Streams that Compose using Macros that Oblige

Martin Hirzel    Bugra Gedik
IBM Research

PEPM 2012
Stream Processing

- Phone calls
- Heart beats
- GPS location
- Stock quotes

Continuous input

Stream graph (in SPL)
- IBM InfoSphere Streams
- Cluster of commodity workstations

Continuous output
- Alerts
- Actions
- Logs

- Long-running applications
- High-throughput, low-latency
- Library of reusable stream operators
- Inherent parallelism
Stream Graphs in SPL

Stream graph

Operator (stream transformer)

Parameter (operator configuration)

Stream (infinite sequence of data items)

SPL code*

(p) <- parse(i);
(m) <- match{pattern}(p);
(a) <- alert(m);

*simplified: no types
Composite Operators

**SPL with composites**
(p) <- parse(i);
(a) <- ma{pattern}(p);
composite (y) <- ma{pat}(x) {
  (m) <- match{pat}(x);
  (y) <- alert(m);
}

**Flat SPL**
(p) <- parse(i);
(m) <- match{pattern}(p);
(a) <- alert(m);

Motivation:
• Modularize large programs
• Reuse common subgraphs

Expand (in compiler)
Expansion via IL

**SPL with composites**

**Flat SPL**

- **Translate**
  - Intermediate language
    - Functional language (simple, to make interpreter easy to implement)
    - Macros containing SPL (opaque, to separate concerns)

- **Evaluate**
  - Unlike traditional compilers (translate)
**SPL with composites**

- \( (p) \leftarrow \text{parse}(i); \)
- \( (a) \leftarrow \text{ma}\{\text{pattern}\}(p); \)
- \( \text{composite} \ (y) \leftarrow \text{ma}\{\text{pat}\}(x) \{
  \quad (m) \leftarrow \text{match}\{\text{pat}\}(x);
  \quad (y) \leftarrow \text{alert}(m);
\} \)

**Flat SPL**

- \( (p) \leftarrow \text{parse}(i); \)
- \( (m) \leftarrow \text{match}\{\text{pattern}\}(p); \)
- \( (a) \leftarrow \text{alert}(m); \)

**Intermediate language**

\[
\text{main} = \text{fn}(a, i) \Rightarrow
\begin{array}{l}
  \ `( (p) \leftarrow \text{parse}(\%i); \\
  \quad (a) \leftarrow \text{ma}(a, \text{pattern}, p) \ )); \\
  \text{ma} = \text{fn}(y, \text{pat}, x) \Rightarrow \\
  \quad `( (m) \leftarrow \text{match}(\%\text{pat})(\%x); \\
  \quad (y) \leftarrow \text{alert}(m); ) ;
\end{array}
\]

**Translate**

**Quote (do not evaluate)**

**Escape (force evaluation)**

**Evaluate**
**Hygiene Problem**

**SPL with composites**

\[
\begin{align*}
(p) & \leftarrow \text{parse}(i); \\
(a) & \leftarrow \text{ma}\{\text{pattern}\}(p); \\
\text{composite } (y) & \leftarrow \text{ma}\{\text{pat}\}(x) \{ \\
& \hspace{1em} (p) <\leftarrow \text{match}\{\text{pat}\}(x); \\
& \hspace{1em} (y) <\leftarrow \text{alert}(p); \\
\}
\end{align*}
\]

**Intermediate language**

\[
\text{main} = \text{fn}(a, i) => \\
\text{`( (p) } <\leftarrow \text{parse}(\%i); \\
\text{(%( ma}(a, \text{pattern}, p)))); \\
\text{ma} = \text{fn}(y, \text{pat}, x) => \\
\text{`( (p) } <\leftarrow \text{match}\{\%\text{pat}\}(\%x); \\
\text{(%y) } <\leftarrow \text{alert}(p); \); \\
\]

**Flat SPL**

\[
\begin{align*}
(p) & \leftarrow \text{parse}(i); \\
(p) & \leftarrow \text{match}\{\text{pattern}\}(p); \\
(a) & \leftarrow \text{alert}(p); \\
\end{align*}
\]

**Accidental name capture**
Hygiene Solution

**SPL with composites**

\[
\begin{align*}
(p) & \leftarrow \text{parse}(i); \\
(a) & \leftarrow \text{ma}\{\text{pattern}\}(p); \\
\text{composite} \ (y) & \leftarrow \text{ma}\{\text{pat}\}(x) \{ \\
& \hspace{1em} (p) \leftarrow \text{match}\{\text{pat}\}(x); \\
& \hspace{1em} (y) \leftarrow \text{alert}(p); \\
\}
\end{align*}
\]

**Flat SPL**

\[
\begin{align*}
(p) & \leftarrow \text{parse}(i); \\
(p_0) & \leftarrow \text{match}\{\text{pattern}\}(p); \\
(a) & \leftarrow \text{alert}(p_0);
\end{align*}
\]

**Intermediate language**

\[
\begin{align*}
\text{main} & = \text{fn}(a, i) => \\
& \hspace{1em} \`\ ( (p) \leftarrow \text{parse}(\%i); \\
& \hspace{2em} \% ( \text{ma}(a, \text{pattern}, p) )); \\
\text{ma} & = \text{fn}(y, \text{pat}, x) => \\
& \hspace{1em} \text{let} \ p = \text{freshId()} \text{ in} \\
& \hspace{2em} \` ( (\%p) \leftarrow \text{match}\{\%pat\}(\%x); \\
& \hspace{3em} (\%y) \leftarrow \text{alert}(\%p); );
\end{align*}
\]

Translate

Generate calls to freshId()

Evaluate

Pre-pass to find names to avoid
Higher-Order Composites

What: Composite operator that takes other operators as parameters
Why: Reuse common graph “structures”
Higher-Order Example

**SPL with composites**
(p) <- parse(i);
(a) <- ma{myMatch}(p);
composite (y) <- myMatch(x) {
    (y) <- match{pattern}(x);
}
composite (y)<-ma{matcher}(x) {
    (m) <- matcher(x);
    (y) <- alert(m);
}

**Flat SPL**
(p) <- parse(i);
(m) <- match{pattern}(p);
(a) <- alert(m);

---

**Intermediate language**
main = fn(a, i) =>
`( (p) <- parse(%i);
  %( ma(a, myMatch, p ) ));
``
ma = fn(y, matcher, x) =>
let m = freshId() in
%( matcher(m, x))
%(y) <- alert(%m); );

---

Use higher-order functions
Contracts Motivation

**SPL with composites**

```plaintext
(p) ← parse(i);
(a) ← ma{“((a|b)*)”}(p);
composite (y) ← ma{pat}(x) {
    (m) ← match{pat}(x);
    (y) ← alert(m);
}
```

**Intermediate language**

```plaintext
main = fn(a, i) =>
    `( (p) ← parse(%i);
        % ( ma(a, “((a|b)*)”, p ) ));
    ma = fn(y, pat, x) =>
        let m = freshId() in
        `( (%m) ← match{%pat}(%x);
            (%y) ← alert(%m); );
```

**Flat SPL**

```plaintext
(p) ← parse(i);
(m) ← match{“((a|b)*)”}(p);
(a) ← alert(m);
```

*Cause: missing ‘)’*

*Effect: error in expanded code*
Contracts Solution

*SPL with Composites and Contracts*

composite (y) <- ma{pattern @ regexp}(x) {...}

**Translate**

*IL with Contracts*

ma = fn(y, pattern @ regexp, x) =>...

**Rewrite**

*IL with Obligations*

ma = oblige(fn(y, pattern, x) =>..., regexp₁)

**Evaluate**

*Flat SPL*

or

*Error report with “blame”*

• Adaptation of Findler/Felleisen, ICFP 2002

• In the general case, the contracts can be higher-order too

Caller violated precondition
## Experiences

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composites</td>
<td>• Used widely</td>
</tr>
<tr>
<td></td>
<td>• Key feature in new product release</td>
</tr>
<tr>
<td>Hygiene</td>
<td>• Essential</td>
</tr>
<tr>
<td></td>
<td>• Also, need to “map back” for debugging</td>
</tr>
<tr>
<td>Higher-order composites</td>
<td>• Not used very often</td>
</tr>
<tr>
<td></td>
<td>• When used, lead to very concise code</td>
</tr>
<tr>
<td>Contracts</td>
<td>• Very powerful for primitive operators</td>
</tr>
<tr>
<td></td>
<td>• For composite operators: structural only</td>
</tr>
</tbody>
</table>

**Telco benchmark:**
- 7,029 LOC (4,350 toolkit + 2,679 in 39 apps)
- After expansion, 316,725 LOC (factor 45x)
Related Work

Stream composites
• LabView, StreamIt, EventFlow: first-order user-defined composites
• DryadLINQ, FlumeJava: object-oriented with streaming extension
• This talk: higher-order, user-defined composites with contracts

Macros
• Kohlbecker et al.; Clinger/Rees: hygiene

Contracts
• Findler/Felleisen: contracts for higher-order functions

This talk
• Combine macros + contracts
• Use macros for a new programming language
Conclusions

Contributions in two areas:

Stream processing: Composite operators

Macros: Contracts