XML Programmierung mit XDuce

Daniel Beck - Proseminar im Sommersemester 2003 bei Prof. Gert Smolka

13th November 2003

Contents

Einführung

In diesem Dokument beschreiben wir eine statisch typisierte Sprache um XML-Dokumente zu manipulieren : XDuce. XDuce ist eine funktionale, domänenspezifische Programmiersprache. Sie wurde in Ocaml implementiert, und man kann den Quellcode frei herunterladen unter [SRC]. Die Hauptautoren sind Benjamin Pierce, Haruo Hosoya, und Peter Bruneman.

Die Werte der Sprache sind XML-Dokumente. Die Typen sind reguläre Ausdrücke Typen die den "Document Schema" entsprechen. Außerdem basiert das Typsystem direkt auf der regulären Baumautomaten-Theorie. Eine andere wichtige Besonderheit von XDuce ist sein flexibles reguläres Pattern-Matching.

Es existieren bereits Programme, welche in XDuce geschrieben wurden. Diff, z.B. zeigt die Unterschiede zwischen zwei Dateien - ob man Zeichen hinzugefügt hat, entfernt hat, oder verschoben hat. Html2Latex wandelt HTML Code zu Latex Code um. Beide Programme bestehen sogar aus weniger als 300 Zeilen! Deren Quellcode kann man frei heruntergeladen unter [WXDUCE].

Wir gehen nur kurz darauf ein, wie Werte und Typen in XDuce aussehen. Dann gehen wir auf das etwas komplexere Pattern-Matching ein. Wir betrachten dann noch wie Funktionen definiert sind, und die Restriktionen die man einführen musste, damit die Sprache entscheidbar bleibt. Anschließend stellen wir noch kurz andere ähnliche Projekte vor.

1 Werte und Typen

1.1 Werte

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

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

Die XDuce Implementation erlaubt zwei Möglichkeiten, um Werte zu deklarieren als XML, oder als "naiver XDuce Wert"- wie obendrüber. Nun zeigen wir noch wie Typen in XDuce aussehen.

1.2 Typen

Das Typsystem in XDuce ist sehr mächtig - mit reguläre Ausdrücke kann man die Typen vereinigen, konkatenieren, etc. Sie sehen der DTD sehr ähnlich sind aber viel Mächtiger.

1.2.1 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.
Typ-ausdrücke sind wie folgt definiert :

1.2.2 Syntax von Typen

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

wobei X eine abzählbare unendliche Menge von Typnamen ist.

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

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

1.2.3 Typ-Semantik

Die Semantik der Typen ist gegeben durch die Relation $v \in T$, lies : Wert v hat den Type T. Sie ist die kleinste Relation die unter den 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)*

1.2.4 Sub-Typen

Typen in XDuce sind Mengen von Werten. Dieses Typ-Konzept ist zwar ungewöhnlich, aber sehr praktisch. Man kann diese Menge dann zum Beispiel vereinigen. Man kann auch entscheiden, ob eine Menge in einer anderen enthalten ist. Man spricht dann von Sub-Typisierung, und schreibt
$a <: b$
falls der Typ a im Typ b enthalten ist.

1.2.5 Entscheidbarkeit von Typen

Baumautomaten sind endliche Automaten die Bäume annehmen. 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 (1.2.3) 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 . Daher musste man das Typsystem einschränken, so dass das Problem entscheidbar wird. Beim Entwurf von XDuce wurde also entschieden, nur rechts rekursive Typen zu erlauben, damit nun das Typsystem genauso stark wie reguläre Baumautomaten ist.

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

2 Pattern Matching

Das Pattern Matching sieht dem aus ML sehr ähnlich - ist aber durch reguläre Ausdrücke viel mächtiger. Wie Pattern eigentlich funktionnieren erklären wir anhand eines Beispiels

2.1 Ein Beispiel

Hier das Beispiel eines Patterns (Funktionsaufrufe werden später in "Funktionen und Terme" definiert):

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

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.

2.2 Erschöpfende Pattern

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

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

Eine liste von Pattern ist nicht redundant, bzgl. T, geschrieben

$T \triangleright p_1, ..., P_n$:*nicht redundant*

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

Mittels Sub-Typisierung wird 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 | ()

Nun definieren wir die Syntax der Pattern

2.3 Pattern-Ausdrücke

2.3.1 Syntax

Pattern-Ausdrücke sind wie folgt definiert :

$P ::= Y$	*Name des Pattern*
val x as P	*Variable Binder*
()	*Leere Sequenz*
P,P	*Konkatenation*
$L[T]$	*Label*
P/P	*Vereinigung*
P^*	*Wiederholung*

Wobei die Menge von Pattern-Namen abzählbar unendlich ist, so wie die Menge der Variablen. P ist entsprechend der Syntax-Regeln ein beliebiges Pattern, x eine beliebige Variable.

2.3.2 Semantik

$V \in P \Rightarrow V$, gelesen "v ist gematcht von P und produziert V", wobei V die endliche Umgebung ist, die Variablen Werten zuordnet (geschrieben $x_1 : v_1, ..., X_n : v_n$). Die Konkatenation von Umgebungen, die bestimmte Variablen bindet, wird mit einem Komma geschrieben.

$\frac{v \in P \Rightarrow V}{v \in (val\ x\ as\ P) \Rightarrow x:v,V}$ *(EP-As)*

$\frac{F(Y)=P\ v \in P \Rightarrow V}{v \in Y \Rightarrow V}$ *(EP-Var)*

$\frac{v \in P \Rightarrow V\ l \in L}{l[v] \in L[P] \Rightarrow V}$ *(EP-Lab)*

$\frac{v_1 \in P_1 \Rightarrow V_1\ \ v_2 \in P_2 \Rightarrow V_2}{v_1, v_2 \in P_1, P_2 \Rightarrow V_1, V_2}$ *(EP-Cat)*

$\frac{v \in P_1 \Rightarrow V}{v \in P_1 | P_2 \Rightarrow V}$ *(EP-Or1)*

$\frac{v \in P_2 \Rightarrow V}{v \in P_1 | P_2 \Rightarrow V}$ *(EP-Or2)*

$\frac{v_1 \in P \Rightarrow V_i\ \forall i}{v_1,...,v_n \in P^* \Rightarrow V_1,...,V_n}$ *(EP-Rep)*

2.4 Ambiguitäten

Bei Pattern können Ambiguitäten auftreten - das heißt, man musste Regeln festlegen, damit das Pattern eindeutig ist, und nicht mehrere Auswertungen möglich sind. Die erste Regel heißt

First Matching policy

Wenn ein Pattern diese Form hat :

$ps : (Name, Addr)* \rightarrow ps$
$|t : (Name, Addr) \rightarrow t$

dann matchen zwei Patterns dieselbe Eingabe. Die Regel besagt, dass man dann das erste passende Pattern nimmt. Die zweite Regel zum Pattern Matching heißt

Longest match rule.

In so einem Fall

$ps : (Name, Addr, Tel?)^*,$
$t : (Name, Addr, Tel),$
$rest : (Name, Addr, Tel?)^*$

könnte die Eingabe von verschiedenen Regeln erkannt werden. - welcher Tripel wird denn nun t zugewiesen ? Es gilt hier: jeder Pattern matcht so viel wie möglich. Hier : das erste auftretende Pattern ist ps:(Name,Addr,Tel?). - er hat nun Priorität, das besagt die Regel "First Matching policy". Und dann matcht dieser Pattern soviel er nur kann - und t wird das letzte Tripel in der Eingabe zugewiesen..Die Eindeutigkeit beim Pattern ist mittels der Parsing-Operation $v \in^{\mu} P$ definiert (man lese : P parst nur v). Die Parsings-Operation ist mit den folgenden Inferenzregeln definiert :

$\frac{v \in^{\mu} P}{v \in^{\mu} (val\ x\ as\ P)}$ *(EUP-As)*

$() \in^{\mu} ()$ *(EUP-Emp)*

$\frac{F(Y)=P\ v \in^{\mu} P}{v \in^{\mu} Y}$ *(EUP-Var)*

$\frac{v \in^{\mu} P\ l \in L}{l[v] \in^{\mu} L[P]}$ *(EUP-Lab)*

es gibt einzelne v_1, v_2 *s.t.* $v = v_1, v_2$

$\frac{v_1 \in^{\mu} P_1\ v_2 \in^{\mu} P_2}{v \in^{\mu} P_1, P_2}$ *(EUP-Cat)*

$\frac{v \in^{\mu} P_1\ v \notin tyof(P_2)}{v \in^{\mu} P_1 | P_2}$ *(EUP-Or1)*

$\frac{v \notin tyof(P_1)\ v \in^{\mu} P_2}{v \in^{\mu} P_1 | P_2}$ *(EUP-Or2)*

Es gibt einzelne $v_1, ..., v_n$ *, so dass* $v = v_1, ..., v_n$

$\frac{v_i \in^{\mu} P\ for\ each\ i}{v \in^{\mu} P^*}$ *(EUP-Rep)*

Diese Regeln sehen den Inferenz-Regeln für Pattern Matching (also die Regeln deren Namen mit EP anfangen) sehr ähnlich, die Unterschiede liegen eigentlich nur darin, dass EUP-Cat und EUP-Rep sicherstellen dass die Eingabe-Sequenz nur bei Konkatenation und Wiederholung-Pattern geteilt werden kann.

3 Funktionen und Termen

Ein Programm in XDuce besteht aus einer Menge von Typ-Definitionen, einer Menge von Pattern Definition, einer Menge von Funktions-Definitionen, und einem Term, mit dem die Auswertung beginnt. Type und Pattern haben wir schon beschrieben. Hier führen wir Funktionen und Terme ein.

3.1 Terme - Syntax

Es wird vorausgesetzt, dass wir eine abzählbar unendliche Menge von Funktionnamen haben. Die Definition von Funktionen wird durch eine globale rekursive Menge dieser Form gegeben

fun f (P):T=e

(Wir stellen hier nur Funktionen mit einem Argument vor). Dazu sollte man auch noch in Betracht ziehen, dass man den Typ des Rückgabe-wert explizit angeben muss.

Die Syntax von Termen, e, ist durch die folgende Grammatik definiert :

e::=x	*Variable*
l[e]	*Label*
()	*Leere Sequenz*
e,e	*Konkatenation*
f(e)	*Applikation*

match e with $\bar{P} \rightarrow \bar{e}$ *Pattern-Matching*

Dabei ist $\bar{P} \rightarrow \bar{e}$ eine Abkürzung für die n-fache Form

$P_1 \rightarrow e_1 | \ldots | P_n \rightarrow e_n$

Auch erlaubt sind diese Abkürzungen :

let P=e_1 *in* e_2≡*match* e_1 *with* P→ e_2

$\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \equiv \mathit{match}\ e_1\ \mathit{with}\ \mathit{True}[\ ] \rightarrow e_2 / \mathit{False}[\ ] \rightarrow e_3$

$e_1; e_2 \equiv \mathit{let}\ \mathit{Any} = e_1\ \mathit{in}\ e_2$

nun noch wie man Typ-Regeln davon ableitet :

3.2 Terme - Typ-Regeln

Die Typrelation $\Gamma \vdash e \in T$, (lies "e" hat den Typ T unter der Umgebung Γ) , ist durch folgende Regeln definiert :

$\frac{\Gamma(x)=T}{\Gamma \vdash x \in T}$ *(TE-Var)*

$\Gamma \vdash () \in ()$ *(TE-Emp)*

$\frac{\Gamma \vdash e \in T}{\Gamma \vdash l[e] \in l[T]}$ *(TE-Lab)*

$\frac{\Gamma \vdash e_1 \in T_1 \ \Gamma \vdash e_2 \in T_2}{\Gamma \vdash e_1, e_2 \in T_1, T_2}$ *(TE-Cat)*

$\frac{\mathit{fun}\ f(P)\text{"}T = e_2 \in G\ \Gamma \vdash e_1 \in U\ U <: \mathit{tyof}(P)}{\Gamma \vdash f(e_1) \in T}$ *(TE-App)*

$$\forall i. \left(\begin{array}{c} \Gamma \vdash e \in R \\ R \triangleright P_1, ..., P_n : \mathit{erschpfend} \\ R \triangleright P_1, ..., P_n : \mathit{nichtredundant} \\ R \backslash (\mathit{tyof}(P_1) | ... | \mathit{tyof}(P_{i-1})) \Rightarrow S \\ S \triangleright P_i : \mathit{nichtambiguieus} \\ S \triangleright P_i : \Gamma_i \\ \Gamma, \Gamma_i \vdash e_i \in T_i \end{array} \right) \over \Gamma \vdash e\ \mathit{with}\ P \rightarrow \bar{e} \in T_1 | ... | T_n$$ *(TE-Match)*

Dann gibt es noch eine einzige Regel um zu überprüfen wann die Definition der Funktion f typ-korrekt ist, geschrieben $\vdash \mathit{fun}\, f(P) : T = e$

$$\frac{\mathit{tyof}(P) \triangleright P \Rightarrow \Gamma \quad \Gamma \vdash e \in S \ \ S <: T}{\vdash \mathit{fun}\, f(P) : T = e}$$

3.3 Terme - Auswertung-regeln

wie die Termen ausgewertet werden wird wie folgt beschrieben. Dabei heißt $V \vdash e \Downarrow v$ "e wertet unter der Umgebung V zu v aus".

$V \vdash x \Downarrow V(x)$ *(EE-Var)*

$$V \vdash () \Downarrow () \textit{(EE-Emp)}$$

$$\frac{V \vdash e \Downarrow v}{V \vdash l[e] \Downarrow l[v]} \textit{(EE-Lab)}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \;\; V \vdash e_2 \downarrow v_2}{V \vdash e_1, e_2 \Downarrow v_1, v_2} \textit{(EE-Cat)}$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow v \\ fun\, f(P) : T = e_2 \in G \\ v \in P \Rightarrow W \;\; W \vdash e_2 \Downarrow w \end{array}}{V \vdash f(e_1) \Downarrow w} \textit{(EE-App)}$$

$$\frac{\begin{array}{c} V \vdash e \Downarrow v \\ v \notin P_1 ... v \notin P_{i-1} \;\; v \in P_i \Rightarrow W \\ V, W \vdash e_i \Downarrow w \end{array}}{V \vdash match\, e\, with\, \bar{P} \rightarrow e \Downarrow w} \textit{(EE-Match)}$$

4 Restriktionen und Probleme

An XDuce wird noch weiter entwickelt. Es fehlen immer noch sehr wichtige Features, die moderne Sprachen haben sollten : Funktionen höherer Ordnung (also Funktionen, welche wiederum Funktionen als Argument haben können), Polymorphismus, parametrisierte Typen (Typen die Typen als Argumente haben können, wie unter ML). Man kann auch Typen nur unter Vereinigung,nicht unter schnitt oder Komplement bilden - Dies wirft nämlich eine Menge nicht trivialer algorithmische Probleme auf.

Außerdem, wegen das Typsystem-Designs, das gleich-mächtig sein sollte zur Baumautomaten-Sprache, kann man auch nur Operationen verwenden, die man auf solchen Automaten verwendet - und die sind leider ziemlich teuer. Prüfen zum Beispiel, ob ein Typ Sub-typ von einem anderen Typ ist, kostet exponentiell viel Zeit. Das kommt daher, dass man den Komplement bilden muss, und das Komplement eines Baumautomaten zu bilden kostet exponentiell viel Zeit, was man unter unter [Niessner] nachlesen kann. Das ist der Preis für so ein mächtiges Typsystem. An der Effizienz wird auch noch weitergearbeitet. Es gibt auch andere ähnliche Projekte, die mehr oder weniger fortgeschritten sind, und manche der Probleme, die XDuce hat nicht haben.Wir stellen hier einige solcher Sprachen vor und geben an, in-wie-fern sie XDuce ähnlich sind, und was dabei anders ist

5 Zusammenfassung und ähnliche Arbeiten

5.1 $XM\lambda$

$XM\lambda$ wurde von Meijier und Shiled entwickelt um XML Dokumente zu verarbeiten[Xmlambda]. Die Sprache ist flexibler als die von XDuce, dank "type-indexed rows"[Xmllambdap]. Labels werden nicht beachtet, nur der Typ der

Elemente selbst wird verwendet. Das Kalkül "TIR" (der "Kernel" von $XM\lambda$) erlaubt das Darstellen von Tupeln, rekursiven Datentypen, Funktionen höherer Ordnung und parametrisierten Typen. Man kann zum Beispiel in $XM\lambda$ eine polymorphe Funktion schreiben :

$$\forall X \notin \{T, U\}.(T \mid X) \rightarrow (U \mid X).$$

Diese Funktion ändert bestimmte Labels, und lässt die anderen unverändert. Durch "Type-indexed row" wird festgelegt, dass der Typ, der in der Eingabe mit T vereinigt wird, genau der selbe ist, wie der, der mit U in der Ausgabe vereinigt wird. Das kann man mit Sub-Typisierung nicht.

Was die Sprache leider nicht unterstützt, ist das Validieren von wichtigen Regeln wie Assoziativität oder Distributivität bei Typen.

5.2 YAT

YAT ist eine Sprache um Datenbanken abzufragen und zu manipulieren, welche von Cluet und Simeon entwickelt wurde. Das Typsystem ist dem aus XDuce sehr nahe. Wie in XDuce benutzt man auch Sub-Typisierung - nur ist die Sub-Typisierung dort strenger, zum Beispiel erlaubt $a[T\,U]$nicht, ein Sub-Typ von $a[T] \mid a[U]$ zu sein. Doch $a[T\,U]$ ist ein Sub-Typ von $a[T] \mid a[U]$. Diese Entscheidung wurde zugunsten eines effizienten Layouts bei XML Datenbanken getroffen. Erlaubt sind aber sind Durchschnitte zwischen Typen zu bilden.

5.3 CDuce

CDuce ist ein Fork von XDuce, entwickelt von der "Group de Languages ENS, Paris", der Database Group über LRI in Orsay, und zwei CNRS Laboratorien[Cduce]. Es wurden überladene Funktionen und Funktionen erster Klasse hinzugefügt. Der Sub-Typisierung Algorithmus funktioniert ohne Backtracking. Das Pattern-Matching wurde ebenfalls erweitert, so dass man nicht aufeinander folgende Unter-Sequenzen auffangen kann. Außerdem ist es auch erlaubt das Komplement und den Schnitt von Typen zu bilden. CDuce soll auch schneller als XDuce sein - vor allem Dank des neuen Sub-Typisierungs Algorithmus. Unter [Bench] gibt es Benchmarks die XDuce, CDuce, und XSLT vergleichen. Außerdem ist CDuce aktiver als XDuce, man kann immer das allerletzte Feature mittels CVS ausprobieren

5.4 Xtatic

Die Autoren von XDuce arbeiten nun auch noch an Xtatic, einen Nachfolger von XDuce, der alle "Fehler" von XDuce beheben soll. Es ist eigentlich eine X# Erweiterung, und daher keine Programmiersprache.

5.5 Zusammenfassung

XDuce ist eine funktionale typisierte Sprache um XML Dokumente zu verarbeiten. Sie ist konform zu Basisch Standard, z.B. Unicode, XML, DTD, Namespaces... Seine Schwerpunkte sind seine Strukturellen Typen mit regulären Ausdrücken, oder auch Pattern-Matching mir regulären Ausdrücken Damit kann man schneller und effizienter Programmieren. Das statische Typsystem verhindert auch viele Fehler.

6 Referenzen

[X]
Homepage von XDuce Homepage. Hauptautoren: Benjamin Pierce, Haruo Hosoya, und Peter Bruneman. http://xduce.sourceforge.net/

[Xmlambda]
Homepage von XMLambda. Entwickelt von Meijier und Shiled. Homepage : www.cse.ogi.edu/~mbs/pub/xmlambda/

[Xduce]
Haruo Hosoya and Benjamin C. Pierce : Regular expression pattern matching for XML. A typed XML processing language preliminary report. Unter http://xduce.sourceforge.net/papers.htm

[Common et al. 1999]
Common, H. Dauchet, M. Gilleron, R. Jacquemard, F. Lugiez, D. Tison, S. and Tommasi, M. 1999. Tree automata techniques and applications. Draft book; online verfügbar unter www.grappa.univ-lille3.fr/tata

[Xmlambdap]
Erik Meijer and Mark Shields. Draft, Dec 1999. online verfügbar unter http://www.cse.ogi.edu/mbs/pub/xmlambda/xmlambda.ps

[Cduce]
entwickelt von der "Group de Languages ENS".
Homepage : http://www.cduce.org/

[Yat]
Cluet and Simeon J. 1998 : using YAT to build a webserver in Intl Workshop on the web database

[Xmltechnologie]
Philip Wadler,Avaya Labs, Programming language technologie for XML. 1999 http://www.research.avayalabs.com/user/wadler/xml/

[Diff]
Quellcode von diff unter : http://sourceforge.net/projects/xduce/

[Niessner]
http://ozelot.psc.informatik.uni-frankfurt.de:8000/english/staff/niessner.html