Seminar "Logische Aspekte von XML"
Ausarbeitung zum 1.Teil des Vortrags
"XML Schema"

Andreas Freier
(freier@studcs.uni-sb.de)

22. Oktober 2003

Inhaltsverzeichnis

1 Einführung

XML Schema ist eine Spezifikationssprache für XML-Dokumente, die vom World-Wide Web Consortium (W3C) als Standard entwickelt wurde.

Aus der Sicht der Entwickler, die sich mit XML beschäftigen, ist XML Schema eine bessere Technologie im Vergleich zu der DTD. DTD hat eine von XML unterschiedliche Syntax, was zusätzliche Parser erforderlich macht, und es ist nicht möglich, andere primitive Typen als Strings für den Inhalt eines Elements anzugeben.

XML Schema ist ein umfangreicher Standard auf über 300 gedruckten Seiten, der dem Entwickler eine Menge Features bietet, es gibt z.B. allein 19 primitive Typen.

Ein Szenario aus der Praxis könnte der Einsatz von XML Schema in XML-Datenbanken sein, die ein oder mehrere Dokumenttypen enthalten. Für jeden Dokumenttyp definiert ein Schema eine Menge von gültigen XML-Dokumenten. Beim Schreiben in die Datenbank werden Dokumente, auf ihre Gültigkeit gegen das entschprechende Schema überprüft, so daß sich Datenbank nur gültige Dokumente befinden. Ein weiterer Vorteil ist, daß jedes Schemas eine schriftliche Referenz für einen Dokumenttyp darstellt.

Trotz der starken Kritik an XML (u.a. [Crane2002]) aus der technologischen Sicht, ist XML stark verbreitet und interessiert zunehmend auch die Wissenschaftler.

In [Essence2003] wird eine Formalisierung des XML Schema als eine der ersten Anwendungen formaler Methoden auf einen Industrie-Standard beschrieben.

Das vorliegende Dokument bietet einen Überblick über das formale Modell des XML Schema von Wadler und Simeon.

2 Die Essenz von XML

XML wird als ein externes Format zur Darstellung von Daten hochgejubelt.

Dafür sind zwei Eigenschaften notwendig:

- *Self-describing:* Man muß aus der externen Darstellung die interne Darstellung ableiten können
- *Round-tripping:* beim Konvertieren aus der internen Darstellung in die externe und wieder zurück muß die neue interne Darstellung mit der alten identisch sein

LISPs S-expressions beispielsweise besitzen beide Eigenschaften.

XML besitzt keine der beiden Eigenschaften.

Es ist nicht immer selbstbeschreibend. Es soll z.B. das folgende XML-Element in die LISP-Syntax überführt werden:

```
<foo>1 2 3</foo>
```

Aus dem XML allein geht nicht hervor, ob das Ergebnis

```
(foo "1 2 3")
```

oder das Ergebnis

```
(foo 1 2 3)
```

richtig ist.

Deswegen hängt das interne Datenformat, das einer externen XML-Darstellung entspricht, stark vom zur Validierung benutzten XML Schema ab.

Auch round-tripping ist nicht immer gegeben, da Schemas zulässig sind, in denen Integers und Strings gemischt werden können.

Die folgenden S-expressions

```
(bar 1 "two" 3)
```

und

```
(bar 1 "two" "3")
```

als internes Format ergeben dieselbe XML-Darstellung

```
<bar>1 two 3</bar>
```

wobei die Information verloren geht, welche der beiden S-expressions die Quelle war. Die Konvertierung in das interne Format zurück ist somit nicht eindeutig.

Philip Wadler und Jerome Simeon behaupten, die Essenz von XML sei, daß die Aufgabe von XML nicht schwer ist und daß XML diese Aufgabe nicht besonders gut löst.

3 Überblick über das formale Modell

Im Jahre 1985 sollte im Rahmen der Strategic Defense Initiative ein Test durchgeführt werden, bei dem ein Laserstrahl aus einem Observatorium auf Hawaii von einem Spiegel auf der Unterseite eines Space Shuttle weitergeleitet werden sollte. Als der Astronaut die Höhe des Lasers eingegeben hat, hat sich das Shuttle umgedreht. Das Problem war, daß der Astronaut die Höhe von 10023 Fuß (unter dem Shuttle) eingegeben hat, das System den Wert jedoch als 10023 Meilen (über dem Shuttle) interpretiert hat. Deswegen die Umdrehung.

An diesem Beispiel wird der Unterschied zwischen benannten und strukturellen Typen verdeutlicht.

Benannte und strukturelle Typen

Wir definieren zwei Typen

```
type Feet = Integer
type Miles = Integer
```

Wenn es sich um strukturelle Typen handelt, sind Feet und Miles nur Synonyme für den Typ Integer. Die beiden Typen sind gleich, weil sie gleiche Struktur haben.

Wenn es bennante Typen sind, sind sie unterschiedlich, weil ihre Namen unterschiedlich sind.

Dem System des Space Shuttle einen Wert von Typ Feet zu übergeben, während es einen Wert vom Typ Miles erwartet, würde eine Fehlermeldung verursachen und eine Möglichkeit zur Verbesserung der Eingabe bieten.

Bennante Typen stellen ein Sicherheitsmechanismus zur Verfügung. Einen weiterer Vorteil wird dargestellt nach der Erklärung der Validierung.

Eine der Besonderheiten des vorliegenden Modells ist die Verwendung benannter Typen für XML Schema, während alle früheren Modelle einen rein strukturellen Ansatz hatten.

Weitere Details

Normalerweise werden Schemas in der XML Notation definiert. Für das Modell wurde jedoch eine kompaktere und besser lesbare Notation verwendet.

Das folgende XML Schema

```
<xs:simpleType name="feet">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:element name="height" type="feet"/>
```

sieht so

```
define type feet restricts xs:integer
define element height of type feet
```

in der benutzten Notation aus, die weitgehend selbsterklärend ist.

Es werden im Modell nur Kernkonzepte von XML Schema behandelt. Statt der 19 primitiven Typen werden z.B. nur string und integer benutzt.

Eine vollständige Formalisierung von XML Schema ist im Rahmen der formalen Sematik von XQuery zu finden. Dort wird auch das Überführen des XML Notation in die benutzte Notation spezifiziert.

Typprüfung, Validierung, Löschung

Frühere Modelle von XML Schema basierten auf der Vorstellung, daß überprüft wird, ob ein Wert einem Typ gehört oder nicht. Eine andere Besonderheit des vorliegenden Modells ist, daß hier behandelt wird, wie die Validierung aus einem exteren, ungetypten Wert, einen internen Wert mit Typannotationen erzeugt und wie die Löschung einen internen Wert in einen externen Wert überführt.

In Programmiersprachen kann ein Wert einem Typ gehören oder nicht.

Wenn wir in Java schreiben

```
int x = 10;
```

dann gehört die 10 in der Variable x dem typ int.

Wenn wir aber

```
int y = true;
```

schreiben, bekommen wir einen Typfehler, da true kein Wert vom Typ int ist.

Im vorliegenden Modell wird ein externer Wert zuerst validiert, das bedeutet, daß der externe ungetypte Wert in einen internen Wert mit Typannotationen überführt wird. Nur getypte Werte können dann einer Typprüfung unterzogen werden. Alle ungetypten Werte sind vom Typ *anyType*, der Wurzel der Typhierarchie in XML Schema, und können nicht einem kleineren Typ zugehören.

Wir betrachten den Wert

```
<height>10023</height>
```

Der Typ height wurde weiter oben bereits definiert.

Im Modell wird vor der Validierung zuerst die Notation entsprechend umgewandelt:

```
  <height>10023</height>
⇒
  element height { ”10023” }
```

Die Validierung annotiert dann den ungetypten Wert mit Typen:

```
  validate as element height { ”10023” }
⇒
  element height of type feet { 10023 }
```

Nun kann eine Typprüfung durchgeführt werden:

```
  element height of type feet { 10023 }
matches
  element height
```

Die Löschung ist das Gegenteil der Validierung. Sie überführt einen internen getypten Wert in einen externen ungetypten.

```
  element height of type feet { 10023 }
erases to
  element height { ”10023” }
```

Gültigkeitstheorem

Mit den oben definierten Begriffen kann nun die zentrale Aussage der Arbeit von Wadler und Simeon formuliert werden.

validate as *Type* { *UntypedValue* } ⇒ *Value*

genau dann, wenn

Value **matches** *Type*

Value **erases to** *UntypedValue*

Die Aussage mag selbstversätndlich klingen, war jedoch zu Beginn der Arbeit an dem Modell keineswegs selbstverständlich.Die Forscher berichten von zwei Überraschungen während der Arbeit daran. Die Schwierigkeit lag daran, die Wechselwirkung zwischen den Typnamen und Strukturen zu erkennen. Die erste Überraschung war, daß als die benannten Typen und die Validierung festgehalten wurden, die große Anzahl der Features von XML Schema auf einmal perfekt in das vorliegende Gerüst gepaßt hat. Die zweite Überraschung war, daß trotz der Komplexität von XML Schema das resultierende Gültigkeitstheorem sich als einfach herausgestellt hat.

Vorteil der benannten Typen

Die Verwendung bennanter Typen macht die Typprüfung unnötig, da durch die Validierung sichergestellt wird, daß jedes validierte Element einen Typnamen enthält, der seinen Inhalt exakt charakterisiert. Das macht die Implementierung einfacher und effizienter.

Um festzustellen, daß ein Element einem Typen zugehört, muß lediglich der Name des Elements betrachtet werden. Somit kann Subtyping-Prüfung durch endliche Automaten und nicht durch Baumautomaten wie in Modellen, in denen strukturelle Typen verwendet wurden, realisiert werden

4 XML Schema im Beispiel

Um die formalen Definitionen von Werten und Typen besser verstehen zu können, werden im Folgenden unterschiedliche Aspekte von XML Schema in Beispielen dargestellt.

Einfache und komplexe Typen.

Hier werden zwei Elemente eines einfachen Typs, ein Element eines komplexen Typs und ein komplexer Typ deklariert.

```
define element title of type string
define element author of type string
define element paper of type paperType
define type paperType {
  element title,
  element author+
}
```

Globale und lokale Deklarationen.

Elemente können auf der globalen Ebene, aber auch lokal, innerhalb einer Typdefinition deklariert werden.

```
define element paper of type paperType
define type paperType {
  element title of type string,
  element author of type string+
}
```

Lokale Deklarationen bieten die Möglichkeit, Elemente mit dem gleichen Namen, im unterschiedlichen Kontext zu deklarieren.

Atomare Typen, Listen und Vereinigungen.

Atomare Typen sind in dem Modell die Type string und integer.

Listen werden durch die regulären Operatoren ?, * und + gebildet.

```
define element ints of type intList
define type intList { integer+ }
```

Vereinigungen werden durch den Operator | erzeugt, wobei das aufgeführte Beispiel einen mehrdeutigen Typ darstellt, der problematisch werden kann.

```
define element fact of type intOrStrList
define type intOrStrList { (integer | string)* }
```

Ableitung durch Einschränkung einfacher Typen.

Neue einfache Typen können durch Einschränkung anderer einfacher Typen gebildet werden.

```
define type miles restricts integer
define type feet restricts integer
```

Im Beispiel des Space Shuttle wären dies die beiden Typen für Werte in Meilen und in Fuß, die die beschriebene Umdrehung vermieden hätten.

Ableitung durch Einschränkung komplexer Typen.

Neue komplexe Typen können durch die Einschränkung vorhandener komplexer Typen entstehen. Es ist dann möglich, den eingeschränkten Typ zu übergeben, wenn der Grundtyp erwartet wird.

```
define type publicationType {
  element author*,
  element title?,
  element journal?,
  element year?
}
define type articleType restricts publicationType {
  element author+,
  element title,
  element journal,
  element year
}
define type bookType restricts publicationType {
  element author+,
  element title,
  element year
}
define element book of type bookType
define element article of type articleType
```

Ableitung durch Erweiterung.

Neue komplexe Typen können auch durch Erweiterung vorhandener komplexer Typen enstehen, die vergleichbar mit der Vererbung in objektorientierten Programmiersprachen ist.

```
define type color restricts string
define type pointType {
  element x of type integer,
  element y of type integer
}
define type colorPointType extends pointType {
  element c of type color
}
define element point of type pointType
define element colorPoint of type colorPointType
```

5 Werte

Hier folgt die formale Definition der Werte.

Namen, Strings und Integers sind Primitive.

Definition getypter Werte:

```
Value   ::= ()
          | Item(,Item)
Item    ::= Element
          | String
          | Integer
Element ::= element ElementName OfType? { Value }
OfType  ::= of type TypeName
```

Werte ohne Typannotationen sind vom Typ anyType.

Definition ungetypter Werte:

```
UntypedValue ::= ()
               | UntypedItem(,UntypedItem)
UntypedItem  ::= element ElementName {UntypedValue}
               | String
```

Ungetypte Werte beschreiben meistens XML-Dokumente vor der Validierung.

6 Typen

Hier folgt die formale Definition der Typen.

```
Type        ::= ()
              | ItemType
              | Type, Type
              | Type | Type
              | Type?
              | Type+
              | Type*
ItemType    ::= ElementType
              | integer
              | string
ElementType ::= element ElementName? OfType?
```

Wenn bei einem ElementType nur der Name des Elements vorkommt, handelt es sich um ein global deklariertes Element. Ein Elementname mit einem Typnamen ist eine lokale Deklaration. Ein Typname allein matcht jedes Element, das vom angegebenen Typ abgeleitet ist. Das Word element allein matcht jedes Element.

```
Definition  ::= define element ElementName OfType
              | define type TypeName TypeDerivation
```

Die Definition ist das Top-Level des Systems, wo global Typen und Elemente deklariert werden können.

TypeDerivation erlaubt Enschränkung einfacher und komplexer Typen, bzw. die Erweiterung komplexer Typen.

```
TypeDerivation ::= restricts integer
                 | restricts string
                 | restricts TypeName { Type }
                 | extends TypeName { Type }
```

Der Typ anyType ist wie folgt deklariert:

```
define type anyType restricts anyType {
  (integer | string | element)*
}
```

7 Schluß

In der Ausarbeitung zum 2.Teil des Vortrags behandelt Tobias Maurer das Gültigkeitstheorem für XML Schema im Detail. Außerdem stellt er formal fest, daß die Überprüfung der Typzugehörigkeit bei validierten Werten nicht notwendig ist, da die Validierung von ungetypten Werten bereits sinnvoll getypte Werte liefert.

Literatur

[Essence2003] Philip Wadler, Jerome Simeon. The Essence of XML, 2003.

[Crane2002] Aaron Crane. Does XML Suck? Or: Why XML is Technologically Terrible, but You Have to Use It Anyway, 2002.