XML Programmierung mit XDuce

Proseminar Logische Aspekte von XML Sommersemester 2003
Prof. Gert Smolka, Tim Priesnitz, Andreas Rossberg

Taoufik Romdhane, taoufic@gmx.de

14th November 2003

Contents

Einführung

In diesem Dokument wird eine statisch typisierte Sprache um XML-Dokumente zu erstellen, bearbeiten und verifizieren: XDuce beschrieben. XDuce ist eine funktionale, domänenspezifische Programmiersprache. Sie wurde in OCaml implementiert, und ist frei (im Sinne von OpenSource). Die Hauptautoren sind Benjamin Pierce, Haruo Hosoya, und Peter Bruneman.

Die Werte der Sprache sind XML-Dokumente. Typen entsprechen den "Document Schema". Es sind reguläre Ausdrücke Typen. Ausserdem basiert das Typsystem direkt auf der regulären Baumautomaten-Theorie. Ein anderes wichtiges Feature von XDuce ist sein flexibles reguläres Pattern-Matching. Sie sehen zwar dem Pattern-Matching anderer Sprache ähnlich (z.B. ML), sind aber viel mächtiger.

Die Entwicklung von XDuce ist noch nicht abgeschlossen, bisher existiert eigentlich nur ein Prototyp. Die eigentliche Arbeit des laufenden XDuce Projektes besteht darin, die Algorithmen für Typechecking und Pattern-Matching zu implementieren. Das Backend ist ein einfacher Interpreter. Es existieren bereits Programme, welche in XDuce geschrieben wurden. Der Quellcode kann frei heruntergeladen werden [WXDUCE].

Zuerst gehen wir kurz drauf ein, wie Werte in XDuce aussehen. Dann wird das Typsystem der Sprache beschrieben und auf das etwas komplexe Pattern-Matching eingegangen. In einer anderen Arbeit (von Daniel Beck) wird noch betrachtet, wie Funktionen definiert sind, und die Restriktionen, die für die Entscheidbarkeit der Sprache notwendig sind.

1 Werte

In diesem Punkt schauen wir uns an, wie Werte aussehen, und wie sie definiert sind. Werte sehen XML Dokumenten sehr ähnlich, bis auf wenige, syntaktische Unterschiede. Folgendes ist ein Beispiel von einem XML Dokument :

<addrbook>
<name>Haruo Hosoya</name>
<addr>Tokio</addr>
<name>ABC</name>
<addr>Def</addr>
<tel>122-3242-324</tel>
<name>Benjamin Pierce</name>
<addr>Philadelphia</addr>
<addrbook>

Benutzt man die XDuce Syntax, kann man das obige XML Dokument wie folgt beschreiben:

val mybook = addrbook[
name["Haruo Hosoya"]
addr["Tokio"]
name["ABC"]
addr["Def"]
tel["122-3242-324"]
name["Benjamin Pierce"]
addr["Philadelphia"]]

Die XDuce Implementierung erlaubt beide gezeigten Möglichkeiten, um Werte zu erzeugen. Man sieht leicht, dass die XDuce BasisTypen eins zu eins in ein XML Dokument umgewandeln werden können, statt <label> ...</label> schreibt man in den XDuce Typen [label...], das vereinfacht das Lesen. Im ersten Fall wird eine Typüberprüfung durchgeführt, um sicherzustellen, dass das Dokument korrekt ist. Wie Typen in XDuce definiert sind, ist Thema des folgenden Kapitels.

2 Typen

Wir gehen auf die nicht ganz trivialen Typeigenschaften von XDuce ein, definieren die Syntax und Semantik von Typen und erklären anhand eines Beispiels, wie entschieden wird, ob ein Typ in einem anderen "enthalten ist".

2.1 Typen - Syntax

Das Typsystem in XDuce ist sehr mächtig - mit regulären Ausdrücke kann man die Typen zusammen bauen. Sie sehen der DTD sehr ähnlich sind aber viel Mächtiger - eigentlich sind sie genauso stark wie das "Typ Schema".

Ein Beispiel :

person[name[String],email[String],tel[String]]

beschreibt alle Werte, die ein Label "Person" enthalten - das Label enthält weitere Label "name, email und tel", die alle drei einen String enthalten. Man kann auch komplexere Typen beschreiben, z.B. :

person[name[String],email[String],tel[String]?]*

"*" heisst, dass null oder mehrere email-Label vorkommen, und ? beudetet, dass ein "tel" vorkommt - oder gar keins. Man kann sich nun ziemlich leicht noch viel komplexere Typen vorstellen. XDuce Typen heissen "reguläre Ausdrucks-Typen". Man kann auch rekursive Typen schreiben. Zum Beispiel ist

Type a = 1 , a

erlaubt. Auch erlaubt ist:

Typ a = b
Typ b = a

Typausdrücke sind wie folgt definiert :

(Syntax von Typen)

T::=X	*Name des Types*
()	*Leere Sequenze*
T,T	*Konkatenation*
$L[T]$	*Label*
T/T	*Vereinigung*
T^*	*Wiederholung*

wobei X eine abzählbar unendliche Menge von Typnamen ist.

Die Symbole "?" und "*", die wir in unserem bisherigem Beispiel benutzt haben, sind nur Abkürzungen für:

$T? \equiv T|()$
$T+ \equiv T, T^*.$

2.2 Typen als Menge von Werten

Typen in XDuce sind als Mengen von Werten zu verstehen. Dieses Typkonzept ist zwar ungewöhnlich, aber sehr mächtig. Man kann diese Menge dann noch zum Beispiel vereinigen, oder Mengen haben die in anderen enthalten sind. - Welche Menge jedem Typ zugeordnet ist, wird mit der Funktion $\langle . \rangle$beschrieben:

$$\langle B \rangle \, \rho = V_B$$

$$\langle () \rangle \, \rho = \{ \, () \, \}$$

$$\langle l[T] \rangle \, \rho = \{ l[f] | f \in \langle T \rangle \, \rho \}$$

$$\langle T_1, T_2 \rangle \, \rho = \{ f, g | f \in \langle T_1 \rangle \, \rho \wedge g \in \langle T_2 \rangle \, \rho \}$$

$$\langle T_1 | T_2 \rangle \, \rho = \langle T_1 \rangle \, \rho \cup \langle T_2 \rangle \, \rho$$

$$\langle X \rangle \, \rho = \rho(X)$$

ρ ist eine Umgebung die Typvariablen auf Mengen von Werten abbildet, $\langle . \rangle$ist die Funktion, die einen Typ und eine Umgebung ρ erhält und eine Menge von Werten zurückgibt. Wir gehen nun zur Semantik der Typen über.

2.3 Typsemantik

Die Semantik der Typen ist gegeben durch die Relation $v \in T$, lies Wert v hat den Typ T - die kleinste Relation, die unter der folgenden Regeln abgeschlossen ist:

$() \in ()$ *(ET-EMP)*

$\frac{E(X)=T \; v \in T}{v \in X}$ *(ET-VAR)*

$\frac{v \in T \; l \in L}{l[v] \in L[T]}$ *(ET-LAB)*

$\frac{v_1 \in T_1 \; v_2 \in T_2}{v_1, v_2 \in T_1, T_2}$ *(ET-CAT)*

$\frac{v \in T_1}{v \in T_1 | T_2}$ *(ET-OR1)*

$\frac{v \in T_2}{v \in T_1 | T_2}$ *(ET-OR2)*

$\frac{v_i \in T \; \forall i}{v_1, \ldots v_n \in T^*}$ *(ET-REP)*

2.4 Subtypen

XDuce ist eine streng typisierte Sprache. Das ist auch gut so - das verhindert viele Programmierfehler. In XDuce können Typen in anderen Typen enthalten sein. Zum Beispiel ist a im Typ a+ enthalten. Wir sagen a ist Subtyp von b wenn

$\langle a \rangle \subseteq \langle b \rangle$

und notieren dann :

a<:b

Alle Werte, die im Typ rechts enthalten sind, sind auch links enthalten. Wir beschreiben nun ein Typ "Addressbook":

type Addrbook = addrbook[(Name, Addr, Tel?)]*

type Name = name[String]

type Addr = addr[String]

type Tel = tel[String]

Und ein Wert "MyBook" *:*

val mybook = addrbook[

name["Peter Meier"],
addr["Saarbrücken"],
name["Hans"],
addr["Bonn"],
tel["123-34-2312"],
name["Franz"],
addr["Berlin"]]

Bevor zugewiesen wird, überprüft XDuce, ob man den Wert mybook den Typ addrbook zuweisen kann. Das wird durch Subtyping überprüft. Wir erklären das anhand dem letzten Beispiel. Zuerst gilt, dass :

Name,Addr <: Name,Addr,Tel?
Name,Addr,Tel<:Name,Addr,Tel?

Das T,T <: T* gilt ist klar. Damit ist auch T,T,T<:T*. Wenn wir alles kombinieren bekommen wir:

(Name,Addr),(Name,Addr,Tel),(Name,Addr)<:(Name,Addr,Tel?)*

Konkatenation ist assoziativ, daher:

Name,Addr,Name,Addr,Tel,Name,Addr<:(Name,Addr,Tel?)*

$\Longleftrightarrow$ _addrbook[Name,Addr,Name,Addr,Tel,Name,Addr]<:addrbook[(Name,Addr,Tel?)*]_
$=^{def}$ _Addrbook._

Also kann man mybook dem Typ Addrbook zuweisen

2.5 Restriktionen auf Typen

Baumautomaten sind endliche Automaten die Bäume aus der Sprache akzeptieren können. Das mathematische Gerüst von Baumautomaten kann man unter [Common et al. 1999] nachlesen. Seien A und B die von zwei Automaten akzeptierten Sprachen. Das Problem, zu entscheiden, ob die Sprache A Untermenge der Sprache B ist, ist entscheidbar, im Gegensatz zu kontextfreien Sprachen.

Andererseits sind Baumautomaten immer noch sehr mächtig. Leider ist die Worst-Case-Komplexität von fundamentalen Baumautomaten-Operationen sehr teuer. Vor allem kostet es exponentiell viel Zeit bzgl. der Größe des Baumautomaten, nachzuprüfen, ob eine Sprache in einer anderen enthalten ist.

So wie wir die Typen in (2.2) definiert haben, handelt es sich immer noch um eine kontextfreie Grammatik. Aber das Testen ob $T_1 <: T_2$ ist auf kontextfreie Grammatiken nicht entscheidbar [UWE]. Daher musste man das Typsystem einschränken - so dass das Problem entscheidbar bleibt, ohne die Mächtigkeit zu sehr einzuschränken. Beim Entwurf von XDuce wurde also entschieden, nur rechtsrekursive Typen zu erlauben, damit nun das Typsystem gleichstark wie reguläre Baumautomaten ist.

$Type\ a = a\ ,\ t$

sind also nicht erlaubt.

Reguläre Ausdrücke kann man auch in Pattern benutzen. Wie Pattern bei XDuce funktionieren erklären wir im nächsten Punkt.

3 Pattern Matching

Das Pattern Matching sieht dem aus ML sehr ähnlich - ist aber durch reguläre Ausdrücke viel mächtiger.

3.1 Pattern

Hier das Beispiel eines Patterns (Funktionsaufrufe werden in der nächsten Arbeit definiert. Das eigentliche wichtige beim Beispiel ist das Pattern):

$$fun\ firstTriple : (Name, Addr, Tel?)^* \rightarrow (Name, Addr, Tel)?$$
$$= ps : (Name, Addr)^*, t : (Name, Addr, Tel), rest : (Name, Addr, Tel?)^*$$
$$\rightarrow t$$
$$\mid whole : (Name, Addr, Tel?)* \rightarrow ()$$

Die Funktion firstTriple überspringt alle Paare (Name,Addr)* bis sie das erste Tripel findet - dann wird es t zugewiesen. Findet die Funktion kein solches Tripel, gibt sie die leere Sequenz zurück.

3.2 Erschöpfende Pattern

Eine Liste $P_1, ..., P_n$ von Pattern heißt erschöpfend bezüglich T, wenn ∀ v, $v \in T$ impliziert $v \in P_i \Rightarrow V$ für P_i's und V. Man schreibt dann:

$T \triangleright p_1, ..., P_n$ *:exhaustive*

Eine liste von Pattern ist irredundant, bzgl. T, geschrieben

$T \triangleright p_1, ..., P_n$ *:irredundant*

genau dann wenn, $\forall P_i$ *es gibt ein Wert* $v \in T$*, so dass* $v \notin P_j$ *für* ∀ *1<j<i-1 und* $v \in P_i \Rightarrow V$ *fuer manche V*

Es wird mittels Subtyping überprüft, ob ein Pattern erschöpfend ist.
z.b. bei dieser Funktion :

$fun\ firstTriple : (Name, Addr, Tel?)^* \rightarrow (Name, Addr, Tel)?$
$= ps : (Name, Addr)^*, t : (Name, Addr, Tel), rest : (Name, Addr, Tel?)^*$
$\rightarrow t$
$\mid whole : (Name, Addr, Tel?)* \rightarrow ()$

Damit das Pattern erschöpfend ist muss gelten, dass

(Name,Addr,Tel?)* <: ((Name,Addr)*,(Name,Addr,Tel),(Name,Addr,Tel?)*) | (Name,Addr,Tel?)*

sowie

(Name, Addr,Tel)? <: t | ()

es ist aber nicht notwendig, dass

((Name,Addr),(Name,Addr,Tel),(Name,Addr,Tel?)*) | (Name,Addr,Tel?)**

<: (Name,Addr,Tel?)*

oder

t | ()<: (Name, Addr,Tel)?

gilt (aus bequemlichkeitsgründen).
Nun definieren wir die Syntax der Patterns

3.3 Zusammenfassung

XDuce ist eine funktionale typisierte Sprache um XML Dokumente zu verarbeiten. Sie ist Konform zu den "Basic Standards", z.B. Unicode, XML, DTD, Namespaces... Seine Schwerpunkte sind seine Strukturelle Typen mit regulären Ausdrücke, oder auch Pattern-Matching mit regulären Ausdrücken. Damit kann man schneller Programmieren.
Programme wie Diff, oder Html2Latex[DIFF] bestehen nicht mal aus 300 Zeilen - dabei könnte man nur mit sehr viel Aufwand so ein Programm in imperative Sprachen (z.B. C++) schreiben. Dazu verhindert das statische Typsystem viele Fehler.

4 Referenzen

[XDUCE]
Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. Journal of Functional Programming, 2002. To appear. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67-80, 2001.

[XMLambda]
E. Meijer and M. Shields. XMLambda: A functional language for constructing and manipulating XML documents. Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from http://www.cse.ogi.edu/ mbs/
1999. 11, http://citeseer.nj.nec.com/meijer00xmlambda.html