Optimierung der Verarbeitung von Dokumenten

Markus L. Noga
Booz Allen Hamilton GmbH*
Grüneburgweg 102, 60322 Frankfurt
ml@noga.de

1 Einleitung

Seit der Einführung von XML als universellem Format für Dokumente und Webdienste haben die funktionalen und softwaretechnischen Merkmale dieses Standards zu seiner breiten Anwendung geführt. Eine tiefere Durchdringung des Marktes behindern in erster Linie nichtfunktionale Eigenschaften:

> *As XML becomes ubiquitous throughout the enterprise, it increasingly taxes the systems that must deal with it. [...] Even though it is inefficient, however, XML's numerous advantages are increasing its use for ever broader and more mission-critical functions. [...] XML's processing overhead, storage requirements, and bandwidth consumption become quite problematic when transaction volumes are high.* [Sc02]

Obwohl z.B. Protokolle für XML-basierte Webdienste wie SOAP beliebige Komponenten miteinander verbinden könnten, werden sie zumeist für schwach gekoppelte Systeme aus wenigen, großen Komponenten eingesetzt. Den Anforderungen eng gekoppelter Systeme aus zahlreichen kleinen Komponenten sind vorhandene Umsetzungen nicht gewachsen.

Diese Arbeit betont nicht Komponenten, sondern die von ihnen ausgetauschten Dokumente. Die betrachteten *dokumentenverarbeitende Systeme* erzeugen in *Quellen* Dokumente, formen diese ein- oder mehrfach um und legen die Ergebnisse in *Senken* ab. Unser Ziel ist die Beschleunigung solcher Systeme.

Da auf Quellen und *Transformationen* der überwiegende Teil der Laufzeit entfällt, entwickeln wir für beide jeweils zwei Verfahren (siehe Abb. 1 auf der nächsten Seite).

Statische Vorberechnung mit *Vorbedingungen* ersetzt *eingabeunabhängige* Berechnungen durch ihren stets gleichen Wert. Sie macht Gebrauch von *Vorbedingungen* an die Eingabe, also *Eigenschaften*, die für alle Eingaben gelten. So können wir auch Ausdrücke statisch vorberechnen, die von der Struktur der Eingabe, nicht aber ihrer konkreten Belegung abhängen (siehe Abb. 2 auf der nächsten Seite).

*Die im m+v Verlag erschienene gleichnamige Dissertation verfaßte der Autor als wissenschaftlicher Mitarbeiter am Lehrstuhl Goos des IPD der Universität Karlsruhe (TH).

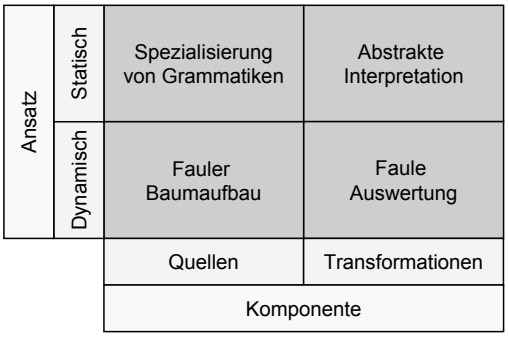

Abbildung 1: Lösungsansatz im Überblick

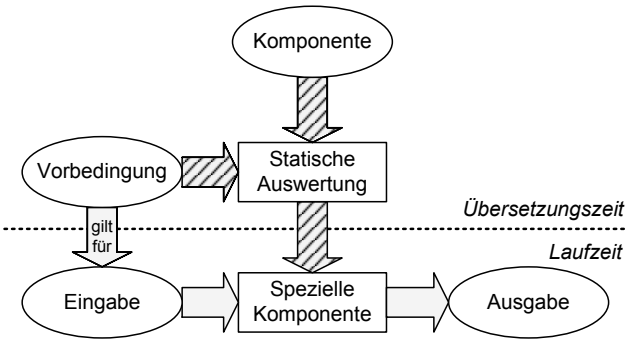

Abbildung 2: Statische Auswertung mit Vorbedingungen

Sind Ausdrücke für alle Eingaben unabhängig voneinander, so können sie zur Laufzeit *in beliebiger Reihenfolge* ausgewertet werden, bei Bedarf auch *parallel*. Werden Transformationen verkettet, so bleiben oft auch Zwischenergebnisse *ungenutzt* (siehe Abb. 3). Ihre Berechnung könnte unterbleiben.

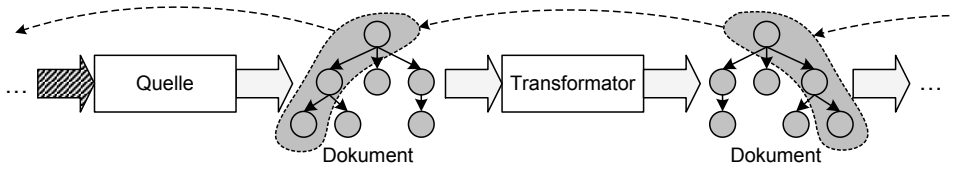

Abbildung 3: Genutzte Zwischenergebnisse (hinterlegt)

Die aus der funktionalen Programmierung bekannte *faule Auswertung* wertet unabhängige Ausdrücke beim ersten Zugriff darauf aus. Wie in der modernen *just-in-time* Fertigung werden Zwischenergebnisse also nicht auf Verdacht im Voraus berechnet und gelagert, sondern genau dann erzeugt und bereitgestellt, wenn sie auch benötigt werden. Diese *lean production* ist frei von Verschnitt, da ungenutzte Zwischenergebnisse auch nie berechnet werden.

Wie optimierende Übersetzer und Laufzeitsysteme im Allgemeinen sind auch unsere Techniken ein Bindeglied zwischen dem Entwickler und der unterliegenden Maschine. Sie weisen daher zwei Gesichter auf.

Zum einen verbessern unsere Techniken unmittelbar die Laufzeiten der übersetzten Programme. Diesen *nichtfunktionalen Vorteil* können wir bereits heute nachweisen. Von ihm können z.B. Webdienste, für die eine WSDL-Beschreibung vorliegt, Editoren und Betrachter für Formate wie DocBook und SMIL sowie Content Management Systeme unmittelbar profitieren.

Zum anderen erlauben unsere Optimierungen es zukünftigen Entwicklern, ihre Programme allgemeiner und abstrakter als bisher zu formulieren, ohne daß entsprechende Strafen in Form erhöhter Laufzeit fällig werden. Historisch gesehen *steigerte* eine solche Befreiung von konkreten Details meist auch die *Produktivität* der Entwickler. In unserem Fall steht ein Nachweis solcher Produktivitätssteigerungen noch aus; dies ist ein Thema für zukünftige Arbeiten.

2 Modell

2.1 Dokumente, Typen und Schemata

In diesem Beitrag verkürzen wir die exakten Begriffe Dokument, Typ und Schema auf folgende Kurzformen:

Definition 1 (Dokument). *Dokumente sind attributierte, geordnete, unbeschränkte Bäume.*

Definition 2 (Typ). *Typen sind Prädikate über einem Knoten eines solchen Baumes. Sie enthalten eine reguläre Sprache, der die Kindern des gegebenen Knotens genügen müssen.*

Diese Sprache heißt auch Kindersprache. Ein Knoten heißt gültig unter einem Typ, wenn das Prädikat auf dem Knoten erfüllt ist.

Definition 3 (Schema). *Schemata bestehen aus Mengen von Typen sowie deterministischen Regeln zur rekursiven Anwendung von Typen. Ein Typ aus der Menge ist gesondert als Typ der Wurzel des Dokuments ausgezeichnet.*

Ein Dokument heißt gültig unter einem Schema, wenn alle Prädikate und Regeln auf seinen jeweiligen Knoten erfüllt sind. In diesem Fall weist das Schema jedem Knoten genau einen Typ zu.

2.2 Kontexte

Wie können wir zu statischen Aussagen die Menge aller unter einem Schema gültigen Dokumente gelangen? Dazu abstrahieren wir konkrete Knoten zu *Kontexten*, d.h., zur Folge der Namen entlang des Pfades von der Wurzel bis zum betrachteten Knoten (siehe Abb. 4).

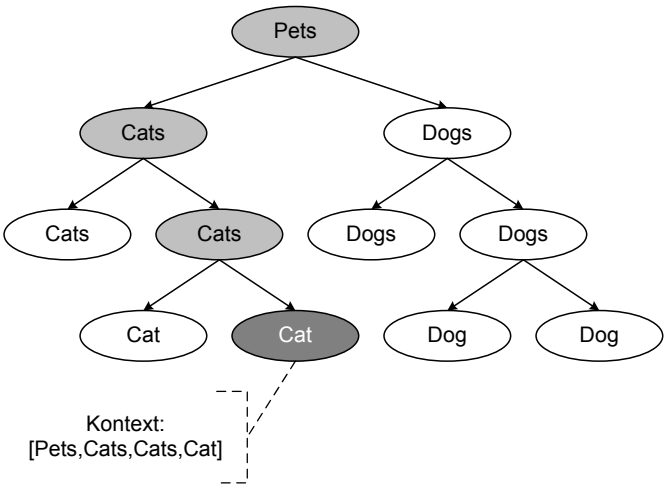

Abbildung 4: Knoten und Kontexte

Ein Kontext bestimmt den Typ der zugehörigen Knoten eindeutig. Gleichzeitig unterliegen Kontexte durch die Kindersprachen der Typen der beteiligten Väter grammatischen Einschränkungen. Daher können wir für jeden Typ eines Schemas die Kontexte aller Knoten dieses Typs angeben.

Definition 4 (Kontext). *Sei N ein Knoten, dann ist sein Kontext $\mathcal{C}(\,N\,) \in \mathcal{B}^*$ gegeben durch die Folge der Namen entlang des Pfades von der Wurzel zu N.*

Satz 1 (Kontext). *Die Abbildungen $\mathcal{C}(\,\cdot\,)$ sind nicht injektiv.*

Für beliebige Knotenmengen M, M' gilt:

$$\begin{array}{ccccccc} & & \mathcal{C}(\,M\,) \cup \mathcal{C}(\,M'\,) & = & \mathcal{C}(\,M \cup M'\,) & & \\ & & \mathcal{C}(\,M\,) \cap \mathcal{C}(\,M'\,) & \supseteq & \mathcal{C}(\,M \cap M'\,) & \supseteq & \emptyset \\ \emptyset & \subseteq & \mathcal{C}(\,M\,) \setminus \mathcal{C}(\,M'\,) & \subseteq & \mathcal{C}(\,M \setminus M'\,) & & \end{array}$$

Es gibt Knotenmengen M, M', für welche Gleichheit auftritt.

Stellen wir zusätzliche Anforderungen an die Mengen M, so wird $\mathcal{C}(\,M\,)$ zu einer injektiven Abbildung.

Definition 5 (Kontextvollständigkeit). *Sei S ein Schema und M eine Knotenmenge. M heißt kontextvollständig für S, wenn für alle Knoten $N \in^* D \in \mathcal{D}(S)$ der unter S gültigen Dokumente D gilt:*

$$\mathcal{C}(\,N\,) \in \mathcal{C}(\,M\,) \Rightarrow N \in M$$

Satz 2 (Kontextvollständigkeit). *Sei S ein Schema und M_1, M_2 Knotenmengen. Ist M_2 kontextvollständig für S, so gilt:*

$$\mathcal{C}(\,M_1\,) \subseteq \mathcal{C}(\,M_2\,) \Rightarrow M_1 \subseteq M_2$$

Auf Kontexten können wir durch Induktion über ihren letzten Buchstaben einen Typbegriff definieren, der mit dem Typbegriff auf Knoten kompatibel ist.

Definition 6 (Kontexttyp). *Sei $S = \{T_0, \ldots T_n\}$ ein Schema und $C \in \mathcal{B}^*$ ein Kontext. Der Kontexttyp $\tau(C) \in S$ ist gegeben durch:*

$$\tau(C) = \begin{cases} T_0 & |C| = 1 \\ \tau(\text{init } C).T_C(\text{last } C) & \textit{sonst} \end{cases}$$

Satz 3 (Kontexttyp). *Sei $S = \{T_0, \ldots T_n\}$ ein Schema, $D \in \mathcal{D}(S)$ ein Dokument und $N \in^* D$ ein Knoten. Dann gilt $\tau(N) = \tau(\mathcal{C}(N))$.*

Betrachten wir das umgekehrte Problem. Gegeben sei ein Typ. Ist es möglich, den *Typkontext* anzugeben, d.h., die Kontexte aller Knoten anzugeben, welche diesem Typ genügen? In nichtrekursiven Schemata hat jeder Typkontext endlich viele Einträge. Diese können wir aufzählen. Liegen hingegen rekursive Typen vor, so ist die Menge aller Knoten dieses Typs unendlich. Dennoch bleibt ihr Typkontext traktabel.

Satz 4 (Typkontext). *Sei S ein Schema und $T \in S$ ein Typ. Sei $M = \{N \in^* D \in \mathcal{D}(S)\}$ die Menge aller Knoten in unter S gültigen Dokumenten D. Dann ist M kontextvollständig für S und der Typkontext $\mathcal{C}(M)$ ist regulär. Der Schemakontext $\mathcal{C}(S) = \bigcup_{T \in S} \mathcal{C}(T)$ ist daher ebenfalls regulär.*

Da Typkontexte die Folgen der Vorfahren eines Knotens verkörpern, liegt es nahe, auch die Folgen seiner Nachkommen zu betrachten. Existiert eine analoge Darstellung als reguläre Sprache? Nein. Betrachte dazu folgendes Gegenbeispiel: Seien die möglichen Kinder von a-Knoten durch die Sprache $L_C = (ab|\epsilon)$ gegeben, und Knoten b stets kinderlos, so bilden die Namen der Nachkommen eines a-Knotens die nichtreguläre Sprache $a^n b^n$.

In Form regulärer Sprachen erlauben Kontexte also statische Aussagen über die Domäne eines Typs. Diese Aussagen decken alle Dokumente eines Schemas ab und sind im Gegensatz zu anderen Arbeiten auch für rekursive Schemata exakt.

2.3 Pfadsprachen

Unser Modell für Pfadausdrücke beruht auf Mengen von Bezügen auf Knoten. So können wir in Form von Kontexten statische Aussagen über die dynamisch im Programmverlauf auftretenden Zwischenergebnisse und ihre Typen machen. Die technischen Standards wie XPath setzen anstelle von Mengen auf Folgen solcher Bezüge auf. Diese Komplikation erweist sich als rein künstlich — unser Modell kann alle nichtreflexiven Ausdrücke in XPath wiedergeben.

Navigationen projizieren die eingehende Knotenmenge entlang der Struktur des Dokuments auf eine neue Knotenmenge. Filterungen entfernen Teile der eingehenden Knotenmenge. Verkettungen solcher Operationen und Mengenoperationen auf ihnen sind ebenfalls zulässig.

2.4 Transformationen

Unser Modell gibt — abgesehen von zwei bewußt ausgeschlossenen Sonderfällen, reflektiven Aufrufen und Kopien untypisierter Eingaben — alle Strukturen, Zustände und Anweisungen des technischen Standards XSLT wieder. Wie im Modell für Pfadausdrücke formulieren wir Zustand und Zwischenergebnissen als Mengen. Dadurch können wir statische Aussagen über die Eigenschaften von Werten in Form von Kontexten machen und so auf die beteiligten Typen schließen.

Die Rückführung aufwendiger Anweisungen auf einfachere Bestandteile fördert dabei Parallelen zu objektorientierten Programmiersprachen zu Tage. Die dort wohlbekannte Rückführung polymorpher auf monomorphe Aufrufe wird so auch für Transformationen anwendbar. Wiederum liefern Kontexte die für die Analysen nötigen Typinformationen.

3 Optimierungen

3.1 Spezielle Zerteiler und statische Typprüfung

Abbildung 5 zeigt eine Quelle mit einem allgemeinen Zerteiler und dynamischer Typprüfung. Welche Beschleunigungen sind unter Kenntnis des Schemas $\mathcal{S}$ möglich?

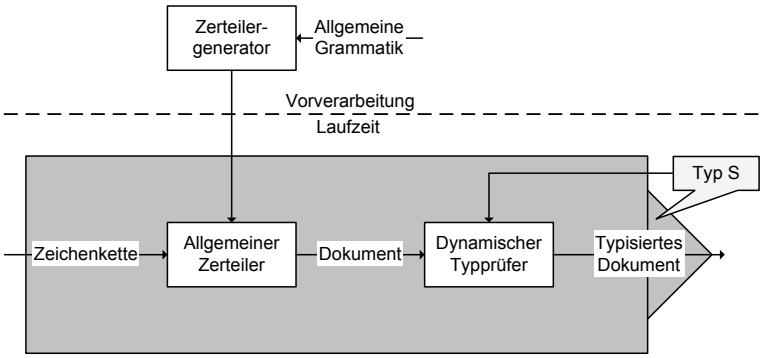

Abbildung 5: Ein allgemeiner Zerteiler mit dynamischer Typprüfung

Dazu spezialisieren wir die allgemeine Grammatik für wohlgeformte XML-Dokumente zu einer speziellen Grammatik $G(\mathcal{S})$ für Dokumente des gegebenen Typs. Aus $G(\mathcal{S})$ erzeugte spezielle Zerteiler akzeptieren ausschließlich Zeichenketten, die unter $G(\mathcal{S})$ gültige Dokumente darstellen. Eine dynamische Typprüfung der aufgebauten Dokumente kann entfallen. Durch Spezialisierung der Automaten arbeiten spezielle Zerteiler außerdem schneller als allgemeine. Abbildung 6 auf der nächsten Seite zeigt die neue Lage.

Zerteilergeneratoren funktionieren allerdings nur, wenn die Grammatik einer zerteilungsfreundlichen Sprachklasse angehört. Die meisten Werkzeuge erwarten heute $\mathrm{SLL}(k)$- oder $\mathrm{LALR}(k)$-Grammatiken [WG85, ASU86].

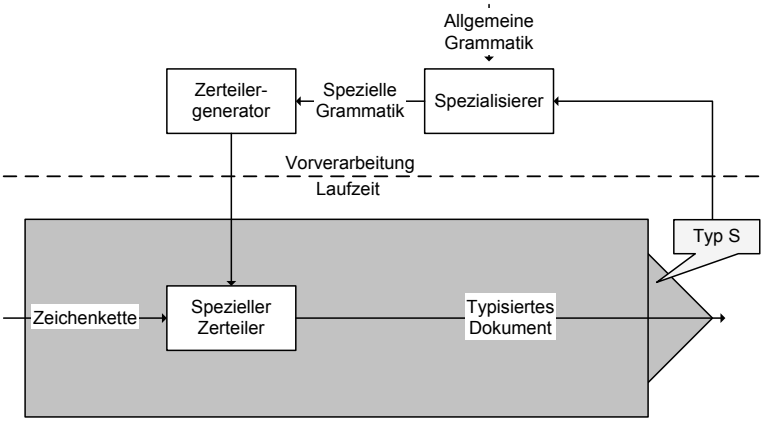

Abbildung 6: Ein spezieller Zerteiler mit inhärenter Typprüfung

Wir konstruieren zerteilbare Grammatiken für die Attributlisten sowie die Folge der Kindernamen eines Typs T. Mit diesen Bestandteilen spezialisieren wir dann die allgemeine Grammatik für XML-Dokumente für ein Schema $\mathcal{S}$ zu einer kontextfreien Grammatik $G(\mathcal{S})$ und weisen deren Zerteilbarkeit nach. Zerteiler dieser Bauart arbeiten ca. 40% schneller als herkömmliche.

Satz 5 ($\mathrm{SLL}(1)$ **und** $\mathrm{SLR}(1)$ **für Schemata).** *Für jedes Schema* $\mathcal{S} = \{T_0, \ldots T_n\}$ *kann in endlicher Zeit* $G(\mathcal{S}$ *mit* $G(\mathcal{S}) \in \mathrm{SLL}(1) \cap \mathrm{SLR}(1)$ *konstruiert werden.*

3.2 Abstrakte Interpretation von Pfadausdrücken

Wir analysieren Pfadausdrücke unseres Modells mit Hilfe abstrakter Interpretation nach [NNH99, Kap. 4]. Dazu formulieren wir Aussagen über die Eigenschaften von Knotenmengen zunächst als vollständigen Verband L_n über der Potenzmenge $2^{\mathcal{N}(S)}$ der Menge aller möglichen Knotenmengen.

Zur traktablen Berechnung vereinbaren wir einen weiteren Verband L_r über Paaren $(\underline{P}, \overline{P})$ regulärer Sprachen. Eine Galoisinjektion $(L_n, \alpha, \gamma, L_r)$ stellt anhand von Kontexten die Verbindung zwischen beiden Formen von Aussagen her. Auf L_r können wir genaue Aussagen effizient berechnen.

Abb. 7 auf der nächsten Seite verdeutlicht das Zusammenspiel von Semantik, Kontexten, Eigenschaften und Auswertern in Vorwärts- und Rückwärtsrichtung $\mathcal{V}[\![\ ]\!]$ bzw. $\mathcal{R}[\![\ ]\!]$. Man beachte, daß die vom Rückwärtsauswerter erstellten Mengen P' in keiner definierten Inklusionsbeziehung zum Kontext $\mathcal{C}(\ M\ )$ stehen. Wenn wir aus ihnen echte Einschränkungen des Suchbaums ableiten wollen, muß dies auch der Fall sein.

Auf Basis dieser Analysen können wir in Pfadausdrücken zahlreiche aus Übersetzern und Datenbanken bekannte Optimierungen anwenden, z.B. Faltung von Konstanten, Entfernung toten Codes sowie Minimierung des Suchbaumes durch Vorziehen von Filterungen.

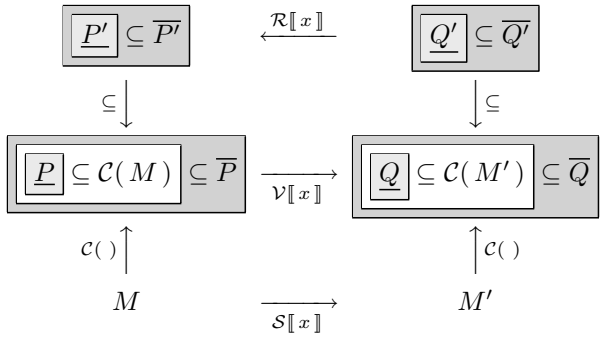

Abbildung 7: Auswertung von Mengenausdrücken

3.3 Datenfluß über Transformationen

Transformationen können rekursive Aufrufketten enthalten. Daher genügt die einfache abstrakte Interpretation aus dem vorigen Abschnitt nicht zu ihrer Behandlung. Wir setzen daher zusätzlich monotone Datenflußrahmen gemäß [NNH99, Abschnitt 2.1 und 2.4] ein. Die zur Lösung von Datenflußproblemen üblichen Fixpunktiterationen setzen vollständige Verbände voraus, um eine Terminierung zu garantieren. Da abzählbare Vereinigungen regulärer Sprachen nicht stets regulär sind, ist der Verband der Kontexte leider unvollständig.

Daher bilden wir den Verband der Kontexte mit einer Galoisinjektion in einen endlichen Verband von Typen ab. Da diese Galoisinjektion mit starkem Informationsverlust einhergeht, führen wir die eigentliche Fixpunktiteration weiterhin auf den ursprünglichen Kontexten durch und prüfen auf dem typbasierten Verband nur das Erreichen des Fixpunktes. Siehe dazu [NNH99, Abschnitt 4.5].

Mit Hilfe dieser Analysen können wir polymorphe Aufrufe auf monomorphe Aufrufe zurückführen und diese offen einbauen, sowie wiederum Konstanten falten und toten Code entfernen. Abb. 8 auf der nächsten Seite zeigt die dadurch mögliche Beschleunigung des Benchmarks XSLTMark auf unterschiedlichen Plattformen.

3.4 Faule Auswertung

Oft nutzen spätere Verarbeitungsschritte nur Teile eines Dokumentenbaums. Nicht betretenen Teile des Baumes sind dann ungenutzte Zwischenergebnisse. Folgenden Satz macht die aus der funktionalen Programmierung bekannte Strategie der faulen Auswertung, die alle solche Zwischenergebnisse vermeidet, zur Beschleunigung des Baumaufbaus nutzbar:

Satz 6 (Fauler Baumaufbau). *Die typspezifischen Grammatiken nach Satz 5 und die allgemeine Grammatik für wohlgeformtes XML sind konfluent und weisen alternative Auswertungsreihenfolgen auf.*

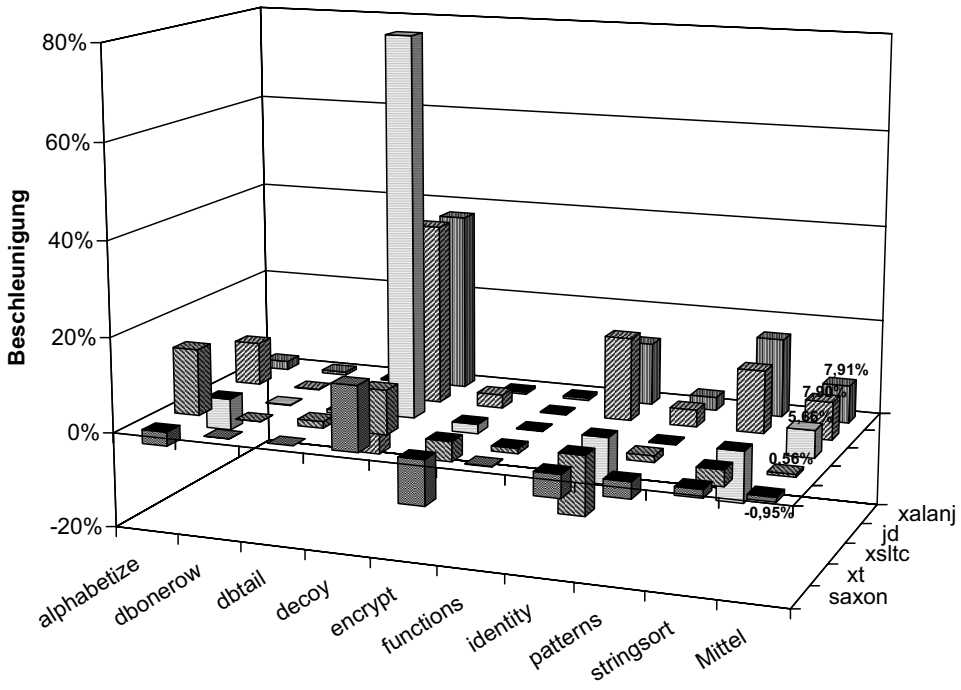

Abbildung 8: Beschleunigung des Benchmarks XSLTMark

Der an die Zerteilung angeschlossene faule Baumaufbau liefert dabei in Kinderfolgen zunächst vorläufige Knoten zurück, welche die zugehörige, partiell reduzierte Teilsymbolfolge enthalten. Beim ersten Zugriff auf einen vorläufigen Knoten wird dieser als echter Knoten materialisiert, wobei seine Kinderfolge wiederum aus vorläufigen Knoten besteht.

Wie der Baumaufbau können auch Transformationen faul ausgewertet werden. So können sich teilweise genutzte Zwischenergebnisse durch die Verarbeitungskette fortpflanzen (siehe Abb. 9). Die dabei erzielte Beschleunigung ist linear in der Abdeckung der Zwischenergebnisse.

Satz 7 (Faule Transformation). *Die Transformationen des Modells sind konfluent und erlauben alternative Auswertungsreihenfolgen.*

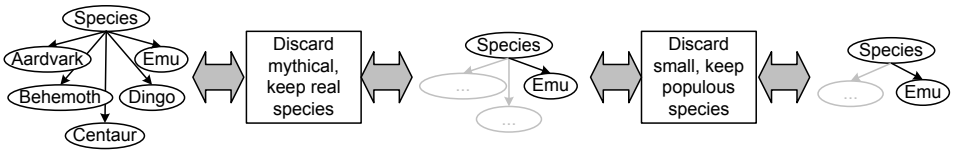

Abbildung 9: Verkettete faule Transformationen

4 Zusammenfassung

Wir haben Verfahren zur statischen und dynamischen Optimierung von Quellen und Transformationen entwickelt. Einige davon erforderten neue Umsetzungen dieser Grundbausteine, andere arbeiten mit bestehenden Umsetzungen zusammen.

Durch Spezialisierung der Grammatik für wohlgeformtes XML anhand eines statisch bekannten Schemas gelingt es uns, die bisher separat erfolgenden Prüfungen auf korrekte Klammerung und Typkonformität in der Zerteilung zu subsumieren. Nach unserem Verfahren erzeugte Grammatiken sind mit verbreiteten Zereilergeneratoren kompatibel.

Abstrakte Interpretation von Pfadausdrücken über einem statisch bekannten Eingabeschema kann die auftretenden Knotenmengen mit Hilfe von Kontexten einschränken. Diese Einschränkungen helfen uns, semantisch äquivalente, aber schneller ausführbare Ausdrücke zu bestimmen, die mit existierenden Umsetzungen ausgeführt werden können.

Unsere Anwendung monotoner Datenflußrahmen auf Transformationen erlaubt Gesamtprogrammanalysen, welche sowohl den Datenfluß zwischen Teilausdrücken als auch den Kontrollfluß zwischen Regeln berücksichtigen. Die darauf aufbauende Optimierung liefert schnellere semantisch äquivalente Transformationen für bestehende Umsetzungen.

Fauler Baumaufbau spart bei teilweisem Zugriff auf eine Quelle Speicherplatz und Rechenzeit, indem nur die tatsächlich betretenen Teile als Datenstrukturen im Hauptspeicher aufgebaut werden. Schlüssel dazu sind die Schachtelungseigenschaften von XML.

Analog berechnen faule Transformationen nur die jeweils dynamisch zugegriffenen Teile ihres Ergebnisses. Schlüssel dazu sind die Konfluenz von Transformationen und das Vorhandensein alternativer Auswertungsreihenfolgen. Wie beim faulen Baumaufbau sind auch hier neue Umsetzungen von Transformationen nötig.

Wie erzielten wir diese Fortschritte? Unsere formalen Modelle schlagen eine Brücke zwischen dem Anwendungsgebiet XML und den Disziplinen der Programmanalyse, der Optimierung und der formalen Sprachen. Aber erst unsere neuartigen Analyserepräsentationen machen sie auch tragfähig. Die ausführliche Version dieser Arbeit zeigt, daß die Brücke zweispurig ist. Ich hoffe, diese kurze Zusammenfassung hat Appetit auf mehr geweckt.

Literatur

[ASU86] Aho, A. V., Sethi, R., und Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Addison Wesley. 1986.

[NNH99] Nielson, F., Nielson, H. R., und Hankin, C. L.: *Principles of Program Analysis*. Springer-Verlag. 1999.

[Sc02] Schmelzer, R.: Breaking xml to optimize performance. Technical report. ZapThink LLC. `http://www.zapthink.com/report.html?id=ZapFlash-10072002`. October 2002.

[WG85] Waite, W. und Goos, G.: *Compiler Construction*. Texts and Monographs in Computer Science. Springer. 1985.