Visibly Pushdown Expression Effects
for XML Stream Processing

Corin Pitcher

DePaul University
XML Stream Processing

Numerous APIs for Java, C#, Perl to operate upon XML streams

- Reads/writes upon XML streams, and other operations, can be interleaved: large documents processed with little memory.

But

- Low level: hand-coded predictive parsers.

- No specification/analysis of the I/O behaviour of programs.

This talk: specification/analysis of the I/O behaviour of programs that use existing APIs to operate upon XML streams.
Challenges

Usual framework:

- Languages describe I/O behaviour.
- Language inclusion determines subtyping/subeffecting.

Challenges:

- Unbalanced XML streams, e.g.,
  
  `<person><name>Alice</name>`

- Lookahead and sequential composition.

Type and effect system based upon Visibly Pushdown Languages [Alur, Madhusudan 2004] resolves these problems.
Outline

Programming model and examples

Visibly Pushdown Expressions

Effect assignment

Decidability of subeffecting and Visibly Pushdown Automata

Summary
Pull API Model

Input

| <person> | <name> | Alice | </name> | <name> | Bob | </name> | ... |

Output

```java
next (s);  // input stream: s
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...```

output stream: t
## Pull API Model

### Input

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;person&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td>Alice</td>
</tr>
<tr>
<td><code>&lt;/name&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td>Bob</td>
</tr>
<tr>
<td><code>&lt;/name&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

### Output

```java
next (s);
while (s matches `<name>`) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
```
Pull API Model

Input:

```
<person> <name> Alice </name> <name> Bob </name> ...
```

Output:

```
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    endElement (t, name);
}
next (s);
...```

Input

```
<person> <name> Alice </name> </person> <name> Bob </name> ...
```

Output

```
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
### Pull API Model

**Input**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>Alice</th>
<th>&lt;/name&gt;</th>
<th>&lt;name&gt;</th>
<th>Bob</th>
<th>&lt;/name&gt;</th>
<th>...</th>
</tr>
</thead>
</table>

**Output**

```java
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
```
Pull API Model

Input:

```
<person> <name> Alice </name> </person> <name> Bob </name> ...
```

Output:

```
<name>
```

```
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
```
Pull API Model

**Input**

| <person> | <name> | Alice | </name> | <name> | Bob | </name> | ... |

**Output**

| <name> | Alice |

```java
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...```

Pull API Model

<table>
<thead>
<tr>
<th>Input</th>
<th>&lt;person&gt;</th>
<th>&lt;name&gt;</th>
<th>Alice</th>
<th>&lt;/name&gt;</th>
<th>&lt;name&gt;</th>
<th>Bob</th>
<th>&lt;/name&gt;</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>&lt;name&gt;</td>
<td>Alice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
  next (s);
  while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    next (s);
    endElement (t, name);
  }
  next (s);
  ...
```
Pull API Model

Input:

| <person> | <name> | Alice    | </name> | <name> | Bob    | </name> | ...

Output:

| <name> | Alice |

```java
next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
```
Pull API Model

Input:

\[
\begin{array}{ccc}
\langle \text{person}\rangle & \langle \text{name}\rangle & \text{Alice} & \langle \text{name}\rangle & \text{Bob} & \langle \text{name}\rangle & \cdots \\
\end{array}
\]

Output:

\[
\begin{array}{ccc}
\langle \text{name}\rangle & \text{Alice} & \langle \text{name}\rangle \\
\end{array}
\]

next (s);

while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    endElement (t, name);
}

next (s);

...
Pull API Model

Input  

\[
\begin{array}{c}
\cdots \\
\langle\text{name}\rangle \\
\langle\text{person}\rangle \\
\text{END\_DOC}
\end{array}
\]

Output  

\[
\begin{array}{c}
\cdots
\end{array}
\]

next (s);
while (s matches <name>) {
    next (s);
    startElement (t, name);
    characters (t, getText (s));
    next (s);
    next (s);
    next (s);
    endElement (t, name);
}
next (s);
...
Pull API Model

Input

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{next (s)}; \\
\text{while (s matches <name>)) \{} \\
\quad \text{next (s)}; \\
\quad \text{startElement (t, name)}; \\
\quad \text{characters (t, getText (s))}; \\
\quad \text{next (s)}; \\
\quad \text{next (s)}; \\
\quad \text{endElement (t, name)}; \\
\}\ \\
\text{next (s)}; \\
\ldots
\]
Pull API Model

Input

\[
\text{\ldots } \quad \langle \text{name} \rangle \quad \langle \text{person} \rangle \quad \text{END\_DOC}
\]

Output

\[
\text{\ldots }
\]

\[
\text{next (s);} \\
\text{while (s matches <name>) { }
\text{next (s);} \\
\text{startElement (t, name);} \\
\text{characters (t, getText (s));}
\text{next (s);} \\
\text{next (s);} \\
\text{endElement (t, name);} \\
\}
\text{next (s);} \\
\ldots
\]
Goals

Find languages to describe the I/O behaviour of programs such that

- A program never fails on a read operation when fed a word in the input language.
- The output of a terminated program is always a word in the output language.

Can tree languages describe the I/O behaviour of streaming processors?
Consider an application firewall that filters SOAP messages:

```xml
<Envelope>
  <Header>
    <AuthHeader> <Password>dDSv$sine93</Password> </AuthHeader>
  </Header>
  <Body>
    <postNames>
      <name>Alice</name> <name>Bob</name> <name>Charlie</name>
    </postNames>
  </Body>
</Envelope>
```

**Constraints:**

- Only the `postNames` operation is permitted. Messages with other operations are discarded in their entirety.
- The operation’s contents are not stored on the firewall.
fun main () {
    next (s);
    let writeHeader = readHeader ();
    next (s);
    if (s matches <postNames>) {
        startElement (t, Envelope);
        writeHeader ();
        startElement (t, Body);
        copyPostNames ();
        next (s);
        next (s);
        endElement (t, Body);
        endElement (t, Envelope);
    }
}

fun readHeader () {
    next (s);
    let passwordTxt = getText (s);
    next (s);
    next (s);
    next (s);
    fun writeHeader () {
        startElement (t, Header);
        startElement (t, Password);
        characters (t, passwordTxt);
        endElement (t, Password);
        endElement (t, Header);
    } return writeHeader;
}

fun copyPostNames () {
    /* copy postNames element */
}
Example: Conflicting Structure

Write `<group>` only when the first child is seen.

Write `</group>` only at a surname or end of the document.
Outline

Programming model and examples

Visibly Pushdown Expressions

Effect assignment

Decidability of subeffecting and Visibly Pushdown Automata

Summary
Visibly Pushdown Expressions (VPEs) provide a convenient syntax for Visibly Pushdown Languages (VPLs) [Alur, Madhusudan 2004].

Matched VPEs describe trees, and are based on regular expression types [Hosoya, Vouillon, Pierce 2000].

Patterns

\[
p ::= \langle l \rangle \ | \ \langle /l \rangle \ | \ \text{string} \ | \ p|p \ | \ p\backslash p \ | \ \sim_c \ | \ \sim_r
\]

Matched VPEs

\[
T ::= \emptyset \ | \ () \ | \ \pi \ | \ p_c[T] \ | \ T,T \ | \ T \ | \ T\&T \ | \ T* \ | \ X
\]

Unmatched VPEs

\[
S ::= T \ | \ p \ | \ S,S \ | \ S \oplus S \ | \ S|S \ | \ S&S \ | \ S*
\]

Example: \(\sim_c, X, \sim\) where \(X = \text{foo}[X]^*\) with abbreviations

\[
\sim = \sim_c \ | \ \sim_r \ | \ \text{string} \qquad \text{foo}[X] = \langle/\text{foo}\rangle[X]
\]

No unguarded recursion.
A VPE $S$ determines a set of words $L(S)$ defined by

\[
\begin{align*}
  a & \text{ matches } p \\
  a & : p \\
  a_c & \text{ matches } p_c \\
  a_c & \text{ matches } p_c \\
  \alpha : T & \text{ matches } p_r \\
  \alpha : T & \text{ matches } p_r \\
  a_c \cdot \alpha \cdot a_r & : \frac{p_c}{p_r}[T] \\
  a_c \cdot \alpha \cdot a_r & : \frac{p_c}{p_r}[T] \\
  \alpha_1 : S_1 & \quad \alpha_2 : S_2 \\
  \alpha_1 \cdot \alpha_2 & : S_1, S_2 \\
  \alpha : S_1 & \quad \alpha : S_2 \\
  \alpha : S_1 & \quad \alpha : S_2 \\
  \alpha_1 \cdot \alpha_2 : S_1 & \quad \alpha_1 \cdot \alpha_2 : S_2 \\
  \alpha : S_1 & \quad \alpha : S_2 \\
  \alpha_1 \cdot \alpha_2 & : S_1 \oplus S_2 \\
  \alpha : S_1 & \quad \alpha : S_2 \\
  \alpha_1 \cdot \alpha_2 & : S_1 \oplus S_2
\end{align*}
\]

Example: $<\text{foo}><\text{foo}></\text{foo}><\text{foo}> \in L(\sim_c, X, \sim)$ where $X = \text{foo}[X]^*$. Elements in recursive matched VPEs can be unwound into start and end tags because $L(\frac{p_c}{p_r}[T]) = L(p_c, T, p_r)$. 
Outline

Programming model and examples

Visibly Pushdown Expressions

Effect assignment

Decidability of subeffecting and Visibly Pushdown Automata

Summary
Effect Assignment for Output

Effects
\[ \varepsilon ::= \emptyset \mid s!S \mid s?S \mid \varepsilon, \varepsilon \]

Judgements
\[ \Gamma \vdash M : \sigma; \varepsilon \]

If \( \Gamma \vdash M : \sigma; \varepsilon, s!S \) then \( M \) writes a word from \( L(S) \) to output stream \( s \) whenever it terminates.

For output:

\[ \Gamma \vdash \text{startElement}(s, l) : \text{unit}; s!<l> \]
\[ \Gamma \vdash \text{endElement}(s, l) : \text{unit}; s!</l> \]
\[ \Gamma \vdash \text{"Smith" : string; } \emptyset \quad \text{\Gamma \vdash M : string; } \varepsilon, s!S \]
\[ \Gamma \vdash \text{characters}(s, M) : \text{unit; } \varepsilon, s!(S, \text{string}) \]
\[ \Gamma \vdash M : \sigma; s!S_1 \quad \Gamma, x : \sigma \vdash N : \tau; s!S_2 \]
\[ \Gamma \vdash \text{let } x = M \text{ in } N : \tau; s!(S_1, S_2) \]

Sequential composition extended for input effects shortly.
Subeffecting based on language inclusion (input subeffecting shortly)

\[
\emptyset <: s!() \quad \frac{L(S_1) \subseteq L(S_2)}{s!S_1 <: s!S_2}
\]

fun f (n) {
    if (n == 0) {
        characters (s, "Smith");
    } else {
        startElement (s, foo);
        f (n - 1);
        endElement (s, foo);
    }
}

Let \( X = \text{string} \mid \text{foo}[X] \).

To assign effect \( s!X \), must show

\[
(\text{string} \mid <\text{foo}>, X, </\text{foo}> ) <: X
\]
Effect Assignment for Input

Record discarded input? Consider

\[
\text{while (s matches string) \{}
\]

\[
\text{\hspace{1.0cm} next (s);
}
\]

Discards string* [latex]
\]

\]

And

\[
\text{if (s matches string) \{}
\]

\[
\text{\hspace{1.0cm} next (s);
}
\]

\[
\text{\hspace{1.0cm} else \{
}
\]

\[
\text{\hspace{2.0cm} fail (s);
}
\]

Discards string [latex]

But the sequential composition always fails, even when the input is in
\[ L(\text{string}^*, \text{string}) \].

The precondition is more than the discarded input.
Effect Assignment for Input

While loop:
- **pre**: string*, ¬string, S₁
- **post**: ¬string, S₁

If-then-else statement:
- **pre**: string, ∼, S₂
- **post**: ∼, S₂

Avoid the need for effect polymorphism with S₁ and S₂ by using string*, ¬string and string, ∼ as the effects.

Convention: the last token has not been discarded, but may have been examined.

Input effects for sequential composition using ⊕:

\[
(string^*, ¬string) ⊕ (string, ∼) = string^*, (¬string ⊕ string), ∼ = 0
\]

If \(\Gamma ⊢ M: σ; ε, s?S\) is started with a word from \(L(S)\) on input stream \(s\) then it will not fail on \(s\).
\[
\Gamma \vdash \text{next}(s) : \text{unit}; s?((\sim, \sim)) \\
\Gamma \vdash \text{getText}(s) : \text{string}; s?\text{string} \\
\Gamma \vdash \text{fail}(s) : \text{unit}; \varepsilon, s?\emptyset \\
\Gamma \vdash \text{assert}(s, p) : \text{unit}; s?p \\
\Gamma \vdash M : \sigma; \varepsilon, s?S_1 \\
\Gamma \vdash N : \sigma; \varepsilon, s?S_2 \\
\Gamma \vdash \text{if}(s \text{ matches } p) \text{ then } M \text{ else } N : \sigma; \varepsilon, s?((p \oplus S_1) | (\neg p \oplus S_2)) \\
\Gamma \vdash M : \text{unit}; s?S_1, t!S_2 \\
S_3 <: (p \oplus S_1 \oplus S_3) | \neg p \\
\Gamma \vdash \text{while}(s \text{ matches } p) \ M : \sigma; \varepsilon, s?S_3, t!S_2^* \\
\Gamma \vdash M : \sigma; \varepsilon_1 \\
\Gamma, x : \sigma \vdash N : \tau; \varepsilon_2 \\
\Gamma \vdash \text{let } x = M \text{ in } N : \tau; \varepsilon_1 \odot \varepsilon_2
\]

where

\[
\emptyset \odot \emptyset = \emptyset \\
(\varepsilon_1, s?S_1) \odot (\varepsilon_2, s?S_2) = (\varepsilon_1 \odot \varepsilon_2), s?(S_1 \oplus S_2) \\
(\varepsilon_1, s!S_1) \odot (\varepsilon_2, s!S_2) = (\varepsilon_1 \odot \varepsilon_2), s!(S_1, S_2)
\]
Subeffecting Revisited

Input subeffecting

\[
\begin{align*}
\emptyset & <: s!(()) \quad \emptyset <: s?\sim
\end{align*}
\]

\[
\frac{L(S_1) \subseteq L(S_2)}{s!S_1 <: s!S_2}
\]

\[
\frac{L(S_2) \subseteq L(S_1)}{s?S_1 <: s?S_2}
\]

fun f () {
    if (s matches <~>) {
        next (s);
        f ();
        next (s);
        f ();
    } else {
    }
}

Let \( X = \sim_c[X]^* \).

To assign effect \( s?(X, \sim_c) \), must show

\[
(X, \sim_c) <: (\sim_c \oplus (\sim, \sim) \oplus (X, \sim_c) \oplus (\sim, \sim) \oplus (X, \sim_c) \mid \sim_c \oplus \sim)
\]
Outline

Programming model and examples

Visibly Pushdown Expressions

Effect assignment

Decidability of subeffecting and Visibly Pushdown Automata

Summary
Decidability of Subeffecting

Must decide language inclusion between VPEs.

Visibly Pushdown Languages (VPL) [Alur, Madhusudan 2004]: for analysis of pushdown systems with stack restrictions, e.g., procedures.
Decidability of Subeffecting

Must decide language inclusion between VPEs.

Visibly Pushdown Languages (VPL) [Alur, Madhusudan 2004]: for analysis of pushdown systems with stack restrictions, e.g., procedures.

Visibly Pushdown Automata (VPA):

- alphabet partitioned as calls $\Sigma_c$, returns $\Sigma_r$, and internal $\Sigma_{int}$
- distinguished bottom-of-stack symbol $\perp$
- call transitions $(q_1, a, q_2, \gamma)$ push one stack symbol ($a \in \Sigma_c, \gamma \neq \perp$)
- return transitions $(q_1, a, \gamma, q_2)$ pop one stack symbol ($a \in \Sigma_r, \gamma \neq \perp$) or operate on empty stack ($q_1, a, \perp, q_2$)
- internal transitions $(q_1, a, q_2)$ ignore the stack ($a \in \Sigma_{int}$)
- final state acceptance condition—not empty stack condition.
Example: Concatenation

VPA for `<foo>`
- initial state $q_1$ and final state $q'_1$
- call transition $(q_1, <foo>, q'_1, \gamma_1)$

VPA for `</bar>`
- initial state $q_2$ and final state $q'_2$
- return transition $(q_2, </bar>, \bot, q'_2)$

To concatenate, $\bot$-transitions on the RHS accept stack symbols from the LHS. VPA for `<foo></bar>`
- initial state $q_3$ and final state $q''_3$
- call transition $(q_3, <foo>, q'_3, \gamma_1)$
- return transition $(q'_3, </bar>, \gamma_1, q''_3)$
Example: Nested Recursion

Let \( x = \text{foo}[x] \star \)

VPA for \((x \mid <\text{foo}>)^\star\)
- initial and final state \(q_1\)
- call \((q_1, <\text{foo}>, q_1, \gamma_1)\)
- return \((q_1, </\text{foo}>, \gamma_1, q_1)\)

VPA for \(x\)
- initial and final state \(q_2\)
- call \((q_2, <\text{foo}>, q_2', \gamma_2)\)
- call \((q_2', <\text{foo}>, q_2, \gamma_2)\)
- return \((q_2', </\text{foo}>, \gamma_2, q_2)\)
- return \((q_2', </\text{foo}>, \gamma_2', q_2)\)
Class of VPA closed under:

- intersection (unlike deterministic and non-deterministic context-free languages)
- complement (unlike non-deterministic context-free languages)

Emptiness is decidable for VPA.

Inclusion between VPA $V_1$ and $V_2$ is decidable by testing:

$$L(V_1) \cap L(V_2)^c = \emptyset$$
Translating a VPE to a VPA:

- First ensure that elements and variables only appear as $a_c[X]$.

- Define a monadic second order logic with matching ($\text{MSO}_\mu$) formula $\psi_S$ for each VPE $S$ such that

$$\forall \alpha. \alpha : S \leftrightarrow \alpha \models \psi_S$$

- $\text{MSO}_\mu$ formulae can be translated to VPA [Alur, Madhusudan].
Some cases from the translation to $\text{MSO}_\mu$ formulae:

\[
\begin{align*}
\llbracket a \rrbracket_{x,y} & \overset{\text{def}}{=} x = y \land x \in Q_a \\
\llbracket S_1, S_2 \rrbracket_{x,y} & \overset{\text{def}}{=} \left( \exists z. x \leq z < y \land \llbracket S_1 \rrbracket_{x,z} \land \llbracket S_2 \rrbracket_{z+1,y} \right) \lor \ldots \\
\llbracket S_1 \oplus S_2 \rrbracket_{x,y} & \overset{\text{def}}{=} \exists z. x \leq z \leq y \land \llbracket S_1 \rrbracket_{x,z} \land \llbracket S_2 \rrbracket_{z,y} \\
\llbracket S_1 | S_2 \rrbracket_{x,y} & \overset{\text{def}}{=} \llbracket S_1 \rrbracket_{x,y} \lor \llbracket S_2 \rrbracket_{x,y} \\
\llbracket S_1 \& S_2 \rrbracket_{x,y} & \overset{\text{def}}{=} \llbracket S_1 \rrbracket_{x,y} \land \llbracket S_2 \rrbracket_{x,y} \\
\llbracket a_{c[X]} \rrbracket_{x,y} & \overset{\text{def}}{=} \mu(x,y) \land x \in \text{positions for strings matching } a_{c[X]}[E(X)]
\end{align*}
\]

To handle recursive VPE variables, use existential quantifiers to assume sets of initial positions that match $a_{c[X]}[E(X)]$, for each variable $X$ and its binding $E(X)$.

The matching operator $\mu$ gives the unique corresponding endpoint.
Summary

Type and effect system for a functional language based on existing “pull” APIs such as XMLPull and the Streaming API for XML (StAX) ensures that programs will not fail on read operations and will produce correct output (see paper for subject reduction theorem).
Summary

Type and effect system for a functional language based on existing “pull” APIs such as XMLPull and the Streaming API for XML (StAX) ensures that programs will not fail on read operations and will produce correct output (see paper for subject reduction theorem).

Sequential composition of input effects requires information about lookahead tokens.
Summary

Type and effect system for a functional language based on existing “pull” APIs such as XMLPull and the Streaming API for XML (StAX) ensures that programs will not fail on read operations and will produce correct output (see paper for subject reduction theorem).

Sequential composition of input effects requires information about lookahead tokens.

Visibly Pushdown Expressions can express both individual start/end tags and trees.
Type and effect system for a functional language based on existing “pull” APIs such as XMLPull and the Streaming API for XML (StAX) ensures that programs will not fail on read operations and will produce correct output (see paper for subject reduction theorem).

Sequential composition of input effects requires information about lookahead tokens.

Visibly Pushdown Expressions can express both individual start/end tags and trees.

Visibly Pushdown Expressions a convenient surface syntax for Alur and Madhusudan’s VPLs, with the same expressive power and decidable inclusion.
Plans

In progress

- Practical implementation of VPE inclusion testing
- Integration with Java or C#

Future work

- Parametric effect polymorphism
- Effect inference
Thank You!