Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 14.01.09: Datenmodellierung mit XML

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke
  - Datenmodellierung mit XML
  - Effizienzerwägungen
  - Grafik
  - Schluss

Inhalt

- Fallbeispiel: XML Feeds verarbeiten
- Protokolle und Standards:
  - XML
  - HTTP
  - RSS
- Haskell-Büchereien dazu

XML

Die Extensible Markup Language (engl. für "erweiterbare Auszeichnungssprache", abgekürzt XML, ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten. XML wird u. a. für den Austausch von Daten zwischen Computersystemen eingesetzt, speziell über das Internet.

Wikipedia

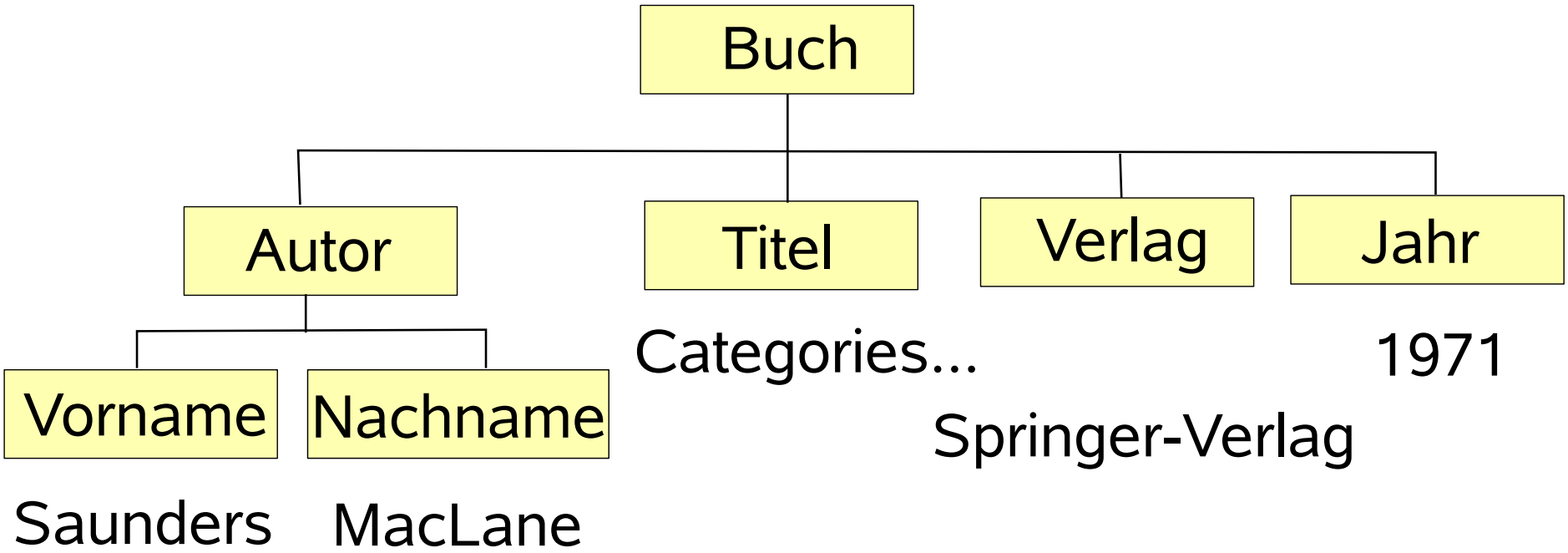
- Textbasiert
- Erweiterbar und anpassbar
- Generisch: unabhänging von Betriebssystem, Programmiersprache, Anwendung

Ein einfaches Beispiel

- Ein Buch hat
  - Autor(en) mit Vorname, Nachname (mind. einen)
  - Titel
  - Verlag, Erscheinungsjahr
  - Zusammenfassung (optional)
- Als XML:

```
<buch>
  <autor><vorname>Saunders</vorname>
         <nachname>MacLane</nachname></autor>
  <titel>Categories for the working mathematician</titel>
  <verlag>Springer-Verlag</verlag><jahr>1971</jahr>
</buch>
```

XML-Dokumente als Bäume



Mixed content

- Beispiel: Zusammenfassung mit ausgezeichneten Schlüsselwörtern

```
<zusammenfassung>
Das <keyword>Lehrbuch</keyword> über
<keyword>Kategorientheorie</keyword>,
eine abstrakte Disziplin der <keyword>Mathematik</keyword>
</zusammenfassung>
```

- Leerzeichen relevant (nur in mixed content)

Namen und Attribute

- Namen:
  - Folge von Buchstaben, Ziffern, Zeichen (keine Sonderzeichen)
  - Fängt an mit Buchstabe, Unterstrich, Zeichen
- Attribute:
  - Paar aus Namen und Werten
  - E.g. `<titel sprache="englisch">...</titel>`

XML-Schichten

1. Wohlgeformtheit (well-formedness)
   - Start-tag $\longleftrightarrow$ end-tag
   - Keine überlappenden Elemente
   - Genau ein Wurzelelement
   - Attribut-Werte in Anführungszeichen
   - Keine zwei Attribute mit gleichen Werten
   - $\Longrightarrow$ Text entspricht Grammatik, parsierbar
2. Gültigkeit (validity)
   - Dokument entspricht Dokumententyp
3. Spezifische Dokumemtentypen
   - E.g. `XHTML`, `SVG`, `XSD`

Dokumententypen

- Gibt Produktionsregeln für Elemente an
- Verschiedene Formate: DTDs, XML-Schema, Relax NG

DTD für das Buch

```
<!DOCTYPE buch [
  <!ELEMENT buch (autor+, titel, verlag, jahr, zusfassung?)>
  <!ELEMENT autor (vorname, nachname)>
  <!ELEMENT vorname (#PCDATA)>
  <!ELEMENT nachname (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ATTLIST title language CDATA #IMPLIED>
  <!ELEMENT zusfassung (#PCDATA|keyword)*>
  <!ELEMENT keyword (#PCDATA)>
]>
```

RelaxNG-Schema für das Buch

```
buch    = element buch    { autor+, titel, verlag, jahr
                          , zusfassung? }
autor   = element autor   { element vorname { text }
                          , element nachname { text } }
titel   = element titel   { attribute language { text }?
                          , text }
verlag = element verlag { text }
jahr    = element jahr    { xsd:integer }
zusfassung = element zusfassung { (text
                                 | element keyword {text})*
```

XML-Schema für das Buch (Auszug)

```
<xs:element name="buch">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="autor"/>
      <xs:element ref="titel"/>
      <xs:element ref="verlag"/>
      <xs:element ref="jahr"/>
      <xs:element minOccurs="0" ref="zusfassung"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="autor">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="vorname"/>
      <xs:element ref="nachname"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Modellierung von XML in Haskell

- Ungetypt (für wohlgeformte Dokumente)
  - Generischer Dokumententyp
- Getypt (bezüglich eines Dokumententyps)
  - DTD/RNC $\longrightarrow$ Haskell-Typen

HaXml

- Getypte Einbettung in Haskell
  - Generischer XML-Parser
  - DTD $\longrightarrow$ algebraischen Datentyp
  - Klasse `XmlContent`
  - Funktionen

```
readXml :: XmlContent a => String -> Maybe a
showXml :: XmlContent a => a -> String
```

  - Jedes Element einen Typ, Instanz von `XmlContent`

HaXml: Übersetzte DTD

```
data Buch = Buch (List1 Autor) Titel Verlag Jahr
                 (Maybe Zusfassung)
data Autor = Autor Vorname Nachname
data Vorname = Vorname String
data Nachname = Nachname String
data Titel = Titel Titel_Attrs String
data Titel_Attrs = Titel_Attrs
                   { titelLanguage :: (Maybe String) }
data Zusfassung = Zusfassung [Zusfassung_]
data Zusfassung_ = Zusfassung_Str String
                 | Zusfassung_Keyword Keyword
data Keyword = Keyword String
```

Einschub: Labelled Records

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

```
data Book = Book { author :: String
                 , title  :: String
                 , publisher :: String }
```

- Konstruktion (Reihenfolge der Argumente irrelevant)

```
b = Book { author = "S. MacLane"
         , publisher = "Springer-Verlag"
         , title  = "Categories" }
```

Selektion, Update und Patternmatching

- Selektion durch Feldnamen:

```
publisher b --> "Springer-Verlag"
author b    --> "S. MacLane"
```

- Update:

```
b{publisher = "Rowohlt Verlag"}
```

  - Rein funktional! (`b` bleibt unverändert)

- Patternmatching:

```
print :: Book-> IO ()
print (Book{author= a, publisher= p, title= t}) =
  putStrLn (a++ " schrieb "++ t ++ " und "++
            p++ " veroeffentlichte es.")
```

Ein einfacher XML-Parser

- Einfacher Parser:

```
parseXMLDoc  :: String -> Maybe Element
```

- Generisches Content-Modell:
  - Element hat
    - Namen
    - Liste von Attributen,
    - mehrere Inhalte;
  - Inhalt kann sein
    - ein Element,
    - Text,
    - Entitätenreferenzen.

Die Datentypen: Content

```
data Content
= Elem Element
| Text CData
| CRef String
```

- CRef "ref" ist &ref;

Die Datentypen: Text

```
data CData = CData {
    cdVerbatim :: CDataKind
    cdData :: String
    cdLine :: Maybe Line
    }
data CDataKind = CDataText
               | CDataVerbatim
               | CDataRaw
```

- `CDataText` ist `#PCDATA` oder `text`
- `CDataVerbatim` ist `<![CDATA[...]]>`
- `CDataRaw` ist `<!...!>`

Die Datentypen: Element

```
data Element = Element {
  elName    :: QName
  elAttribs :: [Attr]
  elContent :: [Content]
  elLine    :: Maybe Line
  }
```

- Qname: qualifizierter Name

```
data Attr = Attr {
  attrKey :: QName
  attrVal :: String
  }
```

Qualifizierte Namen: `QName`

- Namensräume: Abgrenzung der Namen in verschiedenen Dokumententypen
- Namensraumdeklaration bindet Präfix an URI
- Beispiel:

```
<book xmlns:mybook="http://www.informatik.uni-bremen.de/~cx
  <autor><mybook:vorname>Saunders</mybook:vorname>
         <mybook:nachname>MacLane</mybook:nachname>
  </autor>
...
</book>
```

- In Haskell:

```
data QName = QName {     qName    :: String,
   qURI :: Maybe String, qPrefix :: Maybe String }
```

XML in Haskell: Beispiel

```
Prelude Text.XML.Light> parseXMLDoc
                "<book><title lang=\"deutsch\">Beispiel</title></book
Loading package xml-1.3.3 ... linking ... done.
Just (Element {elName = QName {qName = "book",
                               qURI = Nothing, qPrefix = Nothing},
               elAttribs = [], elContent =
 [Elem (Element {elName = QName {qName = "title",
                                 qURI = Nothing, qPrefix = Nothing}
                elAttribs =
    [Attr {attrKey = QName {qName = "lang",
                            qURI = Nothing, qPrefix = Nothing},
           attrVal = "deutsch"}],
                elContent =
    [Text (CData {cdVerbatim = CDataText,
                  cdData = "Beispiel",
                  cdLine = Just 1})],
                elLine = Just 1})],
               elLine = Just 1})
```

Beispiel: RSS-Feeds

- Daten aus RSS-Feeds lesen, bearbeiten, ausgeben
- Eingabeformat: RSS
- Augabeformat: XHTML
- Verarbeitung: Haskell
- Transportprotokoll: HTTP

HTTP

- Hypertext Transfer Transport
- Definiert in RFC 2616
- Vier grundlegende Befehle:
  - `GET` — Ressource lesen
  - `PUT` — Ressource erzeugen
  - `POST` — Ressource schreiben
  - `DELETE` — Ressource löschen
- Verführerisch einfach
- Kann beliebigen Inhalt übertragen: HTML, XML, ...

Very Simple HTTP

- Client (e.g. Web-Browser) sendet Anforderungen (Requests):

```
# telnet www.informatik.uni-bremen.de 80
Trying 134.102.224.17...
Connected to www.informatik.uni-bremen.de.
Escape character is '^]'.

GET http://www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws08/ 1.1
```

- Server sendet Antworten (Responses):

```
HTTP/1.1 200 OK
Date: Wed, 14 Jan 2009 07:52:52 GMT
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>[03-05-G-700.03] Praktische Informatik 3</title>
  <link REL="SHORTCUT ICON" HREF="http://www.informatik.
```

Die HTTP-Bücherei

- Einfaches `GET` mit

```
simpleHTTP :: Request -> IO (Result Response)
```

- Beispiel für Benutzung (siehe `Get.hs`)

```
readURI uri = do
  resp <- simpleHTTP (request uri)
  case resp of
    Right resp ->
      case rspCode resp of
        (2,0,0) -> do hPutStrLn stderr ("Successfully read
                   return (rspBody resp)
        _ -> error (httpError resp)
    Left connerr -> error (show connerr)
```

RSS

- RSS ist ein XML-Format für Schlagzeilen (Kurznachrichten)
- Verschieden Formate: RSS 0.9x, 1.0, 2.0, Atom
- Struktur (2.0):
  - Ein Feed enthält Liste von Channels
  - Ein Channel enthält Name, Link, Liste von Items
  - Ein Item enthält Titel, Link, Beschreibung
- Ein einfaches Beispiel

RSS in Haskell

```
data RSSChannel = RSSChannel { chTitle        :: String
                             , chLink         :: String
                             , chItems        :: [RSSItem]
                             } deriving Show

data RSSItem = RSSItem { itemTitle :: String
                       , itemLink  :: String
                       , itemDescr :: String
                       } deriving Show
```

RSS in Haskell — vereinfacht

Kanal hat nur Anzahl der Artikel

```
data RSSChannel = RSSChannel { chTitle        :: String
                             , chLink         :: String
                             , chItems        :: Int
                             } deriving Show
```

Übersetzung XML nach Haskell

- Kernfunktionalität: `String -> [RSSChannel]`
- Mit `parseXMLDoc` parsieren, gibt `Element`
- Top-Element:
  - Muss Name `RSS` haben
  - Wenn Attribut version, dann `2.0` oder `2.0.1`
  - Ansonsten Liste von Kanälen parsieren

Kanäle parsieren

```
getRSSChannels :: Element-> Either String [RSSChannel]
getRSSChannels (Element{elName= qn, elAttribs= as
                        , elContent= cont})
  | qne qn "rss" =
      case lookupAttr (qname "version") as of
        Just v | v == "2.0" || v == "2.0.1"
               -> Right (getChannels cont)
        Just v -> Left ("Wrong RSS version "++ v)
        Nothing -> Right (getChannels cont)
  | otherwise = Left "Not an RSS feed (no top RSS element)"

qne :: QName-> String-> Bool
qne qn str = qName qn == str
```

Einen Kanal übersetzen

- Kernfunktionalität: `Content-> Maybe RSSChannel`
- `Content` muss `Element` sein
- Pro `Element`:
  - Aus Elementen `title` und `link` Titel und Link extrahieren
  - Aus Element `item` Liste von Items extrahieren
- Wird dann erweitert zu

```
getChannels :: [Content]-> [RSSChannel]
```

Kanal parsieren

```
getChannel :: Content-> Maybe RSSChannel
getChannel (Elem e)
  | qne (elName e) "channel" =
        Just (foldr getChannelData mtChannel (elContent e))
getChannel _ = Nothing

mtChannel = RSSChannel "" "" 0


getChannelData :: Content -> RSSChannel-> RSSChannel
```

Kanal parsieren

```
getChannelData :: Content -> RSSChannel-> RSSChannel
getChannelData (Elem e) ch
  | qne (elName e) "title" = ch{chTitle = strContent e}
  | qne (elName e) "item"  = ch{chItems = 1+ chItems ch}
  | qne (elName e) "link"  = ch{chLink  = strContent e}
getChannelData _ ch = ch
```

Verarbeitung

- Aufgabe: Anzahl aller Artikel zählen
- Liste der Kanäle traversieren, `chItems` addieren

```
processChannels :: [RSSChannel]-> Int
processChannels chs = sum (map chItems chs)
```

Hilfsfunktionen

- Textinhalt aus mixed content extrahieren

```
strContent  :: Element-> String
```

- Element nach Name suchen

```
findElement :: QName -> Element -> Maybe Element
```

Darstellung

- Darstellung der Ergebnisse als `XHTML`
- Damit Nutzung der XML-Bücherei möglich
- Zwei Hilfsfunktionen
  - Einfaches Element (ohne Attribute) erzeugen:

    ```
    selem :: String-> [Content]-> Element
    ```

  - Einfachen Text-Content erzeugen:

    ```
    text :: String-> Content
    ```

HTML-Header erzeugen

```
htmlHeader :: String-> [Content]-> Element
htmlHeader tn cont =
  Element{ elName=qname "html"
         , elAttribs= [sattr "xmlns"
                             "http://www.w3.org/1999/xhtml"]
         , elContent= map Elem [title, body]
         , elLine= Nothing
         } where
    title = selem "head" [Elem (selem "title" [text tn])]
    body  = selem "body" cont
```

Eine ganz einfache HTML-Seite

```
showResult :: Int-> String
showResult n =
  ppTopElement (htmlHeader "Ergebnis" [
    text "Es wurden ",
    Elem (selem "b" [text (show n)]),
    text " Artikel gefunden." ])
```

Alles zusammensetzen

- Feeds aus Kommandozeile lesen (`getArgs`)

```
getArgs :: IO [String]
```

- Feeds lesen und verarbeiten

```
readURL :: String-> IO String
processFeed :: String-> IO [RSSChannel]
processChannels :: [RSSChannel]-> Int
```

- Ergebnis mit `showResult` anzeigen, ausgeben

```
showResult :: Int-> String
```

- Zusammengesetzt:

```
main = do
  feeds <- getArgs
  chs   <- mapM processFeed feeds
  putStrLn (showResult (processChannels (concat chs)))
```

Zusammenfassung

- XML modelliert Daten, Verarbeitung funktional
- XML und HTTP für Haskell:
  - `HTTP-3001.0.0`
  - `xml-1.3.3`
  - Müssen installiert werden
- Beispielprogramm zur Verarbeitung von RSS-Feeds