Visibly Pushdown Expression Effects for XML Stream Processing

Corin Pitcher
cpitcher@cs.depaul.edu
School of CTI, DePaul University

Abstract

We define an effect system, based upon visibly pushdown languages (VPLs), for a programming language that processes streams of tokens with parenthesis-like matching, as found in XML documents or s-expressions. The effect analysis ensures that programs read and write words in which tokens match, despite the fact that tokens are read and written individually. In particular, the novel treatment of input provides a compositional description of the behaviour of programs with lookahead.

We introduce visibly pushdown expressions (VPEs), corresponding to the class of VPLs, as the effects. VPEs generalize regular expression types by incorporating intersection, unmatched tokens, and overlapped concatenation (used in the analysis of operations with lookahead). Hosoya, Vouillon, and Pierce's decision procedure for language inclusion between regular expression types, via a translation to non-deterministic tree automata, does not apply to VPEs. Instead we obtain a decision procedure via a translation of VPEs to Alur and Madhusudan's monadic second order logic with matching relation MSO_μ.

1 Introduction

Large data streams are often processed in a single-pass to avoid construction of an in-memory representation of the data. Hand-written parsers or transducers that operate directly upon input and output streams are prone to errors, but there are at least two strategies for preventing some of those errors. The first is to compile a high-level language into an automaton or transducer, and the second is to use static analysis. In this paper we provide the latter, in the form of an effect system based upon visibly pushdown languages [3].

Several groups of researchers have investigated space-efficient query evaluations on large data streams. For example, the Hancock language [6, 8] assists programmers implementing queries by automating lazy construction of data structures from data in the input stream, amongst other ways. XML [7] data streams have provoked separate lines of research, in particular on the problem of evaluating multiple queries simultaneously upon a stream. The XFilter [2] and YFilter [9] systems compile queries on trees to automata over words. In this case, the queries are fragments of the W3C's XPath 1.0 language [37], and the result is whether or not one of the queries succeeded. Subsequently, it has been shown that significant performance improvements can be achieved by lazily constructing either finite-state automata [14] or a variation of pushdown automata for XPath queries with predicates [15]. An efficient, caching implementation for queries written in a larger fragment of XPath, that includes the parent and ancestor axes, is given in [4]. The parent axis allows a query to reference the parent of a node that it is examining which, for a streaming implementation, requires either multiple passes or caching of previously seen data.

Despite the success of domain-specific languages for efficiently querying streams carrying XML documents, developers continue to use APIs that provide direct access to streams of tokens, such as the Simple API for XML (SAX) [31], the XML Pull API [36], or the Streaming API for XML (StAX) [33]. Typically actions to read from a stream will be interleaved with code to populate data structures, write to a database, or write to another stream. Although low-level, if we have a substantial body of code that operates on existing data structures, it is often easier to populate existing data structures by manipulating streams directly than it is to rewrite the existing code as a query or to fit a serialization format.

This raises the question of whether it is possible to analyze a program manipulating streams, to determine the input that it will accept, and the output that it can

produce. One starting point is the notion of session type introduced by Honda et al [18, 35, 19] to describe the interaction that takes place on session channels in a concurrent programming language. In particular, the type of messages (or tokens) communicated on a channel may change as each message is sent, according to the session type. Gay and Hole [11, 12] have shown how to add subtyping for session types in the context of the π-calculus.

Gay et al [13] presented a λ-calculus with a type and effect system incorporating session types. If channels are restricted to unidirectional communication, their session types allow one to read or write regular languages (upon words) on channels. Their system does not include subtyping at present, and does not allow programs to perform non-destructive reads on input channels. The latter is useful when writing parsers.

In related work, Tabuchi et al [34] proposed a λ-calculus with a type and effect system based upon regular languages (upon words). Stream manipulation is limited to output, so effects describe the language written to an output stream. Subtyping and subeffecting are based upon language inclusion. Instead of an operation to read from an input stream, the authors consider a pattern matching operation over words, and those words are first-class citizens.

Neither λ-calculus incorporates lookahead with input streams. Compositional descriptions of the input behaviour of such programs are more complex than those for output behaviour. To see this, consider a program M that reads as many a tokens as possible from an input stream and a program N that reads exactly one a token from the same input stream. If we sequentially compose these programs as $M; N$ we obtain a program that always fails when it reaches N because no a tokens are left by M. In contrast, the program $N; M$ will succeed whenever the input stream contains at least one a token.

Moreover, neither λ-calculus is sufficient for stream-based processing of XML because their effects are regular languages upon words (see [32] for a characterization of the DTDs that do determine regular languages). However, the usual constraints placed on XML documents, such as RELAX-NG Schema, XML Schema, or DTDs, define, to a first approximation, regular tree languages after a suitable encoding of XML documents as binary trees [30]. LL(1) grammars [29] would also be sufficient to encode many constraints on XML documents, but subtyping and subeffecting based on language inclusion would not be possible, because testing inclusion between LL(k) grammars is undecidable [10].

Hosoya et al [23, 20, 25, 22, 24] proposed regular expression types, a clear, concise notation for regular tree languages, as the types of their programming language XDuce that manipulates XML documents as trees. Subtyping in XDuce is based on language inclusion between regular expression types. Hosoya et al show that this inclusion is decidable by translation to top-down non-deterministic tree automata. Although XDuce provides a very different programming model, it is certainly appealing to reuse regular expression types in the stream processing model that we present. This is not immediately possible because stream processing programs operate on individual tokens that cannot be represented as trees.

Alur and Madhusudan [3] have recently proposed the class of visibly pushdown languages (VPLs) as a tool for program analyses that involve nested, matching pairs of symbols, such as the sequence of calls and returns in a thread, or the start and end tags in an XML document. They show that the class of VPLs is closed under concatenation, union, intersection, complementation, and Kleene-*. This is achieved by distinguishing between symbols of a VPL used for calls (start tags) and those used for returns (end tags). Unlike balanced grammars [5], the class of VPLs includes words with unmatched symbols, facilitating the use of VPLs as effects for stream processing programs.

In this paper we define visibly pushdown expressions (VPEs), prove that they correspond to the class of VPLs, and use them as the basis of a type and effect system for a programming language λ_{str}. The regular expression type constructor $a[R]$ corresponds to the VPE constructor $\binom{a_1}{a_2}[T]$ that constrains the call symbol a_1 and the return symbol a_2 to match in a word $a_1.\alpha.a_2$ (implying that the call and return symbols in α match correctly).

The effects for λ_{str} programs are maps from streams to VPEs. For output streams, this is unsurprising because we need only represent a postcondition (the output). For example, the following program outputs a start tag, a string, and an end tag on stream s, and has type unit and effect $s!^{\texttt{<name>}}_{\texttt{</name>}}[\mathsf{string}]$. We write:

$$\vdash s!\texttt{<name>}; s!\texttt{"Fred"}; s!\texttt{</name>} : \mathsf{unit}; s!^{\texttt{<name>}}_{\texttt{</name>}}[\mathsf{string}]$$

To model existing streaming APIs for XML, we must have both destructive and non-destructive read operations on input streams. The latter forces us to encode a precondition (tokens present before execution) and a postcondition (tokens present after execution) for input streams. Fortunately, with a fixed lookahead of one token, we can use the convention that a VPE effect for an input stream always describes one more token than it reads destructively. For example, the program $s?a$ that destructively reads the token a has effect $s?(a.\sim)$,

meaning that the input stream must have two tokens before execution, the first token must be a, and nothing is known about the current token after execution ($\sim$ is the wildcard for tokens). In contrast, the program $\mathsf{if}\ (s \rhd a)\ \mathsf{then} * \mathsf{else}\ \mathsf{fail}_s$, non-destructively checks that the current token is a and fails if it finds another token. This program has effect $s?a$, meaning that the input stream must have one token a before execution, no input is consumed, and the current token is a after execution.

When composing programs, we must check that postconditions on input streams for the first program are consistent with preconditions on input streams for the second program. We show that it suffices to define an overlapped concatenation operator $\oplus$ on languages. The overlapped concatenation insists that the last symbol of the first string is the same as the first symbol of the second string. For example, $a \oplus (a.\sim)$ denotes the same language as $a.\sim$, but $a \oplus (b.\sim)$ denotes the empty language. This leads to a simple effect assignment for the sequential composition of terms M and N. If the VPEs S_1 and S_1' describe the input behaviour of M and N respectively, then the input behaviour of their sequential composition is the overlapped concatenation $S_1 \oplus S_1'$. This results in the empty language $\emptyset$ when none of the final symbols (postcondition) of S_1 overlap with the initial symbols (precondition) of S_1'. A program with an input effect $s?\emptyset$ is not guaranteed to read any language. To combine the output behaviour of M and N, we need only concatenate their output languages.

$$\frac{\Gamma \vdash M : \sigma; s?S_1, t!S_2 \qquad \Gamma, x:\sigma \vdash N : \tau; s?S_1', t!S_2'}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau; s?(S_1 \oplus S_1'), t!(S_2.S_2')}$$

The decidability of inclusion testing between VPEs, hence of subtyping and subeffecting, is obtained via a compositional translation to Alur and Madhusudan's monadic second order logic with matching relation MSO_μ [3]. This translation handles an intersection operator in the surface language (VPEs), which is not directly possible with the translation for regular expression types used in [23, 20]. A similar translation would be possible for regular expression types extended with intersection via the monadic second order logic of trees. The expressiveness of VPEs and regular expression types is unchanged by the addition of intersection, but intersection does permit more concise descriptions of some languages.

1.1 Outline of Paper

Sections 2 and 3 briefly describe regular expression types, visibly pushdown languages, and the monadic second order logic MSO_μ. Section 4 introduces visibly pushdown expressions, and section 5 proves that language inclusion between VPEs is decidable. Section 6 contains the syntax and operational semantics of λ_{str}, and section 7 describes the type and effect system, and the subject reduction theorem. Section 8 concludes.

2 Regular Expression Types

Regular expression types [23, 24] were introduced to describe trees constructed from XML documents. Assuming a countable set of labels, ranged over by l, for the XML names in start tags and end tags, define label classes by the following grammar:

L	::=	l	atom
	$\mid$	$\sim$	wildcard
	$\mid$	$L \mid L$	union
	$\mid$	$L \setminus L$	difference

Label classes denote the finite and cofinite subsets of the set of all labels, where l denotes the singleton set $\{l\}$, $\sim$ denotes the set of all labels, $L_1 \mid L_2$ is set-theoretic union, and $L_1 \setminus L_2$ is set-theoretic difference.

Assuming a set of variables, ranged over by A, the grammar for regular expression types is[1]:

R	::=	$()$	empty sequence
	$\mid$	$L[R]$	element
	$\mid$	$R.R$	concatenation
	$\mid$	$R \mid R$	union
	$\mid$	A	variable
	$\mid$	$R*$	repetition

The regular expression type $L[R]$ denotes (the binary tree for) a word such as <l>.α.</l>, where l is a label in the denotation of the label class L.

Type definitions bind variables to regular expression types. We let E range over such maps from variables to regular expression types where recursive uses of variables are guarded by element occurrences (this condition is formalized for VPEs in the sequel). For example, $E(A_1) = l[].A_1.l[]$ is not allowed, but $E(A_2) = l[].l[A_2].l[]$ does meet the condition. With a map that binds the appropriate variables, a regular expression type denotes a set of trees, or a set of words consisting of start tags and end tags.

To test inclusion between the languages denoted by two regular expressions, regular expression types are translated into top-down non-deterministic tree automata.

[1] Earlier presentations of XDuce encoded repetition using recursion in tail positions, see the comments in [21]. We also write $R_1.R_2$ instead of R_1, R_2 for concatenation.

As with the conversion of regular expressions (on words) directly to deterministic finite-state automata [1], concatenation and Kleene-* require some care. Hosoya [20] proves termination of a translation procedure that gradually exposes smaller terms at the head of a regular expression type, e.g., rewriting $(R_1.R_2).R_3$ to $R_1.(R_2.R_3)$, and $(R_1 \mid R_2).R_3$ to $(R_1.R_3) \mid (R_2.R_3)$.

Non-deterministic tree automata are closed under complementation and intersection, and emptiness testing is decidable, so there is a decision procedure for inclusion testing between tree automata, which can be pulled back to the original regular expression types.

3 Visibly Pushdown Automata

Visibly pushdown automata (VPA) [3] operate on pushdown alphabets of the form $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_{int})$, where Σ_c, Σ_r, Σ_{int} are disjoint, finite sets of call, return, and internal symbols respectively. For a pushdown alphabet $\tilde{\Sigma}$, we define $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$. Transitions for call symbols are required to push onto the stack, transitions for return symbols must pop from the stack, and transitions for internal symbols must not use the stack.

Definition 3.1 A visibly pushdown automaton (VPA) on finite words over $\tilde{\Sigma}$ is a tuple $(Q, Q_{in}, \Gamma, \delta_c, \delta_r, \delta_{int}, Q_F)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet that contains a special symbol $\perp$ that appears at the bottom of the stack, and $\delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})$, $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, $\delta_{int} \subseteq Q \times \Sigma_{int} \times Q$, and $Q_F \subseteq Q$ is a set of final states. □

Configurations of a VPA have the form (q, σ), where $q \in Q$ is a state and $\sigma \in (\Gamma \setminus \{\perp\})*.\{\perp\}$ is a stack with $\perp$ at the bottom. A pre-run of a VPA on a word $a_1 \ldots a_n$ is a sequence of configurations $(q_1, \sigma_1), \ldots, (q_{n+1}, \sigma_{n+1})$ such that for all $1 \leq i \leq n$:

- If $a_i \in \Sigma_c$, then $\exists \gamma$ such that $(q_i, a_i, q_{i+1}, \gamma) \in \delta_c$ and $\sigma_{i+1} = \gamma.\sigma_i$.
- If $a_i \in \Sigma_r$, then $\exists \gamma$ such that $(q_i, a_i, \gamma, q_{i+1}) \in \delta_r$ and either $\gamma = \sigma_{i+1} = \sigma_i = \perp$ or $\gamma \neq \perp$ and $\gamma.\sigma_{i+1} = \sigma_i$.
- If $a_i \in \Sigma_{int}$, then $(q_i, a_i, q_{i+1}) \in \delta_{int}$ and $\sigma_{i+1} = \sigma_i$.

A pre-run $(q_1, \sigma_1), \ldots, (q_{n+1}, \sigma_{n+1})$ on $a_1, \ldots, a_n$ is a run if $q_1 \in Q_{in}$ and $\sigma_1 = \perp$. A run is accepting if $q_{n+1} \in Q_F$, and a word $\alpha \in \Sigma*$ is accepted by a VPA if there is an accepting run. The language of a VPA consists of the set of words that have accepting runs. A language is a visibly pushdown language if it is the language of some VPA.

For words where call and return symbols are matched, the stack will be $\perp$ in the final configuration. For words where there are unmatched call symbols, the stack will be larger than $\perp$ in the final configuration. When there are unmatched return symbols, the stack will be $\perp$ in the final configuration, and there will have been δ_r transitions with $\gamma = \perp$.

Example 3.2 The language $\mathcal{L} = \{\texttt{<}l\texttt{>}^n.\texttt{</}l\texttt{>}^n \mid n \geq 1\}$ is visibly pushdown when $\Sigma_c = \{\texttt{<}l\texttt{>}\}$, $\Sigma_r = \{\texttt{</}l\texttt{>}\}$, and $\Sigma_{int} = \emptyset$. We might also describe $\mathcal{L}$ using the regular expression type $l[A]$ where $E(A) = () \mid l[A]$. A suitable VPA has states $Q = \{q_0, q_1, q_2, q_3\}$, final states $Q_F = \{q_3\}$, stack alphabet $\Gamma = \{l, \bar{l}, \perp\}$, and transition relations:

$$\begin{aligned}\delta_c &= \{(q_0, \texttt{<}l\texttt{>}, q_1, \bar{l}), (q_1, \texttt{<}l\texttt{>}, q_1, l)\} \\ \delta_r &= \{(q_1, \texttt{</}l\texttt{>}, l, q_2), (q_2, \texttt{</}l\texttt{>}, l, q_2), \\ &\qquad (q_1, \texttt{</}l\texttt{>}, \bar{l}, q_3), (q_2, \texttt{</}l\texttt{>}, \bar{l}, q_3)\} \\ \delta_{int} &= \emptyset\end{aligned}$$

The second stack symbol $\bar{l}$ is used to identify the </l> that matches the original <l>. In contrast, if $Q' = \{q_0', q_1'\}$, $Q_F' = \{q_1'\}$, $\Gamma' = \{l, \perp\}$, and $\delta_c' = \{(q_0', \texttt{<}l\texttt{>}, q_0', l)\}$, $\delta_r' = \{(q_0', \texttt{</}l\texttt{>}, l, q_1'), (q_1', \texttt{</}l\texttt{>}, l, q_1')\}$, then we obtain a VPA that accepts the language $\{\texttt{<}l\texttt{>}^m.\texttt{</}l\texttt{>}^n \mid m \geq n \geq 1\}$. □

Alur and Madhusudan prove that the class of VPLs are closed under concatenation, union, intersection, complementation, and Kleene-*. Intersection relies on the fact that the actions of VPA on their stacks are constrained by their input, unlike normal pushdown automata. Concatenation is an interesting operation because the concept of matching between call and return symbols is based only on the symbols that lie in between them. For example, the concatenation of $\{\texttt{<}l_1\texttt{>}\}$ and $\{\texttt{</}l_2\texttt{>}\}$ is the language $\{\texttt{<}l_1\texttt{>}.\texttt{</}l_2\texttt{>}\}$.

As usual, the closure properties and the decidability of emptiness testing lead to a decision procedure for inclusion between the languages of VPA.

3.1 Monadic Second Order Logic

Alur and Madhusudan define a monadic second order logic MSO_μ that is interpreted over finite words of a pushdown alphabet. MSO_μ extends the usual monadic second order logic with a matching relation μ. Formulae are defined by the following grammar:

$$\phi ::= Q_a(x) \mid x \in X \mid x \leq y \mid \mu(x, y) \mid \neg\phi \mid \phi \vee \phi \mid \exists x.\phi \mid \exists X.\phi$$

First order variables x are interpreted as positions (natural numbers) within the finite word, and second order variables X are interpreted as sets of positions. The formula $Q_a(x)$ is satisfied if the symbol a is at the position (that is, the interpretation of) x. The formula $\mu(x, y)$ is satisfied if the symbols at positions x and y are matching call and return symbols respectively.

The class of VPLs is precisely the class of languages satisfied by MSO_μ formulae, which leads to a decision procedure for MSO_μ satisfiability [3]. In the sequel, MSO_μ is used to provide a compact proof of the decidability of language inclusion between the visibly pushdown expressions introduced in the next section.

4 Visibly Pushdown Expressions

In this section we introduce visibly pushdown expressions (VPEs) as a generalization of regular expression types that can represent words with unmatched call or return symbols. In addition, we add intersection, and, as shown in the next section, we can still achieve decidability of inclusion testing.

Symbol patterns correspond to the label classes used for regular expression types. For now we allow the sets Σ_c, Σ_r, Σ_{int} of a pushdown alphabet $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_{int})$ to be countably infinite. The grammar for symbol patterns is:

p	::=	a	symbol, $a \in \Sigma$
	$\mid$	$p \mid p$	union
	$\mid$	$\neg p$	complement
	$\mid$	$\sim_c$	wildcard for Σ_c
	$\mid$	$\sim_r$	wildcard for Σ_r
	$\mid$	$\sim_{int}$	wildcard for Σ_{int}

The denotation $[\![p]\!]_{\tilde{\Sigma}} \subseteq \Sigma$ of a symbol pattern p in $\tilde{\Sigma}$ is defined by:

$$[\![a]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \{a\} \qquad [\![\sim_c]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \Sigma_c$$
$$[\![p_1|p_2]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} [\![p_1]\!]_{\tilde{\Sigma}} \cup [\![p_2]\!]_{\tilde{\Sigma}} \qquad [\![\sim_r]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \Sigma_r$$
$$[\![\neg p]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \Sigma \setminus [\![p]\!]_{\tilde{\Sigma}} \qquad [\![\sim_{int}]\!]_{\tilde{\Sigma}} \stackrel{\text{def}}{=} \Sigma_{int}$$

We use the abbreviations $p_1 \& p_2 \stackrel{\text{def}}{=} \neg(\neg p_1 | \neg p_2)$ and $\sim \stackrel{\text{def}}{=} \sim_c | \sim_r | \sim_{int}$. In addition, P_c is defined to be the set of symbol patterns of the form $\sim_c \& p$. The sets P_r and P_{int} are defined similarly. For example, if L is a set of XML names, such that $\texttt{foo}, \texttt{bar} \in L$ and $\Sigma_r = \{\texttt{</}l\texttt{>} | l \in L\}$, then the symbol pattern $\sim_r \& \neg(\texttt{</foo>}|\texttt{</bar>}) \in P_r$ denotes the set $\{\texttt{</}l\texttt{>} | l \in L \setminus \{\texttt{foo}, \texttt{bar}\}\} \subseteq \Sigma_r$.

The visibly pushdown expressions (VPEs) are defined in two syntactic categories. The first syntactic category, the matched VPEs, are very similar to regular expression types, except that elements have two patterns: one for the call (start tag) and one for the return (end tag). Assuming a collection of variables, ranged over by A, the matched VPEs are defined by the following grammar:

T	::=	$\emptyset$	empty set
	$\mid$	$()$	empty sequence
	$\mid$	p	symbol pattern, $p \in P_{int}$
	$\mid$	$\begin{smallmatrix} p_1 \\ p_2 \end{smallmatrix}[T]$	element, $p_1 \in P_c$, $p_2 \in P_r$
	$\mid$	$T.T$	concatenation
	$\mid$	$T \mid T$	union
	$\mid$	$T \& T$	intersection
	$\mid$	A	VPE variable
	$\mid$	$T+$	repetition

And the VPEs are defined by:

S	::=	T	regular expression type
	$\mid$	p	symbol pattern
	$\mid$	$S.S$	concatenation
	$\mid$	$S \oplus S$	overlapped concatenation
	$\mid$	$S \mid S$	union
	$\mid$	$S \& S$	intersection
	$\mid$	$S+$	repetition

We omit the empty sequence inside an element, i.e., we write $\begin{smallmatrix} p_1 \\ p_2 \end{smallmatrix}[]$ instead of $\begin{smallmatrix} p_1 \\ p_2 \end{smallmatrix}[()]$. If $\texttt{<}l\texttt{>} \in \Sigma_c$ and $\texttt{</}l\texttt{>} \in \Sigma_r$, then we use the abbreviation $l[T] = \begin{smallmatrix} \texttt{<}l\texttt{>} \\ \texttt{</}l\texttt{>} \end{smallmatrix}[T]$.

Call and return symbols will match in words in the denotation of a matched VPE, i.e., the words represent trees. VPEs in general allow call and return symbols to appear unmatched using symbol patterns. The overlapped concatenation operator is not permitted on matched VPEs because it could generate unmatched call or return symbols.

VPE variables are bound to matched VPEs (not VPEs) by an environment, ranged over by E. As with regular expression types, we impose a constraint that prevents recursive uses of variables unless they are guarded by an element. We formalize the constraint as an inductively defined rewriting relation $S_1 \Downarrow S_2$ that recursively expands variables in S_1, stopping when it reaches an element. The result S_2 denotes the same language, which we exploit in a later normalization result (lemma 5.2). The constraint on bindings is that, for each variable A,

there is a matched VPE T such that $A \Downarrow T$.

$$\frac{}{() \Downarrow ()} \qquad \frac{}{\vdash p \Downarrow p} \qquad \frac{}{\vdash {}^{p_1}_{p_2}[T] \Downarrow {}^{p_1}_{p_2}[T]}$$

$$\frac{S_1 \Downarrow S_1' \qquad S_2 \Downarrow S_2'}{S_1.S_2 \Downarrow S_1'.S_2'} \qquad \frac{S_1 \Downarrow S_1' \qquad S_2 \Downarrow S_2'}{S_1 \oplus S_2 \Downarrow S_1' \oplus S_2'}$$

$$\frac{S_1 \Downarrow S_1' \qquad S_2 \Downarrow S_2'}{S_1|S_2 \Downarrow S_1'|S_2'} \qquad \frac{S_1 \Downarrow S_1' \qquad S_2 \Downarrow S_2'}{S_1\&S_2 \Downarrow S_1'\&S_2'}$$

$$\frac{E(A) \Downarrow S}{A \Downarrow S} \qquad \frac{S \Downarrow S'}{S+ \Downarrow S'+}$$

A VPE S and a binding E denote a set of words over Σ. Membership of a word in the denotation of S is defined inductively by:

$$\frac{}{\vdash () : ()} \qquad \frac{a \in [\![p]\!]_{\tilde{\Sigma}}}{\vdash a : p} \qquad \frac{a_1 \in [\![p_1]\!]_{\tilde{\Sigma}} \quad a_2 \in [\![p_2]\!]_{\tilde{\Sigma}} \quad \vdash \alpha : T}{\vdash a_1.\alpha.a_2 : {}^{p_1}_{p_2}[T]}$$

$$\frac{\vdash \alpha_1 : S_1 \qquad \vdash \alpha_2 : S_2}{\vdash \alpha_1.\alpha_2 : S_1.S_2} \qquad \frac{\vdash \alpha_1.a : S_1 \qquad \vdash a.\alpha_2 : S_2}{\vdash \alpha_1.a.\alpha_2 : S_1 \oplus S_2}$$

$$\frac{\vdash \alpha : S_1}{\vdash \alpha : S_1|S_2} \qquad \frac{\vdash \alpha : S_2}{\vdash \alpha : S_1|S_2} \qquad \frac{\vdash \alpha : S_1 \qquad \vdash \alpha : S_2}{\vdash \alpha : S_1\&S_2}$$

$$\frac{\vdash \alpha : E(A)}{\vdash \alpha : A} \qquad \frac{\vdash \alpha : S}{\vdash \alpha : S+} \qquad \frac{\alpha_1 \neq () \quad \vdash \alpha_1 : S \quad \vdash \alpha_2 : S+}{\vdash \alpha_1.\alpha_2 : S+}$$

Note that ${}^{a_1}_{a_2}[T]$ and $a_1.T.a_2$ denote the same languages. In section 7, we will see that this ability to fold and unfold elements is important when typechecking recursive programs with VPE effects.

When we wish to emphasise the set of symbols $\tilde{\Sigma}$ or the bindings E used, we write $\tilde{\Sigma}, E \vdash \alpha : S$.

If the language defined by a VPE S_1 is a subset of the language defined by a VPE S_2, i.e., $\vdash \alpha : S_1$ implies $\vdash \alpha : S_2$, then we say that S_1 is a subtype of S_2 and write $\tilde{\Sigma}, E \vdash S_1 <: S_2$ or simply $\vdash S_1 <: S_2$.

At the start of this section, we allowed the component alphabets of a pushdown alphabet to be countably infinite. In fact, finite alphabets suffice for testing inclusion as the next lemma shows.

Lemma 4.1 *Consider VPEs S_1, S_2 and binding E over a pushdown alphabet $\tilde{\Sigma}$ with one or more infinite component alphabets. Then there exists a pushdown alphabet $\tilde{\Sigma}'$ with finite component alphabets such that $\tilde{\Sigma}, E \vdash S_1 <: S_2$ iff $\tilde{\Sigma}', E \vdash S_1 <: S_2$.*

We now show that the class of languages defined by VPEs includes the class of VPLs by translating VPAs to VPEs. The proof resembles the proof that VPLs can be represented as tree automata in [3]. We first define a VPE using bindings with unguarded variables in tail positions. Such unguarded variables can be eliminated in place of repetition using standard techniques.

Definition 4.2 For a VPA, define bindings for the variables A_{q_1,q_2}, B_{q_1}, C_{q_2} for all $q_1, q_2 \in Q$ by initially setting each of $E(A_{q_1,q_2})$, $E(B_{q_1})$, and $E(C_{q_2})$ to $\emptyset$, then adding (where we identify $\{S_1, \ldots, S_n\}$ with $S_1| \ldots |S_n$):

- $() \in E(A_{q,q})$.
- If $q \in Q_F$, then $() \in E(B_q)$.
- $B_q \in E(C_q)$
- If $(q_1, a_1, q_2, \gamma) \in \delta_c$ and $(q_3, a_2, \gamma, q_4) \in \delta_r$, then ${}^{a_1}_{a_2}[A_{q_2,q_3}].A_{q_4,q_5} \in E(A_{q_1,q_5})$.
- If $q_2 \in Q$, then $A_{q_1,q_2}.B_{q_2} \in E(B_{q_1})$ and $A_{q_1,q_2}.C_{q_2} \in E(C_{q_1})$.
- If $(q_1, a, q_2, \gamma) \in \delta_c$, then $a.B_{q_2} \in E(B_{q_1})$.
- If $(q_1, \bot, \gamma, q_2) \in \delta_r$, then $a.C_{q_2} \in E(C_{q_1})$.
- If $(q_1, a, q_2) \in \delta_{int}$, then $a.A_{q_2,q_3} \in E(A_{q_1,q_3})$ and $a.B_{q_2} \in E(B_{q_1})$ and $a.C_{q_2} \in E(C_{q_1})$.

□

Each variable A_{q_1,q_2} represents the matched sequences of tokens that can be "accepted" between the states q_1 and q_2. On the other hand, variables of the form B_q represent sequences formed from calls interspersed with words consisting of matched calls and returns that can be "accepted" between q and a final state from Q_F. Finally, each variable C_q represents sequences formed from returns interspersed with words consisting of matched calls and returns, followed by sequences formed from calls interspersed with words consisting of matched calls and returns that can be "accepted" between q and a final state from Q_F.

For $\sigma_1, \sigma_2 \in (\Gamma \setminus \{\bot\}) * . \{\bot\}$, we write $\sigma_1 \sqsupseteq \sigma_2$ when σ_2 is a suffix of σ_1, i.e., there exists σ_3 such that $\sigma_1 = \sigma_3.\sigma_2$.

Theorem 4.3 *Consider a VPA and a sequence $a_1, \ldots, a_n$:*

1. *$\vdash a_1, \ldots, a_n : A_{q_1,q_{n+1}}$ iff there exists a pre-run $(q_1, \sigma_1), \ldots, (q_{n+1}, \sigma_{n+1})$ on $a_1, \ldots, a_n$ such that $\sigma_1 = \sigma_n$ and, for all $1 \leq i \leq n$, $\sigma_i \sqsupseteq \sigma_1$ and $a_i \in \Sigma_{int}$ whenever $\sigma_i = \sigma_{i+1} = \bot$.*

2. $\vdash a_1, \ldots, a_n : B_{q_1}$ *iff there exists a pre-run* $(q_1, \sigma_1), \ldots, (q_{n+1}, \sigma_{n+1})$ *on* $a_1, \ldots, a_n$ *such that* $q_{n+1} \in Q_F$ *and, for all* $1 \leq i \leq n$, $\sigma_i \sqsupseteq \sigma_1$ *and* $a_i \in \Sigma_{int}$ *whenever* $\sigma_i = \sigma_{i+1} = \bot$.

3. $\vdash a_1, \ldots, a_n : C_{q_1}$ *iff there exists a pre-run* $(q_1, \sigma_1), \ldots, (q_{n+1}, \sigma_{n+1})$ *on* $a_1, \ldots, a_n$ *such that* $q_{n+1} \in Q_F$ *and* $\sigma_1 = \bot$.

Corollary 4.4 *Any VPL can be expressed as a VPE.*

5 From VPEs to MSO$_\mu$

In this section we show that VPEs can be translated to MSO$_\mu$ formulae. The decidability of language inclusion between VPEs is an immediate consequence of the translation and the decidability of MSO$_\mu$ satisfaction. The translation acts upon VPEs and bindings over a pushdown alphabet $\tilde{\Sigma}$ with finite component alphabets. The bindings must be in a normal form which ties together elements and variable occurrences.

Definition 5.1 A VPE S and binding E are in normal form if every occurrence of an element or a variable in S or the image of E is of the form ${}^{p_1}_{p_2}[A]$. □

A normal form for bindings can be found by introducing a fresh variable to replace S in an element occurrence ${}^{p_1}_{p_2}[S]$ when S is not a variable, and then using the relation $\Downarrow$ to expand variables occurrences that are not inside elements.

Lemma 5.2 *For all VPEs* S *and bindings* E*, there exists a new VPE* S' *and binding* E' *in normal form, such that* $\tilde{\Sigma}, E \vdash \alpha : S$ *iff* $\tilde{\Sigma}, E' \vdash \alpha : S'$.

Figure 1 defines the translation of a VPE S to a formula $[\![S]\!]^{\mathbf{X}}_{x,y}$ with free first-order variables x and y, and a vector of free second-order variables $\mathbf{X}$, indexed by a call symbol, a return symbol, and a variable. The first-order variables x and y represent the start and end positions (inclusive) of the part of a non-empty word that matches S.

When the inductively defined translation of $[\![S]\!]^{\mathbf{X}}_{x,y}$ reaches an element and variable occurrence ${}^{a_1}_{a_2}[A]$ it checks that the positions x and y match using the μ relation and that x is a member of the set $X_{a_1,a_2,A}$ (drawn from the vector of free second-order variables $\mathbf{X}$). Membership of a position x in $X_{a_1,a_2,A}$ means that the position should be a call symbol a_1 that has a matching return symbol a_2 at position y and the word in between x and y should match $E(A)$. This can be expressed by universal quantification over the set $X_{a_1,a_2,A}$ in tandem with the μ relation. Critically, the μ relation is used to get around the lack of quantification of sets of *pairs* of positions in monadic second order logic.

Empty words are a special case for the translation, because x and y describe the start and end positions of a word inclusively, so the word has length 1 when $x = y$. To identify the special case in the translation, we make use of a propositional formula $empty(S)$, for each VPE S, defined by:

$$empty(S) = \begin{cases} \top & \text{if} \vdash () : S \\ \bot & \text{otherwise} \end{cases}$$

An entirely empty word is described using the formula $\exists x.\bot$, which means that no positions exist. The formulae $\min(x)$ and $\max(y)$ assert that x and y are the minimal and maximal positions respectively.

Theorem 5.3 *Consider a VPE* S_0 *and binding* E *for the variables* $A_1 \ldots A_m$*, over a pushdown alphabet* $\tilde{\Sigma}$ *with finite component alphabets, in normal form. Then, for all words* $\alpha \in \Sigma*$, $\vdash \alpha : S_0$ *iff* $\alpha \models \psi_{S_0}$*, where* ψ_{S_0} *is defined in figure 1.*

Corollary 5.4 *Testing language inclusion between VPEs is decidable.*

6 Syntax and Semantics

In this section we present the syntax and operational semantics for λ_{str}, a λ-calculus with operations for reading tokens from input streams and writing tokens to output streams. Programs can examine the current token on an input stream without consuming it, providing a lookahead of one token, but they do not have random access to streams' contents.

We assume disjoint sets of input streams *In* and output streams *Out*, and use s to range over $In \cup Out$. For each stream, there must be a pushdown alphabet $\tilde{\Sigma^s} = (\Sigma^s_c, \Sigma^s_r, \Sigma^s_{int})$ with $\mathsf{string} \in \Sigma^s_{int}$. The pushdown alphabets need not be disjoint for different streams.

Terms and values are defined by the grammars:

$L, M, N ::=$		term
	$*$	singleton
$\mid$	w	string literal
$\mid$	x	variable
$\mid$	$\mathsf{fun}(f)(x)M$	abstraction
$\mid$	$M\ N$	application
$\mid$	$\mathsf{let}\ x = M\ \mathsf{in}\ N$	sequencing
$\mid$	$s?\sim$	read token on $s \in In$
$\mid$	val_s	read string from $s \in In$
$\mid$	$s!a$	write $a \in \tilde{\Sigma}^s$ to $s \in Out$
$\mid$	$s!M$	write string to $s \in Out$
$\mid$	fail_s	fail on $s \in In$
$\mid$	$\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ N$	test token on $s \in In$
$V ::=$		value
	$*$	singleton
$\mid$	w	string literal
$\mid$	$\mathsf{fun}(f)(x)M$	abstraction

The program $s?\sim$ destructively reads the current token from input stream s, regardless of whether it is a call, return, or internal token. In contrast, $\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ N$ non-destructively tests the current token against a symbol pattern p over $\tilde{\Sigma}^s$, and val_s non-destructively reads, and evaluates to, the current token which must be a string. The program fail_s is used to express the fact that a test has failed and that an input stream cannot be accepted. Finally, $s!a$ and $s!M$ write the token a and the string literal resulting from evaluating M to the output stream s.

The reduction semantics relates pairs of terms and stream configurations, the latter representing the input remaining on input streams and the output that has taken place on output streams. Stream configurations are defined by the grammar:

C	$::=$	$\emptyset$	
	$\mid$	$s?\alpha$	$(s \in In)$
	$\mid$	$s!\alpha$	$(s \in Out)$
	$\mid$	C, C	

Words in configurations contain string literals instead of the token string. We consider stream configurations modulo identity, associativity, and commutativity, and do not use stream configurations that mention a stream more than once. When convenient, we regard stream configurations as partial functions from $In \cup Out$ to words, and write $C(s) = \alpha$ when there exists C' such that $C = C', s?\alpha$ or $C = C', s!\alpha$.

The one-step reduction of a term M_1 and a stream configuration C_1 to M_2 and C_2 is written $M_1; C_1 \twoheadrightarrow M_2; C_2$. The reduction relation is defined in figure 2, where $M\{N/x\}$ denotes the capture-free substitution of N for x in M. Note that fail_s is neither a value nor reducible.

Later examples use the following abbreviations, where x and f are fresh variables:

$$M; N \stackrel{\text{def}}{=} \mathsf{let}\ x = M\ \mathsf{in}\ N$$

$$s?\mathsf{string} \stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{val}_s\ \mathsf{in}\ s?\sim; x$$

$$s?p \stackrel{\text{def}}{=} \mathsf{if}\ (s \rhd p)\ \mathsf{then}\ s?\sim\ \mathsf{else}\ \mathsf{fail}_s$$

$$\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M \stackrel{\text{def}}{=} \mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ *$$

$$\mathsf{while}\ (s \rhd p)M \stackrel{\text{def}}{=} (\mathsf{fun}(f)(x)\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M; f(*)\ \mathsf{else}\ *)(*)$$

In contrast to $s?\sim$, which destructively reads the current token, the program $s?p$ destructively reads the current token only if it matches the symbol pattern p, otherwise it reduces to fail_s. The special case for $s?\mathsf{string}$ also returns the string itself.

Example 6.1 For any token $a \in \tilde{\Sigma}^s$, the program $\mathsf{while}\ (s \rhd a)s?\sim$ destructively reads all of the a tokens at the start of stream s. After running this program, the current token on stream s cannot be an a token, so the following program will always fail when it executes $s?a$:

$$(\mathsf{while}\ (s \rhd a)s?\sim); s?a$$

Example 7.1 demonstrates how the type and effect system identifies the error in the above program by tracking the possible values of the current token for each input stream, in addition to the tokens that have been destructively read. □

When $\texttt{<}l\texttt{>} \in \Sigma_c^s$ and $\texttt{</}l\texttt{>} \in \Sigma_r^s$ we also use the abbreviations:

$$s?l[M] \stackrel{\text{def}}{=} s?\texttt{<}l\texttt{>}; \mathsf{let}\ x = M\ \mathsf{in}\ s?\texttt{</}l\texttt{>}; x$$

$$s!l[M] \stackrel{\text{def}}{=} s!\texttt{<}l\texttt{>}; \mathsf{let}\ x = M\ \mathsf{in}\ s!\texttt{</}l\texttt{>}; x$$

However, these abbreviations can be misleading. For example, $s!l[s!\texttt{</}l\texttt{>}; s!\texttt{<}l\texttt{>}]$ is a legitimate program.

Example 6.2 illustrates the use of abbreviations for reading and writing elements and their interaction with the pattern-matching construct that operates on tokens.

Example 6.2 Suppose that customer records have names and optional addresses, and are stored as $\texttt{customers}[(\texttt{name}[T_1], \texttt{address}[T_2]?)*]$. The following program wraps a `customer` element around each name and address, if present, assuming the existence of programs M and N that copy names and addresses respec-

$$
\begin{aligned}
[\![\emptyset]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \bot \\
[\![()]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \bot \\
[\![a]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} x = y \wedge x \in Q_a \\
[\![{}^{a_1}_{a_2}[A]]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \mu(x,y) \wedge x \in X_{a_1,a_2,A} \\
[\![S_1.S_2]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \Big(\mathit{empty}(S_1) \wedge [\![S_2]\!]^{\mathbf{X}}_{x,y}\Big) \vee \Big([\![S_1]\!]^{\mathbf{X}}_{x,y} \wedge \mathit{empty}(S_2)\Big) \vee \Big(\exists z.x \leq z < y \wedge [\![S_1]\!]^{\mathbf{X}}_{x,z} \wedge [\![S_2]\!]^{\mathbf{X}}_{z+1,y}\Big) \\
[\![S_1 \oplus S_2]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \exists z.x \leq z \leq y \wedge [\![S_1]\!]^{\mathbf{X}}_{x,z} \wedge [\![S_2]\!]^{\mathbf{X}}_{z,y} \\
[\![S_1|S_2]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} [\![S_1]\!]^{\mathbf{X}}_{x,y} \vee [\![S_2]\!]^{\mathbf{X}}_{x,y} \\
[\![S_1\&S_2]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} [\![S_1]\!]^{\mathbf{X}}_{x,y} \wedge [\![S_2]\!]^{\mathbf{X}}_{x,y} \\
[\![S+]\!]^{\mathbf{X}}_{x,y} &\stackrel{\text{def}}{=} \exists Z.\Big(\forall x', y'.(x' \in Z \wedge y'+1 \in Z \wedge [\![S]\!]^{\mathbf{X}}_{x',y'}) \Longrightarrow \exists z.\, [\![S]\!]^{\mathbf{X}}_{y'+1,z} \wedge (z = y \vee z+1 \in Z)\Big) \\
&\qquad \wedge x \in Z \wedge \exists z.\, \Big([\![S]\!]^{\mathbf{X}}_{x,z} \wedge (z = y \vee z+1 \in Z)\Big) \\
\phi^{\mathbf{X}}_{a_1,a_2,A} &\stackrel{\text{def}}{=} \forall x \in X_{a_1,a_2,A}.\exists y.\mu(x,y) \wedge x \in Q_{a_1} \wedge y \in Q_{a_2} \wedge \Big((\mathit{empty}(E(A)) \wedge x+1 = y) \vee \Big([\![E(A)]\!]^{\mathbf{X}}_{x+1,y-1}\Big)\Big) \\
\psi_S &\stackrel{\text{def}}{=} (\mathit{empty}(S) \wedge \exists x.\bot) \vee \exists \mathbf{X}.\exists x, y.\, \min(x) \wedge \max(y) \wedge [\![S]\!]^{\mathbf{X}}_{x,y} \wedge \bigwedge_{\substack{a_1 \in \Sigma_c \\ a_2 \in \Sigma_r \\ 1 \leq i \leq m}} \phi^{\mathbf{X}}_{a_1,a_2,A_i}
\end{aligned}
$$

Figure 1: Translation from VPEs to Formulae

tively:

$s?\texttt{customers}[$
  $t!\texttt{customers}[$
    while $(s \triangleright \texttt{<name>})$
      $t!\texttt{customer}[$
        M;
        if $(s \triangleright \texttt{<address>})$ then N
      $]$
  $]$
$]$

□

Example 6.3 illustrates the need for primitives that read and write tokens individually, rather than taking $s?l[M]$ and $t!l[M]$ as the primitives.

Example 6.3 Suppose that paragraphs consist of bold and italic elements $\texttt{p}[(\texttt{b}[]|\texttt{i}[])*]*$ and we wish to copy the bold and italic elements in their original order and to group them into new paragraphs, so that the output is $(\texttt{p}[\texttt{b}[]*]|\texttt{p}[\texttt{i}[]*])*$. For example, we translate:

```
<p></p>
<p>
  <i></i>
  <b></b>
</p>
<p>
  <b></b>
  <i></i>
</p>
```

to

```
<p>
  <i></i>
</p>
<p>
  <b></b>
  <b></b>
</p>
<p>
  <i></i>
</p>
```

There is no correspondence between the original paragraphs and the new paragraphs that would allow programs of the form $s?\texttt{p}[\ldots t!\texttt{p}[\ldots]\ldots]$ or $t!\texttt{p}[\ldots s?\texttt{p}[\ldots]\ldots]$ to work. It would be possible to read one paragraph at a time, storing data in an intermediate data structure (extending λ_{str} if necessary), and then check whether to write a new paragraph in its entirety at the end of each original paragraph. However, this is contrary to the goal of minimizing space requirements. The solution is to write new paragraph end and start tags whenever a transition between bold and italic

$$(\mathsf{fun}(f)(x)M)\ V; C \twoheadrightarrow M\{\mathsf{fun}(f)(x)M, V/f, x\}; C \qquad \mathsf{let}\ x = V\ \mathsf{in}\ N; C \twoheadrightarrow N\{V/x\}; C$$

$$s?{\sim}; C, s?a.\alpha \twoheadrightarrow *; C, s?\alpha \qquad \mathsf{val}_s; C, s?w.\alpha \twoheadrightarrow w; C, s?w.\alpha$$

$$s!a; C, s!\alpha \twoheadrightarrow *; C, s!\alpha.a \qquad s!w; C, s!\alpha \twoheadrightarrow *; C, s!\alpha.w$$

$$\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ N; C, s?a.\alpha \twoheadrightarrow M; C, s?a.\alpha \qquad \text{if } a \in [\![p]\!]_{\tilde{\Sigma}^s}$$

$$\mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ N; C, s?a.\alpha \twoheadrightarrow N; C, s?a.\alpha \qquad \text{if } a \notin [\![p]\!]_{\tilde{\Sigma}^s}$$

$$\frac{M_1; C_1 \twoheadrightarrow M_2; C_2}{M_1\ N; C_1 \twoheadrightarrow M_2\ N; C_2} \qquad \frac{N_1; C_1 \twoheadrightarrow N_2; C_2}{V\ N_1; C_1 \twoheadrightarrow V\ N_2; C_2} \qquad \frac{M_1; C_1 \twoheadrightarrow M_2; C_2}{\mathsf{let}\ x = M_1\ \mathsf{in}\ N; C_1 \twoheadrightarrow \mathsf{let}\ x = M_2\ \mathsf{in}\ N; C_2} \qquad \frac{M_1; C_1 \twoheadrightarrow M_2; C_2}{c!M_1; C_1 \twoheadrightarrow c!M_2; C_2}$$

Figure 2: Reduction Semantics

elements is found:

$f(x) \stackrel{\text{def}}{=} \mathsf{while}\,(s \rhd \texttt{<p>})$
$\quad s?{\sim};$
$\quad \mathsf{if}\ (s \rhd \texttt{<b>})\ \mathsf{then}\ t!\texttt{<p>}; g(*)$
$\quad \mathsf{else\ if}\ (s \rhd \texttt{<i>})\ \mathsf{then}\ t!\texttt{<p>}; h(*)$
$\quad \mathsf{else}\ s?\texttt{</p>}$

$g(x) \stackrel{\text{def}}{=} \mathsf{if}\ (s \rhd \texttt{<b>})\ \mathsf{then}\ s?\texttt{b}[t!\texttt{b}[*]]; g(*)$
$\quad \mathsf{else\ if}\ (s \rhd \texttt{<i>})\ \mathsf{then}\ t!\texttt{</p>}; t!\texttt{<p>}; h(*)$
$\quad \mathsf{else}\ s?\texttt{</p>}; \mathsf{if}\ (s \rhd \texttt{<p>})\ \mathsf{then}\ s?{\sim}; g(*)\ \mathsf{else}\ t!\texttt{</p>}$

$h(x) \stackrel{\text{def}}{=} \mathsf{if}\ (s \rhd \texttt{<b>})\ \mathsf{then}\ t!\texttt{</p>}; t!\texttt{<p>}; g(*)$
$\quad \mathsf{else\ if}\ (s \rhd \texttt{<i>})\ \mathsf{then}\ s?\texttt{i}[t!\texttt{i}[*]]; h(*)$
$\quad \mathsf{else}\ s?\texttt{</p>}; \mathsf{if}\ (s \rhd \texttt{<p>})\ \mathsf{then}\ s?{\sim}; h(*)\ \mathsf{else}\ t!\texttt{</p>}$

When the function g is called, both the input stream and output stream should be inside paragraph elements, and the output stream's current paragraph should only contain bold elements. Similarly for h and italic elements. □

7 Type and Effect System

In this section we present a type and effect system for λ_{str}, where the effects assign VPEs to each input and output stream, and the subeffecting relation is defined in terms of language inclusion. We conclude with a subject reduction and progression result.

Effects are defined by the following grammar:

ϵ	::=	$\emptyset$	empty effect
	$\mid$	$s?S$	read S on $s \in In$
	$\mid$	$s!S$	write S on $s \in Out$
	$\mid$	ϵ, ϵ	

If an effect $s?S_1, t!S_2$ is assigned to a program, then the program will consume all but the final token on input stream s matching the VPE S_1 and write output to stream t that matches the VPE S_2 (assuming that the program does not diverge). The empty effect $\emptyset$ means that the program does not perform any input or output.

As with configurations, we consider effects modulo identity, associativity, and commutativity, and do not use effects that mention a stream more than once. When convenient, we regard effects as partial functions from $In \cup Out$ to VPEs, and write $\epsilon(s) = S$ when $\epsilon = \epsilon', s?S$ or $\epsilon = \epsilon', s!S$.

The subeffecting relation is a preorder defined by:

$$\frac{}{\vdash \epsilon <: \epsilon} \qquad \frac{\vdash \epsilon_1 <: \epsilon_2 \qquad \vdash \epsilon_2 <: \epsilon_3}{\vdash \epsilon_1 <: \epsilon_3}$$

$$\frac{\vdash \epsilon_1 <: \epsilon_2}{\vdash \epsilon_1 <: s?{\sim}, \epsilon_2} \qquad \frac{\vdash \epsilon_1 <: \epsilon_2}{\vdash \epsilon_1 <: s!(), \epsilon_2}$$

$$\frac{\vdash S_2 <: S_1 \qquad \vdash \epsilon_1 <: \epsilon_2}{\vdash s?S_1, \epsilon_1 <: s?S_2, \epsilon_2} \qquad \frac{\vdash S_1 <: S_2 \qquad \vdash \epsilon_1 <: \epsilon_2}{\vdash s!S_1, \epsilon_1 <: s!S_2, \epsilon_2}$$

The subeffecting relation is decidable because of the decision procedure for inclusion between VPEs sketched in section 5.

Note that if effect ϵ_1 does not contain stream s, then $\vdash \epsilon_1 <: s?S, \epsilon_2$ implies that $S <: {\sim}$, and $\vdash \epsilon_1 <: s!S, \epsilon_2$ implies that $() <: S$. The reason for using $\vdash \epsilon_1 <: s?{\sim}, \epsilon_2$ instead of $\vdash \epsilon_1 <: s?(), \epsilon_2$ is that input streams will normally have a current token that is not read destructively, possibly representing the end of the stream. When a program terminates, it may know something about the current token because it has performed a non-destructive read. In this case, a value can be assigned the effect $s?{\sim}$, which means that nothing is known about the current token on the input stream s.

Types are simply:

$$\sigma, \tau ::= \mathsf{unit} \mid \mathsf{string} \mid \sigma \xrightarrow{\epsilon} \tau$$

The subtyping relation is defined by:

$$\frac{}{\vdash \sigma <: \sigma} \qquad \frac{\vdash \sigma_1 <: \sigma_2 \qquad \vdash \sigma_2 <: \sigma_3}{\vdash \sigma_1 <: \sigma_3}$$

$$\frac{\vdash \sigma_2 <: \sigma_1 \qquad \vdash \tau_1 <: \tau_2 \qquad \vdash \epsilon_1 <: \epsilon_2}{\vdash \sigma_1 \xrightarrow{\epsilon_1} \tau_1 <: \sigma_2 \xrightarrow{\epsilon_2} \tau_2}$$

As discussed in the introduction, the type assignment rules for sequential composition and function application combine input and output effects using $\oplus$ (overlapped concatenation) and . (concatenation) respectively. To specify this more concisely, we define the $\odot$ operator on effects with the same domains by:

$$\emptyset \odot \emptyset \stackrel{\text{def}}{=} \emptyset$$
$$(s?S_1, \epsilon_1) \odot (s?S_2, \epsilon_2) \stackrel{\text{def}}{=} s?(S_1 \oplus S_2), (\epsilon_1 \odot \epsilon_2)$$
$$(s!S_1, \epsilon_1) \odot (s!S_2, \epsilon_2) \stackrel{\text{def}}{=} s!(S_1.S_2), (\epsilon_1 \odot \epsilon_2)$$

Type assignment judgements have the form $\Gamma \vdash M : \sigma; \epsilon$ and are defined in figure 3.

Perhaps the most surprising rule is the axiom $\Gamma \vdash s?\sim : \mathsf{unit}; s?\sim.\sim$. This means that the program consumes one token (the first instance of $\sim$), which leaves us knowing nothing about the next token (the second instance of $\sim$).

As mentioned above, the sequential composition and function application rules make use of the overlapped concatenation operator of VPEs to ensure that the postcondition of one program matches part of the precondition of the next program. In addition, the rules for matching are able to use overlapped concatenation to restrict S_1 and S_2 in such a way that the program is guaranteed to accept the union of the two resulting languages on input stream s.

The following type assignment rule can be derived for while loops:

$$\frac{\Gamma \vdash M : \mathsf{unit}; s?S_1, t!S_2 \qquad S_1' <: (p \oplus S_1 \oplus S_1') \mid \neg p}{\Gamma \vdash \mathsf{while}\,(s \rhd p) M : \mathsf{unit}; s?S_1', t!S_2 *}$$

Example 7.1 The programs in example 6.1 are assigned types and effects:

$$\vdash \mathsf{while}\,(s \rhd a) s?\sim : \mathsf{unit}; s?a*.\neg a$$
$$\vdash s?a : \mathsf{unit}; s?a.\sim$$

The lookahead token for the first program, $\neg a$, causes the composition of the two programs to fail, because:

$$\vdash (\mathsf{while}\,(s \rhd a) s?\sim); s?a : \mathsf{unit}; s?(a*.\neg a) \oplus (a.\sim)$$

And $(a*.\neg a) \oplus (a.\sim) = a*.\emptyset.\sim = \emptyset$, so:

$$\vdash (\mathsf{while}\,(s \rhd a) s?\sim); s?a : \mathsf{unit}; s?\emptyset$$

Although there is a derivation, it is a failure in the sense that input stream s has been assigned the empty language as its effect, and the subject reduction theorem (theorem 7.7) makes no guarantees about such programs. □

Example 7.2 demonstrates that programs can have effects that are VPLs but not regular languages (on words rather than trees).

Example 7.2 If $A = l[A]*$ and $\epsilon = s?A.\neg\texttt{<}l\texttt{>}, t!A$ then:

$$\vdash \mathsf{fun}(f)(x)\mathsf{if}\ (s \rhd \texttt{<}l\texttt{>})\ \mathsf{then}\ s?l[t!l[f(*)]]; f(*)$$
$$: \mathsf{unit} \xrightarrow{\epsilon} \mathsf{unit}; \emptyset$$

□

Example 7.3 The customers program of example 6.2 has the effect $s?S, t!S'$ where the VPEs are:

$$S = \texttt{customers}[(\texttt{name}[T_1], \texttt{address}[T_2]?)*].\sim$$
$$S' = \texttt{customers}[\texttt{customer}[\texttt{name}[T_1'], \texttt{address}[T_2']?]*]$$

When we assume derivations of:

$$\vdash M : \mathsf{unit}; s?\texttt{name}[T_1].\sim, t!\texttt{name}[T_1']$$
$$\vdash N : \mathsf{unit}; s?\texttt{address}[T_2].\sim, t!\texttt{address}[T_2']$$

□

Example 7.4 Recall the paragraph splitting programs of example 6.3. It is possible to assign the types $f : \mathsf{unit} \xrightarrow{\epsilon_f} \mathsf{unit}$, $g : \mathsf{unit} \xrightarrow{\epsilon_g} \mathsf{unit}$, and $h : \mathsf{unit} \xrightarrow{\epsilon_h} \mathsf{unit}$, where $\epsilon_f = s?S_f, t!S_f'$, $\epsilon_g = s?S_g, t!S_g'$, and $\epsilon_h = s?S_h, t!S_h'$. The VPEs are:

$$S_f = \texttt{p}[(\texttt{b}[]|\texttt{i}[])*]*.\neg\texttt{<p>} \qquad S_f' = (\texttt{p}[\texttt{b}[]*]|\texttt{p}[\texttt{i}[]*])*$$
$$S_g = (\texttt{b}[]|\texttt{i}[])*.\texttt{</p>}.S_f \qquad S_g' = \texttt{b}[]*.\texttt{</p>}.S_f'$$
$$S_h = (\texttt{b}[]|\texttt{i}[])*.\texttt{</p>}.S_f \qquad S_h' = \texttt{i}[]*.\texttt{</p>}.S_f'$$

Thus, despite the complexity of the original program, we can provide, and verify, a concise description of its behaviour in terms of the languages S_f and S_f'. □

Example 7.5 Hosoya and Pierce's XDuce address book example from [24] can be translated easily to λ_{str}. The program reads address book entries with optional telephone numbers, but only writes address book entries for those with telephone numbers, whilst removing

$$\frac{\Gamma \vdash M : \sigma; \epsilon_1 \qquad \vdash \sigma <: \tau \qquad \vdash \epsilon_1 <: \epsilon_2}{\Gamma \vdash M : \tau; \epsilon_2} \qquad \frac{}{\Gamma \vdash * : \mathsf{unit}; \emptyset} \qquad \frac{}{\Gamma \vdash w : \mathsf{string}; \emptyset} \qquad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma; \emptyset}$$

$$\frac{\Gamma, f : \sigma \xrightarrow{\epsilon} \tau, x : \sigma \vdash M : \tau; \epsilon}{\Gamma \vdash \mathsf{fun}(f)(x)M : \sigma \xrightarrow{\epsilon} \tau; \emptyset} \qquad \frac{\Gamma \vdash M : \sigma \xrightarrow{\epsilon_3} \tau; \epsilon_1 \qquad \Gamma \vdash N : \sigma; \epsilon_2}{\Gamma \vdash M\ N : \tau; \epsilon_1 \odot \epsilon_2 \odot \epsilon_3}$$

$$\frac{\Gamma \vdash M : \sigma; \epsilon_1 \qquad \Gamma, x : \sigma \vdash N : \tau; \epsilon_2}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau; \epsilon_1 \odot \epsilon_2} \qquad \frac{\Gamma \vdash M : \sigma; \epsilon, s?S_1 \qquad \Gamma \vdash N : \sigma; \epsilon, s?S_2}{\Gamma \vdash \mathsf{if}\ (s \rhd p)\ \mathsf{then}\ M\ \mathsf{else}\ N : \sigma; \epsilon, s?(p \oplus S_1)|(\neg p \oplus S_2)}$$

$$\frac{}{\Gamma \vdash s?\sim : \mathsf{unit}; s?\sim.\sim} \qquad \frac{}{\Gamma \vdash \mathsf{val}_s : \mathsf{string}; s?\mathsf{string}} \qquad \frac{a \neq \mathsf{string}}{\Gamma \vdash s!a : \mathsf{unit}; s!a} \qquad \frac{\Gamma \vdash M : \mathsf{string}; \epsilon, s!S}{\Gamma \vdash s!M : \mathsf{unit}; \epsilon, s!S.\mathsf{string}} \qquad \frac{}{\Gamma \vdash \mathsf{fail}_s : \sigma; \epsilon, s?\emptyset}$$

Figure 3: Type Assignment

email addresses:

```
while (s ▷ <person>)
  s?person[
    let x = s?name[s?string] in
    while (s ▷ <email>)s?email[s?~];
    if (s ▷ <tel>)then
      t!person[
        t!name[x];
        t!tel[t!(s?tel[s?string])]
      ]
  ]
```

This program can be assigned the following VPEs for the input and output effects respectively:

person[name[string].email[string]∗.tel[string]?]∗.¬<person>

person[name[string].tel[string]]∗

□

Subject Reduction and Progression

The subject reduction proof depends upon the following substitution lemma that allows values to be substituted into other terms. The value may have an effect ϵ_1 other than $\emptyset$, but it can be shown that $\vdash \emptyset <: \epsilon_1$ whenever $\Gamma \vdash V : \sigma; \epsilon_1$, so ϵ_1 does not appear in the final effect for the substitution.

Lemma 7.6 (Substitution) *If* $\Gamma \vdash V : \sigma; \epsilon_1$ *and* $\Gamma, x : \sigma \vdash N : \tau; \epsilon_2$, *then* $\Gamma \vdash N\{V/x\} : \tau; \epsilon_2$.

Theorem 7.7 provides both a subject reduction and progression result simultaneously because reduction between configurations is deterministic. In the statement of the theorem we regard a word as a VPE and implicitly map string literals to the token string, allowing us to write, for example, $C_2(s).\epsilon_3(s) <: C_1(s).\epsilon_1(s)$.

Theorem 7.7 (Subject Reduction and Progression) *If* $\vdash M : \sigma; \epsilon_1$ *is not a value,* C_1 *is a stream configuration such that* $Dom(\epsilon_1) \subseteq Dom(C_1)$, *and* ϵ_2 *is an effect such that* $Dom(\epsilon_1) \cap In \subseteq Dom(\epsilon_2)$ *and, for all* $s \in Dom(\epsilon_1) \cap In$, $\vdash C_1(s) : \epsilon_1(s) \oplus \epsilon_2(s)$, *then there exists a term* N *and a stream configuration* C_2 *such that* $M; C_1 \twoheadrightarrow N; C_2$. *In addition, there is an effect* ϵ_3 *such that* $\vdash N : \sigma; \epsilon_3$, $Dom(\epsilon_1) = Dom(\epsilon_3)$, $Dom(C_1) = Dom(C_2)$, *and:*

1. *For all* $s \in Dom(\epsilon_3) \cap In$, $\vdash C_2(s) : \epsilon_3(s) \oplus \epsilon_2(s)$.
2. *For all* $s \in Dom(\epsilon_3) \cap Out$, $C_2(s) \cdot \epsilon_3(s) <: C_1(s) \cdot \epsilon_1(s)$.
3. *For all* $s \in Dom(C_2) \setminus Dom(\epsilon_3)$, $C_1(s) = C_2(s)$.

Now we can show that every well-typed program, starting with input streams matching the effects, either diverges or converges to a value after reading all of its input apart from one token on each input stream, and writing output that matches the corresponding effect for each output stream.

Corollary 7.8 *Consider* $\vdash M : \sigma; \epsilon$ *and* C_1 *satisfying* $Dom(\epsilon) \subseteq Dom(C_1)$. *If* $\vdash C_1(s) : \epsilon(s)$, *whenever* $s \in Dom(\epsilon) \cap In$, *and* $\vdash C_1(s) : ()$, *whenever* $s \in Dom(\epsilon) \cap Out$, *then either* $M; C_1$ *diverges or there exists a value* V *such that* $M; C_1 \twoheadrightarrow^* V; C_2$ *where* $\vdash V : \sigma; \emptyset$ *and:*

1. *For all* $s \in Dom(\epsilon) \cap In$, $\vdash C_2(s) : \sim$.
2. *For all* $s \in Dom(\epsilon) \cap Out$, $\vdash C_2(s) : \epsilon(s)$.

8 Conclusions

We have identified a new notation, visibly pushdown expressions (VPEs), for visibly pushdown languages.

VPEs generalize the well-known regular expression types, often used for XML, in two ways that make them useful for describing the behaviour of stream-based processors: VPEs can describe unmatched start tag or end tag tokens, and they support an overlapped concatenation operation. In addition, the technique that we use to prove decidability of inclusion testing between VPEs is sufficiently general that it handles an intersection operation in the VPE syntax with ease. We have illustrated the use of VPEs for analyzing stream-based processors via a type and effect system for λ_{str}, which includes a concise coding of preconditions and postconditions for input streams in the presence of non-destructive reads using the overlapped concatenation operation on VPEs.

The most pressing future work is to develop implementations of the decision procedure for inclusion testing that are efficient in practice. We plan to investigate how implementation techniques identified in the MONA project [16, 26, 27, 28] carry over to MSO_μ.

We also intend to investigate extensions to the programming language and effect system that will assign a useful type and effect to the identity transformation copying from an input stream to an output stream. The most relevant work in this area appears to be Hole and Gay's bounded polymorphism in session types [17].

8.1 Acknowledgements

I would like to thank Radha Jagadeesan, Alan Jeffrey, Will Marrero, Theresa Walunas, and the anonymous reviewers for their comments.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of VLDB 2000*, pages 53–64, 2000.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *36th ACM Symposium on Theory of Computing*, 2004.

[4] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. An algorithm for streaming XPath processing with forward and backward axes. In *PLAN-X: Programming Language Technologies for XML, Pittsburgh PA*, 2002.

[5] J. Berstel and L. Boasson. *Formal and Natural Computing: Essays dedicated to Grzegorz Rozenberg*, chapter Balanced grammars and their languages, pages 3–25. LNCS 2300.

[6] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. Hancock: A language for processing very large-scale data. In *USENIX 2nd Conference on Domain-Specific Languages*, pages 163–176, 1999.

[7] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, second edition. `http://www.w3.org/TR/REC-xml`

[8] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17. ACM Press, 2000.

[9] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems*, 28(4):467–516, December 2003.

[10] E. P. Friedman. The inclusion problem for simple languages. *Theoretical Computer Science*, 1:297–316, 1976.

[11] S. J. Gay and M. J. Hole. Types for correct communication in client-server systems. Technical Report CSD-TR-00-07, Department of Computer Science, Royal Holloway, 2000.

[12] S. J. Gay and M. J. Hole. Types and subtypes for correct communication in client-server systems. Technical Report TR-2003-131, Department of Computer Science, Royal Holloway, 2003.

[13] S. J. Gay, V. Vasconcelos, and A. Ravara. Session types for inter-process communication. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow, March 2003.

[14] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, pages 173–189, 2003.

[15] Ashish Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 2003.

[16] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[17] M. Hole and S. J. Gay. Bounded polymorphism in session types. Technical Report TR-2003-132, Department of Computing Science, University of Glasgow, March 2003.

[18] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer-Verlag, 1993.

[19] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[20] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, December 2000.

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

[22] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

[23] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases (WebDB)*, May 2000.

[24] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2), May 2003.

[25] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2001.

[26] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS, 1998. LNCS 1414.

[27] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.

[28] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions On Software Engineering*, 1999.

[29] P. M. Lewis and R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3), July 1968.

[30] 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.

[31] Simple API for XML (SAX), 1998. `http://www.saxproject.org/`.

[32] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Proceedings 21th Symposium on Principles of Database Systems (PODS 2002)*, 2002.

[33] *Streaming API for XML (StAX): JSR-173 Specification*, October 2003. Version 1.0, `http://jcp.org/en/jsr/detail?id=173`.

[34] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In *Proceedings of Workshop on Types in Programming (TIP'02), Dagstuhl, Germany, 2002*, volume 75 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, February 2003.

[35] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer-Verlag, 1994.

[36] XML Pull API, 2002. `http://www.xmlpull.org/`.

[37] XPath 1.0, 1999. `http://www.w3.org/TR/xpath`.