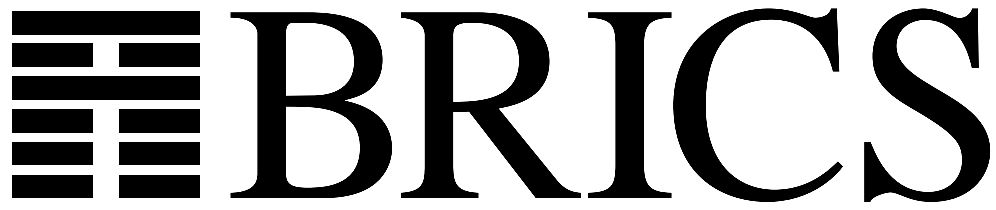


Basic Research in Computer Science

BRICS NS-03-4 M. I. Schwartzbach (ed.): PLAN-X 2004 Informal Proceedings

PLAN-X 2004 Informal Proceedings

Venice, Italy, 13 January, 2004

Michael I. Schwartzbach
(editor)

BRICS Notes Series **NS-03-4**

ISSN 0909-3206 **December 2003**

Copyright © 2003, Michael I. Schwartzbach (editor).
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory NS/03/4/

PLAN-X 2004 Informal Proceedings
Venice, Italy
13 January 2004

The workshop aims at providing a meeting ground for researchers from the XML, programming language, and database communities. XML is already a de-facto industry standard for data exchange, it has from an early stage been embraced by database researchers, and it is gaining increasing interest from programming language researchers.

At this workshop we hope to present recent results, identify new challenges, and inspire the programming language community to focus on XML.

The focus of the workshop is on methods, tools, and theories for processing XML. Example topics include (but are not limited to) XML parsing, XML type system and schemas, analysis and implementation of technologies such as XPath, XSLT, and XQuery, and integration of XML in both general-purpose and domain-specific programming languages.

Greedy Regular Expression Matching 1
(Frisch, Cardelli)

Regular Expression Filters for XML 13
(Hosoya)

Regular Tree Language Recognition with Static Information 28
(Frisch)

An XQuery-Based Language for Processing Updates in XML 40
(Sur, Hammer, Simeon)

Efficient XPath Axis Evaluation for DOM Data Structures 54
(Hidders, Michiels)

STAX/bc: A Binding Compiler for Event-Based XML Data Binding APIs 64
(Reuter, Luttenberger)

Mixed XML/Relational Data Processing 73
(Kadiyska, Suciu)

A Language for Bi-Directional Tree Transformations 83
(Greenwald, Moore, Pierce, Schmitt)

Greedy regular expression matching

Alain Frisch *
École Normale Supérieure

Luca Cardelli
Microsoft Research

Abstract

This paper studies the problem of matching sequences against regular expressions in order to produce structured values. More specifically, we formalize in an abstract way a greedy disambiguation policy and propose efficient matching algorithms. We also formalize and address a folklore problem of non-termination in naive implementations of the greedy semantics.

Regular expression types and patterns have been introduced in the setting of XML-oriented functional languages. Traditionnaly, all the XML values and sequences share a common uniform runtime representation. Our work suggests an alternative implementation technique, where regular expression types define not only a set of abstract flat sequences, but also a custom structured representation for such values. This paves the way to a variety of language designs and implementations to integrate XML structural types in existing languages (class-based OO languages, imperative features, constrained runtime environment, ...).

1 Introduction

1.1 Motivation

Regular expressions play a key role in XML. They are used in XML schema languages (DTD, XML-Schema, Relax-NG, ...) to constrain the possible sequences of children of an element. They naturally lead to the introduction of *regular expression types* and *regular expression patterns* in XML-oriented functional languages (XDUCE [HVP00, HP03, Hos01], XQuery [BCF+03b], ℂDuce [BCF03a]). These works introduce new kinds of questions and give results in the theory of regular expression and regular (tree) languages, such as efficient implementation of inclusion checking and boolean operations, type inference for pattern matching, checking of ambiguity in patterns [Hos03], compilation of pattern matching [Lev03] and optimization of patterns in presence of static information [BCF03a], etc...

This work is a preliminary step in introducing similar ideas to imperative or object-oriented languages. One possible approach is the one pursued by the XTATIC language [GP03], which is a merger between XDUCE and C# [ECM02a]. The value and type algebra are stratified, to allow mixing XDUCE types (regular expressions) and standard .NET CLR [ECM02b] types (classes). Concretely, there is a uniform representation of sequences, and all the XML types collapse to a single native CLR class at runtime. Because of the uniform representation, XTATIC can import subtyping from XDUCE (namely, set inclusion): the CLR does not see any of it at runtime. In order to have a type-safe implicit subtyping (that is, regular language inclusion), the XML fragments must be immutable, or some runtime checks must be introduced at runtime (as for arrays in Java or C#). Also, the uniform representation adds a lot of boxing and indirections. This can be good if the application relies a lot on subtyping, but it can hurt if the application needs fast random access on data (for instance, accessing an element in the middle of a Kleene-star requires a traversal of the sequence), want to mutate data, or needs to cooperate with non-XML parts of the application (that don't expect uniform and boxed values). Finally, collapsing all the XML types to a unique type means that type information is lost after compilation; this raises issues for separate compila-

*This work was supported by an internship at Microsoft Research.

tion.

The starting point of this work was to consider another approach, and study an alternative implementation technique for XTATIC. Instead of having a uniform representation of sequences, we want to represent them with native CLR constructions. In this paper, we consider *types* that are regular expressions. Unlike XDUCE, our types describe not only a set of possible sequences, but also a concrete structured representation of values. We use $\times$, $+$, *, ε to denote concatenation, alternation, Kleene star and the singleton set containing the empty sequence.

Typically, a value of type (`System.Object` $\times$ `int`) should be a struct with two members (that is, a value type in the terminology of the CLR [GS01]), whose second member is a CLR unboxed integer. Similarly, a Kleene-star type `int`* could be an array or another collection with random access features. The benefits of this approach are: (1) data is stored more compactly; (2) it can be accessed (and mutated) efficiently, and (3) it can be made compatible with non-XML specific code (and if the mapping to CLR types retains enough information from XML types, separate compilation is made possible without keeping extra information). The drawback is that we now need coercions between set-theoretic subtypes. For instance, (`int` $\times$ `int`) is a set-theoretic subtype of `int`*, but we need a coercion to use a value of the former where a value of the latter is expected, because the runtime representations of the two types are different.

Such a coercion can always be decomposed (at least conceptually) in two phases: flatten the value of the subtype to a uniform representation, and then match that flat sequence against the super type. The matching process is a generalization of pattern matching in the sense of XDUCE [HP01], and one might want to make it available to the programmer as well. Note that coercions cannot fail, though general pattern matching can.

Another work worth mentioning in this area is the Xen language [MS03], which adds to C# some dose of structural types reminiscent of regular expression types. They bind structural types to native CLR types. However, they loose the nice semantic properties of subtyping and equivalences from regular expression types, and they don't have the equivalent of XDUCE/XTATIC pattern matching. Our work can also be seen as an attempt to follow the Xen approach while sticking to "pure" regular expressions types and patterns.

However, this paper does not propose a language design, such as language features or type systems. Instead, we study the theoretical problem of matching a flat sequence against a type (regular expression). The result of the process is a structured value of the given type. The algorithms we develop could be used in a variety of language designs (how to access information in a structured value; implicit or explicit coercions; mutability of values; different binding semantics for pattern matching, etc). For instance, they can directly be applied to implement the second pass of coercions between subtypes (building structured values from flat sequences).

1.2 The matching problem

The classical theory of regular expressions and automata deals mainly with the recognition problem, namely deciding whether a word belongs to the regular language described by some regular expression. In particular, two regular expressions are equivalent with respect to the recognition problem if they denote the same language.

In the matching problem, regular expressions don't only describe a language, but also a way to extract information from words in this language. This can be formalized in different ways, for instance adding capture variables to regular expressions. In this paper, we take a different approach and say that regular expressions actually denote sets of structured values. Each value can be flattened to obtain a word. For instance, the regular expression t^* denotes an array or list of values, each of type t. In particular, it is possible to access efficiently any element of the array. Similarly, the sequence regular expression $(t_1 \times t_2)$ denotes pairs (v_1, v_2).

Now, regular expression matching can be seen as the process of mapping flat words to structured values. This raises two issues:

- Semantic issue: How to deal with ambiguity in

2

regular expressions? We can distinguish three kinds of solutions: (1) return all the possible matches, (2) disallow ambiguous regular expressions [Hos03], or (3) specify a disambiguation policy to pick a "best" result. In the setting of programming languages, we usually want to return a single result. Also, we don't want to force the programmer to rewrite regular expressions to remove ambiguities, because this process changes the structure of regular expressions and thus the structure of resulting values. Furthermore, accepting ambiguous regular expressions cannot be avoided if we want to import, say XML Schema specifications. Also, they usually lead to more compact expressions. In this paper, we focus on the point (3), and study in detail a given disambiguation policy.

- Implementation issue: How to implement matching efficiently? The classical technique of determinization allows recognition of a regular language in linear time (in the size of the word to be recognized). Can this be adapted to the matching problem?

1.3 Related work

Problematic regular expressions There is a rich literature on efficient implementation of regular expression pattern matching. For instance, Laurikari [Lau01] studies the submatch addressing problem, which extracts less information than our matching problem. Other works [Kea91, DF00] address the matching problem (referred to as "parse extraction"). A key contribution of our work is the treatment of so-called *problematic regular expressions*, together with a clean formalization of a specific disambiguation policy.

Indeed, there is a folklore problem with expression-based implementations of regular expression matching (as opposed to purely automaton-based approaches, which are not suitable for the matching problem): they don't handle correctly the case of a regular expression t^* when t accepts the empty word. Indeed, an algorithm that would naively follow the expansion $t^* \rightsquigarrow (t \times t^*) + \varepsilon$ could enter an infinite loop. Actually, this could even be a problem for *defining* the disambiguation policy when it is only given by a matching algorithm.

Harper [Har99] and Kearns [Kea91] (who speaks of "horrible possibility" for the non-termination problem) propose to keep the naive algorithm, but to use a first pass to rewrite the regular expressions so as the remove the problematic cases. For instance, let us consider the regular expression $t = (a^* \times b^*)^*$. We could rewrite it as $t' = ((a \times a^*) \times b^* + (b \times b^*))^*$. In general, the size of the regular expression can explode in the rewriting. Moreover, this solution has two drawbacks when we consider the matching problem:

- Changing the regular expression changes the type of the resulting values. A value of type t' is not a value of type t.
- The interaction with the disambiguation policy (see below) is not trivial. In particular, we don't see any way to design a rewriting strategy that preserves the disambiguation policy.

Therefore, we do not want to rewrite the regular expressions. Another approach is to patch the naive recognition algorithm to detect precisely the problematic case and cut the infinite loop [Xi01]. This is an *ad hoc* way to define the greedy semantics in presence of problematic regular expressions.

Our approach is different since we want to axiomatize abstractly the disambiguation policy. We identify three notions of problematic words, regular expressions, and values (which represent the ways to match words), relate these three notions, and propose matching algorithms to deal with the problematic case.

Disambiguation policy Specifying a disambiguation policy can be done by providing an explicit matching algorithm. For instance, Vansummeren [Van03] axiomatizes a longest match semantics for the Kleene star with a formal system describing the matching relation. This semantics has a "global" flavor, in the sense that the part of the word matched by a Kleene star t^* depends only on the language ac-

cepted by t and the context of the star, not its internal structure.

A more classical semantics is defined by expanding the Kleene star t^* to $(t \times t^*) + \varepsilon$ and then relying on a disambiguation policy for the alternation (say, first-match policy). This gives a "greedy" semantics, which is sometimes meant as a local approximation of the longest match semantics. However, as described by Vansummeren [Van03], the greedy semantics does not implement the longest match policy. As a matter of fact, the greedy semantics really depends on the internals of Kleene-stars. For instance, consider the regular expressions $t_1 = ((a \times b) + a)^* \times (b + \varepsilon)$ and $t_2 = (a + (a \times b))^* \times (b + \varepsilon)$, and the word $w = ab$. With the greedy semantics, when matching w against t_1, the star captures ab, but when matching against t_2, the star captures only a.

Because of its local nature, the greedy semantics seems easier to implement, and maybe to understand (this point is questionable). Moreover, it has actually been used as the disambiguation policy in several programming languages (XDUCE, ℂDuce, λ^{re} [TSY02]), and at least for this reason it deserves attention.

In this paper, we formalize the greedy semantics by a specification that is independent of any concrete matching algorithm. We define a total ordering on values and specify that the largest possible value must be extracted. The disambiguation policy is then formalized as an optimization problem (extract the largest value with the given flattening). This is similar to the formalization of XDuce pattern matching relation [Hos01, section 2.4.2], except that we tackle with the difficulty of problematic expressions which are rejected in [Hos01].

Implementation of matching A naive backtracking implementation of the greedy semantics is quite easy to give (for the recognition problem, see for instance [TSY02], or [Har99] for the ungreedy variant). In this paper, we provide a linear time algorithm that works in two passes. The idea is to use a first pass to annotate the word and avoid backtracking in the second pass, when the value is constructed.

The first pass scans the word by running a finite state automaton. The automaton is build directly on the syntax tree of the regular expression itself (its states correspond to the nodes of the regular expression syntax tree). A reviewer pointed us to a previous work [Kea91] which uses the same idea. Our presentation is more functional (hence more amenable to reasoning) and is extended to handle problematic regular expressions.

The second pass follows closely the syntax of the regular expression, and is thus very flexible. For instance, one can easily add actions to each node of the regular expression, extract only relevant information, or in the setting of a compiler, generate specialized code for a given regular expression.

2 Notations

Sequences For any set X, we write X^* for the set of sequences over X. Such a sequence is written $[x_1; \ldots; x_n]$. The empty sequence is $[]$. We write $x :: s$ for the sequence obtained by prepending x in front of s and $s :: x$ for the sequence obtained by appending x after s. If s_1 and s_2 are sequences over X, we define $s_1 @ s_2$ as their concatenation. We extend these notations to subsets of X: $x :: X_1 = \{x :: s \mid s \in X_1\}$, $X_1 @ X_2 = \{s_1 @ s_2 \mid s_i \in X_i\}$.

Symbols, words We assume to be given a fixed alphabet Σ, whose elements are called symbols (they will be denoted with c,c_1,...). Elements of Σ^* are called words. They will be denoted with w, w_1,w',...

Types The set of types is defined by the following inductive grammar:

$$t \in T \quad ::= \quad c \mid (t_1 \times t_2) \mid (t_1 + t_2) \mid t^* \mid \varepsilon$$

Values The set of values $\mathcal{V}(t)$ of type t is defined by:

$$\begin{array}{lcl} \mathcal{V}(c) & := & \{c\} \\ \mathcal{V}(t_1 \times t_2) & := & \mathcal{V}(t_1) \times \mathcal{V}(t_2) \\ \mathcal{V}(t_1 + t_2) & := & \mathcal{V}(t_1) + \mathcal{V}(t_2) \\ \mathcal{V}(t^*) & := & \mathcal{V}(t)^* \\ \mathcal{V}(\varepsilon) & := & \{\varepsilon\} \end{array}$$

4

On the right-hand side of this definition, $\times$ denotes the usual Cartesian product, and $+$ the disjoint union. A value of type $t_1 \times t_2$ is written (v_1, v_2) (with $v_i \in \mathcal{V}(t_i)$). A value of type $t_1 + t_2$ is written $e : v$ (with $e \in \{1,2\}$ and $v \in \mathcal{V}(t_i)$). Elements of $\mathcal{V}(t^*)$ can be seen as lists or arrays of values of type t; we will use the letter σ to denote them. Note that the values are structured elements, and no flattening happen automatically.

The flattening $\texttt{flat}(v)$ of a value v is a word defined by:

$$\begin{array}{lll}
\texttt{flat}(c) & := & [c] \\
\texttt{flat}((v_1, v_2)) & := & \texttt{flat}(v_1)@\texttt{flat}(v_2) \\
\texttt{flat}(e : v) & := & \texttt{flat}(v) \\
\texttt{flat}([v_1; \ldots; v_n]) & := & \texttt{flat}(v_1)@\ldots@\texttt{flat}(v_n) \\
\texttt{flat}(\varepsilon) & := & []
\end{array}$$

We write $\mathcal{T}(t) = \{\texttt{flat}(v) \mid v \in \mathcal{V}(t)\}$ for the language accepted by the type t.

3 All-match semantics

In this section, we introduce an auxiliary definition of an all-match semantics that will be used to define our disambiguation policy and to study the problematic regular expressions. For a type t and a word $w \in \texttt{flat}(t)$, we define

$$\texttt{M}_t(w) := \{v \in \mathcal{V}(t) \mid \exists w'.\ w = \texttt{flat}(v)@w'\}$$

This set represents all the possible way to match a prefix of w by a value of type t. For a word w and a value $v \in \texttt{M}_t(w)$, we write w/v the (unique) word w' such that $w = \texttt{flat}(v)@w'$.

Definition 1 *A type is* problematic *if it contains a sub-expression of the form t^* where $[] \in \mathcal{T}(t)$.*

Definition 2 *A value is* problematic *if it contains a sub-value of the form $[\ldots; v; \ldots]$ with $\texttt{flat}(v) = []$. The set of non-problematic values of type t is written $\mathcal{V}^{\texttt{np}}(t)$.*

Definition 3 *A word w is* problematic *for a type t if $\texttt{M}_t(w)$ is infinite.*

The following proposition establish the relation between these three notions.

Proposition 1 *Let t be a type. The following assertions are equivalent:*

1. *t is problematic;*
2. *there exists a problematic value in $\mathcal{V}(t)$;*
3. *there exists a word w which is problematic for t.*

We will often need to do induction both on a type t and a word w. To make it formal, we introduce a well-founded ordering on pairs (t, w): $(t_1, w_1) < (t_2, w_2)$ if either t_1 is a strict sub-expression of t_2 or $t_1 = t_2$ and w_1 is a strict suffix of w_2.

We write $\texttt{M}^{\texttt{np}}_t(w) = \texttt{M}_t(w) \cap \mathcal{V}^{\texttt{np}}(t)$ for the set of non-problematic prefix matches.

Proposition 2 *The following equalities hold:*

$$\begin{array}{lll}
\texttt{M}^{\texttt{np}}_c(w) & = & \left\{ \begin{array}{ll} \{c\} & \textit{if } \exists w'.\ c :: w' = w \\ \emptyset & \textit{otherwise} \end{array} \right. \\
\texttt{M}^{\texttt{np}}_{t_1 \times t_2}(w) & = & \{(v_1, v_2) \mid v_1 \in \texttt{M}^{\texttt{np}}_{t_1}(w), \\
& & v_2 \in \texttt{M}^{\texttt{np}}_{t_2}(w/v)\} \\
\texttt{M}^{\texttt{np}}_{t_1 + t_2}(w) & = & \{e : v \mid e \in \{1,2\}, v \in \texttt{M}^{\texttt{np}}_{t_e}(w)\} \\
\texttt{M}^{\texttt{np}}_{t^*}(w) & = & \{v :: \sigma \mid v \in \texttt{M}^{\texttt{np}}_t(w), \boxed{\texttt{flat}(v) \neq []}, \\
& & \sigma \in \texttt{M}^{\texttt{np}}_{t^*}(w/v)\} \cup \{[]\} \\
\texttt{M}^{\texttt{np}}_{\varepsilon}(w) & = & \{\varepsilon\}
\end{array}$$

This proposition gives a naive algorithm to compute $\texttt{M}^{\texttt{np}}_t(w)$. Indeed, because of the condition $\texttt{flat}(v) \neq []$ in the case for $\texttt{M}^{\texttt{np}}_{t^*}(w)$, the word w/v is a strict suffix of w, and we can interpret the equalities as an inductive definition for the function $\texttt{M}^{\texttt{np}}_t(w)$ (induction on (t, w)).

Note that if we remove this condition $\texttt{flat}(v) \neq []$ and replace $\texttt{M}^{\texttt{np}}_{_}(_)$ with $\texttt{M}_{_}(_)$, we get valid equalities.

Corollary 1 *For any word w and type t, $\texttt{M}^{\texttt{np}}_t(w)$ is finite.*

4 Disambiguation

Let t be a type. The matching problem is to compute from a word $w \in \mathcal{T}(t)$ a value $v \in \mathcal{V}(t)$ whose flattening is w. In general, there are several different solutions. If we want to extract a single value, we need

to define a disambiguation policy, that is, a way to choose a *best* value $v \in \mathcal{V}(t)$ such that $w = \texttt{flat}(v)$. Moreover, we don't want to do it by providing an algorithm, or a set of ad hoc rules. Instead, we want to give a *declarative specification* for the disambiguation policy.

A first step is to reject problematic values. This is meaningful, because if $w \in \mathcal{T}(t)$, then there always exist non-problematic values whose flattening is w. Moreover, if there is a problematic value whose flattening is w, then there are an infinite number of such values. Since we want to specify the best value as being the largest one for a specific ordering (see below), having an infinite number of them is problematic.

Now we need to choose amongst the remaining non-problematic values. To do this, we introduce a total ordering on the set $\mathcal{V}(t)$, and we specify that the best value with a given flattening is the largest *non-problematic* value for this order.

We define a total (lexicographic) ordering $<$ on each set $\mathcal{V}(t)$ by:

$$\begin{array}{ll} c < c & := \texttt{false} \\ (v_1, v_2) < (v_1', v_2') & := (v_1 < v_1') \vee (v_1 = v_1' \wedge v_2 < v_2') \\ (e : v) < (e' : v') & := (e > e') \vee (e = e' \wedge v < v') \\ [] < \sigma' & := \ \sigma' \neq [] \\ v :: \sigma < v' :: \sigma' & := (v < v') \vee (v = v' \wedge \sigma < \sigma') \\ \varepsilon < \varepsilon & := \texttt{false} \end{array}$$

This definition is well-founded by induction on the size of the values. It captures the idea of a specific disambiguation rule, namely a left-to-right policy for the sequencing, a first match policy for the alternation and a greedy policy for the Kleene star.

Definition 4 *Let t be a type and $w \in \mathcal{T}(t)$. We define:*

$$\texttt{m}_t(w) := \max{}_{<}\{v \in \mathcal{V}^{\texttt{np}}(t) \mid \texttt{flat}(v) = w\}$$

The previous section gives a naive algorithm to compute $\texttt{m}_t(w)$. We can first compute the set $\texttt{M}_t^{\texttt{np}}(w)$, then filter it to keep only the values v such that $w/v = []$, and finally extract the largest value from this set (if any). This algorithm is very inefficient because it has to materialize the set $\texttt{M}_t^{\texttt{np}}(w)$, which can be very large.

The recognition algorithm in [TSY02] or [Har99] can be interpreted in terms of our ordering. It generates the set $\texttt{M}_t^{\texttt{np}}(w)$ lazily, in decreasing order, and it stops as soon as it reaches the end of the input. To do this, it uses backtracking implemented with continuations. Adapting this algorithm to the matching problem is possible, but the resulting one would be quite inefficient because of backtracking (moreover, the continuation have to hold partial values, which generates a lot of useless memory allocations).

5 A linear time matching algorithm

In this section, we present an algorithm to compute $\texttt{m}_t(w)$ in linear time with respect to the size of w, in particular without backtracking nor useless memory allocation.

This algorithm works in two passes. The main (second) pass is driven by the syntax of the type. It builds a value from a word by induction on the type, consuming the word from the left to the right. This pass must make some choices: which branch of the alternative type $t_1 + t_2$ to consider, or how many times to iterate a Kleene star t^*. To allow making these choices without backtracking, a first preprocessing pass annotates the word with enough information.

The preprocessing pass consists in running an automaton right-to-left on the word, and keeping the intermediate states as annotations between each symbol of the word.

5.1 Non-problematic case

We first present an algorithm for the case when w is not problematic. Recall the following classical definition.

Definition 5 *A finite state automaton (FSA) with ε-transitions is a triple (Q, q_f, δ) where Q is a finite set (of states), q_f is a distinguished (final) state in Q, and $\delta \subset (Q \times \Sigma \times Q) \cup (Q \times Q)$.*

The transition relation $q_1 \xrightarrow{w} q_2$ (for $q_1, q_2 \in Q$, $w \in \Sigma^*$) is defined inductively by the following rules:

- $q_1 \xrightarrow{[]} q_2$ if $q_1 = q_2$ or $(q_1, q_2) \in \delta$
- $q_1 \xrightarrow{[c]} q_2$ if $(q_1, c, q_2) \in \delta$

6

- $q_1 \xrightarrow{w_1@w_2} q_3$ if $q_1 \xrightarrow{w_1} q_2$ and $q_2 \xrightarrow{w_2} q_3$.

We write $\mathcal{L}(q) = \{w \mid q \xrightarrow{w} q_f\}$.

From types to automata Constructing a non-deterministic automaton from a regular expression is a standard operation. However, we need to keep a tight connection between the automata and the types. To do so, we define a structure of automaton directly on types seen as abstract syntax trees. Formally, we introduce the set of *locations* (or nodes) $\lambda(t)$ of a type t (a location is a sequence over $\{\mathtt{fst}, \mathtt{snd}, \mathtt{lft}, \mathtt{rgt}, \mathtt{star}\}$):

$$\begin{array}{ll} \lambda(c) & := \{[]\} \\ \lambda(t_1 \times t_2) & := \{[]\} \cup \mathtt{fst} :: \lambda(t_1) \cup \mathtt{snd} :: \lambda(t_2) \\ \lambda(t_1 + t_2) & := \{[]\} \cup \mathtt{lft} :: \lambda(t_1) \cup \mathtt{rgt} :: \lambda(t_2) \\ \lambda(t^*) & := \{[]\} \cup \mathtt{star} :: \lambda(t) \\ \lambda(\varepsilon) & := \{[]\} \end{array}$$

For a type t and a location $l \in \lambda(t)$, we define $t.l$ as the subtree rooted at location l:

$$\begin{array}{lcl} t.[] & := & t \\ (t_1 \times t_2).(\mathtt{fst} :: l) & := & t_1.l \\ (t_1 \times t_2).(\mathtt{snd} :: l) & := & t_2.l \\ (t_1 + t_2).(\mathtt{lft} :: l) & := & t_1.l \\ (t_1 + t_2).(\mathtt{rgt} :: l) & := & t_2.l \\ (t^*).(\mathtt{star} :: l) & := & t.l \end{array}$$

Now, let us consider a fixed type t_0. We take: $Q := \lambda(t_0) \cup \{q_f\}$ where q_f is a fresh element.

If l is a location in t_0, the corresponding state will match all the words of the form $w_1@w_2$ where w_1 is matched by $t_0.l$ and w_2 is matched by the "rest" of the regular expression (Lemma 1 below gives a formal statement corresponding to this intuition).

This notion of "rest" is formalized by the successor function $\lambda(t_0) \to Q$.

$$\begin{array}{lcl} \mathtt{succ}([]) & := & q_f \\ \mathtt{succ}(l :: \mathtt{fst}) & := & l :: \mathtt{snd} \\ \mathtt{succ}(l :: \mathtt{snd}) & := & \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{lft}) & := & \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{rgt}) & := & \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{star}) & := & l \end{array}$$

We now define the δ relation for our automaton:

$$\begin{array}{lcl} \delta & := & \{(l, c, \mathtt{succ}(l)) \mid t_0.l = c\} \\ & \cup & \{(l, \mathtt{succ}(l)) \mid t_0.l = \varepsilon\} \\ & \cup & \{(l, l :: \mathtt{fst}) \mid t_0.l = t_1 \times t_2\} \\ & \cup & \{(l, l :: \mathtt{lft}), (l, l :: \mathtt{rgt}) \mid t_0.l = t_1 + t_2\} \\ & \cup & \{(l, l :: \mathtt{star}), (l, \mathtt{succ}(l)) \mid t_0.l = t^*\} \end{array}$$

An example for this construction will be given in the next session for the problematic case.

The following lemma relates the behavior of the automaton, the $\mathtt{succ}(_)$ function, and the flat semantics of types.

Lemma 1 *For any location* $l \in \lambda(t_0)$*:* $\mathcal{L}(l) = \mathcal{T}(t_0.l)@\mathcal{L}(\mathtt{succ}(l))$

First pass We can now describe the first pass of our matching algorithm. Assume that the input is $w = [c_1; \ldots; c_n]$. The algorithm computes $n+1$ sets of states $Q_n, \ldots, Q_0$ defined as $Q_i = \{q \mid q \xrightarrow{[c_{i+1};\ldots;c_n]} q_f\}$. That is, it annotates each suffix w' of the input w by the set of states from which the final state can be reached by reading w'.

Computing the sets Q_i is easy. Indeed, consider the automaton obtained by reversing all the transitions in our automaton (Q, q_f, δ), and use it to scan w right-to-left, starting from q_f, with the classical subset construction (with forward ε-closure). Each step of the simulation corresponds to a suffix $[c_{i+1}; \ldots; c_n]$ of w, and the subset built at this step is precisely Q_i.

This pass can be done in linear time with respect to the length of w, and more precisely in a time $O(|w|\ |t_0|)$ where $|w|$ is the length of w and t_0 is the size of t_0.

Second pass The second pass is written in pseudo-ML code, as a function `build`, that takes a pair (w, l) of a word and a location $l \in \lambda(t_0)$ and returns a value $v \in \mathcal{V}(t_0.l)$.

```
let build(w,l) =
  (* Invariant:  w ∈ L(l)  *)
  match t0.l with
  | c -> c
  | t1 × t2 ->
```


7

```
      let v1 = build(w, l :: fst) in
      let v2 = build(w/v1, l :: snd) in
      (v1, v2)
  | t1 + t2 ->
      if w ∈ L(l :: lft) then
        let v1 = build(w, l :: lft) in
        1 : v1
      else
        let v2 = build(w, l :: rgt) in
        2 : v2
  | t* ->
      if w ∈ L(l :: star) then
        let v = build(w, l :: star) in
        let σ = build(w/v, l) in
        v :: σ
      else
        []
  | ε -> ε
```

The following proposition explains the behavior of the algorithm, and allows us to establish its soundness.

Proposition 3 *If $w \in \mathcal{L}(l)$ and if t_0 is* **non-problematic**, *then the algorithm* build(w, l) *returns* $\max_{<}\{v \in \mathcal{V}(t_0.l) \mid \exists w' \in \mathcal{L}(\texttt{succ}(l)).\ w = \texttt{flat}(v)@w'\}$.

Corollary 2 *If $w \in \mathcal{T}(t_0)$ and if t_0 is* **non-problematic**, *then the algorithm* build$(w, [])$ *returns* $\mathtt{m}_{t_0}(w)$.

Implementation The tests $w \in \mathcal{L}(l)$ can be implemented in constant time thanks to the first pass. Indeed, for a suffix w' of the input, $w' \in \mathcal{L}(l)$ means that the state l is in the set attached to w' in the first pass. Similarly, the precondition $w \in \mathcal{T}(t_0)$ can also be tested in constant time.

The second pass also runs in linear time with respect to the length of the input word (and more precisely in time $O(|w|\ |t_0|)$), because build is called at most once for each suffix w' of w and each location l (the number of locations is finite). This property holds because of the non-problematic assumption (otherwise the algorithm may not terminate).

Note that w is used *linearly* in the algorithm: it can be implemented as a mutable pointer on the input sequence (which is updated when the c case reads a symbol), and it doesn't need to be passed around.

5.2 Solution to the problematic case

Idea of a solution Let us study the problem with problematic types in the algorithm from the previous section. The problem is in the case t^* of the algorithm, when $[] \in \mathcal{T}(t)$. Indeed, the first recursive call to build may return a value v such that $\texttt{flat}(v) = []$, which implies $w/v = w$, and the second recursive call has then the same arguments as the main call. In this case, the algorithm does not terminate.

This can also be seen on the automaton. If the type at location l accepts the empty sequence, there are in the automaton non-trivial paths of ε-transitions from l to l. The idea is to break these paths, by "disabling" their last transition (the one that returns to l) when no symbol has been matched in the input word since the last visit of the state l.

Here is how to do so. A location l is said to be a star node if $t_0.l = t^*$. Any sublocation l' is said to be scoped by l. Note that when the automaton starts an iteration in a star node (by using the ε transition $(l, l :: \texttt{star})$), the only way to exit the iteration (and to reach the final state) is to go back to the star node l. The idea is to prevent the automaton to enter back a star node unless some symbol has been read during the last iteration. This can be done by disabling the ε-transitions of the form $(l, \texttt{succ}(l))$, where $\texttt{succ}(l)$ is a star node scoping l. Concretely, the automaton keeps track of its current state plus a flag b that remembers if something has been read since the last beginning of an iteration in a star.

When a symbol is read, that is, when a transition of the form (l, c, l') is used, the flag is set. When an iteration starts, that is, when a transition $(l, l :: \texttt{star})$ is used, the automaton reset the flag. Then, we just need to disable the ε-transitions $(l, \texttt{succ}(l))$ where $\texttt{succ}(l)$ is a star node that scopes l when the flag is not set. The flag can then be interpreted as the requirement: Something needs to be read in order to exit the current iteration. Consequently, it is natural to start running the automaton with the flag set, and to require the flag to be set at the final node.

From problematic types to automata Let us make this idea formal. We write P for the set of locations l such that $\texttt{succ}(l)$ is an ancestor of l in the

8

abstract syntax tree of t_0 (this implies that $\texttt{succ}(l)$ is a star node). Note that the "problematic" transitions are the ε-transition of the form $(l, \texttt{succ}(l))$ with $l \in P$.

We now take: $Q := (\lambda(t_0) \cup \{q_f\}) \times \{0,1\}$. Instead of (q, b), we write q^b. The final state is q_f^1. Here is the transition relation:

$$\begin{array}{rll} \delta_0 & := & \{(l^b, c, \texttt{succ}(l)^1) \mid t_0.l = c\} \\ & \cup & \{(l^b, l :: \texttt{fst}^b) \mid t_0.l = t_1 \times t_2\} \\ & \cup & \{(l^b, l :: \texttt{lft}^b), (l^b, l :: \texttt{rgt}^b) \mid t_0.l = t_1 + t_2\} \\ & \cup & \{(l^b, l :: \texttt{star}^0) \mid t_0.l = t^*\} \\ & \cup & \{(l^b, \texttt{succ}(l)^b) \mid (*)\} \end{array}$$

where the condition $(*)$ is the conjunction of:

(I) $t_0.l$ is either ε or a star t^*

(II) if $l \in P$, then $b = 1$

Note that the transition relation is monotonic with respect to the flag b: if $q_1^0 \xrightarrow{w} q_2^b$, then $q_1^0 \xrightarrow{w} q_2^{b'}$ for some $b' \geq b$.

We write $\mathcal{L}(q^b) := \{w \mid q^b \xrightarrow{w} q_f^1\}$. As for any FSA, we can simulate the new automaton either forwards or backwards. In particular, it is possible to annotate a word w with a right-to-left traversal (in linear time w.r.t the length of w), so as to be able to answer in constant time any question of the form $w' \in \mathcal{L}(q^b)$ where w' is a suffix of w. This can be done with the usual subset construction. The monotonicity remark above implies that whenever q^0 is in a subset, then q^1 is also in a subset, which allows to optimize the representation of the subsets.

For a set X and a condition C, we write $\mathbf{1}_{[C]}(X)$ to denote X when C holds, and $\emptyset$ otherwise.

Lemma 2 *Let $l \in \lambda(t_0)$ and $L = \mathcal{T}(t_0.l)$. Then:*

$$\begin{array}{rcl} \mathcal{L}(l^1) & = & L @ \mathcal{L}(\texttt{succ}(l)^1) \\ \mathcal{L}(l^0) & = & (L \backslash \{[]\}) @ \mathcal{L}(\texttt{succ}(l)^1) \\ & \cup & \mathbf{1}_{[l \notin P \wedge [] \in L]}(\mathcal{L}(\texttt{succ}(l)^0)) \end{array}$$

Algorithm We now give a version of the linear-time matching algorithm which supports the problematic case. The only difference is that it keeps track (in the flag b) of the fact that something has be consumed on the input since the last beginning of an iteration in a star. The first pass is not modified, except that the new automaton is used. The second pass is adapted to keep track of b.

```
let build'(w, l^b) =
  (* Invariant:  w ∈ L(l^b)  *)
  match t0.l with
  | c -> c
  | t1 × t2 ->
       let v1 = build'(w, l :: fst^b) in
       let b' = if (w/v1 = w) then b else 1 in
       let v2 = build'(w/v1, l :: snd^b') in
       (v1, v2)
  | t1 + t2 ->
       if w ∈ L(l :: lft^b) then
         let v1 = build'(w, l :: lft^b) in
         1 : v1
       else
         let v2 = build'(w, l :: rgt^b) in
         2 : v2
  | t* ->
       if w ∈ L(l :: star^0) then
         let v = build'(w, l :: star^0) in
         (* Invariant: w/v ≠ w *)
         let σ = build'(w/v, l^1) in
         v :: σ
       else
         []
  | ε -> ε
```

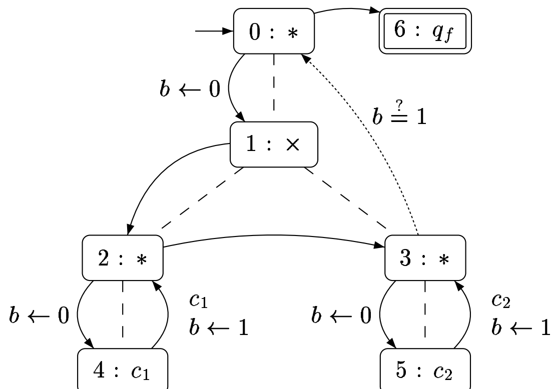
Proposition 4 *Let $w \in \mathcal{L}(l^b)$. Let V be the set of non-problematic values $v \in \mathcal{V}(t_0.l)$ such that $\exists w' \in \mathcal{L}(\texttt{succ}(l)^{b'})$. $w = \texttt{flat}(v)@w'$ with $b' = 1$ if $\texttt{flat}(v) \neq []$ and $((b = 1 \vee l \notin P) \wedge b' = b)$ if $\texttt{flat}(v) = []$. Then the algorithm $\texttt{build}'(w, l^b)$ returns $\max_< V$.*

Corollary 3 *If $w \in \mathcal{T}(t_0)$, then the algorithm $\texttt{build}'(w, []^1)$ returns $\mathtt{m}_{t_0}(w)$.*

Implementation The same remarks as for the first algorithm apply for this version. In particular, we can implement w and b with mutable variables which are updated in the case c (when a symbol is read); thus, we don't need to compute b' explicitly in the case $t_1 \times t_2$.

9

Example To illustrate the algorithm, let us consider the problematic type $t_0 = (c_1^* \times c_2^*)^*$. The picture below represents both the syntax tree of this type (dashed lines), and the transitions of the automaton (arrows). The dotted arrow is the only problematic transition, which is disabled when $b = 0$. Transitions with no symbols are ε-transitions. To simplify the notation, we assign numbers to states.



Let us consider the input word $w = [c_2; c_1]$. The first pass of the algorithm runs the automaton backwards on this word, starting in state 6^1, and applying subset construction. In a remark above, we noticed that if i^0 is in the subset, then i^1 is also in the subset. Consequently, we write simply i to denote both states i^0, i^1. The ε-closure of 6^1 is $S_2 = \{6^1, 0^1, 3^1, 2^1, 1^1\}$. Reading the symbol c_1 from S_2 leads to the state 4, whose ε-closure is $S_1 = \{4, 2, 1, 0, 3^1\}$. Reading the symbol c_2 from S_1 leads to the state 5, whose ε-closure is $S_0 = \{5, 3, 2, 1, 0\}$.

Now we can run the algorithm on the word w with the trace $[S_0; S_1; S_2]$. The flag b is initially set. The star node 0 checks whether it must enter an iteration, that is, whether $1 \in S_0$. This is the case, so an iteration starts, and b is reset. The star node 2 returns immediately without a single iteration, because $4 \notin S_0$. But the star node 3 enters an iteration because $5 \in S_0$. This iteration consumes the first symbol of w, and sets b. After this first iteration, the current subset is S_1. As 5 is not in S_1, the iteration of the node 3 stops, and the control is given back to the star node 0. Since $1 \in S_1$, another iteration of the star 0 starts, and then similarly with an inner iteration of 2. The second symbol of w is consumed. The star node 3 (resp. 0) refuses to enter an extra iteration because $5 \notin S_2$ (resp. $1 \notin S_2$); note that $1^1 \in S_2$, but this is not enough, as this only means that an iteration could take place without consuming anything - which is precisely the situation we want to avoid.

The resulting value is $[([], [c2]); ([c1], [])]$. The two elements of this sequence reflect the two iterations of the star node 0.

6 Extensions, variants

More regular expressions We have presented a limited set of regular expression constructors. We could easily extend all our definitions and results, to include for instance:

- Kleene-star with ungreedy-semantics: for this constructor, the empty sequence is the largest value instead of being the smallest one, in the disambiguation ordering. The corresponding case in the matching algorithm simply tries to return [] when possible, instead of trying to make an extra iteration. Note that the two kinds of Kleene-star could easily cohabit in our framework.
- Non-empty iteration operators t^+, with two variants (greedy and ungreedy).
- Right-context sensitivity operator: $\%t$ that "matches" t but do not remove the corresponding subsequence from the input sequence.

It would be possible to adapt our formalism to capture only interesting parts of sequences by introducing explicit capture variables.

Whether our technique can be adapted to deal with the longest match semantics [Van03] is an open question.

Language design We presented here regular expressions over a finite set of symbols. In a real language design with named typing, we could imagine that regular expressions are built on top of named types, plus some singleton values (character singletons, for instance).

In the design of a language, we can imagine that the programmer could provide custom types to implement containers for regular expression types (this would allow the programmer to use pre-existing types of the language). For instance, in the setting of an extension to C#, the programmer could match a sequence against an existing struct type with public fields, or a class type (the pattern matcher would then call the constructor of this class to store the result). In the setting of an XML-oriented language, there would probably be a specific type `PCDATA`, equivalent to `char*` as for the denoted languages, but with a compact string representation (for instance `System.String`). The idea here is that our matching algorithm allows the implementation to choose custom concrete representation for values.

Our algorithm can also be adapted to add to an existing language (*without* regular expression types) some kind of regular expression pattern matching of sequences. For instance, we have implemented as an extension to a C# compiler a construction that matches an array of type `object[]` against a regular expression built from C# types; it is possible to bind identifiers to elements of the array and to perform arbitrary operations (C#statement) at each node of the regular expression.

Type inference In this work we don't address the question of type inference, that is computing for each node of a type the regular language of all substrings that can be matched by that node (with the given disambiguation policy), when the input is restricted to belong to some given regular language. We believe that an algorithm for computing these regular languages could be derived from our matching algorithm, by applying it symbolically to a whole regular language instead of a single input word.

Optimizations We presented our algorithm as a two passes process. The first one scans completely the input word. In some cases, this can be avoided. For instance, if one knows statically that the flat sequence is matched by the type (for instance because the flat sequence was obtained from flattening a structured value whose type is known at compile time), we can start running the main algorithm, and only when some test is needed, we run the automaton (on the current suffix of the input). In some case, a bounded look-ahead on the sequence can completely avoid the scan. This is the case if the automaton associated to the type has the so-called Glushkov deterministic property, namely that looking at the next symbol of the input removes non determinism. In particular, this is the case with regular expressions in DTD and in XML Schema specifications.

Acknowledgments

We would like to express our gratitude to the anonymous reviewers of PLAN-X 2004 for their comments and in particular for their bibliographical indications.

References

[BCF03a] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: An XML-centric general-purpose language. In *ICFP '03*, 2003.

[BCF+03b] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language.* W3C Working Draft, `http://www.w3.org/TR/xquery/`, May 2003.

[DF00] Danny Dub and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[ECM02a] ECMA. *C# Language Specification.* `http://msdn.microsoft.com/net/ecma/`, 2002.

[ECM02b] ECMA. *CLI Partition I - Architecture.* `http://msdn.microsoft.com/net/ecma/`, 2002.

[GP03] V. Gapayev and B.C. Pierce. Regular object types. In *Proceedings of the 10th workshop FOOL*, 2003.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[Har99] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–469, 1999.

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[Hos03] H. Hosoya. Regular expressions pattern matching: a simpler design. Unpublished manuscript, February 2003.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.

[Kea91] Steven. M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - practice and experience*, 21(8):787–804, 1991.

[Lau01] Ville Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[Lev03] Michael Levin. Compiling regular patterns. In *ICFP '03*, 2003.

[MS03] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *DP-COOL 2003*, 2003.

[TSY02] Naoshi Tabuchi, Eijiro Sumii, , and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Workshop on Types in Programming (TIP)*, 2002.

[Van03] Stijn Vansummeren. Unique pattern matching in strings. Technical report, University of Limburg, 2003. `http://arXiv.org/abs/cs/0302004`.

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *Logic in Computer Science*, 2001.

12

Regular Expression Filters for XML

Haruo Hosoya

The University of Tokyo and École Normale Supérieure
hahosoya@is.s.u-tokyo.ac.jp

Abstract

XML data are described by types involving regular expressions. This raises the question of what language feature is convenient for manipulating such data. Previously, we have given an answer to this question by proposing regular expression pattern matching. However, since this construct is derived from ML pattern matching, it does not have an iteration functionality in itself, which makes it cumbersome to process data typed by Kleene stars. In this paper, we propose a novel programming feature *regular expression filters*. This construct extends the previous proposal by permitting pattern clauses to be closed under arbitrary regular expression operators. This yields many convenient programming idioms such as non-uniform processing of sequences and almost-copying of trees. We further develop a type inference mechanism that obtains (1) types for pattern variables that are *locally precise* with respect to the type of input values and (2) a type for the result of the whole filter expression that is also locally precise with respect to the types of the body expressions. We discuss how our construct is useful in the practice of XML processing and, in particular, how our type inference is crucial for avoiding changes of programs when types of data to be processed frequently evolve.

1 Introduction

XML [3] is a representation of typed data structures for trees. As it becomes popular, a big demand has emerged for special-purpose languages dedicated to processing XML data, in particular, those capable of statically guaranteeing generated data to conform to given types [7, 22, 11, 25, 23, 14, 28, 18].

A peculiarity of XML data is that there is ordering among sibling nodes and, in order to give structure on these nodes, types for XML data typically involve regular expressions [3, 10, 6, 19]. This gives rise to the question: what is a convenient programming feature for manipulating such data? Previously, we have given an answer to this question by proposing *regular expression pattern matching* [17, 16]. In one sense, this design is natural since it combines regular expressions with the ML-style pattern matching paradigm, which has already established its usefulness in tree processing in functional programming languages. However, since pattern matching does not have an iteration functionality in itself, it is quite cumbersome to process data values typed with regular expressions, in particular, those involving Kleene stars. Typically, an explicit recursive function is required each time the program processes such data.[1] As a construct more suitable for this purpose, many languages for XML, such as XSLT [5], XQuery [11], and ℂDuce [2], provide a "for-each" iterator, which performs a given operation on each node of a sequence independently. However, such a construct is *ad hoc* in the sense that it has no connection to regular expressions. More practically, it is difficult for this to manipulate data that repeat groups of several elements. For example, for a value of type `(a,b)*`, we would typically want to process each pair of `a` and `b` rather than each `a` or `b`. (Although such "complex" regular expressions are not very typical, they can occasionally be found in real XML schemas, such as DocBook [26], MusicXML [20], and RecipeML [13]. We give concrete instances in Appendix B.)

This paper proposes *regular expression filters* as an XML manipulation construct that is elegant, i.e., strongly connected to regular expressions, and has both functionalities of pattern matching and iteration. The idea behind this is to extend regular expression pattern matching, which can be seen as an alternation of pattern clauses, by allowing arbitrary regular expressions over pattern clauses. This yields a significant expressiveness, allowing various convenient programming idioms, such as "non-uniform processing of sequences" (processing nodes with the same label differently depending on their positions) and "almost-copying of trees" (copying the whole tree structure with slight changes), that go beyond the capability of pattern matching or a naive for-each style iterator.

We will give a plenty of examples in Section 2, but let us see here a small one. Suppose that we have a value `items` of type `Item*` where `Item` is defined as follows

```
type Item = item[Content]
```

(where the `Content` type is defined somewhere else), that is, the value `items` is a sequence of `item` labels each of which contains a value of type `Content`. Let us format each item in this value, inserting separators between the resulting values. Make sure not to put a separator in the end, and return the empty sequence if the input is the empty sequence. To do this with filters, we first regard the input type `Item*` as the following equivalent one

```
() | (Item*,Item)
```

[1] This might not be the case if the language supported parametric polymorphism and higher-order functions like ML. However, how to incorporate them in our setting, especially polymorphism, is still a difficult question. ℂDuce [14] gives an answer for higher-order functions.

and write the filter expression below, which processes the input value according to the structure indicated by the new regular expression.

```
filter items {
   ()                { () }
 | (item[Any as c]   { format(c), sep[] })*,
   (item[Any as c]   { format(c) })
}
```

That is, when the input is the empty sequence, we use the first clause to emit the empty sequence. In the other case, we use the second clause to format each node and append a separator except for the last node, for which we use the third clause to format the node without adding a separator.

In order for our construct to comfortably be used in a statically typed language, we have designed a type inference mechanism. Similarly to our previous inference technique for regular expression pattern matching [17], the inference here is *local* and *locally precise*. By local, we mean that we calculate types relevant to a filter by using only type information obtained from its adjacent expressions. By locally precise, we mean that the types to be computed precisely represent the values of the corresponding expressions, with the conservative assumption that the type information from the adjacent expressions is also precise.

More specifically, the inference computes types for two parts of a given filter: the bound variables and the result. For the first part, we have already shown [17], in the setting of regular expression patterns, how to calculate types for bound variables from a given type for the input. However, the previous technique has the limitation that it can infer types only for "tail" variables (binding a sub-sequence up to the end of the input sequence), which is not acceptable in the present setting since most filters actually use variables to capture intermediate sequences (as in the above example). Therefore we have developed a novel inference technique based on the combination of a tree automata encoding involving "sequence-capturing variables" (Section 4) and product construction of automata (Section 5). (Frisch, Castagna, and Benzaken have solved the same problem independently using a different algorithm [14].) The second part of the inference is to compute a type for the result of the filter from both the type for the input and a given type for each body expression. We have achieved this by using automata augmented with "action annotations" (Section 4). For the practical implication, the inference is, of course, quite convenient for eliding obvious type annotations, thus increasing the writability and readability of programs. However, in the setting of XML, we believe that such type inference is more important than this usual benefit, in particular, it is actually mandatory in reducing the burden of changes of programs caused by evolution of the types the programs work with, as we will argue in Section 2.5.

An important piece of related work is ℂDuce's `map` and `transform` constructs [2], which are similar to our regular expression filters except that theirs are restricted to uniform processing of sequences. They also have a somewhat similar but different type inference technique for both bound variables and the result. We will make a more detailed comparison in Section 6.

Although we intend this work to be a feature proposal independent of a specific language, we have incorporated regular expression filters and the type inference in our design and implementation of the typed language XDuce for XML processing. The reader is encouraged to try out our prototype system available through:

`http://xduce.sourceforge.net`

The rest of the paper is organized as follows. Section 2 shows a series of examples to illustrate regular expression filters and the type inference. We then formalize these in Section 3. In Section 4, we introduce an automata model for regular expression filters that is suitable for performing the type inference. The inference algorithm itself is described in Section 5. Section 6 compares our work with other work and Section 7 closes this paper. Appendix A shows an algorithm from filters in the surface language to our automata model. Appendix B collects examples of "complex" regular expressions found in real-world DTDs.

2 Examples

This section gives an informal presentation of our language constructs and illustrates their practical uses. The formal definitions can be found in Section 3.

2.1 Values and types

Values in our type system are sequences of labeled values (or base values) and thus representing fragments of XML structures. For example, a sequence of several labeled values like

```
name["Hosoya"],email["hahosoya"],tel["123-456"]
```

and a single label containing some other sequence like

```
person[name["Pierce"],email["bcpierce"]]
```

are values.

Types are regular expressions over labeled types (or base types such as `String`). For example, the following type

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

allows a sequence of a `name` label followed by zero or more `email` labels and an optional `tel` label where each label contains a string. We can also nest types as in the following.

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

We also allow recursive types (and type abbreviations) through type definitions. For example, the following defines a type for trees where each node can have an arbitrary number of subtrees.

```
type Tree = node[Tree*] | leaf[String]
```

We do not impose any restriction on regular expressions, such as determinism or unambiguity (e.g., we allow unions of the same labels like `a[b[]]|a[c[]]`). This makes types to correspond to *nondeterministic finite tree automata*, which form the basis of our framework. More discussions can be found in [19].

A labeled type can actually have a *label set* instead of a single label. For example, we can use union label sets as in `(name|email)[String]`, the universal label set as in `~[String]`, and negation label sets as in `^(tel|email)[String]`. In particular, we use the types `Any` (matching any value) and `AnyOne` (matching any singleton sequence) defined as follows.

```
type Any = AnyOne*
type AnyOne = ~[Any] | String
```

(We assume here that the only base type is String.)

The subtype relation between two types is simply inclusion between the sets of values that they denote. For example, (Name*,Tel*) is a subtype of (Name|Tel)* since the first one is more restrictive than the second. That is, Names must appear before any Tel in the first type, while Names and Tels can appear in any order in the second type.

2.2 Regular expression filters

The basic blocks of regular expression filters are *clauses*. A clause has the form *pattern* { *expression* } and means "if the input value is matched by the pattern, execute the expression," just like ML pattern matching. Patterns are syntactically identical to types except that patterns may contain variables to which the corresponding substructures of the input value are bound. (Therefore a type itself can be used as a pattern by putting no variable.) We do not give a specific definition of expressions in this paper; however, examples shown in the sequel use the following kinds of expression, labelling constructors l[e], concatenations e1,e2, the empty sequence (), variables x, base values such as strings, function calls f(e), and filter expressions themselves (here, e is another expression) .

We can connect or enclose clauses by regular expression operators, forming *filters*. For example, the following filter expression gives a value v to the filter connecting several clauses by the union operator |.[2]

```
filter v {
   person[Any as i]    {  li[...]  }
 | company[Any as j]   {  li[...]  }
 | comment[Any as s]   {  p[]      }
}
```

Here, the form *pattern as variable* binds the variable to the value matched by the pattern. Thus, the above filter matches any singleton sequence where the label of the only element is either person, company, or comment. In the first case, it executes the first body expression creating a label li (containing some sequence not shown here), and similarly for the other cases. As one can see, this use of filters is similar to ML pattern matching and, in fact, is exactly the same as our previous proposal, regular expression pattern matching [17] (except that the union operator in filter expressions does not have the first-match semantics, i.e., top-to-bottom evaluation of clauses, but instead it chooses an arbitrary match if there are multiple possibilities; we will come back to this point in Section 3.)

A filter can be enclosed by Kleene closure *, enabling iteration on sequences. For example, the following wraps the above filter by *.

```
filter v {
 ( person[Any as b]    {  li[...]  }
 | company[Any as f]   {  li[...]  }
 | comment[Any as s]   {  p[s]     } )*
}
```

[2]The precedence rule for operators usable in filters is as follows (from stronger to weaker):

```
* + ? as {}   (suffix operators)
,
|
```

The filter matches any sequence of labels person, company, or comment, converts each label to either li or p in the same way as above, and concatenates all the result in the left-to-right order.

The concatenation of two filters splits the given sequence in the middle so that the first sub-sequence matches the first filter and similarly for the second one, then evaluates each filter with the corresponding sub-sequence, and finally concatenates the two results. (Again, when there are multiple ways of matching, the system chooses an arbitrary one.) For example, the following filter expression

```
filter v {
  (email[Any as e]   { emailAddress[e]  }),
  (tel[Any as t]     { telNumber[t]     })
}
```

matches any value of type email[Any],tel[Any] and replaces the label email with emailAddress and tel with telNumber.

When a filter is enclosed by a label, it matches a value with this label, processes the content with the enclosed filter, and puts the label back to the result. For example, the following filter

```
filter v {
  person[ Any as pc  { proc_person_content(pc) } ]
}
```

processes the content of a person label by the proc_person_content function (defined somewhere else), keeping the label itself.

2.3 Non-uniform sequence processing

Since we can use arbitrary combinations of regular expressions over clauses, this allows us to process a sequence in a much more complex way than "for-each" iteration—such as processing elements with the same label in a different way depending on their positions, or operating on each group of elements in a sequence rather than on each individual element.

We have already seen a small example of non-uniform processing. Let us show here another, slightly more complex one. In the second example shown in Section 2.2, the output may mix lis and ps in the same sequence. However, since this is actually not allowed in XHTML, we want to put consecutive lis together in a dl label this time. For this, we need to process each consecutive list of persons and companys separately from intervening comments. Our solution is to first view the type

```
(Person | Company | Comment)*
```

as the following equivalent type

```
((Person | Company)*, Comment)*, (Person | Company)*.
```

Here, we assume that the following type definitions are given.

```
type Person   = person[Info]
type Company  = company[Info]
type Comment  = comment[String]
```

Then, we write a filter that corresponds to the above regular expression.

3

```
filter v {
  (((Person|Company)* as s   {  dl[proc_mix(s)]  }),
   comment[Any as s]         {  p[s]  })*,
  ((Person|Company)* as s    {  dl[proc_mix(s)]  })
}
```

This filter calls the function `proc_mix` defined below (which takes an argument of type `(Person|Company)*`), which in turn uses another filter to process a sequence of `persons` and `companys`.

```
fun proc_mix ((Person|Company)* as seq) : ... =
  filter seq {
   ( person[Any as i]    {  li[...]  }
   | company[Any as i]   {  li[...]  } )*
  }
```

2.4 Almost-copying of trees

In practical XML processing, we often want to modify small bits of the input document, retaining the rest of the structure. For this purpose, regular expression filters are quite convenient.

For example, consider the following type definitions.

```
type Person = person[Name,Email*,Tel?]
type Name   = name[String]
type Email  = email[String]
type Tel    = tel[String]
```

Suppose that we want to "clean up" a sequence of `persons` by processing the string of each `name`, `email`, or `tel` by a corresponding tidying function. Then, we can write a filter expression by copying the structure of the `Person` type and inserting an appropriate variable binder and a body expression after each `String` type.

```
filter ps {
  person[
    name[String as n  { tidy_name(n) }],
    email[String as e { tidy_email(e) }]*,
    tel[String as t   { tidy_tel(t) }]?
  ]*
}
```

(We assume that the `tidy_name` etc. functions from strings to strings are defined somewhere else.) Note that two Kleene stars are nested, around `person` and around `email`. If we had to write the same function only with pattern matching, we would need two recursive functions, which would be much more cumbersome.

Although the examples so far have been horizontal processing of XML data, filters can also easily express vertical processing. Let us slightly change the last example by allowing each `person` to have a sequence of `persons` recursively.

```
type Person = person[Name,Email*,Tel?,Person*]
```

We would like to tidy leaf information in the same way as before, but, this time, the type involves recursion. For such situation, we can use a recursively defined filter. In our example, we first declare a filter named `tidy_persons` in the following way.

```
rule tidy_persons =
  person[
    name[String as n  { tidy_name(n) }],
    email[String as e { tidy_email(e) }]*,
    tel[String as t   { tidy_tel(t) }]?,
    tidy_persons
  ]*
```

That is, the filter `tidy_persons` processes a sequence of `persons` similarly to the previous paragraph, except that, for the sequence of `persons` appearing in the end of the content of each `person`, we apply the `tidy_persons` filter recursively. To invoke this filter for a given value, we simply refer to the filter's name in a filter expression as in the following.

```
filter ps { tidy_persons }
```

In writing an almost-copying program, we often want to retain the whole substructure of a value, delete it, or insert a substructure. For example, suppose that we have the following type definitions

```
type Person1 = person[Name,Email*]
type Person2 = person[Name,Tel]
```

and want to convert a value from type `Person1` to type `Person2`, retaining the `name` element, delete the sequence of `email` elements, and insert a "default" `tel` element. The following filter achieves this in a simple way.

```
filter p {
  person[
    Name,
    Email*  {  ()  },
    ()      {  tel["unknown"]  }
  ]
}
```

Here, we use the type `Name` as a filter, which simply retains the value matched by the type. Also, note that we use a clause with the empty sequence pattern, by which we can insert any value even though there is nothing corresponding in the original value.

2.5 Type inference

We now turn our attention to the type inference mechanism dedicated to our filter facilities. As mentioned in the introduction, the type inference has two parts: inference of types for bound variables and that of a type for result values.

2.5.1 Types for variables

The type inference for variables is *local* in the sense that it depends only on a type for input values and a subject filter (thus using no constraint from distant expressions). The inference is *locally precise* in the following sense. First, we conservatively assume that all the values from the input type may be passed to the filter. (Note that, at *real* run time, not all of the values may be passed.) Then, under this assumption, we compute, for each pattern variable, a type that contains exactly the values that may be bound to the variable.

For example, consider the following filter where the input `bookcontent` has type `(Person|Company|Comment)*`.

```
filter bookcontent {
   ()
   {  ()  }
 | AnyOne+ as c
   {  dl[
```

```
        filter c {
          (AnyOne as e { li[proc_each(e)] })+
        }]
    }
}
```

The purpose of the filter is to process each node in the input by the `proc_each` function (taking type `(Person|Company|Comment)` and defined somewhere else), put a `li` to each result, and enclose all the results by a `dl`. We need, however, to handle the case of the empty sequence specially since XHTML requires that a `dl` must contain one or more `li`s. In this case, we return the empty sequence as the whole result. From the fact that input values have type `(Person|Company|Comment)*` and the second clause matches only sequences of length one or more, the inference computes the type `(Person|Company|Comment)+` for the variable `c`. From this type, we further infer the type `(Person|Company|Comment)` for the variable `e`.

One benefit from this type inference is, of course, to avoid verbose type annotations. Another, potentially bigger benefit is that the inference can make program code robust against changes of types. For example, suppose that we have changed the input type `(Person|Company|Comment)*` to `(Person|Company|Shop|Comment)*`. (In general, the most common way of evolving a type is to make it larger in denotation.) If we want to process the input exactly in the same way as before (except that the `proc_each` function should now handle the new case), then it is desirable that we need not change the above code fragment. Thanks to the precise inference, the types computed for `c` and `e` will be augmented with `Shop` in their choices, which makes the code remain the same.

2.5.2 Types for result values

The inference for result types is also local in the sense that it depends only on a type for each body expression in addition to the input type and the filter. The type for each body expression may have been obtained by some typing algorithm from the types inferred for bound variables. We do not, however, assume any concrete typing algorithm for body expressions; rather, this is given as a parameter to our inference scheme.

Then, the inference is, again, locally precise. Before describing what we mean by this, let us show an example. Consider the following filter

```
filter v {
  (email[String as s]  { emailAddr[s] } )*
}
```

Suppose that we have the type `emailAddr[String]` for the body expression. What should be the result type of this filter? Naively, we may answer `emailAddr[String]*` from the structure of the filter. However, we could go further. That is, the result type can depend on the input type. For example, it can be `emailAddr[String]+` if the input type is `email[String]+` since the filter produces one output node for each input node. In general, we first conservatively assume that all the values in the input type may be passed to the filter, as before, and that all the values in the type of each body may be returned by it. (Again, some values from the body type may not actually be returned at run time.) Under these assumptions, we compute a type, for result values of the filter, that contains exactly the values that may be returned by the filter.

One may wonder why we need such a precision. Our answer is, again, robustness against type evolution. Let us consider the following hypothetical scenario. Suppose that a schema for address book documents is maintained by a (big) standard committee and you are an engineer in a (small) company writing filters from address books to address books. As in the reality, the committee often changes the type when there are enough external requirements. However, you may not want to change your filter programs every time they change the type especially when you write many different filters for address books. Therefore you may want to make your programs as general as possible in the first place so that they work after foreseeable type changes.

For concreteness, suppose that the address book schema contains the type `PersonInfo`, which is defined as follows at the beginning.

```
type PersonInfo = Name,Addr+,Email*,Tel?
```

You could write a filter with the same structure as this type, but this would not be robust against any type change. The committee might allow any `addr` to be omitted or allow more than one `tel` to be present. So it is better to write in the following way.

```
filter content {
  (name[Any as n]  {  name[tidy_name(n)]   }),
  (addr[Any as a]  {  addr[tidy_addr(a)]   })*,
  (email[Any as e] {  email[tidy_email(e)] })*,
  (tel[Any as t]   {  tel[tidy_tel(t)]     })*
}
```

The question is: does this filter typecheck (with the expected type `Name,Addr+,Email*,Tel?`) before the anticipated type change actually happens? The naive inference would compute the result type as `Name,Addr*,Email*,Tel*` and makes the filter ill-typed since this type is larger than the expected type. On the other hand, our precise inference answers exactly the same type as the input type `Name,Addr+,Email*,Tel?` (which is the same as the expected type) since our inference recognizes, from the input type, that there are only one or more `addr`s and zero or one `tel` in any input value, and therefore so in any output value.

If you further suspect that the committee might change the `PersonInfo` type in a way that allows an arbitrary order among `addr`s, `email`s, and `tel`s. Then, you can make the filter more general in the first place as follows.

```
filter content {
 ( name[Any as n]  {  name[tidy_name(n)]   }
 | addr[Any as a]  {  addr[tidy_addr(a)]   }
 | email[Any as e] {  email[tidy_email(e)] }
 | tel[Any as t]   {  tel[tidy_tel(t)]     })*
}
```

The type inference still gives you the right type—`Name,Addr+,Email*,Tel?`—for the result. If you guess that they might allow more possible fields to be added, then you can write an even more general filter to ignore such fields.

```
filter content {
 ( name[Any as n]  {  name[tidy_name(n)]   }
 | addr[Any as a]  {  addr[tidy_addr(a)]   }
 | email[Any as e] {  email[tidy_email(e)] }
 | tel[Any as t]   {  tel[tidy_tel(t)]     }
 | ^(name|addr|email|tel)[Any]   { ()      } )*
}
```

The type inference still does the right job.

Caveat: our type inference has the restriction that it uses *only one* type for each body expression. For example, consider the following filter expression with the input type `a[T]|b[U]`.

```
filter v {
  ~[ Any as x  { c[x] } ]
}
```

This filter copies the input value, inserting an intermediate label `c` just under the top label `a` or `b`. One may expect the result type to be `a[c[T]]|b[c[U]]`. However, since we can give only one type to the body, the best type we can give is `c[T|U]`, and therefore the result type is `a[c[T|U]]|b[c[T|U]]`, which is larger than the expected one. We know that this limitation is undesirable and can have a negative impact on practice. Unfortunately, this seems the best we could do in the present local inference approach. Indeed, if we remove this restriction, then there is an example where we can obtain unboundedly better types by repeating the inference on the same body expression, which makes it difficult to define the specification of the inference. We will describe this point in Section 3.3 in more detail.

3 Formalization

In this section, we give the syntax and semantics of types, patterns, and filters, as well as the specification of our type inference.

3.1 Syntax

We assume a (possibly infinite) set $\mathcal{A}$ of *labels*, ranged over by a. A *value* v is a sequence of labeled values, where a labeled value is a pair of a label and a value. We use the following syntax for writing values.

$$
\begin{array}{rcll}
v & ::= & \epsilon & \text{empty sequence} \\
 & & a[v] & \text{labeled value} \\
 & & v\,v & \text{concatenation}
\end{array}
$$

For brevity, we omit base values and types (such as strings) from the formalization. (The changes required to add them are straightforward.)

We assume a countably infinite set $\mathcal{S}$ of sets of labels. Each member of $\mathcal{S}$ is called *label set* and ranged over by L. Let the set $\mathcal{S}$ closed under union, intersection, and complementation. We also assume countably infinite sets of pattern names ranged over by X, filter names ranged over by Y, variables ranged over by x, and body ids ranged over by e. Then, patterns P and filters F are defined as follows.

$$
\begin{array}{rcll}
P & ::= & P \text{ as } x & \text{binder} \\
 & & P\,P & \text{concatenation} \\
 & & P \mid P & \text{alternation} \\
 & & P^* & \text{repetition} \\
 & & L[X] & \text{label} \\
 & & \epsilon & \text{empty sequence} \\
 & & & \\
F & ::= & P \to e & \text{clause} \\
 & & F\,F & \text{concatenation} \\
 & & F \mid F & \text{alternation} \\
 & & F^* & \text{repetition} \\
 & & L[Y] & \text{label} \\
 & & \epsilon & \text{empty sequence}
\end{array}
$$

A *pattern grammar* [*filter grammar*] is a finite mapping from pattern names [filter names] to patterns [filters] where the names appearing in each pattern [filter] must be in the domain of the grammar. Throughout this paper, we assume fixed, global pattern grammar E and filter grammar G to be given. (Note that, in the formalization, we forbid nesting of labels and require names to occur only inside labels. This does not, however, lose generality since any definitions without these restrictions can be translated to ones with the restrictions[3].) We define *types*, ranged over by T, to be patterns from which no variables are reachable, i.e., $\mathbf{reach}(T) = \emptyset$.

In the formal system here, we do not concretely specify what "body expressions" are, but rather treat them in an abstract way by using *body ids*. We will show the operational semantics and the type inference specification for filters, which take evaluation and typing algorithms defined on body ids as parameters.

In order to ensure a given pattern to always bind the same set of variables, we impose a "linearity" restriction on them. Let $\mathbf{reach}(P)$ be the set of all variables reachable from P—that is, the smallest set satisfying the following:

$$
\mathbf{reach}(P) \quad = \quad \mathbf{BV}(P) \cup \bigcup_{X \in \mathbf{FN}(P)} \mathbf{reach}(E(X)),
$$

where $\mathbf{BV}(P)$ is the set of variables bound in P and $\mathbf{FN}(P)$ is the set of pattern names appearing in P. We say that a pattern P is *linear* iff, for any (reachable) subphrase P' of P, the following conditions hold.

- $x \notin \mathbf{reach}(P_1)$ if $P' = P_1$ as x.
- $\mathbf{reach}(P_1) \cap \mathbf{reach}(P_2) = \emptyset$ if $P' = P_1\, P_2$.
- $\mathbf{reach}(P_1) = \mathbf{reach}(P_2)$ if $P' = P_1 \mid P_2$.
- $\mathbf{reach}(P_1) = \emptyset$ if $P' = {P_1}^*$.

In what follows, we assume that all patterns are linear and that, if there are clauses $P_1 \to e_1$ and $P_2 \to e_2$ in a filter with $e_1 \neq e_2$, then P_1 and P_2 do not contain the same variable. (We will see examples where different occurrences of clauses have the same body ids.)

Several features not appearing in the syntax can be expressed as shorthands. An optionality operator $F?$ can be rewritten to $F \mid \epsilon$ and a one-or-more-repetition filter F^+ to $F\, F^*$. (Note that each body appearing in F here is duplicated in the expanded form but still has the same id, e.g., no new id is not allocated. This is important for ensuring the expansion not to break the restriction that each body in the original program is typechecked only once.)

As mentioned in Section 2.2, our alternation operator has the nondeterministic semantics, i.e., $F_1|F_2$ matches F_1 or F_2 nondeterministically, as opposed to the first-match semantics used in our previous framework [17]. However, since first-matching is often convenient for writing "default" clauses, we suggest providing (and indeed the current XDuce does support) a separate "first-match" alternation operator $F_1||F_2$, where F_2 matches only when F_1 does not match. One easy way to implement this is to convert the filter $F_1||F_2$ to $F_1|F_2'$ (using the nondeterministic alternation) where F_2' is a filter that behaves exactly the same as F_2 except it

[3] More precisely, even if we drop the above-mentioned restrictions, we still need a restriction to ensure patterns not to have the power of context-free grammars. See [19] for details.

matches only values not matched by F_1. This conversion can easily be done by taking a "set-difference" between F_2 and F_1 preserving binding and action information in F_2. Further details are omitted in this paper. (The first-match alternation suggested here is less expressible than our previous first-match semantics of patterns. In particular, the previous can express greedy matching, e.g., longest or shortest matching. However, we chose a nondeterministic semantics since the first-match semantics introduces a significat complication in the language definition and implementation. More discussions on this topic can be found in [16].)

3.2 Evaluation

We now define the operational semantics of patterns and filters. As mentioned above, the semantics is parametrized over an evaluation algorithm $\mathbf{eval}(V, e)$ that takes an environment V and a body id e and returns a value. An environment is a finite mapping from variables to values, written $x_1 : v_1 \ldots x_n : v_n$.

The semantics of patterns is described by the matching relation $v \in P \Rightarrow V$, read "value v is matched by pattern P and yields environment V." The relation is defined by the following rules.

$$\frac{v \in P \Rightarrow V}{v \in P \text{ as } x \Rightarrow (x : v)\, V} \ \textsc{PVar}$$

$$\frac{\forall i.\ v_i \in P_i \Rightarrow V_i}{v_1\, v_2 \in P_1\, P_2 \Rightarrow V_1\, V_2} \ \textsc{PCat} \qquad \frac{\exists i.\ v \in P_i \Rightarrow V}{v \in P_1 \mid P_2 \Rightarrow V} \ \textsc{POr}$$

$$\frac{\forall i.\ v_i \in P \Rightarrow V_i}{v_1 \ldots v_n \in P^* \Rightarrow V_1 \ldots V_n} \ \textsc{PRep}$$

$$\frac{a \in L \qquad v \in E(X) \Rightarrow V}{a[v] \in L[X] \Rightarrow V} \ \textsc{PLab}$$

We write $v \in P$ when $v \in P \Rightarrow V$ for some V. Given a type T, we write $\mathcal{L}(T)$ for the set of values matched by T.

For the semantics of filters, a straightforward way would be to define a three-place relation on input values, filters, and output values, but we make a slight detour for the ease of formalizing the specification of type inference later. We first define the relation $v \in F \Rightarrow h$, read "from input value v, filter F yields *thunk* h." A thunk h is a sequence of pairs of environments and expressions or labeled thunks, as defined by:

$$\begin{array}{rcl} h & ::= & V \to e \\ & & h\, h \\ & & a[h] \end{array}$$

After evaluating a filter, we execute each body in the resulting thunk under the corresponding environment, and combine all the results by concatenation and labeling. The filter evaluation relation $v \in F \Rightarrow h$ is defined by the following set of rules

$$\frac{v \in P \Rightarrow V}{v \in P \to e \Rightarrow V \to e} \ \textsc{FCla}$$

$$\frac{\forall i.\ v_i \in F_i \Rightarrow h_i}{v_1\, v_2 \in F_1\, F_2 \Rightarrow h_1\, h_2} \ \textsc{FCat} \qquad \frac{\exists i.\ v \in F_i \Rightarrow h}{v \in F_1 \mid F_2 \Rightarrow h} \ \textsc{FOr}$$

$$\frac{\forall i.\ v_i \in F \Rightarrow h_i}{v_1 \ldots v_n \in F^* \Rightarrow h_1 \ldots h_n} \ \textsc{FRep}$$

$$\frac{a \in L \qquad v \in G(Y) \Rightarrow h}{a[v] \in L[Y] \Rightarrow a[h]} \ \textsc{FLab}$$

and the thunk evaluation relation $h \Rightarrow v$ by the following.

$$\frac{}{V \to e \Rightarrow \mathbf{eval}(V, e)} \ \textsc{TCla}$$

$$\frac{h_1 \Rightarrow v_1 \qquad h_2 \Rightarrow v_2}{h_1\, h_2 \Rightarrow v_1\, v_2} \ \textsc{TCat} \qquad \frac{h \Rightarrow v}{a[h] \Rightarrow a[v]} \ \textsc{TLab}$$

3.3 Type Inference

The type inference specification is also parameterized over a typing algorithm $\mathbf{type}(\Gamma, e)$ that takes a type environment Γ and a body id e and returns a type. A type environment is a mapping from variables to types.

In the first step of type inference, we assume that a type T is given for input values and obtain a type environment Γ for the variables appearing in the filter. (Note that we compute one type environment for the whole filter. There is no danger of name crashes since, as already mentioned, patterns with different body ids have distinct sets of variables.) We compute the type environment in the way that, for each variable x, the type $\Gamma(x)$ contains exactly the set of values captured by x as a result of matching values from the input type T against the filter. Formally, we first define the set, written $h(x)$, of values assigned to x in a given thunk h.

$$\begin{array}{rcl} (V \to e)(x) & = & \{V(x)\} \\ (h_1\, h_2)(x) & = & h_1(x) \cup h_2(x) \\ (a[h])(x) & = & h(x) \end{array}$$

Then, the type environment Γ satisfies:

$$\mathcal{L}(\Gamma(x)) \quad = \quad \bigcup \{h(x) \mid v \in T,\ \ v \in F \Rightarrow h\}$$

In the second step of inference, we assume that a type for each body id e is given by $\mathbf{type}(\Gamma, e)$ and obtain a type U for result values. We compute it in the way that U contains the set of values returned by the filter with the assumption that all values from T may be passed to the filter and all values from $\mathbf{type}(\Gamma, e)$ may be returned by the body e. Formally, we first define the set, written $h(\Gamma)$, of values resulted from a given thunk h.

$$\begin{array}{rcl} (V \to e)(\Gamma) & = & \mathcal{L}(\mathbf{type}(\Gamma, e)) \\ (h_1\, h_2)(\Gamma) & = & \{v_1\, v_2 \mid v_1 \in h_1(\Gamma),\ v_2 \in h_2(\Gamma)\} \\ (a[h])(\Gamma) & = & \{a[v] \mid v \in h(\Gamma)\} \end{array}$$

Then, the result type U to compute satisfies:

$$\mathcal{L}(U) \quad = \quad \bigcup \{h(\Gamma) \mid v \in T,\ \ v \in F \Rightarrow h\}.$$

As mentioned before, our inference uses only one type for each body. This restriction is reflected in the fact that we

obtain one type environment for the whole filter, from which the typechecking of each body can yield only one result type. What if we remove this restriction and allow more than once to typecheck each body? The answer is that there is an example where we can always get better types by typechecking the same body more times. Consider the filter

```
filter v {
  a[]* as x   {  x, b[], x  }
}
```

with the input type `a[]*`. The current scheme typechecks the body by giving the type `a[]*` to `x` and obtains the result type

`a[]*,b[],a[]*.`

However, if we split the input type into the case of the empty sequence and the case of sequences of length one or more, we obtain the result type

`(),b[],()   |   a[]+,b[],a[]+`

which is more specific than the previous one. In this way, by splitting the input type into more cases, we can always obtain strictly better types. In the limit, we could split the input type into all the cases of possible input values, where the set of result values would be

$$\{\mathtt{a[]}^n,\mathtt{b[]},\mathtt{a[]}^n \mid n \geq 0\}.$$

But this set is not expressible by our types (since it is not regular). (Note that this argument already applies to regular expression pattern matching: the problem appears when we consider the precision of the result type, not when we extend the language feature.) Although this limitation is quite disappointing, this seems inevitable as long as we stick to the current approach. We will continue this issue in Section 7, where we will show some possible future directions to address this problem.

4 Automata model

In this section, we introduce a notion of filter automata, which is a finite-state machine model corresponding to regular expression filters. We will use this for describing our type inference algorithm in the next section. The basic part of the model is standard, nondeterministic top-down tree automata accepting binary trees. We extend these to handle the additional functionalities provided by filters. First, recall that a filter has the two-layered structure, i.e., the whole is a regular expression over clauses and each clause associates a body id with a pattern, which is a regular expression augmented with variable binders. We represent the whole structure by a tree automaton and add extra annotations to indicate which parts correspond to clauses or variable binders. For each clause, we insert a special transition marked $\mathbf{in}(e)$ (where e is the associated body id) at the entrance of the clause and another transition marked $\mathbf{out}$ at the exit. We will call the subpart of the automaton between the $\mathbf{in}(e)$ and the $\mathbf{out}$ transitions *scope* of e. For each variable binder, we put the variable on each of the transitions corresponding to the regular expression captured by the binder.

Formally, a *filter automaton* A is a tuple $(Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T)$ where Q is a finite set of states, $Q^{\mathbf{init}} \subseteq Q$ is a set of initial states, $Q^{\mathbf{fin}} \subseteq Q$ is a set of final states, and T is a set of transition rules of the form $q_1 \xrightarrow{\lambda} q_2$ where λ is defined by the syntax below, each q is a member of Q, and $\overline{x}$ is a set of variables.

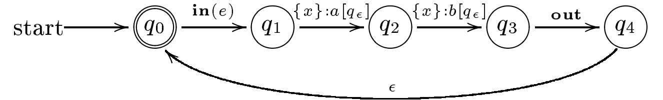
$$\begin{array}{rcll} \lambda & ::= & \overline{x} : L[q] & \text{label} \\ & & \mathbf{in}(e) & \text{action-in} \\ & & \mathbf{out} & \text{action-out} \\ & & \epsilon & \text{silent} \end{array}$$

For a transition $q_1 \xrightarrow{\lambda} q_2$, we call q_1 *source state* and q_2 *sink state*; when $\lambda = \overline{x} : L[q]$, we call q *content state*. Note that, as in the surface language, we allow a label set in each label transition (as opposed to standard automata formalisms using single labels). We annotate a label transition with a set of variables rather than a single variable since variable binders may be nested in the surface language (e.g., `((a[] as x),b[]) as y)`. We sometimes omit the set of variables from a label transition if the set is empty. A filter automaton is said *ϵ-free* if it does not have an ϵ-transition. A *tree automaton* is a filter automaton that has no action transition and whose each label transition has the empty variable set.

For example, we can represent the following filter in the surface language

`((a[],b[]) as x  { e })*`

by the filter automaton depicted below.



Here, we suppose that the state q_ϵ accepts the empty sequence. The filter automaton accepts a sequence of interchanging a and b nodes where each node contains the empty sequence. For each pair of a and b nodes in the input, we first enter the scope of e, then capture both nodes by the variable x, and finally exit from the scope ("executing" the body e).

We make several syntactic restrictions on filter automata to reflect the structure of the surface language syntax. Let us first make some definitions for a given filter automaton $A = (Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T)$. A *(global) path* from q_1 to q_n is a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, either

C1 $q_i \xrightarrow{\lambda} q_{i+1} \in T$ for some λ, or

C2 $q_i \xrightarrow{\overline{x}:L[q_{i+1}]} q' \in T$ for some q', $\overline{x}$, and L.

In this case, q_n is said *(globally) reachable* from q_1; we define $\mathbf{reach}_A(q)$ to be the set of states reachable from q and $\mathbf{reach}_A(R)$ (for a set R of states) to be $\bigcup_{q \in R} \mathbf{reach}_A(q)$. A *local path* from q_1 to q_n is a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, either **C1** holds with λ having the form $\overline{x} : L[q']$ or ϵ, or **C2** holds. In this case, we say that q_n is *locally reachable* from q_1 and define $\mathbf{local}_A(q)$ to be the set of locally reachable states from q. We define

$$\mathbf{BV}_A(q) = \bigcup\{\overline{x} \mid q_1 \in \mathbf{local}_A(q),\ \ q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T\}.$$

A state q is in the *scope* of e if $q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T$ and $q \in \mathbf{local}_A(q_2)$ for some q_1 and q_2. Define $\mathbf{scope}_A(e)$ to be the

set of states in the scope of e. A state q is said in scope when q is in the scope of some body id, and said out of scope when q is not in the scope of any body id. A *horizontal path* from q_1 to q_n is a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, **C1** holds with λ having the form $\overline{x} : L[q']$ or ϵ. In this case, we say that q_n is *horizontally reachable* from q_1 and define $\mathbf{horiz}_A(q)$ to be the set of horizontally reachable states from q. A state q_1 is an **in**-state when $q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T$ for some e, q_2, and an **out**-state when $q_1 \xrightarrow{\mathbf{out}} q_2 \in T$ for some q_2. Now, we impose the following restrictions:

- Each state is either in the scope of a unique body id or out of scope.
- No state is an **out**-state if it is out of scope.
- No state is an **in**-state if it is in scope.
- When $q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T$ with q_1 in scope, no state reachable from q_2 is either an **in**-state or an **out**-state.
- When $q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T$ with q_1 in scope, we have that $\mathbf{BV}_A(q_2) \cap (\overline{x} \cup \mathbf{BV}_A(q_3)) = \emptyset$.

The last condition corresponds to the linearity restrictions in the surface language.

Next, we define the semantics of filter automata. Analogously to the surface language, a given automaton produces, from an input value, an intermediate structure called annotated value rather than directly emitting an output. The produced annotated value is, in fact, identical to the input value except that it is augmented with variable and action annotations. The annotated value is then processed for forming environments, executing bodies, and constructing a result value. Formally, an *annotated value* ρ is defined by the following syntax.

$$
\begin{array}{rcll}
\rho & ::= & \rho\,\rho & \text{concatenation} \\
 & & a[\rho]^{\overline{x}} & \text{label with variable set} \\
 & & \epsilon & \text{empty sequence} \\
 & & \mathbf{in}(e) & \text{action-in} \\
 & & \mathbf{out} & \text{action-out}
\end{array}
$$

An annotated value is **in**-**out**-*free* when it does not contain $\mathbf{in}(e)$ or $\mathbf{out}$. We define $\mathbf{BV}(\rho)$ by the union of the variable sets appearing in ρ. Now, the semantics of filter automata is described by the relation $A \vdash_q v \Rightarrow \rho$, read "automaton A in state q accepts value v and yields annotated value ρ," and inductively defined by the following set of rules.

$$
\frac{q \in Q^{\mathbf{fin}}}{A \vdash_q \epsilon \Rightarrow \epsilon}\ \textsc{Fin}
\qquad
\frac{q_1 \xrightarrow{\epsilon} q_2 \in T \quad A \vdash_{q_2} v_1 \Rightarrow \rho}{A \vdash_{q_1} v_1 \Rightarrow \rho}\ \textsc{Eps}
$$

$$
\frac{\begin{array}{c} a \in L \quad q_1 \xrightarrow{\overline{x}:L[q_3]} q_2 \in T \\ A \vdash_{q_3} v_1 \Rightarrow \rho_1 \quad A \vdash_{q_2} v_2 \Rightarrow \rho_2 \end{array}}{A \vdash_{q_1} a[v_1]\,v_2 \Rightarrow a[\rho_1]^{\overline{x}}\,\rho_2}\ \textsc{Lab}
$$

$$
\frac{q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T \quad A \vdash_{q_2} v \Rightarrow \rho}{A \vdash_{q_1} v \Rightarrow \mathbf{in}(e)\,\rho}\ \textsc{Enter}
$$

$$
\frac{q_1 \xrightarrow{\mathbf{out}} q_2 \in T \quad A \vdash_{q_2} v \Rightarrow \rho}{A \vdash_{q_1} v \Rightarrow \mathbf{out}\,\rho}\ \textsc{Exit}
$$

We form an environment from an **in**-**out**-free annotated value by using the following **envof** function

$$
\begin{array}{l}
\mathbf{envof}(a[\rho_1]^{\overline{x}}\,\rho_2)(x) \\
= \left\{ \begin{array}{ll} a[\mathbf{erase}(\rho_1)]\,\mathbf{envof}(\rho_2)(x) & (x \in \overline{x}) \\ \mathbf{envof}(\rho_1)(x) & (x \notin \overline{x},\ x \in \mathbf{BV}(\rho_1)) \\ \mathbf{envof}(\rho_2)(x) & (x \notin \overline{x},\ x \notin \mathbf{BV}(\rho_1)) \end{array} \right. \\
\mathbf{envof}(\epsilon)(x) \qquad = \quad \epsilon
\end{array}
$$

where $\mathbf{erase}(\rho)$ is the value after eliminating all variables from ρ:

$$
\begin{array}{rcl}
\mathbf{erase}(a[\rho_1]^{\overline{x}}\,\rho_2) & = & a[\mathbf{erase}(\rho_1)]\,\mathbf{erase}(\rho_2) \\
\mathbf{erase}(\epsilon) & = & \epsilon
\end{array}
$$

To understand the definition of **envof**, first note that linearity ensures that, in any annotated value resulted from a matching, the same variables occur only in the same sequence. For example, we may have an annotated value

$$b[a[]^{\{x\}}\,a[]^{\{x\}}]^{\emptyset}$$

but never

$$b[a[]^{\{x\}}]^{\emptyset}\,a[]^{\{x\}}.$$

Thus, when the **envof** function visits each node $a[\rho_1]^{\overline{x}}\,\rho_2$ of the given annotated value, there are only three cases. First, the node's variable set contains x, in which case we retain this node (by erasing all the annotations from its content ρ_1) and proceed to the remaining sequence ρ_2. In the second case, x occurs in ρ_1. Then, x does not in ρ_2, so we ignore ρ_2. In the third case, x does not occur in ρ_1. Then, x may occur in ρ_2, so we proceed to ρ_2.

Finally, to construct the result value from an annotated value, we use the relation $\rho \rightsquigarrow v$ defined as follows.

$$
\frac{\mathbf{eval}(\mathbf{envof}(\rho), e) = v_1 \quad \sigma \rightsquigarrow v_2}{\mathbf{in}(e)\,\rho\,\mathbf{out}\,\sigma \rightsquigarrow v_1\,v_2}\ \textsc{RCla}
$$

$$
\frac{\sigma_1 \rightsquigarrow v_1 \quad \sigma_2 \rightsquigarrow v_2}{a[\sigma_1]\,\sigma_2 \rightsquigarrow a[v_1]\,v_2}\ \textsc{RLab}
\qquad
\frac{}{\epsilon \rightsquigarrow \epsilon}\ \textsc{RFin}
$$

The relation uses the same evaluation function **eval** as in the previous section.

It is easy to translate regular expression filters to filter automata by using a variation of the standard translation algorithm from string regular expressions to string automata [15]. A translation algorithm is given in Appendix A. Also, we can easily convert any filter automata to ϵ-free filter automata by the standard ϵ-elimination. The details are omitted from this abstract.

Since the presented semantics contains nondeterminism, a naive implementation of this with backtracking would be inefficient. Although we have not yet figured it out, we believe that we can construct a linear-time algorithm for evaluating filters by adapting existing linear-time algorithms for checking membership of tree automata, e.g., [24].

5 Inference algorithm

In this section, we describe our inference algorithm using filter automata introduced in the last section and give a brief discussion of its correctness.

5.1 Inference for variables

For bound variables, the inference takes as inputs a tree automaton A (representing the input type) and a filter automaton B. For simplicity, we assume that both automata are ϵ-free. The inference algorithm works in two steps. First, we specialize the filter automaton B with respect to the tree automaton A such that the resulting automaton D behaves exactly the same as the original B except that it accepts only values accepted by the automaton A. For this, we use a variation of standard product construction where we preserve the action and the variable binding behaviors in the automaton B. Second, we obtain, for each variable x, a tree automaton $H^{e,x}$ (where x is bound in the scope of e) such that $H^{e,x}$ accepts a value v if and only if the filter automaton D accepts some value from A and binds x to v. We compute the automaton $H^{e,x}$ from the filter automaton D by retaining all the transitions with variable sets containing x and eliminating all the other transitions.

Formally, we first construct a product automaton $C = (Q_C, Q_C^{\mathbf{init}}, Q_C^{\mathbf{fin}}, T_C)$ from $A = (Q_A, Q_A^{\mathbf{init}}, Q_A^{\mathbf{fin}}, T_A)$ and $B = (Q_B, Q_B^{\mathbf{init}}, Q_B^{\mathbf{fin}}, T_B)$ as follows.

$$
\begin{array}{rcl}
Q_C & = & Q_A \times Q_B \\
Q_C^{\mathbf{init}} & = & Q_A^{\mathbf{init}} \times Q_B^{\mathbf{init}} \\
Q_C^{\mathbf{fin}} & = & Q_A^{\mathbf{fin}} \times Q_B^{\mathbf{fin}} \\
T_C & = & \left\{ \langle p_1, q_1 \rangle \xrightarrow{\overline{x}:(K \cap L)[\langle p_3, q_3 \rangle]} \langle p_2, q_2 \rangle \mid p_1 \xrightarrow{K[p_3]} p_2 \in T_A, \quad q_1 \xrightarrow{\overline{x}:L[q_3]} q_2 \in T_B, \quad K \cap L \neq \emptyset \right\} \\
& \cup & \left\{ \langle p, q_1 \rangle \xrightarrow{\mathbf{in}(e)} \langle p, q_1 \rangle \mid p \in Q_A, \quad q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T_B \right\} \\
& \cup & \left\{ \langle p, q_1 \rangle \xrightarrow{\mathbf{out}} \langle p, q_2 \rangle \mid p \in Q_A, \quad q_1 \xrightarrow{\mathbf{out}} q_2 \in T_B \right\}
\end{array}
$$

The first clause of T_C follows the standard technique of product construction except that we take the intersection of the label sets from both transitions (making sure that the intersection is not empty) and that we copy the variable set in the transition from the automaton B. For the second and the third clauses, we keep the action annotation on the transition from the automaton B. Since the automaton A should not consume any input while the automaton B takes this action, the created transition connects the state $\langle p, q_1 \rangle$ to $\langle p, q_2 \rangle$ where the first component remains the same.

Next, we eliminate all useless states from the automaton C and obtain the automaton $D = (Q_D, Q_D^{\mathbf{init}}, Q_D^{\mathbf{fin}}, T_D)$ defined as follows.

$$
\begin{array}{rcl}
Q_D & = & \left\{ r \mid r \in \mathbf{reach}_C(Q_C^{\mathbf{init}}), \ \ C \vdash_r v \Rightarrow \rho \right\} \\
Q_D^{\mathbf{init}} & = & Q_C^{\mathbf{init}} \cap Q_D \\
Q_D^{\mathbf{fin}} & = & Q_C^{\mathbf{fin}} \cap Q_D \\
T_D & = & \left\{ q_1 \xrightarrow{\overline{x}:a[q_3]} q_2 \in T_C \mid q_1, q_2, q_3 \in Q_D \right\} \\
& \cup & \left\{ q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T_C \mid q_1, q_2 \in Q_D \right\} \\
& \cup & \left\{ q_1 \xrightarrow{\mathbf{out}} q_2 \in T_C \mid q_1, q_2 \in Q_D \right\}
\end{array}
$$

That is, we keep only the states that are reachable from initial states and accept some values.[4]

The final step is, from the automaton D, to extract, for each variable x, an automaton $H^{e,x}$ representing the set of values that x may capture. Note that such a captured value is the concatenation of the nodes that are matched by x-annotated label transitions between an $\mathbf{in}(e)$-transition and an $\mathbf{out}$-transition in the automaton D. Thus, the basic idea of the inference is to obtain an automaton after extracting only such label transitions from D. A subtlety here is, however, that each state in the automaton D can have two different behaviors. That is, after following an $\mathbf{in}$-transition, we are in the "capture mode," where we skip the nodes matched by transitions without x; when we take a label transition *with* x, we get into the "duplicate mode" from the content state of the transition, where we completely retain the structure of the input. Thus, in creating the new automaton $H^{e,x}$, we copy two complete sets of states from the automaton D: a capture-mode state $\langle r, 0 \rangle$ and a duplicate-mode $\langle r, 1 \rangle$ for each state r of D. Formally, the final step is to compute $H^{e,x} = (Q_{H^{e,x}}, Q_{H^{e,x}}^{\mathbf{init}}, Q_{H^{e,x}}^{\mathbf{fin}}, T_{H^{e,x}})$ defined as follows.

$$
\begin{array}{rcl}
Q_{H^{e,x}} & = & \left\{ \langle r, 0 \rangle, \langle r, 1 \rangle \mid r \in Q_D \right\} \\
Q_{H^{e,x}}^{\mathbf{init}} & = & \left\{ \langle r, 0 \rangle \mid r' \xrightarrow{\mathbf{in}(e)} r \in T_D \right\} \\
Q_{H^{e,x}}^{\mathbf{fin}} & = & \left\{ \langle r, 0 \rangle, \ \langle r, 1 \rangle \mid r \in Q_D^{\mathbf{fin}} \right\} \\
& \cup & \left\{ \langle r, 0 \rangle \mid r \xrightarrow{\mathbf{out}} r' \in T_D \right\} \\
T_{H^{e,x}} & = & \left\{ \langle r_1, 0 \rangle \xrightarrow{L[\langle r_3, 1 \rangle]} \langle r_2, 0 \rangle \mid r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D, \ \ x \in \overline{x} \right\} \\
& \cup & \left\{ \langle r_1, 0 \rangle \xrightarrow{\epsilon} \langle r_2, 0 \rangle \mid r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D, \quad x \in \mathbf{BV}_D(r_2) \setminus \overline{x} \right\} \\
& \cup & \left\{ \langle r_1, 0 \rangle \xrightarrow{\epsilon} \langle r_3, 0 \rangle \mid r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D, \quad x \notin \mathbf{BV}_D(r_2) \cup \overline{x} \right\} \\
& \cup & \left\{ \langle r_1, 1 \rangle \xrightarrow{L[\langle r_3, 1 \rangle]} \langle r_2, 1 \rangle \mid r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D \right\}
\end{array}
$$

In the first clause of $T_{H^{e,x}}$, we copy all the x-annotated label transitions in the capture mode, where the content state is in the duplicate mode. In the second and third clauses, we create an ϵ-transition in the capture mode for each label transition without x in D, where the sink state is either the remainder r_2 or the content r_3 depending on whether x is bound in r_2 or not. (Recall that the linearity restriction ensures that x is never bound both in r_2 and in r_3.) The fourth clause copies all the label transitions in the duplicate mode.

Let k be the number of variables appearing in B. The complexity of the inference algorithm for variables is $O(|T_A| \cdot |T_B| \cdot k)$ since the number of the generated transitions in the first phase (C) is proportional to $|T_A| \cdot |T_B|$[5], the second phase (D) is linear, and the final phase ($H_{e,x}$) creates k automata each with a linear number of transitions relative to the one in the second phase.

5.2 Inference for result values

For result values, the inference takes as inputs a tree automaton J_e for each body id e (representing the set of values returned by e), in addition to the filter automaton D

[4] In tree automata, states that can reach final states do not necessarily accept some values, unlike string automata (cf. [8]).

[5] Here, the complexity of computing $K \cap L$ and checking its emptiness is not made clear. However, it does not, at least, depend on m or n.

obtained in the last subsection (which represents the target filter restricted to the input type). For simplicity, we assume that J_e is ϵ-free. (Note that D is also ϵ-free from its definition.) We then produce a tree automaton K representing the set of values returned by the filter automaton D.

To see what values should be contained in K, let us trace the behavior of the filter automaton D. Starting from its initial state, we retain all the labels matched by label transitions that are out of scope. After entering in the scope of a body id e, we perform variable binding. When exiting from the scope, we execute the body and emit the resulting value, which, as we have assumed, is contained in J_e. We then go back to the behavior of out-of-scope states. We continue these until we reach a final state.

Thus, in building the automaton K, we basically need to replace each subautomaton in D enclosed by an $\mathbf{in}(e)$-transition and an **out**-transition by the corresponding automaton J_e. The replacement can be done by reconnecting the source state of each $\mathbf{in}(e)$-transition to J_e's initial states and reconnecting J_e's final states to the sink state of the **out**-transition. However, there are two subtleties here. First, several subautomata in different places may have the same body id but may have **out**-transitions with different sink states. Therefore we cannot simply reconnect the automaton J_e's final states to all these sink states; rather, we have to duplicate the automaton J_e for each of such subautomata. Second, since we intend to concatenate a value resulted from e to the continuing value, we need to link only the "top-level" (i.e., horizontally reachable from J_e's initial states) final states of J_e to the sink state of the **out**-transition. Therefore we duplicate the top-level states of the automaton J_e, but we leave the final states in deeper levels not to be reconnected. Thus, in creating K, we copy one complete set of states from D ("out-of-scope" states), one complete set of states from J_e for each subautomaton described above ("top-level" states), and one complete set of states from J_e ("deeper-level" states). Each state in the second set is written $\langle p, q\rangle$ where p is the sink state of the $\mathbf{in}(e)$-transition and q is a state from D. Formally, given $J_e = (Q_{J_e}, Q^{\mathbf{init}}_{J_e}, Q^{\mathbf{fin}}_{J_e}, T_{J_e})$, we obtain $K = (Q_K, Q^{\mathbf{init}}_K, Q^{\mathbf{fin}}_K, T_K)$ defined as follows.

$$
\begin{array}{rcl}
Q_K & = & Q_D \cup \bigcup_e \{(Q_D \times Q_{J_e}) \cup Q_{J_e}\} \\
Q^{\mathbf{init}}_K & = & Q^{\mathbf{init}}_D \\
Q^{\mathbf{fin}}_K & = & Q^{\mathbf{fin}}_D \cup Q^{\mathbf{fin}}_{J_e} \\
T_K & = & \left\{ p_1 \xrightarrow{L[p_3]} p_2 \in T_D \right\} \\
& \cup & \left\{ p_1 \xrightarrow{\epsilon} \langle p_2, q_3\rangle \mid p_1 \xrightarrow{\mathbf{in}(e)} p_2 \in T_D,\ \ q_3 \in Q^{\mathbf{init}}_{J_e} \right\} \\
& \cup & \left\{ \langle p, q_1\rangle \xrightarrow{L[q_2]} \langle p, q_3\rangle \mid p \in Q_D,\ \ q_1 \xrightarrow{L[q_2]} q_3 \in T_{J_e} \right\} \\
& \cup & \bigcup_e T_{J_e} \\
& \cup & \left\{ \langle p_0, q_1\rangle \xrightarrow{\epsilon} p_3 \mid q_1 \in Q^{\mathbf{fin}}_{J_e},\ \ p_2 \in \mathbf{horiz}_D(p_0),\ \ p_2 \xrightarrow{\mathbf{out}} p_3 \in T_D \right\}
\end{array}
$$

In the first clause of T_K, we simply copy all the transitions from D. In the second clause, we connect the source state of each $\mathbf{in}(e)$-transition to each initial state of J_e duplicated for the transition's sink state. The third clause copies the transitions of J_e for each duplicate, where we tag the duplicate's "representative" state on their source and sink states. The fourth clause copies these transitions without tagging. Note that each transition in the third clause has a content state in the fourth clause. The fifth clause connects a final state of p_0's duplicate of J_e to the sink state of each corresponding **out**-transitions. Such a transition has the source state horizontally reachable from the state p_0.

Let l be the sum of the numbers of transitions in the automata J_e for all e. By noticing that each clause of T_K contains at most $|T_D| \cdot l$ transitions, we can easily see that the complexity of the inference algorithm for result values is $O(|T_A| \cdot |T_B| \cdot l)$.

5.3 Correctness

Let us first discuss the correctness of the first part of the inference—for bound variables. We prove that, for each variable x, the automaton $H^{e,x}$ accepts the values that x is bound to as a result of matching the filter automaton B against values from the input type A. To precisely state that x is bound to a value w, we actually need to say that the annotated value yielded by the matching contains an **in**-**out**-free annotated value ρ as a substructure enclosed by an $\mathbf{in}(e)$ and an **out**, and that the extraction of the x-marked subnodes from ρ results in the value w. Formally, we first define *contexts*, i.e., annotated values containing a single hole $[\cdot]$:

$$
\begin{array}{rcl}
S & ::= & a[S]^{\overline{x}}\,\rho \\
& & a[\rho]^{\overline{x}}\,S \\
& & [\cdot]\,\rho \\
& & \mathbf{in}(e)\,S \\
& & \mathbf{out}\,S
\end{array}
$$

We write $S[\rho]$ for the annotated value after replacing S's hole with ρ. Now, the main theorem for the first part of the inference is as follows.

Theorem 1 *The following are equivalent.*

1. $A \vdash v$ *and* $B \vdash v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ *where* ρ *is* **in**-**out**-*free and* $\mathbf{envof}(\rho)(x) = w$.
2. $H^{e,x} \vdash w$.

This theorem follows from three lemmas shown below (Lemma 1, Lemma 2, and Lemma 3) corresponding to the three steps of the algorithm.

For the first step, we show that the created filter automaton C (in the state $\langle p, q\rangle$) exactly simulates the behavior of the original filter automaton B (in the state q) for the values accepted by the tree automaton A (in the state p).

Lemma 1 $A \vdash_p v$ *and* $B \vdash_q v \Rightarrow \rho$ *if and only if* $C \vdash_{\langle p,q\rangle} v \Rightarrow \rho$.

For the second step, we show that the automaton C and the automaton D with no useless states behave the same in the states that are reachable from C's initial states.

Lemma 2 $C \vdash_q v \Rightarrow \rho$ *and* $q \in \mathbf{reach}_C(Q^{\mathbf{init}}_C)$ *if and only if* $D \vdash_q v \Rightarrow \rho$.

For the third step, we show that the automaton $H^{e,x}$ accepts a value w if and only if the automaton D yields an annotated value that contains an **in**-**out**-free annotated value between an $\mathbf{in}(e)$ and an **out** and w is the extraction of the x-marked subnodes.

Lemma 3 *The following are equivalent.*

1. $D \vdash v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ *where* ρ *is* **in**-**out**-*free and* $\mathbf{envof}(\rho)(x) = w$.

2. $H^{e,x} \vdash w$.

Next, we turn our attention to the second part of the inference—for result values. First, we define a relation $\rightsquigarrow_J$ for constructing a result value from an annotated value, which slightly modifies the relation $\rightsquigarrow$ defined in Section 4 so as to incorporate the assumption that e's result values come from J_e (instead of **eval** applied to e with the environment $\mathbf{envof}(\rho)$).

$$\frac{J_e \vdash v_1 \qquad \sigma \rightsquigarrow_J v_2}{\mathbf{in}(e)\,\rho\,\mathbf{out}\,\sigma \rightsquigarrow_J v_1\, v_2}\ \text{JCLA}$$

$$\frac{\sigma_1 \rightsquigarrow_J v_1 \qquad \sigma_2 \rightsquigarrow_J v_2}{a[\sigma_1]\,\sigma_2 \rightsquigarrow_J a[v_1]\,v_2}\ \text{JLAB} \qquad \frac{}{\epsilon \rightsquigarrow_J \epsilon}\ \text{JFIN}$$

(Note that JCLA ignores the annotated value ρ between the $\mathbf{in}(e)$ and **out**.) Clearly, the relation $\rightsquigarrow_J$ is more conservative than $\rightsquigarrow$ in the sense that $\sigma \rightsquigarrow v$ implies $\sigma \rightsquigarrow_J v$, provided $J_e \vdash \mathbf{eval}(\mathbf{envof}(\rho), e)$ for any ρ where $\sigma = S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$.

Then, the second main theorem shown below states that the tree automaton K accepts the values that are obtained by executing the filter automaton B for values from A and evaluating the produced annotated value σ by $\rightsquigarrow_J$.

Theorem 2 *$A \vdash v$ and $B \vdash v \Rightarrow \sigma$ with $\sigma \rightsquigarrow_J w$ if and only if $K \vdash w$.*

6 Related Work

With the same motivation as ours, the ℂDuce language [2] supports a feature called `map` (and a similar one called `transform`) that combines pattern matching and iteration. Unlike ours, their constructs are limited to the uniform processing of each element, that is, equivalent to writing filters like

```
filter e {
  ( p1  {  e1  }
  | ...
  | pn  {  en  } )*
}
```

where each pattern here is required to match only a single element. (They also support another feature called `xtransform` to allow recursive descending.) Another similar proposal is a simple for-each style iterator with a type-matching facility in XML Query Algebra [12]. As briefly discussed in the introduction, since these constructs do not have a strong connection to regular expressions, it is quite difficult to process XML data with slightly unusual types such as `(a,b*)*`[6]or process data in a non-uniform way as discussed in Section 2.3 and Section 2.4.

ℂDuce also supports type inference both for bound variables and result values. The difference from ours is, however, what types to be computed. First, they allow the inference to go over each body expression more than once, whereas we limit it to only once. As a result, they can obtain better types than ours. However, in return, they have no accurate specification of the type inference and therefore the user might have a difficulty in figuring out the reasons of type errors reported by the system. XML Query Algebra, on the other hand, uses a syntax-based specification of their type inference, with no desirable property of precision.

A popular idiom found in many query languages for XML [9, 4, 1, 11] is a construct of the form `select` e `where` p, which collects the set of all bindings resulted from matching the input value against the pattern p, then evaluates the expression e under each binding, and finally concatenates all the results. Usually in this style of features, the "matching" part uses powerful pattern languages and therefore is quite expressive, whereas the "processing" part is not as satisfactory since it allows only one expression for any match. In particular, it is typically difficult to process different occurrences (or cases) of data in different ways, which is exactly what regular expression filters are good at.

Other type inference methods are known for other computation models for XML, including k-pebble tree transducers [23], subsets of XSLT [28, 21], and extended-path-based queries [25]. Although these use their own techniques for handing special properties of their models, all of these, like us, use product construction for specializing the target program with respect to the input type.

7 Conclusions

We have shown that the simple idea of regular expressions on pattern clauses can yield significant expressiveness allowing non-trivial and useful programming idioms. Though, no single language feature is suitable for every purpose. Indeed, the kind of processing that filters permit is, roughly, the *map* operation as in usual functional languages; we cannot express *fold*-like processing, for example. However, rather than trying to extend our feature to allow as many programming patterns as possible, we prefer to keep it simple and easy to use. On the other hand, the effort required for implementing filters is not so big. The type inference is a series of simple operations on automata and the addition from our previous inference for pattern matching is rather moderate.

One dissatisfaction about the present proposal is the precision of the type inference, as discussed in Section 3—the computed types are sometimes not precise enough due to the restriction that the inference can use only one type for each body expression. For breaking through this obstacle, there are at least two directions. One possibility, taken by ℂDuce's map and transform [2], is to give up having an accurate specification of type inference and compute *some* type that is more precise than ours. Whether this lack of specification is acceptable from the user's point of view is, however, yet to be seen. (There might be an intermediate solution that gives both a simple specification and a better precision, but whether it exists is still an open question.) Another possibility is to pursue a *backward inference* approach, which computes input types from output types (the opposite to our inference) [23, 28]. This approach has successfully dealt with similar problems in several different settings, and therefore seems to be the most promising at the moment.

[6]For skeptists who do not believe that such types actually exist, the following line appears in line 1667 of file `dbhierx.mod` of the DTD of DocBook XML 4.2.

```
<!ELEMENT indexentry %ho; (primaryie, (seeie|seealsoie)*,
(secondaryie, (seeie|seealsoie|tertiaryie)*)*)>
```

Acknowledgments

I would like to express my best gratitude to Vladimir Gapeyev, Michael Levin, Makoto Murata, and Alain Frisch for precious comments and useful discussions. The paper was greatly improved by the comments made by the anonymous reviewers of POPL'04 and PLAN-X'04. This work was supported by Kayamori Foundation of Information Science Advancement.

References

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2003.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML™). `http://www.w3.org/XML/`, 2000.

[4] Luca Cardelli and Giorgio Ghelli. A query language for semistructured data based on the Ambient Logic. In *Proceedings of 10th European Symposium on Programming*, number 2028 in LNCS, pages 1–22, 2001.

[5] James Clark. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[6] James Clark and Makoto Murata. RELAX NG. `http://www.relaxng.org`, 2001.

[7] Sophie Cluet and Jérôme Siméon. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB)*, pages 118–135, 1998.

[8] Hubert Comon, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`, 1999.

[9] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql`.

[10] David C. Fallside. XML Schema Part 0: Primer, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`, 2001.

[11] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, 2001.

[12] Mary F. Fernández, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In Jan Van den Bussche and Victor Vianu, editors, *Proceedings of 8th International Conference on Database Theory (ICDT 2001)*, volume 1973 of *Lecture Notes in Computer Science*, pages 263–300. Springer, 2001.

[13] FormatData. RecipeML. `http://www.formatdata.com/recipeml/`.

[14] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, pages 137–146, 2002.

[15] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[16] Haruo Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[17] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(4), 2002. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.

[18] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.

[19] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, September 2000.

[20] Recordare LLC. MusicXML. `http://www.recordare.com/xml.html`.

[21] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *Proceedings of International Conference on Database Theory*, pages 64–78, 2003.

[22] Erik Meijer and Mark Shields. XMλ: A functional programming language for constructing and manipulating XML documents. Manuscript, 1999.

[23] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[24] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[25] Makoto Murata. Extended path expressions for XML. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, 2001.

[26] OASIS. DocBook. `http://www.docbook.org`.

[27] Newspaper Association of America. NAA Classified Advertising Standards Task Force. `http://www.naa.org/technology/clsstdtf/`.

[28] Akihiko Tozawa. Towards static type checking for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.

[29] W3C. Synchronized Multimedia Integration Language. `http://www.w3.org/AudioVideo/`.

[30] W3C. W3C XML Specification. `http://www.w3.org/XML/1998/06/xmlspec-report.htm`.

A Translation from filters to filter automata

In this section, we show an algorithm to translate a filter to a filter automaton. The correctness proof of the translation is omitted here.

Given a pattern definition E, a filter definition G, and a "starting" filter name Y_0, we create an automaton $A^{\mathbf{all}} = (Q_{A^{\mathbf{all}}}, Q^{\mathbf{init}}_{A^{\mathbf{all}}}, Q^{\mathbf{fin}}_{A^{\mathbf{all}}}, T_{A^{\mathbf{all}}})$ where

$$\begin{array}{rcl}
Q_{A^{\mathbf{all}}} & = & \bigcup_{X \in \mathbf{dom}(E)} (Q_{E(X),\emptyset} \cup \{q_X\}) \\
& \cup & \bigcup_{Y \in \mathbf{dom}(G)} (Q_{G(Y)} \cup \{q_Y\}) \\
Q^{\mathbf{init}}_{A^{\mathbf{all}}} & = & q_{Y_0} \\
Q^{\mathbf{fin}}_{A^{\mathbf{all}}} & = & \bigcup_{X \in \mathbf{dom}(E)} Q^{\mathbf{fin}}_{E(X),\emptyset} \cup \bigcup_{Y \in \mathbf{dom}(G)} Q^{\mathbf{fin}}_{G(Y),\emptyset} \\
T_{A^{\mathbf{all}}} & = & \bigcup_{X \in \mathbf{dom}(E)} (T_{E(X),\emptyset} \cup (\{q_X\} \stackrel{\epsilon}{\to} Q^{\mathbf{init}}_{E(X),\emptyset})) \\
& \cup & \bigcup_{Y \in \mathbf{dom}(G)} (T_{G(Y),\emptyset} \cup (\{q_Y\} \stackrel{\epsilon}{\to} Q^{\mathbf{init}}_{G(Y),\emptyset}))
\end{array}$$

and $A_{P,\overline{x}} = (Q_{P,\overline{x}}, Q^{\mathbf{init}}_{P,\overline{x}}, Q^{\mathbf{fin}}_{P,\overline{x}}, Q^{\mathbf{fin}}_{P,\overline{x}})$ and $A_F = (Q_F, Q^{\mathbf{init}}_F, Q^{\mathbf{fin}}_F, Q^{\mathbf{fin}}_F)$ are inductively defined as follows.

$$\begin{array}{rcl}
A_{\epsilon,\overline{x}} & = & \langle\{q_1\},\ \{q_1\},\ \{q_1\},\ \emptyset\rangle \\
A_{L[X],\overline{x}} & = & \langle\{q_1,q_2\},\ \{q_1\},\ \{q_2\},\ \{q_1 \xrightarrow{\overline{x}:L[q_X]} q_2\}\rangle \\
A_{(P_1\ P_2),\overline{x}} & = & \langle Q_{P_1,\overline{x}} \cup Q_{P_2,\overline{x}},\ \ Q^{\mathbf{init}}_{P_1,\overline{x}},\ \ Q^{\mathbf{fin}}_{P_2,\overline{x}},\ \ T_{P_1,\overline{x}} \cup \\
 & & T_{P_2,\overline{x}} \cup (Q^{\mathbf{fin}}_{P_1,\overline{x}} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{P_2,\overline{x}})\rangle \\
A_{(P_1\ |\ P_2),\overline{x}} & = & \langle Q_{P_1,\overline{x}} \cup Q_{P_2,\overline{x}},\ \ Q^{\mathbf{init}}_{P_1,\overline{x}} \cup Q^{\mathbf{init}}_{P_2,\overline{x}},\ \ Q^{\mathbf{fin}}_{P_1,\overline{x}} \cup \\
 & & Q^{\mathbf{fin}}_{P_2,\overline{x}},\ T_{P_1,\overline{x}} \cup T_{P_2,\overline{x}}\rangle \\
A_{P^*,\overline{x}} & = & \langle Q_{P,\overline{x}} \cup \{q_1,q_2\},\ \{q_1\},\ \{q_2\},\ T_{P,\overline{x}} \cup ((Q^{\mathbf{fin}}_{P,\overline{x}} \cup \\
 & & \{q_1\}) \xrightarrow{\epsilon} (Q^{\mathbf{init}}_{P,\overline{x}} \cup \{q_2\}))\rangle \\
A_{(P \text{ as } x),\overline{x}} & = & A_{P,\overline{x}\cup\{x\}}
\end{array}$$

$$\begin{array}{rcl}
A_{\epsilon} & = & \langle\{q_1\},\ \{q_1\},\ \{q_1\},\ \emptyset\rangle \\
A_{L[Y]} & = & \langle\{q_1,q_2\},\ \{q_1\},\ \{q_2\},\ \{q_1 \xrightarrow{L[q_Y]} q_2\}\rangle \\
A_{(F_1\ F_2)} & = & \langle Q_{F_1} \cup Q_{F_2},\ \ Q^{\mathbf{init}}_{F_1},\ \ Q^{\mathbf{fin}}_{F_2},\ \ T_{F_1} \cup T_{F_2} \cup \\
 & & (Q^{\mathbf{fin}}_{F_1} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{F_2})\rangle \\
A_{(F_1\ |\ F_2)} & = & \langle Q_{F_1} \cup Q_{F_2},\ Q^{\mathbf{init}}_{F_1} \cup Q^{\mathbf{init}}_{F_2},\ Q^{\mathbf{fin}}_{F_1} \cup Q^{\mathbf{fin}}_{F_2},\ T_{F_1} \cup \\
 & & T_{F_2}\rangle \\
A_{F^*} & = & \langle Q_F \cup \{q_1,q_2\},\ \{q_1\},\ \{q_2\},\ T_F \cup ((Q^{\mathbf{fin}}_F \cup \\
 & & \{q_1\}) \xrightarrow{\epsilon} (Q^{\mathbf{init}}_F \cup \{q_2\}))\rangle \\
A_{P\to e} & = & \langle Q_{P,\emptyset} \cup \{q_1,q_2\},\ \ \{q_1\},\ \ \{q_2\},\ \ T_{P,\emptyset} \cup \\
 & & (\{q_1\} \xrightarrow{\mathbf{in}(e)} Q^{\mathbf{init}}_{P,\emptyset}) \cup (Q^{\mathbf{fin}}_{P,\emptyset} \xrightarrow{\mathbf{out}} \{q_2\})\rangle
\end{array}$$

Here, $Q_1 \xrightarrow{\lambda} Q_2$ means $\{q_1 \xrightarrow{\lambda} q_2 \mid q_1 \in Q_1,\ q_2 \in Q_2\}$. Note that the construction for the empty sequence, labels, concatenations, alternations, and repetitions of both patterns and filters is exactly the same as the standard one [15].

B Complex Regular Expressions

In this section, we give a collection of regular expressions taken from real-world DTDs. These regular expressions have a certain complex structure that can defeat the use of a standard for-each style of language features mentioned in the introduction. Specifically, we choose the following characterization of such structure: a regular expression is *complex* if it has, as subexpression, either

- a concatenation of two expressions containing the same label, or
- a repetition containing a concatenation.

An example of the first is `((a,b),a)` and an example of the second is `(a,b)*`.

Below, we pick up six DTDs and quote their element and entity declarations with complex regular expressions. We omit the definitions of some of the entities referenced here when they are not important for the present purpose. The other entities are already expanded.

B.1 DocBook 4.2 [26]

The element declarations shown below can be found in the file `dbhierx.mod`. Almost the same structure as the element `appendix` is used for the elements `chapter`, `preface`, `section`, `sect1`, `sect2`, `sect3`, `sect4`, and `sect5` (omitted here).

```
<!ELEMENT appendix
    (beginpage?,
     appendixinfo?,
     (%bookcomponent.title.content;),
     (toc|lot|index|glossary|bibliography)*,
     tocchap?,
     (%bookcomponent.content;),
     (toc|lot|index|glossary|bibliography)*)>

<!ELEMENT indexentry
    (primaryie, (seeie|seealsoie)*,
     (secondaryie, (seeie|seealsoie|tertiaryie)*)*)>

<!ELEMENT refmeta
    ((indexterm)*,
     refentrytitle, manvolnum?, refmiscinfo*,
     (indexterm)*)>
```

B.2 MusicXML 0.8 [20]

The elements `key` and `time` are declared in the file `attributes.dtd`, the element `harmony` in `direction.dtd`, and the elements `ornaments` and `lyric` in `note.dtd`.

```
<!ELEMENT key
    ((cancel?, fifths, mode?) |
     ((key-step, key-alter)*))>

<!ELEMENT time
    ((beats, beat-type)+ | senza-misura)>

<!ELEMENT harmony
    (((root | function), kind,
     inversion?, bass?, degree*)+)>

<!ELEMENT ornaments
    (((trill-mark | turn | delayed-turn |
       shake | wavy-line | mordent |
       inverted-mordent | schleifer |
       other-ornament),
      accidental-mark*)*)>

<!ELEMENT lyric
    ((((syllabic?, text),
       (elision, syllabic?, text)*, extend?) |
      extend | laughing | humming),
     end-line?, end-paragraph?)>
```

B.3 RecipeML 0.5 [13]

```
<!ENTITY % amt.cont '(amt, (sep?, amt)*)'>

<!ENTITY % time.cont '(time, (sep?, time)*)'>

<!ENTITY % temp.cont '(temp, (sep?, temp)*)'>

<!ELEMENT equipment
    (equip-div+ | (note*, tool, (note | tool)*))>

<!ELEMENT equip-div
    (title?, description?, note*, tool,
     (note | tool)*)>
```

```
<!ELEMENT ingredients
    (ing-div+ | (note*, ing, (note | ing)*))>

<!ELEMENT ing-div
    (title?, description?, note*, ing,
     (note | ing)*)>

<!ELEMENT directions
    (dir-div+ | ((note | ing)*, step,
                 (note | ing | step)*))>

<!ELEMENT dir-div
    (title?, description?, (note | ing)*, step,
     (note | ing | step)*)>

<!ELEMENT amt
    ((qty | range)?, size?, unit?, size?)>
```

B.4 Adex 1.2 [27]

```
<!ELEMENT transfer-info
    (transfer-number, (from-to, company-id)+,
     contact-info)*>

<!ELEMENT days-and-hours
    (date, time)+>
```

B.5 SMIL 2.0 [29]

The following is in the file `smil-model-1.mod`.

```
<!ENTITY % SMIL.head.content
    (meta*,
     (customAttributes, meta*)?,
     (metadata, meta*)?,
     ((layout|switch), meta*)?,
     (transition+, meta*)?)>
```

B.6 W3C XML Specification 2.1 [30]

```
<!ELEMENT prod
    (lhs, (rhs, (com|wfc|vc|constraint)*)+)>
```

Regular tree language recognition with static information

Alain Frisch
Département d'Informatique
École Normale Supérieure, Paris, France
Alain.Frisch@ens.fr

Abstract

This paper presents our compilation strategy to produce efficient code for pattern matching in the CDuce compiler, taking into account static information provided by the type system. Indeed, this information allows in many cases to compute the result (that is, to decide which branch to consider) by looking only at a small fragment of the tree. Formally, we introduce a new kind of deterministic tree automata that can efficiently recognize regular tree languages with static information about the trees and we propose a compilation algorithm to produce these automata.

1 Introduction

Emergence of XML has given tree automata theory a renewed importance[Nev02]. Indeed, XML schema languages such as DTD, XML-Schema, Relax-NG describe more or less regular languages of XML documents (considered as trees). Consequently, recent XML-oriented typed programming languages such as XDuce [Hos00, HP02], CDuce [BCF03, FCB02], Xtatic [GP03] have type algebras where types denote regular tree languages. The type system of these languages relies on a subtyping relation, defined as the inclusion of the regular languages denoted by types, which is known to be a decidable problem (new and efficient algorithms have been designed for this purpose, and they behave well in practice).

An essential ingredient of these languages is a powerful pattern matching operation. A pattern is a declarative way to extract information from an XML tree. Because of this declarative nature, language implementors have to propose efficient execution models for pattern matching.

This paper describes our approach in implementing pattern matching in CDuce[1]. To simplify the presentation, the paper studies only a restricted form of pattern matching, without capture variable and with a very simple kind of trees. Of course, our implementation handles capture variables and the full set of types and patterns constructors in CDuce. In the simplified form, the pattern matching problem is a recognition problem, namely deciding whether a tree v belongs to a regular tree language X or not.

1. If the regular language is given by a tree automaton, a top-down recognition algorithm may have to backtrack, and the recognition time is not linear in the size of the input tree.

2. It is well-known that any tree automaton can be transformed into an equivalent bottom-up deterministic automaton, which ensures linear execution time; the size of the automaton may be huge even for simple languages, which can make this approach unfeasible in practice.

3. The static type system of the language provides an upper approximation for the type of the matched tree v, that is some regular language X_0 such that v is necessarily in X_0. Taking this information into account, it should be possible to avoid looking at some subtree of v. However, classical bottom-up tree automata are bound to look at the whole tree, and they cannot take this kind of static knowledge into account.

Let us give an example to illustrate the last point. Consider the following CDuce program:

[1] CDuce is available for download at http://www.cduce.org/.

```
type A = <a>[ A* ]
type B = <b>[ B* ]

let f ((A|B)->Int) A    ->0 | B ->1
let g ((A|B)->Int) <a>_->0 | _ ->1
```

The first lines introduce two types `A` and `B`. They denote XML documents with only `<a>` (resp. `<b>`) tags and nothing else. Then two functions `f` and `g` are defined. Both functions take an argument which is either a document of type `A` or of type `B`. They return 1 when the argument is of type `A`, and 0 when the argument is of type `B`. The declaration of `g` suggests an efficient execution schema: one just has to look at the root tag to answer the question. Instead, if we consider only the body of `f`, we have to look at the whole argument, and check that every node of the argument is tagged with `<a>` (resp. with `<b>`); whatever technique we use - deterministic bottom-up or backtracking top-down - it will be less efficient than `g`.

But if we use the information given by the function interface, we know that the argument is necessarily of type `A` or of type `B`, and we can compile `f` exactly as we compile `g`.

This example demonstrates that taking static information into account is crucial to provide efficient execution for declarative patterns as in `f`.

The main contribution of this paper is the definition of a new kind of deterministic bottom-up tree automata, called NUA (non-uniform automata) and a compilation algorithm that produces an efficient NUA equivalent to a given non-deterministic (classical) automaton, taking into account static knowledge about the matched trees.

Informally, non-uniform automata enrich classical bottom-up automata with a "control state". The control state is threaded through the tree, during a top-down and left-to-right traversal. In some cases, it is possible to stop the traversal of a whole subtree only by looking at the current state. Non-uniform automata combine the advantage of deterministic bottom-up and deterministic top-down tree automata, and they can take benefit from static information.

In order to discriminate several regular tree languages, a NUA does not necessarily need to consider a given subtree at all. The "magic" of the compilation algorithm is to compute a NUA that will extract only a *minimal* set of information from trees. More precisely, a NUA will skip a subtree if and only if looking at this tree does not bring any additional relevant information, considering the initial static information and the information already gathered during the beginning of the run.

Remark 1.1 *A central idea in XDuce-like languages is that XML document live in an untyped world and that XML types are structural. This is in contrast with the XML Schema philosophy, whose data model (after validation) attaches type* names *to XML nodes. Moreover, in XML Schema, the context and the tag of an element are enough to know the exact XML Schema type of the element. In XDuce-like languages, in general, one may have to look deep inside the elements to check type constraints. Our work shows how an efficient compilation of pattern matching can avoid this costly checks: our compilation algorithm detects when the context and the tag are enough to decide of the type of an element without looking at its content. This work supports the claim that a structural data model à la XDuce can be implemented as efficiently as a data model with explicit type names à la XML Schema.*

Related work Levin [Lev03] also addresses the implementation of pattern matching in XDuce-like programming languages. He introduces a general framework (intermediate language, matching automata) to reason about the compilation of patterns, and he proposes several compilation strategies. He leaves apart the issue of using static types for compilation, which is the main motivation for our work. So the two works are complementary: our compilation algorithm could probably be re-casted in his formalism.

Neumann and Seidl [NS98] introduce push-down automata to locate efficiently nodes in an XML tree. Our automata shares with push-down automata the idea of threading a control-state through the tree. The formalisms are quite different because we work with simpler kind of automata (binary trees with labeled leaves, whereas they have unranked labeled forests), and we explicitly distinguish betwen control states (threaded through the tree) and results (used in particular to update the state). However, using an encoding of unranked trees in binary trees, we believe that the two notions of automata are isomorphic. But again, they don't address the issue of using

static information to improve the automata, which is our main technical contribution. It should be possible to adapt our compilation algorithm to their push-down automata setting, but it would probably result in an extremely complex technical presentation. This motivates us working with simpler kinds of tree and automata.

2 Technical framework

In this section, we introduce our technical framework. We consider the simplest form of trees: binary trees with labeled leafs and unlabeled nodes. Any kind of ordered trees (n-ary, ranked, unranked; with or without labeled nodes) can be *encoded*, and the notion of regular language is invariant under these encodings. Using this very simply kind of trees simplifies the presentation.

2.1 Trees and classical tree automata

Definition 2.1 *Let Σ be a (fixed) finite set of symbols. A tree v is either a symbol $a \in \Sigma$ or a pair of trees (v_1, v_2). The set of trees is written $\mathscr{V}$.*

CDuce actually uses this form of trees to represent XML documents: forgetting about XML attributes, an XML element is represented as a pair $(tag, content)$ where tag is a leaf representing the tag and $content$ is the encoding of the sequence of children. The empty sequences is encoded as a leaf nil, and a non-empty sequence is encoded as a pair $(head, tail)$. We recall the classical definition of a tree automaton, adapted to our definition of trees.

Definition 2.2 (Tree automaton) *A* (non-deterministic) tree automaton (NDTA) *is a pair $\mathscr{A} = (R, \delta)$ where R is a finite set of* nodes*, and $\delta \subseteq (\Sigma \times R) \cup (R \times R \times R)$.*

Each node r in a NDTA defines a subset $\mathscr{A}[\![r]\!]$ of $\mathscr{V}$. These sets can be defined by the following mutually recursive equations:

$$\mathscr{A}[\![r]\!] = \{a \in \Sigma \mid (a, r) \in \delta\} \cup \bigcup_{(r_1, r_2, r) \in \delta} \mathscr{A}[\![r_1]\!] \times \mathscr{A}[\![r_2]\!]$$

We write $\mathscr{A}[\![r]\!]^2 = \mathscr{A}[\![r]\!] \cap \mathscr{V} \times \mathscr{V}$. By definition, a regular language is a subset of the form $\mathscr{A}[\![r]\!]$ for some NDTA $\mathscr{A}$ and some node r. We say that this language is *defined* by $\mathscr{A}$. There are two classical notions of deterministic tree automata:

- Top-down deterministic automata (TDDTA) satisfy the property: $\{(r_1, r_2) \mid (r_1, r_2, r) \in \delta\}$ has at most one element for any node r. These automata are strictly weaker than NDTA in terms of expressive power (they cannot define all the regular languages).
- Bottom-up deterministic automata (DTA) satisfy the property: $\{r \mid (r_1, r_2, r) \in \delta\}$ has at most one element for any pair of nodes (r_1, r_2), and similarly for the sets $\{r \mid (a, r) \in \delta\}$ with $a \in \Sigma$. These automata have the same expressive power as NDTA.

Remark 2.3 *We use the non-standard terminology of nodes instead of states. The reason is that we are going to split this notion in two: results and control states. Results will correspond to nodes in a DTA, and control states will correspond to nodes in TDDTA.*

In order to motivate the use of a different kind of automata, let us introduce different notions of context. During the traversal of a tree, an automaton computes and gathers information. The amount of extracted information can only depend on the context of the current location in the tree. A top-down recognizer (for TDDTA) can only propagate information downwards: the context of a location is thus the path from the root to the location ("upward context"). A bottom-up recognizer propagates information upwards: the context is the whole subtree rooted at the current location ("downward context").

Top-down algorithms are more efficient when the relevant information is located near the root. For instance, going back to the CDuce examples in the introduction, we see easily that the function `g` should be implemented by starting the traversal from the root of the tree, since looking only at the root tag is enough (note that because of the encoding of XML documents in CDuce, the root tag is actually the left child of the root). Patterns in CDuce tend to look in priority near the root of the trees instead of their leafs. However, because of their lack of expressive power, pure TDDTA cannot be used in general. Also, since they perform independant computations of the left and the right children of a location in a tree, they cannot use information gathered in the left subtree to guide the computation in the right subtree.

The idea behind push-down automata is to traverse each node twice. A location is first entered in a given context, some computation is performed on the subtree, and the location is entered again with a new context. When a location is first entered, the context is the path from the root, but also all the "left siblings" of these locations and their subtrees (we call this the "up/left context" of the location). After the computation on the children, the context also include the subtree. The notion of non-uniform automata we are going to introduce is a slight variation on this idea: a location is entered three times. Indeed, when computing on a tree which is a pair, the automaton considers the left and right subtree in sequence. Between the two, the location is entered again to update its context, and possibly use the information gathered on the left subtree to guide the computation on the right subtree.

This richer notion of context allows to combine the advantages of DTA and TDDTA, and more.

Due to lack of space, we cannot give more background information about regular language and tree automata. To understand the technical details that follow, some familiarity with the theory of tree automata is expected (see for instance [Nev02] for an introduction to automata theory for XML and relevant bibliographical references). However, we give enough intuition so that the reader not familiar with this theory can (hopefully) grasps the main ideas.

2.2 Non-uniform automata

We now introduce a new kind of tree automata: non-uniform automata (NUA in short). They can be seen as (a generalization of) a merger between DTA and TDDTA. Let us call "results" (resp. "control state") the nodes of DTA (resp. TDDTA). We are going to use these two notions in parallel. A current "control state" is threaded and updated during a depth-first left-to-right traversal of the tree (this control generalizes the one of TDDTA, where the state is only propagated downwards), and each control state q has its own set of results $R(q)$. Of course, the transition relation is parametric in q.

Remark 2.4 *We can see these automata as deterministic bottom-up automata enriched with a control state that makes their behavior change according to the current location in the tree (hence the terminology "non-uniform").*

When the automaton is facing a tree (v_1, v_2) in a state q, it starts with some computation on v_1 using a new state $q_1 = \texttt{left}(q)$ computed from the current one, as for a TDDTA. This gives a result r_1 which is immediately used to compute the state $q_2 = \texttt{right}(q, r_1)$. Note that contrary to TDDTA, q_2 depends not only on q, but also on the computation performed on the left subtree. The computation on v_2 is done from this state q_2, and it returns a result r_2. As for classical bottom-up deterministic automata, the result for (v_1, v_2) is then computed from r_1 and r_2 (and q).

Let us formalize the definition of non-uniform automata. We define only the deterministic version.

Definition 2.5 *A non-uniform automaton $\mathscr{A}$ is given by a finite set of states Q, and for each state $q \in Q$:*

- *A finite set of results $R(q)$.*
- *A state $\texttt{left}(q) \in Q$.*
- *For any result $r_1 \in R(\texttt{left}(q))$, a state $\texttt{right}(q, r_1) \in Q$.*
- *For any result $r_1 \in R(\texttt{left}(q))$, and any result $r_2 \in R(\texttt{right}(q, r_1))$, a result $\delta^2(q, r_1, r_2) \in R(q)$.*
- *A* partial *function $\delta^0(q, _) : \Sigma \to R(q)$.*

The *result* of the automaton from a state q on an input $v \in \mathscr{V}$, written $\mathscr{A}(q, v)$, is the element of $R(q)$ defined by induction on v:

$$\begin{array}{lcl} \mathscr{A}(q,a) & = & \delta^0(q,a) \\ \mathscr{A}(q,(v_1,v_2)) & = & \delta^2(q,r_1,r_2) \\ \quad\text{where} & & r_1 = \mathscr{A}(\texttt{left}(q),v_1) \\ & & r_2 = \mathscr{A}(\texttt{right}(q,r_1),v_2) \end{array}$$

Because the functions $\delta^0(q, _)$ are partial, so are the $\mathscr{A}(q, _)$. We write $\texttt{Dom}(q)$ the set of trees v such that $\mathscr{A}(q, v)$ is defined.

Remark 2.6 *Note that this definition boils down to that of a DTA when Q is a singleton $\{q\}$. The set of results of the NUA (for the only state) corresponds to the set of nodes of the DTA.*

It is also possible to convert a TDDTA to a NUA of the same size. The set of states of the NYA corresponds to the

set of nodes of the TDDTA, and all the states have a single result.

Our definition of NUAs (and more generally, the class of push down automata [NS98]) is flexible enough to simulate DTA and TDDTA (without explosion of size). They allow to merge the benefit of both kind of deterministic automata, and more (neither DTA nor TDDTA can thread information from a left subtree to the right subtree).

Neumann and Seidl [NS98] introduce a family of regular languages to demonstrate that their push-down automata are exponentially more succint than deterministic bottom-up automata. The same kind of example applies for NUAs.

A pair (q, r) with $q \in Q$ and $r \in R(q)$ is called a *state-result* pair. For such a pair, we write $\mathscr{A}[\![q; r]\!] = \{v \mid \mathscr{A}(q, v) = r\}$ for the set of trees yielding result r starting from initial state q. The reader is invited to check that a NUA can be interpreted as a non-deterministic tree automata whose nodes are state-result pairs. Consequently, the expressive power of NUAs (that is the class of languages of the form $\mathscr{A}[\![q; r]\!]$) is the same as NDTAs (ie: they can define only regular languages). The point is that the definition of NUAs gives an efficient execution strategy.

Running a NUA The definition of $\mathscr{A}(q, v)$ defines an effective algorithm that operates in linear time with respect to the size of v. We will only run this algorithm for trees v which are known *a priori* to be in $\mathtt{Dom}(q)$. This is because of the intended use of the theory (compilation of CDuce pattern matching): indeed, the static type system in CDuce ensures exhaustivity of pattern matching.

An important remark: the flexibility of having a different set of results for each state makes it possible to short-cut the inductive definition and completely ignore subtrees. Indeed, as soon as the algorithm reaches a subtree v' in a state q' such that $R(q')$ is a singleton, it can directly returns without even looking at v'.

Reduction A first criterion for a NUA to be good is the following condition:

Definition 2.7 *A NUA $\mathscr{A}$ is reduced if:*

$$\forall q \in Q.\ \forall r \in R(q).\ \exists v \in \mathtt{Dom}(q).\ \mathscr{A}(q, v) = r$$

This condition means that any result of any state can be reached for some tree in the domain of the state. Since the NUA will skip a subtree if and only if the set $R(q)$ is a singleton, it is important to enforce this property.

Of course, the set of reachable results can be computed (this amounts to checking emptiness of states of a NDTA, which can be done in linear time), and so we can remove unreachable results.

The point is that we are going to present a top-down compilation algorithm (it defines a NUA by giving explicitly for each state q the set of results $R(q)$ and the transition functions, see Section 3.8). Hence the produced NUA does not need to be fully built at compile time. Consequently, it is meaningful to say that this construction directly yields a reduced NUA, and does not require to fully materialize the automaton in order to remove unreachable results.

Tail-recursion Running a NUA on a tree requires a size of stack proportional to the height of the tree. This is problematic when dealing with trees obtained by a translation from, say, huge XML trees to binary trees. Indeed, this translation transforms long sequences to deep trees, strongly balanced to the right.

The following definition will help us in these cases.

Definition 2.8 *A NUA is* tail-recursive *if, for any state $q \in Q$:*

$$\forall r_1 \in R(\mathtt{left}(q)).\forall r_2 \in R(\mathtt{right}(q, r_1)).$$
$$\delta^2(q, r_1, r_2) = r_2$$

The idea is that a tail-recursive NUA can be implemented with a tail-recursive call on the right subtree. The stack-size used by a run of the NUA is then proportional to the largest number of "left" edges on an arbitrary branch of the tree.

The theorem above shows how to turn an arbitrary NUA into a tail-recursive one.

Theorem 2.9 *Let $\mathscr{A}$ be an arbitrary NUA, and $\mathscr{A}'$ be the NUA defined by:*

$$\begin{array}{l}
Q' = \{(q, q', \sigma) \mid q, q' \in Q, \sigma : R(q) \to R(q')\} \\
R((q, q', \sigma)) = \sigma(R(q)) \\
\mathtt{left}((q, q', \sigma)) = (\mathtt{left}(q), q, Id) \\
\mathtt{right}((q, q', \sigma), r_1) = (\mathtt{right}(q, r_1), q', \sigma \circ \delta^2(q, r_1, _)) \\
\delta^2((q, q', \sigma), r_1, r_2) = r_2 \\
\delta^0((q, q', \sigma), a) = \sigma \circ \delta^0(q, a)
\end{array}$$

Then:

- $\mathscr{A}'$ *is tail-recursive*
- *For any tree* v*:* $\mathscr{A}'((q,q',\sigma),v) = \sigma \circ \mathscr{A}(q,v)$
- *If* $\mathscr{A}$ *is reduced, then* $\mathscr{A}'$ *is also reduced.*

When the original automaton enters the right subtree, the set of results changes: a result returned by the computation on the right subtree will have to be translated to make it compatible with the set of result for the current location.

The idea is to push this translation in the computation on the right subtree. This is done by encoding the translation in the control state passed to the right subtree. In a triple (q, q', σ), the real "control-state" is q and σ encodes the translation from results of q to result of q'. This means that if (q, q', σ) is the current control state of the new NUA, then q would be the control state of the original NUA at the same point of the traversal, and σ would be the translation to be applied to the result by ancestors.

Remark 2.10 *If for some state q, any* $R(\texttt{right}(q, r_1))$ *is a singleton* $\{\sigma'(r_1)\}$*, it is possible to implement this state with a tail recursive-call on the left-subtree; in the construction above, we would take:* $\texttt{left}(q, q', \sigma) = (\texttt{left}(q), q', \sigma \circ \sigma')$

3 The algorithm

Different NUA can perform the same computation with different complexities (that is, they can ignore more or fewer subtrees of the input). To obtain efficient NUA, the objective is to keep the set of results $R(q)$ as small as possible, because when $R(q)$ is a singleton, we can drop the corresponding subtree.

Also, we want to build NUAs that take static information about the input trees into account. Hopefully, we have the opportunity of defining *partial* states, whose domain is not the whole set of trees.

In this section, we present an algorithm to build an efficient NUA to solve the dispatch problem under static knowledge. Namely, given $n + 1$ regular languages $X_0, \ldots, X_n$, we want to compute efficiently for any tree v in X_0 the set $\{i = 1..n \mid v \in X_i\}$.

3.1 Intuitions

Let us consider four regular languages X_1, X_2, X_3, X_4, and let $X = (X_1 \times X_2) \cup (X_3 \times X_4)$. Imagine we want to recognize the language X without static information ($X_0 = \mathscr{V}$). If we are given a tree (v_1, v_2), we must first perform some computation on v_1. Namely, it is enough to know, after this computation, if v_1 is in X_1 or not, and similarly for X_3. It is not necessary to do any other computation; for instance, we don't care whether v_1 is in X_2 or not. According to the presence of v_1 in X_1 and/or X_3, we continue with different computations of v_2:

- It v_1 is neither in X_1 nor in X_3, we already know that v is not in X without looking at v_2. We can stop the computation immediately.
- If v_1 is in X_1 but not in X_3, we have to check whether v_2 is in X_2.
- If v_1 is in X_3 but not in X_1, we have to check whether v_2 is in X_4.
- If v_1 is in X_1 and in X_3, we must check whether v_2 is in X_2 or not, and in X_4 or not. But actually, this is too much work, we only have to find whether it is in $X_2 \cup X_4$ or not, and this can be easier to do (for instance, if $X_2 \cup X_4 = \mathscr{V}$, we don't have anything to do at all).

This is the general case, but in some special cases, it is not necessary to know both wgther v_1 is in X_1 *and* whether it is in X_3. For instance, imagine that $X_2 = X_4$. Then we don't have to distinguish the three cases $v_1 \in X_1 \backslash X_3$, $v_1 \in X_3 \backslash X_1$, $v_1 \in X_1 \cap X_3$. Indeed, we only need to check whether v_1 is in $X_1 \cup X_3$ or not. We could as well have merged $X_1 \times X_2$ and $X_3 \times X_4$ into $(X_1 \cup X_3) \times X_2$ in this case. We can also merge them if one is a subset of the other.

Now imagine we have some static information X_0. If for instance, $X_0 \cap (X_1 \times X_2) = \varnothing$, we can simply ignore the rectangle $X_1 \times X_2$. Also, in general, we deduce some information about v_1: it belongs to $\pi_1(X_0) = \{v_1^0 \mid (v_1^0, v_2^0) \in X_0\}$. After performing some computation on v_1, we get more information. For instance, we may deduce $v_1 \in X_1 \backslash X_3$. Then we know that v_2 is in $\pi_2(X_0 \cap (X_1 \backslash X_3) \times \mathscr{V})$. In general, we can combine the

static information and the results we get for the a left subtree to get a better static information for the right subtree. Propagating a more precise information allows to ignore more rectangles.

The static information allows us to weaken the condition to merge two rectangles $X_1 \times X_2$ and $X_3 \times X_4$. Indeed, it is enough to check whether $\pi_2(X_0 \cap (X_1 \times X_2)) = \pi_2(X_0 \cap (X_3 \times X_4))$ (which is strictly weaker than $X_2 = X_4$).

In some cases, there are decisions to make. Imagine that $X_0 = X_1 \times X_2 \cup X_3 \times X_4$, and we want to check if a tree (v_1, v_2) is in $X_1 \times X_2$. If we suppose that $X_1 \cap X_3 = \varnothing$ and $X_2 \cap X_4 = \varnothing$, we can work on v_1 to see if is in X_1 or not, or we can work on v_2 to see if is in X_2 or not. We don't need to do both, and we must thus choose which one to do. We always choose to perform some computation on v_1 if it allows to gain useful knowledge on v. This choice allows to stop the top-down left-to-right traversal of the tree as soon as possible. This choice is relevant when considering the encoding of XML sequences and trees in our binary trees. Indeed, the choice correspond to: (1) extracting information from an XML tag to guide the computation on the content of the element, and (2) extracting information from the first children before considering the following ones.

3.2 Types

We have several regular languages $X_0, X_1, \ldots, X_n$ as inputs, and our algorithm needs to produce other languages as intermediate steps.

Instead of working with several different NDTA to define these languages, we assume that all the regular languages we will consider are defined by the same fixed NDTA $\mathscr{A}$ (each language is defined by a specific state of this NDTA). This assumption is not restrictive since it is always possible to take the (disjoint) union of several NDTA. Moreover, we assume that this NDTA has the following properties:

- **Boolean-completeness.** The class of languages defined by $\mathscr{A}$ (that is, the languages of the form $\mathscr{A}[\![r]\!]$), is closed under boolean operations (union, intersection, complement with respect to $\mathscr{V}$).
- **Canonicity.** If $(r_1, r_2, r) \in \delta$, then: $\mathscr{A}[\![r_1]\!] \neq \varnothing, \mathscr{A}[\![r_2]\!] \neq \varnothing$. Moreover, if we consider another pair $(r_1', r_2') \neq (r_1, r_2)$ such that $(r_1', r_2', r) \in \delta$, then $\mathscr{A}[\![r_1]\!] \cap \mathscr{A}[\![r_1']\!] = \varnothing$ and $\mathscr{A}[\![r_2]\!] \neq \mathscr{A}[\![r_2']\!]$.

It is well-known that the class of all regular tree languages is closed under boolean operations. The first property says that the class of languages defined by the fixed NDTA $\mathscr{A}$ is closed under these operations. Starting from an arbitrary NDTA, it is possible to extend it to a Boolean-complete one. If r_1, r_2 are two nodes, we write $r_1 \vee r_2$ (resp. $r_1 \wedge r_2$, $\neg r_1$) for some node r such that $\mathscr{A}[\![r]\!] = \mathscr{A}[\![r_1]\!] \cup \mathscr{A}[\![r_2]\!]$ (resp. $\mathscr{A}[\![r_1]\!] \cap \mathscr{A}[\![r_2]\!]$, $\mathscr{V} \backslash \mathscr{A}[\![r_1]\!]$).

The Canonicity property forces a canonical way to decompose the set $\mathscr{A}[\![r]\!]^2$ as a finite union of rectangles of the form $\mathscr{A}[\![r_1]\!] \times \mathscr{A}[\![r_2]\!]$. For instance, it disallows the following situation: $\{(r_1, r_2) \mid (r_1, r_2, r) \in \delta\} = \{(a, c), (b, c)\}$. In that case, the decomposition of $\mathscr{A}[\![r]\!]^2$ given by δ would have two rectangles with the same second component. To eliminate this situation, we can merge the two rectangles, to keep only $(a \vee b, c)$. We also want to avoid more complex situations, for instance where a rectangle in the decomposition of $\mathscr{A}[\![r]\!]^2$ is covered by the union of others rectangles in this decomposition. It is always possible to modify the transition relation δ of a Boolean-complete NDTA to enforce the Canonicity property (first, by splitting the rectangles to enforce non-intersecting first-components, and then by merging rectangles with the same second component).

We will use the word "type" to refer to the nodes of our fixed NDTA $\mathscr{A}$. Indeed, they correspond closely to the types of the CDuce (internal) type algebra, which support boolean operations and a canonical decomposition of products. Note that the set of types is finite, here. We write $[\![t]\!]$ instead of $\mathscr{A}[\![t]\!]$, $\Delta^2(t) = \{(t_1, t_2) \mid (t_1, t_2, t) \in \delta\}$, and $\Delta^0(t) = \{a \mid (a, t) \in \delta\}$. This allows us to reuse the symbols $\mathscr{A}, r, \ldots$ to refer to the NUA we will build, not the NDTA we start from.

3.3 Filters

Even if we start with a single check to perform ("is the tree in X?"), we may have several check to perform in parallel on a subtree ("is v_1 in X_1 and/or in X_3?"); we will call filter a finite set of checks to perform.

A filter is intended to be applied to any tree v from a given language; for such a tree, the filter must compute

which of its elements contain v.

Definition 3.1 *Let τ be a type. A τ-filter is a set of types ρ such that $\forall t \in \rho.\ [\![t]\!] \subseteq [\![\tau]\!]$.*
If $\rho' \subseteq \rho$, let $\rho'|\rho$ be a type such that:

$$[\![\rho'|\rho]\!] = [\![\tau]\!] \cap \bigcap_{t \in \rho'} [\![t]\!] \cap \bigcap_{t \in \rho \setminus \rho'} \mathscr{V} \setminus [\![t]\!]$$

(such a type exists thanks to Boolean-completeness.)
The result of a τ-filter ρ for a tree $v \in [\![\tau]\!]$, written v/ρ, is defined by:

$$v/\rho = \{t \in \rho \mid v \in [\![t]\!]\}$$

Equivalently, we can define v/ρ as the only subset $\rho' \subseteq \rho$ such that $v \in [\![\rho'|\rho]\!]$.

Our construction consists in building a NUA whose states are pairs (τ, ρ) of a type τ and a τ-filter ρ. Note that the set of all these pairs is finite, because we are working with a fixed NDTA to define all the types, so there is only a finite number of them.

3.4 Discussion

The type τ represents the static information we have about the tree, and ρ represents the tests we want to perform on a tree v which is known to be in τ. The expected behavior of the automaton is:

$$\forall v \in [\![\tau]\!].\ \mathscr{A}((\tau, \rho), v) = v/\rho$$

Moreover, the state (τ, ρ) can simply reject any tree outside $[\![\tau]\!]$. Actually, we will build a NUA such that:

$$\mathtt{Dom}((\tau, \rho)) = [\![\tau]\!]$$

The rest of the section describes how the NUA should behave on a given input. It will thus mix the description of the expected behavior of the NUA at runtime and the (compile-time) construction we deduce from this behavior.

Results In order to have a reduced NUA, we take for the set of results of a given state (τ, ρ) only the $\rho' \subseteq \rho$ that can be actually obtained for an input in τ:

$$R((\tau, \rho)) = \{\rho' \subseteq \rho \mid [\![\rho'|\rho]\!] \neq \varnothing\}$$

Note that ρ' is in this set if and only if there is a $v \in [\![\tau]\!]$ such that $v/\rho = \rho'$.

Left Assume we are given a tree $v = (v_1, v_2)$ which is known to be in a type τ. What can we say about v_1? Trivially, it is in one of the sets $[\![t_1]\!]$ for $(t_1, t_2) \in \Delta^2(\tau)$. We define:

$$\pi_1(\tau) = \bigvee_{(t_1,t_2) \in \Delta^2(\tau)} t_1$$

It is the best information we can find about v_1 [2]. Note that: $[\![\pi_1(\tau)]\!] = \{v_1 \mid (v_1, v_2) \in [\![\tau]\!]\}$.

Now assume we are given a τ-filter ρ that represents the tests we have to perform on v. Which tests do we have to perform on v_1? It is enough to consider those tests given by the $\pi_1(\tau)$-filter:

$$\pi_1(\rho) = \{t_1 \mid (t_1, t_2) \in \Delta^2(t),\ t \in \rho\}$$

This set is indeed a $\pi_1(\tau)$-filter. It corresponds to our choice of performing any computation on v_1 which can potentially simplify the work we have to do later on v_2. Indeed, two different rectangles in $\Delta^2(t)$ for some $t \in \rho$ have different second projections because of the Canonicity property.

This discussion suggests to take:

$$\mathtt{left}((\tau, \rho)) = (\pi_1(\tau), \pi_1(\rho))$$

Right Let us continue our discussion with the tree $v = (v_1, v_2)$. The NUA performs some computation on v_1 from the state (τ_1, ρ_1) with $\tau_1 = \pi_1(\tau)$ and $\rho_1 = \pi_1(\rho)$. Let ρ'_1 be the returned result, which is the set of all the types $t_1 \in \rho_1$ such that $v_1 \in [\![t_1]\!]$.

What can be said about v_2? It is in the following type:

$$\pi_2(\tau; \rho'_1) = \bigvee_{(t_1,t_2) \in \Delta^2(\tau)\ \mid\ [\![t_1 \wedge (\rho'_1|\rho_1)]\!] \neq \varnothing} t_2$$

This type represents the best information we can get about v_2 knowing that $v \in [\![\tau]\!]$ and $v_1 \in [\![\rho'_1|\rho_1]\!]$. Indeed, its interpretation is:

$$\{v_2 \mid (v_1, v_2) \in [\![\tau]\!], \rho'_1 = v_1/\rho_1\}$$

Now we must compute the checks we have to perform on v_2. Let us consider a given type $t \in \rho$. If $(t_1, t_2) \in \Delta^2(t)$, we have $t_1 \in \rho_1$, so we know if $v_1 \in [\![t_1]\!]$ or

[2] Here we use the assumption that the rectangles in the decomposition are not empty - this is part of the Canonicity property.

not (namely, $v_1 \in [\![t_1]\!] \iff t_1 \in \rho'_1$). There is at most one pair $(t_1, t_2) \in \Delta^2(t)$ such that $v_1 \in [\![t_1]\!]$. Indeed, two rectangles in the decomposition $\Delta^2(t)$ have non-intersecting first projection (Canonicity). If there is such a pair, we must check if v_2 is in $[\![t_2]\!]$ or not, and this will be enough to decide if v is in $[\![t]\!]$ or not. We thus take:

$$\pi_2(\rho;\rho'_1) = \{t_2 \mid (t_1,t_2) \in \Delta^2(t), t \in \rho, t_1 \in \rho'_1\}$$

The cardinal has at most as many elements as ρ by the remark above. Finally, the "right" transition is:

$$\texttt{right}((\tau,\rho),\rho'_1) = (\pi_2(\tau;\rho'_1), \pi_2(\rho;\rho'_1))$$

Computing the result We write $\tau_2 = \pi_2(\tau;\rho'_1)$ and $\rho_2 = \pi_2(\rho;\rho'_1)$. We can run the NUA from this state (τ_2,ρ_2) on the tree v_2, and get a result $\rho'_2 \subseteq \rho_2$ collecting the $t_2 \in \rho_2$ such that $v_2 \in [\![t_2]\!]$. For a type $t \in \rho$, and a rectangle (t_1,t_2) in its decomposition $\Delta^2(t)$, we have:

$$v \in [\![t_1]\!] \times [\![t_2]\!] \iff (t_1 \in \rho'_1) \wedge (t_2 \in \rho'_2)$$

So the result of running the NUA from the state (τ,ρ) on the tree v is:

$$\delta^2((\tau,\rho),\rho'_1,\rho'_2) = \{t \in \rho \mid \Delta^2(t) \cap (\rho'_1 \times \rho'_2) \neq \varnothing\}$$

Result for atomic symbols Finally, we must consider the case when the tree v is a symbol $a \in \Sigma$. The NUA has only to accept for the state (τ,ρ) trees in the set $[\![\tau]\!]$; so if $a \notin \Delta^0(\tau)$, we can let $\delta^0((\tau,\rho),a)$ undefined. Otherwise, we take:

$$\delta^0((\tau,\rho),a) = \{t \in \rho \mid a \in \Delta^0(t)\}$$

3.5 Formal construction

We can summarize the above discussion by an abstract construction of the NUA:

- the set of states are the pairs (τ,ρ) where τ is a type and ρ a τ-filter;
- $R((\tau,\rho)) = \{\rho' \subseteq \rho \mid [\![\rho'|\rho]\!] \neq \varnothing\}$;
- $\texttt{left}((\tau,\rho)) = (\pi_1(\tau),\pi_1(\rho))$ where: $\pi_1(\tau) = \bigvee\{t_1 \mid (t_1,t_2) \in \Delta^2(\tau)\}$ and $\pi_1(\rho) = \{t_1 \mid (t_1,t_2) \in \Delta^2(t),\ t \in \rho\}$;
- $\texttt{right}((\tau,\rho),\rho'_1) = (\pi_2(\tau;\rho'_1),\pi_2(\rho;\rho'_1))$ where: $\pi_2(\tau;\rho'_1) = \bigvee\{t_2 \mid (t_1,t_2) \in \Delta^2(\tau), [\![t_1 \wedge (\rho'_1|\rho_1)]\!] \neq \varnothing\}$ and $\pi_2(\rho;\rho'_1) = \{t_2 \mid (t_1,t_2) \in \Delta^2(t), t \in \rho, t_1 \in \rho'_1\}$;
- $\delta^2((\tau,\rho),\rho'_1,\rho'_2) = \{t \in \rho \mid \Delta^2(t) \cap (\rho'_1 \times \rho'_2) \neq \varnothing\}$;
- $\delta^0((\tau,\rho),a) = \{t \in \rho \mid a \in \Delta^0(t)\}$ if $a \in \Delta^0(\tau)$ (undefined otherwise)

Once again, it is out of question to actually materialize this NUA. Indeed, we are interested only in the part accessible from a given state (τ,ρ) corresponding to the pattern matching we need to compile. This abstract presentation has the advantage of simplicity (exactly as for the abstract subset construction for the determinization of automata).

Remark 3.2 *This construction has a nice property: the efficiency of the constructed NUA (that is, the positions where it will ignore subtrees of an input) does not depend on the type τ and the types in ρ (which are syntactic objects), but only on the languages denoted by these types. This is because of the Canonicity property. As a consequence, there is no need to "optimize" the types before running the algorithm.*

3.6 Soundness

The following theorem states that the constructed NUA computes what it is supposed to compute.

Theorem 3.3 *The above construction is well defined and explicitly computable. The resulting NUA is* reduced *and it satisfies the following properties for any state (τ,ρ):*

- $\mathit{Dom}((\tau,\rho)) = [\![\tau]\!]$
- $\forall v \in [\![\tau]\!].\ \mathscr{A}((\tau,\rho),v) = v/\rho$

The proof is by induction on trees, and follows the lines of the discussion above.

Remark 3.4 *It is possible to relax the Canonicity property for types and keep a sound compilation algorithm. However, optimality properties (Section 3.9) crucially depends on the simplifications dictated by the Canonicity property.*

3.7 An example

In this section, we give a very simple example of a NUA produced by our algorithm. We assume that Σ contains at least two symbols a,b and possibly others. We consider a type t_a (resp. t_b) which denotes all the trees with only a leaves (resp. b leaves). Our static information τ_0 is $t_a \vee t_b$, and the filter we are interested in is $\rho_0 = \{t_a, t_b\}$. Assuming proper choices for the NDTA that defines the types, the construction gives for the initial state $q_0 = (\tau_0, \rho_0)$:

- $R(q_0) = \{\{t_a\}, \{t_b\}\}$
- $\texttt{left}(q_0) = q_0$
- $\texttt{right}(q_0, \{t_a\}) = (t_a, \{t_a\})$
- $\texttt{right}(q_0, \{t_b\}) = (t_b, \{t_b\})$
- $\delta^2(q_0, \{t_a\}, \{t_a\}) = \{t_a\}$
- $\delta^2(q_0, \{t_b\}, \{t_b\}) = \{t_b\}$
- $\delta^0(q_0, a) = \{t_a\}$
- $\delta^0(q_0, b) = \{t_b\}$
- $\delta^0(q_0, c)$ undefined if $c \neq a, c \neq b$

There is no need to give the transition functions for the states $q_a = (t_a, \{t_a\})$ and $q_b = (t_b, \{t_b\})$ because they each have a single result ($R(q_a) = \{\{t_a\}\}$ and $R(q_b) = \{\{t_b\}\}$), so the NUA will simply skip the corresponding subtrees. Note that the NUA is tail-recursive. Its behavior is simple to understand: it goes directly to the leftmost leaf and returns immediatly. In particular, it traverses a single path from the root to a leaf and ignore the rest of the tree.

As another example, we can consider the functions `f` and g from the introduction, together with the CDuce encoding of XML documents. Our compilation algorithm indeed produces equivalent automata for the two pattern matchings: they directly fetch the root tag and ignore the rest of the tree.

3.8 Implementation

We rely a lot on the possibility of checking emptiness of a type ($[\![t]\!] = \varnothing$). For instance, the definition of $R((\tau, \rho))$ requires to check a lot of types for emptyness. All the techniques developed for the implementation of XDuce and CDuce subtyping algorithms can be used to do it efficiently. In particular, because of caching, the total cost for all the calls to the emptiness checking procedure does not depend on the number of calls (there is a single exponential cost), so they are "cheap" and we can afford a lot of them. CDuce also demonstrates an efficient implementation of the "type algebra" with boolean combinations and canonical decomposition.

The number of states (τ, ρ) is finite, but it is huge. However, our construction proceeds in a top-down way: starting from a given state (τ, ρ), it defines its set of results and its transitions explicitly. Hence we are able to build the NUA "lazily" (either by computing all the reachable states, or by waiting to consume inputs - this is how the CDuce implementation works).

We haven't studied the theoretical complexity of our algorithm, but it is clearly at least as costly as the inclusion problem for regular tree languages. However, in practice, the algorithm works well. It has been successfully used to compile non-trivial CDuce programs.

Preliminary benchmarks [BCF03] suggests very good runtime performances, and we believe that our compilation strategy for pattern matching is the main reason for that.

3.9 Optimality

Remember that one the advantages of NUAs over DTAs is that they can ignore a whole subtree of input when the set $R(q)$ for the current state q is a singleton. We would like to have some *optimality* for the NUA we have built, to be sure that no other construction would yield a more efficient NUA for the same problem. Due to the lack of space, and because this part is work in progress, we keep the presentation informal.

First, we make precise the notion of information. We say that an information is a partial equivalence relation (PER) $\equiv$ on $\mathscr{V}$ (that is, an equivalence relation whose domain $\texttt{Dom}(\equiv)$ is a subset of $\mathscr{V}$). We define an ordering on PERs. Let $\equiv_1$ and $\equiv_2$ two PERs. We say that the information $\equiv_2$ is larger than $\equiv_1$ and we write $\equiv_1 \leq \equiv_2$ if either:

- the domain of $\equiv_2$ is a strict subset of the domain of of $\equiv_1$: $\texttt{Dom}(\equiv_2) \subsetneq \texttt{Dom}(\equiv_1)$).

- or they have the same domain, and $\equiv_2$ is coarser than $\equiv_1$: $v_1 \equiv_1 v_2 \Rightarrow v_1 \equiv_2 v_2$.

Now we define the information of a state q in a NUA $\mathscr{A}$ as the PER $\equiv_q$ with domain $\texttt{Dom}(q)$ defined by:

$$v_1 \equiv_q v_2 \iff \mathscr{A}(q, v_1) = \mathscr{A}(q, v_2)$$

Let us give the intuition about the ordering on PERs. The idea is that the domain of a PER represents the information we have before doing any computation (static information), and the partition itself represents the information we extract by doing some computation on a tree (namely, finding the class of the PER the tree belongs to). We want to minimize both the static information we propagate and the amount of computation we require, so we want a NUA to traverse states with largest possible PERs in its traversal of a tree[3].

Here we need to take the traversal order into account, because we have made the following choice: when facing a tree $v = (v_1, v_2)$, the NUA we have built in previous sections extracts any information from v_1 that allows it to get more precise static information on v_2 (using the static information it has on v and the result of the computation on v_1).

For an arbitrary NUA $\mathscr{A}$, we have defined the result of $\mathscr{A}$ on an input v from a state q. In this section, we need to consider that running a NUA annotates the tree. For each subtree, we can define a state q such that the NUA entered this subtree in state q. We annotate the subtree with the PER associated to the state q. So we get a tree of PERs. We can flatten it using a right-to-left traversal (opposite to the operation of the NUA), to get a sequence of PERs (whose length correspond to the number of nodes and leaves in the tree v). We call it the trace of $(\mathscr{A}, q)$ on v.

We can compare the runs of two NUAs with initial states $(\mathscr{A}_1, q_1)$ and $(\mathscr{A}_2, q_2)$ (provided that the domains of the initial states contain v). We say that $(\mathscr{A}_1, q_1)$ is better than $(\mathscr{A}_2, q_2)$ for the input v, if the trace of $(\mathscr{A}_1, q_1)$ on v is larger than the trace of $(\mathscr{A}_2, q_2)$, for a lexicographic extension of the ordering on PERs (note that the two traces have the same length).

[3]Note that a state has a trivial PER (a partition with only one class) if and only if it has only one result (provided the NUA is reduced), which is the case that allows the NUAs to stop the traversal.

Now let τ be a type and ρ a τ-filter. Let $\mathscr{A}$ be an arbitrary NUA with an initial state q. We assume that this state extracts enough information from the inputs, as specified by the filter ρ. Formally, we assume the existence of a function $\sigma : R_2(q) \to \mathscr{P}(\rho)$ such that

$$\forall v \in [\![\tau]\!].\ \sigma(\mathscr{A}_2(q, v)) = \{t \in \rho \mid v \in [\![t]\!]\}$$

(or equivalently, that the PER associated to this state q is smaller than the one associated to the state (τ, ρ) in the constructed NUA). We say that $(\mathscr{A}, q)$ is correct for (τ, ρ). The optimality property can now be stated:

Claim 3.5 (Optimality) *Let $\mathscr{A}$ be the NUA built in the previous sections. For any tree $v \in [\![\tau]\!]$, the trace of $(\mathscr{A}, (\tau, \rho))$ on v is better than the trace of any other NUA which is correct for (τ, ρ).*

The proof should follow the lines of the discussion we used to establish the construction. However, we don't have a formal proof of this property yet. The intuition is that the NUA performs as much computation on a left subtree as necessary to get the most precise information on the right subtree (combining static information and the result on the left subtree) - but no more. So it is not possible, under the static knowledge at hand, to extract more static information about the rest of the tree in the traversal of the NUA. Having more information means having less computation to do on the rest of the tree, hence smaller numbers of possible results, and more opportunities for stopping the traversal early.

However, our ordering on PERs makes it a priority to maximize the static information, before minimizing the amount of computation to do. This corresponds to the choice in our algorithm to performs as much computation on a left subtree as necessary to get the most precise information on the right subtree (combining static information and the result on the left subtree). This is motivated by the fact that having more information means having less computation to do on the rest of the tree, hence more opportunities for stopping the traversal early.

But it is not always true that having strictly more information allows us to do strictly less computation, and this depends on the way the atomic cases (dispatch on the value of the leaves) are implemented. Let us give an example [4] . Let $\Sigma = \{a, b, c\}$ and $X_0 = \{a\} \times \Sigma \cup \{b\} \times$

[4]In this example, we manipulate subsets of Σ instead of types for simplicity.

$\{a, b\}$, $X_1 = \{a\} \times \{b, c\} \cup \{b\} \times \{b\}$. Given the static information X_0, we want to recognize X_1. The NUA that we have constructed will start on the left subtree with the filter $\{\{a\}, \{b\}\}$, that is, it wants to know if the left component in a or b (we are necessarily in one of these two cases because of X_0). If it is a, the static information about the right subtree is Σ, and the filter is $\{\{b, c\}\}$. If it is b, the static information about the right subtree is $\{a, b\}$, and the filter is $\{\{b\}\}$. Note that in both cases, it is enough to check if the right subtree is not a, so we're not doing less computation by distinguishing these two cases, even if we have more precise static information for the right subtree. It would be possible to avoid any computation on the left subtree because the information it gives cannot be used to improve the rest of the computation.

Note that this depends on low-level implementation details, namely the way to implement the dispatch for atomic symbols. It could be the case that indeed, the computation in the second case above is more efficient than the one in the first case (because of the representation of transition tables, ...), thus motivating the computation on the left subtree. This kind of situation occurs in the actual CDuce implementation, because of complex basic types (the analog of the symbols in Σ in this presentation): integer intervals, finite or cofinite sets of atoms, ... A more extensive discussion on this issue is left for a future publication.

4 Conclusion

In this paper, we have formalized the core of the compilation algorithm for pattern matching as implemented in CDuce. To simplify the presentation, we have considered only basic trees, and a pure recognition problem (no capture variable).

The actual implementation deals with:

- The full type algebra, including records, infinite basic types, and arrow types.
- The full patern algebra, with capture variables (including non-linear ones), default values, alternation and disjunction patterns, ...

We plan to report on the complete algorithm and study the optimality property in more details in a future publication.

Acknowledgments

I would like to express my best gratitude to Haruo Hosoya for his his help in improving the presentation of this paper.

References

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *ICFP*, 2003.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *LICS*, 2002.

[GP03] Vladimir Gapeyev and Benjamin Pierce. Regular object types. In *FOOL*, 2003.

[Hos00] Haruo Hosoya. Regular expression types for XML. *Ph.D thesis. The University of Tokyo*, 2000.

[HP02] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 2002.

[Lev03] Michael Levin. Compiling regular patterns. In *ICFP*, 2003.

[Nev02] Frank Neven. Automata theory for XML researchers. In *SIGMOD Record, 31(3), 2002.*, 2002.

[NS98] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, 1998. Extended abstract available at `http://www.informatik.uni-trier.de/~seidl/conferences.html`.

An XQuery-Based Language for Processing Updates in XML

Gargi M. Sur, Joachim Hammer, and Jérôme Siméon‡

CISE Department, University of Florida, *{gsur, jhammer}@cise.ufl.edu*
‡Bell Labs, Lucent Technologies, *simeon@research.bell-labs.com*

Abstract

Extensive usage of XML for information exchange and data processing has led to the development of standard languages for querying and publishing of XML data. However, a key problem in XML data management is the absence of a standard language to *update* XML. As a result, updates are carried out in a mostly ad-hoc fashion at the application level by writing code to modify tree-based structures representing XML documents in memory. So far very little research has been conducted on language-based updates and their impact on XML processing. In this paper we present UpdateX, our implementation of a declarative update language that can be used to directly update XML data using queries. Our update language is based on XQuery 1.0, W3C's XML query language, and seamlessly integrates with all of its constructs and capabilities. We describe the user-level syntax of our update language and present a framework for its implementation. The UpdateX prototype, which has been implemented on top of the Galax XQuery 1.0 processor, is fully functional and its source code distribution is publicly available on the Web.

1. Introduction

In the recent years, many applications have been migrated to XML to satisfy flexible data processing needs. As larger amounts of XML data must be processed and maintained efficiently, an expressive language to update XML becomes of primary importance. Update capabilities are necessary not only to modify existing XML documents but also to manage native XML repositories or XML data published from relational sources. However, there is no agreed upon XML update language yet and there is very little experience in building XML update processors. As a result, existing XML storage engines have little support for XML updates, and often rely on ad hoc solutions. Most XML applications use the XML Document Object Model[1] (DOM) programming interface to update XML. The DOM interface contains primitives which operate on a tree-based representation of the XML document or fragment in memory. Hence, programmers are subjected to writing code tailored to every XML document that must be processed. In the absence of a high-level, declarative update language, maintaining that code can also be tedious and error-prone. Finally, the lack of an easily accessible XML update infrastructure makes research about the impact of updates on XML processing difficult to carry out.

In this paper, we describe an *XML update language* that is *tightly integrated with XQuery 1.0* [BCF+], the W3C XML query language, and present an implementation of that language. The language itself is based on an internal W3C working draft [CFL+] that extends the powerful, declarative syntax of XQuery to provide update semantics. Thanks to its tight integration with XQuery, this language is powerful yet intuitive and easy to

[1] http://www.w3c.org/DOM/

learn. Moreover, an implementation can be obtained fairly rapidly by modifying an existing XQuery processor. We have developed a complete implementation of that language as part of the UpdateX project between University of Florida and Bell Laboratories. Our implementation has been carried out on top of the *Galax XQuery 1.0* engine. Galax and its source code, including the update implementation, are publicly available from the Galax Web site[2].

In this paper, we describe the update language itself using examples based on the XML Benchmark Project [AWK+], also known as XMark. The XMark benchmark proposes a set of queries operating on a document containing information about auctions, and has been widely accepted in the XML research community. The XMark queries have been carefully picked up in order to exercise various aspects of XML querying as is meant to evaluate the performances of XML query implementations for different application contexts. It is interesting to note here that the authors of XMark have cited the lack of support for updates as one of the prominent shortcomings of the current W3C XQuery specification. The demand for concurrent XML queries and updates is in fact not specific to the XMark scenario, but prevails in a number of other application domains, such as in Web services (e.g., an on-line travel agent must support both access to flight information and perform changes to reservations), in the context of the Semantic Web (to query and maintain an ontology written in RDF), in XML messaging (e.g., to route existing customer transactions or add annotations to them), or in XML publishing (e.g., to update an XML view over an underlying relational store).

Our work is based on an XML update language originally proposed in [CFL+], which addresses the issues discussed in this paper by providing the required update extensions on top of XQuery itself. Although a number of previous XML updates proposals have been made [TIH+,MR02,PL01], this language is the first that provides XML update support tightly coupled with XQuery. Our UpdateX prototype, which is based on Galax, validates the claim that this language can be easily incorporated into existing XML query engines, and is targeted to provide the desired DML functionalities in the above-mentioned and many other domains.

The rest of the paper is organized as follows. Section 2 gives and overview of the XML update language itself by means of examples. Section 3 presents the low-level XML update operations at the data model level, which are used to implement the language. Section 4 presents the rest of the architecture for UpdateX and details about the compilation process from the declarative XML updates to the low-level operations. Section 5 reviews related work and Section 6 concludes the paper.

2. The XML Update Language

The proposed XML update language borrows design principles from both XQuery and SQL. The main features of the language include statement-based update execution, use of XQuery expressions to compute target nodes and update content, constraint checking, complex updates, and snapshot semantics to enforce consistency of complex updates.

The primary building block in XQuery is an *expression*, which always evaluates to a value. XQuery uses various types of expressions such as path, sequence, arithmetic, logical, comparison, conditional, and *FLWOR* expressions. The XML update language,

---

[2] http://db.bell-labs.com/galax/

on the other hand, introduces the notion of *statement*, which is semantically different from the *expression* used in XQuery: an update statement modifies an existing value rather than simply returning one.

Updates are classified into *simple* and *complex updates*. Simple updates represent the basic data modification operations such as add, remove, or update. Complex updates can be either *conditional* or *iterative*, allowing complex operations based on simple updates. In the following subsections we highlight some of the most important features of our update language.

3.1. Simple Updates

Simple updates support either insertion of new XML fragments, deletions of existing XML fragments, or 'replacement' of an existing XML fragment by a new one. In each case, XQuery is used to compute the location where the update occurs and the content of the update. The syntax of the simple update statements is as follows:

- ***INSERT*** *UpdateContent InsertLocation TargetNode*
  where InsertLocation = ***INTO*** /***AS FIRST INTO / AS LAST INTO / BEFORE / AFTER***
- ***DELETE*** *TargetNode*
- ***REPLACE <value of>*** *TargetNode* ***WITH*** *UpdateContent*

For example, the simple insert statement inserts a copy of the item sequence returned by the 'UpdateContent' expression into the location determined by the 'TargetNode'. The order of the inserted nodes is specified using the 'InsertLocation' clause. The language provides several means to indicate the insert location, either as the children of a given node, or before/after a given node as a sibling. For example, a new item can be inserted into the XMark database using following insert statement:

```
insert
  <item id="{id}">
    <location>Brazil</location>
    <quantity>200</quantity>
    <name>XML in a Nutshell</name>
    <payment>Credit Card, Personal check</payment>
    <shipping>Will ship internationally</shipping>
    <incategory category="category1"/>
  </item>
as last into document(“xmark.xml”)/site/regions/samerica;
```

Update 1. Insert Statement.

Note that in this case the item is inserted as the last element. Also note that XQuery element node constructor is used to construct a new item element, and the parameter 'id' can be computed using XPath, e.g. $auction/regions//item[last()] + 1, where the $auction variable binds to the document node 'site'. Similarly the following delete statement can be used to delete the last bid of the second open auction:

```
delete
  $auction//open_auctions/open_auction[@id = "open_auction2"]/bidder[last()];
```

Update 2. Delete Statement.

Finally, the following replace statement updates the credit card information of person with reference identifier 10:

```
replace
  $auction/site/people/person[@id="person10"]/creditcard
with <creditcard>2334 3423 3484 2743</creditcard>;
```

Update 3. Replace Statement.

3.2. Complex Updates

Complex updates can be built from simple updates using either conditionals or FLWOR expressions. The syntax for conditional update and FLWUpdate is as follows:

- Conditional Update:
  UPDATE
  IF *(ConditionExpr)* ***THEN*** *SimpleUpdate1* ***ELSE*** *SimpleUpdate2*
- FLWUpdate:
  UPDATE
  *(**FOR**Clause|**LET**Clause)+ **WHERE**Clause? SimpleUpdate+*

A complex conditional update first computes the value of the 'ConditionExpr'. If the *Effective Boolean Value*[3] of ConditionalExpr is true, then the SimpleUpdate in the *then* clause is evaluated; otherwise, the SimpleUpdate in the *else* clause is evaluated. The 'then' and 'else' branches can also contain an empty update statement, which is written as "( )" and used to indicate that no data modification needs to be performed. The following conditional update statement replaces the category name of 'category4' if it exists or inserts a new category into the database as follows:

```
update
  if ($auction/site/categories/category[@id="category4"])
  then
    replace $auction/site/categories/category[@id="category4"]/name
    with <name>2003 Car Sales</name>
  else
    insert <category id="category4">
         <name>2003 Car Sales</name>
           <description>
             <parlist>
               <listitem>
               <text>New  <bold> Toyota Camry </bold> 2003</text>
             </listitem>
             </parlist>
           </description>
       </category>
    into $auction/site/categories;
```

Update 4. Conditional Update Statement.

Finally, for more complicated updates, the FLWUpdate statement can be used to apply simple updates through iterations. The *for* (and *let*) clause in the FLWUpdate allows the binding of a variable to the results of a query expression. The FLWUpdate then iterates

[3] The effective Boolean value is defined as the result of invoking the fn:boolean function on a XQuery sequence.

through this sequence of variable bindings to execute the list of simple updates sequentially. For instance, the following FLWUpdate can be used to insert a count of total number of bids for every open auction into the database:

```
update
  for $a in $auction/site/open_auctions/open_auction
    where fn:count($a/bidder) > 1
    insert <bid_count>{ fn:count($a/bidder) }</bid_count> as last into $a;
```

Update 5. FLWUpdate Statement.

3.3. Joins

The FLWUpdate can also compute joins over various parts of a document or across documents. For example, the following FLWUpdate performs a join between ‘person’ and ‘closed auction’ elements to compute the total purchase amount for each person and inserts it into his profile:

```
update
  for $p in $auction/site/people/person
  let $s := fn:sum(for $o in $auction/site/closed_auctions/closed_auction
                   where $o/buyer/@person=$p/@id
                   return $o/price)
    insert <purchase_history>{ $s }</purchase_history> into $p
```

Update 6. FLWUpdate Statement.

3.4. Constraints and Semantics

A set of basic semantic constraints must be preserved while executing updates. These constraints aim to preserve the logical structure of the data model instance. The most important constraints are as follows:

- The target node of a simple insert must be a single node. If the insert location evaluates to an empty value, or contains more than one node, then an error must be raised and the insertion is not performed.
- When the *into* clause is used in a simple insert, the target node must evaluate to document or element node; when the *after* or *before* clause is used, the target node must be an element, comment, or processing instruction node.
- During insertion or execution of a replace statement, for each adjacent sequence of one or more atomic values, a new text node is constructed. This new text node contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values.
- After insertion or deletion, adjacent text nodes must be coalesced into a single text node by concatenating their contents, with no intervening blanks.
- While replacing the value of an element or document node, the update content must be a *content sequence*. A content sequence is any sequence of zero or more element nodes, atomic values, processing instruction, and comment nodes.

In addition to these basic constraints, the data model must satisfy additional constraints in the case the input XML document has a DTD or XML Schema associated with it. Using XML schema one can describe key and referential integrity constraints, and

restrictions on the values of attributes and elements. Such *value-based* constraints must be handled before the updates are processed. Most of the value-based constraints can be checked explicitly using the conditional update statement. Note that a promising direction would be to hook up the update processor with an incremental constraint checker, such as ΔX [BBG+], which generates code based on first order predicate logic that can be used to check constraints on the fly while processing updates.

One of the main difficulties in designing and implementing such an XML update language is to ensure its semantic integrity. Most notably, one must carefully deal with cases where consecutive updates in a single FLWUpdate statement impact the same XML nodes. To avoid inconsistent results, the language imposes a so-called "snapshot semantics". According to snapshot semantics, *all the variables in the 'for' and 'let' clauses of FLWUpdate must be bound with respect to the initial snapshot before the simple updates in the body of the FLWUpdate are executed*. The simple update statements are then executed sequentially based on the initial snapshot and evaluated independently of each other. In the future, a complete specification for XML updates using XQuery will be published by the W3C XML Query Group, highlighting all the necessary constraints and semantics.

3. The UpdateX Data Model API

We now describe the underlying data model and its API which is based on the XQuery data model. Again, all examples are drawn from the XMark benchmark application.

3.1 The XQuery Data Model

In XQuery and XPath, every XML document instance is represented using a set of nodes. The XQuery 1.0 and XPath 2.0 data model specification [FMM+] defines seven different kinds of *nodes*: document, element, attribute, namespace, comment, processing instruction, and text. These nodes are used to capture the abstract, logical structure of a document, which is known as its *data model*. Every node in the data model is assigned a unique identifier by the query processor. These identifiers are used to maintain the original document order amongst the data model nodes. The query specification also defines various terms to describe content type such as atomic value, item, sequence, etc. *Atomic value* defines a value corresponding to the XML Schema atomic type (a simple type). An *item* is either a node or atomic value. A *sequence* is an ordered collection of zero or more items.

The XMark database, which we have used in our examples, is modeled after an Internet auction site. The main entities are: item, open auction, closed auction, person, and category. An item is an object on sale. Every item is assigned a unique identifier. An open auction has properties like privacy status, bid history, along with references to the bidders and sellers, the current bid and default increase, the type of auction, status of transaction, and a reference to the item being sold. Closed auctions describe completed auctions and include seller and buyer references, sold item reference, date when the transaction was closed, its type, and the annotations after the bidding process. The person elements are characterized by personal details, credit card number, profile of their interests, and a set of open auctions they watch. Categories feature name and descriptions used to classify items while a category-graph links categories into a network.

Figure 1 shows the data model for the complete XMark document. As one can see, the root entity site is represented by a top-level document node while the remaining entities and their properties are represented using nested element and associated attribute nodes.

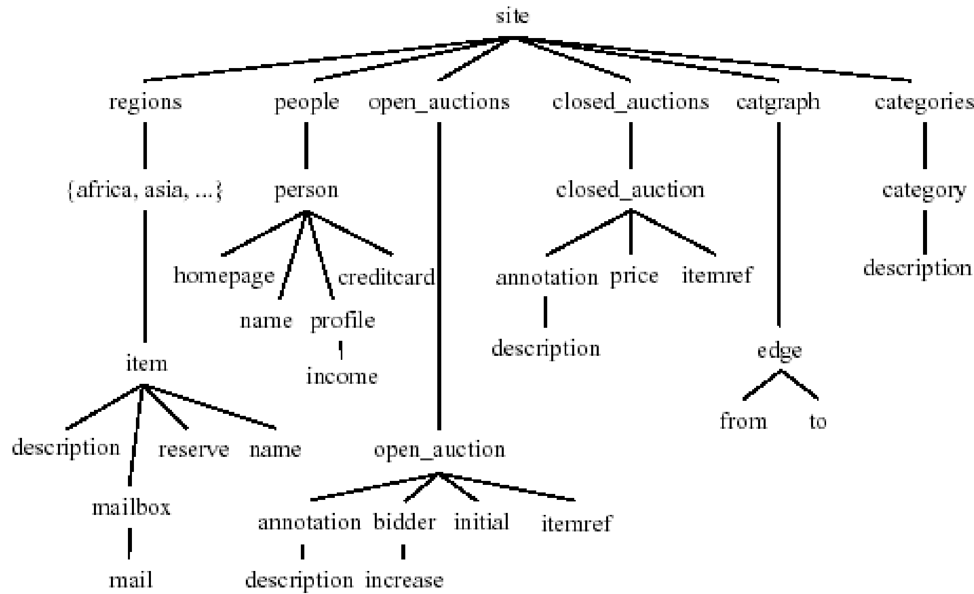


Figure 1. Data model of XMark Auction Database

3.2 Data Model Update Primitives

We require two data interfaces for the UpdateX language. Firstly, basic primitives to create and access the data model elements, and secondly, update primitives to execute basic data-modification operations such as insert, delete, and replace. Specifically, the update API has the following interfaces:

- *Insert(Sequence update-content, Node parent-node, Node sibling-node).* The insert primitive adds new nodes into the data model which are inserted as children of the parent node and can be ordered with respect to a sibling node.
- *Delete(Node target-node)*. The delete primitive removes given target node from the data model.
- *ReplaceNode(Sequence update-content, Node target-node, Node parent-node).* This particular replace primitive replaces a given target node with one or more new nodes and updates the parent node pointers.
- *ReplaceValue(Sequence update-content, Node target-node).* This replace primitive replaces the value of given target node with one or more nodes given in the update content.

Other useful update operations are move and rename [TIH+, MR02], but we do not include them in our API as they can be simulated using basic operations. For example, the move operation which moves an existing sub-tree from one location to a new one can be simulated by an insertion of a copy of the sub-tree at the new location followed by a

deletion of the original sub-tree. Similarly, the rename operation is a specific case of the replace, i.e., replacement of qualified name of a data model node. We have implemented the update primitives using tree-based, recursive algorithms. These primitives are available to the query engine in the form of library functions in Galax.

4. Implementation of the UpdateX Language

We have implemented a fully functional prototype to serve as a reference implementation of the UpdateX language. The prototype was implemented on top the Galax query engine [FSB+] developed at Bell Laboratories. Galax is an open source implementation of XQuery 1.0 and closely conforms to the XQuery specification suite. It implements most of the features of XQuery like path and FLWOR expressions, arithmetic and comparison operators, node constructors, document order, etc. It also supports several advanced features namely XML Schema import and validation, static type checking, and type/value based optimization. Galax is written in OCaml and is portable to various platforms (Linux, Solaris, Macintosh, and Windows). OCaml [XL02] is an object-oriented variant of ML with sophisticated features, like typed compilation, automatic memory management, polymorphism and abstraction. It is an ideal language to implement XQuery as well as update extensions, as its algebraic types and higher-order functions help to simplify the symbolic manipulation that is central to query transformation, analysis, and optimization.

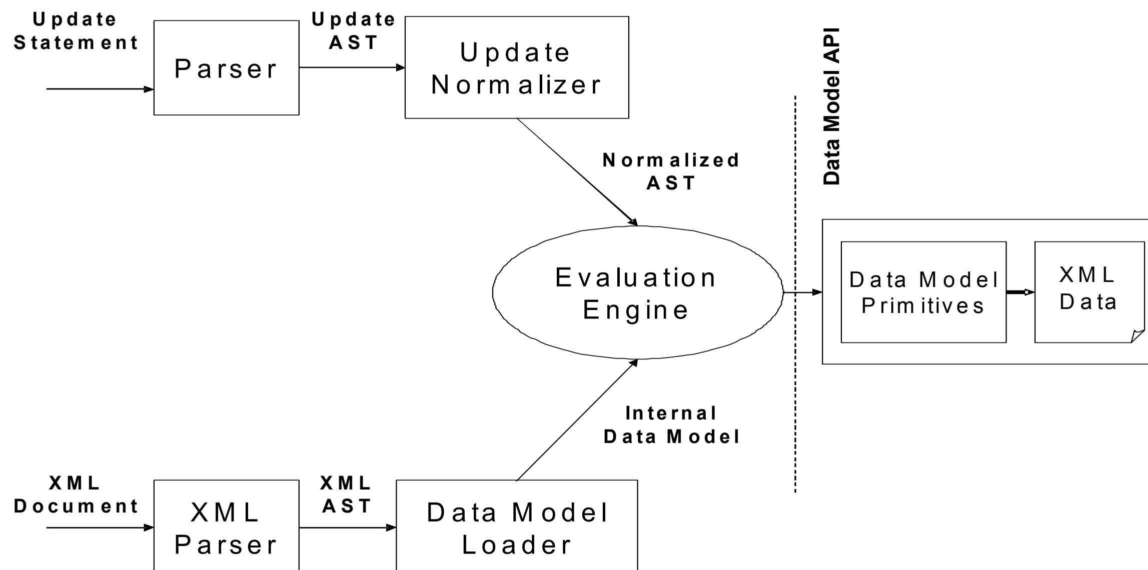


Figure 2. Architecture of the Update Processor.

The update processing model is based on the XQuery Formal Semantics [DFF+]. The architecture of the UpdateX processor is shown in Figure 2. The inputs to the processor consist of update statements and one (or more) XML document(s) that are subject to modification. The system consists of three main layers: parsing, normalization and execution. We explain the main features in more detail below.

4.1. Parsing Layer

The parsing layer implements the parsing phase of the XQuery processing model. It takes the inputs, parses them, and builds abstract syntax trees (AST) corresponding to the inputs. The parsing layer consists of two modules: update parser and XML parser. The *update parser* parses the input update statement to build an update AST. Updates are parsed according to the grammar rules specified in the Appendix. The input XML document is processed by a SAX-based *XML parser*. The advantage of using a SAX parser is that the AST corresponding to the XML document is never materialized. The SAX parser optimizes memory usage by generating stream of SAX events which are directly consumed by the *data model loader* to create an instance of the XQuery data model in memory.

4.2. Normalization Layer

The *update normalizer* implements the normalization phase of the XQuery processing model. It maps the update AST into an internal representation that can be directly executed by the evaluation engine. XQuery normalization judgments are applied to transform the update AST to equivalent XQuery Core expressions. Normalization judgments are transformations or rewriting rules that expand the abbreviated, hidden, or implied syntax of the top-level language and make them explicit.

For instance, the FLWUpdate syntax allows one or more For- (and Let) clauses in the same statement. During normalization, every FLWUpdate statement containing nested For- (and Let) clauses is normalized separately. Also, a For-clause in a FLWUpdate can contain more than one variable, where as a core expression can bind and iterate over only one variable. In such a case the FLWUpdate is normalized to sequence of nested For-expressions with a single variable binding.

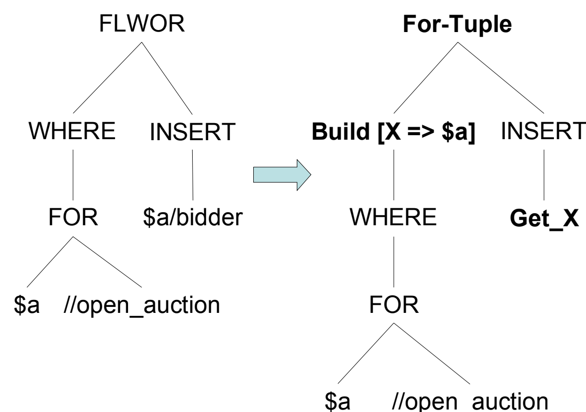


Figure 3. Normalization process for FLWUpdate

In addition to the above judgments, a special tuple construction rule is applied that helps to enforce snapshot semantics. Consider the FLWUpdate described in Update 5. When the FLWUpdate executes, the For-expression is evaluates to a *sequence of tuples,* which is used to evaluate expressions in the simple update list. This implies that a *tuple* (structure) must be present in the update AST to hold the tuple sequence values. When

the FLWUpdate is normalized, we insert a "for-tuple" construct into the update AST, which is later materialized during execution phase. The normalization process is shown in Figure 3. The "Build" and "Get_X" clauses represent internal tuple construction and extraction operations respectively.

4.3. Execution Layer

The execution layer implements the dynamic evaluation phases of the processing model. After an input update statement is parsed and normalized, it is executed by the evaluation engine. The update execution consists of three logical phases: pre-update processing, semantic checking, and update evaluation, which are shown in Figure 4 below.

4.3.1. Pre-update Processing

In the pre-update processing phase, all the XQuery expressions in the normalized AST are evaluated. A pre-update list is used in the algorithm, which holds the values of the query expressions based on the initial snapshot. For example, when FLWUpdate is pre-processed, the following steps are performed:

- The variable bindings in the 'for' and 'let' clauses of the update statement are computed by evaluating the query expressions and stored in a tuple sequence. These bindings actually define the scope of the update.
- The evaluation engine iterates through the list of simple updates and evaluates the sub-expressions in each simple update statement, with respect to the values in the tuple sequence.
- The resulting values are stored in a pre-update list according to the order of update statements in the simple update list.

For instance, the statement in Update 5 is normalized to the following core expression:

```
update
    for-tuple $dot in
      (for $a in $auction/site/open_auctions/open_auction
       return [ a: $a ])
    insert <bid_count>{fn:count($dot#a/bidder)}</bid_count> as last into $a;
```

Normalized Update 5. Core expression.

When this expression is executed, the 'for' expression is evaluated first. The intermediate results are stored in the *for-tuple* data structure that is inserted in the AST during the normalization phase. Next, the values of the 'bid-count' element for each iteration of '$a' variable in the simple insert is computed and stored in a pre-update list.

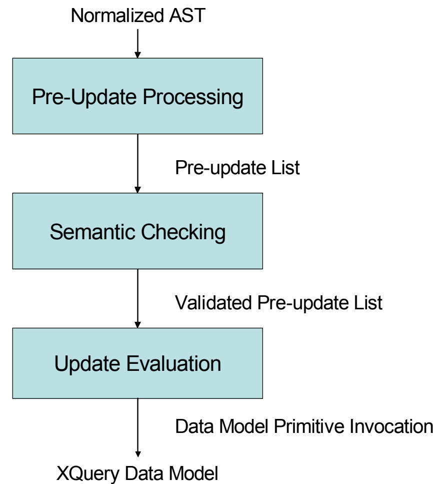


Figure 4. Update Execution Phases

4.3.2. Semantic Checking

In this step, semantic checks are applied to ascertain whether the target nodes and update content values are valid and whether basic constraints are satisfied. The evaluation engine performs necessary checks on the values in the pre-update list. During this validation step, if any constraint is not met, the execution is aborted and an update error is raised.

4.3.3. Update Evaluation

In the final step, updates are executed by invoking the data model primitives. Actual modifications to the underlying data model are performed by these primitives. Based on the type of update, the evaluation engine invokes the required primitives and passes the values stored in the validated pre-update list as arguments. The invoked primitive modifies the data model and returns control to the evaluation engine for further processing. After the update operations are completed, the update processor may perform additional book-keeping tasks such as updating the node identities and indexes if required.

The data model primitives are implemented as atomic functions. Here we discuss the implementation of the insert primitive. Its signature is given below:

```
Insert(Sequence update-content, Node parent-node, Node sibling-node)
```

The *update-content* consists of a sequence of items, which are inserted as children of the parent node. The insert operation is deterministic. It preserves the order of children nodes in the parent node. A pseudo-code version of the insert algorithm is as follows:

- Create a deep copy of the sequence of items to be inserted, viz., copy of update-content.
- Check the node kind of the parent; the parent node should be a document or an element node; otherwise raise an error.
- Compute the child axis of the parent node to retrieve its children.

- If a sibling node is specified, determine its position in the children node sequence.
- Compute the insert location as follows: if the sibling node parameter is null, then the insert location is after the last child of the parent node; otherwise the insert location is before the current position of the sibling node in document order.
- Insert the items in the update-content recursively at the insert location.
- Update the parent pointers of the newly inserted items.

It is important to note that the syntax for simple insert allows five *InsertLocation* clauses: insertion into, as last, as first, before, or after the target node. Although this may indicate the need for multiple insert primitives, we have implemented a single primitive that simulates all the required variations. The evaluation engine computes the values of the parent node and target node based on the *InsertLocation* clause, and then determines if the sibling node value is to be computed. These values are passed to the insert primitive when it is invoked during update execution.

The distinction made between low-level data model operations and a higher-level update evaluation provides a degree of independence with respect to the physical implementation of the update processor. The data model primitives can be implemented using in-memory, tree-based algorithms (this prototype), on top of indexed XML structures in a native XML repository, or by converting them into SQL DML queries which can be issued to a relational database system. For example, the delete primitive can be implemented as a pre-delete SQL trigger that can be fired when parent-child nodes of an XML document are stored in the form of tuples in a relational database [SGT+] and a parent tuple is deleted. The body of the trigger contains SQL code to delete the relevant tuples from the child relations. Moreover, since data model operations execute atomically, it becomes possible to define transactions over the XML data model by enhancements to the update processing model.

5. Related Work

Until recently, there as been relatively little work on XML updates. One of the first languages for XML updates was proposed by Tatarinov et al in [TIH+]. They proposed a fairly simple update language not directly integrated with XQuery, and whose expressiveness is limited to the basic updates supported by UpdateX at the data model level. In [LM00], Laux and Martin proposed an ad hoc language using XPath 1.0 to support XML updates in the XML:DB native XML database. Many of the updates presented in Section 3 cannot be expressed with those languages. Our work relies on a more recent proposal by the XML query working group [CFL+], which itself was inspired by proprietary proposals in [MR02] and [PL01]. An important limitation of the language is the lack of compositionality between the update expressions and the rest of the language. Even though powerful, the update primitives can only occur as top-level statements, in a way similar to SQL. This has to be distinguished from 'updates', or in-place modifications as present in most modern functional programming languages, such as ML [LW91]. Trying to support this kind of feature raises difficult questions in terms semantics and optimizations. We believe this is an important requirement for complex

XML applications, notably for Web services, and we hope to work on XML updates as first-class expressions in the language in the future.

6. Conclusion

We have described a new approach to XML update processing that uses a declarative, functional language based on XQuery 1.0. The language is easy to learn and powerful. It consists of various constructs, such as simple insert, delete, replace, conditional, and FLWUpdate statements, which can be used to efficiently modify XML data model instances. These constructs have been explained with examples from the XMark auction database. We also describe a complete framework for processing and implementation.

One of the most important contributions of the processing framework is that it eliminates tedious and ad hoc programming techniques (e.g., updating XML using DOM primitives) that are currently used to update XML documents. In addition, application developers benefit from use of an integrated query and update language to perform many different types of retrievals and modifications on XML documents (e.g. embedded XQuery queries or statements) in their software applications. Moreover, in this research, we have developed a vendor-independent, standard API to update native and relational XML repositories, which is beneficial to the database community at large. Lastly, for researchers, we provide a flexible architecture to study the impact of updates on XML processing and scope for refinement or improvement of update processing algorithms.

References

[AWK+] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse, "XMark: A Benchmark for XML Data Management," in *Proceedings of International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, pages 974-985, August 2002.

[BBG+] Micheal Benedikt, Glenn Bruns, Julie Gibson, Robin Kuss, and Amy Ng, "Automated Update Management for XML Integrity Constraints", PLAN-X 2002, Pittsburg, PA, October 2002.

[BCF+] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and J. Siméon, "XQuery 1.0: An XML Query Language," World Wide Web Consortium, W3C Working Draft, 22 August 2003.

[CFL+] Don Chamberlin, Daniela Florescu, Patrick Lehti, Jim Melton, Jonathan Robie, Michael Rys, and Jerome Simeon . Updates for XQuery. W3C working draft. October 2002, Unpublished.

[DFF+] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jerome Simeon, Philip Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics", W3C Working Draft, 22 August 2003.

[FMM+] M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model," W3C Working Draft, 2 May 2003.

[FSC+] Mary Fernandez, Jerome Simeon, Byron Choi, Amelie Marian, Gargi Sur, "Implementing XQuery 1.0: The Galax Experience", in Proceedings of VLDB 2003, Berlin, Germany, September 2003.

[LM00] Andreas Laux and Lars Matin. XUpdate working draft. October 2000. `http://www.xmldb.org/xupdate`

[LW91] Xavier Leroy, Pierre Weis: Polymorphic Type Inference and Assignment. POPL 1991: 291-302.

[MR02] Michael Rys, “Proposal for an XML Data Modification Language,” Microsoft Corp., Redmond, WA, Proposal May 2002.

[PL01] Patrick Lehti, “Design and Implementation of a Data Manipulation Processor for an XML Query Processor,” Technical University of Darmstadt, Darmstadt, Germany, Diplomarbeit, August 2001.

[SGT+] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. F. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities.” In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB’99)*, Morgan Kaufmann, pages 302-304, Sept. 1999.

[TIH+] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld, “Updating XML.” In *Proceedings of ACM SIGMOD Conference*, Santa Barbara, California, pages 413-424, May 2001.

[XL02] X. Leroy, “The Objective Caml system, release 3.06, Documentation and User’s Manual.” Institut National de Recherche en Informatique et en Automatique (INRIA), August 2002.

Appendix UpdateX Grammar in EBNF

```
[1]  Update             ::= SimpleUpdate | ComplexUpdate
[2]  SimpleUpdate       ::= Insert | Delete | Replace | EmptyUpdate
[3]  ComplexUpdate      ::= FLWUpdate | ConditionalUpdate
[4]  FLWUpdate          ::= "update" (ForClause | LetClause)+ WhereClause?
                              SimpleUpdate+
[5]  ConditionalUpdate  ::= "update" <"if" "("> ConditionExpr ")"
                              "then" SimpleUpdate "else" SimpleUpdate
[6]  ConditionExpr      ::= XQueryExpr
[7]  Insert             ::= "insert" UpdateContent InsertLocation
[8]  InsertLocation     ::= ((<"as" "last"> | <"as" "first">)? "into" TargetNode_
                              | ("after" TargetNode)
                              | ("before" TargetNode)
[9]  UpdateContent      ::= XQueryExpr
[10] TargetNode         ::= XQueryExpr
[11] Replace            ::= "replace" <"value" "of">? TargetNode
                              "with" UpdateContent
[12] EmptyUpdate        ::= "(" ")"
[13] Delete             ::= "delete" TargetNodes
[14] TargetNodes        ::= XQueryExpr
```

Note: XQueryExpr refers to the Expr [40] grammar production in the XQuery specification.

Efficient XPath Axis Evaluation for DOM Data Structures

Jan Hidders Philippe Michiels
University of Antwerp
Dept. of Math. and Comp. Science
Middelheimlaan 1, BE-2020 Antwerp, Belgium,
{jan.hidders,philippe.michiels}@ua.ac.be

Abstract

In this article we propose algorithms for implementing the axes for element nodes in XPath given a DOM-like representation of the document. Each algorithm assumes an input list that is sorted in document order and duplicate-free and returns a sorted and duplicate-free list of the result of following a certain axis from the nodes in the input list. The time complexity of all presented algorithms is at most $O(l + m)$ where l is the size of the input list and m the size of the output list. This improves upon results in [4] where also algorithms with linear time complexity are presented, but these are linear in the size of the entire document whereas our algorithms are linear in the size of the intermediate results which are often much smaller.

1 Introduction

The XQuery Formal Semantics [3] requires that the result of an XPath path expression returns a list of document nodes that is sorted in document order and duplicate free. This can be achieved by always sorting the list at the end of the evaluation of the path expression or even sorting after each step in the path expression. The first approach may seem more efficient but as shown in [4] can lead to an exponential blow-up of the intermediate results in the size of the query. The second approach, however, has the drawback that the sorting operations become a major bottleneck in the evaluation of the expression. One way to improve this situation is presented in [8] where given a straightforward implementation of the axes unnecessary sorting operations are detected and removed. In this paper we investigate the possibilities for alternative implementations of the axes such that these use the fact that the previous intermediate result is sorted and return a result that is always sorted and duplicate-free. For this purpose we will assume that the document is stored in a DOM-like pointer structure [1] and that the nodes are numbered with their so-called pre-numbers and post-numbers, i.e., their position in a preorder and postorder tree-walk, respectively.

2 Related Work

Previous research on the complexity of XPath evaluation has shown us that it is possible to construct efficient algorithms for the evaluation of XPath [5, 9]. In fact, there even exists a fragment of XPath (Core XPath) that can be evaluated in the linear combined complexity $O(|D| * |Q|)$, where $|D|$ is the size of the instance and $|Q|$ the size of the query [4]. We improve these results in the sense that our approach leads to linear evaluation in the size of the intermediate results, which can be much smaller than the size of the document.

This work is mostly inspired by the results presented in [6] and [7]. There it is shown that if the document nodes have prenumbers and postnumbers associated with them then it possible to efficiently retrieve the results of certain axes in document order and without retrieving nodes that are not in the result. For example, if we let $\mathsf{pre}(n)$ and $\mathsf{post}(n)$ be the pre- and postnumber of n then we can efficiently test their relative positions because then n' is a descendant of n iff $\mathsf{pre}(n') > \mathsf{pre}(n) \wedge \mathsf{post}(n') < \mathsf{post}(n)$, and n' is a follower of n' iff $\mathsf{pre}(n) < \mathsf{pre}(n') \wedge \mathsf{post}(n) < \mathsf{post}(n')$.

3 The Data Model

The *logical data model* that we will use is a simplification of the XML data model where a *document* is an ordered node-labelled tree and for such a tree the *document order* is defined as the strict total order over its nodes that is defined by the preorder tree-walk over the tree.

The *physical data model* describes in an abstract way how we assume that documents are stored. The first assumption that we make is that the following partial functions are available for document nodes (if undefined the result as assumed to be **null**) and can be evaluated in $O(1)$ time:

- $\mathsf{fc}(n)$ returns the first child of n
- $\mathsf{ns}(n)$ returns the next sibling of n
- $\mathsf{ps}(n)$ returns the previous sibling of n
- $\mathsf{pr}(n)$ returns the parent of n
- $\mathsf{fo}(n)$ returns the first follower of n
- $\mathsf{pr}(n)$ returns the last predecessor of n

Note that except for the last two functions these are all existing pointers in the Document Object Model.

The second assumption that we make is that there are functions such that we can retrieve the pre- and postnumbers in $O(1)$ time:

- $\mathsf{pre}(n)$ returns the prenumber of n
- $\mathsf{post}(n)$ returns the postnumber of n

The reasonableness of these assumptions is demonstrated by the fact that this physical data model can be generated from a SAX representation [2] that consists of a string of opening and closing tags in LogSpace. This can be shown by an extension of the proof given in [9] for the original DOM data model.

The data type we will use the most is that of *List*. We use the following operations:

- $\mathit{newList}()$ returns a new list
- $\mathit{first}(L)$ returns first element of L
- $\mathit{last}(L)$ returns last element of L
- $\mathit{empty}(L)$ determines if L is empty
- $\mathit{addAfter}(L, n)$ adds n at end of L
- $\mathit{addBefore}(L, n)$ adds n at begin of L
- $\mathit{delFirst}(L)$ removes and returns the first element of L
- $\mathit{delLast}(L)$ removes and returns the last element of L
- $\mathit{isList}(L)$ determines if L is a list

The lists are assumed to be represented as a reference to a pair that consists of a reference to the beginning and the end, respectively, of a doubly linked list. Therefore we can assume that all the operations above and assignments and parameter passing can be done in $O(1)$ time. Furthermore this means that if an argument of a function or procedure is a *List* then it is passed as a reference and therefore all the operations applied to the formal argument are in fact applied to the original list that was passed as an argument.

We now proceed by giving for each axis the corresponding algorithm.

4 Descendant Axis

4.1 Informal Description

Given a list of document nodes that is in document order and without duplicates we cannot compute a sorted list of descendants by simply concatenating the lists of descendants. The reason for this is that if n_1 and n_2 appear in the list in that order and n_2 is a descendant of n_1 then the descendants of n_2 will appear twice in the result of the concatenation. However, if n_2 is *not* a descendant of n_1 then all descendants of n_2 will follow in document order the descendants of n_1. It follows that we only have to skip the nodes in the input list that are preceded by an ancestor, to get a result that is in document order and without duplicates.

4.2 The Algorithm

We first present a procedure that adds the descendants of a document node n behind a list L in document order.

```
1 proc addDesc(L, n)
2   n' := fc(n);
3   while n' ≠ null do
4         addAfter(L, n');
5         addDesc(L, n');
6         n' := ns(n')
7   od
8 end
```

The procedure iterates over all children of n in document order and for each (1) adds the child to L and (2) adds its descendants to L. Since the descendants of a node precede in document order the descendants of the following siblings, it follows that the result is indeed that all descendants of n are added to L. Furthermore, it is easy to see that the time complexity is $O(m)$ where m is the number of added elements to L.

Next, we present the function that given a sorted and duplicate-free list L_{in} returns the sorted and duplicate-free list of descendants of the nodes in L_{in}.

```
 1 funct allDescOrd(L_in)
 2   L_out := newList();
 3   while ¬empty(L_in) do
 4         n := delFirst(L_in);
 5         addDesc(L_out, n));
 6         while ¬empty(L_in) ∧
 7               post(first(L_in)) < post(n)
 8               do
 9               delFirst(L_in)
10         od
11   od;
12   L_out
13 end
```

The function iterates over the elements in L_{in} and adds their descendants to L_{out} unless they are preceded in the list by an ancestor. In line 7 this is tested by comparing the post numbers of $first(L_{in})$ and n. Since n appeared in L_{in} before $first(L_{in})$ if follows that $\mathsf{pre}(n) < \mathsf{pre}(first(L_{in}))$ and therefore that $first(L_{in}))$ is a descendant of n iff this condition is true. The time complexity of the function is $O(l+m)$ where l is the size of L_{in} and m the size of L_{out}.

5 Descendant-or-self Axis

The algorithm for this axis is identical to that of the descendant axis except that for each node in L_{in} that is not preceded by an ancestor we retrieve not only the descendants but also the node itself. The time complexity is therefore the same as the previous algorithm.

6 Ancestor Axis

6.1 Informal Description

The problem for this axis is similar to the descendant axis because two distinct nodes can have common ancestors. Moreover, this can not only happen for nodes that have an ancestor-descendant relationship, but also for nodes that do not. The solution for this problem is to retrieve for each node in the input list only those ancestors that were not already retrieve before. Because the input list is sorted in document order we can do this by walking up the tree and stopping if we find a node that is an ancestor of the previous node in the input list.

6.2 The Algorithm

We first present two helper procedures. The first retrieves all ancestors of a document node n and appends them in document order after a list L.

```
1 proc addAnc(L, n)
2   n' := pa(n);
3   if n' ≠ null
4     then addAnc(L, n');
5          addAfter(L, n')
6   fi
7 end
```

If the number of ancestor nodes in m then the time complexity if this procedure is in $O(m)$.

The next helper procedure will, given a list L, a document node n and a document node n' that precedes n in document order, retrieve the ancestors n that are not ancestors of n' and append them in document order after L.

```
1 proc addAncUntilLeft(L, n, n')
2   n'' := pa(n);
3   if n'' ≠ null ∧ pre(n'') ≥ pre(n')
4     then addAncUntilLeft(L, n'', n');
5          addAfter(L, n'')
6   fi
7 end
```

Note that since n' precedes n in document order it holds that the condition $\mathsf{pre}(n'') \geq \mathsf{pre}(n')$ indeed checks if an ancestor n'' of n is an ancestor of n'. Also here the time complexity is $O(m)$ where m is the number of retrieved ancestors.

Finally, we present the function that given a sorted and duplicate-free list of document nodes L_{in} returns a sorted duplicate-free list of all their ancestors.

```
1 funct allAncOrd(L_in)
2   L_out := newList();
3   if ¬empty(L_in)
4     then n := delFirst(L_in);
5          addAnc(L_out, n)
6   fi;
7   while ¬empty(L_in) do
8         n' := delFirst(L_in);
```

```
9         addAncUntilLeft(L_out, n', n);
10        n := n'
11    od;
12    L_out
13 end
```

In the first part of the algorithm (line 3-6) all ancestors of the first node in L_{in} are retrieved. After this a while loop (line 7-11) iterates over the remaining nodes in L_{in} and retrieves for each node n' all ancestors of n' that are not ancestors of n, the node that preceded n' in L_{in}. Since n also precedes n' in document order it follows that all the ancestors of n' that are retrieved indeed follow those that were retrieved for n. As a result all ancestors that are retrieved are appended in document order. The time complexity of this function is $O(l+m)$ if l is the size of L_{in} and m is the size of the result.

7 Ancestor-or-self Axis

The algorithm for this axis as similar to the one for the ancestor axis except that we we retrieve the ancestors of a node we also add the node itself. The time complexity is therefore also the same.

8 Child Axis

8.1 Informal Description

We cannot use the approach of the previous axes here. Consider for example the fragment in Figure 1. If, for example, we only retrieve for each node the children that we know to precede in document order the children of the next node then for the list $L_{in} = [1, 3]$ we only obtain $[2, 3, 4]$. To solve this we introduce a stack on which we store the children of node 1 which were not retrieved already such that we can return to them when we are finished with the children of node 3.

```
<a id="1">
  <b id="2"/>
  <b id="3"> <c id="4"/> </b>
  <b id="5"/>
</a>
```

Figure 1: An XML fragment

8.2 The Algorithm

Before we present the actual algorithm we present a helper function that results in a list of all children of a document node n in document order.

```
1 funct allChildren(n)
2    L := newList();
3    n' := fc(n);
4    while n' ≠ null do
5          addAfter(L, n');
6          n' := ns(n')
7    od;
8    L
9 end
```

The function simply goes to the first child of n and then follows the following-sibling reference until there is no more following sibling. The time complexity of this function is $O(m)$ if m is the number of retrieved children.

Next, we present the actual algorithm that given a sorted and duplicate-free list of document nodes L_{in} returns a sorted duplicate-free list of all their children.

```
1 funct allChildOrd(L_in)
2    L_out := newList();
3    L_st := newList();
4    while ¬empty(L_in) do
5          n := first(L_in);
6          if empty(L_st)
7            then L' := allChildren(n);
8                 addBefore(L_st, L');
9                 delFirst(L_in);
10         elsif empty(first(L_st))
11              then delFirst(L_st)
12         elsif pre(first(first(L_st))) > pre(n)
13              then L' := allChildren(n);
14                   addBefore(L_st, L');
15                   delFirst(L_in)
16               else n' := delFirst(first(L_st));
17                   addAfter(L_out, n')
18         fi
19   od;
20   while ¬empty(L_st) do
21         if empty(first(L_st))
22           then delFirst(L_st)
23            else n' := delFirst(first(L_st));
24                 addAfter(L_out, n')
25         fi
26   od;
27   L_out
28 end
```

The algorithm consists of two while loops. The first (line 4-19) iterates over the nodes in L_{in} and retrieves the children that it knows it can send to the output list L_{out} and stores the

others on the stack L_{st}. The second while loop (line 20-26) iterates over the remaining children on the stack L_{st} and appends those behind L_{out}. In the following we discuss each while loop in more detail.

The first loop stores unprocessed children on the stack L_{st} where the beginning of L_{st} is the top of the stack. Each position on the stack contains a sorted list of siblings that were not yet transferred to L_{out}. The loop maintains an invariant that states that the nodes in lists that are higher on the stack precede in document order those that are lower on the stack. This is mainly achieved by the **if** statement on line 12 that tests if the node at the beginning of L_{in} precedes the first child node on top of the stack. If this is true then the list of children of n are pushed on the stack and n is removed from L_{in}, otherwise the first child node on top of the stack is moved to the end of L_{out}. Note that in the latter case it indeed holds that all the children of the remaining nodes in L_{in} indeed succeed this child in document order.

The second loop simply flushes the stack which indeed results in adding the remaining nodes to L_{out} in document order because of the invariant that was described for the previous loop.

Since the algorithm iterates over all the nodes in L_{in} and retrieves only those nodes that are added to L_{out} it follows that the time complexity is $O(l+m)$ where l is the size of L_{in} and m the size of L_{out}.

9 Parent Axis

9.1 Informal Description

The fundamental property that will be used for this axis is that if for a duplicate-free sorted list of document nodes we retrieve the parent nodes we obtain a sublist of the list of nodes that we meet when we follow the contour of the tree. For example, if we follow the contour of the nodes in the tree for the fragment in Figure 1 then we obtain the list $[1, 2, 1, 3, 4, 3, 1, 5, 1]$. If we start with the list $[2, 3, 4, 5]$ and we retrieve the list of parents, then we obtain $[1, 1, 3, 1]$ which is indeed a sublist of the first list.

This information can be used by the algorithm because when it iterates over the list of parents and encounters a parent n that precedes the last parent in the output list then it is walking up the tree in the contour walk. As a consequence it knows that after it inserts n in the output list the tail of the output list that starts with n will not change anymore because all the following nodes in the input list will either be after or before this tail in document order. Therefore the algorithm can simply summarize this tail and pretend it corresponds to the node n. It does this by replacing it with a nested list that contains this tail.

As an illustration consider the following possible list of parents: $[1, 2, 5, 4, 9, 8, 2]$. For reasons of homogeneity we represent the output list as a list of lists and if we add a single node it is represented as a singleton list. Therefore after processing the nodes 1, 2 and 5 we obtain the list $[[1], [2], [5]]$. Since the next node 4 precedes 5 the algorithm represents the tail as a nested list that starts with 4 and obtains $[[1], [2], [4, [5]]]$. From this point on the nested list $[4, [5]]$ will be considered as if equal to $[4]$, i.e., the algorithm considers only the first node of the nested lists. Since the next node 9 follows 4 it is simply added, giving $[[1], [2], [4, [5]], [9]]$. The next node is 8 which precedes 9 but follows 4, so we obtain $[[1], [2], [4, [5]], [8, [9]]]$. Also here the nested list $[8, [9]$ is considered as equivalent to $[8]$. Finally the node 2 is added and since it precedes node 4 the two lists starting with 4 and 8 are nested in a list starting with 2 and we obtain $[[1], [2], [2, [4, [5]], [8, [9]]]]$.

As will be clear from the previous example, the result is a nested list that when flattened gives the sorted list of parents but may still contain duplicates. Since the list is sorted these can be eliminated easily.

9.2 The Algorithm

We first present a helper function and a helper procedure. The following function will, given a sorted list L, return a sorted list that contains all the elements in L but no duplicates.

```
1  funct dupElimSort(L)
2    L_out := newList();
3    if ¬empty(L) then n := delFirst(L) fi;
4    while ¬empty(L) do
5           n' := delFirst(L);
6           if n' ≠ n
7             then addAfter(L_out, n')
8                  n := n'
9           fi
10   od;
11   L_out
12 end
```

The time complexity of this function is clearly $O(l)$ if l is the size of L.

Te following procedure will, given a list L and a nested list L_{tr} of document nodes, flatten the list L_{tr} and append it to L.

```
1 proc addFlatList(L, L_tr)
2    while ¬empty(L_tr) do
3          n := delFirst(L_tr);
4          if isList(n)
5             then addFlatList(L, L_tr)
6              else addAfter(L, n)
7          fi
8    end
9 end
```

If at each level every nested list in L_{tr} contains at least one document node then the time complexity if this procedure is $O(m)$ if m is the number of document nodes in the result.

Finally, we present the actual algorithm that given a duplicate-free sorted list of document nodes will return a duplicate-free sorted list of their parents.

```
1  funct allParOrd(L_in)
2     L_tr := newList();
3     while ¬empty(L_in) do
4           n := pa(delFirst(L_in));
5           L := newList();
6           while ¬empty(L_tr) ∧
7                 pre(first(last(L_tr))) > pre(n)
8                 do
9                 n' := delLast(L_tr);
10                addBefore(L, n')
11          od;
12          addBefore(L, n);
13          addAfter(L_tr, L)
14    od;
15    L_dup := newList();
16    addFlatList(L_dup, L_tr);
17    L_out := dupElimSort(L_dup);
18    L_out
19 end
```

The list L_{tr} is used to represent the list of nested lists. Note that in L_{tr} every nested list will always start with a document node. The crucial part is the while loop on lines 6-11 that determines the tail L of L_{tr} where the first nodes of the lists in this tail follow n in document order and removes this tail from L_{tr}. On line 12 this tail is extended with n and finally on line 13 the tail is put back as a nested list at the end of L_{tr}. At the end of the algorithm, when all the nodes of L_{in} have been processed, the resulting nested list L_{tr} is flattened and duplicates are removed from it.

The time complexity if this algorithm is $O(l)$ where l is the size of Lin. To understand this consider the number of times the pre-numbers of two nodes are compared in the while condition starting on line 6. The number of equations that were false are at most l, one for each parent that is considered. The number of successful equations is also at most l because a successful comparison means that the node is from then on nested and will no longer be considered, so in the final L_{tr} every document node has been successfully compared at most once.

10 Following Axis

10.1 Informal Description

To find all the followers of the nodes in a duplicate-free sorted list of document nodes it is sufficient to retrieve the followers of the first node in the list that is not an ancestor of the next node in the list. To understand this consider the following. Let n be this node and n' a node that is in the list after n. Since the node in the list immediately after n is not its descendant n' and its followers are also not descendants of n. Therefore it follows that (1) n' is a follower of n and (2) all followers of n' are also followers of n. Since n' is not a follower of itself, it holds that the set of followers of n' is a proper subset of those of n. On the other hand it can be shown that if n' is an ancestor of n then the set of followers of n' is a subset of those of n.

10.2 The Algorithm

We first present a helper procedure that, given a list L and a document node n, appends to L all followers of n.

```
1 proc addFoll(L, n)
2    if fo(n) ≠ null
3       then addAfter(L, fo(n));
4            addDesc(L, fo(n));
5            addFoll(L, fo(n))
6    fi
7 end
```

The correctness of this procedure follows from the fact that $\mathsf{fo}(n)$ returns the smallest node (in document order) that is a follower of n, and that the followers of a node n are defined as those nodes that are larger in document order but not a descendant of n. Its time complexity is $O(m)$ where m is the number of followers added to L.

Next we present the actual algorithm that given a duplicate-free sorted list L_{in} of document nodes returns a duplicate-free sorted list of all the followers of these nodes.

```
1  funct allFollOrd(L_in)
2    L_out := newList();
3    if ¬empty(L_in)
4      then n := delFirst(L_in);
5           while ¬empty(L_in) ∧
6                 pre(first(L_in)) > pre(n) ∧
7                 post(first(L_in)) < post(n)
8                 do
9                 n := delFirst(L_in)
10          od;
11          addFoll(L_out, n);
12   fi;
13   L_out
14 end
```

The correctness of this function follows from what was said before and the fact that the while condition indeed tests that the first node in L_{in} is a descendant of n. Because the algorithm iterates over all the nodes in L_{in}, determines a single node n and then applies *addFoll*, it follows that the time complexity is $O(l + m)$ if l is the size of L_{in} and m the size of L_{out}.

11 Preceding Axis

11.1 Informal Description

To find the preceding nodes of a sorted list of document nodes we only have to retrieve the preceding nodes of the last node in the list. If this is node n we can retrieve its preceding nodes in document order as follows. We first apply to n the function fo repeatedly until there is no more immediate predecessor. Let the nodes we encounter be $n_0 = n, n_1, \ldots, n_k$. Then n_k is the first predecessor of n in document order. For this node we first retrieve all its ancestors in document order that are not ancestors of n and n_k itself. After this we return to n_{k-1} and retrieve all its ancestors in document order that are not ancestors of n_k and also not ancestors of n, and we retrieve n_{k-1} itself. We repeat this for each n_i with $0 < i < k$ by retrieving all ancestors of n_i in document order that are not ancestors of n_{i+1} or n, and n_i itself. It is easy to see that for each n_i the retrieved nodes follow in document order those of n_{i+1} and that those nodes are predecessors of n. Conversely all predecessors of n are either in $n_1, \ldots, n_k$ or one of their ancestors.

11.2 The Algorithm

Before we present the actual algorithm we present three helper algorithms. The first algorithm will, given a list L and three document nodes n, n' and n'' such that n' is a predecessor of n and n is a predecessor of n'', adds after L in document order the ancestors of n that are not ancestors of n' or n''.

```
1 proc addAncBetween(L, n, n', n'')
2   n''' := pa(n);
3   if n''' ≠ null ∧
4     (pre(n''') ≥ pre(n')) ∧
5     (post(n''') ≤ post(n''))
6     then addAncUntil(L, n''', n');
7          addAfter(L, n''')
8   fi
9 end
```

Note that to test if n''' is not an ancestor of n' it must be tested whether $\neg(\mathsf{pre}(n''') < \mathsf{pre}(n') \wedge \mathsf{post}(n''') > \mathsf{post}(n'))$ or equivalently $\mathsf{pre}(n''') \geq \mathsf{pre}(n') \vee \mathsf{post}(n''') \leq \mathsf{post}(n')$. However, since n is a follower of n' it holds that $\mathsf{post}(n) > \mathsf{post}(n')$ and since n''' is the parent of n it holds that $\mathsf{post}(n''') > \mathsf{post}(n)$, from which it follows that $\mathsf{post}(n''') > \mathsf{post}(n')$. A similar argument shows that the test for n''' and n'' in the if-expression is also sufficient to test whether n''' is not an ancestor of n''. The time complexity of this procedure is $O(m)$ if m is the number of nodes added to L.

The second helper function is similar and will, given a list L and document nodes n and n' such that n is a predecessor of n', add all the ancestors of n that are not ancestors of n' to L in document order.

```
1 proc addAncUntilRight(L, n, n')
2   n'' := pa(n);
3   if n'' ≠ null ∧ post(n'') ≤ post(n')
4     then addAncUntilRight(L, n'', n');
5          addAfter(L, n'')
6   fi
7 end
```

The correctness of this procedure can be shown in a way similar to that of the previous one. Also here the time complexity is $O(m)$ if m is the number of added document nodes.

The third helper procedure will, given a list L and two document nodes n and n' such that n is a predecessor of n', adds all those nodes to L in document order which are (1) predecessors of n but not ancestors of n', (2) ancestors of n but not ancestors of n' or (3) n itself.

```
1 proc addLeftUntil(L, n, n')
```

```
2     if pr(n) ≠ null
3       then n'' := pr(n);
4               addLeftUntil(L, n'', n');
5               addAncBetween(L, n, n'', n')
6       else addAncUntilRight(L, n, n')
7     fi;
8     addAfter(L, n)
9   end
```

The correctness of this procedure follows from the correctness of the previous procedures. The time complexity is $O(m)$ if m is the number of added document nodes.

Finally, we present the algorithm itself which, given a list L_{in} of document nodes returns a duplicate-free sorted list of all their predecessors.

```
1   funct allPredOrd(L_in)
2     L_out := newList();
3     if ¬empty(L_in)
4       then n := last(L_in);
5               if pr(n) ≠ null
6                 then addLeftUntil(L_out, pr(n), n)
7               fi
8     fi;
9     L_out
10  end
```

The correctness follows from the correctness of the helper procedures. The time complexity is $O(m)$ if m is the size of L_{out}.

12 Following-Sibling Axis

12.1 Informal Description

The problems for this axis are very similar to those of the child axis and can be solved in the same way, i.e., by introducing a stack of lists of nodes that contains the nodes that still need to be move to the output list. An extra complication is here that the following siblings of two different nodes may have nodes in common. The solution for this is simple: if we encounter simultaneously the same node in the input list and at the beginning of the list on top of the stack the we ignore the node in the input list.

12.2 The Algorithm

We first present a helper function that given a document node n returns a duplicate-free sorted list of all the following siblings of n.

```
1   funct allFollSibl(n)
2     L := newList();
3     n' := ns(n);
4     while n' ≠ null do
5             addAfter(L, n');
6             n' := ns(n')
7     od;
8     L
9   end
```

Correctness of this function follows from the fact that the function ns returns the first sibling of n that follows n in document order. The time complexity is $O(m)$ where m is the size of the result.

Next we present the actual algorithm which given a list L_{in} of document nodes returns a duplicate-free sorted list of all following siblings of the nodes in this list.

```
1   funct allFollSiblOrd(L_in)
2     L_out := newList();
3     L_st := newList();
4     while ¬empty(L_in) do
5             n := first(L_in);
6             if empty(L_st)
7               then L' := allFollSibl(n);
8                       addBefore(L_st, L');
9                       delFirst(L_in);
10            elsif empty(first(L_st))
11                then delFirst(L_st)
12            elsif pre(first(first(L_st))) > pre(n)
13                then L' := allFollSibl(n);
14                        addBefore(L_st, L');
15                        delFirst(L_in)
16            elsif pre(first(first(L_st))) < pre(n)
17                then n' := delFirst(first(L_st));
18                        addAfter(L_out, n')
19                else delFirst(L_in)
20            fi
21    od;
22    while ¬empty(L_st) do
23            if empty(first(L_st))
24              then delFirst(L_st)
25              else n' := delFirst(first(L_st));
26                      addAfter(L_out, n')
27            fi
28    od;
29    L_out
30  end
```

The correctness of this function is similar to that of the corresponding function for the child axis. The main difference is in line 19 where the case is considered that the current node in the input list, n, is equal to the first node of the list on top of the stack L_{st}. In this case the node n is removed from the input list without copying its siblings to the stack or the output list. The time complexity is also similar, i.e., $O(l + m)$

where l is the size of L_{in} and m is the size of L_{out}.

13 Preceding-Sibling Axis

13.1 Informal Description

This axis is symmetric to the following-sibling axis in the sense that we can use the same algorithm except that we have to do everything in reverse, i.e., we iterate over the input list in reverse and we move nodes from the back of the lists on the stack to the front of the output list.

13.2 The Algorithm

We first present a helper function that give a document node n returns a duplicate-free sorted list of all preceding siblings of n.

```
1 funct allPrecSibl(n)
2   L := newList();
3   n' := ps(n);
4   while n' ≠ null do
5         addBefore(L, n');
6         n' := ps(n')
7   od;
8   L
9 end
```

Similar to the previous axis correctness of this function follows from the fact that the function ps returns the last sibling of n that precedes n in document order. Also here the time complexity is $O(m)$ where m is the size of the result.

Finally we present the algorithm that given a duplicate-free and sorted list of document nodes L_{in} returns a duplicate-free and sorted list of all the preceding siblings of these document nodes.

```
1  funct allPrecSiblOrd(L_in)
2    L_out := newList();
3    L_st := newList();
4    while ¬empty(L_in) do
5          n := last(L_in);
6          if empty(L_st)
7             then L' := allPrecSibl(n);
8                  addBefore(L_st, L');
9                  delLast(L_in);
10         elsif empty(first(L_st))
11              then delFirst(L_st)
12         elsif pre(last(first(L_st))) > pre(n)
13              then L' := allFollSibl(n);
14                   addBefore(L_st, L');
15                   delLast(L_in)
16         elsif pre(last(first(L_st))) < pre(n)
17              then n' := delLast(first(L_st));
18                   addBefore(L_out, n')
19              else delFirst(L_in)
20         fi
21   od;
22   while ¬empty(L_st) do
23         if empty(first(L_st))
24            then delFirst(L_st)
25            else n' := delLast(first(L_st));
26                 addBefore(L_out, n')
27         fi
28   od;
29   L_out
30 end
```

The correctness follows from the symmetry with the previous axis, and for the same reason the time complexity is also $O(l + m)$ with l the size of L_{in} and m the size of L_{out}.

14 Conclusion

The presented algorithms allow us to efficiently evaluate XPath axes, preserving both order and duplicate freeness, assuming that the document is stored as a DOM structure and pre- and post-numbers are available. The algorithms only evaluate the axis and do not evaluate node tests or predicates. As long as the predicates do not refer to the context set of the resulting nodes, i.e., use directly or indirectly the functions `position()` and `last()`, these can be easily applied to the result of the algorithms afterwards. If there is a reference to the context set then there is a problem because the resulting list of nodes does not contain any information about the context set. In this case we can either attempt to reconstruct the context set (e.g., if the step was `child::*` then the context set of n is simply all its siblings) or fall back to the optimization techniques proposed in [8]. Because both techniques guarantee that the result of each step is duplicate-free and sorted it is possible to mix the two techniques within the same path expression.

References

[1] Document Object Model (DOM). Available at: `http://www.w3c.org/dom/`.

[2] Simple API for XML (SAX). Available at: `http://www.saxproject.org/`.

[3] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath

2.0 formal semantics, w3c working draft 2 may 2003, 2002. `http://www.w3.org/TR/query-semantics`.

[4] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.

[5] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. of the 22nd ACM SIGACT-SIGMOD-GIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

[6] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, Madison, 2002.

[7] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB 2003*, 2003.

[8] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *DBPL 2003*, 2003.

[9] L. Segoufin. Typing and querying XML documents: Some complexity bounds. In *Proc. of the 22nd ACM SIGACT-SIGMOD-GIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

STAX/bc: A binding compiler for event-based XML data binding APIs

Florian Reuter, Norbert Luttenberger
Communication Systems Research Group
Christian-Albrechts-University in Kiel, Germany
{flr|nl}@informatik.uni-kiel.de

ABSTRACT

In this article we present the STAX/bc[1] binding compiler which generates event-based data binding APIs out of W3C Schema descriptions. The event-based nature of the generated data binding APIs allows programmers the development of:

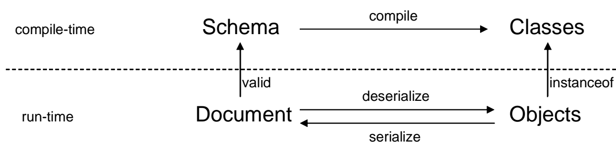
- applications for processing large XML documents or XML data streams,
- applications for resource constraint systems without enough space to deserialize XML documents into main memory, and
- applications which must have full control about the data structures used.

STAX/bc binding compiler can be written targeting any object oriented language, like C++, Java or C#, e.g.

1. Introduction

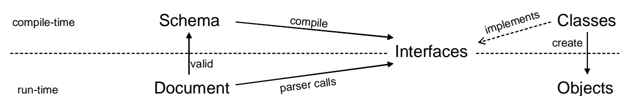
For the development of XML-processing applications powerful XML-APIs are needed which allow programmers to concentrate on the application logic rather than on the mapping between XML documents and programming language objects.

Data binding frameworks provide a good way to achieve this *deserialization*- and *serialization* mapping of XML documents to programming language objects resp. vice versa instead of using low-level APIs like DOM and SAX. Data binding frameworks provide a *binding compiler* which, given a grammar of the XML documents as input, produces a set of classes together with deserialization and serialization functions, which transform XML documents into instances of the generated classes and vice versa. Current data binding frameworks use schema languages like DTDs, W3C Schema and RelaxNG to describe the document grammar. The following figure shows the functioning of data binding frameworks graphically:



[1] STAX is a four letter acronym for Simple *Typed* API for XML, which alludes to the event-based nature of its namesakes SAX (Simple API for XML) and StAX (Streaming API for XML) but stresses the fact that STAX is Schema-*typed*. To avoid confusion with StAX we will write STAX/bc for the presented binding compiler.

The STAX/bc binding compiler presented in this paper works differently. STAX/bc shares the idea with data binding frameworks of using a binding compiler which takes a schema description as input, but the STAX/bc-binding compiler does not produce classes. Instead, the STAX/bc-binding compiler produces interfaces between the application logic and the XML parser. By implementing these interfaces an application can create *any kind* of objects at runtime out of the parsed XML document. The following figure shows the functioning of STAX/bc schematically:



This concept allows the development of

- applications for typed processing of XML documents or XML data streams,
- applications for resource constraint systems without enough space to deserialize XML documents into main memory,
- applications which must have full control about the data structures used.

2. Related Work

Currently available data binding frameworks like the W3C schema based XSD[28], JAXB[20,9], Castor[21], jBIND[22], gSOAP[2], the RelaxNG-based Relaxer[10] or the DTD-based Zeus[23,9] or Quick[24,9] generate classes together with deserialization and serialization functions as shown above. These classes explicitly define the in-memory representation of the deserialized XML documents.

When an XML document is parsed, the whole XML document is deserialized into main memory. This is problematic for large XML documents and impossible for XML streams.

Another disadvantage of the previously mentioned data binding frameworks is that the programmer has no, or little, control about the data structures in which the information from the XML documents is deserialized. Some data binding compiler apply design pattern like factory e.g. or customization options to overcome this, but for example the deserialization of XML documents directly into Java/Swing components as shown in chapter 8 of this paper is not possible with current data binding frameworks.

Event-based XML-APIs generate a sequence of events while parsing XML documents. These events then can be handled by an

application which allows an application to process XML documents on-the-fly. A well known event-based API is SAX. The SAX-API reports XML Infoset-like events to the application via the ContentHandler interface. Applications can handle the events by implementing this interface.

The difference to its namesake STAX/bc is that SAX reports XML Infoset-like events via a single interface ContentHandler, whereas the STAX/bc binding compiler generates interfaces out of a schema description. Informally spoken, SAX generates "untyped" events whereas STAX/bc generates "typed" events.

The advantages of event-based APIs are, that XML parsers with event-based XML-APIs can be implemented very resource sparing, since they only generate events and do not have to load the whole XML document into main memory. Furthermore programmers can, by handling the events in an appropriate way, store the information from the XML documents in data structures of their choice or process the information on-the-fly.

Beside the mentioned data binding frameworks there exist several other kinds of data binding XML-APIs. Another class of XML-APIs are used in data binding frameworks like [1,18,21]. Here a binding function explicitly defines how XML documents should be bound to programming language objects. E.g. a XML-document <member><name>Mr. X</name></member> can be bound to the class class Member{String name;}; by the following binding function: <member>→Member, <name>→Member.name without using an intermediate schema. Often this binding function is (semi-)automatically inferred by the binding framework. These kinds of XML-APIs are favorably used, when "real" semistructued data – e.g. data for which no schema is available or can't be specified for some reasons – must be processed[1].

Yet another class of XML-APIs are used in XML programming languages like [5,14, 30]. XML programming languages are based on XML Infoset-like structures and they use schemas to perform XML type checking[13].

To the best knowledge of the authors STAX/bc is the first data binding compiler which generates event-based XML data binding APIs out of W3C schema descriptions. Herewith STAX/bc combines the advantages of data binding frameworks and event-based XML-APIs, which are:

- processing of large XML documents and data streams as well as on-the-fly processing,
- free choice of data structures used to maintain the data, and
- easy access to the information within your programming language due to specially tailored interfaces.

3. STAX/bc's concept

STAX/bc's technical concepts can be summarizes as follows:

- Every W3C schema type is mapped into an interface. By implementing these interfaces an application can bind itself to a STAX/bc-aware XML parser.
- The interfaces own create and add methods which are called at runtime by the STAX/bc-aware XML parser.
- The create method plays the role of a factory method with which the XML parser creates new objects. The add method is used by the XML parser to report the generated objects to the application logic.

Since STAX/bc deals only with interfaces and makes no assumptions about the programming languages primitive types, the STAX/bc binding compiler can be used to generate code for any object oriented programming language.

We will first present in chapter 4 the STAX/bc-binding compiler which maps W3C schema descriptions to the interfaces. Thereafter we explain in chapter 5 how the generated interfaces are invoked by the STAX/bc-runtime system. In chapter 6 we present a slight modification of the basic rules given in chapter 4 and 5 which lead to a structural improvement of the generated interface structure. In chapter 7 we address the naming problem and chapter 8 shows the STAX/bc programming exemplarily.

4. STAX/bc-Binding Compiler

This section describes the STAX/bc-binding compiler. As mentioned above the binding compiler is responsible for mapping W3C schema descriptions into interfaces.

The STAX/bc-binding compiler applies four mapping-rules, which are sufficient to map any entirely named W3C schema description into interfaces. A W3C schema description is entirely named if it does not contain neither an anonymous complex type definition, nor an anonymous simple type definition, nor an anonymous model group. This can be achieved for any W3C schema by applying a naming preprocessor. This step is explained in section 7. In the following we assume that the input W3C schema is entirely named.

The mapping-rules are shown in figure 1. We will discuss them briefly now.

Rule 1: Built-in types:

The W3C Schema built-in types, as defined in W3C specification, are mapped to the interfaces bearing the same name as the built-in type suffixed by Creator as shown in figure 1.

Any simple type inherits the addContent(content:char[]) method from the AnySimpleTypeCreator-Interface.

The interface representing the W3C Schema built-in type QName additionally has an addNamespaceBinding(prefix:char[], namespace:char[]) method, since QNames need access to the actual prefix-namespace mapping.

Rule 2: Top level attribute and element declarations:

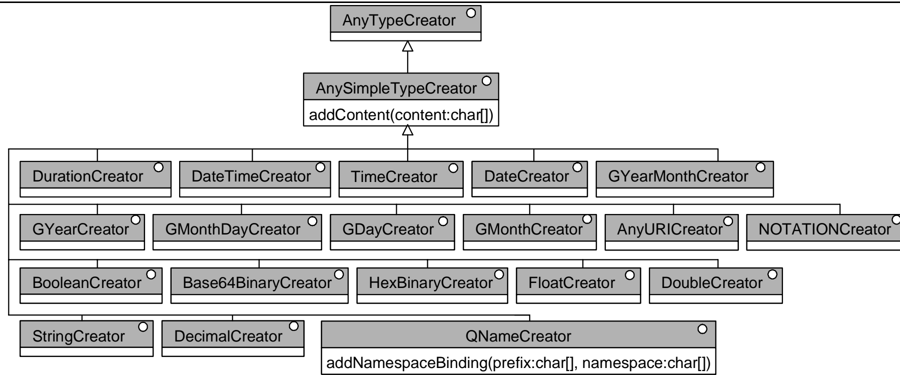
Each schema definition is mapped into an interface named DocumentCreator. For every top level particle a create and an add method are added to the DocumentCreator interface.

The DocumentCreator interface represents a whole XML document and is always generated. It serves as an entry point for the XML parser.

Rule 3: Type definitions, named model groups, attribute groups:

A named complex type with complex content, a named simple type, a named model group and an attribute group definition are mapped into an interface with the type name suffixed by Creator:

Figure 1: The STAX-binding compiler's mapping rules:



Rule 1. Built-in types.

```
<schema>
    (<element name="ElementName" type="ElementType"/>
    |<attribtue name="AttrName" type="AttrType"/>
    |...)*
</schema>
```

→

DocumentCreator

Creator-Methods:
createQName():QNameCreator
create*ElementType*():*ElementType*Creator
create*AttrType*():*AttrType*Creator
Add-Methods:
add*ElementName*(i:*ElementType*Creator)
add*AttrName*(i:*AttrType*Creator)

Rule 2. Top level element and top level attribute declarations.

```
<(complexType|simpleType|group|attributeGroup) name="TypeName">
    (<(complexContent|simpleContent)>
     <(restriction|extension) base="BaseType">)?
        ((<(all|sequence|choice)>)?
            (<element name="ElementName" type="ElementType"/>
            |<group ref="GroupRef"/>
            |<any ...>)*
        (</(all|sequence|choice)>)?)?
        (<attribute name="AttrName" type="AttrType"/>
        |<attributeGroup ref="AttrGroupRef"/>
        |<anyAttribute .../>)*
    (</(restriction|extension)>
     <(complexContent|simpleContent)>)?
</(complexType|simpleType|group|attributeGroup)>
```

→

***BaseType*Creator** ***AttrGroupRef*Creator**

***TypeName*Creator**

Creator-Methods:
create*ElementType*():*ElementType*Creator
create*GroupRef*():*GroupRef*Creator
create*AttrType*():*AttrType*Creator
createDocument():DocumentCreator
Add-Methods:
add*ElementName*(i:*ElementType*Creator)
add*GroupRef*(i:*GroupRef*Creator)
add*AttrName*(i:*AttrType*Creator)
addDocument(i:DocumentCreator)

Rule 3. Type definitions, named model groups and attribute groups

```
<simpleType name="TypeName">
        <list itemType="ItemType"/>
</simpleType>
```

→

***TypeName*Creator**

Creator-Methods:
create***ItemType***():***ItemType***Creator
Add-Methods:
add***ItemType***(i:***ItemType***Creator)

Rule 4. Simple type lists.

```
<simpleType name="TypeName">
        <union memberTypes="MemberType_1 ... MemberType_n"/>
</simpleType>
```

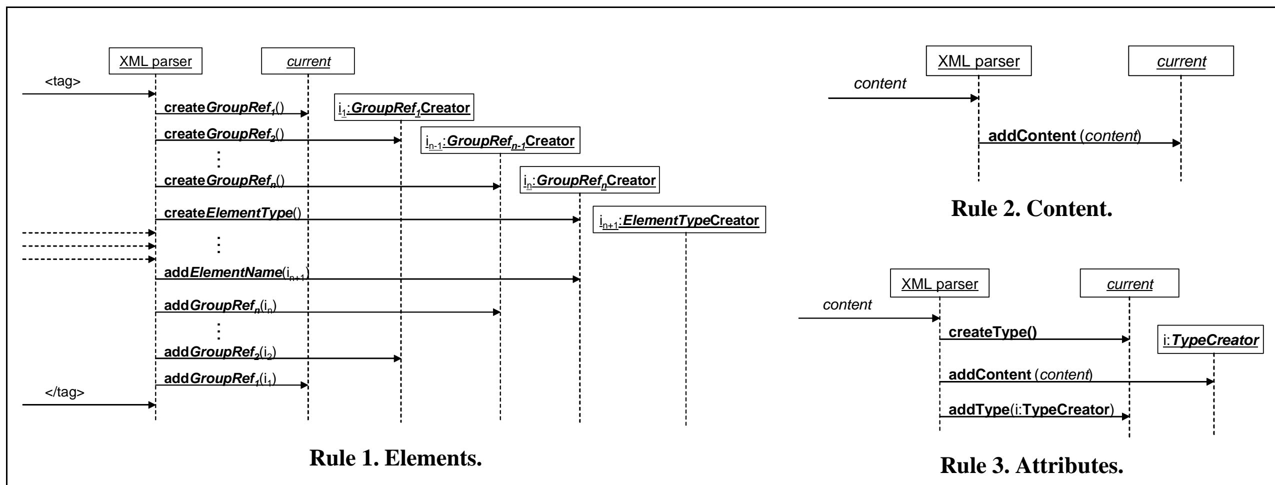
→

***TypeName*Creator**

Creator-Methods:
create***MemberType***$_1$():***MemberType***$_1$Creator
create***MemberType***$_2$():***MemberType***$_2$Creator
...
create***MemberType***$_n$():***MemberType***$_n$Creator
Add-Methods:
add***MemberType***$_1$ (i:***MemberType***$_1$Creator)
add***MemberType***$_2$ (i:***MemberType***$_2$Creator)
...
add***MemberType***$_n$ (i:***MemberType***$_n$Creator)

Rule 5. Simple type unions.

Figure 2: The STAX-runtime system:



W3C schema construct	mapped to
<complexType name="***TypeName***"> <simpleType name="***TypeName***"> <group ref="***TypeName***"> <attributeGroup name="***TypeName***"/>	Interface ***TypeName***Creator

A derivation is mapped to a corresponding generalization:

W3C schema construct	mapped to
<extension base="***BaseType***"> <restriction base="***BaseType***">	Generalization. from ***BaseType***Creator to ***TypeName***Creator

Particles and attribute declarations are mapped into create and add methods:

W3C schema construct	mapped to
<element name="***ElementName***" type="***ElementType***"> <group ref="***GroupRef***"> <attribute name="***AttrName***" type="***AttrType***">	create***ElementType*** resp. create***GroupRef*** resp. create***AttrType*** and add***ElementName*** resp. add***GroupRef*** resp. add***AttrType***

An attribute group reference is mapped into a generalization:

W3C schema construct	mapped to
<attributeGroup ref="***AttrGroupRef***"/>	Generalization from ***AttrGroupRef***Creator to ***TypeName***Creator.

The wildcards <any> and <anyAttribue> are mapped into a createDocument and an addDocument method.

Rule 4: Simple type lists

A simple type which defines a list is mapped into an interface with appropriate add and create methods:

W3C schema construct	mapped to
<simpleType name="***TypeName***"> <list itemType="***ItemType***"> </simpleType>	Interface ***TypeName***Creator with create***ItemType*** and add***ItemType***.

Rule 5: Simple type unions

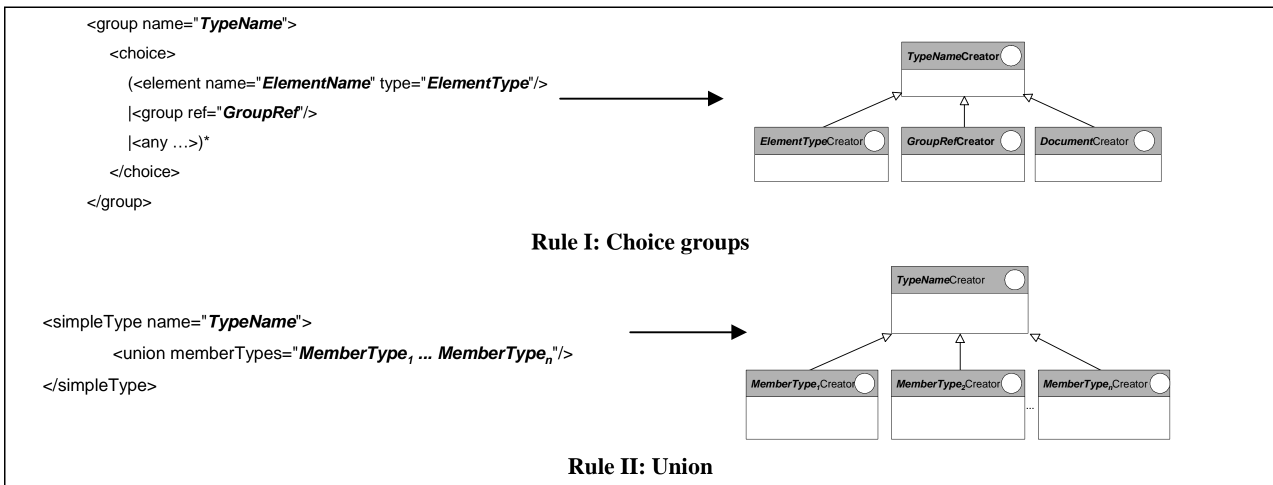
A simple type which defines a union is mapped into an interface with appropriate create and add methods.:

W3C schema construct	mapped to
<simpleType name="***TypeName***"> <union memberType="***MemberType***$_1$ …***MemberType***$_n$"/> </simpleType>	Interface named ***TypeName***Creator with create***MemberType***$_i$ and add***MemberType***$_i$ methods.

The above rules cover all W3C schema key concepts namely <schema>, <element>, <group>, <attribute>, <simpleType>, <complexType>, <simpleContent>, <complexContent>, <restriction>, <extension>, <all>, <sequence>, <choice>, <any>, <anyAttribute>, <list>, <union>, <attributeGroup>, <enumeration>, <pattern>, <any>, <anyAttribute>. However, there is one unsupported special case: Element declarations of type anyType (e.g. <element name="*name*" type="xsd:anyType"/>). This is, because anyType elements allow "real" semistructured data, which means that a priory no classes can be generated.

The create-Methods install an abstract factory into every interface which arose from a complex type definition. This gives every implementing class the chance to define the type of its children.

Figure 3: Statically structure improving rules:



An alternative would be the usage of a single global factory. This would remove the create-Methods from the generated interfaces and thus would reduce the number of methods an implementing class must implement. We choose not to use a single global factory, since we want to give parent classes full control over their child types.

5. STAX/bc-Runtime System

In this section we describe the STAX/bc-Runtime system which builds upon a validating W3C Schema-aware XML parser. Depending on the currently parsed XML information item, the runtime system invokes appropriate create and add methods.

A W3C schema is a singe-type tree grammar which can be parsed by an event-based deterministic top-down tree automaton[15]. Consider the following algorithm:

1. Upon every start element <tag> we differentiate two cases:
   a. We are currently at the root state. We then search for a top level element declaration <element name="*tag*" type="*T*"/> and push *T* onto the stack.
   b. The stack is non-empty. Let *T* be the stack's topmost type. We search for a <element name="*tag*" type="T_{tag}"/> declaration (resp. <element ref="*tag*"> together with the top-level element declaration <element name="*tag*" type="T_{tag}"/>) reachable via none or more <group>s in *T*'s complex type definition and push T_{tag} onto the stack.
2. Upon every end element tag </tag> we check the sequence of child types T_1, … T_n against the regular expression defined by the current type.
3. Simple content *c* is checked according to the simple type definition.
4. Attributes name="*value*" are checked by searching an appropriate <attribute name="name" type="*T*"/> declaration in the current type and then checking the value using the type *T*.

The above algorithm can easily be extended to produce the STAX/bc create and add events by applying the following rules:

Rule 1: Elements

For every starting tag <tag> validated by the schema constructs <group ref="***GroupRef***$_1$"/>…<group ref="***GroupRef***$_n$"/><element name="***ElementName***" type="***ElementType***"/> in step 1 of the above algorithm the runtime environment generates the following create methods: First every group involved in the validation of <tag> is created using the appropriate create function. Then the complex type associated with the tag is created by the appropriate create function.

After the tag's attributes and children have been processed in the same manner, the created instances are added in step 2 of the above algorithm in reverse order to the belonging parent instance by performing the appropriate add calls. Figure 2 illustrates the invocation mechanism.

Rule 2: Content

For every content event *content* processed in step 3 of the above algorithm the runtime system uses the addContent method to pass the incoming content to the simple type (resp. complex type with simple content) instance.

Rule 3: Attributes

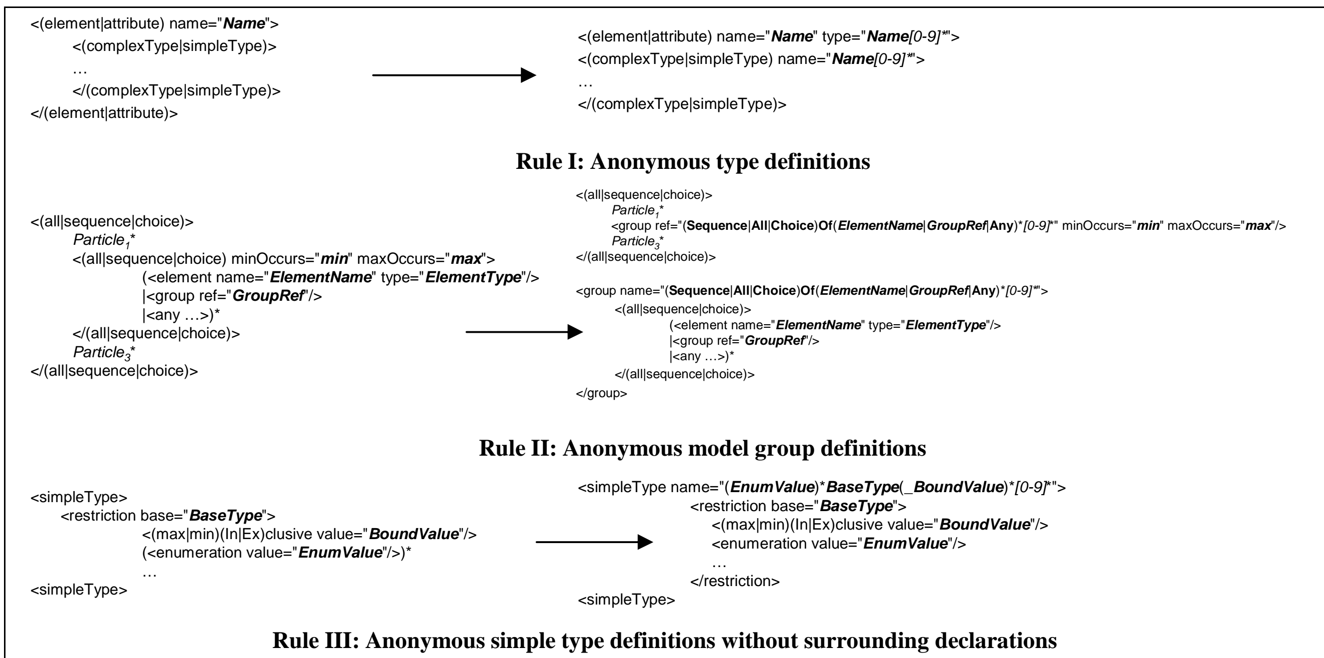
For every attribute name="value" processed in step 4 of the above algorithm the runtime system first creates the appropriate simple type by calling the appropriate create method, and then uses the addContent method to pass the incoming content to the simple type instance. Finally the simple type instance is added to the parent instance by using the appropriate add method.

6. Improvement of the static structure

There exist two W3C schema constructs for which only one of more given possibilities can occur: The <choice> and the <union> constructs.

With the previously presented mapping rules these constructs would be mapped to interfaces in a way that this "either ... or" property gets lost.

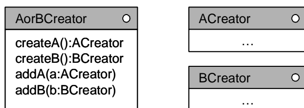
Figure 4: Naming Rules:



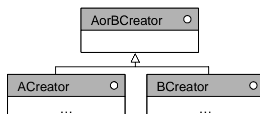
For example consider the schema fragment

```
<group name="AorB">
   <choice>
   <element name="a" type="A"/>
   <element name="b" type="B"/>
   </choice>
</group>
```

It would be mapped to the following interfaces by a non-improved binding compiler:



A much better design is the usage of the OOP concepts polymorphism and inheritance to express the “either ... or” relationship:



This idea is close to the façade design pattern, which provides a unified interface for a set of interfaces in the subsystem[11].

We will now show how the rules from section 4 must be altered in order improve the static design of the generated interfaces.

We highly recommend the usage of these improvements. In all our tests the improvements lead to an interface structure, which was more understandable and easier to program with.

The rules to improve the static structure are shown in fig. 3. We will discuss them briefly now:

Rule I: Choice groups

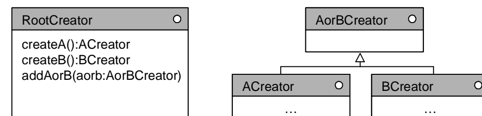
Model Group definitions whose variety is a disjunction are mapped into an empty interface and a generalization is added to all particles' types.

Furthermore every `create`*`TypeName`* method in referencing interfaces is replaced by the create methods which would have been generated by the default rule.

Consider for the example the following schema fragment:

```
<complexType name="Root">
   <group ref="AorB"/>
</complexType>
<group name="AorB">
   <choice>
   <element name="a" type="A"/>
   <element name="b" type="B"/>
   </choice>
</group>
```

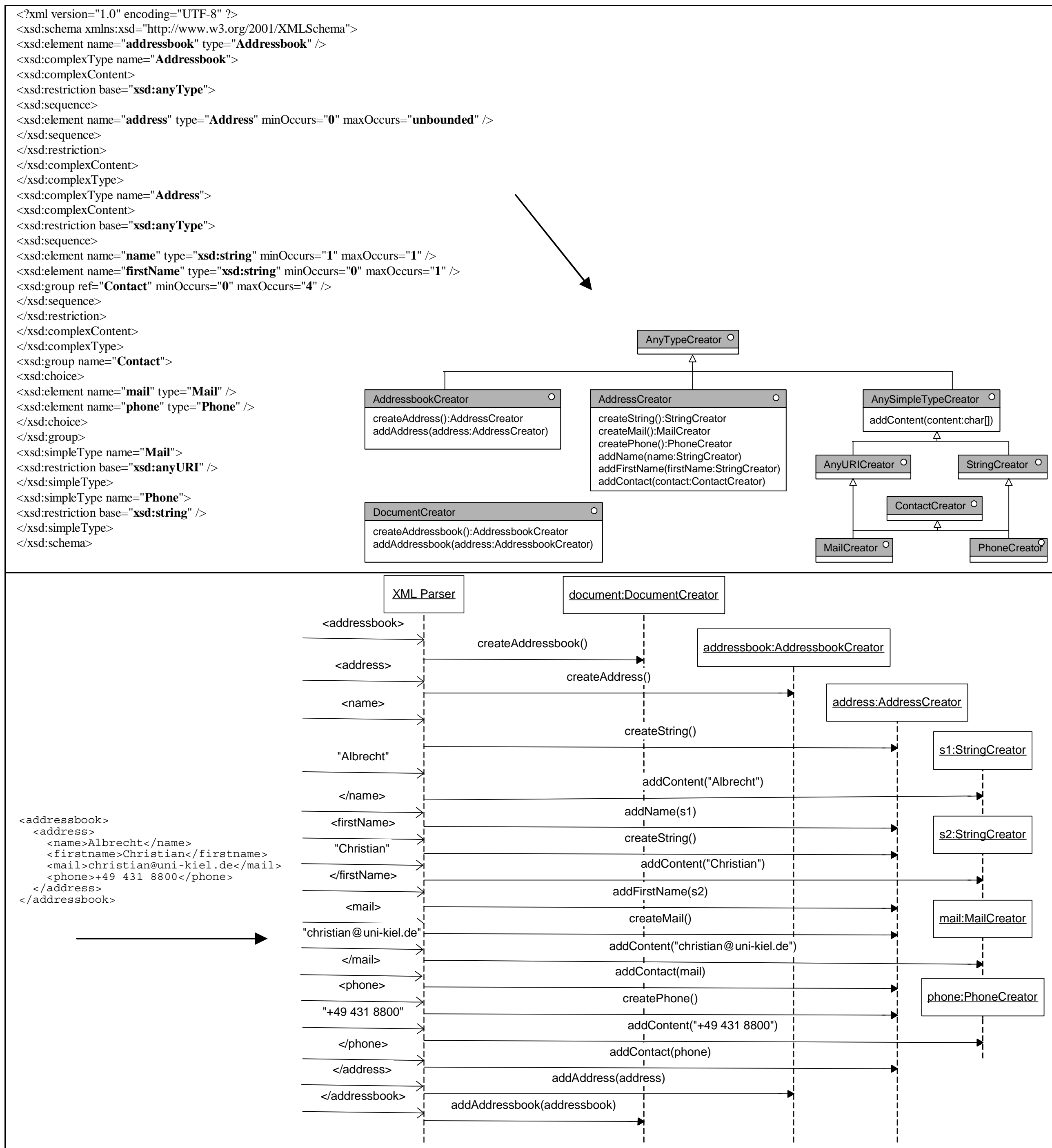
which leads to the following interface structure by applying rule I:



Rule II: Union

The interface structure generated for simple type definitions which define new simple types by uniting a set of other simple types is altered by applying the façade-like pattern in the same way as above.

Figure 5: The address book sample



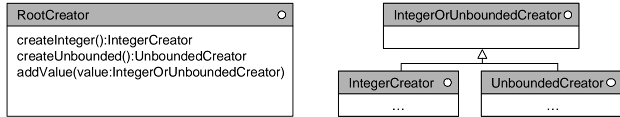
For every member type of the union a generalization is added and the `create`*`TypeCreator`* method in referencing types is replaced by the appropriate `create` methods.

Consider for the example the schema fragment:

```
complexType name="Root">
<element name="value" type="IntegerOrUnbounded"/>
</complexType>
<simpleType name="IntegerOrUnbounded">
   <union memberType="xsd:integer unbounded"/>
</simpleType>
<simpleType name="unbounded">
   <restriction base="xsd:token">
```

```
      <enumeration value="unbounded"/>
   </restriction>
</simpleType>
```

By applying rule II the following interface structure is generated:



7. Naming

The interfaces generated by the STAX/bc-binding compiler need to be named. If the type declaration from which an interface should be generated is anonymous, then the binding compiler must generate a name. For code quality this name should be meaningful, i.e. the binding compiler should not generate type names from "unknown0" to "unknown999".

We next present the naming applied in the STAX/bc-binding compiler. The rules are summarized in figure 4.

Rule I: Anonymous type definitions

Anonymous type definitions are named with the surrounding declaration's name. Additionally a number may be appended when conflicts must be resolved.

For example the element declaration which contains an anonymous type definition

```
<element name="a">
   <complexType>
      <sequence>…</sequence>
   </complexType>
</element>
```

is mapped to:

```
<element name="a" type="A">
<complexType name="A">
   <sequence>…</sequence>
</complexType>
```

Rule II: Anonymous model group definitions

Anonymous model group definitions are named by creating a new named group whose name is build out of the model group variety (all, sequence or choice) and the concatenation of the particles' names. Again, additionally a number may be appended when conflicts must be resolved.

Rule III: Anonymous simple type definitions without surrounding declarations

Anonymous simple type definitions without surrounding declarations are named by prefixing the ***BaseType*** with the enumeration values and suffixing the ***BaseType*** with the given bounding values.

8. STAX/bc Programming

We will now explain the STAX/bc-based programming by example of the address book schema shown in fig. 5. It models an address book which contains a set of address entries. Each address entry contains a name, optionally a first name and up to four contacts. A contact information is either an e-mail address or a telephone number. A sample address book is shown in fig. 5.

The interfaces generated by the STAX/bc-compiler (structural improvements enabled) are also shown in fig. 5. For convenience we also illustrated in the sequence diagram of fig. 5 how the STAX/bc-runtime system calls the generated interfaces when the address book sample of fig. 5 is parsed.

We will now develop an Java-application which deserializes the sample address book of fig. 5 diretly into a Swing instance as shown in fig. 6. This is done without any intermediate representation.

Figure 6.

First we implement the interfaces `StringCreator`, `MailCreator` and `PhoneCreator` as a Swing `JLabel`:

```
public class MyString extends JLabel implements
StringCreator, PhoneCreator, MailCreator {
  public void addContent(java.lang.String content) {
  setText(content);
}}
```

Next we implement the `AddressCreator` interface as a Swing `JPanel` which holds the `JLabels`:

```
public class MyAddress extends JPanel implements
AddressCreator {
public StringCreator createString() {
  return new MyString();}
public void addName(StringCreator name) {
  add((MyString) name);}
public void addFirstname(StringCreator name) {
  add((MyString) name);}
public ContactCreator createPhone() {
  return new MyString();}
public ContactCreator createMail() {
  return new MyString();}
public void addContact(ContactCreator contact) {
  add((MyString) contact);
}}
```

Finally the `AddressbookCreator` interface is implemented by a Swing `JFrame` which holds the addresses in a tabbed pane:

```
public class MyAddressbook extends JFrame implements
AddressbookCreator {
private JTabbedPane pane = new JTabbedPane();
MyAddressbook() {
  super("STAX sample");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  getContentPane().add(pane);}
public AddressCreator createAddress() {
  return new MyAddress();}
public void addAddress(AddressCreator address) {
  pane.add("Address", (MyAddress) address);
}}
```

The following implementation of the `DocumentCreator` interface will instantiate the frame:

```
public class GUI implements DocumentCreator {
public AddressbookCreator createAddressbook() {
  return new MyAddressbook();}
public void addAddressbook(AddressbookCreator
addressbook) {
  ((MyAddressbook)addressbook).pack();
  ((MyAddressbook)addressbook).show();}}
```

Further applications of the STAX/bc-API are e.g. the storage of XML encoded data into relational data bases. By performing appropriate `INSERT` resp. `UPDATE` SQL-functions within the `create` and `add` methods, XML content can easily be stored into a relational data base – typed and on-the-fly.

9. Conclusion

We presented the STAX/bc data binding compiler. STAX/bc is the first binding compiler which generates event-based data binding APIs out of W3C schema descriptions. Therewith STAX/bc unites the advantages of data binding frameworks together with the advantages of event-based XML-APIs. These are:

- the processing of large XML documents and data streams as well as on-the-fly processing
- free choice of data structures used to maintain the data
- easy access to the information within your programming language due to specially tailored interfaces.

STAX/bc generated XML-APIs are successfully deployed in several applications. Many of these applications use non-trivial W3C schemata.

The STAX/bc binding compiler is integrated in the <<astax-Framework which is available in an early alpha version at http://www.ccastax.net.

Literature

[1] Fabio Simeoni, David Lievens, Richard Connor, Paolo Manghi. Language Bindings to XML. IEEE Internet Computing 7(1), Jan., Feb. 2003, p. 19-27.

[2] Robert van Engelen, Gunjan Gupta, Saurabh Pant. Developing Web Services for C and C++. IEEE Internet Computing 7(2), Mar., Apr. 2003, p. 53-61.

[3] Joseph Williams. J2EE vs. .NET, The Web Services Debbate. Communications of the ACM 46(6), June 2003, p. 59-63.

[4] Gerry Miller. .NET vs. J2EE. Communications of the ACM 46(6), June 2003, p. 64-67.

[5] Haruo Hosoya, Benjamin C. Pierce. XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT) 3(2), May 2003.

[6] Richard Connor, David Lievens, Fabio Simeoni, Steve Neely, George Russell. Projector: A Partially Typed Language for Querying XML. Plan-X: Programming Language Technologies for XML, 2002.

[7] Paolo Manghi, Fabio Simeoni, David Lievens, Richard Connor. Hybrid Applications over XML: Integrating the Procedural and Declarative Approaches. Proceedings of the fourth international workshop on Web information and data management 2002, McLean, Virginia, USA. Nov. 2002.

[8] Simeoni, F., Manghi, P., Lievens, D., Connor, R.C.H. and Neely, S. An approach to high-level language bindings to XML. Information and Software Technology 44(4), Mar. 2002, p. 217–228.

[9] Brett McLaughlin. Java & XML Data Binding. O'Reilly & Associates. 1st. ed. 2002.

[10] Hiroshi Maruyama, Kent Tamura, Naohiko Uramoto, Makoto Murata, Andy Clark, Yuichi Nakamura, Ryo Neyama, Kazuya Kosaka, Satoshi Hada. XML and Java: developing Web applications. Pearson Education. 2nd ed. 2002.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: elements of reusable object-oriented software. Addison-Wesley Publishing. 1996.

[12] Eric van der Vlist. XML Schema. O'Reilly & Associates. 1st. ed. 2002.

[13] Dan Suciu. The XML Typechecking Problem. SIGMOD Record 31(1), Mar. 2002, p. 89-96.

[14] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. Proceedings of the ACM International Conference on Functional Programming, 2003.

[15] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. Extreme Markup Languages 2000, August 13-14, 2000. Montreal, Canada.

WWW-Links

[16] XML Data Binding Resources. http://www.rpbourret.com/xml/XMLDataBinding.htm.

[17] Code Fast, Run Fast with XML Data Binding. http://java.sun.com/xml/jaxp/dist/1.0.1/docs/binding/DataBinding.html.

[18] XML and Java technologies: Data binding. http://www-106.ibm.com/developerworks/xml/library/x-databdopt/.

[19] Java Specification Request 31: XML Data Binding Specification. http://www.jcp.org/en/jsr/detail?id=031.

[20] Java Architecture for XML Binding (JAXB). http://developer.java.sun.com/developer/technicalArticles/WebServices/jaxb/.

[21] Castor. http://www.castor.org.

[22] jbind. http://jbind.sourceforge.net/.

[23] Zeus. http://zeus.enhydra.org/.

[24] Quick. http://qare.sourceforge.net/web/2001-12/products/index.html#quick.

[25] SAX. http://www.saxproject.org.

[26] DOM. http://www.w3.org/DOM/.

[27] JDOM. http://www.jdom.org/.

[28] XML Schema Definition Tool (Xsd.exe). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconxmlschemadefinitiontoolxsdexe.asp.

[29] XML Information Set. http://www.w3.org/TR/xml-infoset/.

[30] XML Processing Plus Plus. A typed and stream-based XML processing extension for Java. http://alphaworks.ibm.com/aw.nsf/FAQs/xmlprocessingplusplus

Mixed XML/Relational Data Processing

Yana Kadiyska*and Dan Suciu
University of Washington

Abstract

Presently, XML is either processed with an XQuery engine, or shredded into relations and processed with a SQL engine. We argue that neither of these strategies is satisfactory, given today's state of the art relational databases and XML query engines. This paper proposes a mixed processing model, in which XML native data coexists with relational data. In this model, two different query engines, a SQL engine and an XQuery engine are also integrated. We discuss several alternatives for shredding XML data into a mixed storage, and show that the main challenge in any such approach is query performance. We introduce several optimization methods that improve the query performance dramatically, by essentially pushing more work from the XQuery to the SQL query engine. Finally, we describe a system implementing a mixed XML/relational storage, and evaluate it on the XMark benchmark.

1 Introduction

Currently, two approaches for XML processing exist. In *native XML processing* a specialized XML engine evaluates queries, expressed in XQuery or XSLT, over XML native data [FM01, FS02]. In *relational processing* the XML data is shredded into tables, and XML queries are translated into SQL then executed entirely on the relational engine [STH+99, CFI+00, FST00, SSB+00, FMS01, SK+01, TVB+02, Moe02, FKM+02].

There are tradeoffs between these two approaches. Native XML processors can support the entire XML query language, and can be optimized to handle XML specific features, such as ordered data, mixed content, and data whose schema is difficult to map to a relational structure. Moreover, native XML engines are easy to extend to support new XML features, for example XML encryption [IDS02, HIM02], XML signatures [BBF+02], XML key management [HB02] or XLink [DMOT01]. Native XML processors, however, do not scale to large data instances and large numbers of concurrent users, and do not offer transaction management, recovery, and ACID semantics.

This has led researchers to propose the second approach to XML processing that "shreds" the entire XML data into tables, then translates all XML queries into relational queries on the shredded data. Work in this direction has shown that a large fragment of XQuery can be efficiently translated to a relational query [FKM+02] and run on a relational engine. It is possible to translate nested queries, queries on ordered XML data [TVB+02], queries performing complex transformations on the XML data, and updates [TIHW01]. Even fragments of XSLT can be translated to SQL [JMS02, Moe02]. However, these techniques are restricted to data-oriented XML queries, do not support features such as mixed content, entities, or processing instructions, and are more difficult to extended to new features, say XLink or encryption. Moreover, despite the existing technology for mapping XQuery to SQL, major database vendors adopt a wait-and-see approach, and only support some fragments of XPath in their engines.

In this paper, we argue for the need of an integrated approach to processing XML data, which uses both a relational engine and a native XML engine. In our approach, XML data kept as text may coexist with XML data shredded and stored in a relational engine. Correspondingly, two query engines coexist, a native XML query engine and a relational database engine, and they are coordinated by a middleware in order to process the mixed data. This approach offers the best of both worlds. The part of the XML documents that is regular, order independent, and generally data-like can be shredded, while the part that is too irregular, ordered, with an unknown schema, or with features not supported by the relational engine is kept natively. While attractive, such an approach raises a major challenge, because it has to combine two different query engines, a SQL engine and an XQuery engine. The two engines are asymmetric. The SQL engine is fast and scales to large data instances, but runs on a different server than the application and it is expensive to retrieve data over an ODBC or JDBC connection. The XQuery engine is light-weight, and can be integrated with the client application, but it does not scale to large data instances, has no cost-based optimizer, and uses no indexes: this is typical for today's XML engines, such as `xalan` [Apa02] and `galax` [FS02].

This paper makes the following contributions. First, it describes the space of XML shredding strategies. Our

*Contact author: yana@cs.washington.edu.

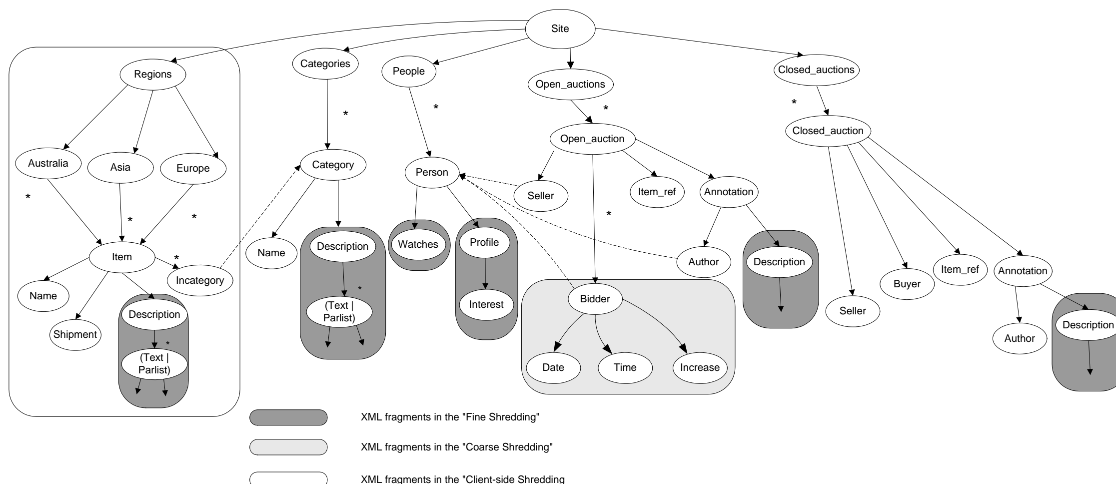


Figure 1: The DTD graph of the XMark benchmark (fragment). A `*` denotes a one-many relationship, while no label denotes a one-one relationship. Dotted edges denote `href(s)` references.

second contribution consists of a framework for specifying and processing mixed storage. An administrator defines a logical XML view over a relational schema that contains XML fields. Users see the resulting virtual XML view, and query it with XQuery. The XQuery is composed with the logical XML view, and results in a mixed XQuery/SQL expression, which is then processed by the two engines. Third, we describe how to evaluate the mixed expression on the two engines, and propose several optimization techniques. Finally, we evaluate our techniques on the XMark benchmark [SWK+01], using different shredding strategies.

Related Work Several techniques for mapping relations to XML, or storing XML into relations have been discussed in the past [DFS99, STH+99, CFI+00, FST00, SSB+00, FMS01, SK+01, YASU01, TVB+02, Moe02, FKM+02]. Our work is based on SilkRoute [FKM+02], which describes a translation of XQuery into SQL[1]. All these systems assume a pure relational storage, with the exception of STORED([DFS99]) which also considers an *overflow* storage.

A mixed storage of XML data into relations and XML files is described by Deutsch and Tannen [Ali02] and in Deutsch's thesis [Deu02]. That approach is far more general than ours: XML data can be stored redundantly, and the connections between different parts are expressed by constraints. XQueries are reformulated into queries over the stored data using a technique called *chase-backchase*. By contrast, our approach is simpler, less powerful, and lets us focus on system-level optimization techniques.

Organization Section 2 describes the mixed XML/relational processing model by presenting different shredding alternatives and by describing several heuristic-based optimizations for queries in this model. Section 3 describes the SR2 system's architecture. Central to SR2 is a formalism for specifying mappings from the mixed XML/relational storage to XML: we describe it in Sec. 4, and describe the query translation, optimization, and scheduling in Sec. 5. We provide an experimental evaluation in Sec. 6, and conclude in Sec. 7.

[1]SR2 stands for SilkRoute 2.

2 The XML/Relational Processing Model

In this section, we describe the XML/relational processing model, by examining the shredding alternatives and by illustrating query processing in this model. Throughout the paper, we use XML data and queries from the XMark benchmark [SWK+01] for illustration. The DTD for this data has 77 elements and 14 attributes; a small fragment of the DTD graph is shown in Fig. 1. The graph shows some of the elements in the DTD and their inclusion relationship. For example, the root element is `Site`, and since its content model is[2]:

```
<!ELEMENT Site (Regions, Categories, People,
                Open_auctions, Closed_auctions)>
```

the node `Site` has five children in Fig 1: `Regions`, `Categories`, `People`, `Open_auctions`, and `Closed_auctions`.

2.1 Types of Shreddings

We classify the different shredding alternatives according to the granularity of XML fragments that are kept in na-

[2]We omit a sixth subelement, to reduce clutter.

tive format (text format). The choice between the alternatives is dictated by the type and availability of a DTD. Although the different types of shreddings affect query performance, we assume that the shredding is determined purely based on the schema properties and without knowledge of the query load that the query engines will face.

Pure relational shredding In this method, the entire XML data is mapped into tables. The advantage is that every XQuery expression can be translated entirely into SQL and, thus, processed efficiently on the relational engine. The disadvantage is that the entire XML schema needs to be known in advance, and the method is restricted to schemas that are relatively regular and non-recursive. Shredding techniques for XML data without a schema have also been studied [FK99, DFS99, YASU01] but they are, in general, less efficient for querying and, especially, reconstructing the XML data. The pure relational shredding method has been studied intensively in the past and we will not discuss it further in this paper.

Relational shredding with fine XML fragments Here, most of the XML data is mapped into tables, except for some relatively small elements that happen to have a highly irregular structure. These fragments are often small enough to be stored in attributes of type VARCHAR, and do not require a CLOB. The schema needs to be known in advance, but some isolated irregular parts may be tolerated. In some case query performance can be better than in pure relational shredding because it does not decluster small, irregular XML fragments.

The term regular is defined here in relation to ease of conversion to a relational schema. For example, elements that occur infrequently would likely require the frequent insertion of null values in the database, or introduce the need for an extra join if such elements are shredded into a separate table. Similarly, recursive XML schemas would require the computation of a transitive closure over a given table. (For example, see the `Description` element in 1).

Example 2.1 Fig. 2 illustrates a shredding with fine XML fragments for the data in Fig. 1. It generally corresponds to the mappings described in [STH+99], except for the dark blocks in Fig. 1, which are stored as XML text in the tables. An example is the attribute `Profile` of `People`. There are a total of six attributes of type XML in the tables shown in Fig. 2, corresponding to the six dark boxes in XML schema in Fig. 1. In most cases, such small XML fragments can be implemented as `varchar`, with some fixed upper bound on he size. This allows for relatively efficient storage.

Relational shredding with coarse XML fragments In this strategy larger (coarser) XML fragments are kept as text, even if they could be further shredded into relations. The reasons for keeping them as native XML

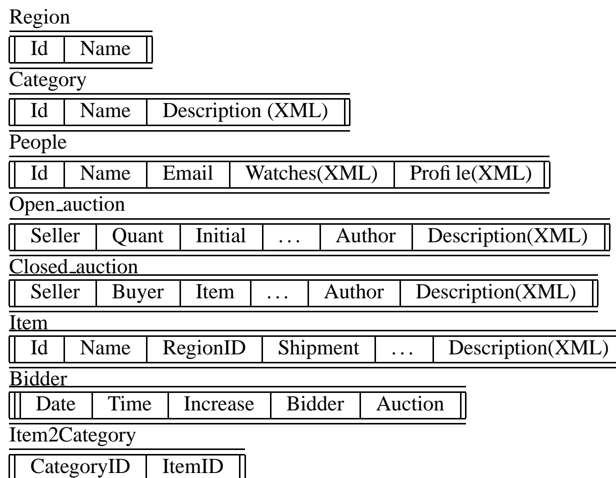


Figure 2: Part of the relational schema with fine XML fragments.

Figure 3: Part of the relational schema with coarse XML fragments. Tables `Open_auction` and `Bidder` in Fig. 2 have been replaced by the single table `Open_auction` with a `Bidder` attribute of type XML.

may vary: their schema may not be known, or they may have specific features that are not supported by the relational engine, or by the specific shredding technique. For example, consider an XML message broker that needs to manage SOAP messages exchanged by different applications. The message `header` has a well-defined structure [GHMN01a, GHMN01b] and the broker can define a fixed relational schema to store it. But the message `body` depends on the application, and a fixed relational schema is not possible. A less efficient schema-independent shredding technique [FK99, YASU01] is also not possible, because some of the esoteric features of XML, such as processing instruction, comments, and white spaces between elements, are lost when the XML message body is shredded into relational format, and this may be unacceptable for some applications. The solution is to store the message body as a coarse XML fragment. In general, coarse XML fragments need to be stored as CLOBS, and this contributes to decreased performance. In addition, any query that accesses the data in the CLOB will further degrade the performance significantly.

Example 2.2 Fig. 3 illustrates the table `Open_auctions` with coarse XML fragments: the corresponding XML fragment kept natively is shown in light gray in Fig. 1. Previously, we had two tables, `Open_auctions` and `Bidder`, representing the nested `Bidder` elements (Fig. 2): now the table `Bidder` disappeared and, instead,

we store the Bidder elements as XML text.

Client-server XML storage The last shredding alternative is to keep an entire fragment of the XML data in text format, and store it in an XML file at the client site. At a logical level, this is equivalent to a very coarse shredding, where the table containing the XML fragment has been reduced to a single tuple. But at the physical level there is an important difference, since the XML fragment is now on the client's site, and we must integrate relational data at the database server with XML files at the client.

Example 2.3 Continuing our example, we may decide to store the entire Regions element natively, as XML (see the white box in Fig. 1). Since there is only one such element we store it locally, at the client, in one XML file.

Under all strategies the user sees a pure XML view of the entire, integrated data, and queries it with XPath or XQuery expressions. We explain now how XQueries can be processed on this mixed storage.

2.2 Queries in the XML/Relational Model

For our running example,we assume the fine shredding in Fig. 2.

Pure relational queries As a warm-up, consider the following simple XPath query:

```
/Site/Regions/Europe/Item
      [Incategory/Idref()->Category
                 [Name/text()= "auction"]]/Name
```

Following the navigation path in the graph in Fig. 1, one can see that this query touches only the pure relational data. Our system translates it into the following SQL query:

```
select Item.Name
from   Region, Item,
       Category, Item2Category
where
       Item2Category.ItemID = Item.Id and
       Item2Category.CategoryID = Category.Id and
       Category.Name='auction' and
       Item.regionID=Region.Id and
       Region.Name='Europe'
```

Naive navigation inside the XML data Now consider the following:

```
/Site/Categories/Category
  [Description[Text/Bold/text()="main"]
   //Parlist//Listitem/Text/text()= "closeout"]
 /Name
```

Here, the predicate

```
  [Description[Text/Bold/text()="main"]
   //Parlist//Listitem/Text//text()= "closeout"]
```

navigates through the XML fragment stored in the Description attribute of the table Category. This query is evaluated in two stages, as follows. In the first stage, the following SQL statement is submitted to the relational engine:

```
select Category.Name, Category.Description
from   Category
```

The result consists of a set of tuples of the form (Name, Description). In the second stage, each tuple is submitted to an XQuery engine which evaluates the following predicate on the Description attribute (assuming $D to be bound to the Description XML fragment):

```
if not empty($D[Description
                          [Text/Bold/text()="main"]
      //Parlist//Listitem/Text/text()="closeout"])
   then true
else false
```

If the predicate returns false, then the tuple is discarded; otherwise the Name component is added to the answer set. Thus, the initial query was translated into one SQL query followed by several XQuery predicates.

Notice that in this example all Category records need to be retrieved from the relational database and all are submitted to the XQuery engine. This is inefficient, because most will be discarded by the XQuery filter.

Navigation using LIKE A better way to answer the previous query is to issue and optimized SQL query to the relational backend:

```
select Category.Name, Category.Description
from   Category
where  Category.Description LIKE "%main%" and
       Category.Description LIKE "%closeout%"
```

This returns only tuples where Description contains both strings main and closeout. We still need to run the same XQuery predicates as before, but on much fewer tuples.

Clearly, the LIKE operator will still return many false positives. The next optimization addresses this problem.

Navigation with the Occurs table The idea is as follows: for each attribute of type XML, create a separate table storing all attribute values (CDATA) and all element values (#PCDATA) that occur in that XML fragment. In our example, there will be six such tables, because there are six XML attributes in Fig. 2:

```
Occurs1(CategoryID, Path,Data,Position)
Occurs2(PersonID, Path,Data,Position)
. . .
Occurs6(ItemID,Path, Data,Position)
```

Consider the table Category. For every #PCDATA value in the XML fragment stored in Description there will be one entry in Occurs1. The same will be done for every CDATA value (i.e., attribute value). These tables can be used to further filter the tuples at the database server, as illustrated by the following SQL query:

```
select Category.Name, Category.Description
from   Category, Occurs1 t1, Occurs1 t2
where  Category.Id = t1.CategoryID and
       Category.Id = t2.CategoryID and
       t1.Data = "main" and t1.Path="Description/Text/Bold"
       and t2.Data = "closeout" and
       t2.Path LIKE '%Parlist/Listitem/Text'
```

Joins between XML fragments So far, all queries have been simple navigational queries, and we have already seen that simple optimizations can improve queries significantly. Now consider a more complex XQuery, involving a join between two XML fragments:

```
from $x in /Site/Regions/Item,
     $y in /Site/Categories/Category
where $x/Mail/Parlist/Text/text() =
      $y/Description/Listitem/Text/text()
return <Answer> $x/Name, $y/Price </Answer>
```

The join condition between `$x` and `$y` requires us to query both XML fragments under `Item` and `Category`. We still want to answer such queries in two stages: a SQL query followed by several XQuery queries. A naive way to achieve that is to issue the following SQL query, which computes the cross product of `Item` and `Category`:

```
select Item.Name as IName, Item.Mail,
       Category.Name as CName, Category.Description
from   Item, Category
```

This results in a tuple stream where each tuple has four attributes: `IName`, `Mail`, `CName`, and `Description`. Then, for each tuple, an XQuery expression can decide whether that tuple satisfies the join condition:

```
if $m/Mail/Parlist/Text/text() =
      $d/Description/Listitem/Text/text()
   then true
else false
```

where `$m` and `$d` refer to `Mail` and `Description`, respectively. Clearly, this is very inefficient since the product of the two tables is much larger than the result that we need. We can optimize this by using the `Occurs` tables:

```
select Item.Name, Item.Mail,
       Category.Name, Category.Description
from   Item, Category, Occurs1 d, Occurs6 m
where  Item.Id = d.ItemID and
       Category.Id = m.CategoryID and
       d.Path = '/Description/Listitem/Text' and
       m.Path = '/Mail/Parlist/Text'
       d.Data = m.Data
```

The relational engine will only return those tuples for which the `Mail` and the `Category` XML fragments have some word in common. Notice that the query is still evaluated in two stages: one SQL query followed by several XQuery queries.

It is possible, depending on the schema, that the SQL query above produces no false positives. In such cases, the XQuery stage is no longer needed: the entire query would be pushed to the relational engine.

Mixing Server-site and Client-site data Finally, we illustrate how queries over both server-site and client-site data can still be executed in two stages. Assume that the `Regions` element is stored at the client site, in a large XML file (this is the white rectangle in Fig. 1). Consider the first query presented in this section:

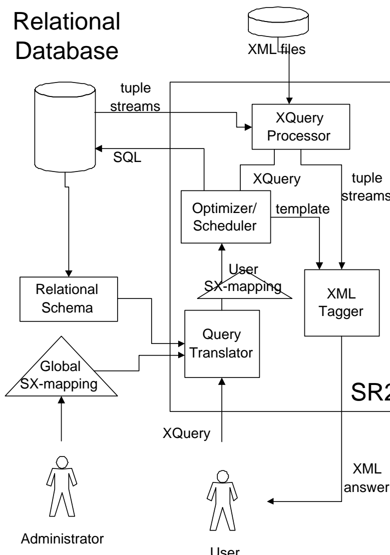


Figure 4: Architecture of the SR2 system

```
/Site/Regions/Europe/Item
      [Incategory/Idref()->Category
                 [Name/text()= "auction"]]/Name
```

Apparently, this requires us to navigate through the `Regions` first, and only then to enter the table `Category`, but this would violate our two stage evaluation strategy. Instead, we start by issuing the following SQL query:

```
select distinct Item2Category.ItemId
from   Category, Item2Category
where  Item2Category.CategoryID = Category.Id and
       Category.Name='auction' and
```

This results in a tuple stream where each tuple contains one `ItemId`. Next, each tuple is submitted to the XQuery engine as follows:

```
for $x in document("regions.xml")
    /Regions/Europe/Item[@id=$ItemID]/Name
return $x
```

3 Architecture

We have built a system, called SR2, that allows data to be shredded into a mixed relational/XML storage, and translates XQuery expressions into SQL and XQueries. Its architecture is shown in Fig. 4. SR2 integrates two query engines: the relational query engine, running on a database server, and an XQuery engine running in the client's address space. Users formulate XQuery expressions over the integrated XML view, and the SR2 system evaluates them in two stages: first, by issuing a small number of SQL queries to the database backend, then by running some XQuery on each tuple in the tuple stream(s). It should be

noted that the number of SQL queries executed is always small, typically 1 or 2, and depends only on the query, and not the data. By contrast, many XQueries may be executed, one for each tuple.

To start, the data administrator specifies a mapping from the mixed storage (relational database plus various XML fragments) into a unique XML view. The mapping is called the *global SQL/XQuery-mapping*, or SX-mapping[3] and embeds both SQL expressions and XQuery expressions. Users see the virtual, unified XML view, and never see the underlying storage directly. They formulate XQueries against that view, which are translated by the system into an internal representation called *user SX-mapping*. The user SX-mapping is then split by the *optimizer/scheduler* into three parts: a set of SQL queries that are submitted to the database engine, a set of XQueries sent to the XQuery engine, and a template for reassembling the data into XML. Several heuristic-based optimizations, including those described in Sec. 2.2, are applied at this stage. Each SQL query generates a tuple stream, which may have some attributes containing XML fragments. Next, one tuple at a time is submitted to the XQuery engine, which may discard the tuple, or keep it and construct some transformed data. While processing the XML fragment in a given tuple, the XQuery engine may also inspect the (much larger) XML files stored locally. Finally, the tuple stream is submitted to the XML tagger, which adds XML tags and nests the XML elements according to the template. The tagger is very simple and efficient, because the tuples are already sorted by the relational engine: it is a *constant space tagger* according to the terminology in [SSB+00].

4 The SQL-XQuery Mapping

We describe now our logical framework for specifying mixed XML/relational storage. In this framework a database administrator defines a logical XML view of the relational data containing XML fragments. Users query this view with XQuery. The system composes the XQuery with the logical XML view, and generates a mixed SQL/XQuery mapping, called SX-mapping.

Definition of the SX-Mapping An SX-mapping consists of a tree, whose nodes have two labels: an XML tag label or data type, and a query. The query consists of a SQL fragment, an XQuery fragment, and a combined `where` clause. The SQL fragment consists of a `from` and a `where` clause, while the XQuery fragment consists of a `for` and a `where` clause. Each fragment defines and uses variables in that language. There is uni-directional communication between these two fragments: a tuple variable defined in SQL may be used as a starting point for navigation in XQuery. More complex predicates that involve both SQL and XQuery variables are moved to the joint `where` clause. Finally, leaf nodes in the SX-mapping tree have an additional clause, which may be either a SQL `select` clause, or an XQuery `return` clause.

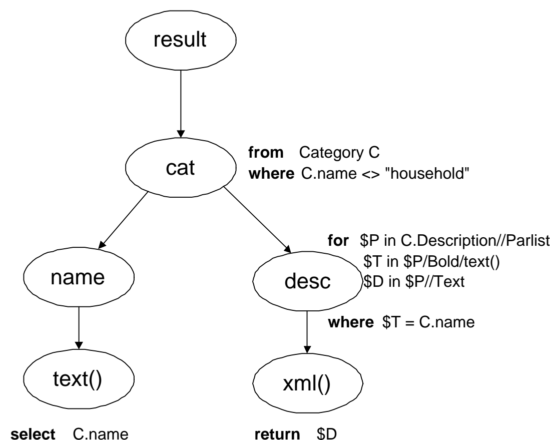


Figure 5: An Example of an SX-mapping.

Example 4.1 Consider the following XQuery expression, over an XML data whose schema is shown in Fig. 1:

```
<result>
{ for $x in /Site/Categories/Category
      $y in $x/name/text()
  where $y <> "household"
  return
   <cat>
       <name> { $y } </name>
       { for $z in $x/Descriptions//Parlist
                              [Bold/text()=$y]//Text
         return <desc> { $z } </desc>
       }
   </cat>
}
</result>
```

Assume that the data is shredded in relations with fine XML fragments (Example 2.1), with the tables in Fig. 2. Then the corresponding SX-mapping is shown in Fig. 5. The top four nodes are labeled with the XML labels `result`, `cat`, `name`, and `desc`, while the two leaves are labeled with the data types `text()` and `xml()`: the latter means a `text()` type whose string denotes a well-formed XML fragment. The nodes are also labeled with query fragments: missing query fragments are empty. The `cat` node is labeled with a pure SQL fragment:

```
from Category C
where C.name <> "household"
```

Because of the iteration over `Category`, several `cat` nodes will be created, one for each binding of the tuple variable `C`. The node `name` has no query fragment, meaning that exactly one such node will be created for each instance of the parent node: in other words, every `cat`

[3] This generalizes the *public view* in [FKM+02].

will have exactly one name child. The text() node is labeled with the SQL fragment:

```
select C.name
```

meaning that the actual value of the XML text node is the result of the expression C.name. Notice that variables introduced by some query fragments may be used in query fragments below the point where they were introduced: for example, C is introduced by the query fragment for cat, and used in the query fragment for text(). Now consider the node desc. Its associated query fragment is an XQuery for expression, that binds three variables, plus a mixed SQL/XQuery where expression. The tuple variable C is used twice: once in the for statement, as starting point of the navigation, and once in the mixed where clause, to connect the XML part to the SQL part. The meaning is that one instance of a cat node will be created for each combination of variables introduced in the for clause that satisfy the where predicate. Finally, the query fragment on the xml() node consists of a single return clause. The meaning is that the entire XML tree bound to the variable $D is returned here.

SX-mappings occur in two places in the system, see Fig. 4. First the *global SX-mapping* describes how the entire XML view is obtained from the mixed storage. This mapping is described in XQuery, by the data administrator, and is typically a large query. In one application to a medical database the XQuery describing this mapping had over one thousand lines, and it took a programmer a few weeks to write, which included designing the XML schema. Second,SX-mappings occur as representations of user queries, and are called the *user SX-mappings*. Such queries needs to be "composed" with the global SX-mapping using a complex composition algorithm (denoted *query translator* in Fig. 4). The user SX-mapping is much smaller, since users usually request simple XML structures, and only require small fragments of the data. The most complex part of the of our system is this composition algorithm.

5 Query Translation, Optimization, and Scheduling

The query translation takes a user query, expressed in XQuery, and translates it into an SX-mapping, i.e., essentially a mixed SQL/XQuery expression. To do that it needs the global SX-mapping, which specifies how the XML view is obtained from the mixed storage. In essence, the translation consists of composing the two queries, and simplifying the resulting expressions.

The optimizer is heuristic-based and its aim is to shift as much of the load as possible into the database, assuming that state of the art XML processors are still lagging behind relational database engines in terms of performance. In addition to the heuristics described in Sec. 2.2, we are interleaving SQL and XQuery joins, performing, in effect, a semi-join between the relational and XML sources, and also eliminating some SQL joins that return almost all pairs, e.g. like in cartesian products.

The scheduler takes an SX-mapping and decomposes it into a number of SQL queries, plus a number of XQueries to be applied to the tuples. We apply the following heuristics: nodes in the SX-mapping that are in a one-one relationship become part of the same SQL queries, while one-many relationships are part of multiple queries. For example, referring to Fig. 5, there is a one-many relationship between the following pairs of nodes: (result, cat), and (cat, desc), hence at most three SQL queries will be generated. However, the SQL query for result is empty, hence no query is issued, and the SQL query for desc is empty too (it has no from clause). As a consequence, a single SQL query is issued to the database engine, on behalf of the nodes cat, name, and text():

```
select *
from Category C
where C.name <> "household"
```

An XQuery is needed for the nodes desc and xml().

6 Experiments

We evaluated our system on the XMark benchmark [SWK+01] using two different shreddings: a fine shredding and a coarse shredding that also included a client-site XML file: they correspond to the dark boxes in Fig. 1 and to the gray and white boxes respectively. We measured three parameters: the total number of tuples transfered between the relational database and the middleware, the total number of bytes transferred, and the total query running time, which consists of the SQL running time plus the XQuery running time. In our experiments, we used the full version the benchmark, whose XML document (before shredding) has 116MB. For the SQL engine we used SQL Server 8.0 and for the XQuery engine we used Galax 0.2.0 [FS02]. The average and maximum sizes of the native XML fragments in the two shreddings are shown in Fig. 11.

In the graphs, we only show those queries that touched at least one XML fragment, i.e., we omit the pure relational queries. We investigated several approaches to evaluating mixed queries, corresponding to the optimizations described in Sec. 2.2, and call them "naive", "LIKE", and "Occurs".

Fig. 6 shows the total number of tuples shipped from the relational engine to the XQuery engine. Notice that the scale is logarithmic. The improvements in the number of tuples due to the various optimizations is dramatic: most of the queries involving joins timed out after 10 minutes when using the naive approach, but using the Occurs

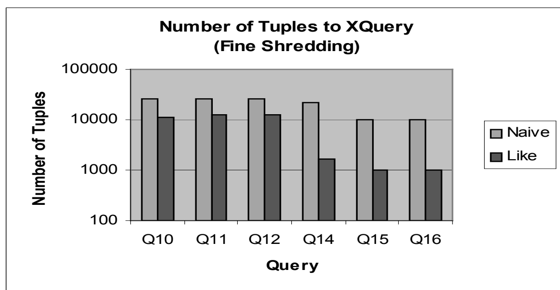


Figure 6: Number of tuples sent to client site under fine shredding

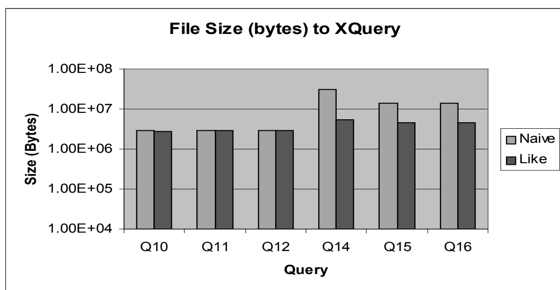


Figure 7: Total size of the data sent to client site (bytes) under fine shredding

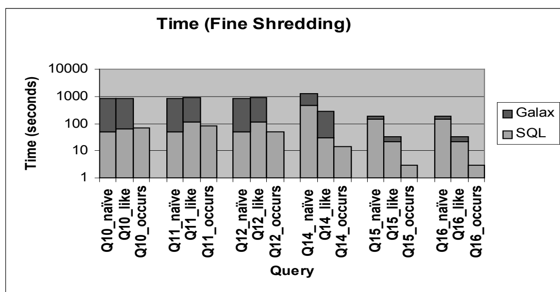


Figure 8: Total query time (RDBMS+XQuery engine). XQuery timed out for Q10, Q11, and Q12 under the naive and Like approaches.

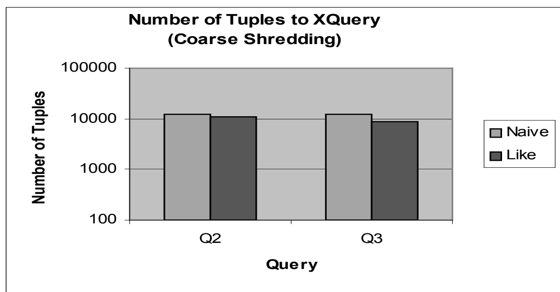


Figure 9: Number of tuples sent to client. Coarse Shredding

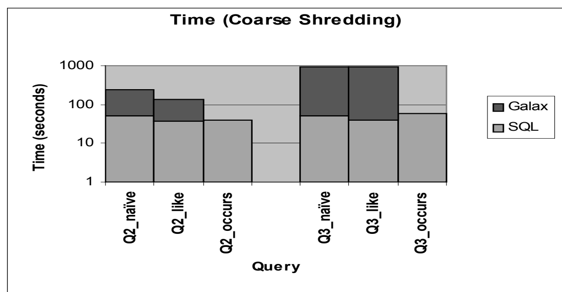


Figure 10: Total query time (RDBMS+XQuery engine) XQuery timed out for Q3 under the naive and LIKE approaches

	XML attribute	Avg	Max
Fine	`Item(description)`	1389	13262
	`Person(profile)`	223	1222
	`Closed_auction(description)`	1409	16644
	`Open_auction(description)`	1397	13312
Coarse	`Open_auction(bidder)`	644	7933

Figure 11: Sizes (in bytes) of the XML fragments stored in the relational tables for two different shreddings of the XMark benchmark

table eliminated the need for an XQuery stage completely. For comparison, we also show the total amount of data shipped to the client in Fig 7: the relative savings is still high, but a bit smaller, because the LIKE predicate is a better filter for small XML fragments than for larger fragments. Fig. 8 reports the total running time for the fine shredding under each optimization method, broken into the SQL running time plus the XQuery running time. In three cases (Q10, Q11, Q12) the XQuery engine did not finish and we aborted it. Queries Q15 and Q16 returned much fewer tuples under "LIKE" optimization,since the navigation path is relatively long and thus ensured high selectivity. Using the Occurs tables, all queries could be answered entirely in the SQL engine, without a need for the XQuery stage.

The graphs 9, 10, show the same experiments but for the coarse shredding. Four queries (Q2,Q3,Q9 and Q14) accessed the coarse XML fragments, but only two of these (Q2 and Q3) *benefited* from the "Occurs" method, and only these are included in Fig. 10. Overall, the coarse shredding method resulted in poorer query performance.

The reason no performance improvement was observed with queries 9 and 14 is that they accessed the large XML file stored on the client site (the white box in Fig. 1). This document was not subject to our optimizations.

Summary In summary, the experiments show the following. First, query evaluation over mixed XML/relational data is perfectly feasible, under two conditions: one implements some simple optimizations,

and there are no joins between XML fragments stored in as text. When the latter occurs then it is almost impossible to do efficient filtering on the relational engine. In applications where such joins are indeed needed one needs to complement the XML fragments with a pure relational shredding of those XML fragments, using a schema-less technique like [YASU01]. Then the query can be processed by the relational engine, while the XML fragment can still be returned in its original format. Our second finding confirms one of the premises of this work: today, the XQuery engine is much slower than the relational engine, and forms the main bottleneck in mixed data processing.

7 Conclusions

We have described a flexible method for storing and processing XML data that mixes native XML data (stored as text) with relational data. We argue that such a processing model is needed, given the current state of database and XML technology. While the performance gap between the two types of engines may narrow in the future, it will likely remain significant, because XQuery engines need to emphasize full compatibility with the complete XML specification and other standards, while relational engines can afford to support a cleaner, data-like fragment. We showed that a mixed XML/relational processing model is possible, offering the best of both worlds.

References

[Ali02] Alin Deutsch and Val Tannen. Querying XML with Mixed and Redundant Storage. Technical Report MS-CIS-02-01, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 2002.

[Apa02] Apache. Xalan-C++, 2002. http://xml.apache.org.

[BBF+02] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML-signature syntax and processing, 2002. `http://www.w3.org/TR/xmlsig-core`.

[CFI+00] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. subramanian. XPERANTO: publishing object-relational data as XML. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[Deu02] Alin Deutsch. *An Experimental Evaluation of the MARS System*. PhD thesis, University of Pennsylvania, 2002.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

[DMOT01] S. DeRose, E. Maler, D. Orchard, and B. Trafford. XML linking language (xlink) version 1.0, 2001. `http://www.w3.org/TR/xlink`.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[FKM+02] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML. *ACM Transactions on Database Technology*, 27(4), December 2002.

[FM01] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in natix. *Journal of the WWW*, 4(3):167–187, 2001.

[FMS01] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.

[FS02] M. Fernandez and J. Simeon. Galax: the XQuery implementation for discriminating hackers, 2002. available from `http://db.bell-labs.com/galax/`.

[FST00] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[GHMN01a] M. Gudgin, M. Hadley, J.J. Moreau, and H. Nielsen. SOAP version 1.2 part 1: Messaging framework, 2001. available from the W3C, `http://www.w3.org/2000/xp/Group/`.

[GHMN01b] M. Gudgin, M. Hadley, J.J. Moreau, and H. Nielsen. SOAP version 1.2 part 2: Adjuncts, 2001. available from the W3C, `http://www.w3.org/2000/xp/Group/`.

[HB02] P. Hallam-Baker. XML key management specification, 2002. `http://www.w3.org/TR/xkms2`.

[HIM02] M. Hughes, T. Imamura, and H. Maruyama. Decryption transform for XML signature, 2002. `http://www.w3.org/TR/xmlenc-decrypt`.

[IDS02] T. Imamura, B. Dilaway, and E. Simon. XML encryption syntax and processing, 2002. `http://www.w3.org/TR/xmlenc-core`.

[JMS02] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT programs to efficient SQL queries. In *Proceedings of WWW*, pages 616–626, 2002.

[Moe02] G. Moerkotte. Incorporating xsl processing into database engines. In *VLDB*, Hong Kong, August 2002.

[SK+01] J. Shanmugasundaram, , J. Kiernana, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB*, pages 261–270, Rome, Italy, September 2001.

[SSB+00] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egipt, September 2000.

[STH+99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: limitations and opportunities. In *Proceedings of VLDB*, pages 302–314, Edinburgh, UK, September 1999.

[SWK+01] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *Sigmod Record*, 30(5), 2001.

[TIHW01] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *SIGMOD*, May 2001.

[TVB+02] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, May 2002.

[YASU01] M. Yoshikawa, Oshiyuki Amagasa, T. Shimura, and S. Uemura. Xrel: A path-based approach to storage and retrieval of xml documents using relational databases. In *ACM TOIT, 1(1)*, 2001.

A Language for Bi-Directional Tree Transformations

Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt
University of Pennsylvania

ABSTRACT

We present a semantic foundation and a core programming language for bi-directional transformations on tree-structured data such as XML. In one direction, these transformations map a "concrete" tree into a simplified "abstract" one; in the other, they map a modified abstract tree, together with the original concrete tree, to a correspondingly modified concrete tree. The challenge of understanding and designing these transformations—called *lenses*—arises from their asymmetric nature: information is discarded when mapping from concrete to abstract, and must be restored on the way back.

We identify a natural mathematical space of well-behaved lenses, in which the two components are constrained to fit together in a sensible way. We study definedness and continuity in this setting and state a precise connection with the classical theory of "update translation under a constant complement" from databases. We then instantiate our semantic framework as a small programming language, called Focal, whose expressions denote well-behaved lenses operating on trees. The primitives include familiar constructs from functional programming (composition, mapping, projection, recursion) together with some novel primitives for manipulating trees (splitting, pruning, pivoting, etc.). An extended example shows how Focal can be used to define a lens that translates between a native XHTML representation of browser bookmarks and a generic abstract bookmark format.

1. INTRODUCTION

Computing is full of situations where one wants to transform some structure into a different form—a *view*—in such a way that changes made to the view can be reflected back as updates to the original structure.

This paper addresses a specific instance of "updating through a view" that arises in a larger project called Harmony [25]. Harmony is a generic framework for synchronizing tree-structured data—a tool for propagating updates between different copies, possibly stored in different formats, of tree-shaped data structures. For example, Harmony can be used to synchronize the bookmark files of several different web browsers, allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and propagated to the others. The ultimate aim is to provide a platform on which a Harmony programmer can quickly assemble a high-quality synchronizer for a new type of tree-structured data stored in a standard low-level format such as XML. Other Harmony instances currently used daily or under development include synchronizers for calendars (Palm DateBook, ical, and iCalendar formats), address books, Keynote presentations, structured documents, and generic XML and HTML.

Views play a key role in Harmony: to synchronize disparate data formats, we define a single common abstract view as well as *lenses* that transform each concrete format into this abstract view. For example, we can synchronize a Mozilla bookmark file with an Explorer bookmark file by using appropriate lenses to transform each into an *abstract bookmark structure* and synchronizing the results. However, we are not done: we then need to take the updated abstract structures resulting from synchronization and transform them back into correspondingly updated concrete structures. To achieve this, a lens must include not one but *two* functions—one for extracting an abstract view from a concrete one and another for pushing an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *put* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *put* direction amounts to putting a new abstract part into an old concrete structure.

Not surprisingly, the tricky aspects of constructing lenses arise in the *put* direction. If the *get* part of a lens is a projection—i.e., information is suppressed when moving from concrete to abstract—then the *put* part must restore this information in some appropriate way. (We will see a concrete example shortly.) The difficulty is that there may, in general, be many ways of doing so.

Our approach to this problem is to design a language in which every expression simultaneously specifies both a *get* function and the corresponding *put*. All the primitives in this language denote lenses whose *get* and *put* functions fit together in a suitable sense, and all the combining forms preserve this property.

We begin by identifying a natural mathematical space of well-behaved lenses. There is a fair amount to be said about this space at a general level, even before we fix the domain of structures being transformed (trees) or the syntax for writing down transformations. First, we must phrase our basic definitions to allow lenses to be partial—i.e., to capture the fact that there may be structures to which a given lens cannot sensibly be applied. Second, we need some laws that express our intuitions about how the *get* and *put* parts of a lens should behave in concert. For example, if we use the *get* part of a lens to extract an abstract view a from a concrete

view c and then use the *put* part to push the very same a back into c, then we should get back to the original c. These laws must take partiality into account. Third, we must deal with the fact that we will later want to define lenses by recursion (because the trees that our lenses manipulate may in general have arbitrarily deep nested structure—e.g., when they represent directory hierarchies, bookmark folders, etc.). This raises familiar issues of monotonicity and continuity.

With these semantic foundations in place, we develop a syntax for constructing lenses for the specific domain of trees. Our surface language, Focal, comprises a collection of *primitive lenses* for tree transformations and lens *combinators* (composition, conditionals, mapping, etc.) that allow complex lenses to be built up from simpler ones. From these basic constructs, we can build a rich variety of useful derived forms—e.g., lenses for manipulating list-structured data encoded as trees.

We begin in Section 2 with a small example illustrating the fundamental ideas. Section 3 develops the semantic foundations of lenses in a general setting, addressing issues of partiality and continuity. Section 4 instantiates this generic framework with primitive lenses and lens combinators for our specific application domain of lenses over trees. Section 5 illustrates the use of these constructs in actual lens programming by walking through a substantial example derived from the Harmony bookmark synchronizer. Section 6 surveys a variety of related work from both the programming languages and the database literature and states a precise correspondence (amplified in [24]) between our well-behaved lenses and the closely related idea of "update translation under a constant complement" from databases. Section 7 sketches some directions for future research. For brevity, proofs are omitted; these can be found in an accompanying technical report [15].

2. A SMALL EXAMPLE

Suppose our concrete data source c is a small address book, represented as the following tree:

$$c \quad = \quad \begin{cases} \texttt{Pat} \mapsto & \begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array} \\ \texttt{Chris} \mapsto & \begin{array}{l} \texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org} \end{array} \end{cases}$$

We draw trees sideways to save space. Each curly brace denotes a node, and each "X ↦ ..." on the right of the curly brace denotes a child labeled X. In running text, we add closing braces to show where trees end. Also, to avoid clutter, when an edge leads to an empty tree, we usually omit the braces, the ↦ symbol and the final childless node—e.g., "333-4444" above actually stands for "{333-4444 ↦ {}}." In the parts of the paper that concern the surface language (i.e., everywhere except Section 3), we work with unordered, edge-labeled trees in which each node has at most one child of a given name. (Although our trees are unordered, we will see in Section 4.4 that they can also be used to represent and manipulate ordered data such as XML via a simple encoding.)

Suppose that, for some reason, we want to edit the data from this concrete tree in a simplified format, where each name is associated directly with a phone number.

$$a \quad = \quad \begin{array}{l} \texttt{Pat} \mapsto \texttt{333-4444} \\ \texttt{Chris} \mapsto \texttt{888-9999} \end{array}$$

Why would we want this? Perhaps because the edits are going to be performed by synchronizing this abstract tree with another replica of the same address book in which no URL information is recorded, or perhaps there is no synchronizer involved but the edits are going to be performed by a human who is only interested in phone information and whose screen should not be cluttered with URLs. Whatever the reason, we are going to make our changes to the abstract tree a, yielding a new abstract tree a' of the same form but with modified content.

For example, let us change Pat's phone number, drop Chris, and add a new friend, Jo.

$$a' \quad = \quad \begin{array}{l} \texttt{Pat} \mapsto \texttt{333-4321} \\ \texttt{Jo} \mapsto \texttt{555-6666} \end{array}$$

Note that we are only interested in the final tree a', not the actual sequence of edit operations that may have been used to transform a into a'. This design choice arises from the fact that synchronization in Harmony is based on the current states of the replicas, rather than on a trace of modifications (the tradeoffs between state-based and trace-based synchronizers are discussed in [26]).

Finally, we want to compute a new concrete tree c' reflecting the new abstract tree a'. That is, we want the parts of c' that were kept when calculating a (e.g., Pat's phone number) to be overwritten with the corresponding information from a', while the parts of c that were filtered out (e.g., Pat's URL) should have their values carried over from c.

$$c' \quad = \quad \begin{cases} \texttt{Pat} \mapsto & \begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4321} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array} \\ \texttt{Jo} \mapsto & \begin{array}{l} \texttt{Phone} \mapsto \texttt{555-6666} \\ \texttt{URL} \mapsto \texttt{http://google.com} \end{array} \end{cases}$$

We also need to "fill in" appropriate values for the parts of c' (in particular, Jo's URL) that were created in a' and for which c therefore contains no information. Here, we simply set the URL to a constant default, but in more complex situations we might want to compute it from other information.

Together, the transformations from c to a and from a' and c to c' form a lens. Our goal is to design a programming language that allows lenses to be described in a concise, natural, and mathematically coherent manner.

3. SEMANTIC FOUNDATIONS

While our surface language, Focal, is specialized for dealing with tree transformations, its semantic underpinnings can be presented in an abstract setting that is parameterized by the data structures manipulated by lenses, which we call *views*.[1] In this section, we simply assume we are given some set C of concrete views and some set A of abstract views; in Section 4 we will choose both of these to be the set of trees.

3.1 Basic structure

A lens is a pair of partial functions: one that gets an abstract view from a concrete view, and one that puts a new abstract view into an old concrete view to yield a new concrete view.

[1]Note that we use the word "view" here in a slightly different sense than some of the database papers that we cite, where a "view" is a *query* that maps concrete to abstract states—i.e., it is a function that, for each concrete database state, picks out a view in our sense.

3.1.1 Definition [Lenses]: A *lens* l comprises two partial functions: a *get* function from C to A, written $l\nearrow$, and a *put* function from $A \times C$ to C, written $l\searrow$.

We write $\mathsf{dom}(l\nearrow)$ for the subset of C on which $l\nearrow$ is defined and $\mathsf{dom}(l\searrow)$ for the subset of $A \times C$ on which $l\searrow$ is defined. We often say "put a into c" instead of "apply the *put* function to (a, c)." The intuition behind the notations $l\nearrow$ and $l\searrow$ is that the *get* part of a lens "lifts" an abstract view out of a concrete one, while the *put* part "pushes down" a new abstract view into an existing concrete view.

3.1.2 Definition [Well-behaved lenses]: A lens is *well behaved* iff its *get* and *put* functions obey the following laws:

$$(\textsc{GetPut})\quad c \in \mathsf{dom}(l\nearrow) \implies l\searrow(l\nearrow c,\, c) = c$$

$$(\textsc{PutGet})\quad (a,c) \in \mathsf{dom}(l\searrow) \implies l\nearrow(l\searrow(a,\, c)) = a$$

The GetPut law states that, if some abstract view obtained from a concrete view c is unmodified, putting it back into c must yield the same concrete view. This law also requires the *put* function to be defined on $(l\nearrow c, c)$ whenever $l\nearrow c$ is defined. The PutGet law states that the *put* function must capture all of the information contained in the abstract view: if putting a view a into a concrete view c yields a view c', then the abstract view obtained from c' is exactly a. This law also requires that the *get* function be defined at least on the range of the *put* function.

An example of a lens satisfying PutGet but not GetPut is the following. Let $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$, and define l as:

$$l\nearrow(s,n) = s$$
$$l\searrow(s',\,(s,n)) = (s',0)$$

Then $l\searrow(l\nearrow(s,1),\,(s,1)) = (s,0) \neq (s,1)$. Intuitively, this law fails because the *put* function has some "side effects": it modifies information from the concrete view that is not contained in the abstract view.

An example of a lens satisfying GetPut but not PutGet is the following. Let $C = \texttt{string}$ and $A = \texttt{string} \times \texttt{int}$, and define l as:

$$l\nearrow s = (s,0)$$
$$l\searrow((s',n),\, s) = s'$$

PutGet fails in this case because some information contained in the abstract view does not get propagated to the new concrete view. For example, $l\nearrow(l\searrow((s',1),\, s)) = l\nearrow s' = (s',0) \neq (s',1)$.

The GetPut and PutGet laws are essential, reflecting fundamental expectations about the behavior of lenses. Removing either law significantly weakens the semantic foundation. We may also optionally consider a third law, called PutPut:

$$(a,c) \in \mathsf{dom}(l\searrow) \implies l\searrow(a',\, l\searrow(a,\, c)) = l\searrow(a',\, c)$$

This law states that the effect of a sequence of two *put*s is just the effect of the second, as long as the first *put* is defined. We say that a well-behaved lens that also satisfies PutPut is *very well behaved*. Both well-behaved and very well behaved lenses correspond (modulo some details about partiality) to well-known classes of "update translators" from the classical database literature; see Section 6.

The PutPut law intuitively states that a series of changes to an abstract view may be applied incrementally or all at once, resulting in the same final concrete view in both cases. This is a natural and intuitive constraint, and the foundational development in this section is valid for both well-behaved and very well behaved variants of lenses. However, when we come to defining Focal in Section 4, we will drop PutPut because one of our most important lens combinators, `map`, fails to satisfy it. This point is discussed in more detail in Section 4.2.

3.2 Basic Properties

We now explore some simple consequences of the lens laws.

Let f be a partial function from $A \times C$ to C. Abusing terminology slightly, we will say that f is *injective* if it is injective in its first argument wherever it is defined—i.e., if, for all views a, a', and c such that $f(a,c)$ and $f(a',c)$ are defined, we have $a \neq a' \implies f(a,c) \neq f(a',c)$.

The following lemma provides an easy way to show that a lens is *not* well behaved. We used it many times while designing the Focal surface language, to quickly generate and test candidate lenses.

3.2.1 Lemma: Let l be a well-behaved lens. Then the function $l\searrow$ is injective, and, for all $(a,c) \in \mathsf{dom}(l\searrow)$, we have $l\searrow(a,\, l\searrow(a,\, c)) = l\searrow(a,\, c)$.

Conversely, the next Lemma shows that for each injective *put* function satisfying the additional condition of Lemma 3.2.1, there is a unique *get* function that makes a well-behaved lens.

3.2.2 Lemma: Let $l\searrow$ be an injective partial function from $A \times C$ to C such that for all $(a,c) \in \mathsf{dom}(l\searrow)$ we have $l\searrow(a,\, l\searrow(a,\, c)) = l\searrow(a,\, c)$. Then there is exactly one function $l\nearrow$ such that $l = (l\nearrow, l\searrow)$ is a well-behaved lens[2].

This lemma shows that we can define a well-behaved lens simply by giving a suitable *put* function. However, in most cases, we have found it more convenient to write out both *get* and *put* functions explicitly and directly check the original laws.

3.3 Recursion

Since our lens framework will be instantiated with trees, and since trees in many interesting application domains may have unbounded depth (e.g., a bookmark item can be either a link or a folder containing a collection of bookmark items), we will need to define lenses by recursion. Our final task for this foundational section is to set up the necessary structure for interpreting recursive definitions in the surface language.

The development follows familiar lines. We introduce an information ordering on lenses and show that the set of lenses equipped with this ordering is a complete partial order (cpo). We then apply standard tools from domain theory, giving us interpretations of a variety of common syntactic forms from programming languages—in particular, functional abstraction and application (i.e., "higher-order lenses") and lenses defined by (single or mutual) recursion.

We say that a lens l' is *more informative* than a lens l, written $l \prec l'$, if the *put* function of l' is an extension of the *put* function of l—that is, if $l'\searrow$ is defined on a larger domain

[2]This *get* function is total on the range of $l\searrow$; it is defined by taking $l\nearrow(l\searrow(a,\, c)) = a$ for all $(a,c) \in \mathsf{dom}(l\searrow)$.

than $l\searrow$ and if the two *put* functions are equal on their common domain, $\mathsf{dom}(l\searrow)$. As the *put* function determines the *get* function (Lemma 3.2.2), this ordering only needs to take the *put* function into account. This relation is a partial order.

A *cpo* is an ordered set in which every increasing chain of elements has a least upper bound in the set. A *cpo with bottom* is a cpo that contains an element, $\perp$, that is smaller than every other element. In our setting, this is the lens whose *put* and *get* functions are undefined everywhere.

3.3.1 Theorem: Let $\mathcal{L}$ be the set of well-behaved lenses between C and A. Then $(\mathcal{L}, \prec)$ is a cpo with bottom.

We can now apply standard domain theory to interpret a variety of constructs for defining continuous lens combinators. In particular, every continuous function on well-behaved lenses has a least fixed point that is a well-behaved lens.

4. TRANSFORMING TREES

We now describe our surface language, Focal. We first introduce convenient notations for trees and simple operations on them. We then present some primitive lenses and lens combinators, which we assemble to create several derived lenses. We finally describe an encoding of lists as trees and introduce some specialized derived lenses for manipulating them. We give intuitions and small examples along the way; an extended example using most of the lenses together appears in Section 5. The expressiveness of the language is discussed in Section 7.1.

4.1 Trees

From now on, we will be working with the set of finite, unordered, edge-labeled trees, with labels drawn from some infinite set of *names*—e.g., character strings. Each tree can be viewed as a partial function from names to other trees. We write $\mathsf{dom}(t)$ for the domain of a tree t—i.e. the set of the names of its immediate children. The variables a, c, d, and t range over trees; by convention, we use a for trees that are thought of as abstract and c or d for concrete trees.

The Harmony system targets data stored in XML (among other formats). However, the form of trees that we use internally is much simpler than XML, which associates each node with both unordered children (attributes) and ordered ones (sub-elements). We show in Section 5 how XML trees can be encoded into ours. We chose to restrict Focal to unordered trees for engineering reasons: experience has shown that the (huge) resulting reduction in the complexity of the lens *definitions* far outweighs the modest increase in complexity of lens *programs* due to manipulating XML via this encoding instead of primitively.

There are cases where we need to apply a *put* function, but where no old concrete tree is available (as we saw with `Jo`'s `URL` in Section 2). To deal with these cases, we enrich the set of trees with a special placeholder Ω, pronounced "missing." Intuitively, $l\searrow(a, \Omega)$ means "create a *new* concrete tree from the information in the abstract tree a." Formally, we write T for the set of trees and T_Ω for $T \cup \{\Omega\}$, and take both C and A in the definition of lenses to be T_Ω. By convention, Ω is only used as the second argument to the *put* function: in all of the lenses defined below, we maintain the invariants that $\Omega \notin \mathsf{dom}(l\nearrow)$, that $(\Omega, c) \notin \mathsf{dom}(l\searrow)$ for any c, that $\Omega \notin \mathsf{ran}(l\nearrow)$, and that $\Omega \notin \mathsf{ran}(l\searrow)$. To simplify some lens definitions below, we sometimes treat Ω as a tree with $\mathsf{dom}(\Omega) = \emptyset$. There are other, formally equivalent, ways of handling missing concrete trees. We chose this one for pragmatic reasons, after exploring several alternatives; its advantages are discussed below, after the definition of the `map` combinator.

Let t be the tree that associates t_1 to $\mathtt{n}_1$, t_2 to $\mathtt{n}_2$, ..., and t_k to $\mathtt{n}_k$. We write t as $\{\mathtt{n}_1 \mapsto t_1 \ldots \mathtt{n}_k \mapsto t_k\}$ when it appears in running text, and as an opening brace and a vertical list of name/subtree pairs (dropping the closing curly brace) when it appears in a displayed figure. We write "{}" (in running text) or "{" (in displays) for the empty tree, and $t(n)$ for the tree associated to name n in t.

We often define trees by extension. For instance, let t be a tree and p be a set of names such that $p \subseteq \mathsf{dom}(t)$; we may define a tree w as $w = \{n \mapsto t(n) \mid n \in p\}$. When p is a set of names (not necessarily a subset of $\mathsf{dom}(t)$), we write $t|_p$ for the tree $\{n \mapsto t(n) \mid n \in p \cap \mathsf{dom}(t)\}$. By convention, we take $\Omega|_p = \Omega$ for any p (this shortens some definitions below), and we write $\overline{p}$ for the complement of the set p.

It is also convenient to define a notation for *concatenation* of trees. Let $t, t' \in T$, with $\mathsf{dom}(t) \cap \mathsf{dom}(t') = \emptyset$. We write $t + t'$ (in running text) or $\begin{matrix} t \\ t' \end{matrix}$ (in displays) for the tree

$$\begin{cases} n \mapsto t(n) & n \in \mathsf{dom}(t) \\ n \mapsto t'(n) & n \in \mathsf{dom}(t'). \end{cases}$$

A *value* is a tree of the special form $\{k \mapsto \{\}\}$, often written just k. For instance, the phone number $\{\mathtt{333\text{-}4444} \mapsto \{\}\}$ in the example of Section 2 is a value.

4.2 Primitive Lenses

In this section we define several atomic lenses and lens combinators (we will often just say "lenses" for both). We begin with a few generic lenses that do not depend on the data structure being trees: the identity lens, the constant lens, and sequential composition of lenses. We then introduce several lenses that inspect and manipulate tree structures—three atomic lenses (`rename`, `hoist`, and `pivot`) and two lens combinators (`xfork` and `map`).

All lenses introduced in this section, with the exception of the `const` lens, preserve all information when building the abstract tree in the *get* direction. The two lens combinators also preserve all information when applied to information-preserving lenses. Most lenses thus do not need to use the concrete tree in the *put* direction.

Every atomic lens defined in this section is very well behaved, and every lens combinator is continuous and preserves well-behavedness. In fact, most lens combinators preserve very well behavedness, with the single exception of `map`.

In the following, we assume that lens application is *strict*, i.e. the application of a lens to some arguments is defined only if the computations it depends on (in particular other lens applications) are all defined.

Generic Lenses

The simplest lens is the identity. It does nothing in the *get* direction and copies the whole abstract tree in the *put* direction.

$$\begin{array}{rcl} \texttt{id} \nearrow c & = & c \\ \texttt{id} \searrow (a,\, c) & = & a \end{array}$$

Another simple lens is the constant lens, `const` t d, which transforms any tree into the provided constant t in the *get* direction. In the *put* direction, it is defined iff the abstract tree is equal to t.[3] In this case, the *put* function of `const` simply restores the old concrete tree if it is available; if the old concrete tree is missing (Ω), the *put* function returns a default tree d.

$$\begin{array}{rcll} (\texttt{const}\ t\ d) \nearrow c & = & t & \\ (\texttt{const}\ t\ d) \searrow (a,\, c) & = & c & \text{if } a = t \text{ and } c \neq \Omega \\ & & d & \text{if } a = t \text{ and } c = \Omega \\ & & \text{undef.} & \text{otherwise} \end{array}$$

The lens composition combinator $l; k$ places two lenses l and k in sequence.

$$\begin{array}{rcll} (l; k) \nearrow c & = & k \nearrow l \nearrow c & \\ (l; k) \searrow (a,\, c) & = & l \searrow (k \searrow (a,\, l \nearrow c),\, c) & \text{if } c \neq \Omega \\ & & l \searrow (k \searrow (a,\, \Omega),\, \Omega) & \text{if } c = \Omega \end{array}$$

The *get* direction applies the *get* function of l to yield a first abstract tree, on which the *get* function of k is applied. In the *put* direction, if a concrete tree is available, the *put* functions are applied in turn: first, the *put* function of k is used to put a into the concrete tree that the *get* of k was applied to, i.e., $l \nearrow c$; the result of this *put* is then put into c using the *put* function of l. If the concrete tree is missing, then k is used to put a into the missing tree and then l to put the result again into the missing tree.

Atomic Lenses

The `rename` lens changes the names of the immediate children of a tree according to some bijection b on names.

$$\begin{array}{rcl} (\texttt{rename}\ b) \nearrow c & = & b(n) \mapsto c(n) \\ (\texttt{rename}\ b) \searrow (a,\, c) & = & b^{-1}(n) \mapsto a(n) \end{array}$$

In examples, we use the notation

```
{'h3' = 'name'  'dl' = 'contents'}
```

for the bijection that maps `'h3'` to `'name'`, `'name'` to `'h3'`, `'dl'` to `'contents'`, and `'contents'` to `'dl'`.

In practice, one also sometimes wants a "deep rename" lens that changes all the names in a tree, rather than just the immediate children of the root; this can be defined from `rename` using `map` and recursion.

The lens `hoist` n is used to "shorten" a tree by removing an edge. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *put* direction, the value of the concrete tree is ignored and a new tree is created, with a single edge n pointing to the given abstract tree.

$$\begin{array}{rcll} (\texttt{hoist}\ n) \nearrow c & = & t & \text{if } c = \ n \mapsto t \\ & & \text{undef.} & \text{otherwise} \\ (\texttt{hoist}\ n) \searrow (a,\, c) & = & n \mapsto a & \end{array}$$

[3] By the PutGet law, this is the only possible definition: if `const` t $d \searrow (a,\, c)$ is defined, then `const` t $d \nearrow (\texttt{const}\ t\ d \searrow (a,\, c)) = a = t$.

The lens `pivot` n rearranges the structure at the top of a tree.

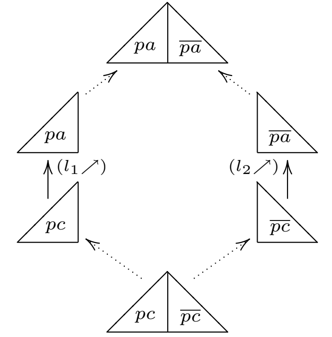
$$\begin{array}{ll} n \mapsto & k \\ t & \end{array} \quad \text{becomes} \quad k \mapsto t$$

Intuitively, the value $\{k \mapsto \{\}\}$ under n represents a *key* k (a name uniquely identifying the tree) for the rest of the tree t. The *get* function of `pivot` returns a tree where k points directly to t. The *put* function performs the reverse transformation, ignoring the old concrete tree.

$$\begin{array}{rcll} (\texttt{pivot}\ n) \nearrow c & = & k \mapsto t & \text{if } c = \begin{array}{ll} n \mapsto & k \\ t & \end{array} \\ & & \text{undef.} & \text{otherwise} \\ (\texttt{pivot}\ n) \searrow (a,\, c) & = & \begin{array}{ll} n \mapsto & k \\ t & \end{array} & \text{if } a = \ k \mapsto t \\ & & \text{undef.} & \text{otherwise} \end{array}$$

Lens Combinators

The lens combinator `xfork` is used to apply different lenses to different parts of a tree. Intuitively, it splits a tree into two parts according to the names of its immediate children. It then applies one lens to the first part and another lens to the other part and concatenates the results. Formally, `xfork` takes as arguments two predicates on names and two lenses. The *get* direction of `xfork` pc pa l_1 l_2 can be visualized as in the inset figure (the concrete tree is at the bottom).



The triangles labeled pc denote trees whose immediate child edges have labels satisfying pc; dotted arrows represent splitting or concatenating trees. The result of applying $l_1 \nearrow$ to $c|_{pc}$ (the tree formed by dropping the immediate children of c whose names do not satisfy pc) must be a tree whose top-level labels satisfy the predicate pa, and, similarly the result of applying $l_2 \nearrow$ to $c|_{\overline{pc}}$ must satisfy $\overline{pa}$. That is, the lenses l_1 and l_2 are allowed to change the sets of names in the trees they are given, but each must map from its own part of pc to its own part of pa. Conversely, in the *put* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Formally:

$$\begin{array}{ll} (\texttt{xfork}\ pc\ pa\ l_1\ l_2) \nearrow c \ = & \\ \quad (l_1 \nearrow c|_{pc}) + (l_2 \nearrow c|_{\overline{pc}}) & \text{if (1)} \\ \quad \text{undef.} & \text{otherwise} \\ (\texttt{xfork}\ pc\ pa\ l_1\ l_2) \searrow (a,\, c) \ = & \\ \quad (l_1 \searrow (a|_{pa},\, c|_{pc})) + (l_2 \searrow (a|_{\overline{pa}},\, c|_{\overline{pc}})) & \text{if (2)} \\ \quad \text{undef.} & \text{otherwise} \end{array}$$

$$\begin{array}{lll} (1) & & \mathsf{dom}(l_1 \nearrow c|_{pc}) \subseteq pa \\ & \text{and} & \mathsf{dom}(l_2 \nearrow c|_{\overline{pc}}) \subseteq \overline{pa} \\ (2) & & \mathsf{dom}(l_1 \searrow (a|_{pa},\, c|_{pc})) \subseteq pc \\ & \text{and} & \mathsf{dom}(l_2 \searrow (a|_{\overline{pa}},\, c|_{\overline{pc}})) \subseteq \overline{pc} \end{array}$$

We now define our last primitive lens and only iterator, `map`. This is a combinator that, in the *get* direction, applies a given lens l one level deeper in the tree, leaving the top of

the tree intact.

$$\begin{cases} n_1 \mapsto t_1 \\ \dots \\ n_k \mapsto t_k \end{cases} \text{becomes} \begin{cases} n_1 \mapsto l \nearrow t_1 \\ \dots \\ n_k \mapsto l \nearrow t_k \end{cases}$$

The *put* direction of `map` is more interesting. In the simple case where a and c have equal domains, the definition is straightforward:

$$(\texttt{map}\ l) \searrow \left(\begin{cases} n_1 \mapsto t_1 \\ \dots \\ n_k \mapsto t_k \end{cases}, \begin{cases} n_1 \mapsto t'_1 \\ \dots \\ n_k \mapsto t'_k \end{cases} \right) = \begin{cases} n_1 \mapsto l \searrow (t_1, t'_1) \\ \dots \\ n_k \mapsto l \searrow (t_k, t'_k) \end{cases}$$

The general case is a bit more involved. If $(\texttt{map}\ l) \searrow (a, c)$ is defined, then, by rule PutGet, we should have $(\texttt{map}\ l) \nearrow ((\texttt{map}\ l) \searrow (a, c)) = a$. Thus we necessarily have $\mathsf{dom}((\texttt{map}\ l) \searrow (a, c)) = \mathsf{dom}(a)$. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as described above. Children bearing names that only occur in $\mathsf{dom}(c)$ are dropped. Children of a that have names that only appear in $\mathsf{dom}(a)$ need to be passed through l so that they can be included in the result; to do this, we need to *put* them into some concrete tree. No suitable tree can be obtained from c, as there is no corresponding child in c, so instead these abstract trees are put into the missing tree Ω.

$$\begin{array}{rcl} (\texttt{map}\ l) \nearrow c & = & n \mapsto l \nearrow c(n) \quad n \in \mathsf{dom}(c) \\ (\texttt{map}\ l) \searrow (a, c) & = & \begin{cases} n \mapsto l \searrow (a(n), c(n)) \\ \qquad n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto l \searrow (a(n), \Omega) \\ \qquad n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{cases} \end{array}$$

Interestingly, the `map` combinator does not obey the PutPut law. Consider a lens l and $(a, c) \in \mathsf{dom}(l\searrow)$ such that $l \searrow (a, c) \neq l \searrow (a, \Omega)$. We have

$$\begin{array}{rl} & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\}, ((\texttt{map}\ l) \searrow (\{\}, \{\texttt{n} \mapsto c\}))) \\ = & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\}, \{\}) \\ = & \{\texttt{n} \mapsto l \searrow (a, \Omega)\} \\ \neq & \{\texttt{n} \mapsto l \searrow (a, c)\} \\ = & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\}, \{\texttt{n} \mapsto c\}). \end{array}$$

Intuitively, there is a difference between, on the one hand, modifying a child n and, on the other, removing it and then adding it back: in the first case, any information in the concrete view that is "projected away" in the abstract view will be carried along to the new concrete view; in the second, such information will be replaced with default values.

Another point to note is the relation between the `map` lens combinator and the missing tree Ω. The *put* function of every other lens combinator only results in a *put* into the missing tree if the combinator itself is called on Ω. In the case of `map` l, calling its *put* function on some a and c where c is not the missing tree may result in the application of the *put* of l to Ω if a has some children that are not in c. In an earlier version of Focal, we dealt with missing children by providing a default concrete child tree to `map`, which would be used when no actual concrete tree was available. However, we discovered that, in practice, it is often difficult to find a single default concrete tree that fits all possible abstract trees, particularly because of `xfork` (where different lenses are applied to different parts of the tree) and recursion (where the depth of a tree is unknown). We tried parameterizing this default concrete tree by the abstract tree and the lens, but noticed that most primitive lenses ignore the concrete tree when defining the *put* function, as enough information is available in the abstract tree. The natural choice for a concrete tree parameterized by a and l was thus $l \searrow (a, \Omega)$, for some special tree Ω. The only lens for which the *put* function needs to be defined on Ω is `const`, as it is the only lens that discards information. This led us to the present design, where the `const` lens expects a default tree d. This approach is much more "local" than the others we tried, since one only needs to provide a default tree at the exact point where information is discarded.

4.3 Derived Lenses

We now derive some useful lenses from the primitive ones of Section 4.2. Most of the lenses defined here and in Section 4.4 are used in the example of Section 5. Note that these derived lenses are all well behaved by construction.

In many uses of `xfork`, the predicates specifying where to split the concrete tree and where to split the abstract tree are identical. We define the simpler `fork` as:

$$\texttt{fork}\ p\ l_1\ l_2 = \texttt{xfork}\ p\ p\ l_1\ l_2$$

We may now define a lens that discards all of the children of a tree that do not satisfy some predicate p:

$$\texttt{filter}\ p\ d = \texttt{fork}\ p\ \texttt{id}\ (\texttt{const}\ \{\}\ d)$$

In the *get* direction, this lens takes a concrete tree, keeps the part of the tree whose children have names in p (using the lens `id`), and throws away the rest of the tree (using the lens `const` $\{\}$ d). The default tree d is used when putting an abstract tree into a missing concrete tree. It provides a default for the information that does not appear in the abstract tree and is necessary to build a concrete tree.

Another way to filter, or prune, a tree is to explicitly specify a child that should be removed from the tree:

$$\texttt{prune}\ n\ d = \texttt{fork}\ \overline{\{n\}}\ \texttt{id}\ (\texttt{const}\ \{\}\ \{n \mapsto d\})$$

This lens is very similar to `filter`, with two differences: the name given is the one of the child to be removed, thus the predicate "all other names" $\overline{\{n\}}$ must be built, and the default tree is the one to go under n if the concrete tree is missing.

The next lens is useful to focus on a single child n:

$$\texttt{focus}\ n\ d = (\texttt{filter}\ \{n\}\ d);\ (\texttt{hoist}\ n)$$

In the *get* direction, it filters away all other children, then removes the edge n to return the corresponding subtree. As usual, the default tree is only used in case of creation, where it is the default for children that have been filtered away.

It is often useful to restrict the action of `map` to a subset of all children of a tree using a predicate p.

$$\texttt{mapp}\ p\ l = \texttt{fork}\ p\ (\texttt{map}\ l)\ \texttt{id}$$

This lens splits the tree in two according to the predicate p, applies `map` to the first half, and passes the rest through unmodified.

In order to apply different lenses to different parts of the tree and concatenate the results, we define the lens

dispatch, parameterized on a list of tuples each containing a concrete predicate, an abstract predicate, and a lens:

$$\texttt{dispatch}\ [\,] = \texttt{id}$$
$$\texttt{dispatch}\ (pc, pa, l) :: rest = \texttt{xfork}\ pc\ pa\ l\ (\texttt{dispatch}\ rest)$$

In the *get* direction, dispatch considers the first tuple (pc, pa, l). It splits the concrete tree according to pc and applies l to it. It recurses with the rest of the tuples and the rest of the tree, and concatenates the results back together. A typical use of dispatch is as a conditional, assuming all the lenses it uses return the empty tree when given the empty tree (see for instance the item lens in Figure 3).

4.4 Lists

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we represent lists, using a standard cons cell encoding, and introduce some derived lenses to manipulate them.

4.4.1 Definition: A tree t is a list iff either it is empty or else it has exactly two children, one named $*h$ and another named $*t$, and $t(*t)$ is also a list.

In the following, we use the lighter notation $[t_1 \dots t_n]$ (writing, in displays, t_1 through t_n vertically and dropping the closing bracket) for the tree

$$\left\{\begin{array}{l} \texttt{*h} \mapsto \texttt{t}_1 \\ \texttt{*t} \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto \texttt{t}_2 \\ \texttt{*t} \mapsto \quad \dots \mapsto \begin{array}{l} \texttt{*h} \mapsto \texttt{t}_n \\ \texttt{*t}. \end{array} \end{array}\right. \end{array}\right.$$

We now define some lenses to work on lists. The first ones extract the head or the tail of the list.

$$\texttt{hd}\ d = \texttt{focus}\ \{\texttt{*h}\}\ \{\texttt{*t} \mapsto d\}$$
$$\texttt{tl}\ d = \texttt{focus}\ \{\texttt{*t}\}\ \{\texttt{*h} \mapsto d\}$$

The lens hd expects a default tree which will be the tail of the created tree if the concrete tree is missing. In the get direction, the lens hd returns the tree under name *h. The lens tl works analogously

The map_list lens iterates over a list, applying its argument to every element of the list:

$$\texttt{map_list}\ l = \texttt{mapp}\ \{\texttt{*h}\}\ l;\ \texttt{mapp}\ \{\texttt{*t}\}\ (\texttt{map_list}\ l)$$

This lens simply applies l to every child named *h and recurses on every child named *t.

We close by defining a lens that transforms a list into a "bush," flattening it. Such a lens may only be defined on lists of trees that have pairwise-disjoint domains. Depending on whether the order matters if creation occurs, two such lenses may be defined. One lens, flatten (not shown here), does not care about the order when putting an abstract tree into a missing concrete tree, and the resulting list has an arbitrary order. The other, hoist_list, expects a list of (pairwise-disjoint) predicates describing the elements of the list: each view in the list must satisfy the corresponding predicate in the predicate list. The predicate list indicates, in the put direction, the position in the list where each child of the flattened bush should be put.

$$\begin{array}{rl} \texttt{hoist_list}\ [\,] = & \texttt{id} \\ \texttt{hoist_list}\ p :: rest = & \texttt{xfork}\ \{\texttt{*h}\}\ p \\ & (\texttt{hoist}\ \{\texttt{*h}\}) \\ & (\texttt{hoist}\ \{\texttt{*t}\};\ \texttt{hoist_list}\ rest) \end{array}$$

```
{* ->
 [{html -> {* ->
   [{head -> {* -> [{title ->
                     {* ->
                      [{PCDATA -> Bookmarks}]}}]}}
    {body -> {* ->
     [{h3 -> {* ->
              [{PCDATA -> Bookmarks Folder}]}}
      {dl -> {* ->
       [{dt -> {* ->
          [{a -> {* -> [{PCDATA -> Google}]
                 add_date -> 1032458036
                 href -> www.google.com}}]}}
        {dd -> {* ->
          [{h3 -> {* -> [{PCDATA ->
                           Conferences Folder}]}}
           {dl -> {* ->
             [{dt -> {* ->
                [{a ->
                   {* -> [{PCDATA -> POPL}]
                    add_date -> 1032528670
                    href -> cristal.inria.fr/POPL2004
                   }}]}}]}}]}}]}}]}}]}}]}}]}}
```

Figure 1: Bookmarks (concrete tree)

5. A BOOKMARK LENS

With these definitions in hand, we are ready to develop an extended example of programming in Focal. The example comes from a demo application of our data synchronization framework, Harmony, in which bookmark information from diverse browsers, including Internet Explorer, Mozilla, Safari, Camino, and OmniWeb, is synchronized by transforming each format from its concrete native representation into a common abstract form. We show here a slightly simplified form of the Mozilla lens, which handles the HTML-based bookmark format used by Netscape and its descendants.

The overall path taken by the bookmark data through the Harmony system can be described as follows. Harmony first uses a generic HTML reader to transform the HTML bookmark file into an isomorphic concrete tree. This concrete tree is then transformed, using the *get* direction of the bookmark lens, into an abstract "generic bookmark tree." The abstract tree is synchronized with some other abstract bookmark tree (obtained from some other bookmark file by transforming its native format using an appropriate lens, not shown here), yielding a new abstract tree, which is transformed into a new concrete tree by passing it back through the *put* direction of the bookmark lens (supplying the original concrete tree as the second argument). Finally, the new concrete tree is written back out to the filesystem as an HTML file. We now discuss these transformations in detail.

Abstractly, bookmark data has this recursive structure: an *item* is either a *link*, with a name and a url, or a *folder* with a name and a contents, which is a list of items. Concretely, in HTML, a bookmark item is represented by a <dt> element containing an <a> element whose href attribute gives the link's url and whose content defines the name. The <a> element also includes an add_date attribute, which we have chosen not to reflect in the abstract form because it is

```
{name -> Bookmarks Folder
 contents ->
  [{link -> {name -> Google
             url -> www.google.com}}
   {folder ->
     {name -> Conferences Folder
      contents ->
       [{link ->
         {name -> POPL
          url -> cristal.inria.fr/POPL2004}}]}}}]}
```

Figure 2: Bookmarks (abstract tree)

```
link = rename {dt = link};
       map (hoist *;
            hd {};
            hoist a;
            rename {href = url  * = name};
            prune add_date {$today};
            mapp {name} (hd {}; hoist PCDATA))

folder = rename {dd = folder};
         map (hoist *; folder_contents)

folder_contents =
   hoist_list [{h3} {dl}];
   rename {h3 = name  dl = contents};
   mapp {name} (hoist *; hd {}; hoist PCDATA);
   mapp {contents} (hoist *; map_list item)

item =
  dispatch [({dd},{folder},folder)
            ({dt},{link},link)]

bookmarks =
  hoist *; hd {}; hoist html; hoist *;
  tl {head -> {* -> [{title -> {* ->
                  [{PCDATA -> Bookmarks}]}}]}};
  hd {}; hoist body; hoist *;
  folder_contents
```

Figure 3: Bookmark lenses

not supported by all browsers. A bookmark folder is represented by a `<dd>` element containing an `<h3>` header (giving the folder's name) followed by a `<dl>` list containing the sequence of items in the folder. The whole HTML bookmark file follows the standard `<head>`/`<body>` form, where the contents of the `<body>` have the format of a bookmark folder, without the enclosing `<dd>` tag.

The generic HTML reader and writer know nothing about the specifics of the bookmark format; they simply transform between HTML syntax and trees in a mechanical way, mapping an HTML element named `tag`, with attributes `attr1` to `attrm` and sub-elements `subelt1` to `subeltn`,

```
<tag attr1="val1" ... attrm="valm">
   subelt1 ... subeltn
</tag>
```

into a tree of this form:

$$\left\{\ \texttt{tag} \mapsto \left\{\begin{array}{l}\texttt{attr1} \mapsto \texttt{val1} \\ \quad\vdots \\ \texttt{attrm} \mapsto \texttt{valm} \\ * \mapsto \left[\begin{array}{l}\texttt{subelt1} \\ \quad\vdots \\ \texttt{subeltn}\end{array}\right.\end{array}\right.\right.$$

Note that the sub-elements are placed in a *list* under a distinguished child named `*`. This preserves their ordering from the original HTML file. (The ordering of sub-elements is sometimes important—e.g., in the present example, it is important to maintain the ordering of the items within a bookmark folder. Since the HTML reader and writer are generic, they *always* record the ordering from the the original HTML in the tree, leaving it up to whatever lens is applied to the tree to throw away ordering information where it is not needed; the `flatten` lens described in Section 4.4 provides one convenient way to do this.) A leaf of the HTML document—i.e., a "parsed character data" element containing a text string `str`—is converted to a tree of the form `{PCDATA -> str}`. Figure 1 shows a tree representing a small bookmark file.

The transformation between this concrete tree and the abstract bookmark tree shown in Figure 2 is implemented by means of the collection of lenses shown in Figure 3. Most of the work of these lenses involves (in the *get* direction) stripping out various extraneous structure and then renaming certain branches to have the desired "field names." Conversely, the *put* direction restores the original names and rebuilds the necessary structure.

In practice, composite lenses are developed incrementally, gradually massaging the trees into the correct shape. Figure 4 shows the process of developing the `link` lens above by transforming the representation of the HTML `<dt>` element containing a link into the desired abstract form. At each level of the structure, tree branches are relabeled with `rename`, undesired structure is removed with `prune`, `hoist`, and/or `hd`, and then work is continued at a lower level via `map` or `mapp`.

The *put* direction of the `link` lens restores original names and structure automatically, by composing the *put* directions of the constituent lenses of `link` in turn. For example, suppose the abstract link tree at the bottom right of Figure 4 were updated to change `name` from `Google` to `Search-Engine`. The *put* direction of the `hoist PCDATA` lens, corresponding to moving from step *ix* to step *viii* in Figure 4, puts the updated string in the abstract tree back into a more concrete tree by replacing `Search-Engine` with `{PCDATA -> Search-Engine}`. In the transition from step *vii* to step *vi*, the *put* direction of `prune add_date {$today}` utilizes the concrete tree to restore the value, `add_date -> 1032458036`, projected away in the abstract tree. If the concrete tree had been Ω, in the case of a new bookmark added in the new abstract tree, then the default argument `{$today}` would have been used to fill in today's date. (Formally, the whole set of lenses is parameterized on the variable `$today`, which ranges over names.)

In a manner similar to the `link` lens, the *get* direction of the `folder` lens renames the `<dd>` tag to `folder`, then proceeds to separate out the folder name and its contents, strip-

Step	Lens expression	Resulting abstract tree (from 'get')

i:	`id`	`{dt -> {* ->`
		`[{a -> {* -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`href -> www.google.com}}]}}`

ii:	`rename {dt = link}`	`{link -> {* ->`
		`[{a -> {* -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`href -> www.google.com}}]}}`

iii:	`rename {dt = link};`
	`map (hoist *)`
		`{link -> [{a -> {* -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`href -> www.google.com}}]}}`

iv:	`rename {dt = link};`
	`map (hoist *;`
	`hd {})`
		`{link -> {a -> {* -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`href -> www.google.com}}}`

v:	`rename {dt = link};`
	`map (...;  hd {};`
	`hoist a)`
		`{link -> {* -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`href -> www.google.com}}`

vi:	`rename {dt = link};`
	`map (... ; hoist a;`
	`rename {* = name, href = url})`
		`{link -> {name -> [{PCDATA -> Google}]`
		`add_date -> 1032458036`
		`url -> www.google.com}}`

vii:	`rename {dt = link};`
	`map (...; rename {* = name, href = url};`
	`prune add_date {$today})`
		`{link -> {name -> [{PCDATA -> Google}]`
		`url -> www.google.com}}`

viii:	`rename {dt = link};`
	`map (...; prune add_date {$today};`
	`mapp {name} (hd {}))`
		`{link -> {name -> {PCDATA -> Google}`
		`url -> www.google.com}}`

ix:	`rename {dt = link};`
	`map (...; mapp {name} (hd {};`
	`hoist PCDATA))`
		`{link -> {name -> Google`
		`url -> www.google.com}}`

Figure 4: Building up a link lens incrementally.

ping out undesired structure where necessary. (We have separated out the content processing into the `folder_contents` lens for the sake of code reuse, since the top-level bookmark file itself looks almost like a folder.) Note the use of `hoist_list` instead of `flatten` to access the `<h3>` and `<dl>` tags containing the folder name and contents respectively; although the order of these two tags does not matter to us, it matters to Mozilla, so we want to ensure that the *put* direction of the lens puts them to their proper position in case of creation. Finally, we use `map_list` to iterate over the contents.

The `item` lens processes one element of a folder's contents; this element might be a link or another folder, so we want to either apply the `link` lens or the `folder` lens. Fortunately, we can distinguish them by whether they are contained within a `<dd>` element or a `<dt>` element, and we use `dispatch` to call the correct sublens.

The main lens is `bookmarks`, which (in the *get* direction) takes a whole concrete bookmark tree, strips off the boilerplate header information using a combination of `hoist`, `hd`, and `tl`, and then invokes `folder_contents` to deal with the rest. The huge default tree supplied to the `tl` lens corresponds to the head tag of the html document, which is filtered away in the abstract bookmark format. This default tree would be used to recreate a well-formed head tag if it was missing in the original concrete tree.

6. RELATED WORK

Focal is the product of a long odyssey through a large design space, driven by the practical needs of the Harmony system as it evolved. The overall architecture of Harmony and the role of lenses in building synchronizers for various forms of data are described in [25], along with a detailed discussion of related work on synchronization.

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our "programming language treatment" of these structures has led us to a formulation that is arguably simpler (transforming states rather than "update functions") and more refined in its treatment of partiality. Our formulation is also novel in considering the issue of continuity (not addressed in earlier work), thus supporting a rich variety of surface language structures including definition by recursion.

The idea of defining a programming language for constructing bi-directional transformations has also been explored previously. However, we appear to be the first to have connected it with a formal semantic foundation, choosing primitives that can be combined into composite lenses whose well-behavedness is guaranteed by construction.

6.1 Foundations of View Update

The foundations of view update translation were studied intensively by database researchers in the late '70s and '80s. This thread of work is closely related to our semantics of lenses in Section 3.

Dayal and Bernstein [11] gave a seminal formal account of the theory of "correct update translation." Their notion of "exactly performing an update" corresponds to our PutGet law. Their "absence of side effects" corresponds to our

GetPut and PutPut laws. Their requirement of preservation of semantic consistency corresponds to the partiality of our *put* functions.

Bancilhon and Spyratos [6] developed an elegant semantic characterization of the update translation problem, introducing the notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement". In general, a view may have many complements, each corresponding to a possible strategy for translating view updates to database updates. The problem of translating view updates then becomes a problem of finding, for a given view, a suitable complement.

Gottlob, Paolini, and Zicari [14] offered a more refined theory based on a syntactic translation of view updates. They identified a hierarchy of restricted cases of their framework, the most permissive form being their "dynamic views" and the most restrictive, called "cyclic views with constant complement," being formally equivalent to Bancilhon and Spyratos's update translators.

Recent work by Lechtenbörger [19] establishes that translations of view updates under constant complements are possible precisely if view update effects may be undone using further view updates.

In a companion paper [24], we establish a precise correspondence between our definition of lenses and the structures studied by Bancilhon and Spyratos and by Gottlob, Paolini, and Zicari. Briefly, our set of very well behaved lenses is isomorphic to the set of translators under constant complement in the sense of Bacilhon and Spyratos, while our set of well-behaved lenses is isomorphic to the set of *dynamic views* in the sense of Gottlob, Paolini, and Zicari. To be precise, both of these results must be qualified by an additional condition regarding partiality. The frameworks of Bacilhon and Spyratos and of Gottlob, Paolini, and Zicari are both formulated in terms of translating *update functions* on A into update functions on C, i.e., their *put* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *put* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *put* functions) are defined only over some particular chosen set U of abstract update functions, not over all functions from A to A. These update translators return *total* functions from C to C. Our *put* functions, on the other hand, are more general as they are defined over all abstract states and return *partial* functions from C to C. Finally, the *get* functions of lenses are allowed to be partial, whereas the corresponding functions (called *views*) in the other two frameworks are assumed to be total. In order to make the correspondences tight, the sets of well-behaved and very well behaved lenses need to be restricted to subsets that are "total" in a suitable sense.

The view update problem has also been studied in the context of object-oriented databases. School, Laasch, and Tresch [27] restrict the notion of views to queries that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update of objects of the database.

6.2 Updates for Relational Views

Research on view update translation in the database literature has tended to focus on taking an existing language (e.g., relational algebra) for defining *get* functions and then considering how to infer (either automatically or with some programmer assistance) corresponding *put* functions. By contrast, we have designed a completely new language in which the definitions of *get* and *put* go hand-in-hand. Our approach also goes beyond classical work in the relational setting by directly transforming and updating tree-structured data, rather than flat relations. (Of course, trees can be encoded as relations, but it is not clear how the operations we consider could be expressed using the recursion-free relational languages considered in previous work in this area.) Conversely, the problem we address here is easier because our language lacks an analog of relational join, a major source of update ambiguity in the relational world. (We have not yet encountered a need for join in the setting in which Focal is being used—transforming trees to render them suitable for synchronization.) We briefly review the most relevant research from the relational setting.

Masunaga [20] described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller [17] outlined all possible strategies for handling updates to a select-project-join view, and showed that these are exactly the set of translations that satisfy a small set of intuitive criteria. He later [18] proposed allowing users to choose an update translator at view definition time by engaging in an interactive dialog with the system and answering questions about potential sources of ambiguity in update translation. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold [7] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [21] presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid concrete states, the tool predicted whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

Atzeni and Torlone [5, 4] described a tool for translating views and observed that if one can translate any concrete view to and from a *meta-model* (shared abstract view), one then gets bi-directional transformations between any pair of concrete views. They limited themselves to mappings where the concrete and abstract views are isomorphic.

A variety of complexity results have been shown for different versions of the view update inference problem. In one of the earliest, Cosmadakis and Papadimitriou [9, 10] considered the view update problem for a single relation, where the view is a projection of the underlying relation, and showed that there are polynomial time algorithms for determining whether insertions, deletions, and tuple replacements to a projection view are translatable into concrete updates. More recently, Buneman, Khanna, and Tan [8] established a variety of intractability results for the problem of inferring "minimal" view updates in the relational setting for query languages that include both join and either project

or union.

Another body of work that is sometimes mentioned in connection with view update translation is the problem of *incremental view maintenance* (e.g., [3])—efficiently recalculating an abstract view after a small update to the underlying concrete view. Although the phrase "view update problem" is sometimes (confusingly) used for work in this domain, there is little technical connection with our problem of translating view updates to updates on an underlying concrete structure.

6.3 Languages for View Update

In the programming languages literature, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *put* functions) appear in Oles' category of "state shapes" [23] and in Hofmann and Pierce's work on "positive subtyping" [16]. Another related idea, proposed by Wadler [30], extended algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators. This is essentially the special case of our lenses in which the *get* and *put* functions always form an isomorphism.

Abiteboul, Cluet, and Milo [1] defined a declarative language for describing *correspondences* between parts of trees in a data forest. In turn, these correspondence rules can be used to translate one tree format into another through nondeterministic Prolog-like computation; however, this process requires an isomorphism between the two data formats (again, a special case of our lenses).

The same authors [2] later defined a system for bidirectional transformations based around the concept of *structuring schemas* (parse grammars annotated with semantic information). Thus their *get* involved parsing, whereas their *put* consisted of "unparsing." Again, to resolve ambiguous abstract updates, they restrict themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

Ohori and Tajima [22] developed a statically-typed polymorphic record calculus for defining views on object-oriented databases. They specifically restricted which fields of a view are updatable, allowing only those with a ground (simple) type to be updated, whereas our lenses can accommodate structural updates as well.

6.4 Updates and Trees

There have been many proposals for query languages for trees (e.g., XQuery [13] and its forerunners, UnQL, StruQL, and Lorel), but these either do not consider the view update problem at all or else handle update only in situations where the abstract and concrete views are isomorphic.

For example, Braganholo, Heuser, and Vittori [12], and Braganholo, Davidson, and Heuser [29] studied the problem of updating relational databases "presented as XML." Their solution requires a 1:1 mapping between XML view elements and objects in the database, to make view updates unambiguous.

Tatarinov, Ives, Halevy, and Weld [28] described a mechanism for translating updates on XML structures that are stored in an underlying relational database. In this setting there is again an isomorphism between the concrete relational database and the abstract XML view, so updates are unambiguous—rather, the problem is choosing the most efficient way of translating a given XML update into a sequence of relational operations.

7. FUTURE WORK

Our interest in bi-directional tree transformations arose in the context of our data synchronization framework, Harmony. We are currently developing a number of additional synchronizers as instances of Harmony; this exercise provides valuable stress-testing for both the concrete Focal language and its formal foundations. Further ahead, we see a number of opportunities for future work.

7.1 Expressiveness

We have not yet addressed the important question of the *expressiveness* or *completeness* of our approach.

To begin with, the Focal language—i.e., the particular collection of lenses and lens combinators described in Section 4—is certainly *not* complete, in the sense that any tree transformation that might ever be required in the context of XML synchronization can be described in it. For example, Focal includes no facilities for string manipulation (e.g., concatenation of node labels), for arithmetic, etc., etc. Fortunately, our goal for Focal in the context of the Harmony system is modest: we do not hope to capture all conceivable tree transformations, but just to be able to express 99% of the ones needed to build a range of synchronizer instances. As in most high-level programming languages, we expect, occasionally, to need to drop down to a lower level language to code some special-purpose "native function" to be used as a new primitive in the high-level language. In Focal, this simply means implementing a pair of *get* and *put* functions and declaring this pair to be a lens; the obligation of verifying that the lens laws are satisfied rests on the programmer. In the early days of constructing and using the Harmony prototype, we added new lenses (and lens combinators) fairly often. Lately, we do it rather rarely.

Nevertheless, from a theoretical perspective the question of expressiveness is of obvious interest—and, we have found, quite challenging.

To get at the question, we might first think of trying to show that, if we project out the *get* functions from all the lenses expressible in Focal (or some other language sharing Focal's basic goal of guaranteeing that the lens laws are satisfied by construction for all expressible lenses), we obtain a class C of tree transformations that is previously known or, at least, easily characterizable. That is, we would like to show that Focal is *complete* for C.

However, by itself, this form of completeness is uninteresting. To see this, take an arbitrary set G of *get* functions—i.e., of partial functions from C to A. For every $g \in G$, we can define a *put* function p from $A \times C$ to C such that (g, p) is immediately a well-behaved lens: for every $a, c \in A \times C$, if $g(c) = a$ then $p(a, c) = c$, otherwise p is undefined. This lens language is complete (the first projection of the set of lenses contains every desired *get* function), but useless: for every concrete view c, a put function can only push back the abstract view obtained from c, and no other. In other words, these lenses solve the view update problem for the trivial case where there is no update of the view!

Clearly, in order for completeness to mean anything interesting, we must make stronger demands on the *put* functions of a complete set of lenses: they should be able to put back "enough" abstract views into each concrete view. We can formalize this notion by saying that a lens is *closed* if every abstract view generated by its *get* function can be put into every concrete view accepted by its *get* function,

that is when $\mathsf{dom}(l\searrow) \supseteq \mathsf{ran}(l\nearrow) \times \mathsf{dom}(l\nearrow)$. For our synchronization application, this property is both necessary and sufficient: if one replica has not changed since the previous synchronization, then Harmony will completely overwrite its abstract view with the abstract view from the other replica; thus, the abstract argument to the *put* function may be any abstract view that can be obtained from the *get* function (of the other replica).

We conjecture that the lenses defined in this paper are all closed. In view of this, it seems reasonable to add the closure requirement as an additional lens law. Indeed, doing so actually seems to simplify the development in several places.

Having discussed closure, we must again face the question of completeness. The approach we are currently pursuing is twofold. First, continue a practical exploration based on the development of many synchronizers, adding primitives to Focal as they become necessary. Second, consider a more theoretical approach based on the classical framework of *tree transducers* (over ranked, node-labeled trees)—i.e., try to find some tree-transducer-like syntax for binary *put* functions (we call them "tree combiners"), with the property that, for every definable tree combiner p, we can find an ordinary tree transducer g (drawn from some well-known class such as linear, top-down, finite-lookahead transducers) such that (g, p) satisfies the lens laws. So far, though, we have been able to create such correspondences only for extremely simple classes of transducers.

7.2 Static Analysis

From a programming point of view, a static type system for views and Focal programs would be useful, particularly when building large and complicated lenses. Such a type system would allow the programmer to express the well-formedness of certain views (*e.g.*, "this tree should have exactly one child, named *foo* (because I want to `hoist` it)").

We also plan to address other questions related to static analysis of lenses. For instance, can we characterize the complexity of Focal programs? Is there an algebraic theory of lens combinators that would underpin optimization of Focal programs in the same way that the relational algebra and its algebraic theory are used to optimize relational database queries? (For example, the combinators we have described here have the property that `map` l_1; `map` l_2 = `map` $(l_1; l_2)$ for all l_1 and l_2, but the latter should run substantially faster.)

7.3 Lens Inference

It would be useful to generate lens programs automatically from schemas for concrete and abstract views, or by inference from a set of pairs of inputs and desired outputs ("programming by example"). Such a facility could perhaps do most of the work for a programmer wanting to add synchronization support for a new application (where the abstract form was already defined, for example), leaving just a few spots to fill in.

7.4 Beyond Trees

A growing body of work deals with the problem of translating between heterogeneous representations of similar data to enable different applications to cooperate. Such representations (e.g. directed graphs and XML) are a superset of the concrete views (namely trees) that we handle. Although much of this work (one way transformations that do not address the update problem) is not directly relevant to Focal, it may be useful as a set of examples against which to compare the expressiveness of Focal. Further, one class of proposed solutions uses schema matching (as well as representation mapping and model mapping) to perform all or part of the translation automatically. We may be able to employ similar methods to automatically construct lenses to translate between two given views.

Finally, we would like to experiment with instantiating our semantic framework with relations instead of trees, thereby establishing a closer link with existing research in databases.

Acknowledgements

The Harmony project was begun in collaboration with Zhe Yang; Zhe contributed numerous insights whose generic material can be found (generally in much-recombined form) in this paper. Trevor Jim provided the initial push to start the project by observing that the next step beyond the Unison file synchronizer (of which Trevor was a co-designer) would be synchronizing XML. Conversations with Nate Foster, Owen Gunden, Martin Hofmann, Zack Ives, Sanjeev Khanna, William Lovas, Kate Moore, Cyrus Najmabadi, and Steve Zdancewic helped us sharpen our ideas. Serge Abiteboul, Zack Ives, Dan Suciu, and Phil Wadler pointed us to related work. We would also like to thank Karthik Bhargavan, Vanessa Braganholo, Peter Buneman, Owen Gunden, Michael Hicks, Zack Ives, Trevor Jim, Kate Moore, Wang-Chiew Tan, Stephen Tse, Zhe Yang, and the anonymous referees for very helpful comments on earlier drafts of this paper.

The Harmony project is supported by the National Science Foundation, grant ITR-0113226, *Principles and Practice of Synchronization*.

8. REFERENCES

[1] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proceedings of 6th Int. Conf. on Database Theory (ICDT)*, 1997.

[2] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structure files. *VLDB Journal*, 7(2):96–114, 1998.

[3] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. Very Large Data Bases (VLDB)*, 1998.

[4] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of EDBT'96, LNCS 1057*, 1996.

[5] P. Atzeni and R. Torlone. MDM: a multiple-data model tool for the management of heterogeneous database schemes. In *Proceedings of ACM SIGMOD, Exhibition Section*, pages 528–531, 1997.

[6] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

[7] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *PODS'91*, pages 248–257, 1991.

[8] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *PODS'02*, pages 150–158, 2002.

[9] S. S. Cosmadakis. Translating updates of relational data base views. Master's thesis, Massachusetts Institute of Technology, 1983. MIT-LCS-TR-284.

[10] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[11] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, September 1982.

[12] V. de Paula Braganholo, C. A. Heuser, and C. R. M. Vittori. Updating relational databases through XML views. In *Proc. 3rd Int. Conf. on Information Integration and Web-based Applications and Services (IIWAS)*, 2001.

[13] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, 2001.

[14] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *TODS*, 13(4):486–524, 1988.

[15] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, 2003.

[16] M. Hofmann and B. Pierce. Positive subtyping. In *POPL'95*, 1995.

[17] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS'85*, 1985.

[18] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB'86*, 1986.

[19] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–55. ACM, June 9–12 2003. San Diego, CA.

[20] Y. Masunaga. A relational database view update translation mechanism. In *VLDB'84*, 1984.

[21] C. M. B. Medeiros and F. W. Tompa. Understanding the implications of view update policies. In *VLDB'85*, 1985.

[22] A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *PODS'94*, 1994.

[23] F. J. Oles. Type algebras, functor categories, and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambrige University Press, 1985.

[24] B. C. Pierce and A. Schmitt. Lenses and view update translation. Manuscript; available at `http://www.cis.upenn.edu/~bcpierce/harmony`, 2003.

[25] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing Harmony to optimistic replication: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.

[26] B. C. Pierce and J. Vouillon. How to specify a file synchronizer. Technical Report MS-CIS-03-36, Dept. of CIS, University of Pennsylvania, 2003.

[27] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Yasunga, editors, *Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, number 566. Springer, 1991.

[28] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[29] Vanessa Braganholo and Susan Davidson and Carlos Heuser. On the updatability of XML views over relational databases. In *WebDB 2003*, 2003.

[30] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL'87*. 1987.

Recent BRICS Notes Series Publications

NS-03-4 **Michael I. Schwartzbach, editor.** ***PLAN-X 2004 Informal Proceedings,*** **(Venice, Italy, 13 January, 2004), December 2003. ii+95.**

NS-03-3 **Luca Aceto, Zoltán Ésik, Willem Jan Fokkink, and Anna Ingólfsdóttir, editors.** ***Slide Reprints from the Workshop on Process Algebra: Open Problems and Future Directions, PA '03,*** **(Bologna, Italy, 21–25 July, 2003), November 2003. vi+138.**

NS-03-2 **Luca Aceto.** ***Some of My Favourite Results in Classic Process Algebra*****. September 2003. 21 pp. To appear in the** ***Bulletin of the EATCS*****, volume 81, October 2003.**

NS-03-1 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Kurtz Alexander, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '03,*** **(Marseille, France, September 6, 2003), August 2003. vi+54.**

NS-02-8 **Peter D. Mosses, editor.** ***Proceedings of the Fourth International Workshop on Action Semantics, AS 2002,*** **(Copenhagen, Denmark, July 21, 2002), December 2002. vi+133 pp.**

NS-02-7 **Anders Møller.** ***Document Structure Description 2.0*****. December 2002. 29 pp.**

NS-02-6 **Aske Simon Christensen and Anders Møller.** ***JWIG User Manual*****. October 2002. 35 pp.**

NS-02-5 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '02,*** **(Toulouse, France, October 30–31, 2002), October 2002. vi+97.**

NS-02-4 **Daniel Gudbjartsson, Anna Ingólfsdóttir, and Augustin Kong.** ***An BDD-Based Implementation of the Allegro Software*****. August 2002. 2 pp.**

NS-02-3 **Walter Vogler and Kim G. Larsen, editors.** ***Preliminary Proceedings of the 3rd International Workshop on Models for Time-Critical Systems, MTCS '02,*** **(Brno, Czech Republic, August 24, 2002), August 2002. vi+141 pp.**