Seminar Funktionale Programmiertechniken

SS 2007

Thema:

Funktionale XML-Verarbeitung

von Björn-Peter Tietjens
(Matrikelnummer: 518042)

Betreuer: Prof. Dr. Michael Hanus

Inhaltsverzeichnis

1. Einleitung

In dieser Seminararbeit sollen die Möglichkeiten der XML Verarbeitung an Hand von funktionaler Programmierung erörtert werden. XML an sich ist ein sehr umfangreiches Gebiet und so ebenfalls das Thema der funktionalen Programmierung. Folgende Einschränkungen sollen in dieser Arbeit erfolgen:

- aus dem Bereich der „Extensible Markup Language“ (XML)[1] wird nur das für die Beispiele Nötigste betrachtet. Ein gewisses Grundverständnis wird vorausgesetzt.
- Ebenfalls wird ein Grundverständnis der funktionalen Programmierung[2] vorausgesetzt.

Nach einer Einführung in XML und funktionale Programmierung werden im Anschluss Bibliotheken betrachtet, die das Arbeiten mit XML in Haskell ermöglichen.
Es werden zwei Techniken der XML-Verarbeitung betrachtet. Am Beispiel der HaXML Bibliothek werden die Konzepte näher vorgestellt. Zunächst wird das Konzept der Repräsentation eines XML-Dokumentes in Form einer generischen Baumstruktur dargestellt.
Kerngedanke dieses Konzeptes ist die Abbildung und Bearbeitung aller gültigen XML Dokumente in einer generischen Struktur. Im Anschluss daran wird das Konzept des Databinding, welches eine Typ-Übersetzung darstellt, erklärt. XML-Strukturen werden in Datentypen der Programmiersprache gewandelt. Man arbeitet dann nicht mehr auf Elementen des XML- Dokumentes, sondern auf Konstrukten, die die Programmiersprache zur Verfügung stellt. Zum Abschluss werden dann einige Beispiele betrachtet und die Vor- und Nachteile der Verarbeitung von XML-Dokumenten nach den erwähnten Konzepten mit Hilfe von funktionalen Programmiersprachen erörtert.

1.1 Extensible Markup Language – XML

Die „erweiterbare Auszeichnungssprache“, kurz XML, ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien. XML ist eine vereinfachte Teilmenge von SGML. Die vom World Wide Web Consortium (W3C) herausgegebene XML-Spezifikation definiert eine Metasprache, auf deren Basis durch strukturelle und inhaltliche Einschränkungen anwendungsspezifische Sprachen definiert werden. Diese Einschränkungen werden durch Schemasprachen wie DTD oder XML-Schema ausgedrückt. Beispiele für XML-Sprachen sind: RSS, MathML, GraphML, XHTML, Scalable Vector Graphics, aber auch XML-Schema. Der logische Aufbau eines XML-Dokumentes ist eine Baumstruktur und damit hierarchisch strukturiert. Ein XML Dokument besteht aus sog. Entitäten:

1 Für weitere Informationen sei auf http://www.w3.org/XML/ verwiesen
2 Für weitere Informationen ist http://en.wikipedia.org/wiki/Functional_programming eine Einstiegsmarke

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adressbuch>
   <titel>Bjoerns Adressen</titel>
   <eintrag id="1">
      <name>
         <vorname>Sandy</vorname>
         <nachname>Schoenefeld</nachname>
      </name>
      <adresse typ="zuhause">
         <strasse>Nettelbeckstr</strasse>
         <hausnummer>13</hausnummer>
         <plz>24105</plz>
         <ort>Kiel</ort>
      </adresse>
   </eintrag>
</adressbuch>
```

Man unterscheidet Processing Instructions und Elemente. Die Elemente werden auch Knoten genannt und können in verschiedenen Ausprägungen, z. B. Tags, Attribute und Text auftreten.
Folgende Begriffe sind für den Inhalt dieser Arbeit von besondere Bedeutung:

- Wohlgeformtheit
  Ein XML-Dokument ist wohlgeformt, wenn es die Regeln für XML einhält. z. B.
  - Das Dokument besitzt genau ein Wurzelelement.
  - Alle Tags mit Inhalt besitzen ein Beginn- und ein Ende-Tag. Tags ohne Inhalt können auch mit /> abschließen.
  - Die Beginn- und Ende-Tags sind korrekt verschachtelt.
  - Ein Element darf nicht mehrere Attribute mit dem gleichen Namen besitzen, etc.
- Gültigkeit
  Soll XML für den Datenaustausch verwendet werden, ist es von Vorteil, wenn das Format mittels einer Grammatik (z. B. einer Dokument-Typ-Definition (DTD) oder eines XML-Schemas) definiert ist. Der Standard definiert ein XML-Dokument als gültig, wenn es wohlgeformt ist, den Verweis auf eine Grammatik enthält und das durch die Grammatik beschriebene Format einhält.

1.2 Funktionale Programmiersprachen

Eine funktionale Programmiersprache ist eine Sprache, die funktionale Programmierung unterstützt, indem sie Sprachelemente zur Kombination und Transformation von Funktionen anbietet. Rein funktional ist eine Sprache, die die Verwendung von Elementen ausschließt, die im Widerspruch zum funktionalen Programmierparadigma stehen.

Funktionale Programmierung ist ein Programmierparadigma. Programme bestehen hier ausschließlich aus einer Vielzahl von Funktionen. Das Hauptprogramm ist eine Funktion, welche die Eingabedaten als Argument erhält und die Ausgabedaten als seinen Wert zurückliefert. Diese Hauptfunktion verwendet ihrerseits wieder (Unter-)Funktionen für ihre Teilbereiche.

Ein Ansatz der funktionalen Programmierung ist, dass sich ihre Funktionen eher wie Funktionen aus der Mathematik verhalten. Dadurch kann man die Rechen- und Beweismethoden der Mathematik besser auf Programme anwenden, um vor allem ihre Korrektheit beweisen zu können.

Momentan konzentriert sich die Forschung auf dem Gebiet der funktionalen

Programmiersprachen auf statisch typisierte Sprachen, insbesondere auf solche, die das Typsystem nach Hindley und Milner verwenden. Der Vorteil dieses Typsystems ist die Verfügbarkeit von parametrischem Polymorphismus zusammen mit Typinferenz: Programmierer müssen die Typen ihrer Funktionen und anderer Werte nicht angeben, sondern bekommen sie gratis vom Übersetzer ausgerechnet, der zugleich noch während der Übersetzung Typfehler moniert.

In dieser Arbeit soll XML-Verarbeitung am Beispiel von Haskell betrachtet werden. Eigenschaften von Haskell sind z. B.

- Haskell ist stark typisiert. Es wird also streng zwischen Typen von Wahrheitswerten, Zeichen, ganzen Zahlen und Gleitkommazahlen u.s.w. Unterschieden.
- Haskell erlaubt Typvariablen. Damit können Funktionen sehr allgemein formuliert werden. Wird eine allgemein gehaltene Funktion für bestimmte Typen verwendet, werden automatisch die Typen abgeglichen (Typinferenz).
- Haskell ist statisch typisiert. Das bedeutet, dass die Typen bereits zum Zeitpunkt der Programmübersetzung feststehen, also Typfehler bereits hier erkannt werden.
- Haskell erlaubt benutzerdefinierte Datentypen. Diese algebraischen Datentypen werden mit Hilfe von Datenkonstruktoren definiert.

In funktionalen Programmiersprachen sind Datenstrukturen hierarchisch in Baumstrukturen aufgebaute Terme. In Haskell wird das Konzept der „lazy evaluation“ verfolgt. Haskell ist gut ausgestattet zur Verarbeitung von Baumstrukturen und ist erweiterbar.[1]

1.3 Bibliotheken

Zur Verarbeitung von XML-Dokumenten werden in allen gängigen Programmiersprachen Programmierkonzepte angeboten, die teilweise das eine oder das andere Konzept der Verarbeitung (siehe Kap. 2) umsetzen. Man kann sehr mächtige Bibliotheken für viele Programmiersprachen zur XML-Verarbeitung erhalten. Mit der zunehmenden Verwendung von Webservices wird das automatische Umsetzen von XML in Programmcode immer wichtiger. In dieser Arbeit soll ein Blick auf HaXML[3], HXML[4] und Haskell XML Toolbox[5] geworfen werden.

1.3.1 HaXML

HaXml wurde an der „University of York, Department of Computer Science“ entwickelt und wird dort auch weiter betreut. HaXML ist eine Sammlung von Werkzeugen zum Parsen, Filtern, Transformieren und Erstellen von XML-Dokumenten mit Haskell. Dazu gehören

- ein XML Parser (prüft Wohlgeformtheit)
- ein separater HTML-Parser (mit Fehlerkorrektur)
- eine Gültigkeitsüberprüfung für XML-Dokumente
- Textformatierer (pretty-printer) für XML- und HTML-Dokumente

3 http://www.cs.york.ac.uk/fp/HaXml/
4 http://www.flightlab.com/~joe/hxml/
5 http://www.fh-wedel.de/~si/HXmlToolbox/

Zum Verarbeiten von XML Dokumenten beinhaltet HaXML folgende Komponenten:

- „Combinators“ eine Kombinatoren-Bibliothek für generische XML-Verarbeitung, z. B. Transformation, Bearbeitung und Erstellung von XML-Dokumenten
- „Haskell2Xml“ ersetzt Haskell's Show/Read Klassen und wird benutzt um XML-Dokumente als einfache Haskell Daten zu lesen bzw. zu schreiben. „DrIFT“[6] kann diese Klassen automatisch ableiten.
- „DtdToHaskell“ ist ein Werkzeug zum Übersetzen von gültigen DTDs in äquivalente Haskell Typen.
- In Verbindung mit dem Xml2Haskell Framework, bietet DtdToHaskell also die Möglichkeit XML-Dokumente als normale getypte Werte in Haskell Programmen zu erstellen, zu bearbeiten und die Daten als XML-Dokumente einzulesen oder auszugeben.

1.3.2 HXML

HXML ist ein in Haskell geschriebener nicht validierender XML-Parser. Geschrieben wurde dieser Parser von Joe English bei Advanced Rotorcraft Technology, Inc. Er wurde entwickelt um Speicherplatzeffizienzvorteile aus dem Konzept der „lazy evaluation“ zu ziehen. HXML kann den umfangreichen HaXML Parser ersetzen, wenn ein speicherplatzeffizienter Parser innerhalb von Applikationen gebraucht wird. HXML verfolgt das Konzept der generischen Baumstruktur- Repräsentation von XML-Dokumenten.

1.3.3 Haskell XML Toolbox

Die „Haskell XML Toolbox“ ist eine Sammlung von Werkzeugen zum Bearbeiten von XML-Dokumenten mit Haskell. Sie ist selbst komplett in Haskell geschrieben. Die „Haskell XML Toolbox“ ist ein Projekt der Universität Wedel unter der Leitung von Uwe Schmidt.

Der Kern der „Haskell XML Toolbox“ ist ein validierender XML-Parser der XML 1.0 (Second Edition) voll unterstützt.

Die „Haskell XML Toolbox“ basiert auf den Ideen von HaXml und HXML, verfolgt aber eine generellere Herangehensweise für XML Verarbeitung mit Haskell. Die „Haskell XML Toolbox“ benutzt ein generisches Datenmodell für die Repräsentation von XML-Dokumenten. Dieses Datenmodell macht es möglich Filterfunktionen als ein allgemeines Design für XML-verarbeitende Applikationen zu benutzen. Eigenschaften :

- Unicode und UTF-8, US-ASCII und ISO-Latin-1 Unterstützung
- http: und file: Protokoll Unterstützung
- http Zugang auch über Proxy möglich
- Parsen von wohlgeformten Dokumenten und Überprüfung der Gültigkeit
- Namensraum Unterstützung, XPath Unterstützung, liberal HTML
- Relax NG Schema Validator, Integrierter XSLT Transformer

6 http://repetae.net/~john/computer/haskell/DrIFT/

2. Methoden der XML Verarbeitung

Es werden zwei wichtige Konzepte der Verarbeitung von XML Dokumenten betrachtet, die sich in Hinblick auf Generalität, Performance und Sicherheit zum Teil stark unterscheiden.

2.1 Generische Baumstruktur Repräsentation

Hierbei wird das Konzept verfolgt, möglichst mit einer Struktur alle wohlgeformten XML-Dokumente verarbeiten zu können, und nicht nur solche, die die gleiche DTD verwenden. Hierzu hat das W3C[7] Recommendations veröffentlicht, in denen XML quasi standardisiert wird.

2.1.1 Allgemeines

Das Document Object Model (DOM) ist eine solche Programmierschnittstelle (API) für den Zugriff auf XML-Dokumente. Sie wird vom World Wide Web Consortium definiert. Im Sinne der objektorientierten Programmierung besteht das DOM aus einem Satz von Klassen zusammen mit deren Methoden und Attributen. Es erlaubt Computerprogrammen dynamisch den Inhalt, die Struktur und das Layout eines Dokuments zu verändern.

Es wird also eine Baumstruktur im Arbeitsspeicher des Parsers aufgebaut, die das XML-Dokument darstellt. Programmatisch hat man die Möglichkeit, sich mit Hilfe der bereitgestellten Methoden in diesem Baum zu bewegen, Unterbäume und Inhalte von Knoten[8] hinzuzufügen, zu verändern und zu löschen, um dann das Modell wieder in einer Textdatei-Repräsentation in ein XML-Dokument zu schreiben. Es existiert keine Notwendigkeit für eine DTD, denn es wird keine tiefere Bedeutung der Tags unterstellt. In diesem Modell spielt nur die Wohlgeformtheit des XML-Dokuments eine Rolle. In der Regel werden Methoden zur Verarbeitung von XML bereitgestellt. D. h. statt die „<a>“ Syntax selber auszuprogrammieren, abstrahiert dieses Modell von der XML- Syntax und bietet Datenstrukturen gemäß der W3C Empfehlung.

Ein Beispiel für eine wirklich generische Anwendung in diesem Modell ist das Filtern von Inhalten eines XML-Dokumentes. Man stelle sich vor, man bekommt ein umfangreiches XML-Dokument, etwa einen Katalog mit vielen Produktinformationen und man möchte immer wieder nur eine entscheidende Information aus diesem Dokument extrahieren - etwa den Preis bestimmter Artikel in einer bestimmten Warengruppe: Dazu ist es hilfreich, dieses Konzept der XML-Verarbeitung zu wählen, denn es bietet einem die Möglichkeit, relativ unproblematisch genau die Information, die man extrahieren möchte, mit Hilfe von Filtern zu erhalten.

2.1.2 Generisches Baumstruktur Modell in HaXML

Ähnlich wie das DOM Modell welches z. B. in Java und Xercesc verwendet wird, bietet HaXML mit seinem Parser die Möglichkeit eine solche Repräsentation des XML-Dokumentes in Baumstruktur anzulegen. Dabei benutzt HaXML aber ausschließlich Haskell.

Alle Details darüber, was Wohlgeformtheit bzgl. eines XML-Dokumentes bedeutet, werden in Funktionen der Bibliothek HaXML versteckt. Der Benutzer kann sich voll auf die Erstellung des Inhaltes konzentrieren und kann sicher sein, dass das Ergebnis seiner

7 http://www.w3.org/TR/REC-xml/

8 Knoten sind auch Attribute, Kommentare und Plaintext – siehe: http://www.w3.org/DOM/

Arbeit ein dem Standard entsprechendes XML-Dokument ist, das auch von anderen Parsern akzeptiert werden wird.

In HaXML werden die Vorteile von Haskell für die Implementierung dieser Abstraktion genutzt. Es werden Funktionen definiert, die die Basisaufgaben übernehmen, sie werden hier „content filter“ genannt. Diese können dann mit sog. „combinators“ zu mächtigeren spezifischen Funktionen kombiniert werden.

Da Haskell besonders gut geeignet ist Baumstrukturen zu verarbeiten, ist es nahe liegend XML mit Haskell zu bearbeiten. Wir benötigen nur zwei Datentypen: „Element“ und „Content“. Während ein Element einen Namen, ggf. Attribute und dann einen Inhalt (Content) besitzt, und Content wiederum nur aus einem Element bzw. aus einem Text (String) besteht, lässt sich mit diesen zusammen ein mehrfach verzweigter Baum rekursiv erstellen. Genau das ist das grundlegende Konzept von HaXML:

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

Alle Basisfunktionen zur Bearbeitung von XML sind vom Typ „content filter“:

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

CFilter erwartet Inhalt, also ein Stück XML, und liefert dann eine (evtl. leere) Liste von Content. Alle Filter sind vom selben Typ, da man so bei deren Anwendung Teile des alten XML-Dokumentes in dem neuen XML-Dokument wieder verwenden kann. Filter können als Selektoren verstanden werden, um Teile des XML zu extrahieren. Filter können zur Erstellung von XML benutzt werden (Konstruktoren) und Filter können auch als Prädikate betrachtet werden, um zu entscheiden, ob ein Eingabe-Content Teil des Ausgabe-Content sein soll.

Hierzu kommt nun nur noch ein „Program Wrapper“:

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

Dies ist als eine „main“-Funktion zu verstehen, die Kommandozeilenargumente erwartet, die ein XML-Dokument (Eingabe) in den Content-Typ einliest und die den Filter dann auf das Wurzel-Element der Eingabe anwendet, um dann das Ergebnis des Filters in eine XML-Datei (Ausgabe) zurückzuschreiben.

Die in HaXML zur Verfügung gestellten Basisfilter lassen sich in Prädikatfilter, Selektorfilter und Konstruktorfilter[9] unterteilen. Hier nur einige Beispiele:

- Prädikatfilter:
  - `elm :: CFilter`
    gibt diese Entity zurück, wenn es ein Element ist.
  - `txt :: CFilter`
    gibt diese Entity zurück, wenn es ein Plaintext ist.
  - `tag :: String -> CFilter`
    gibt dieses Entity zurück, wenn sein Name dem übergebenen String entspricht.

- Selektorfilter:
  - `children :: CFilter`
    gibt die Kindelemente der Entity zurück, ist ggf. leer.

9 Eine komplette Liste aller Filter ist unter: http://www.cs.york.ac.uk/fp/HaXml/icfp99.ps.gz zu finden

  - `showAttr :: String -> CFilter`
    gibt den Wert des angegebenen Attributes als Content zurück (eigentlich Selektor und Konstruktor)
- Konstruktorfilter:
  - `literal :: String -> CFilter` erstellt plaintext content.
  - `mkElem :: String -> [Cfilter] -> CFilter`
    erstellt ein Element mit dem angegebenen String als Namen und wendet die Liste der Filter an, deren Ergebnisse Kindelemente werden.
  - `MkElemAttrs :: String -> [(String,CFilter)] -> [CFilter] -> CFilter`
    erstellt ein Element mit dem String als Namen und einer Attributliste aus der Liste der String, CFilter Tupel.

An dieser Stelle ist es Zeit das Beispiel wieder aufzugreifen:

```
module Main where

import Text.XML.HaXml
import Text.XML.HaXml.Wrappers
import Text.XML.HaXml.Combinators
import Text.XML.HaXml.Types
import Text.XML.HaXml.Pretty

main = processXmlWith adressbuch

adressbuch = mkElem "adressbuch" [ titel, eintrag ]

eintrag = mkElemAttr "eintrag" [ ( "typ", typ ) ] [ name, adresse ]

typ = literal "zuhause"

titel = mkElem "titel" [ literal "Bjoerns Adressbuch" ]

name = mkElem "name" [ vorname, nachname ]

vorname = mkElem "vorname" [ literal "Sandy" ]

nachname = mkElem "nachname" [ literal "Schoenefeld" ]

adresse = mkElem "adresse" [ strasse, hausnummer, plz, ort ]

strasse = mkElem "strasse" [ literal "Nettelbeckstr"]

hausnummer = mkElem "hausnummer" [ literal "13" ]

plz = mkElem "plz" [ literal "24105" ]

ort = mkElem "ort" [ literal "kiel" ]
```

mit den einfachen Funktionen „`mkElem`“ und „`mkElemAttrs`“ und dem Konstruktor für „`content`“ ist man bereits in der Lage ein XML-Dokument zusammenzubauen, ohne sich mit der XML Syntax auskennen zu müssen. Dieser Code erzeugt das oben gezeigte XML Beispiel.

2.1.3 Kombinatoren in HaXML

Wenn man allerdings hier „stehen bleiben“ würde, so könnte man auch gleich das XML-Dokument mit Zettel und Stift aufschreiben! Wichtig ist nun eine sinnvolle Erweiterung zu schaffen, mit der man problembezogene XML-Dokumente erstellen kann. Funktionale Programmiersprachen sind leicht erweiterbar und bieten somit die Möglichkeit eine Bibliothek zu erstellen, mit der man Filter kombinieren kann, um die Basisfilter zu

mächtigeren Konstrukten zu verbinden.

Ein wichtiger Combinator in HaXML ist 'o' die „Irish Composition“:

```
text `o` children `o` tag „title“
```

„der linke Filter wird auf das Ergebnis des rechten Filters angewendet“. Dies bedeutet hier: „nur die plaintext Kindelemente von dieser Entity, vorausgesetzt, es ist das 'title' Tag“. Weitere Kombinatoren[10] sind:

- `deep –` rekursive Suche bis ein Element gefunden wurde
  `deep f = f |>| (deep f 'o' children)`, wobei `|>| der` „directed choice operator“[11] ist
- `foldXml::CFilter->CFilter` - rekursive applikation
  `foldXml f = f 'o' (chip (foldXml f))`, wobei chip eine „inplace application“[12] auf die Kindelemente ist.

Auf diese Weise ist es schon fast intuitiv, komplexe Filter für spezifische Zwecke zusammenzusetzen. Man könnte nun Funktionen definieren, um applikationsspezifische Spracherweiterungen zu erhalten, wie zum Beispiel zum Erstellen von HTML aus XML:

```
module Main where
import Xml
main =  processXmlWith (albumf `o` deep (tag "album"))

albumf =
html
 [ hhead
 [ htitle
 [ txt `o` children `o` tag "artist"
 `o` children `o` tag "album"
 , literal ": "
 , keep /> tag "title" /> txt
 ]
 ]
 , hbody [("bgcolor",("white"!))]
 [ hcenter
 [ h1 [ keep /> tag "title" /> txt ] ]
 , h2 [ ("Notes"!) ]
      , hpara [ notesf `o` (keep /> tag "notes") ]
      , summaryf
      ]
   ]

notesf =
  foldXml (txt           ?> keep          :>
           tag "trackref" ?> replaceTag "EM" :>
           tag "albumref" ?> mkLink         :>
           children)

summaryf = ...
```

(Das komplette Beispiel ist im Anhang B zu finden und ist aus [2] entnommen)

Betrachtet man ein solches Beispiel, so kann man argumentieren so etwas gäbe es ja

10 Eine komplette Liste aller Kombinatoren ist unter: http://www.cs.york.ac.uk/fp/HaXml/icfp99.ps.gz zu finden

11 Directed choice: „`f |>| g`“ gibt entweder das Ergebnis von `f`, oder das von `g, aber nur genau dann wenn f unproduktiv ist.`

12 „`chip f`“ verarbeitet die Kindelemente der Entity „in-place“: der Filter `f` wird auf die Kindelemente angewendet;

schon in XPath zum Beispiel, allerdings muss man zur Benutzung dann auch XPath beherrschen. Ein offensichtlicher Vorteil hier ist: man bleibt in Haskell und damit in der bekannten Umgebung.

Funktionale Programmiersprachen wie Haskell ermöglichen die Definition von Funktionen höherer Ordnung, die Funktionen als Eingabe erwarten oder eine Funktion zurückgeben. Filter Kombinatoren sind Funktionen höherer Ordnung, zum Kombinieren mehrerer Filter. Da alle Filter den gleichen Typ haben ist es möglich, beliebige Filter miteinander zu kombinieren, denn Funktionen in Haskell haben keine Seiteneffekte. Die Benutzung von Kombinatoren ermöglicht es, die Details der „Programmierung mit Datenstrukturen“ zu verstecken. Alle Details der Datenmanipulation werden in Kombinatorfunktionen versteckt. Die Kombinatoren definieren problemspezifische Kontrollstrukturen. Durch diese Herangehensweise wird es ermöglicht, eine für das Problem natürliche Ausdruckweise zu finden. Kombinatoren sind den Unix pipes sehr ähnlich, denn diese erlauben es, durch die Kombination von sehr spezifischen Werkzeugen, komplexere Sequenzen aufzubauen. Äquivalent zu Unix pipes ist der "Irish composition" Kombinator, den wir oben bereits kennen gelernt haben. Das ähnelt einer pipe, welche die Ausgabe eines Programms als Eingabe an ein anderes Programm weitergibt. Die Kombinator- Bibliothek stellt alle Funktionen, die für die XML-Verarbeitung nötig sind, zur Verfügung.

2.1.4 HaXML als Abstraktion der XML-Syntax

Das Lesen und Schreiben von XML wird durch die HaXML Bibliothek also sehr stark erleichtert. HaXML bietet eine Abstraktion von der XML Syntax und man kann mit den Basisfiltern neue Filter generieren, um eigene Abstraktionen zu generieren. Im Gegensatz zum „selber schreiben“ ist das ein ganz wichtiges Ergebnis, denn durch die Benutzung dieses Modells in HaXML erreicht man automatisch wohlgeformtes XML-Output der Applikation.

2.1.5 Einschränkungen im Generischen Baumstruktur Modell

Was bis jetzt nicht erreicht wurde, ist eine Prüfung der Gültigkeit des In- bzw. Output-XMLs in Bezug auf eine DTD. Lediglich die Wohlgeformtheit wird durch den Parser sichergestellt.

So wäre es z. B. denkbar in dem Beispiel mehrere Postleitzahlen anzugeben. Das wäre sicher nicht besonders sinnvoll, doch momentan wird man nicht daran gehindert, so etwas in das XML zu schreiben. Eine Zielapplikation, die das Adressbuch verarbeiten soll, würde nicht wissen, was sie mit mehreren Postleitzahlen anfangen soll.

Ein größeres Risiko wäre es aber, die Anzahl von Tags nicht einzuschränken. So erwartet man, dass jemand vermutlich mehrere Telefonnummern hat. Man muss also im Beispiel oben zulassen, dass in dem XML-Dokument auch mehrfache Tags z. B. für Telefonnummern zulässig sind, wobei man im Vornherein nicht genau weiß, wie viele benötigt werden. Man sollte eine maximale Tag Anzahl im DTD definieren. Sonst kann es dazu führen, das ein potentieller Angreifer ein XML-Dokument zur Verfügung stellt, das eine sehr sehr große Zahl an z. B. Telefonnummern-Tags enthält. Der Parser würde dieses akzeptieren, wenn es wohlgeformt ist und vermutlich irgendwann keinen Speicher mehr für seine Baumstruktur zur Verfügung haben. Dieses Problem wird hier noch nicht weiter berücksichtigt, kann aber mit Hilfe von XML-Dokumenten Strukturbeschreibungen, wie DTD oder XML-Schema, vermieden werden.

---

die Ergebnisse werden wieder zusammengebaut zu Kindlementen der gleichen Entity.

Ein anderes Szenario wäre die Typangabe im Adress-Tag: eine solche Angabe ist nur dann sinnvoll, wenn man weiß, was sie bedeutet. Hier wäre also eine Aufzählung der gültigen Werte erforderlich. Momentan könnte man aber noch beliebige Typen angeben.

Weiter kann man sich eine Einschränkung der Postleitzahl auf fünf Zahlzeichen (für Deutschland), ggf. einen Integer-Typ für das ID-Attribut oder gar eine Beschreibung der zulässigen Werte durch reguläre Ausdrücke vorstellen.

Um diese Einschränkungen, die ein DTD dem XML-Dokument hinzufügt auch im HaXML-Framework umzusetzen, beshäftigt sich diese Arbeit nun im folgenden Kapitel mit dem Databinding[13] unter Haskell.

2.2 Databinding

2.2.1 Document Type Definitions

DTDs haben die Aufgabe, die Struktur eines XML-Dokumentes festzulegen. Man sagt auch es stellt eine Grammatik für eine Klasse von XML-Dokumenten dar. Es wird die Reihenfolge, Verschachtelung und Art des Inhalts von Elementen des XML festgelegt. Man möchte z. B. versuchen zu definieren, welche Elemente im XML-Dokument und welche Attribute in einem Element vorkommen dürfen etc. Würde so ein DTD für das o. g. Adressbuch existieren, könnte das in etwa so aussehen:

```
<!DOCTYPE adressbuch [
<!ELEMENT adressbuch (eintrag*)>
<!ELEMENT eintrag (name,adresse)>
<!ATTLIST eintrag id CDATA #REQUIRED>
<!ELEMENT adresse (strasse, hausnr, plz, ort)>
<!ATTLIST adresse typ (zuhause | buero) #REQUIRED>
<!ELEMENT name (vorname, nachname)>
<!ELEMENT vorname  (#PCDATA)>
<!ELEMENT nachname  (#PCDATA)>
<!ATTLIST nachname titel (Dipl-Inf | Dr | Prof) #IMPLIED>
<!ELEMENT strasse  (#PCDATA)>
<!ELEMENT hausnr  (#PCDATA)>
<!ELEMENT plz  (#PCDATA)>
<!ELEMENT ort  (#PCDATA)>
]>
```

Man schreibt also vor, wie ein Eintrag im Adressbuch auszusehen hat und vermeidet damit Probleme, die mit ungültigen aber dennoch wohlgeformten XML-Dokumenten auftreten könnten (siehe oben). Dazu muss allerdings der Compiler das DTD interpretieren und auf das XML-Dokument beim Parsen „anwenden“.

Im folgenden Abschnitt wird beschrieben, wie man noch einen Schritt weitergehen kann. Anstatt das man eine DTD einfach nur an den Parser weiterreicht und dieser die DTD auf das XML-Dokument „anwendet“, könnte man auch die Dokumentenbeschreibung (DTD) dazu benutzen, (Benutzer-) Datentypen der Programmiersprache zu generieren. Die erstellten Datentypen können dann an die Daten des zu lesenden XML-Dokuments gebunden werden. Diese Idee kennt man zum Beispiel auch aus JAXB[14], wo aus einer DTD/XML-Schema Java Klassen kompiliert werden, welche dann beim Einlesen eines XML-Dokumentes zu Objekten instantiiert werden und so das XML-Dokument repräsentieren.

13 Siehe http://en.wikipedia.org/wiki/XML_data_binding für Näheres zu Databinding
14 http://java.sun.com/developer/technicalArticles/WebServices/jaxb/ Näheres zu JAXB

Man hat also eine Datenstruktur, die genau das eine gewünschte XML-Dokument abbildet. Zusätzlich verwendet man einen Mechanismus, der die Daten aus dem XML-Dokument direkt an diese Datenstruktur bindet (Databinding). Ein offensichtlicher Vorteil ist, das man alle Funktionalitäten eines Datentyps und damit der Programmiersprache direkt verwenden kann und nicht auf ein XML-Framework, das Funktionen zum Verarbeiten des XML bereitstellt, angewiesen ist. Schlägt nun dieses Databinding fehl, so war das XML nicht gültig, aber evtl wohlgeformt und man hat einen Ansatzpunkt für eine Fehlerbehandlung. Funktioniert das Databinding, so bekommt man das Prüfen der Gültigkeit geschenkt und kann sogar aus den gewonnenen Datentypen nach einer beliebigen Verarbeitung wieder ein wohlgeformtes und bzgl. der DTD gültiges XML-Dokument generieren, z. B. durch Serialisieren in eine Textdatei.

Der Prozess des Konvertierens eines XML-Dokumentes in eine Datenstruktur wird „unmarshalling“ genannt. Der umgekehrte Prozess, eine Datenstruktur in ein XML-Dokument zu serialisieren, heißt „marshalling“.

2.2.2 Typübersetzung von DTDs in Haskell Datentypen

Der Begriff der „gültigen Dokumenten Verarbeitung“ („valid document processing“) wurde von Malcom Wallace in seinem Papier [2] benutzt. Er beschreibt ein gültiges Verarbeitungsskript als eines, das eine gültige XML-Ausgabe generiert, wenn es gültiges XML als Eingabe erhält. Mit der streng getypten Sprache Haskell wäre dies denkbar, wenn man eine DTD in Haskell-Typen übersetzt.

Angenommen man erstellt sich ein solches Skript, das Datentypen verwendet, die aus einer DTD erstellt worden sind. Erhält dieses Skript dann ein gültiges XML und bindet dies an die generierten Typen, so kann hier kein Typfehler auftreten. Haskell an sich hat mit seiner Typprüfung zur Kompilierzeit ebenfalls keine Typfehler zur Laufzeit zu erwarten. Die Ausgabe ist wiederum über die aus dem DTD definierten Typen gesteuert und kann somit wieder nur gültiges XML generieren.

Es ist also erst einmal denkbar, mit Haskell und einer entsprechenden typbasierten Übersetzung ein Programm im Sinne der „gültigen Dokumenten Verarbeitung“ zu erstellen.

Schaut man sich die DTD an, so kann man eine Verknüpfung zwischen der sehr eingeschränkten Typbeschreibung in DTDs und dem umfangreichen Typsystem von Haskell herstellen. Grob gesagt heisst das, man kann jede `<!ELEMENT...>` Beschreibung in einen Haskell Datentyp übersetzen.

Sequenzen sind wie Produkttypen[15], Auswahllisten sind wie Summentypen, optinale Elemente sind „Maybe Typen“ und Wiederholungen sind Listen in Haskell. Für Attributlisten können Haskells„Named-Fields“ eingesetzt werden. Für die Attributwerte, die auf eine endliche Menge beschränkt sind, können Aufzählungstypen in Haskell eingesetzt werden. Hier noch mal ein Blick darauf im Einzelnen:

- Ein Element, welches eine Sequenz von Elementen beinhaltet, wird in einen Produkttyp übersetzt. Der Datenkonstruktor wird um die Typvariablen aller Kind-Elemente erweitert.

```
<!ELEMENT  farbe  (rot, blau)>
```

15 Siehe Appendix A zur Beschreibung der Haskelltypen

```
<!ELEMENT  rot    EMPTY>
<!ELEMENT  blau   EMPTY>

data Farbe = Farbe Rot Blau
data Rot   = Rot
data Blau  = Blau
```

- Ein Element, welches eine Alternative von Elementen beinhaltet, wird in einen Aufzählungstyp (Spezialisierung vom Summentyp siehe Anhang A) übersetzt. Der Name des Datenkonstruktors wird aus dem Element-Namen und dem Kind-Element-Namen zusammengesetzt.

```
<!ELEMENT  farbe  (rot | blau)>
<!ELEMENT  rot    EMPTY>
<!ELEMENT  blau   EMPTY>

data Farbe = FarbeRot Rot
           | FarbeBlau Blau
data Rot  = Rot
data Blau = Blau
```

- Ein Element, welches optional ein weiteres Element beinhaltet, wird in einen Produkttyp übersetzt, dessen Typvariable vom Datentyp Maybe Elementname ist.

```
<!ELEMENT  farbe  rot?>
<!ELEMENT  rot    EMPTY>

newtype Farbe = Farbe (Maybe Rot)
data Rot = Rot
```

- Ein Element, welches ein Kind-Element kein Mal oder beliebig oft (*) enthält, wird in einen Datentyp übersetzt, welcher eine Liste des Kind-Elements aufnimmt.

```
<!ELEMENT farbe rot*>
<!ELEMENT  rot EMPTY>

newtype Farbe = Farbe [Rot]
data Rot = Rot
```

  Ein Element, welches ein anderes Element mindestens ein Mal oder beliebig oft (+) enthält, wird ebenfalls in einen Datentyp übersetzt, welcher eine Liste des Kind-Elements aufnimmt. Da der Haskell-Datentyp Liste nicht vorschreibt, dass er mindestens ein Element enthalten muss, besteht an dieser Stelle eine Diskrepanz zwischen der DTD und dem Haskell-Typsystem.

- Enthält ein Element Textdaten, wird für diese der Haskell-Datentyp String verwendet.

```
<!ELEMENT  farbe  (#PCDATA)>

newtype Farbe = Farbe String
```

- Enthält ein Element beliebigen deklarierten Inhalt, kann HaXml dieses nicht in das Typsystem von Haskell überführen.

```
<!ELEMENT  farbe  ANY>

Program error: NYI
```

Es lässt sich so ein Regelwerk erstellen, mit dem ein großer Teil der DTD-Definitionen in

Haskell übersetzt werden kann. Eine formalisierte DTD-Übersetzungsregelliste findet man in [2]. Eine Implementierung dieser Regeln ist dann in der Lage, die obige Beispiel-DTD in gültigen Haskell Code zu übersetzen.

Die Werkzeuge „DtdToHaskell“ und „Haskell2Xml“ aus dem HaXML-Framework sind Implementierungen, die nicht nur ein DTD in Haskell, sondern auch Haskell Code wieder in gültiges XML übersetzen können. Das obige Beispiel DTD in Haskell übersetzt:

```
newtype Adressbuch = Adressbuch [Eintrag]                    deriving (Eq,Show)
data Eintrag = Eintrag Eintrag_Attrs Name Adresse            deriving (Eq,Show)
data Eintrag_Attrs = Eintrag_Attrs { eintragId :: String }   deriving (Eq,Show)
data Adresse = Adresse Adresse_Attrs Strasse Hausnr Plz Ort deriving (Eq,Show)
data Adresse_Attrs = Adresse_Attrs {adresseTyp::Adresse_typ}deriving (Eq,Show)
data Adresse_typ = Adresse_typ_zuhause
                 | Adresse_typ_buero           deriving (Eq,Show)
data Name = Name Vorname Nachname              deriving (Eq,Show)
newtype Vorname = Vorname String               deriving (Eq,Show)
data Nachname = Nachname Nachname_Attrs String deriving (Eq,Show)
data Nachname_Attrs = Nachname_Attrs
    { nachnameTitel :: (Maybe Nachname_titel)
    } deriving (Eq,Show)
data Nachname_titel = Nachname_titel_Dipl_Inf
                    | Nachname_titel_Dr
                    | Nachname_titel_Prof    deriving (Eq,Show)
newtype Strasse = Strasse String             deriving (Eq,Show)
newtype Hausnr  = Hausnr String              deriving (Eq,Show)
newtype Plz     = Plz String                 deriving (Eq,Show)
newtype Ort     = Ort String                 deriving (Eq,Show)
```

(weitere Beispiele sind in Anhang C zu finden)

Zusätzlich zur Generierung der Typen werden noch Funktionen zum Schreiben und Lesen von XML benötigt (siehe Beispiele in Anhang C). Damit kann ein XML-Element mit seinen Daten an den neuen Haskell Typ gebunden werden. In HaXML wurde dies mit Hilfe einer Implementierung Namens: „DrIFT“ realisiert. Es werden hier die benötigten Funktionen aus den Typdeklarationen generiert. Dazu wird eine Sammlung von vordefinierten Typklassen benutzt, von denen dann mit dem Werkzeug für jeden neuen Typ eine Instanz abgeleitet wird.

2.2.3 Einschränkungen der Typübersetzung

Sicherlich ist HaXML gut einsetzbar für allgemeine Anwendungen, die keine besonderen Anforderungen an die DTD aufweisen. Es muss aber erwähnt werden, dass immer ein neuer Typ (datatypes oder newtypes) für jedes XML-Element von der Implementierung der Übersetzungsregeln generiert werden muss. Typsynonyme sind momentan nicht denkbar. Dies ist der Fall, da Typsynonyme in Haskell nicht von ihren Herkunftstypen zu unterscheiden sind. Es ist notwendig, die Typen unterscheiden zu können, um in den XML-Ausgabe-Funktionen die schließenden Tags richtig zu setzen und damit die Wohlgeformtheit der XML-Ausgabe zu gewährleisten.

Hinzu kommt noch das für jede Attributliste ein neuer Typ eingeführt werden muss, da ein Element aus einem Paar von Attributen und Inhalt besteht.

Ein viel größeres Problem tritt bei der Namensgebung auf. In Haskell müssen Typnamen mit einem Großbuchstaben und Feldnamen mit Kleinbuchstaben beginnen. XML ist case-sensitive[16] und kann damit zwischen Tag-/Attributnamen wie „adresse“ und „Adresse“ unterscheiden. Würde man nun in einem XML-Dokument sowohl <Adresse> als auch <adresse> Elemente verwenden, so kann der Haskell Typ nicht einfach in beiden Fällen mit `data Adresse = ...` benannt werden! Hier und an einigen anderen Stellen können Namenskonflikte entstehen, die nur teilweise behandelt werden. Ein weiteres Problem werden dann auch die möglichen Namensräume in XML sein, denn dazu gibt es in Haskell kein direktes Äquivalent. Eine mögliche Lösung könnte das Konkatenieren von Namensraumbezeichnung und Tagname sein, damit der Typname in Haskell eindeutig bleibt.

Weiter ist die Benutzung von „named-fields“ auf einen einzelnen Typ beschränkt. Man kann also ein „named-field“, das evtl. in mehreren Typen genau gleich vorkommt nicht in allen benutzen, sondern man muss für jeden Typ ein neues Feld anlegen. Alternativen wie z. B. getypte erweiterbare Rekords, sind in [3] erörtert.

Obwohl es schwierig erscheint, die Hürden der Typbenennung bei der Übersetzung von DTDs in Haskell Datentypen zu meistern, gibt das Haskelltypsystem wesentlich mehr an Handhabungen für Typen her als die DTD.

Denkt man hier an die Möglichkeit in Haskell polymorphe Typen zu verwenden, so wird man ohne nicht standardisierte, nicht XML-konforme „work-arounds“ in der DTD nichts dergleichen finden. Weitergehende DTDs wie zum Beispiel XML-Schema-Language sollen ähnliche Konzepte leisten können[17].

3. Gegenüberstellung

3.1 Vor- und Nachteile der generischen Baumstruktur

Die Herangehensweise in Haskell hat gegenüber der bisher bekannten XML Verarbeitung in anderen Sprachen gewisse Vorteile.

Oft sind Skriptsprachen, die zur Verarbeitung von XML verwendet werden, nicht sinnvoll oder nur mit großen Schwierigkeiten zu erweitern. Spracherweiterungen sind oft unübersichtlich und bedürfen weiterführender Kenntnisse. Das führt im Allgemeinen dann auch zu erhöhtem Wartungsaufwand. Hingegen ist in Haskell und auch in anderen funktionalen Programmiersprachen, wie man an den Kombinatorbibliotheken gesehen hat, eine Erweiterung der Sprache selbst vorgesehen. Man kann mit einfachen Sprachkonstrukten, z. B. Funktionen höherer Ordnung, die Verarbeitung von XML beliebig verändern, kombinieren und erweitern. Problemspezifische Lösungen lassen sich so leicht einbauen ohne erst neue Bibliotheken oder eine weitere Sprache lernen zu müssen. Hinzu kommt, dass sich die Funktionen in Haskell in fast beliebig kleine Teilfunktionen verteilen lassen und so eine bessere Übersichtlichkeit durch größere Modularisierung gegeben ist.

Häufig werden in der XML Verarbeitung Skriptsprachen verwendet, die leicht zu erlernen

16 „case-sensitive“ ist nichts Anderes, als die Erkennung von großer bzw. kleiner Schreibweise von Buchstaben.
17 Siehe http://w3.org/TR/xmlschema-1 und http://w3.org/TR/xmlschema-2

sind. Deswegen fehlt ihnen oft eine generellere Einsatzmöglichkeit. Um etwa Netzwerkverbindungen oder Verbindungen zu Datenbanken herzustellen, müssen oft erst zusätzliche Erweiterungen oder gar andere Programmiersprachen bemüht werden. Wird ein XML-Dokument jedoch anhand von den oben vorgeschlagenen Methoden mit Kombinatoren in ein Haskell Programm umgesetzt, so hat man automatisch alle Möglichkeiten der mächtigeren Programmiersprache Haskell zur Verfügung.

Da in der funktionalen Programmierung in der Regel keine Nebeneffekte auftreten, ist es unproblematisch, Funktionen mit bestimmten Aufgaben wieder zu verwenden. Es ist möglich, Funktionen aus älteren Applikationen, ggf. in einem neuen Kontext, ohne Veränderung sondern nur durch Kombination mit weiteren Funktionen, wiederzuverwenden.

Bisher war es immer sehr schwierig zu entscheiden, ob ein Programm wirklich das tut, was es soll. Programmverifikation in Skriptsprachen ist noch komplizierter, da oft nur eine ungenaue Beschreibung darüber existiert, was in welchen Fällen wirklich geschieht. Bei großen Sprachen ist eine solche Beschreibung oft so umfangreich, dass es sehr mühsam ist, diese durchzuarbeiten. Da funktionale Programmiersprachen sich sehr eng an der mathematischen Definition von Funktionen gehalten haben, ist es hier wesentlich einfacher, den Überblick zu behalten und Beweise darüber zu führen, warum ein Programm korrekt ist.

Am Ende spielt es noch eine wichtige Rolle zu berücksichtigen, dass viele Skriptsprachen keinen Kompiler oder Typchecker mitbringen. Es muss immer durch Testen zur Laufzeit „probiert“ werden, ob evtl. noch Fehler in der Verarbeitung vorkommen. All diese Dinge bekommt man bei der Verwendung von Haskell gratis dazu.

Das alles nutzt einem natürlich nur etwas, wenn man in Haskell programmieren kann.

3.2 Vor- und Nachteile der typbasierten Übersetzung

Die volle Typübersetzung ist sicherlich aufwendiger, leistet aber, wie wir gesehen haben, viele Dinge, die die Kombinatorenbibliotheken nicht leisten.

Wichtig ist, dass ein korrektes Programm, welches typbasierte Übersetzung benutzt, auch nur gültiges XML generieren kann (siehe oben: „valid document processing“).

In der generischen Variante kann es unter Umständen nötig sein, ein und das selbe Element irgendwo im XML-Dokument mehrfach heraus zu suchen. Dagegen hat man bei einer Typübersetzung immer direkten Zugriff auf alle Elemente.

Ein Nachteil kann sein, das tritt auch in anderen Frameworks auf, dass man Zeit investieren muss, um die Voraussetzungen für eine Typübesetzung zu schaffen. Zusätzlich müssen die DTDs, die verwendet werden sollen, auch gut durchdacht sein, damit Probleme, wie sie bereits erörtert wurden auch wirklich durch die Typübersetzung vermieden werden können.

Die Mächtigkeit des Haskell Typsystems ist noch lange nicht ausgeschöpft. Es wird die DTD-Language zunehmend von XML-Schema-Language abgelöst. Es bleibt zu untersuchen, in wie weit zum Beispiel polymorphe Typen in Haskell mit den neuen Möglichkeiten von XML-Schema kombiniert und ausgenutzt werden können.

3.3 Fazit

Es macht den Eindruck, dass Haskell und funktionale Sprachen im Allgemeinen bei der

Verarbeitung von XML-Dokumenten sehr nützlich sein können. Für generische Applikationen kann eine Lösung mit Kombinatoren ein gutes Stück weiterhelfen, ohne das man auf etwaige nicht auf XML bezogene Funktionalitäten verzichten muss.

Wenn es um speziellere Applikationen auf einer definierten Menge an DTDs geht, so kann das Typübersetzungskonzept dazu verhelfen, nicht mehr in XML denken und handeln zu müssen. Dem Haskell-Programmierer werden Datentypen an die Hand gegeben, mit denen er in einer uniformen Umgebung all das erledigen kann, was er für seine Applikation benötigt.

3.4 Ausblick

In dem Papier von Malcolm Wallace werden noch weiterführende Ideen zu den hier behandelten Konzepten betrachtet. Dazu gehören Vorschläge, wie die Kombinatoren weiter verfeinert werden können. Man stellt sich zum Beispiel vor:

- breitere Funktionalitäten einzubauen,
- ggf. mehrfach Ein- und Ausgaben möglich zu machen und
- noch bessere Generizität einzubauen.

Ebenso könnte die Effizienz der Kombinatoren auch ein interessantes Thema sein, mit dem man sich in Richtung von Platz- und Zeitbedarfsminimierung oder einer Erweiterung bzgl. DTD-Einbindung beschäftigen könnte.

3.5 Vergleich mit JAXB

Weil Webservices, die ihre Informationen in Form von XML-Datenströmen austauschen, populärer werden, sind automatisierte Verfahren, um XML bzgl. DTD/XML-Schema zu überprüfen und die Daten möglichst effizient zu verarbeiten, immer wichtiger.

Ein Vergleich der Haskell Typübersetzung realisiert in HaXML zwingt einem den Vergleich mit anderen Typübersetzern, wie zum Beispiel JAXB, auf. Ein offensichtlicher Unterschied liegt in der Natur der Sache: mit Java als imperativer und Haskell als funktionaler Programmiersprache.

In einigen IDEs sind sogenannte Webresourcen-Kompiler sogar schon eingebaut. Man braucht nur noch die Webresource (z. B. die URL des Webservices) anzugeben, aus der die IDE automatisch Klassen in der gewünschten Sprache erstellt, mit denen dann die Applikation den Webservice ansprechen kann. Der Programmierer bekommt hier also immer für seine Sprache „eigene“ Datenstrukturen aus einer XML-Quelle geliefert und kann diese dann nach Belieben verarbeiten.

Dies geschieht auch in JAXB, wo aus einem XML-Schema-Dokument Java Klassen generiert werden und mit Hilfe der marschalling und unmarschalling Prozesse, diese Klassen dann mit den Daten aus einem XML-Dokument zu Objekten instantiiert werden.

Wie schon erwähnt, hinkt ein Vergleich wegen der unterschiedlichen Programmierparadigmen etwas, aber das Konzept des oben genannten „valid document processing“ wird in HaXML deutlich energischer verfolgt. Das liegt auch daran, dass eine Typprüfung zur Kompilierzeit, die in Haskell schon realisiert ist, in Java wesentlich aufwendiger ist. „Valid Processing“ ist hier als wichtiger Unterschied zwischen JAXB und HaXML zu sehen. HaXML bietet somit einen nicht zu unterschätzenden Vorteil.

Außerdem ist die Übersichtlichkeit von Haskell gegenüber Java Code von Vorteil. Bei der

automatischen Erzeugung von Code spielt es eine wichtige Rolle, wie lange der Benutzer des Codes am Ende braucht, um diesen zu durchschauen und korrekt zu benutzen.

4. Beispiel

Zusätzlich zu den oben genannten kleinen übersichtlichen Beispielen, soll an dieser Stelle noch ein Beispiel erwähnt werden, das aus dem „richtigen Leben“ kommt. Es soll zeigen, dass es durchaus sinnvolle Anwendungen für XML und Haskell geben kann.

Grundlage für dieses Beispiel ist Glade[18], ein Werkzeug mit dem sich GTK Benutzerschnittstellen „zusammen-klicken“ lassen. Entscheidend ist hier, dass Glade alle Informationen in einem XML-Dokument abspeichert. Die gesamte GUI-Beschreibung steckt also in dem produzierten XML-Dokument.

Die „Gtk2Hs“ Bibliothek kann nun dieses XML einlesen und in eine gültige grafische Haskell- Benutzerschnittstelle umwandeln.

Hier ein Ausschnitt aus solch einem Dokument:

```
<?xml version="1.0" standalone="no"?> <!--*- mode: xml -*-->

<!DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-2.0.dtd">
<glade-interface>
<widget class="GtkWindow" id="mainwin">
  <property name="width_request">800</property>
  <property name="height_request">600</property>
  <property name="visible">True</property>
  <property name="title" translatable="yes">Bomberman.net</property>
  <property name="type">GTK_WINDOW_TOPLEVEL</property>
  <property name="window_position">GTK_WIN_POS_CENTER</property>
  <property name="modal">False</property>
  <property name="default_width">800</property>
  <property name="default_height">600</property>

  <child>
    <widget class="GtkVBox" id="vbox1">
      <property name="visible">True</property>
      <property name="homogeneous">False</property>
      <property name="spacing">0</property>

      <child>
      <widget class="GtkMenuBar" id="menubar1">
        <property name="border_width">1</property>
        <property name="visible">True</property>
        <property name="pack_direction">GTK_PACK_DIRECTION_LTR</property>
        <property name="child_pack_direction">GTK_PACK_DIRECTION_LTR</property>
        <child>
          <widget class="GtkMenuItem" id="menuitem1">
            <property name="visible">True</property>
            <property name="label" translatable="yes">_File</property>
            <property name="use_underline">True</property>
            <child>
            <widget class="GtkMenu" id="menu1">
              <child>
                <widget class="GtkImageMenuItem" id="new1"> ...
```

Mit den Funktionen der *Graphics.UI.Gtk.Glade*-Bibliothek lässt sich nun sehr einfach diese

18 Für weitere Informationen siehe: http://developer.gnome.org/tools/glade.html

XML-Beschreibung in ein Haskell Programm einlesen:

```
module BNet.Gui.Gui where

import Graphics.UI.Gtk
import Graphics.UI.Gtk.Glade

main = do
  -- initialisieren der gui engine
  initGUI
  -- glade xml laden
  lohaXmlM <- xmlNew "BNet/Gui/bomberman.net.glade"
  let lohaXml = case lohaXmlM of
                    (Just lohaXml) -> lohaXml
                    Nothing -> error "file not found"
  -- window handler besorgen
  window <- xmlGetWidget lohaXml castToWindow "mainwin"
```

...

Es wird mit „xmlNew“ das XML-Dokument geladen. Ein Ausschnitt:

```
-- | Create a new XML object (and the corresponding widgets) from the given
-- XML file.
-- This corresponds to 'xmlNewWithRootAndDomain', but without the ability
-- to specify a root widget or translation domain.
--
xmlNew :: FilePath -> IO (Maybe GladeXML)
xmlNew file =
  withCString file               $ \strPtr1 -> do
  xmlPtr <- {#call unsafe xml_new#} strPtr1 nullPtr nullPtr
  if xmlPtr==nullPtr then return Nothing
              else liftM Just $ constructNewGObject mkGladeXML (return xmlPtr)
```

Die zentrale Funktion zum erstellen der Widgets heißt „xmlGetWidget“, mit der alle „handles“ auf die GUI-Elemenete geladen werden können. Ein Ausschnitt:

```
-- | Get the widget that has the given name in
-- the interface description. If the named widget cannot be found
-- or is of the wrong type the result is an error.
--
xmlGetWidget :: (WidgetClass widget) =>
    GladeXML
 -> (GObject -> widget) -- ^ a dynamic cast function that returns the type of
                        -- object that you expect, eg castToButton
 -> String              -- ^ the second parameter is the ID of the widget in
                        -- the glade xml file, eg \"button1\".
 -> IO widget
xmlGetWidget xml cast name = do
  maybeWidget <- xmlGetWidgetRaw xml name
  case maybeWidget of
    Just widget -> evaluate (cast (toGObject widget))
        --the cast will return an error if the object is of the wrong type
    Nothing -> fail $ "glade.xmlGetWidget: no object named " ++ show name ++ "
in the glade file"
```

Das in Graphics.UI.Gtk.Glade verfolgte Konzept, ist das der Kombinatoren. Die Bibliothek besteht also hauptsächlich aus Kombinatoren, um aus der XML-Beschreibung der Oberfläche ein „Widget-Tree“ aufzubauen.

Literaturverzeichnis

- 1: Paul Hudak, Conception, evolution, and application of functional programming languages, September 1989, ACM Press New York, NY, USA, , ,ACM Computing Surveys (CSUR), Volume 21, Issue 3, Pages: 359 - 411, ISSN:0360-0300

- 2: Malcolm Wallace, Colin Runciman, Haskell and XML: Generic Combinators or Type-Based Translation, University of York, UK, Proceedings of the International Conference of Functional Programming ICFP, Paris, 1999, http://www.cs.york.ac.uk/fp/HaXml/icfp99.ps.gz

- 3: Benedict R Gaster, Record, Variants and Qualified Types, 1998, Dept. of Computer Science, University of Nottingham, http://www.cs.nott.ac.uk/Research/fop/gaster-thesis.pdf, PhD Thesis,Technical report NOTTCS-TR-98-3

- 4: Frank Atanassow, Johan Jeuring, Customizing an XML-Haskell data binding with type isomorphism inference in generic Haskell, 2007, Science of Computer Programming 65 (2007) 72-107, www.cs.uu.nl/~johanj, ,

- 5: David Mertz, XML Matters: Transcending the limits of DOM, SAX, and XSLT, Oct 2001, David Mertz (mertz@gnosis.cx), Prestidigitator, Gnosis Software, Inc., http://www.ibm.com/developerworks/xml/library/x-matters14.html, The HaXml functional programming model for XML,

- 6: Koen Roelandt, Using HaXml to clean legacy HTML pages, 2007, , http://www.krowland.net/tutorials/haxml_tutorial.html, This page explains how HaXml could be used to clean legacy HTML pages,

- 7: Paul Hudak, John Peterson, Joseph Fasel, A Gentle Introduction To Haskell, version 98, Revised June, 2000, Yale University, Los Alamos National Laboratory, http://haskell.org/tutorial/, ,

Weitere Resourcen:

- Glade: http://developer.gnome.org/tools/glade.html
- XML-Schema: http://w3.org/TR/xmlschema-1 und http://w3.org/TR/xmlschema-2
- JAXB: http://java.sun.com/developer/technicalArticles/WebServices/jaxb/
- Databinding: http://en.wikipedia.org/wiki/XML_data_binding
- Malcolm Wallace, Colin Runciman, „Haskell and XML: Generic Combinators or Type-Based Translation“: http://www.cs.york.ac.uk/fp/HaXml/icfp99.ps.gz [2]
- XML: http://www.w3.org/TR/REC-xml/
- DOM: http://www.w3.org/DOM/

- Funktionale Programmierung: http://en.wikipedia.org/wiki/Functional_programming
- HaXML: http://www.cs.york.ac.uk/fp/HaXml/
- HXML: http://www.flightlab.com/~joe/hxml/
- Haskell Xml Toolbox: http://www.fh-wedel.de/~si/HXmlToolbox/
- DrIFT: http://repetae.net/~john/computer/haskell/DrIFT/
- Validierung durch Ableitung regulärer Ausdrücke: http://www.fh-wedel.de/~si/seminare/ss02/Ausarbeitung/5.validierung/validation4.html basiert auf Janusz A. Brzozowski; *Derivatives of Regular Expressions*; Journal of the ACM, Volume 11, Issue 4, 1964
- GTK2HS: http://haskell.org/gtk2hs/

Anhang

A. Datentypen in Haskell

Um die Beispiele im Text besser nachvollziehen zu können hier nun eine kurze Übersicht über die erwähnten Datentypen mit Beispielen [7]:

- Aufzählungstypen

  Typen mit endlich vielen Werten. Der allgemeine Fall des Boolean Datentyps, entspricht im wesentlichen dem enum von C, ist allerdings wesentlich leistungsfähiger.

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
data Form       = Kreis | Rechteck | Quadrat | Dreieck
data Bool       = True | False
```

- Produkttypen (synonym: Verbundtypen) (single-constructor-values)

  Typen mit möglicherweise unendlich vielen Tupelwerten. Ein Produkt ist im wesentlichen ein "Typisiertes Tupel" mit Konstruktor. Es gibt hier *alternativlose*, *mehrstellige* Konstruktoren.

```
type Name  = String
type Alter = Int
data Leute = Person Name Geschlecht Alter
```

  Der Konstruktor von Leute sieht folgendermaßen aus:

```
Person :: Name -> Geschlecht -> Alter -> Leute
```

  Vergleich zu Tupel

```
type Leute = (Name, Geschlecht, Alter)
```

  Vorteile gegenüber Tupel

  - Markierung mit dem Konstruktor -- dient der Dokumentation.
  - Komponentenweise Typensicherheit
  - daraus folgend aussagekräftigere Fehlermeldungen

  Nachteile gegenüber Tupel

  - Längere Definitionen, weniger kompakt
  - Polymorphe Standard-Tupelfunktionen wie fst, snd, zip und unzip stehen nicht zur Verfügung.

  Beispiele

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float
```

- Summentypen (synonym: Vereinigungstypen) (multi-constructor-values)

  Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten. Wenn man Aufzählungstypen und Produkttypen kreuzt, bekommt man *mehrstellige Konstruktoren* mit *Alternativen*. Ein anderes Wort dafür lautet "Summentyp".

```
type Radius = Float
type Breite = Float
type Hoehe  = Float
data Figur  = Kreis Radius
            | Rechteck Breite Hoehe
```

  Aufzählungstypen sind nun (naheliegender Weise) nichts anderes, als eine Spezialisierung von Summentypen.

- Rekursive Typen

  Kennen wir in ähnlicher Form schon aus C, für die Definition von verketteten Listen und Bäumen. In Haskell ist es so, dass zu definierende Typnamen rechtsseitig der Definition benutzt werden.

```
data BinTree = Nil
             | Node Int BinTree BinTree
```

- Polymorphe Typen

  Ein (Daten-) Typ T heißt polymorph, wenn bei der Deklaration von T der Grundtyp der Elemente (in Form einer Typvariablen als Parameter) angegeben wird.

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)

data List a = Empty
            | (Head a) (List a)
```

- Maybe-Typ

  Maybe ist ein Monad zur Fehlerbehandlung. Es wird im allgemeinen zur Anzeige verwendet, ob ein Wert berechnet werden konnte. War eine Berechnung erfolgreich, hat es den Wert Just, ansonsten den Wert Nothing.

  Das Maybe Monad ist wie folgt definiert:

```
data Maybe a = Nothing
             | Just a

instance Monad Maybe where
    Just x  >>= k = k x
    Nothing >>= k = Nothing
    return        = Just
    fail s        = Nothing
```

  Beispiel zur Verwendung des Maybe Monads:

```
errDiv :: Int -> Int -> Maybe Int
errDiv x y
     | (y /= 0)  = Just (x 'div' y)
     | otherwise = Nothing
```

- Named-Fields

  Named fields sind nichts anderes als Namen für die Felder in einem Datentyp

```
data Configuration =
    Configuration { username      :: String,
                    localhost     :: String,
                    remotehost    :: String,
                    isguest       :: Bool,
                    issuperuser   :: Bool,
                    currentdir    :: String,
                    homedir       :: String,
                    timeconnected :: Integer
                  }
```

  Es werden die Zugriffsfunktionen:

```
username  :: Configuration -> String
localhost :: Configuration -> String
...
```

  gartis dazu geliefert.

B. Beispiel für Kombinatoren

Ein Beispiel Dokument-Verarbeitungs-Skript, das die generischen Filter-Kombinatoren benutzt.

```
module Main where
import Xml
main =
  processXmlWith (albumf `o` deep (tag "album"))

albumf =
  html
    [ hhead
      [ htitle
        [ txt `o` children `o` tag "artist"
              `o` children `o` tag "album"
        , literal ": "
        , keep /> tag "title" /> txt
        ]
      ]
    , hbody [("bgcolor",("white"!))]
      [ hcenter
          [ h1 [ keep /> tag "title" /> txt ] ]
      , h2 [ ("Notes"!) ]
      , hpara [ notesf `o` (keep /> tag "notes") ]
      , summaryf
      ]
    ]

notesf =
  foldXml (txt            ?> keep            :>
           tag "trackref" ?> replaceTag "EM" :>
           tag "albumref" ?> mkLink          :>
           children)

summaryf =
  htable [("BORDER",("1"!))]
    [ hrow [ hcol [ ("Album title"!) ]
           , hcol [ keep /> tag "title" /> txt ]
```

```
            ]
    , hrow [ hcol [ ("Artist"!) ]
            , hcol [ keep /> tag "artist" /> txt ]
            ]
    , hrow [ hcol [ ("Recording date"!) ]
            , hcol [ keep />
                        tag "recordingdate" /> txt ]
            ]
    , hrow [ hcola [ ("VALIGN",("top"!)) ]
                   [ ("Catalog numbers"!) ]
            , hcol
              [ hlist
                [ catno `oo`
                   numbered (deep (tag "catalogno"))
                ]
              ]
            ]
    ]

catno n =
  mkElem "LI"
    [ ((show n++". ")!),  ("label"?),  ("number"?)
    , (" ("!),  ("format"?),  (")"!) ]

mkLink =
  mkElemAttr "A" [ ("HREF",("link"?)) ]
    [ children ]
```

C. Beispiel für Typübersetzung

Ein Beispiel DTD2Haskell „***Adressbuch“***:

```
<!DOCTYPE adressbuch [
<!ELEMENT adressbuch (eintrag*)>
<!ELEMENT eintrag (name,adresse)>
<!ATTLIST eintrag id CDATA #REQUIRED>
<!ELEMENT adresse (strasse, hausnr, plz, ort)>
<!ATTLIST adresse typ (zuhause | buero) #REQUIRED>
<!ELEMENT name (vorname, nachname)>
<!ELEMENT vorname  (#PCDATA)>
<!ELEMENT nachname  (#PCDATA)>
<!ATTLIST nachname titel (Dipl-Inf | Dr | Prof) #IMPLIED>
<!ELEMENT strasse  (#PCDATA)>
<!ELEMENT hausnr  (#PCDATA)>
<!ELEMENT plz  (#PCDATA)>
<!ELEMENT ort  (#PCDATA)>
]>

module Adressbuch where
import Text.XML.HaXml.Xml2Haskell
import Text.XML.HaXml.OneOfN
import Char (isSpace)

{-Type decls-}
newtype Adressbuch = Adressbuch [Eintrag]            deriving (Eq,Show)
data Eintrag = Eintrag Eintrag_Attrs Name Adresse
             deriving (Eq,Show)
data Eintrag_Attrs = Eintrag_Attrs
    { eintragId :: String
```

```
    } deriving (Eq,Show)
data Adresse = Adresse Adresse_Attrs Strasse Hausnr Plz Ort
             deriving (Eq,Show)
data Adresse_Attrs = Adresse_Attrs
    { adresseTyp :: Adresse_typ
    } deriving (Eq,Show)
data Adresse_typ = Adresse_typ_zuhause  |  Adresse_typ_buero
                 deriving (Eq,Show)
data Name = Name Vorname Nachname
          deriving (Eq,Show)
newtype Vorname = Vorname String         deriving (Eq,Show)
data Nachname = Nachname Nachname_Attrs String
              deriving (Eq,Show)
data Nachname_Attrs = Nachname_Attrs
    { nachnameTitel :: (Maybe Nachname_titel)
    } deriving (Eq,Show)
data Nachname_titel = Nachname_titel_Dipl_Inf  |  Nachname_titel_Dr
                    |  Nachname_titel_Prof
                  deriving (Eq,Show)
newtype Strasse = Strasse String         deriving (Eq,Show)
newtype Hausnr = Hausnr String           deriving (Eq,Show)
newtype Plz = Plz String           deriving (Eq,Show)
newtype Ort = Ort String           deriving (Eq,Show)

{-Instance decls-}
instance XmlContent Adressbuch where
    fromElem (CElem (Elem "adressbuch" [] c0):rest) =
        (\(a,ca)->
           (Just (Adressbuch a), rest))
        (many fromElem c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Adressbuch a) =
        [CElem (Elem "adressbuch" [] (concatMap toElem a))]
instance XmlContent Eintrag where
    fromElem (CElem (Elem "eintrag" as c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (Just (Eintrag (fromAttrs as) a b), rest))
           (definite fromElem "<adresse>" "eintrag" ca))
        (definite fromElem "<name>" "eintrag" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Eintrag as a b) =
        [CElem (Elem "eintrag" (toAttrs as) (toElem a ++ toElem b))]
instance XmlAttributes Eintrag_Attrs where
    fromAttrs as =
        Eintrag_Attrs
          { eintragId = definiteA fromAttrToStr "eintrag" "id" as
          }
    toAttrs v = catMaybes
        [ toAttrFrStr "id" (eintragId v)
        ]
instance XmlContent Adresse where
    fromElem (CElem (Elem "adresse" as c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (\(c,cc)->
                 (\(d,cd)->
```

```
                    (Just (Adresse (fromAttrs as) a b c d), rest))
                (definite fromElem "<ort>" "adresse" cc))
            (definite fromElem "<plz>" "adresse" cb))
        (definite fromElem "<hausnr>" "adresse" ca))
    (definite fromElem "<strasse>" "adresse" c0)
fromElem (CMisc _:rest) = fromElem rest
fromElem (CString _ s:rest) | all isSpace s = fromElem rest
fromElem rest = (Nothing, rest)
toElem (Adresse as a b c d) =
    [CElem (Elem "adresse" (toAttrs as) (toElem a ++ toElem b ++
                                         toElem c ++ toElem d))]
instance XmlAttributes Adresse_Attrs where
fromAttrs as =
    Adresse_Attrs
      { adresseTyp = definiteA fromAttrToTyp "adresse" "typ" as
      }
toAttrs v = catMaybes
    [ toAttrFrTyp "typ" (adresseTyp v)
    ]
instance XmlAttrType Adresse_typ where
fromAttrToTyp n (n',v)
    | n==n'     = translate (attr2str v)
    | otherwise = Nothing
  where translate "zuhause" = Just Adresse_typ_zuhause
        translate "buero" = Just Adresse_typ_buero
        translate _ = Nothing
toAttrFrTyp n Adresse_typ_zuhause = Just (n, str2attr "zuhause")
toAttrFrTyp n Adresse_typ_buero = Just (n, str2attr "buero")
instance XmlContent Name where
fromElem (CElem (Elem "name" [] c0):rest) =
    (\(a,ca)->
       (\(b,cb)->
          (Just (Name a b), rest))
       (definite fromElem "<nachname>" "name" ca))
    (definite fromElem "<vorname>" "name" c0)
fromElem (CMisc _:rest) = fromElem rest
fromElem (CString _ s:rest) | all isSpace s = fromElem rest
fromElem rest = (Nothing, rest)
toElem (Name a b) =
    [CElem (Elem "name" [] (toElem a ++ toElem b))]
instance XmlContent Vorname where
fromElem (CElem (Elem "vorname" [] c0):rest) =
    (\(a,ca)->
       (Just (Vorname a), rest))
    (definite fromText "text" "vorname" c0)
fromElem (CMisc _:rest) = fromElem rest
fromElem (CString _ s:rest) | all isSpace s = fromElem rest
fromElem rest = (Nothing, rest)
toElem (Vorname a) =
    [CElem (Elem "vorname" [] (toText a))]
instance XmlContent Nachname where
fromElem (CElem (Elem "nachname" as c0):rest) =
    (\(a,ca)->
       (Just (Nachname (fromAttrs as) a), rest))
    (definite fromText "text" "nachname" c0)
fromElem (CMisc _:rest) = fromElem rest
fromElem (CString _ s:rest) | all isSpace s = fromElem rest
fromElem rest = (Nothing, rest)
toElem (Nachname as a) =
    [CElem (Elem "nachname" (toAttrs as) (toText a))]
instance XmlAttributes Nachname_Attrs where
```

```
    fromAttrs as =
        Nachname_Attrs
          { nachnameTitel = possibleA fromAttrToTyp "titel" as
          }
    toAttrs v = catMaybes
        [ maybeToAttr toAttrFrTyp "titel" (nachnameTitel v)
        ]
instance XmlAttrType Nachname_titel where
    fromAttrToTyp n (n',v)
        | n==n'     = translate (attr2str v)
        | otherwise = Nothing
      where translate "Dipl-Inf" = Just Nachname_titel_Dipl_Inf
            translate "Dr" = Just Nachname_titel_Dr
            translate "Prof" = Just Nachname_titel_Prof
            translate _ = Nothing
    toAttrFrTyp n Nachname_titel_Dipl_Inf = Just (n, str2attr "Dipl-Inf")
    toAttrFrTyp n Nachname_titel_Dr = Just (n, str2attr "Dr")
    toAttrFrTyp n Nachname_titel_Prof = Just (n, str2attr "Prof")
instance XmlContent Strasse where
    fromElem (CElem (Elem "strasse" [] c0):rest) =
        (\(a,ca)->
           (Just (Strasse a), rest))
        (definite fromText "text" "strasse" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Strasse a) =
        [CElem (Elem "strasse" [] (toText a))]
instance XmlContent Hausnr where
    fromElem (CElem (Elem "hausnr" [] c0):rest) =
        (\(a,ca)->
           (Just (Hausnr a), rest))
        (definite fromText "text" "hausnr" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Hausnr a) =
        [CElem (Elem "hausnr" [] (toText a))]
instance XmlContent Plz where
    fromElem (CElem (Elem "plz" [] c0):rest) =
        (\(a,ca)->
           (Just (Plz a), rest))
        (definite fromText "text" "plz" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Plz a) =
        [CElem (Elem "plz" [] (toText a))]
instance XmlContent Ort where
    fromElem (CElem (Elem "ort" [] c0):rest) =
        (\(a,ca)->
           (Just (Ort a), rest))
        (definite fromText "text" "ort" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (Ort a) =
        [CElem (Elem "ort" [] (toText a))]
{-Done-}
```

Ein Beispiel DTD2Haskell „***TV-Zeitschrift***“:

```
<!DOCTYPE TVSCHEDULE [
<!ELEMENT TVSCHEDULE (CHANNEL+)>
<!ELEMENT CHANNEL (BANNER,DAY+)>
<!ELEMENT BANNER (#PCDATA)>
<!ELEMENT DAY (DATE,(HOLIDAY|PROGRAMSLOT+)+)>
<!ELEMENT HOLIDAY (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT PROGRAMSLOT (TIME,TITLE,DESCRIPTION?)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ATTLIST TVSCHEDULE NAME CDATA #REQUIRED>
<!ATTLIST CHANNEL CHAN CDATA #REQUIRED>
<!ATTLIST PROGRAMSLOT VTR CDATA #IMPLIED>
<!ATTLIST TITLE RATING CDATA #IMPLIED>
<!ATTLIST TITLE LANGUAGE CDATA #IMPLIED>
]>

module Tv_schedule where
import Text.XML.HaXml.Xml2Haskell
import Text.XML.HaXml.OneOfN
import Char (isSpace)

{-Type decls-}
data TVSCHEDULE = TVSCHEDULE TVSCHEDULE_Attrs (List1 CHANNEL)
                deriving (Eq,Show)
data TVSCHEDULE_Attrs = TVSCHEDULE_Attrs
    { tVSCHEDULENAME :: String
    } deriving (Eq,Show)
data CHANNEL = CHANNEL CHANNEL_Attrs BANNER (List1 DAY)
             deriving (Eq,Show)
data CHANNEL_Attrs = CHANNEL_Attrs
    { cHANNELCHAN :: String
    } deriving (Eq,Show)
newtype BANNER = BANNER String           deriving (Eq,Show)
data DAY = DAY DATE (List1 (OneOf2 HOLIDAY (List1 PROGRAMSLOT)))
         deriving (Eq,Show)
newtype HOLIDAY = HOLIDAY String         deriving (Eq,Show)
newtype DATE = DATE String         deriving (Eq,Show)
data PROGRAMSLOT = PROGRAMSLOT PROGRAMSLOT_Attrs TIME TITLE
                               (Maybe DESCRIPTION)
                 deriving (Eq,Show)
data PROGRAMSLOT_Attrs = PROGRAMSLOT_Attrs
    { pROGRAMSLOTVTR :: (Maybe String)
    } deriving (Eq,Show)
newtype TIME = TIME String         deriving (Eq,Show)
data TITLE = TITLE TITLE_Attrs String
           deriving (Eq,Show)
data TITLE_Attrs = TITLE_Attrs
    { tITLERATING :: (Maybe String)
    , tITLELANGUAGE :: (Maybe String)
    } deriving (Eq,Show)
newtype DESCRIPTION = DESCRIPTION String       deriving (Eq,Show)

{-Instance decls-}
instance XmlContent TVSCHEDULE where
    fromElem (CElem (Elem "TVSCHEDULE" as c0):rest) =
        (\(a,ca)->
           (Just (TVSCHEDULE (fromAttrs as) a), rest))
        (definite fromElem "CHANNEL+" "TVSCHEDULE" c0)
```

```
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (TVSCHEDULE as a) =
        [CElem (Elem "TVSCHEDULE" (toAttrs as) (toElem a))]
instance XmlAttributes TVSCHEDULE_Attrs where
    fromAttrs as =
        TVSCHEDULE_Attrs
          { tVSCHEDULENAME = definiteA fromAttrToStr "TVSCHEDULE" "NAME" as
          }
    toAttrs v = catMaybes
        [ toAttrFrStr "NAME" (tVSCHEDULENAME v)
        ]
instance XmlContent CHANNEL where
    fromElem (CElem (Elem "CHANNEL" as c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (Just (CHANNEL (fromAttrs as) a b), rest))
           (definite fromElem "DAY+" "CHANNEL" ca))
        (definite fromElem "<BANNER>" "CHANNEL" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (CHANNEL as a b) =
        [CElem (Elem "CHANNEL" (toAttrs as) (toElem a ++ toElem b))]
instance XmlAttributes CHANNEL_Attrs where
    fromAttrs as =
        CHANNEL_Attrs
          { cHANNELCHAN = definiteA fromAttrToStr "CHANNEL" "CHAN" as
          }
    toAttrs v = catMaybes
        [ toAttrFrStr "CHAN" (cHANNELCHAN v)
        ]
instance XmlContent BANNER where
    fromElem (CElem (Elem "BANNER" [] c0):rest) =
        (\(a,ca)->
           (Just (BANNER a), rest))
        (definite fromText "text" "BANNER" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (BANNER a) =
        [CElem (Elem "BANNER" [] (toText a))]
instance XmlContent DAY where
    fromElem (CElem (Elem "DAY" [] c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (Just (DAY a b), rest))
           (definite fromElem "(HOLIDAY|PROGRAMSLOT+)+" "DAY" ca))
        (definite fromElem "<DATE>" "DAY" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (DAY a b) =
        [CElem (Elem "DAY" [] (toElem a ++ toElem b))]
instance XmlContent HOLIDAY where
    fromElem (CElem (Elem "HOLIDAY" [] c0):rest) =
        (\(a,ca)->
           (Just (HOLIDAY a), rest))
        (definite fromText "text" "HOLIDAY" c0)
    fromElem (CMisc _:rest) = fromElem rest
```

```
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (HOLIDAY a) =
        [CElem (Elem "HOLIDAY" [] (toText a))]
instance XmlContent DATE where
    fromElem (CElem (Elem "DATE" [] c0):rest) =
        (\(a,ca)->
           (Just (DATE a), rest))
        (definite fromText "text" "DATE" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (DATE a) =
        [CElem (Elem "DATE" [] (toText a))]
instance XmlContent PROGRAMSLOT where
    fromElem (CElem (Elem "PROGRAMSLOT" as c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (\(c,cc)->
                 (Just (PROGRAMSLOT (fromAttrs as) a b c), rest))
              (fromElem cb))
           (definite fromElem "<TITLE>" "PROGRAMSLOT" ca))
        (definite fromElem "<TIME>" "PROGRAMSLOT" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (PROGRAMSLOT as a b c) =
        [CElem (Elem "PROGRAMSLOT" (toAttrs as) (toElem a ++ toElem b ++
                                                 maybe [] toElem c))]
instance XmlAttributes PROGRAMSLOT_Attrs where
    fromAttrs as =
        PROGRAMSLOT_Attrs
          { pROGRAMSLOTVTR = possibleA fromAttrToStr "VTR" as
          }
    toAttrs v = catMaybes
        [ maybeToAttr toAttrFrStr "VTR" (pROGRAMSLOTVTR v)
        ]
instance XmlContent TIME where
    fromElem (CElem (Elem "TIME" [] c0):rest) =
        (\(a,ca)->
           (Just (TIME a), rest))
        (definite fromText "text" "TIME" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (TIME a) =
        [CElem (Elem "TIME" [] (toText a))]
instance XmlContent TITLE where
    fromElem (CElem (Elem "TITLE" as c0):rest) =
        (\(a,ca)->
           (Just (TITLE (fromAttrs as) a), rest))
        (definite fromText "text" "TITLE" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (TITLE as a) =
        [CElem (Elem "TITLE" (toAttrs as) (toText a))]
instance XmlAttributes TITLE_Attrs where
    fromAttrs as =
        TITLE_Attrs
          { tITLERATING = possibleA fromAttrToStr "RATING" as
```

```
          , tITLELANGUAGE = possibleA fromAttrToStr "LANGUAGE" as
          }
    toAttrs v = catMaybes
        [ maybeToAttr toAttrFrStr "RATING" (tITLERATING v)
        , maybeToAttr toAttrFrStr "LANGUAGE" (tITLELANGUAGE v)
        ]
instance XmlContent DESCRIPTION where
    fromElem (CElem (Elem "DESCRIPTION" [] c0):rest) =
        (\(a,ca)->
           (Just (DESCRIPTION a), rest))
        (definite fromText "text" "DESCRIPTION" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (DESCRIPTION a) =
        [CElem (Elem "DESCRIPTION" [] (toText a))]
{-Done-}
```

Ein Beispiel DTD2Haskell „***Zeitungsartikel***“:

```
<!DOCTYPE NEWSPAPER [
<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>
<!ENTITY NEWSPAPER "Vervet Logic Times">
<!ENTITY PUBLISHER "Vervet Logic Press">
<!ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">
]>

module Newspaper_article where
import Text.XML.HaXml.Xml2Haskell
import Text.XML.HaXml.OneOfN
import Char (isSpace)

{-Type decls-}
newtype NEWSPAPER = NEWSPAPER (List1 ARTICLE)        deriving (Eq,Show)
data ARTICLE = ARTICLE ARTICLE_Attrs HEADLINE BYLINE LEAD BODY
                       NOTES
             deriving (Eq,Show)
data ARTICLE_Attrs = ARTICLE_Attrs
    { aRTICLEAUTHOR :: String
    , aRTICLEEDITOR :: (Maybe String)
    , aRTICLEDATE :: (Maybe String)
    , aRTICLEEDITION :: (Maybe String)
    } deriving (Eq,Show)
newtype HEADLINE = HEADLINE String       deriving (Eq,Show)
newtype BYLINE = BYLINE String           deriving (Eq,Show)
newtype LEAD = LEAD String         deriving (Eq,Show)
newtype BODY = BODY String         deriving (Eq,Show)
newtype NOTES = NOTES String       deriving (Eq,Show)
```

```
{-Instance decls-}
instance XmlContent NEWSPAPER where
    fromElem (CElem (Elem "NEWSPAPER" [] c0):rest) =
        (\(a,ca)->
           (Just (NEWSPAPER a), rest))
        (definite fromElem "ARTICLE+" "NEWSPAPER" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (NEWSPAPER a) =
        [CElem (Elem "NEWSPAPER" [] (toElem a))]
instance XmlContent ARTICLE where
    fromElem (CElem (Elem "ARTICLE" as c0):rest) =
        (\(a,ca)->
           (\(b,cb)->
              (\(c,cc)->
                 (\(d,cd)->
                    (\(e,ce)->
                       (Just (ARTICLE (fromAttrs as) a b c d e), rest))
                    (definite fromElem "<NOTES>" "ARTICLE" cd))
                 (definite fromElem "<BODY>" "ARTICLE" cc))
              (definite fromElem "<LEAD>" "ARTICLE" cb))
           (definite fromElem "<BYLINE>" "ARTICLE" ca))
        (definite fromElem "<HEADLINE>" "ARTICLE" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (ARTICLE as a b c d e) =
        [CElem (Elem "ARTICLE" (toAttrs as) (toElem a ++ toElem b ++
                                             toElem c ++ toElem d ++ toElem e))]
instance XmlAttributes ARTICLE_Attrs where
    fromAttrs as =
        ARTICLE_Attrs
          { aRTICLEAUTHOR = definiteA fromAttrToStr "ARTICLE" "AUTHOR" as
          , aRTICLEEDITOR = possibleA fromAttrToStr "EDITOR" as
          , aRTICLEDATE = possibleA fromAttrToStr "DATE" as
          , aRTICLEEDITION = possibleA fromAttrToStr "EDITION" as
          }
    toAttrs v = catMaybes
        [ toAttrFrStr "AUTHOR" (aRTICLEAUTHOR v)
        , maybeToAttr toAttrFrStr "EDITOR" (aRTICLEEDITOR v)
        , maybeToAttr toAttrFrStr "DATE" (aRTICLEDATE v)
        , maybeToAttr toAttrFrStr "EDITION" (aRTICLEEDITION v)
        ]
instance XmlContent HEADLINE where
    fromElem (CElem (Elem "HEADLINE" [] c0):rest) =
        (\(a,ca)->
           (Just (HEADLINE a), rest))
        (definite fromText "text" "HEADLINE" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (HEADLINE a) =
        [CElem (Elem "HEADLINE" [] (toText a))]
instance XmlContent BYLINE where
    fromElem (CElem (Elem "BYLINE" [] c0):rest) =
        (\(a,ca)->
           (Just (BYLINE a), rest))
        (definite fromText "text" "BYLINE" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
```

```
    fromElem rest = (Nothing, rest)
    toElem (BYLINE a) =
        [CElem (Elem "BYLINE" [] (toText a))]
instance XmlContent LEAD where
    fromElem (CElem (Elem "LEAD" [] c0):rest) =
        (\(a,ca)->
           (Just (LEAD a), rest))
        (definite fromText "text" "LEAD" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (LEAD a) =
        [CElem (Elem "LEAD" [] (toText a))]
instance XmlContent BODY where
    fromElem (CElem (Elem "BODY" [] c0):rest) =
        (\(a,ca)->
           (Just (BODY a), rest))
        (definite fromText "text" "BODY" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (BODY a) =
        [CElem (Elem "BODY" [] (toText a))]
instance XmlContent NOTES where
    fromElem (CElem (Elem "NOTES" [] c0):rest) =
        (\(a,ca)->
           (Just (NOTES a), rest))
        (definite fromText "text" "NOTES" c0)
    fromElem (CMisc _:rest) = fromElem rest
    fromElem (CString _ s:rest) | all isSpace s = fromElem rest
    fromElem rest = (Nothing, rest)
    toElem (NOTES a) =
        [CElem (Elem "NOTES" [] (toText a))]
{-Done-}
```