XML Verarbeitung mit einer in Haskell eingebetteten DSL

Manuel Ohlendorf (xi2079)

Übersicht

Übersicht

Motivation

- Heterogene Systemlandschaft vs. Einheitliche Kommunikation

  ➔ XML als Schnittstelle

- Baumstruktur von XML

  ➔ Listenverarbeitung mit Haskell

- Allgemeiner und flexibler Ansatz

  ➔ Haskell XML Toolbox mit Domain Specific Language

Topics

1	Einführung
2	Datenstruktur
3	Verarbeitung
4	Vergleich mit anderen Verfahren
5	Fazit

Idee

- Baum, mit Blättern und Knoten ➔ verkettete Liste
- Neuer Datentyp: Tree

```
data Tree =  Text  String
           | Cmt   String
           | Cdata String
                .
                .
           | Tag  String String [Tree]
```

- Für jedes Element einen neuen Datentyp
- Probleme?

Probleme mit Idee Nr.1

```
data Tree =  Text  String
           | Cmt   String
           | Cdata String
               .
               .
           | Tag  String String [Tree]
```

- Für jede „Version“ von *Tree* neue Funktionen zur Verarbeitung nötig
- Funktionen lassen sich nicht kombinieren
- Mehrere Traversierungsroutinen

➔ **Schlecht!**

Idee Nr.2

- Universelles Datenmodell
- Baumstruktur von den Informationen der Knoten trennen

➔ Generischer n-stelliger Baum (Rose Tree)

```
data NTree node  = NTree node (NTrees node)
                   deriving (Eq, Ord, Show, Read)

type NTrees node = [NTree node]
```

Idee Nr.2 für XML-Bäume

- Basierend auf *NTree*, ein neues Typsynonym:

```
type XmlTree  = NTree XNode

type XmlTrees = NTrees XNode
```

- Repräsentiert ein XML-Baum
- *XNode* definiert alle möglichen XML-Elemente

Definition von *XNode*

```
data XNode = XText          String
           | XCharRef       Int
           | XEntityRef     String
           | XCmt           String
           | XCdata         String
           | XPi            TagName XmlTrees
           | XTag           TagName XmlTrees
           | XDTD           DTDElem Attributes
           | XAttr          AttrName
           | XError         Int String
             deriving (Eq, Ord, Show, Read)
```

Hilfstypen für *XNode*:

```
type TagName  = QName
type AttrName = QName
type Attributes = AssocList String String
```

Der Datentyp *QName* / Exkurs Namensräume

- Namensräume (Namespaces) für eindeutige XML-Elemente, keine Element-Kollision
  - Kollidierende Elemente besitzen den gleichen Namen, haben aber eine unterschiedliche Bedeutung

```
<xsl:stylesheet xmlns:xsl='http://www.w3c.org/1999/XSL/Transform'>
    ...
</xsl:stylesheet>
```

```
data QName = QN { namePrefix     :: String
                , localPart      :: String
                , namespaceUri   :: String
                }
                deriving (Eq, Ord, Show, Read)
```

Der Datentyp *DTDElem*

- Verwendung der gleichen Datenstruktur für DTDs wie für XML-Dokumenten
- Nur ein spezieller Knotentyp in *XNode*

```
data DTDElem = DOCTYPE
             | ELEMENT
             | CONTENT
             | ATTLIST
             | ENTITY
             | PENTITY
             | NOTATION
             | CONDSECT
             | NAME
             | PEREF
               deriving (Eq, Ord, Show, Read)
```

Topics

1	Einführung
2	Datenstruktur
3	Verarbeitung
4	Vergleich mit anderen Verfahren
5	Fazit

Ansatz

- Aufteilung der verarbeitenden Funktionen in:
  - Einfache Funktionen, auf Knoten-Ebene
    - Zum Selektieren
    - Zum Testen
    - Zum Modifizieren
  - Zusammengesetzte Funktionen, auf der Ebene der Teilbäume
  - Kombination der einfachen Funktion zur Erzeugung beliebiger komplexer Funktionen

➔ Domain Specific Language

Exkurs: Domain Specific Language (DSL)

- Sprache, die für einen bestimmten Zweck entwickelt wurde, Bsp:
  - Macros, Csound für die Generierung von Audiofiles
- Problemorientierte Sprache im Gegensatz zu general-purpose Sprachen wie Java, C etc.
- Das Problem hier:

XML-Verarbeitung

Ein Typ für alles

- Zur Verarbeitung werden Filter verwendet
- Jeder Filter vom gleichen Typ:

```
type TFilter node  = NTree node  -> NTrees node
type TSFilter node = NTrees node -> NTrees node
```

- Resultate der Prädikate:
  - Leere Liste ([ ]) = False
  - Nicht-leere Liste ([_:_]) = True

```
type XmlFilter  = TFilter XNode
type XmlSFilter = TSFilter XNode
```

Einfache Filter

Nullfilter:

```
none  :: a -> [b]
none _  = []
```

Identitätsfilter:

```
this  :: a -> [a]
this n  = [n]
```

Selektor-Filter

```
getChildren  :: NTree node -> NTrees node
getChildren (NTree _ cs)
              = cs
```

Filter zum Modifizieren:

```
replaceNode  :: node -> TFilter node
replaceNode n (NTree _ cs)
               = [NTree n cs]
```

Filter-Kombinatoren

Logisches 'Und':

```
o  :: TFilter f -> TFilter f -> TFilter f
o  :: (a -> [b]) -> (c -> [a]) -> (c -> [b])
f `o` g
      = concatMap f . g
```

Logisches 'Oder':

```
(+++)  :: (a -> [b]) -> (a -> [b]) -> (a -> [b])
f +++ g
      = \ t -> f t ++ g t
```

Auswahl-Filter:

```
when  :: (a -> [a]) -> (a -> [a]) -> (a -> [a])
f `when` g
      = iff g f this
```

Rekursive Filter

- Zwei Strategien zum Traversieren:
  - Bottom-Up: Von der Wurzel zur Spitze
  - Top-Down: Von der Spitze zur Wurzel

```
processBottomUp  :: TFilter node -> TFilter node
processBottomUp f
      = processChildren (processBottomUp f) .> f
```

```
processTopDown  :: TFilter node -> TFilter node
processTopDown f
      = f .> processChildren (processTopDown f)
```

Beispiel

- Entfernen aller *XText*-Elemente aus einem XML-Baum
- Schritte:
  1. Prädikat; ob Knoten ein *XText* ist, oder nicht
  2. Filter; das Prädikat als *XmlFilter*
  3. Filter zum Entfernen
  4. Rekursiver Filter

Topics

HXML und HaXML

- HXML
  - generischer Baum für XML-Dokumente
  - DTDs nicht als Baum

  ➔ keine einheitliche Verarbeitung
- HaXML
  - für jedes XML-Element eigener Datentyp
  - Verarbeitung mit Filtern

  ➔ Filter nicht flexibel und allgemein definierbar

Topics

1	Einführung
2	Datenstruktur
3	Verarbeitung
4	Vergleich mit anderen Verfahren
5	Fazit

Fazit

- Kombination der beiden Ansätze HaXML und HXML
- flexibles Design
- leicht erweiterbar
- Fragen??