty.userland.com
Content-Length: 197

<?xml version="1.0" ?>

<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value>
        <i4>40</i4>
      </value>
    </param>
  </params>
</methodCall>
```

xml-rpc: http response

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 143
Content-Type: text/xml
Date: Sun, 16 Feb 2003 23:10:47 GMT
Server: UserLand Frontier/9.0-WinNT

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>South Carolina</value>
    </param>
  </params>
</methodResponse>
```

communicating in xml-rpc

```
xtc-xml-rpc(url) =
    xtc-as-xml(!"xml-rpc-response.artg",
      xtc-http-transform(url)
    )

xml-rpc(url) =
  xtc-oi-wrap(xtc-xml-rpc(url))

xtc-as-xml(artg, s) =
    xtc-data2xml
  ; xtc-pp-xml
  ; s
  ; xtc-xml-interpret(artg)
```

statename client

```
io-xml-rpc-statename =
  option-wrap(general-options, xtc-output(statename))

statename =
    <write-to>
      methodCall(
        methodName("examples.getStateName")
      , params(
         [ param(value(i4("40"))) ]
        )
      )
  ; xtc-xml-rpc(!URL("http://betty.userland.com/RPC2"))
```

statename client, without xml-rpc

```
module statename-client
imports xml-rpc-client new-options

strategies

  io-statename-client =
    output-wrap(<statename> 40)

  // :: Int -> String
  statename =
    one-param-one-result(
      as-xml-rpc(!"examples.getStateName",
        xml-rpc(!URL("http://betty.userland.com/RPC2"))
      )
    )
```

google client, without xml-rpc

```
io-google-client =
  output-wrap(<google-search; just-urls> "stratego xml transformation")

google-search =
  <as-xml-rpc(!"googleGateway.search",
    xml-rpc(!URL("http://google.xmlrpc.com/RPC2"))
  )> [<id>, 0, 10, "", "", False(), "latin1", "latin1", <google-key> ()]

just-urls =
  collect(?URL(_))

google-key =
  <xtc-find-file; read-from> "google.key"
```