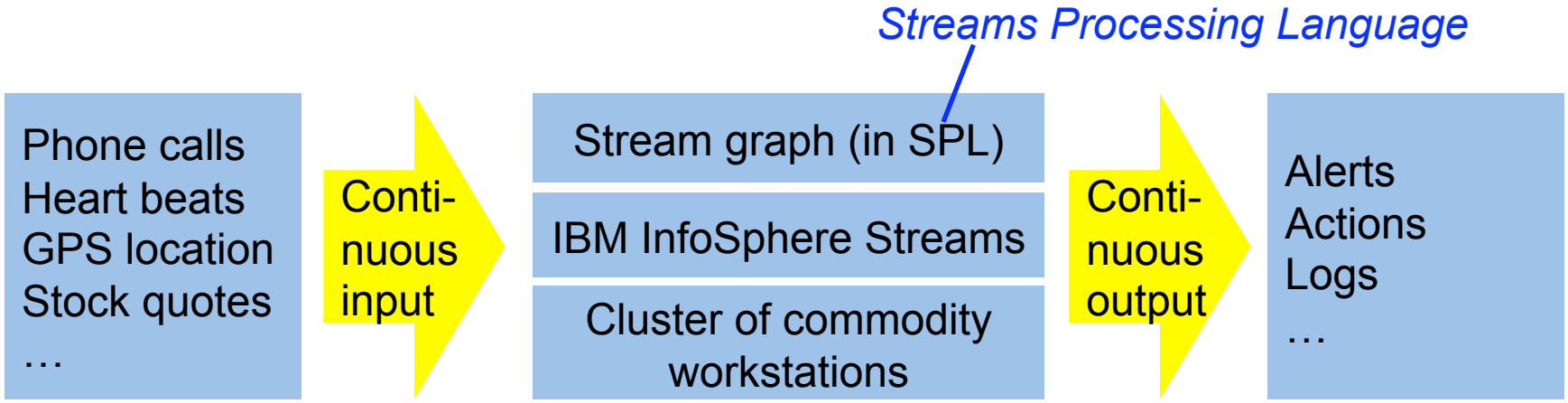
Streams that Compose using Macros that Oblige

Martin Hirzel Bugra Gedik

IBM Research

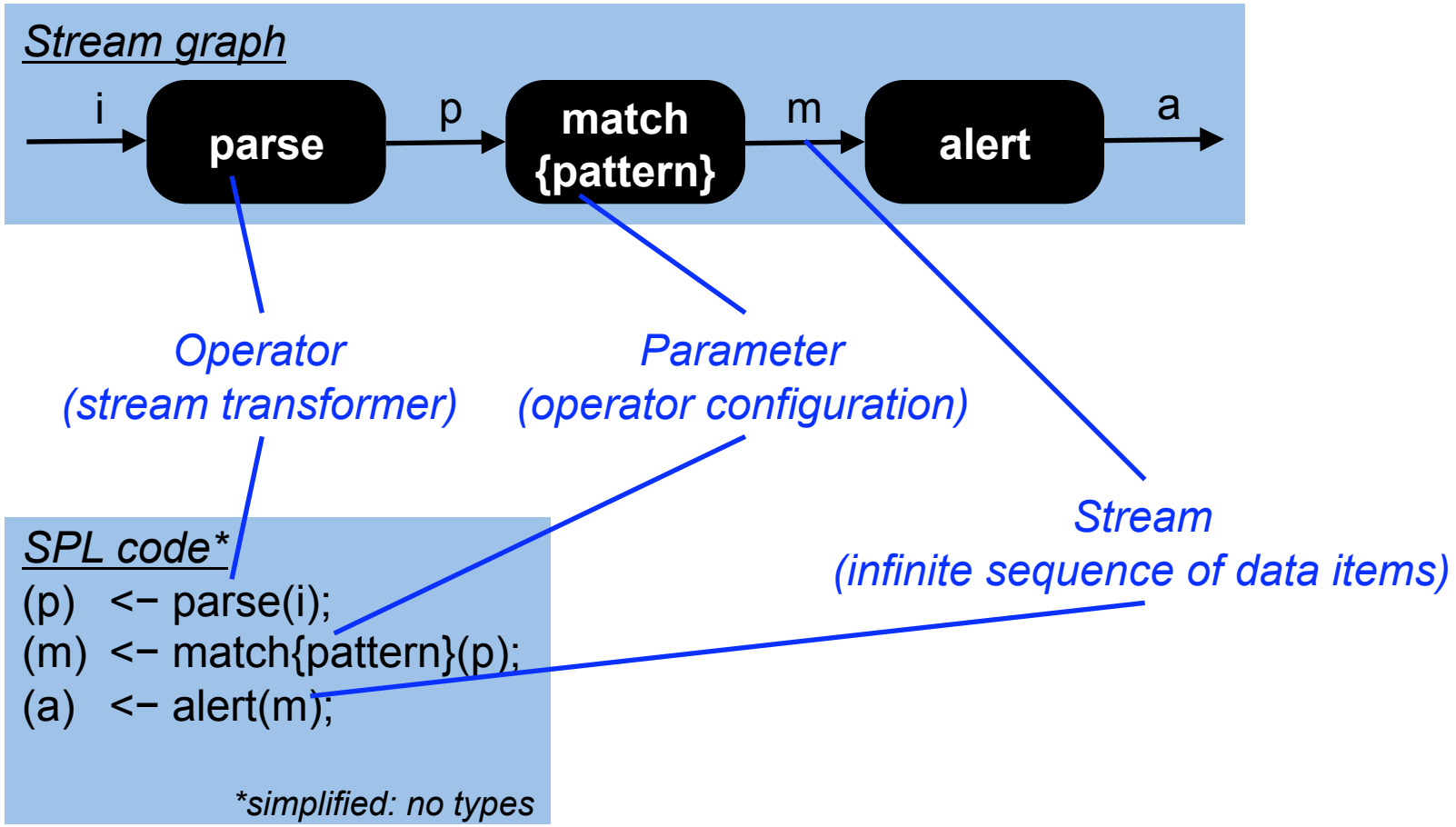
PEPM 2012

Stream Processing



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

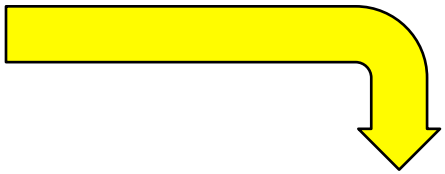
Stream Graphs in SPL



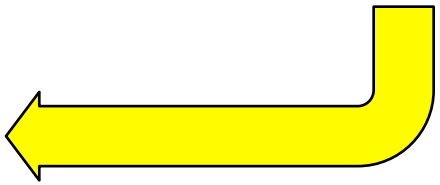
Composite Operators

SPL with composites

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



Expand
(in compiler)



Flat SPL

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

Motivation:

- **Modularize large programs**
- **Reuse common subgraphs**

Expansion via IL

SPL with composites

Translate

Intermediate language

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

Flat SPL

Evaluate

Unlike traditional compilers (translate)

IL Example

SPL with composites

```
(p) <− parse(i);
(a) <− ma{pattern}(p);
composite (y) <− ma{pat}(x) {
   (m) <− match{pat}(x);
   (y)  <− alert(m);
}
```

Translate

Intermediate language

```
main = fn(a, i) =>
 `( (p) <− parse(%i);
    %( ma(a, pattern, p) ));
ma = fn(y, pat, x) =>
 `( (m)  <− match{%pat}(%x);
    (%y) <− alert(m); );
```

Quote (do not evaluate)

Escape (force evaluation)

Flat SPL

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

Evaluate

Hygiene Problem

Accidental name capture

SPL with composites

```
(p) <− parse(i);
(a) <− ma{pattern}(p);
composite (y) <− ma{pat}(x) {
   (p)  <− match{pat}(x);
   (y)  <− alert(p);
}
```

Translate

Intermediate language

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

Evaluate

Flat SPL

```
(p)   <− parse(i);
(p)   <− match{pattern}(p);
(a)   <− alert(p);
```

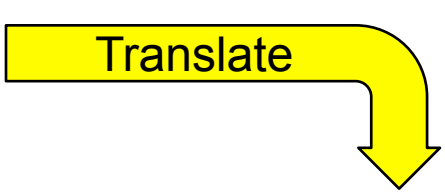
Hygiene Solution

SPL with composites

```
(p) <− parse(i);
(a) <− ma{pattern}(p);
composite (y) <− ma{pat}(x) {
   (p)  <− match{pat}(x);
   (y)  <− alert(p);
}
```

Flat SPL

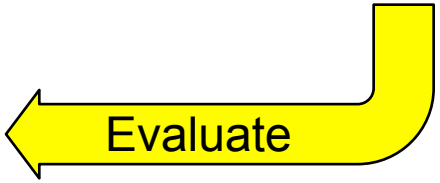
```
(p)   <− parse(i);
(p0)  <− match{pattern}(p);
(a)   <− alert(p0);
```



Generate calls to freshId()

Intermediate language

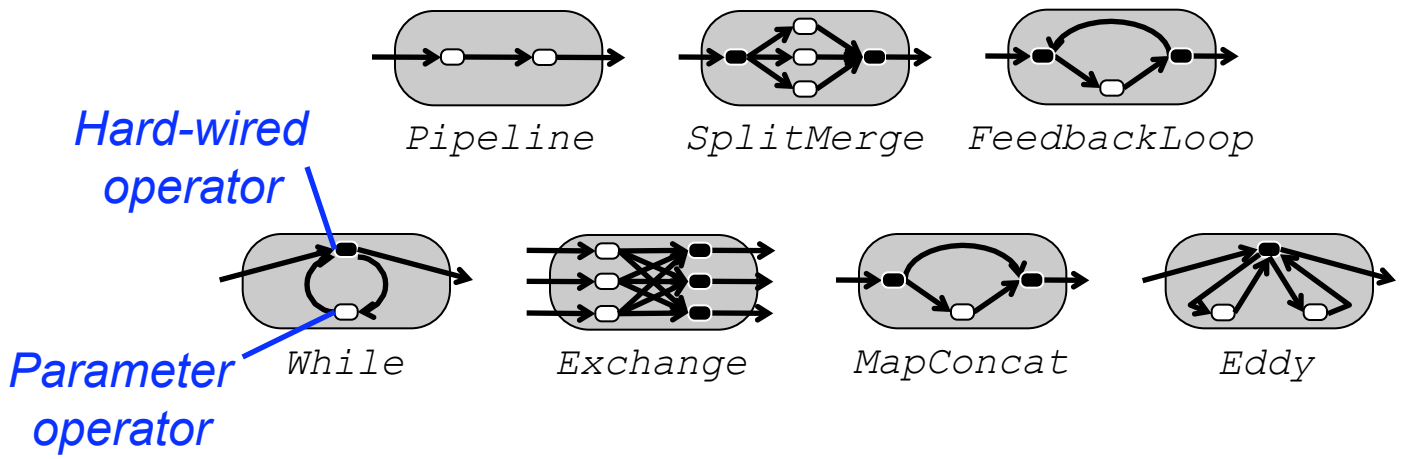
```
main = fn(a, i) =>
 `( (p) <− parse(%i);
    %( ma(a, pattern, p) ));
ma = fn(y, pat, x) =>
 let p = freshId() in
 `( (%p) <− match{%pat}(%x);
    (%y) <− alert(%p); );
```



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

Use higher-order functions

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);
```

Translate

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); );
```

Evaluate

Contracts Motivation

Cause: missing ‘)’

SPL with composites

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

Translate

Intermediate language

```
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); );
```

Evaluate

Flat SPL

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

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) =>…, $regexp_1$)

Evaluate

Flat SPL

or

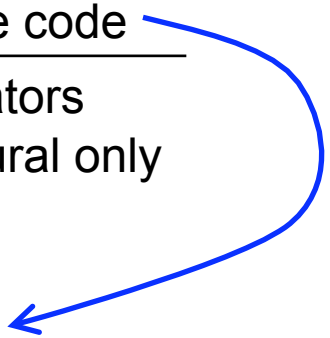
Error report with "blame"

Caller violated precondition

- ***Adaptation of Findler/Felleisen, ICFP 2002***
- ***In the general case, the contracts can be higher-order too***

Experiences

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



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