Visibly Pushdown Automata for Streaming XML

Viraj Kumar[*] P. Madhusudan Mahesh Viswanathan[†]
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{kumar, madhu, vmahesh}@cs.uiuc.edu

ABSTRACT

We propose the study of visibly pushdown automata (Vpa) for processing XML documents. Vpas are pushdown automata where the input determines the stack operation, and XML documents are naturally visibly pushdown with the Vpa pushing onto the stack on open-tags and popping the stack on close-tags. In this paper we demonstrate the power and ease visibly pushdown automata give in the design of *streaming* algorithms for XML documents.

We study the problems of *type-checking* streaming XML documents against SDTD schemas, and the problem of *typing* tags in a streaming XML document according to an SDTD schema. For the latter problem, we consider both pre-order typing and post-order typing of a document, which dynamically determines types at open-tags and close-tags respectively as soon as they are met. We also generalize the problems of pre-order and post-order typing to *prefix querying*. We show that a deterministic Vpa yields an algorithm to the problem of answering in one pass the set of all answers to any query that has the property that a node satisfying the query is determined solely by the prefix leading to the node. All the streaming algorithms we develop in this paper are based on the construction of deterministic Vpas, and hence, for any fixed problem, the algorithms process each element of the input in constant time, and use space $O(d)$, where d is the depth of the document.

Categories and Subject Descriptors

H.2.1 [**Information Systems**]: Logical Design—*schema and subschema*; H.2.3 [**Information Systems**]: Lanuages—*data description languages (DDL), query languages*; F.1.1 [**Theory of Computation**]: Models of Computation—*automata*

General Terms

Algorithms, Theory

Keywords

XML, streaming algorithms, schema, typing, query, pushdown automata

[*]Supported by NSF CCF 04-29639
[†]Supported by NSF CCF 04-48178

1. INTRODUCTION

The eXtensible Markup Language (XML) has become the standard for data exchange, particularly on the web [26, 1]. An XML document is a linear representation of hierarchically structured data. The latter is best viewed as a tree, with the edges of the tree encoding the hierarchy. Hierarchically structured data can be encoded into linear structures in different ways; XML is the popular way, where the tree is represented using open-tags and close-tags, and the paranthesis structure defined by the tags encodes the hierarchical information. Naturally, the study of XML has concentrated on the tree it represents; for example, document types are represented using the parse-trees generated by specialized DTDs [22], XML query languages have modalities like "parent" and "child" that refer to the tree edges, and tree automata (over unranked trees) have been used to model and solve decision problems for XML [25, 20, 9, 11, 16].

An XML document is a linear word structure, and though formalisms based on the tree representation of the document are useful, they are awkward when designing algorithms for processing XML. While one could take an XML document, convert it into a tree, and then apply algorithms for the tree, there are several settings where such changes of representation is not feasible.

One class of applications for which building the tree representation of a document is infeasible is in the processing of *streams* of large XML documents. Streaming algorithms process input data as and when they arrive (for example, streaming stock market data), and have to process them with as little time and space as possible. Building trees representing the document would make little sense. Not surprisingly, streaming algorithms proposed for XML do not build the tree [7, 13], and in fact theoretical algorithms for streaming XML are often based on *pushdown automata* (not tree automata) [13, 24, 11]. However, there are no standard automata models for streaming XML; pushdown automata are not appealing as they do not define a robust tractable class of languages: they are not closed under complement, inclusion is undecidable, etc.

The main thesis of this paper is that *visibly pushdown automata* (Vpa [4]) are the right model for processing streaming XML. A Vpa is a pushdown automaton whose stack operations are determined by the input letter it reads. An XML document is best seen as a *nested* word: a linear structure (word) with a nesting relation formed by associating open-tags with their matching close-tags. A visibly pushdown automaton can reconstruct the nesting relation by pushing onto the stack on open-tags and popping from it on close-tags. Visibly pushdown automata, unlike pushdown automata, define a robust class of languages. The class is closed under all boolean operations, admits decidable procedures for problems such as inclusion and emptiness, and we can show that it is *precisely* as powerful as regular tree languages accepting the tree representation of the data [4]. The most important feature is that these automata are *determinizable*, which will help us in building streaming algorithms. Furthermore, recent work on visibly pushdown languages have exposed several interesting results including a congruence based characterization and minimization results that use a modular notion of these automata. These results affirm the view that this class is very similar in tractability and robustness to that of regular word languages.

In this paper, we argue that visibly pushdown automata are an apt model for XML by studying algorithms for streaming XML documents. We demonstrate the use of VPAs in solving two main problems: (a) *type-checking* streaming XML documents against SDTDs (specialized data-type definitions) [22], and of assigning types to tags while streaming the document, and (b) in querying *prefix*-based queries on XML streams, where the problem is to anwer the set of all positions satisfying queries that are determined by the prefix leading to a position.

An SDTD (which extends the notion of DTDs) defines a class of XML documents using an extended context-free grammar. The first result in this paper is a visibly pushdown automaton model tailored for XML: we define a notion of modular VPA called XVPA (X for XML!) that exactly corresponds to schema defined by SDTDs. This gives a natural machine characterization of SDTDs. By applying results in the literature on visibly pushdown automata to XVPAs, in particular (a) determinability of VPAs [4], (b) the expressive power of deterministic single-entry modular VPAs [3], and (c) congruence-based minimization results for complete multi-entry modular VPAs [14], we derive (minimal) streaming machines.

We first study the problem of *type-checking* streaming XML documents against SDTDs. Using the fact that VPAs can be *determinized* to get single-entry modular VPAs (SEVPAs), we show that these automata type-check streaming XML documents against SDTD-defined types. Further, by combining the set of single-entry modules into a single module with multiple entries, SEVPAs can be interpreted as *complete multi-entry* modular VPAs, for which we have recently obtained a minimization result [14]. Unlike the pushdown machines constructed in [24], we obtain provably minimal recognizers that optimize the number of states in two ways: they combine strucural conditions shared both between specializations of the *same* tag, and across *different* tags.

We then turn to the problem of *typing* a document. For a document that conforms to an SDTD, the types associated with a tag are the specializations that admit a parsing of the document with respect to the SDTD. Typing a document facilitates querying (for example, a query asking for all tags that have a particular specialization can be answered using such a typing). We study the problem of assigning types to tags while reading a streaming XML document.

A tag is said to be pre-order typed if its type is unique and can be assigned as soon as meeting its open-tag; similarly, a tag is said to be post-order typed if its unique type is determined at the close-tag. An SDTD is said to be pre-order typed if *all* its tags are pre-order typed in *all* documents that accord with the SDTD. Pre-order typed SDTDs correspond to restrained competition grammars and have been syntactically and semantically characterized [18, 17]. We show *automata-theoretic* characterizations of pre- and post-order typed schemas [1]. The characterization of pre-order typed schemas is beautifully simple in terms of XVPAs: a schema is pre-order typed if and only if it can be accepted by a *deterministic XVPA*. This is a very natural and intuitive characterization– when reading an open-tag, the XVPA must be able to determine the type of the tag and hence call the appropriate module deterministically.

We then turn to *dynamically* typing streaming documents. Even though a schema may not be pre-order typed, some or all of its tags may be pre-order typed in a *particular* document. We show that for any SDTD, we can build a deterministic VPA that reads streaming XML documents, and dynamically pre-order types the open-tags whenever possible. More precisely, if in the current document read, the type of a tag is determined by the prefix of the document read thus far, then the automaton will determine its type when reading the open-tag. We also prove a similar result for partially post-order typing documents at close-tags for streaming documents. We also show that checking if a tag is pre-order typed in an SDTD (across all documents) is decidable in polynomial time.

Finally, we generalize the results on pre-order and post-order typing to *prefix querying*. We consider monadic queries expressed in a sub-logic of monadic second order logic (MSO), which we call Pre-MSO. Queries here are MSO formulas with one free variable, where all quantified positions are required to occur before the position denoted by the free variable. We show that for such queries, determining if a position in a document satisfies a query depends solely on the tags seen before the position. Furthermore, for any such query we can build a deterministic VPA that can answer such queries while processing an XML document in a streaming fashion. Since the problems of dynamically typing (pre-order and post-order) can be expressed in our logic Pre-MSO, our observations on pre-order and post-order typing can be seen as a consequence of these results. However, our direct constructions for the pre-order and post-order typing are more efficient in terms of the resources used by the resulting streaming algorithm. There has been a previous characterization of queries that can be answered by a 1-pass streaming algorithm (see [19]) in terms of constraint systems and pushdown forest automata; our results in terms of Pre-MSO and VPAs can be seen a reformulation of the Neumann-Seidl result in terms of logic and pushdown machines on words.

In summary, VPAs emerge as a simple apt model for processing XML, particularly in the design of streaming algorithms. The rich set of results obtained for visibly pushdown languages, including determinization and minimization of modular machines, find immediate use in designing streaming algorithms for XML.

We have already several other results which we have not reported here (including results on *typability*, improved complexity of static type-checking, etc.) that we have proved using the VPA model; these and proofs of the theorems in this paper can be found in the technical report [15].

Related work in automata theory: Viewing XML doc-

[1]By "schemas", we mean a collection of documents defined by an SDTD. We do not mean the language "XML Schema".

uments as trees, visibly pushdown automata correspond to automata that process trees by reading them on an infix traversal, using a stack to push whenever they go down a left branch, and popping it when they return to process the right branch. The notion of visibly pushdown automata have been used implicitly in processing XML streams in the literature (in [24, 9, 19, 10] and as XPush machines in [11]). Note that there is only *one* copy of the automaton processing the tree (in contrast to tree automata). An alternate model defined in the literature is *tree-walking* automata [2, 21], which essentially has only one copy of the automaton but can walk up and down the input tree. However these automata do not have access to a stack and are *strictly weaker* than tree automata in expressive power [6]. In contrast, visibly pushdown automata capture the entire class of regular tree languages [4].

2. PRELIMINARIES

Let Σ be a fixed finite alphabet of "open tags", and let $\overline{\Sigma} = \{\overline{c} \mid c \in \Sigma\}$ be the corresponding alphabet of "close tags". Let $\widehat{\Sigma} = (\Sigma \cup \overline{\Sigma})$. XML documents will be treated as words over the input alphabet $\widehat{\Sigma}$.

A *well-matched* word is any word generated by the grammar: $W \to cW\overline{c}$, $W \to WW$, $W \to \epsilon$, where we have a rule $W \to cW\overline{c}$ for every $c \in \Sigma$. The grammars and automata we consider in this paper will always accept only languages of well-matched words. The set of all well-matched words over $\widehat{\Sigma}$ will be denoted by $WM(\widehat{\Sigma})$. A word $u \in \widehat{\Sigma}^*$ is said to have *matched closing tags* if there is some $w \in WM(\widehat{\Sigma})$ such that u is a prefix of w. The set of all words with matched returns (closing tags) will be denoted by $MR(\widehat{\Sigma})$.

Document-type definitions (DTDs) define restrictions on the structure of documents. These definitions hence describe languages over $\widehat{\Sigma}$, that correspond to the documents that accord with the document type. Since DTDs are not powerful enough, we use in this paper an extended version of DTDs called *specialized document-type definitions* (SDTDs) [22]. We give pushdown automata based descriptions of these languages, which are a subclass of context-free languages.

An SDTD is essentially a context-free grammar; however, each non-terminal of the SDTD is associated with a *tag*, and the idea is that every non-terminal implicitly generates the open-tag (c) and close-tag ($\overline{c}$) whenever the non-terminal is expanded. Non-terminals are called "specializations" in the XML context, and we will also call them "modules" in this paper.

Let M be a finite set of *specializations* or *modules* and let $\mu : M \to \Sigma$ be a *surjective* mapping. Elements of $\mu^{-1}(c)$ correspond to specializations of symbol c, and μ is the mapping between the specialized alphabet M and the unspecialized alphabet Σ.

SDTDs will be specified using extended context-free grammars. An extended context-free grammar d is a set of rules that map each $m \in M$ to a regular expression over M. We now define its semantics. Let us denote $\overline{M} = \{\overline{m} \mid m \in M\}$. For every $m \in M$, we define $L_d^{\mathrm{spl}}(m) \subset (M \cup \overline{M})^*$ as the smallest sets such that

$$L_d^{\mathrm{spl}}(m) \supseteq \{m.x.\overline{m} \mid x \in R_d(w), w \in d(m)\}$$

where: (1) $R_d(\epsilon) = \{\epsilon\}$, and (2) $R_d(m.w) = L_d^{\mathrm{spl}}(m).R_d(w)$.

The language over unspecialized symbols that m defines is then $L_d(m) = \{\mu(w) \mid w \in L_d^{\mathrm{spl}}(m)\}$, where μ is extended to words over $(M \cup \overline{M})^*$ in the obvious way: $\mu(\overline{m}) = \overline{\mu(m)}$ and $\mu(m.w) = \mu(m).\mu(w)$.

Definition 1 (SDTD). *An SDTD over (Σ, M, μ) is a tuple (d, m_0), where d is a mapping from M to regular expressions over M, and m_0 is the* start symbol. *The language of an SDTD (d, m_0) is defined as $L_d(m_0)$. The* size *of an SDTD (d, m_0) is defined as the sum of the lengths of the regular expressions defined by d.*

Example 1. *Let $\Sigma = \{movie, vhs, dvd, title, lang, subtitle\}$, and let the specialized alphabet be*

$$M = \{Movie, VHS, DVD, Title, Lang, Unisub, Multsub\}$$

where $\mu : M \to \Sigma$ is defined so that Unisub and Multsub are specializations of subtitle, and capitalized names are specializations of their uncapitalized counterparts. Finally, let d be defined as follows:

$$\begin{array}{rcl} d(Movie) &=& VHS + DVD \\ d(VHS) &=& Title.Unisub \\ d(DVD) &=& Title.(Unisub + Multsub) \\ d(Title) &=& \epsilon \\ d(Unisub) &=& Lang \\ d(Multsub) &=& Lang.Lang^{+} \\ d(Lang) &=& \epsilon \end{array}$$

Then $(d, Movie)$ is an SDTD over (Σ, M, μ) accepting words like:

$$movie.dvd.title.\overline{title}.subtitle.lang.\overline{lang}.lang.\overline{lang}.\overline{subtitle}.\overline{dvd}.\overline{movie}$$

Type-checking and typing XML documents

The two problems we consider in this paper are type-checking streaming XML documents, and pre- and post-order typing of XML documents. We define these notions now.

The type-checking problem is to check, given an SDTD, whether an input document belongs to the language of the SDTD. In the streaming context, we assume that the document is presented as a word, and the type-checking must be accomplished reading the word only once, left to right, and using as little time and space as possible.

Given an SDTD, and a document that accords to the type defined by the SDTD, a tag c in the document can get different types depending on how the document parses. More precisely, if $w \in \widehat{\Sigma}^*$ belongs to the language of an SDTD (d, m_0), and $w[i] = c \in \Sigma$, then we say that (w, i) has type m (where m is a specialization), if there exists $x \in L_d^{\mathrm{spl}}(m_0)$ such that $\mu(x) = w$ and $x[i] = m$. In general, (w, i) can have many types with respect to an SDTD.

An occurrence of an open (or close) tag in a document is prefix typed (with respect to a schema) if the tag's type is determined by the prefix of the document till that point. Pre-order typing refers to open tags being prefix typed while post-order typing refers to close tags being prefix typed.

Definition 2. *For an open or close tag $a \in \Sigma \cup \overline{\Sigma}$, we say a is* prefix typed *at position i in $w \in WM(\widehat{\Sigma})$, if $w = uav$, $|ua| = i$, and if there is a unique $m \in \mu^{-1}(a)$ such that for all $v' \in \widehat{\Sigma}^*$, whenever $uav' \in L_d(m_0)$ and $(uav', |ua|)$ has type m', then $m' = m$. In this case, we say that a has prefix-type m at position i in w.*

A tag $c \in \Sigma$ is pre-order typed *(post-order typed) in an SDTD (d, m_0) if for every document $w \in L_d(m_0)$ and every* *position $i \leq |w|$ such that $w[i] = c$ (resp. $w[i] = \overline{c}$), c (resp. $\overline{c}$) is prefix-typed at position i in w.*

An SDTD (d, m_0) is pre-order typed *(post-order typed) if every tag is pre-order typed (resp. post-order typed).*

Said in other words, a position i is prefix-typed in w if a streaming algorithm that reads w can determine the (unique) type of (w, i) at position i.

We study three problems related to typing:

- Automata characterization: which kind of VPAs capture SDTDs that are pre-order and post-order typed.
- Dynamic pre-order typing: Can we build an automaton that streams input and determines the type of open-tags (or close-tags) as soon as it meets them, provided the type has been determined by the prefix read till that point.
- Given an SDTD and a tag c, can we effectively decide if in *all* documents conforming to the SDTD, the type of every occurrence of the open-tag (or close-tag) corresponding to c is prefix-typed?

Visibly pushdown automata

The languages defined by SDTDs do not encompass all context-free languages. Viewing an SDTD as a normal grammar, a rule in the SDTD of the form $m_a \rightarrow m_b.m_c$, where $\mu(m_a) = a$, $\mu(m_b) = b$ and $\mu(m_c) = c$, translates to a rule $m_a \rightarrow a.m_b.m_c.\overline{a}$. In other words, each non-terminal is "guarded" by a tag that occurs before and after the expansion of the non-terminal. The usual translation of this grammar into a pushdown automaton will result in a machine that pushes at the open-tags and pops at the close-tags. Visibly pushdown automata are precisely these kind of restricted machines [4]. Since VPAs were first motivated in the program analysis context, the symbols on which the automaton pushes and pops are called *calls* and *returns*, instead of open-tags and close-tags.

DEFINITION 3 (VPA). *A visibly pushdown automaton (VPA) over $(\Sigma, \overline{\Sigma})$ is a tuple $A = (Q, q_0, Q^F, \Gamma, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q^F \subseteq Q$ is the set of final states, Γ is a finite stack alphabet, and $\delta = \delta^{\text{call}} \cup \delta^{\text{ret}}$ is the transition relation, where:*

- $\delta^{\text{call}} \subseteq ((Q \times \Sigma) \times (Q \times \Gamma))$;
- $\delta^{\text{ret}} \subseteq ((Q \times \overline{\Sigma} \times \Gamma) \times Q)$.

We denote a transition $(q, c, q', \gamma) \in \delta^{\text{call}}$ as $q \xrightarrow{c/\gamma} q'$, and a transition $(q, \overline{c}, \gamma, q') \in \delta^{\text{ret}}$ as $q \xrightarrow{\overline{c}/\gamma} q'$. A transition $q \xrightarrow{c/\gamma} q'$ is a push-transition, where the automaton reading c changes state from q to q', pushing γ onto the stack. Similarly, a transition $q \xrightarrow{\overline{c}/\gamma} q'$ is a pop-transition, where on reading $\overline{c}$ with γ on the top of the stack, the automaton pops γ off the stack and changes state from q to q'.

A *configuration* of a VPA A is a pair $(q, s) \in Q \times (\Gamma^*.\perp)$, where $\perp$ is a special bottom-of-stack symbol ($\perp \notin \Gamma$). If $a \in \widehat{\Sigma}$, we say that $(q_1, s_1) \xrightarrow{a}_A (q_2, s_2)$ if and only if one of the following conditions are true:

- $a = c \in \Sigma$, $s_2 = \gamma.s_1$ and $(q_1, c, q_2, \gamma) \in \delta^{\text{call}}$, or
- $a = \overline{c} \in \overline{\Sigma}$, $s_1 = \gamma.s_2$ and $(q_1, \overline{c}, \gamma, q_2) \in \delta^{\text{ret}}$.

We extend the definition of $\xrightarrow{a}_A$ to words over $\widehat{\Sigma}^*$ in the natural manner. The language $L(A)$ accepted by VPA A is the set of words $w \in \widehat{\Sigma}^*$ such that $(q_0, \perp) \xrightarrow{w}_A (q, \perp)$ for some $q \in Q^F$. A language L is called a *visibly pushdown language* (VPL) if there some VPA A such that $L = L(A)$.

We now review some of the basic properties of visibly pushdown automata and their languages which we will appeal to later in the paper. To begin with, VPLs define a robust class of languages closed under boolean operations.

PROPOSITION 1. *[4] If L_1 and L_2 are VPLs over a common alphabet $(\Sigma, \overline{\Sigma})$, then $L_1 \cup L_2$, $L_1 \cap L_2$ and $\overline{L_1}$ are also VPLs with respect to $(\Sigma, \overline{\Sigma})$.*

In addition, VPLs have a logical characterization using the monadic second order theory over words augmented with a binary matching predicate ν, denoted MSO_ν.

PROPOSITION 2. *[4] A language L over $(\Sigma, \overline{\Sigma})$ is a VPL iff there is an MSO_ν sentence φ over $(\Sigma, \overline{\Sigma})$ that defines L.*

Unlike pushdown automata, VPAs can be determinized without any loss in expressive power. We state this formally:

PROPOSITION 3. *[4] For any n-state VPA M over $(\Sigma, \overline{\Sigma})$, there is a deterministic VPA M' over $(\Sigma, \overline{\Sigma})$ with $O(2^{n^2})$ states and with stack alphabet of size $O(2^{n^2} \cdot |\Sigma|)$ such that $L(M') = L(M)$.*

Finally, the inclusion problem (which is undecidable for pushdown automata) is decidable for VPAs.

PROPOSITION 4. *[4] Given VPAs A_1 and A_2, the inclusion problem $L(A_1) \subseteq L(A_2)$ is decidable in* EXPTIME. *Furthermore, if A_2 is* deterministic*, then the problem is decidable in* PTIME.

Modular visibly pushdown automata

We now introduce *modular* VPAs. Given an SDTD (or in fact, any context-free grammar), one can associate a machine (module) with every non-terminal, which essentially checks whether the word it reads belongs to a derivation of the non-terminal. While doing so, such a module may need to expand other non-terminals, and can do so by "calling" the modules corresponding to these non-terminals. This intuition will lead us to capturing SDTDs precisely using a modular notion of VPAs. We have studied modular VPAs in an earlier paper [3], where we were motivated by their natural use in program modeling: the various modules correspond to procedures of a program which can call each other.

We first define a notion of a modular VPA, and then define restricted versions of it called XVPA (XML VPA) and SEVPA (single-entry VPA). The former will turn out to be the exact machine analog of SDTDs, while the latter has been studied earlier by us, and will help in proving various constructions in this paper. Modular VPAs have states partitioned into modules, demand that the symbol pushed onto the stack is always the current state, and ensure that if a module calls another, then upon return the control returns to a state of the calling module.

DEFINITION 4 (MODULAR VPA (μ-VPA) ; SEVPA). *Fix Σ, M and $\mu : M \rightarrow \Sigma$. A modular VPA (or μ-VPA) over (Σ, M, μ) is a tuple $A = (\{(Q_m, e_m, \delta_m)\}_{m \in M}, m_0, F)$, where for each $m \in M$,*

- *Q_m is a finite set of states of module m* [2]
- *e_m is a distinguished* entry state *of module m* [3]
- *$\delta_m = \delta_m^{\text{call}} \cup \delta_m^{\text{ret}}$, where*

  $\delta_m^{\text{call}} \subseteq \{q_m \xrightarrow{c/q_m} e_n \mid n \in \mu^{-1}(c)\}$

  $\delta_m^{\text{ret}} \subseteq \{q_m \xrightarrow{\overline{c}/p_n} q_n \mid n \in \mu^{-1}(c)\}$ *and is deterministic, i.e. $q_n = q'_n$ whenever $q_m \xrightarrow{\overline{c}/p_n} q_n$ and $q_m \xrightarrow{\overline{c}/p_n} q'_n$*
- *$m_0 \in M$ is a distinguished* start module
- *$F \subseteq Q_{m_0}$ is the set of* final states*.*

A single-entry VPA *(*SEVPA*) over (Σ, M, μ) is a μ-*VPA *such that the mapping $\mu : M \to \Sigma$ is a* bijection*.*

Note that we have defined μ-VPAs such that the return transitions are always deterministic; this is for technical convenience. A μ-VPA A is said to be *deterministic* if δ_m^{call} is deterministic as well, i.e. if $(q_m, c, e_n, q_m), (q_m, c, e_{n'}, q_m) \in \delta_m$, then $e_n = e_{n'}$. An SEVPA is a μ-VPA that has exactly one module for each tag.

Semantics. The semantics of a μ-VPA is defined by its corresponding VPA. Let $A = (\{(Q_m, e_m, \delta_m)\}_{m\in M}, m_0, F)$ be a μ-VPA over (Σ, M, μ). Then $A' = (Q, q_0, \{q_f\}, \Gamma, \delta)$ is the VPA over $(\Sigma, \overline{\Sigma})$ where $Q = \{q_0, q_f\} \cup (\bigcup_{m\in M} Q_m)$, $\Gamma = Q$ and

$$\delta = (\bigcup_{m\in M} \delta_m) \cup \{q_0 \xrightarrow{\mu(m_0)/q_0} e_{m_0}\} \cup \{q \xrightarrow{\mu(\overline{m}_0)/q_0} q_f \mid q \in F\}$$

We define $L(A)$, the language accepted by μ-VPA A, as $L(A')$. Note that a μ-VPA always accepts well-matched words which are of the form $\mu(m_0)w\mu(\overline{m_0})$.

A μ-VPA module m intuitively accepts many well-matched word languages. For example, a single module can distinguish two languages L_1 and L_2, and use return edges ("exits") to convey the difference to the calling module. A specialization (non-terminal) of an SDTD, however, is more restricted: it can be utilized to capture only *one* language. We will define XVPAs as essentially automata that have this restriction, and the concept of *exit* below will be useful.

DEFINITION 5 (EXIT). *Let $A = (\{(Q_m, e_m, \delta_m)\}_{m\in M}, m_0, F)$ be a μ-*VPA *over (Σ, M, μ). A non-empty set $X_m \subseteq Q_m$ is a (p_n, q_n)-exit for module m in A if $\forall q_m.\ q_m \xrightarrow{\overline{c}/p_n} q_n$ iff $q_m \in X_m$. In other words, a non-empty set X_m is a (p_n, q_n)-exit if it is the exact set of states of module m from which popping p_n from the stack leads to state q_n. A non-empty set X_m is an exit of module m in A if there exist states p_n, q_n such that X_m is a (p_n, q_n)-exit.*

REMARK 1. *Note that for every p_n, q_n, m, there is at most one (p_n, q_n)-exit for module m in a μ-*VPA *A. If such an exit exists, we denote it by $X_m(p_n, q_n)$.*

[2]Unless otherwise specified, m, n, etc. will range over modules, and p_m, q_m, q'_m, etc. will range over states in Q_m.

[3]Modular VPAs can be defined in a more general way where every module can have multiple entries $e_m^1, \dots e_m^k$. In this paper, we will largely focus on modular VPAs that have only a single entry per module.

Intuitively, the language of well-matched words that will take the automaton from p_n to q_n is the language defined by the module m with final states $X_m(p_n, q_n)$.

DEFINITION 6 (XVPA). *An* XVPA *over (Σ, M, μ) is a tuple $A = (\{(Q_m, e_m, X_m, \delta_m)\}_{m\in M}, m_0, F)$ such that $(\{(Q_m, e_m, \delta_m)\}_{m\in M}, m_0, F)$ is a μ-*VPA *over (Σ, M, μ), and the following* **single-exit property** *holds:*

for every $m \in M$, X_m is the unique *exit of module m*

An XVPA is intuitively a μ-VPA where each module defines only one language, no matter which state it is called from. Recall that we require μ-VPAs to be deterministic on returns; hence single-exit ensures that if we call a module m from state p_n, then exiting from m we end in a unique state q_n no matter which state in m we exit from.

EXAMPLE 2. *An* XVPA *that defines the same language as the SDTD in Example 1 is given in Figure 1, and for example, $\{x_{multlang}\}$ is the unique exit for module "multlang".*

Equivalence of SDTD and XVPA

We now show that XVPAs are an appropriate automaton model for SDTDs, by demonstrating their equivalence. More specifically, we will show that for every SDTD there exists an equivalent XVPA where the modules of the XVPA exactly correspond to the specializations in the SDTD, and vice versa. Intuitively, given an SDTD, we can construct an XVPA in which each module corresponds to a specialization in the SDTD, and the module checks whether a word belongs to the language of the specialization.

THEOREM 1. *For every SDTD (d, m_0) over (Σ, M, μ), there is an* XVPA *A over (Σ, M, μ) such that $L(A) = L_d(m_0)$. Furthermore, for every* XVPA *A over (Σ, M, μ), there is an SDTD (d, m_0) over (Σ, M, μ) such that $L(d, m_0) = L(A)$.*

PROOF. Let (d, m_0) be an SDTD over (Σ, M, μ). For every $m \in M$, let D_m be the deterministic (but perhaps *incomplete*) finite automaton (DFA) obtained from the minimized automaton for the regular expression $d(m)$ after discarding all "dead" states, i.e. states from which no final states can be reached. Specifically, let $D_m = (Q_m, e_m, Q_m^F, \delta_m)$, where

- Q_m is a finite set of states, $e_m \in Q_m$ is the initial state
- $Q_m^F \subseteq Q_m$ is the set of final states
- δ_m is a (partial) function from $Q_m \times M$ to Q_m
- for every $q \in Q_m$, $\exists x \in M^*$ such that $\delta_m(q, x) \in Q_m^F$ (no dead states)

Let $A = (\{(Q_m, e_m, X_m, \delta'_m)\}_{m\in M}, m_0, F)$ be the XVPA, where

- $X_m = Q_m^F$ for every $m \in M$, $F = Q_{m_0}^F$
- for every $m \in M$, $\delta'_m = {\delta'}_m^{\text{call}} \cup {\delta'}_m^{\text{ret}}$, where:

$${\delta'}_m^{\text{call}} = \{(q_m, \mu(n), e_n, q_m) \mid \delta_m(q_m, n) = p_m\}$$
$${\delta'}_m^{\text{ret}} = \{(q, \mu(\overline{m}), q_n, p_n) \mid \delta_n(q_n, m) = p_n \text{ and } q \in Q_m^F\}$$

Note that ${\delta'}_m^{\text{ret}}$ is deterministic, since δ_n is a (partial) function. Also, by definition of ${\delta'}_m^{\text{ret}}$, X_m is the unique exit of module m. Hence, A is indeed an XVPA.

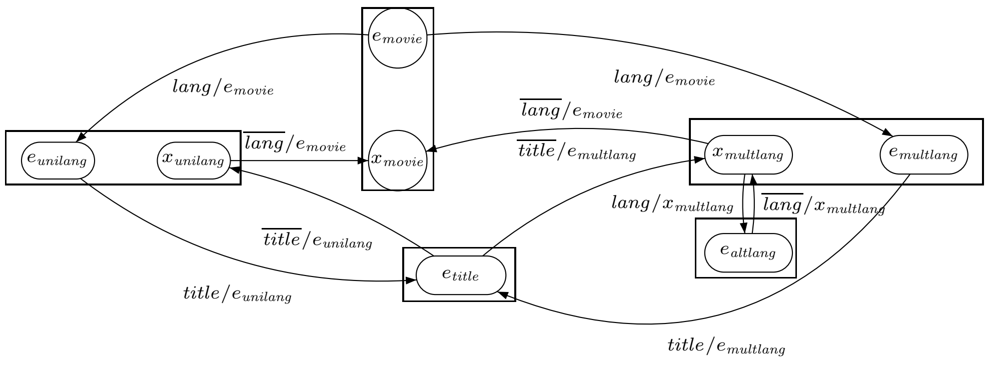

Figure 1: XVPA for the given SDTD

Further, for every well-matched word $w \in (M \cup \overline{M})^*$, it is easy to show by induction on the length of w, that $mw\overline{m} \in L_d^{\text{spl}}(m)$ if and only if $(e_m, \bot) \xrightarrow{\mu(w)}_A (q, \bot)$ for some $q \in X_m$. Since $X_{m_0} = F$, it follows that $L(A) = L_d(m_0)$.

Conversely, let $A = (\{(Q_m, e_m, X_m, \delta_m)\}_{m \in M}, m_0, F)$ be an XVPA. Define an SDTD (d, m_0) where, for every $m \in M$, $d(m)$ is the regular expression corresponding to the DFA $D_m = (Q_m, e_m, Q_m^F, \delta'_m)$, where $Q_m^F = X_m$ and δ'_m is defined as follows: for every $q_m \in Q_m$ and $n \in M$, if $(q, \mu(\overline{n}), q_m, p_m) \in \delta_m$ for some $q \in X_n$, then $\delta'_m(q_m, n) = p_m$. Note that this is well-defined because A has the single-exit property.

Once again, for every well-matched word $w \in (M \cup \overline{M})^*$, it is easy to show by induction on the length of w, that $mw\overline{m} \in L_d^{\text{spl}}(m)$ if and only if $(e_m, \bot) \xrightarrow{\mu(w)}_A (q, \bot)$ for some $q \in X_m$. Hence, $L(A) = L_d(m_0)$. ☐

Remark 2. *We will use the translation from SDTDs to XVPAs in the sequel. For this purpose we will assume that the XVPA is* trimmed*: for every state q in any module m, we will assume that there is a well-matched word w that leads from q to some state in X_m. If this wasn't the case, we can remove q and the transitions incident on q without changing the language of the XVPA.*

Now observe that in such an XVPA, for every run on $uc \in MR(\widehat{\Sigma})$ there exists a v such that the run can be extended on v such that it is accepting (showing $ucv \in L(A)$). This property will simplify many of our constructions.

In Section 4, we will prove that deterministic XVPAs correspond exactly to pre-order typed SDTDs. Hence, by Theorem 1, we have:

Remark 3. *If the regular expressions in* SDTD_1 *and* SDTD_2 *are given as deterministic finite automata (DFA), and* SDTD_2 *is pre-order typed, then the sub-typing problem* $\text{SDTD}_1 \subseteq \text{SDTD}_2$ *is decidable in* PTIME.

Note that a similar result was obtained in [17], for the case when *both* SDTD_1 and SDTD_2 are pre-order typed.

The sub-typing problem was also studied in [24], in the context of streaming algorithms for validating XML documents using a finite amount of memory, under the assumption that the XML document is well-matched. If A_1 and A_2 are DFAs over $\widehat{\Sigma}$, then an EXPTIME upper bound for deciding whether $L(A_1) \cap WM(\widehat{\Sigma}) \subseteq L(A_2) \cap WM(\widehat{\Sigma})$ was obtained by a reduction to the inclusion problem for tree automata. It was left as an open problem whether this bound could be improved. By Proposition 4, however, we immediately have a PTIME upper bound for this problem, even when A_1 and A_2 are deterministic VPAs:

Theorem 2. *If A_1 and A_2 are DFAs (or even deterministic* VPAs*) over $(\Sigma, \overline{\Sigma})$, then checking if $L(A_1) \cap WM(\widehat{\Sigma}) \subseteq L(A_2) \cap WM(\widehat{\Sigma})$ is decidable in* PTIME.

3. TYPE-CHECKING

In this section, we present several results (some of which are new) pertaining to SDTDs, that follow immediately from existing results for VPAs. We begin with results in the *streaming* context, where one is required to determine whether or not an XML document belongs to the language defined by an SDTD.

Streaming

Deterministic automata working on strings are the most natural model for processing streams of data. Since VPLs always have deterministic acceptors [4], VPAs are a convenient abstraction for studying streaming problems. In this section we look at some known results for VPAs and examine their consequences for type checking XML documents.

The following Proposition shows that deterministic SEVPAs capture the class of SDTDs.

Proposition 5. *[3] For every SDTD (d, m_0) over (Σ, M, μ), there is a* deterministic SEVPA *A over $(\Sigma, \Sigma, \text{id})$* [4] *such that $L(A) = L(d, m_0)$.*

The above proposition is true since an SDTD is always captured by an XVPA, and results in [3] show that any language accepted by a VPA can be accepted by a deterministic SEVPA.

Hence, deterministic SEVPAs give a *streaming* algorithm to recognize SDTDs. The streaming algorithm simply simulates the deterministic SEVPA on the input and checks if it

[4]id denotes the identity function

is accepting. Note that the algorithm will use only constant space in addition to a stack of symbols over a fixed alphabet, where the stack height is bounded by the *depth* of the nesting of tags in the document (which is typically not very large when compared to the size of the document).

Such an algorithm was already observed in [24] using pushdown automata; in fact the pushdown automaton they construct is essentially an SEVPA! By combining the set of single-entry modules into a common module, any such SEVPA can be reinterpreted as a *complete multi-entry* μ-VPA. We can now appeal to the following Proposition to prove that there is a *minimal* streaming automaton recognizing a given SDTD.

PROPOSITION 6. *[14] For every complete multi-entry μ-VPA A, there is a unique (up to isomorphism)* minimum-state *deterministic multi-entry μ-VPA A' such that $L(A') = L(A)$.*

Hence, the minimization construction outlined in [14] gives a way to minimize the space used by a streaming type-checking algorithm that is based on pushdown automata. Thus, unlike the result in [24], we are able to construct a provably minimal streaming recognizer for an SDTD.

4. PRE-ORDER TYPING

In this section, we study the class of XML schema that are pre-order typed, give an automata theoretic characterization of them, develop a streaming algorithm that determines the types of open-tags whenever possible, and also show that checking whether a tag can be pre-order typed is solvable in polynomial time.

4.1 Automata Theoretic Characterization

Pre-order typing has a very simple characterization in terms of the structure of XVPA defining the XML schema. An SDTD is pre-order typed exactly when its XVPA translation is deterministic:

LEMMA 1. *An SDTD (d, m_0) is pre-order typed if and only if there is a* deterministic *XVPA A over the same modules such that $L(A) = L_d(m_0)$.*

PROOF. Let (d, m_0) over (Σ, M, μ) be a pre-order typed SDTD and let $A = (\{(Q_m, e_m, X_m, \delta_m)\}_{m\in M}, m_0, F)$ be the corresponding XVPA as defined in Theorem 1. Recall that by definition of XVPA δ_m^{ret} is deterministic for each m. Consider some state q of A and let u be a string such that A has some computation on u that ends up in state q. Now since (d, m_0) is pre-order typed, there is a unique specialization m associated with c in the string uc. Thus, the state q must have *only* the transition to e_m on the symbol c. Hence, δ_m^{call} is deterministic for every module m.

Conversely, suppose $A = (\{(Q_m, e_m, X_m, \delta_m)\}_{m\in M}, m_0, F)$ is a deterministic XVPA and let (d, m_0) be the SDTD corresponding to it (as defined in Theorem 1). Consider a string u, and let q be the state reached by A on the string u. Since A is deterministic, for any $c \in \Sigma$ there is at most one m such that q has a transition to e_m on symbol c. Thus, m is the unique specialization associated with c in the string uc. Hence, (d, m_0) is pre-order typed. $\square$

4.2 Dynamic pre-order typing

We now study the problem of dynamic pre-order typing. Recall that this problem asks for an automaton that streams input and determines the type of open-tags as soon as it meets them, provided the type has been determined by the prefix read till that point. Consider the XVPA $A' = (\{(Q_m, e_m, X_m, \delta_m)\}_{m\in M}, m_0, F)$ corresponding to (d, m_0) as defined in Theorem 1. This XVPA is deterministic on return transitions but, in general, is non-deterministic on call symbols. We construct a deterministic automaton A that simulates the behaviour of A' on all applicable specializations of call symbols. After reading prefix u, if there is only *one* applicable specialization m of call symbol c, then A outputs m as the uniquely determined prefix-type of c at position $|uc|$. For every $c \in \Sigma$, let $Q_c = \{q \in Q_m \mid \mu(m) = c\}$. Let $A = (Q, q_0, Q, \delta)$ be a VPA without final states, where $Q = \{P \subseteq Q_c \mid P \neq \emptyset, c \in \Sigma\}$, $q_0 = \{e_{m_0}\}$ and δ is defined as follows:

$$P \xrightarrow{c/P} P', \text{ where}$$
$$P' = \{e_m \mid \mu(m) = c \text{ and } \exists p \in P.\ p \xrightarrow{c/p}_{A'} e_m\} \neq \emptyset$$
$$P \xrightarrow{\overline{c}/P''} P', \text{ where}$$
$$P' = \{p' \mid \exists p \in P, p'' \in P''.\ p \xrightarrow{\overline{c}/p''}_{A'} p'\} \neq \emptyset$$

Further, A outputs m on transition $P \xrightarrow{c/P} P'$ if and only if $P' = \{e_m\}$ for some $m \in \mu^{-1}(c)$. We claim that for every open tag c that has prefix-type m at position $|uc|$ in $|ucv|$, the VPA A outputs m after reading the input uc. First observe that by construction of A and by Remark 2, if $q_0 \xrightarrow{uc}_A P$ and $|P| > 1$, then there are two distinct $m, m' \in \mu^{-1}(c)$ and strings $v, v' \in \widehat{\Sigma}^*$ such that $ucv, ucv' \in L_d(m_0)$ and $(ucv, |uc|)$ has type m whereas $(ucv', |uc|)$ has type m'. Hence, c is not prefix-typed at $|uc|$ and the automaton A does not output anything on reading c after input u. However, if $P = \{e_m\}$, then by construction of A, for every $v' \in \widehat{\Sigma}^*$ such that $ucv' \in L_d(m_0)$, $(ucv', |uc|)$ has type m. Therefore, c has prefix-type m at position $|uc|$ in ucv, and A in fact outputs m on reading uc.

THEOREM 3. *For any SDTD, we can effectively construct an algorithm that dynamically pre-order types the tags in a streaming XML document. Further, this algorithm uses only space $O(s.d)$, where s is the size of the SDTD and d is the depth of the document.*

4.3 Checking partial pre-order typing of schema

We now consider the following problem: Given an SDTD (d, m_0), which open tags are pre-order typed in *every* document defined by (d, m_0)?

In the procedure for converting an SDTD into an XVPA defined in Theorem 1, we chose a deterministic finite state automaton D_m corresponding to every regular expression $d(m)$. This results in an automaton that may be exponential in the size of the SDTD (d, m_0). Instead, for every m, let D_m be a non-deterministic finite state automaton (one that is linear in the size of $d(m)$ can be constructed efficiently). We therefore obtain a *non-deterministic* VPA $A = (Q, q_0, F, Q, \delta)$ such that $L(A) = L_d(m_0)$, and further, $|Q| = O(n)$, where n is the size of (d, m_0). To efficiently determine whether an open tag c is pre-order typed or not, we search for a *witness* to the fact that c is not pre-order typed at some position. Such a witness consists of words

$ucv, ucv' \in L_d(m_0)$ such that $(ucv, |uc|)$ has type m whereas $(ucv', |uc|)$ has type m' for some $m \neq m'$. Searching for such witnesses reduces to performing reachability in a VPA A' without final states defined as follows: $A' = (Q', q_0', Q \times Q, \delta')$ where $Q' = Q \times Q \times (\Sigma \cup \{*\})$, $q_0' = (q_0, q_0, *)$ and δ' is defined as follows:

$$\begin{aligned}
\mathbf{T1}: \quad & (q, q', *) \xrightarrow{c/(q,q')}_{A'} (p, p', *) \quad \text{if } q \xrightarrow{c/q}_A p \\
& \text{and } q' \xrightarrow{c/q'}_A p' \\
\mathbf{T2}: \quad & (q, q', *) \xrightarrow{\overline{c}/(q_1,q_2)}_{A'} (p, p', *) \quad \text{if } q \xrightarrow{\overline{c}/q_1}_A p \\
& \text{and } q' \xrightarrow{\overline{c}/q_2}_A p' \\
\mathbf{T3}: \quad & (q, q', *) \xrightarrow{c/(q,q')}_{A'} (p, p', c) \quad \text{if } q \xrightarrow{c/q}_A p, \\
& q' \xrightarrow{c/q'}_A p' \text{ and } p \neq p'
\end{aligned}$$

LEMMA 2. *For every open tag $c \in \Sigma$, c is pre-order typed in (d, m_0) if and only if no state (p, p', c) is reachable from the initial state in A'.*

Intuitively, a state of the form (p, p', c) is reachable in A' whenever there are states q, q' reachable on some common input u in A, and there are distinct specializations m, m' of c such that module m (resp. m') can be called from state q (resp. q'). Then by Remark 2, there are strings v, v' such that ucv, ucv' witness the fact that c not pre-order typable at position $|uc|$. Since A' has size $O(n^2)$ and reachability in a VPA can be determined in cubic time, we have:

THEOREM 4. *Given an SDTD (of size n), the problem of checking whether c is pre-order typed in the SDTD is solvable in time polynomial in n.*

5. POST-ORDER TYPING

In this section, we give an automata theoretic characterization of the class of XML schema that can be post-order typed. Given a post-order typed SDTD, consider the XVPA corresponding to it. After reading a word u, if the XVPA meets an open-tag c, then c need not be prefix-typed at position $|uc|$. Hence, the XVPA may call several modules $m_1, \ldots, m_k$. However, since the SDTD is post-order typed, the type of c will get determined at the closing tag $\overline{c}$, i.e. at the time of exit from these modules. Hence it is clear that the *languages* accepted by these modules must be *disjoint*. More formally, we say that an XVPA A has *disjoint calls* if for all states q in A and open tags $c \in \Sigma$, if $q \xrightarrow{c}_A e_m$ and $q \xrightarrow{c}_A e_{m'}$ for distinct $m, m' \in \mu^{-1}(c)$, then $L_d(m) \cap L_d(m') = \emptyset$. We therefore have:

LEMMA 3. *An SDTD (d, m_0) is post-order typed if and only if there is an XVPA A with disjoint calls over the same modules such that $L(A) = L_d(m_0)$.*

5.1 Dynamic post-order typing

We now turn to the problem of *dynamically* post-order typing a streaming XML document. Our construction of the deterministic VPA $A = (Q, q_0, Q, \delta)$ is as in Section 4.2 with the only modification that A outputs m on a return transition $P \xrightarrow{\overline{c}/P''} P'$ if and only if $P \subseteq Q_m$ for some $m \in \mu^{-1}(c)$. Using a similar argument as before, we can show that on any input, the above VPA outputs the type of every closing tag $\overline{c}$ at position $|u\overline{c}|$ whenever this can be uniquely determined by reading the prefix u.

Hence an algorithm analogous to that reported in Theorem dynpre follows.

5.2 Checking partial post-order typing of schema

We now consider the problem: Given an SDTD (d, m_0), which open tags are post-order typed in *every* document defined by (d, m_0)?

Once again we can express this as a reachability problem on an appropriately defined VPA A'. Our construction is identical to the construction defined in Section 4.3 except that the condition **T3** in the defintion of the transition function δ' is replaced with:

$$\begin{aligned}
\mathbf{T3'}: \quad (q, q', *) \xrightarrow{\overline{c}/(q_1,q_2)}_{A'} (p, p', c) \text{ if} \\
q \xrightarrow{\overline{c}/q_1}_A p, q' \xrightarrow{\overline{c}/q_2}_A p' \\
\text{and } q \in Q_m, q' \in Q_{m'} \\
\text{with } m \neq m'
\end{aligned}$$

Similar to Lemma 2, we can show that a tag c is post-order typed if and only if *no* state (p, p', c) is reachable. Hence:

THEOREM 5. *Given an SDTD (of size n), the problem of checking whether c is post-order typed in the SDTD is solvable in time polynomial in n.*

6. PREFIX QUERYING

The problem of pre-order (respectively post-order) typing can be viewed abstractly as answering which open-tags (respectively close-tags) satisfy the property of being *uniquely* parsed as a particular specialized tag in an SDTD, provided the property of whether a tag has this property is determinable by reading the *prefix* of the document up till the open-tag (respectively close-tag).

In this section, we generalize this idea, by defining a generic class of queries (formalized using monadic second-order logic) that have the property that a position in a document satisfying the query is solely determined by the prefix of the document till that point. This class of queries is independent of the document-type definitions used to describe input documents, and pre-order and post-order typing are simply special instances of this class of queries.

The monadic second-order logic structures that we consider will be over the *nested structure* [5] defined by a document. More precisely, given any document $w \in \widehat{\Sigma}$, let us view the word as a linear $\widehat{\Sigma}$ labeled structure $([1, |w|], \leq, \{Q_a\}_{a \in \Sigma}, \nu)$, where there are $|w|$ elements corresponding to each letter in w, $\leq$ is the linear order on this set of positions, each Q_a is a unary predicate that is true on exactly the positions labeled a (i.e. $Q_a = \{i \mid w[i] = a\}$, and ν is the *matching* binary relation that associates each open-tag with the corresponding close-tag (i.e. $\nu(x, y)$ holds iff $w[x] \in \Sigma$, $w[y] \in \overline{\Sigma}$, and $w[x] \ldots w[y]$ is well-matched).

Monadic second-order logic can be now defined as the canonical logic over this structure with interpreted relations $\leq$, Q_a's and ν. Since we are interested in properties that are determined by the prefix in a document, we define a restricted version of MSO, called Pre-MSO, which allows only quantification over positions that occur before the query position.

Formally, fix a first-order variable x. Then the set of all Pre-MSO(x) formulas $\varphi(x)$ (with x being the only free variable) is defined as:

$$\begin{aligned}
\varphi \quad ::= \quad & y \in X \mid Q_i(y) \mid y \leq y' \mid \nu(y, y') \mid \varphi \wedge \varphi \mid \neg\varphi \\
& \mid \exists z (z \leq x \wedge \varphi) \mid \exists X(\varphi) \\
& \textit{where } y, y', z \in FV, z \neq x, \ X \in SV.
\end{aligned}$$

The logic Pre-MSO is a restriction that forces any query written in the logic to depend only on the prefix of the word till the particular point of query. Formally, for any word $w \in \widehat{\Sigma}^*$, let $Answers(w, \varphi)$ be the set of of all positions i, $1 \leq i \leq w$, such that $\varphi(x)$ holds in w when x is interpreted to be i.

Proposition 7. *Let $\varphi(x)$ be any Pre-MSO(x) formula and let $i \in Answers(w, \varphi)$. Then, for any word w', $|w'| \geq i$, with $w[1, i] = w'[1, i]$, $i \in Answers(w', \varphi)$ as well.*

Consequently the set of answers to a query $\varphi(x)$ in a word w, which is the set of all positions i such that $\varphi(x)$ is satisfied when x is interpreted to be i, is determined when the prefix up till i has been read. Hence, technically, we should be able to output the answer positions i as soon as we read the i'th letter in a document. We show that indeed this is true, and there is a deterministic visibly pushdown automaton that can perform this task. From this we obtain a streaming algorithm that uses space only linear in the depth of a document, and can output the set of *all* answers to any Pre-MSO(x) query.

Let us first define VPA with output.

Definition 7 (VPA with output). *A marking visibly pushdown automaton (*VPA*) over $(\Sigma, \overline{\Sigma})$ is a tuple $A = (Q, q_0, Q^F, \Gamma, \delta, M)$, where $(Q, q_0, Q^F, \Gamma, \delta)$ is a VPA and $M \subseteq Q$ is a subset of marked states.*

We say such a marking VPA A working on a word $w \in \widehat{\Sigma}$ *marks position* i ($1 \leq i \leq |w|$) if there is some run of A on w which reaches a marked state in M just after reading the i'th letter in w. (Note that the set of final states play no role in this definition.)

We can now show:

Theorem 6. *Let φ be a Pre-MSO(x) formula over $\widehat{\Sigma}$. Then there is a* deterministic *marking VPA A_φ that on any word $w \in \widehat{\Sigma}^*$ marks exactly the set of positions $Answers(w, \varphi)$.*

Proof. Let $\varphi(x)$ be a Pre-MSO(x) formula. Now, consider the set *PrefixLang*(φ), which is the set of all words u, not necessarily well-matched, such that $|u| \in Answers(u, \varphi)$, i.e. the set of all words u such that $\varphi(x)$ holds in u when x is interpreted to be the last letter of u. It can be seen that *PrefixLang*(φ) is expressible in (full) MSO.

Using the correspondence between monadic second-order logic on nested structures and visibly pushdown automata [4, 5], it follows that the set $Prefix - Lang(\varphi)$ can be (effectively) recognized by a VPA. Let $B_\varphi = (Q, q_0, Q^F, \Gamma, \delta)$ be an automaton accepting this language. Then the automaton which will answer the queries will be $A_\varphi = (Q, q_0, Q^F, \Gamma, \delta, M)$, where $M = Q^F$.

Intuitively, when reading a word w and when at position i, the automaton B will mark i iff the prefix u uptil position i already satisfies φ, when i is interpreted for x. Since we know that w satisfies φ when x is interpreted to be i iff u satisfies φ when x is interpreted to be i, the correctness follows. □

We can also show the converse, namely that every marking VPA (deterministic or nondeterministic) defines a query that is expressible by a Pre-MSO formula $\varphi(x)$. The proof of this relies on the translation from VPAs to MSO, alongwith observations similar to the ones made in the above proof; we skip further details:

Theorem 7. *For every query captured by a marking VPA, there is an equivalent Pre-MSO formula $\varphi(x)$ that defines the same query.*

As a corollary to Theorem 6 we have:

Corollary 1. *For any Pre-MSO formula $\varphi(x)$, there is a streaming algorithm (which is effectively constructible) that processes documents and outputs the set of all answers to the query defined by $\varphi(x)$. Moreover, this algorithm processes each letter of the document in constant time, and utilizes space at most $O(d)$, where d is the depth of the document.*

The algorithm simply simulates the deterministic VPA constructed in Theorem 6; processing a letter can be done in constant time (for a fixed formula φ) and the space required is to store the state and the stack, which is $O(d)$.

Sequential XPath is a restricted version of XPath that has the property that a position satisfying a query is determined by the prefix till the node [8] (it however has further restrictions aimed at limiting buffering input). Sequential XPath has been defined and studied precisely to facilitate one-pass querying with minimal buffering of input. Our result can be seen as a generalization of this idea to the much larger class of MSO-definable queries. A similar precise characterization of queries that can be answered by 1-pass streaming algorithms is provided in [19]. There the characterization is in terms of constraint systems and pushdown forest automata. Our results here can be seen as a reformulation of those results in terms of logic and word automata.

We conclude this section by observing that our results on dynamic pre-order and post-order typing can be seen as special cases of Theorem 6.

Proposition 8. *For any SDTD (d, m_0) and specialization m, there is a Pre-MSO formula* $\mathrm{Preorder}^m_{(d,m_0)}(x)$ *such that for any document w, $Answers(w, \mathrm{Preorder}^m_{(d,m_0)})$ is exactly the set of positions that can be pre-order typed with specialization m. Analogously, there is a* $\mathrm{Postorder}^m_{(d,m_0)}(x)$ *that exactly describes the positions that can be post-order typed with specialization m.*

Proof. The construction of the VPA in Section 4.2, when restricted to only answering the positions getting type m, can be seen as a VPA with output. Thus, the proposition follows from Theorem 7. The proof for post-order typing is similar. □

Proposition 8 can be used obtain a streaming algorithm for dynamic pre-order (and post-order) typing. The streaming algorithm simulates the deterministic marking VPA from Theorem 6 for $\mathrm{Preorder}^m_{(d,m_0)}(x)$ (or $\mathrm{Postorder}^m_{(d,m_0)}(x)$) for each specialization m simultaneously on the document w, and outputs m whenever the marking VPA for m enters a marking state.

While the results of Section 4.2 can thus be seen as a corollary to the results presented in this section, the algorithm described in the previous paragraph is likely to be more inefficient when compared to that presented in Section 4.2. Thus, in practice it will be more useful to use the direct construction presented earlier for determining types.

7. CONCLUSIONS

We have shown that visibly pushdown automata is a convenient and powerful model for studying problems for XML that intrinsically involve processing documents from left to right. Modular VPAs emerge as an elegant model to study problems such as streaming and typing XML documents, and constructions and algorithms based on modular VPAs are intuitive and simple, giving clean proofs and efficient algorithms.

Modular VPAs are useful in the program verification context as well, and hence our results make an unusual connection between the two fields, which could be mutually beneficial. For instance, the congruence based characterizations developed in the verification setting have been useful in the XML setting, and the unique minimal modular VPA result presented herein has potential uses in building minimal program models.

There are several other problems that can be addressed using the XVPA model. In particular, checking whether SDTDs can be transformed into another which is pre-order (or post-order) typed is decidable and these results can be proved using XVPAs (we refer the reader to the technical report [15]). Pre-order typability of SDTDs seems related to LL[1] grammars [12] and is worth studying. Intuitively, an LL[1] grammar requires that the rule to be applied is determined when each symbol is read, which greatly resembles pre-order typing and deterministic XVPAs.

The most interesting future direction we see is in defining *querying* automata using the VPA model. Note that while we have studied *prefix* querying in this paper, general querying (like general XPath queries) can select a node depending on properties of the document that lie in the future of the node. In work not reported here, we have extended the VPA model to a query model where the automaton is powerful enough to store fragments of the document, and hence answer all XPath/MSO queries. In both this and the work reported in this paper, the ability to minimize VPA seems to reflect the kind of optimizations in memory proposed by other researchers [7, 11], and we believe that minimization of visibly pushdown automata will be a formal and perhaps more effective way to achieve space optimizations. Finally, we plan to also study transformations of XML documents in the streaming setting using VPAs (see also [23], where visibly pushdown expressions have been used to study effect systems on streaming XML).

8. REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] A. V. Aho and J. D. Ullman. Translations on a context free grammar. *Information and Control*, 19(19):439–475, 1971.

[3] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *ICALP*, pages 1102–1114, 2005.

[4] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211. ACM Press, 2004.

[5] R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT*, LNCS 4036, pages 1–13, 2006.

[6] Mikolaj Bojanczyk and Thomas Colcombet. Tree-walking automata do not recognize all regular languages. In *STOC*, pages 234–243, New York, NY, USA, 2005. ACM Press.

[7] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient XPath query processor for XML streams. In *ICDE*, page 79, 2006.

[8] A. Desai. Introduction to sequential xpath. In *Proc. of IDEAlliance XML Conference*, 2001.

[9] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *ICDT '03*, pages 173–189, 2003. Springer-Verlag.

[10] Martin Grohe and Nicole Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *PODS '05*, pages 238–249, 2005. ACM Press.

[11] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, pages 419–430. ACM Press, 2003.

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

[13] Christoph Koch and Stefanie Scherzinger. Attribute grammars for scalable query processing on XML streams. In *DBPL*, volume 2921 of *LNCS*, pages 233–256, 2003.

[14] V. Kumar, P. Madhusudan, and M. Viswanathan. Minimization, learning, and conformance testing of boolean programs. In *CONCUR*, 2006. to appear.

[15] V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown languages for xml. Technical Report UIUCDCS-R-2006-2704, UIUC, 2006.

[16] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pages 227–238, 2002.

[17] Wim Martens, Frank Neven, and Thomas Schwentick. Which XML schemas admit 1-pass preorder typing? In *ICDT*, pages 68–82, 2005.

[18] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[19] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *FSTTCS*, pages 134–145, 1998. Springer-Verlag.

[20] Frank Neven. Automata, logic, and XML. In *CSL*, pages 2–26, London, UK, 2002. Springer-Verlag.

[21] Frank Neven and Thomas Schwentick. On the power of tree-walking automata. *Information Computing*, 183(1):86–103, 2003.

[22] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[23] Corin Pitcher. Visibly pushdown expression effects for XML stream processing. In *Programming Language Technologies for XML*, pages 1–14, 2005.

[24] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, New York, NY, USA, 2002. ACM Press.

[25] Victor Vianu. XML: From practice to theory. In *SBBD*, pages 11–25, 2003.

[26] World Wide Web Consortium. Extended Markup Language (XML). `http://www.w3.org`.