HaXML: Haskell and XML

Jevgeni Võssotski

21.05.2007

Why Functional Programming

- Declarative nature
- Meta-language features(a language to build domain-specific language on top of)
- XML defines documents in terms of their logical features rather than particular rendering procedures, FP strives to specify computations in mathematical terms rather than machine- or recipe-oriented terms.
- XSLT is a subset of Scheme

HaXML components

- *Combinators* - a combinator library for generic XML document processing, including transformation, editing, and generation
- *DtdToHaskell* - a tool for translating any valid XML DTD into equivalent Haskell types.

Combinators

- Data modelling

```
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
             | CText String
```

- The filter type

```
type CFilter = Content -> [Content]
```

- Program wrapper

```
processXmlWith :: CFilter -> IO ()
```

Basic filters

- Predicates

```
none,                 zero/failure
keep,                 identity/success
elm,                  tagged element?
txt                   plain text?
   :: CFilter
tag,                  named element?
attr                  element has attribute?
   :: String -> CFilter
attrval               element has attribute/value?
   :: (String,String) -> CFilter
```

Basic filters

- Selection

```
children :: CFilter

showAttr, (?) :: String -> Cfilter
```

- Construction

```
literal, (!)   :: String -> CFilter

mkElem :: String -> [CFilter] -> CFilter

mkElemAttrs :: String -> [(String,CFilter)]
                      -> [CFilter] -> CFilter
```

Filter combinators

```
o,              Irish composition
(|||),          append results
with,           guard
without,        negative guard
(/>),           interior search
(</),           exterior search
(|>|)           directed choice
   :: CFilter -> CFilter -> CFilter

f `o` g = concat . map f . g
f ||| g = \c -> f c ++ g c
f `with` g = filter (not.null.g) . f
f `without` g = filter (null.g) . f
f /> g = g `o` children `o` f
f </ g = f `with` (g `o` children)
f |>| g = f ?> f :> g
```

Filter combinators

```
cat      --concatenate results
   :: [CFilter] -> CFilter

cat fs = \c -> concat . map (\f -> f c) fs

(?>)     --if-then-else choice
   :: CFilter -> ThenElse CFilter -> CFilter

data ThenElse a = a :> a
p ?> f :> g = \c -> if (not.null.p) c
                    then f c
                    else g c
```

Recursive combinators

```
chip,       ``in-place'' application to children
deep,       recursive search (topmost)
deepest,    recursive search (deepest)
multi,      recursive search (all)
foldXml     recursive application
   :: CFilter -> CFilter

deep f = f |>| (deep f `o` children)
deepest f = (deepest f `o` children) |>| f
multi f = f ||| (multi f `o` children)
foldXml f = f `o` (chip (foldXml f))
```

Examples (XSL -> HaXML)

- Copying Elements to the Output

```
<xsl:template match="title">
  <xsl:copy>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
->
multi(tag "title")
```

- Deleting Elements

```
<xsl:template match="nickname">
</xsl:template>
->
foldXml(keep `without` tag "nickname")
```

Examples (XSL -> HaXML)

- Changing Element Names

```
<xsl:template match="article">
  <html>
    <xsl:apply-templates/>
  </html>
</xsl:template>
->
foldXml(mkElem "html" [children] `when` tag "article")
OR
foldXml(replaceTag "html" `o` tag "article")
```

Algebraic properties of combinators

- **Irish composition**

```
f `o` (g `o` h) = (f `o` g) `o` h          (associativity)
none `o` f = f `o` none = none                (zero)
keep `o` f = f `o` keep = f                   (identity)
```

- **Guards**

```
f `with` keep = f                             (identity)
f `with` none = none `with` f = none          (zero)
(f `with` g) `with` g = f `with` g          (idempotence)
(f `with` g) `with` h = (f `with` h) `with` g (promotion)
(f `o` g) `with` h = (f `with` h) `o` g       (promotion)
f `without` keep = none `without` f = none    (zero)
f `without` none = keep      (identity)
(f `without` g) `without` g = f `without` g  (idempotence)
(f `without` g) `without` h = (f `without` h) `without` g
                                              (promotion)
(f `o` g) `without` h = (f `without` h) `o` g (promotion)
```

Algebraic properties of combinators

- **Path selectors**

```
f /> (g /> h) = (f /> g) /> h          (associativity)
none /> f = f /> none = none           (zero)
keep /> f = f `o` children
f /> keep = children `o` f
keep /> keep = children
none </ f = f </ none = none           (zero)
f </ keep = f `with` children
(f </ g) </ g = f </ g                 (idempotence)
(f </ g) /> g = f /> g                 (idempotence)
```

- **Directed choice**

```
(f |>| g) |>| h = f |>| (g |>| h)      (associativity)
keep |>| f = keep
none |>| f = f |>| none = f            (identity)
f |>| f = f                            (idempotence)
```

Algebraic properties of combinators

- **Recursion**

```
deep keep = keep          (simplification)
deep none = none          (simplification)
deep children = children     (simplification)
deep (deep f) = deep f       (depth law)
```

- **Misc**

```
elm |>| txt = txt |>| elm = keep      (completeness)
elm `o` txt = txt `o` elm = none      (excl. middle)
children `o` elm = children
children `o` txt = none
```

Labellings

```
type LabelFilter a = Content -> [ (a,Content) ]
numbered     :: CFilter -> LabelFilter Int
interspersed :: a -> CFilter -> a
                                -> LabelFilter a
tagged       :: CFilter -> LabelFilter String
attributed   :: CFilter ->
                  LabelFilter [(String,String)]
`oo` :: (a->CFilter) -> LabelFilter a -> Cfilter
```

- Example

```
catno `oo` numbered (deep (tag "catalogno"))
catno n =
   mkElem "LI"
   [ ((show n++". ")!),  ("label"?),  ("number"?)
   , (" ("!),  ("format"?),  (")"!) ]
```

DdtToHaskell & Xml2Haskell

```
<?xml version='1.0'?>
<!DOCTYPE album SYSTEM "album.dtd" [
<!ELEMENT album (title, artist, recordingdate?,
                 coverart, (catalogno)+,
                 personnel, tracks, notes) >
<!ELEMENT title #PCDATA>
<!ELEMENT artist #PCDATA> ...

->

module AlbumDTD where
data Album =
    Album Title Artist (Maybe Recordingdate)
          Coverart [Catalogno] Personnel
          Tracks Notes
newtype Title = Title String
newtype Artist = Artist String ...
```

Alternatives: special purpose languages

- Statically typed functional languages which have XML documents as their basic data types: XMλ, XDuce, Cduce
- XMLambda example

```
myTitle :: title
myTitle = <title>Hello world in XMLambda</title>
<html>
<head>
        <%= myTitle %>
</head>
<body>
        <h1>Hello world</h2>
</body>
</html>
```

More HaXml examples

```
module Main where
import Text.XML.HaXml

main = processXmlWith (wrap (img `oo` numbered allImgs))
allImgs = multi(tag "img")
wrap f = html [ hbody [ htable [ f ]]]
img n = hrow [hcol [ (n!) ], hcol [ ("src"?) ]]
```