Streams that Compose using Macros that Oblige

Martin Hirzel Buğra Gedik
IBM Watson Research Center
{hirzel,bgedik}@us.ibm.com

Abstract

Since the end of frequency scaling, the programming languages community has started to embrace multi-core and even distributed systems. One paradigm that lends itself well to distribution is stream processing. In stream processing, an application consists of a directed graph of streams and operators, where streams are infinite sequences of data items, and operators fire in infinite loops to process data. This model directly exposes parallelism, requires no shared memory, and is a good match for several emerging application domains. Unfortunately, streaming languages have so far been lacking in abstraction. This paper introduces higher-order composite operators, which encapsulate stream subgraphs, and contracts, which specify pre- and post-conditions for composites. Composites are expanded at compile time, in a manner similar to macros. Their contractual obligations are also checked at compile-time. We build on existing work on macros and contracts to implement higher-order composites. The user-visible language features provide a consistent look-and-feel for the streaming language, whereas the underlying implementation provides high-quality static error messages and prevents accidental name capture.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Performance, Reliability

1. Introduction

The research for this paper was triggered by the requirement to make it easier to reuse SPL code [10]. SPL is the streaming language for IBM's InfoSphere Streams platform. An SPL program describes a stream graph in which directed edges are streams of data, and vertices are operators that transform streams. Operators run continuously, and since they are parallel and communicate only via streams, the system can distribute them on a shared-nothing cluster for scaling. SPL and its platform have been applied in diverse domains, including but not limited to algorithmic trading, telecommunications, traffic monitoring, health care, and large-scale data analysis. As users started to build more and more sophisticated applications, SPL needed better abstraction features.

A *composite* operator encapsulates a stream subgraph. It helps avoid repetitive code, because the subgraph is written only once, but gets expanded in each place the composite is used. We wanted to add *higher-order* composites to SPL, meaning they can take other operators, including other composites, as parameters. The benefit of a composite that takes other operators as parameters is that it abstracts over the shape of a subgraph with details to be filled in. And the benefit of permitting any operators, including composites, as parameters is generality and orthogonality.

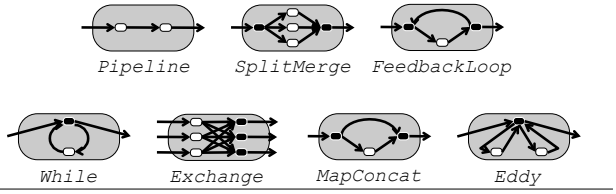


Figure 1. Example higher-order composite stream processing operators. White vertices indicate parameter operators.

Figure 1 shows several common higher-order composites. `Pipeline`, `SplitMerge`, and `FeedbackLoop` are built into the StreamIt language [20]; in SPL, they are not built-in, but can be user-defined. `While` helps express iterative algorithms such as PageRank [15]; `Exchange` reshuffles data between parallel segments [8]; `MapConcat` is useful for translating nested-relational algebra [16]; and `Eddy` encapsulates an online feedback-directed optimization [2]. SPL is the first streaming language that allows users to define these and more higher-order composites. Our design had the following objectives:

- Consistent look-and-feel. Using a composite operator should feel just like using a primitive operator. Furthermore, the definition of a composite operator should follow the same graph-of-operators model as the rest of the language.
- Hygiene [13]. Composite operator expansion should not lead to accidental name capture.
- Good error messages. If there are errors during expansion, their messages should be concise and understandable based on the signature, not the implementation, of the composite.

Much like macros, stream graphs are expanded at compile time. The literature on macros has established solutions for hygiene, for instance, *macros that work* [4]. And the literature on higher-order functions has established solutions for obtaining good error messages via contracts, in other words, preconditions and postconditions [7]. Intuitively, we felt that these techniques should help satisfy the requirements of look-and-feel, hygiene, and good error messages for composites. However, macros and higher-order contracts have not been combined with each other in prior published literature. Hence, the research questions were: how can we adapt established techniques for macros and contracts such that they can be combined? And how can we then use macros and contracts to expand composite operators in SPL?

The main challenges were supporting an intuitive syntax, and dividing the responsibility for each of the objectives between

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

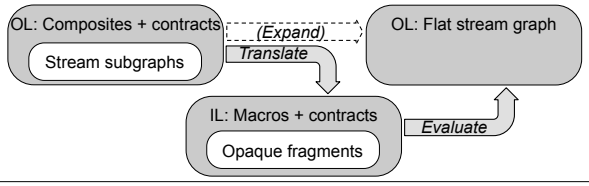


Figure 2. Compiler steps for Streams that Compose.

the compiler components, such that each component has just the knowledge it needs. Figure 2 outlines our solution. We refer to SPL as the object-language (OL). The goal is to expand a program with composites that encapsulate subgraphs into a flat OL stream graph. We accomplish that in two steps: the first step, which is OL-dependent, translates from SPL to an intermediate language (IL). The IL is a simple functional language that supports macros with contracts. Macros encapsulate fragments of OL code; we set things up such that the fragments are opaque, in other words, the interpreter for the IL need not know their meaning. The second step, which is OL-independent, evaluates the IL code with the interpreter and yields the expanded flat OL stream graph. Separating the two steps helped us obtain a cleaner implementation, and also a cleaner description in this paper so readers can separately understand the components. The contributions of this paper are:

- The design of higher-order composite operators for the SPL stream processing language.
- Compile-time contracts for composites in stream processing.
- An approach for implementing composites by translation to an intermediate language with macros and contracts.

We have implemented the SPL language in our production compiler. The customer response has been extremely encouraging. This paper describes both the design and the implementation of composites, which make it easier for SPL users to build distributed stream processing applications. While this paper uses macros that oblige to implement a streaming language, it has a broader impact. Since higher-order macros are effective at avoiding repetitive code, they are widely used in practice, including the C preprocessor (the popular Boost library provides higher-order macros) and C++ templates (which are effectively a higher-order macro-like language feature). This paper shows a way to make higher-order macros more robust.

2. Macros with Contracts

This section describes our intermediate language, which adds contracts to higher-order macros. The advantage of macros is that they extend their base language and avoid repetitive code, but the drawback is that error messages can be hard to understand, because they refer to implementation details of the macro. Contracts fix that, since contract error messages refer to the pre- and post-conditions, not the implementation, of a macro. No background on stream processing is needed to understand this section.

2.1 Intermediate Language

The design of the intermediate language (IL) is shaped by the following goals: first, it obviously needs to support both macros and contracts. The macros need to be higher-order, and they need to support object-language (OL) hygiene. At the same time, the IL should be independent of the OL: while the IL can express macros that generate OL code, the IL semantics should not be intertwined with the OL. As Figure 2 shows, the IL is the target of a translator from OL, and the source of an interpreter that evaluates IL. Therefore, it must strike the right balance between expressiveness (to make the translator easy to write) and simplicity (to make the evaluator easy to write).

expr ::= *primExpr* | *prefixExpr* | *infixExpr* | *listExpr* | *indexExpr*
| *recordExpr* | *attribExpr* | *ifExpr* | *letExpr* | *callExpr*
| *fnExpr* | *quoteExpr*

The start symbol of the IL grammar is the expression. Before elaborating on the individual expression kinds, we need some overview information. The IL manipulates seven kinds of values: booleans, integers, strings, lists, records, closures, and fragments. The last two are the most interesting: a closure is a function value, created by *fnExpr* and called by *callExpr*; and a fragment is an opaque piece of OL code, created by *quoteExpr* but never evaluated by the IL interpreter. A macro is a function that returns a fragment. There is no separate kind of value or expression for contracts; as we shall see, contracts are syntactically part of *fnExpr*, and their values are composed of normal records and functions. An intrinsic function can be called like any other function, but is implemented as a special case by the interpreter. Intrinsics access internals of the interpreter to implement functionality that cannot be expressed otherwise, such as finding the line number of the caller on the call-stack.

2.1.1 Core IL Features

The core expressions of the IL are unsurprising and are listed here for completeness. For each expression, we first show the syntax rules, followed by a description paragraph.

primExpr ::= *ID* | *LITERAL* | '`(`' *expr* '`)`'

A primitive expression can be an identifier like `x`; a literal like **`true`**, `42`, or `"Hello"`; or a parenthesized expression like `(x+y)`.

prefixExpr ::= '`!`' *expr* | '`-`' *expr*

A prefix expression can be a logical negation like `!isDone`; or an arithmetic negation like `-(x+y)`.

infixExpr ::= *expr infixOp expr*

An infix expression like `isDone || (x+y == 0)` consists of two expressions connected by an infix operator, including C-style arithmetic, logic, and comparison operators with their usual precedence.

listExpr ::= '`[`' *expr** '`]`'

A list expression constructs a list value from zero or more comma-separated elements, like `[42, x+y, -1]`. Lists can contain any kinds of values, including other lists, records, closures, or fragments. Lists are implemented as arrays.

indexExpr ::= *expr* '`[`' *expr* '`]`'

An index expression accesses a list element, like `a[2]`. Indexing is 0-based: if `len(a)==3`, then `a` consists of `a[0]`, `a[1]`, and `a[2]`.

recordExpr ::= '`{`' (*ID* '`=`' *expr*)$^+$ '`}`'

A record expression constructs a record value from one or more comma-separate attributes, like `{x=1, y=2}`. Records are similar to structs in C. Attributes can contain any values, including other records, lists, closures, or fragments.

attribExpr ::= *expr* '`.`' *ID*

An attribute expression accesses a record attribute, like `r.x`.

ifExpr ::= '**`if`**' *expr* '**`then`**' *expr* '**`else`**' *expr*

An if expression like **`if`** `x<y` **`then`** `x` **`else`** `y` first evaluates the condition, and, depending on the result, returns either the value of the then-expression or the else-expression. If-expressions are similar to the `?:` operator in C.

letExpr ::= '**`let`**' (*ID* '`=`' *expr*)$^+$ '**`in`**' *expr*

A let expression creates a scope with local variable bindings, like **`let`** `x=1, y=x+1` **`in`** `y+1`. Let-expressions are similar to letrec-expressions in Scheme: they permit recursion in case the local variables are bound to closures.

callExpr ::= *expr* '`(`' *expr** '`)`'

A call expression invokes a closure with zero or more comma-separated actual parameter values, like `gcd(9, 6)`.

2.1.2 Advanced IL Features

Besides the core features, the IL provides function expressions with contracts, and quote expressions with escapes.

fnExpr ::= '**fn**' '(' *ID** ')' ('@' *expr*)? '=>' *expr*

A function expression constructs an anonymous closure, such as `fn(x, y) @ c => x / y`. Section 2.3 will describe what the optional contract, such as `@ c` in the example, means. Closures can be recursive.

quoteExpr ::= '`' *fragment*
escape ::= '%' *primExpr*

A quote expression constructs an opaque fragment of OL code, like `` `{int %old=%x; bar(%x); %x=%old;} ``, that may contain escaped IL expressions, like `%old`. The quote `` ` `` and escape `%` operators are similar to quasi-quote and comma in Lisp: the quote prevents a fragment from being evaluated, and the escape forces an expression to be evaluated, locally reversing the effect of quote.

A compiler that uses the IL as shown in Figure 2 uses the OL parser to turn OL fragments into abstract syntax trees (ASTs) annotated with source locations. When the IL interpreter encounters a quote, it walks the AST of the fragment to find escaped expressions, evaluates those expressions, and grafts the resulting OL fragments into a copy of the AST.

2.2 Hygiene Support in the IL

Rather than fully automating hygiene in the IL interpreter, we provide sufficient IL features for implementing OL hygiene. Hygiene means avoiding accidental name capture and is an essential quality for macro systems. Hygienic macros are macros that obey HC/ME, the hygiene condition for macro expansion: "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated in the same transcription step" [13]. The C preprocessor does not support hygiene, and thus, the following example represents a typical bug in C [6]:

```
1 #define foo(x) {int old=x; bar(x); x=old;}
2 int old = Methuselah();
3 foo(old);
```

Line 3 passes `old` as an actual parameter to `foo`. The programmer expects `old` to be bound to the declaration from Line 2. But the problem is that in Line 1, after substituting `old` for `x`, it will get captured by the local declaration of `old` instead. Thus, the statement `x=old` fails to restore the value of the variable from Line 2, and instead only pointlessly re-assigns the variable from Line 1. In our IL, we would fix the example as follows:

```
let foo = fn(x)=>
  let old = freshId()
  in `{int %old=%x; bar(%x); %x=%old;}
in `{ int old = Methuselah();
      %(foo(`old)) }
```

Each call to the `freshId` intrinsic generates a fresh identifier. The IL interpreter does a pre-pass before evaluation, which gathers the set of all OL identifiers in all code fragments, and initializes the state for `freshId` with that set. Later during evaluation, each `freshId` call returns an identifier that is not already in its state, and records it in the state for the next call. It is up to the OL translator to generate the correct calls to `freshId`, since this requires understanding the OL scoping rules. In general, `freshId` must be called for each binding instance of a variable, such as `int old=x`. Specifically, we will see an example for an OL translator when we discuss the implementation of our streaming language.

2.3 Contracts in the IL

Figure 3 shows an example with a higher-order contract. There are two kinds of contracts. A *flat* contract, such as `fn(x)=> true` in the example, is just a boolean function on a flat value (not a closure). A *non-flat* contract applies to a closure. It is a record with three attributes **param**, **pre**, and **post**, such as `cCmp` or `cFind` in the example. Attribute **param** holds a list of (flat or non-flat)

```
1  let
2    cCmp = {
3      param = [ fn(x)=> true, fn(y)=> true ],
4      pre   = fn(x,y)=> true,
5      post  = fn(x,y,r)=> r==-1 || r==0 || r==1 },
6    cFind = {
7      param = [ fn(a)=> true, fn(v)=> true, cCmp ],
8      pre   = fn(a,v,c)=> true,
9      post  = fn(a,v,c,r)=> r==-1 || c(a[r],v)==0 },
10   find = fn(a,v,c) @ cFind =>
11     let f = fn(i)=> if i >= len(a) then -1
12                     else if c(a[i],v) == 0 then i
13                     else f(i + 1)
14     in f(0),
15   cmpAsc = fn(x,y)=>
16     if x < y then -1 else if x == y then 0 else 1
17 in
18   find([4,9,7,3], 7, cmpAsc)
```

Figure 3. Example with higher-order contract.

```
1 oblige = fn(ctr, fun)=>
2   fn(fun.formals)=>
3     let
4       pos = fun.srcLoc,
5       neg = getCaller().srcLoc
6     in obligePN(ctr, fun, pos, neg)(fun.formals)
```

Figure 4. Pseudo-code for intrinsic `oblige`.

contracts; attribute **pre** holds a boolean function of the parameters; and attribute **post** holds a boolean function of the parameters and the result of the computation. In Line 5 of the example, the postcondition of contract `cCmp` promises that the return value is in $\{-1, 0, 1\}$. Line 7 in contract `cFind` requires that the third parameter satisfies contract `cCmp`. Line 10 decorates the definition of function `find` with contract `cFind`. Function `find(a,v,c)` finds value `v` in list `a`, using comparator `c`.

Line 18 calls `find`. Function `find` is higher-order, because it takes function `cmpAsc` as a parameter. Contract `cFind` is higher-order, because it applies contract `cCmp` to a parameter. The contract checks that `cmpAsc` always returns values in $\{-1, 0, 1\}$. In this program, there are no contract violations, and the program returns 2, which is the 0-based index of 7 in `[4,9,7,3]`.

2.3.1 Obligation Rewrite

To support contracts, the IL interpreter does a pre-pass before evaluation, rewriting each function expression with a contract `ctr`

fn(x_1,...,x_n) @ *ctr* => *expr*

into a call to the `oblige` intrinsic function

oblige(*ctr*, **fn**(x_1,...,x_n)=> *expr*)

Figure 4 shows the pseudo-code for `oblige`. It takes two parameters, a contract `ctr` and a function `fun`, and generates a function that has the same signature as `fun`, but checks `ctr`. Because `oblige` reflects over dynamic interpreter-internal meta-data, it is implemented as an intrinsic. Most of the work of `oblige` actually happens in another intrinsic function `obligePN` that takes two additional parameters `pos` (for *positive* contract partner) and `neg` (for *negative* contract partner). Parameters `pos` and `neg` hold the source locations, i.e., file names and line and column numbers, of the function and its dynamic caller. When a precondition or postcondition fails, the error message can blame `pos` or `neg` as appropriate.

```
1  obligePN = fn(ctr, val, pos, neg)=>
2    if isFlat(ctr)
3      then
4        if ctr(val)
5          then val
6          else blame(pos, neg, ctr.srcLoc)
7      else
8        fn(ctr.post.formals[0, …, N-2])=>
9          if ctr.pre(ctr.post.formals[0, …, N-2])
10           then
11             let ctr.post.formals[N-1] = val(
12                   obligePN(ctr.params[0],
13                            ctr.post.formals[0],
14                            neg, pos),
15                   …,
16                   obligePN(ctr.params[N-2],
17                            ctr.post.formals[N-2],
18                            neg, pos))
19             in
20               if ctr.post(ctr.post.formals)
21                 then ctr.post.formals[N-1]
22                 else blame(pos, neg, ctr.post.srcLoc)
23           else blame(neg, pos, ctr.pre.srcLoc)
```

Figure 5. Pseudo-code for intrinsic *obligePN*.

2.3.2 Obligation Contravariance

The insight behind the *pos* and *neg* parameters is that blame alternates in higher-order contracts [7]:

1. If the precondition of a function is violated, blame the caller *neg*; if the postcondition is violated, blame the callee *pos*.
2. If the precondition of a parameter to the function is violated, blame the callee *pos*; if the postcondition is violated, blame the caller *neg*.
3. If the precondition of a parameter to a parameter to the function is violated, blame the caller *neg*; if the postcondition is violated, blame the callee *pos*.
4. And so on, flipping the "sign" of which partner, *pos* or *neg*, to blame with each level of higher-order parameters.

For an example of a contract violation, assume function *cmpAsc* in Lines 15-16 of Figure 3 is replaced by:

```
cmpAsc = fn(x,y)=> x - y
```

This version of *cmpAsc* fails the postcondition, because the result is not always in $\{-1, 0, 1\}$. The scenario matches the second-order case 2 above: the postcondition of parameter *cmpAsc* to function *find* is violated. Therefore, the error message blames the caller:

```
find.il:5:13: contract violated
find.il:18:3: ... blamed partner in contract
find.il:10:10: ... innocent partner in contract
```

That makes sense, because the caller (Line 18) is responsible for passing a function to *find* that fails its postcondition.

2.3.3 The *obligePN* Function

Figure 5 shows the pseudo-code for the *obligePN* intrinsic function. It has two cases: the flat (zero-order) case where the contract is just a function, and the non-flat (first- or higher-order) case where the contract is a **param**/**pre**/**post** record. The flat case in Lines 4-6 checks contract *ctr* on value *val*, returning *val* on success and generating an error blaming *pos* on failure. For example, in the call

```
obligePN(fn(x)=> x>0, 0, srcLoc_0, neg)
```

the value `0` violates the flat contract `x>0`. Therefore, this call yields an error message blaming the source location *srcLoc_0* responsible for the value `0`.

The non-flat case of Figure 5 checks the precondition **pre** (Line 9); wraps all parameters in contracts from **param** (Lines 12-18); calls the original function (Line 11); and finally checks the postcondition **post** (Line 20). In Line 8, the list slicing notation *ctr*.**post**.*formals*`[0,…,N-2]` refers to all but the last formal parameters of the postcondition, since the last parameter *ctr*.**post**.*formals*`[N-1]` holds the result and needs to be treated specially. The following call illustrates the non-flat case:

```
obligePN({ param = [fn(x)=> true, fn(y)=> true],
           pre   = fn(x, y)=> x * y > 0,
           post  = fn(x, y, res)=> res >= x + y },
         fun, pos, neg )
```

This call wraps the function *fun* in contract checks:

```
fn(x, y)=>
  if x * y > 0
    then
      let res = fun(
            obligePN(fn(x)=> true, x, neg, pos),
            obligePN(fn(y)=> true, y, neg, pos))
      in
        if res >= x + y
          then res
          else blame(pos, neg, "input:3:21")
    else blame(neg, pos, "input:2:21")
```

If the precondition `x * y > 0` is violated, Line 11 generates an error message blaming *neg*. If the postcondition `res >= x + y` is violated, Line 10 generates an error message blaming *pos*. Lines 5-6 apply the contracts from **param** to the parameters of *fun*.

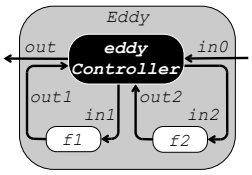
2.4 Macros with Contracts: Putting it All Together

This section described an IL that provides both macros (via quote `` ` `` and escape `%`) and contracts (via `@` decorations on function expressions). The implementation uses two pre-passes (one to find OL identifiers for hygiene, the other to rewrite contracts), and an interpreter. The interpreter evaluates the IL code, and the end result is the returned OL code. The evaluation uses a handful of special intrinsic functions: *freshId*, *oblige*, *getCaller*, *obligePN*, and *blame*. Functions and their contracts can be higher-order. The responsibility for hygiene is shared between the IL interpreter and the OL translator by using *freshId*.

3. Composite Operators

Composite operators in a streaming language are operators that encapsulate a stream subgraph. Our composites are higher-order: a composite operator can take another operator, including another composite, as a parameter. We expand composites (replace them by their subgraphs until only primitive operators remain) at compile-time. Our composites can have contracts, which lead to better expansion-time error messages. Expansion is implemented in two steps: translate to the IL from Section 2, then evaluate the IL.

3.1 Eddy Example



The running example for this section is the *Eddy* higher-order composite operator [2]. It has two parameters: operator *f1* and operator *f2*. Each time a data item arrives on *in0*, the *eddyController* decides whether to send it first to *f1* and then to *f2*, or vice versa. Both *f1* and *f2* are assumed to be selection operators, meaning they might drop data items. If a data item is not dropped,

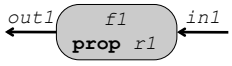
it goes to the other operator next; if it is not dropped there either, it goes to `out`. The `Eddy` dynamically optimizes performance by deciding the order in which data items go through `f1` and `f2` based on their observed cost and selectivity. For example, if both cost the same, but `f1` drops more data, it is cheaper to send data through `f1` first: if a data item gets dropped, the time for `f2` is saved.

3.2 Streaming Language with Composite Operators

We have designed and implemented composite operators for our production streaming language SPL [10], but for now, we will describe them based on Soulé et al.'s core streaming calculus Brooklet [18]. The calculus helps focus on the essentials while keeping the description self-contained; Section 5 will discuss our experiences with composites in the full language. This section describes Brooklet, extended with parameters, composite operators, and contracts. One of the design goals for composites was to have a consistent look-and-feel with the rest of the language. No background on macros is needed to understand Brooklet. For each feature, we first show the syntax rules, followed by a description paragraph.

opInvoke ::= '(' *ID*$^+$ ')' '<-' *ID* *actuals*$^?$ '(' *ID*$^+$ ')' ('**prop**' *ID*)$^?$ ';'
actuals ::= '{' *expr*$^+$ '}'

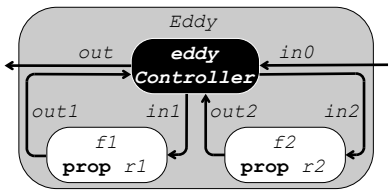
An operator invocation specifies a vertex and its immediate edges, such as `(out1) <- f1(in1) prop r1;`. The example uses an operator `f1` to consume a stream `in1` and produce a stream `out1`. The operator `f1` might be a primitive operator implemented in a traditional language such as C++ or Java, or it might be a composite operator; the invocation looks the same either way for orthogonality. The optional *actuals*$^?$ are actual parameters; e.g., a primitive selection operator can be parameterized by its filter condition, or a composite operator can be parameterized by other operators. During expansion, an operator invocation returns not just the generated code, but also domain-specific properties such as a data rate. The optional ('**prop**' *ID*)$^?$, like **prop** `r1` in the example, gives a name to the properties returned by the operator invocation. Properties can be used from postconditions, as we shall see later.



composite ::= '**composite**' *signature* *pre*$^?$ *compPost*$^?$ '{' *opInvoke*$^+$ '}'
signature ::= '(' *ID*$^+$ ')' '<-' *ID* *formals*$^?$ '(' *ID*$^+$ ')'
formals ::= '{' (*ID* ('@' *ID*)$^?$)$^+$ '}'

A composite definition encapsulates a subgraph, such as

```
1 composite (out) <- Eddy{f1, f2}(in0) {
2   (out,in1,in2) <- eddyController(in0,out1,out2);
3   (out1) <- f1(in1) prop r1;
4   (out2) <- f2(in2) prop r2;
5 }
```



The example omits the optional embedded contract *pre*$^?$, *compPost*$^?$, and ('@' *ID*)$^?$, which will be described later. The signature of the composite definition is designed to resemble an operator invocation. Syntactical resemblance between definitions and invocations makes code more readable, because the correspondence between each part of the signature and the invocation is immediately obvious. The parts of the signature specify the composite's output streams, name, optional formal parameters, and input streams. In the example, `f1` and `f2` are formals, and the body of the composite uses them as operators. That means that `Eddy` is higher-order.

pre ::= '**pre**' *expr*
compPost ::= '**prop**' *recordExpr*
| '**prop**' *ID* '=' *recordExpr* '**post**' *expr*

```
1 composite (out) <- Eddy{f1 @ cf, f2 @ cf}(in0)
2   prop r = { use = union(r1.use, r2.use),
3              def = union(r1.def, r2.def) }
4   post    disjoint(r1.use, r2.def)
5        && disjoint(r1.def, r2.use)
6 {
7   (out,in1,in2) <- eddyController(in0,out1,out2);
8   (out1) <- f1(in1) prop r1;
9   (out2) <- f2(in2) prop r2;
10 }
```

Figure 6. `Eddy` composite with contract.

The embedded contract of a composite consists of checks to perform during expansion. Figure 6 shows the `Eddy` composite from before with an embedded contract. The **post** clause in Lines 4-5 specifies a postcondition predicate. The compiler checks this predicate after expanding the composite. The predicate refers to property records `r1` and `r2` of operators invoked in the subgraph. The **prop** clause in Lines 2-3 specifies properties of this composite operator itself. The compiler returns them along with the expanded subgraph. They can be referred to by contracts further out in the expansion hierarchy. In the `Eddy` example, the operators `f1` and `f2` are expected to return, at compile-time, property records with attributes `use` and `def`, holding sets that describe used or defined stream data. The postcondition of the `Eddy` checks that the sets do not interfere, and the property record of the `Eddy` combines and returns the sets to the site where the `Eddy` was invoked. The names `use` and `def` have no special meaning to Brooklet, the user can freely choose any identifiers that make sense for the properties they care about. In practice, we assume that the names reside in namespaces to avoid clashes in large development teams.

contract ::= '**contract**' *signature* *pre*$^?$ *contPost*$^?$ ';'
contPost ::= '**prop**' *ID* '**post**' *expr*

A named contract resembles a composite operator definition without a subgraph, like **contract** `(qo) <- cf(qi);`. This example specifies a contract `cf` for an operator with exactly one output stream `qo` and one input stream `qi`. Such a named contract can then be attached to formal parameters, like `f1 @ cf, f2 @ cf` in Line 1 of Figure 6. Unlike a composite, a named contract does not define its own properties record, though it may give a name to the existing properties record of the operator it attaches to for use in the postcondition.

program ::= (*composite* | *contract*)$^+$

A Brooklet program consists of one or more composite and named contract definitions. One of the composites must be called `main`, such as:


```
1 composite (alerts) <- main(video) {
2   (alerts) <- Eddy { SeizureDetector,
3                      VisitorDetector } (video);
4 }
```

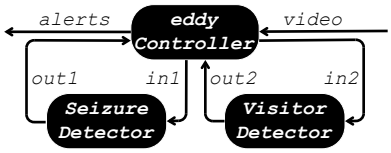
This example passes two video analysis operators to the `Eddy` for a health-care application. The `SeizureDetector` generates an alert when the video shows a patient having a seizure. The `VisitorDetector` cancels the alert when the video shows that visitors are present, to suppress false positives. The `Eddy` operator dynamically decides the order in which it sends the video feed to both operators, optimizing for observed cost and selectivity [2].

3.3 Composite Operator Expansion

Expansion starts from the `main` composite. In the running example, the expansion of `main` triggers the expansion of `Eddy` with the

following bindings: `out` ↦ `alerts`, `f1` ↦ `SeizureDetector`, `f2` ↦ `VisitorDetector`, and `in0` ↦ `video`.

The result is the flat graph consisting of the subgraph of the `Eddy` with the appropriate substitutions. While this expansion only has two levels, in our experience with SPL, multi-level expansions are not uncommon in practice. The graphical view of expansion is intuitive to users. But the implementation of expansion must take care to preserve hygiene and to check contracts. It does so by translating composites in Brooklet to macros in the IL. The Brooklet program *before* translation is:




```
1 composite (alerts) <- main(video) …
2 contract (qo) <- cf(qi);
3 composite (out) <- Eddy{f1 @ cf, f2 @ cf}(in0) …
```

The IL program *after* translation is:

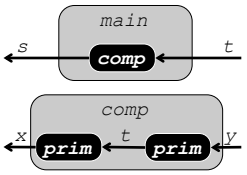
```
1 let main = fn(alerts, video) …,
2     cf = …,
3     Eddy = fn(out, f1, f2, in0) …
4 in let p = main …
5    in p.code
```

All streams and formals in an operator's signature turn into formals in the corresponding macro (compare for example Line 3 before vs. after translation). Each macro returns a record with a `code` attribute. Expansion starts by calling `main` (Line 4) and ends by returning the resulting code (Line 5). The remainder of this section refers to Brooklet as the object language (OL), and explains how hygiene and contracts work for Brooklet.

3.4 Hygiene in the OL

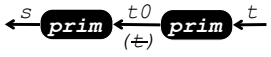
Hygiene is about avoiding accidental name captures. In the case of Brooklet, the names that are at risk of being captured are stream names. For example, the following program uses the name `t` both in `main` and in `comp`, and the expansion needs to make sure it does not get them mixed up.

```
composite (s) <- main(t) {
  (s) <- comp(t);
}
composite (x) <- comp(y) {
  (t) <- prim(y);
  (x) <- prim(t);
}
```



The Brooklet code after expansion is:

```
(t0) <- prim(t);
(s) <- prim(t0);
```



A naive substitution would have captured the name `t`. But instead, the compiler created a fresh identifier `t0` to use in place of the version of `t` that was strictly internal to `comp`. The implementation for this relies on the IL intrinsic `freshId`. The translator from OL to IL is responsible for putting in calls to `freshId`, and the interpreter of IL is responsible for evaluating `freshId`. We already saw the interpreter part in Section 2.2, so here, we focus on the translator part.

The translator creates one IL-level variable for each OL-level stream, holding a fragment with the identifier. Each OL fragment uses escapes to splice in those identifier-fragments. The IL-level variables can be either parameters of the current function (macro), or they can be local to the current function. If they are local to the current function, they are initialized with `freshId` to generate a fresh identifier. The IL code for the hygiene example is:

```
1  let main = fn(s, t) =>
2        let p0 = comp(s, t)
3        in { code = p0.code },
4      comp = fn(x, y) =>
5        let t = freshId("t")
6        in { code = `( (%t) <- prim(%y);
7                       (%x) <- prim(%t); ) }
8  in let s = freshId("s"), t = freshId("t"),
9         p = main(s, t)
10    in p.code
```

Unlike in Section 2.2, here the `freshId` intrinsic takes a string parameter with the original name, e.g., `freshId("t")` in Line 5. This enables the interpreter to generate names that are similar to the original name, e.g. `t0`, making the generated code easier to understand. The above IL program is actually simplified compared to what the translator really produces: it omits any IL code related to contracts, those are the subject of the following section.

3.5 Contracts in the OL

The OL has contracts, and the IL has contracts, so obviously, the goal is to translate OL contracts into IL contracts. That way, the IL interpreter can take care of the contra-variant blame tracking discussed in Section 2.3. However, there are a couple of complications to consider in the translation. First, the OL contract contains not just the explicit user-defined predicates, but also implicit structural information such as arity. Second, the OL contract can be higher-order, by attaching a named contract to a parameter with the `@`-syntax. Third, if the graph clause of one composite (the caller, such as `Eddy`) invokes another operator (the callee, such as `f1`), then the caller's contract can refer to the callee's properties (such as `r1`) in the OL. Hence, the translated IL must arrange for these properties to be passed around.

The translator from OL to IL establishes the following data structure conventions. Each parameter stream identifier is a record `{code=..., type=...}`, where `code` is a fragment with just an identifier, and `type` is either the string `"out"` or the string `"in"`, used to check structural properties. The result of each composite is a record `{code=..., props}`, where `code` is the fragment with the expanded subgraph, and *props* are the other attributes for user-defined properties in the contract. Since the postcondition needs access to not just the result of the composite itself, but also the results of nested operator invocations, the function returns not just one `{code=..., props}` record, but rather a whole list of them. The result of the composite itself is at index 0 in that list.

Before we explain these data structure conventions with an example, recall that in the IL, a non-flat contract is a record `{param=..., pre=..., post=...}`. We will use the `Eddy` with an embedded contract from Figure 6 as the example illustrating the translation. The explanation works up to Figure 7, introducing before and after snippets of OL and IL bit by bit.

In the OL, the `Eddy` signature is (Figure 6 Line 1):

```
composite (out) <- Eddy{f1 @ cf, f2 @ cf}(in0)
```

In the IL, the first three lines are:

```
Eddy = fn(out, f1, f2, in0)
  @ { param = [ fn(out)=> "out" == out.type, cf,
                cf, fn(in0)=> "in" == in0.type ],
```

Explanation: Predicate `fn(out)=> "out" == out.type` checks the structural contract that `out` is an output stream identifier. The two occurrences of `cf` check the higher-order contract on `f1` and `f2`. Finally, predicate `fn(in0)=> "in" == in0.type` checks the structural contract that `in0` is an input stream identifier.

In the OL, the `Eddy` subgraph is (Figure 6 Lines 7-9):

```
(out,in1,in2) <- eddyController(in0,out1,out2);
(out1) <- f1(in1) prop r1;
(out2) <- f2(in2) prop r2;
```

In the IL, the corresponding calls to *f1* and *f2* are:

```
let in1=freshId("in1"), out1=freshId("out1"),
    in2=freshId("in2"), out2=freshId("out2"),
    r1 = f1({ type="out", code=out1 },
            { type="in",  code=in1  })[0],
    r2 = f2({ type="out", code=out2 },
            { type="in",  code=in2  })[0]
```

Explanation: The *freshId* calls were explained in Section 3.4 on hygiene. Each parameter is tagged with its type ("in" or "out"). By the data structure conventions, the call to *f1* returns a list of results, and subscript [0] retrieves the result of *f1* itself, which gets bound to *r1*. Likewise, the result of *f2* gets bound to *r2*.

In the OL, the **prop** clause is (Figure 6 Lines 2-3):

```
prop r = { use = union(r1.use, r2.use),
           def = union(r1.def, r2.def) }
```

In the IL, the code that returns the result is:

```
[{ code = `( (%(out.code), %in1, %in2)
               <- eddyController
                  (%(in0.code), %out1, %out2);
             %(r1.code)
             %(r2.code) ),
   use = union(r1.use, r2.use),
   def = union(r1.def, r2.def) },
 r1, r2]
```

Explanation: By the data structure conventions, the result of the *Eddy* function is a list: the first element (at index 0) holds the expanded code and properties of the *Eddy* itself, whereas the other elements *r1* and *r2* hold results of nested operator invocations.

In the IL, the **post** clause is (Figure 6 Lines 4-5):

```
post    disjoint(r1.use, r2.def)
     && disjoint(r1.def, r2.use)
```

In the IL, the **post** attribute of the contract is:

```
post = fn(out, f1, f2, in0, r0)=>
         let r1=r0[1], r2=r0[2]
         in    disjoint(r1.use, r2.def)
            && disjoint(r1.def, r2.use)
```

Explanation: The return value from the function serves as the last parameter *r0* to the postcondition. By the data structure conventions, *r0* is a list, and **let** *r1=r0*[1], *r2=r0*[2] pulls out elements from the list. The body of the postcondition is copied verbatim from the OL to the IL.

3.6 Composites: Putting it all Together

This section describes our design and implementation of composite streaming operators with contracts. To simplify the presentation, this section uses an extended version of the Brooklet calculus. The design provides a syntax for composites that has the same look-and-feel as the base language. To reuse mechanisms for hygiene and contract checking, the compiler translates to the IL from Section 2. Brooklet composites and contracts turn into IL functions and contracts, respectively. The functions return not just the generated code, but also user-defined properties of the code, and the contracts use those properties to check user-defined predicates. After translation to IL, the interpreter takes over. However, both steps (translation and interpretation) are integrated into a single compiler, interfacing via the abstract syntax tree. Working with trees instead of text enables the compiler to graft OL code safely and efficiently, and to track line number information as tree node decorations.

4. Brooklet vs. SPL

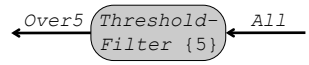
We have implemented compilers for both the Brooklet calculus and the SPL language [10]. While Section 3 uses Brooklet to describe the syntax and semantics of composites in a self-contained manner, most of this section uses SPL to report field experiences. To compare the two, consider a Brooklet operator invocation:

```
1  Eddy = fn(out, f1, f2, in0)
2    @ { param = [ fn(out)=> "out" == out.type, cf,
3                  cf, fn(in0)=> "in" == in0.type],
4        pre   = fn(out, f1, f2, in0)=> true,
5        post  = fn(out, f1, f2, in0, r0)=>
6                  let r1=r0[1], r2=r0[2]
7                  in    disjoint(r1.use, r2.def)
8                     && disjoint(r1.def, r2.use) }
9    => let in1=freshId("in1"), out1=freshId("out1"),
10          in2=freshId("in2"), out2=freshId("out2"),
11          r1 = f1({ type="out", code=out1 },
12                  { type="in",  code=in1  })[0],
13          r2 = f2({ type="out", code=out2 },
14                  { type="in",  code=in2  })[0]
15       in [{ code = `( (%(out.code), %in1, %in2)
16                        <- eddyController
17                        (%(in0.code), %out1, %out2);
18                      %(r1.code)
19                      %(r2.code) ),
20             use = union(r1.use, r2.use),
21             def = union(r1.def, r2.def) },
22           r1, r2]
```

Figure 7. Translated IL for the OL in Figure 6.

```
(Over5) <- ThresholdFilter{5}(All);
```

In SPL, the same operator invocation looks as follows:




```
stream<int32 x> Over5 = ThresholdFilter(All)
  { param threshold : 5; }
```

In SPL, streams are typed, and the type is declared where the stream is produced. In the example, **stream**<**int32** *x*> is the type for a stream of tuples, where each tuple has an **int32** attribute named *x*. In SPL, actual parameters are named. The example passes 5 to parameter *threshold* of the *ThresholdFilter* operator. Like in Brooklet, the syntax for invoking primitive or composite operators is the same.

To illustrate a composite operator definition in SPL, the following example defines the operator invoked previously:


```
1 composite ThresholdFilter(output F; input A) {
2   param expression $threshold;
3   graph stream<A> F = Filter(A)
4           { param filter : x > $threshold }
5 }
```

Line 2 declares the formal parameter, indicating that it expects an expression as its actual. The SPL compiler substitutes the actual for each occurrences of the formal during expansion. Besides the **expression** parameter mode, formals can also be declared to expect types, operators, or various other kinds of entities. In Line 3, **stream**<*A*> *F* declares stream *F* with the same type as stream *A*.

Both Brooklet and SPL provide higher-order composites, which are novel and a central contribution of this paper. Both compilers implement hygiene. In Brooklet, the only names at risk of capturing are stream names. In SPL, on the other hand, various entities can have names, including streams, types, attributes, variables, functions, etc. Furthermore, in SPL, names can be qualified by namespaces. This makes name handling in the SPL compiler more complicated, but it uses the same techniques for hygiene as described earlier. When the SPL compiler generates fresh identifiers to prevent name clashes, it encodes the expansion context in the name to help with visualization and debugging.

```
1  composite Uniq(output Out; input In) {
2    graph
3      stream<In> Out = Custom(In) {
4        logic state : {
5                mutable boolean first = true;
6                mutable In prev; }
7              onTuple In: {
8                if (first || prev != In) {
9                  submit(In, Out);
10                 first = false;
11                 prev = In; } } } }
```

Figure 8. Example for `Custom`-in-composite pattern.

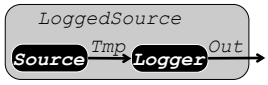
Both Brooklet and SPL provide contracts, but to different extents. Both compilers check implicit structural contracts during expansion. Also, both check user-defined first-order contracts. Only Brooklet provides user-defined higher-order contracts as described in this paper. SPL provides rich contracts on primitive operators, which are written as a combination of an XML file and an optional Perl script. The XML file, also known as the *operator model*, specifies commonly needed properties, which may be used for optimizations. The Perl script serves as a powerful escape hatch to check arbitrary domain-specific predicates.

5. Experiences

This section discusses experiences implementing and using the techniques described in this paper. Composites have no impact on run-time performance, because they are expanded at compile-time.

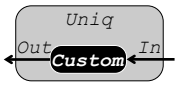
5.1 Simple Composites

Like the `ThresholdFilter` from earlier, SPL composites are often just first-order encapsulations of reusable stream subgraphs.



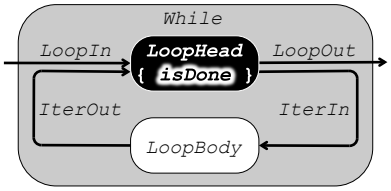
Even in the first-order case, our compiler must watch out for hygiene. For example, a composite `LoggedSource` might invoke a `Source` to produce an internal stream `Tmp`, and then invoke a `Logger` to gather some statistics, such as measuring throughput or counting duplicates. If the composite operator is instantiated multiple times, `Tmp` must be renamed to avoid clashes.

Primitive operators for SPL are written in C++, but the language also allows users to define the logic for simple operators directly in SPL code. This is typically done in an invocation of a `Custom` operator, and in practice, developers frequently encapsulate a single `Custom` in a composite for modularity. Figure 8 shows an example. The composite `Uniq` invokes `Custom` on the input stream `In` to produce the output stream `Out`. The invocation has a **`logic`** clause with two subclauses: **`state`** declares variables, and **`onTuple`** `In` declares a handler that fires each time a tuple arrives on stream `In`. The declaration **`mutable`** `In prev` in Line 6 declares variable `prev` to have the same type as tuples on stream `In`. The **`onTuple`** logic forwards a tuple if either it is the first tuple on the stream, or it is different from the previous tuple on the stream. When a stream name is used in a value context, it refers to the current tuple on the stream. For example, `prev != In` in Line 8 compares `prev` to the current tuple on stream `In`, and `submit(In, Out)` in Line 9 submits that current tuple to `Out`.



5.2 Higher-Order Composites

A higher-order composite is like a stencil for a subgraph with blanks to be filled in by parameters. Figure 1 in the introduction listed several examples: `Pipeline`, `SplitMerge`, `FeedbackLoop`, `Exchange`, `MapConcat`, and `Eddy`. We already saw `Eddy` in the running example for Section 3. Here, we will use `While` as another example.



Composite `While` has two parameters. Parameter `isDone` is an expression used in the loop head, and parameter `LoopBody` is an operator. As long as `isDone` returns false, data is sent repeatedly through the `LoopBody`; in the end, it is sent to `LoopOut`. Among other things, the `While` operator is useful for iterative data mining algorithms on large data sets, such as linear regression, matrix factorization, or PageRank [15]. For instance, in PageRank, parameter `isDone` is a convergence check that returns true when the difference between iteration results falls below a threshold. Parameter `LoopBody` is an operator that adds a fraction of the rank of each page to the ranks of all the pages it links to.

5.3 Telco Benchmark

We have built a telecommunications benchmark in SPL, which makes heavy use of composite operators, including higher-order ones. It consists of 39 applications implementing common operations used to monitor logs produced from server-side telecommunications software. The goal is to collect, summarize, and report operational statistics in real-time. The applications cover parsing, formatting, filtering, enrichment, projection, aggregation, state management, splitting, correlation, and pattern detection to name a few.

An important aspect of the benchmark applications is their scale and distributed nature. Each application processes 40 data sources divided into two groups, where data from pairs of sources belonging to different groups needs to be brought together for correlation. The correlated results are further categorized into 4 pools and then merged on a per-pool basis. Eventually, per-pool results are combined into a final result stream. The average benchmark application has a complex topology consisting of 337 primitive operator instances. Not only are portions of these topologies common across many applications, but they also have similar constraints for which operators to place on which hosts. Two challenges we faced during the design of this benchmark were i) to minimize code repetition and ii) to abstract away the details of the topology and placement by separating it from the core application logic.

Composites in SPL in general, and user-defined higher-order composites in particular, have effectively resolved both challenges. As part of the benchmark suite, we have created a toolkit of composites that are shared across all benchmark applications. This toolkit contains two sets of artifacts.

1. First-order composite operators that encapsulate common stream manipulations used in different applications.
2. Higher-order composite operators that encapsulate common topologies and placements. These composites take as parameters other operators with the core application logic to be embedded into the boiler-plate topology.

The first set of composites facilitate reuse, reducing code repetition. The second set of operators not only achieve the same reuse goals, but also abstract away the details of the topology and placement from the main application logic. With this toolkit in place, each application only defines its core logic as a few composites, and instantiates the topology by calling one of the higher-order composites from the toolkit. Furthermore, the core application logic frequently uses the first-order composites from the toolkit.

The Telco benchmark has a total of 7,029 lines of code (LOC), including 4,350 LOC in the toolkit and the remaining 2,679 LOC in the 39 applications. We modified the SPL compiler to print the

	Toolkit			Applications		
	Avg.	None	≥ 1	Avg.	None	≥ 1
Input ports	0.42	34	19	0.36	39	22
Output ports	0.55	28	25	0.36	39	22
Formal parameters	4.13	1	52	3.23	0	61
... operators	1.62	20	**33**	0.00	61	0
OpInvokes in subgraph	5.25	**3**	50	4.38	0	61
... composite	4.77	15	38	1.92	6	55
... parameter	0.21	44	**9**	0.00	61	0

Table 1. Statistics about the 114 non-main composite operator definitions in the Telco benchmark. Columns Avg. show the average number of occurrences per composite. Columns None count the number of composites with zero occurrences, and Columns ≥ 1 count remaining composites.

source code for the stream graph after expanding all composite operators. After expansion, the Telco benchmark has a total of 316,725 LOC, an expansion factor of 45. Before expansion, the average application has 69 LOC, with a minimum of 36 and a maximum of 184. That means that the applications really only define core logic. After expansion, the average application has 8,121 LOC, with a minimum of 3,369 and a maximum of 13,010. Composite operators saved us from writing a lot of repetitive code in the Telco benchmark.

The Telco benchmark has a total of 153 composite operator definitions, including 39 main composites (one per application) and 53 composites in the toolkit. Table 1 characterizes the non-main composites, separated into toolkit vs. application. Row "Formal parameters: operators" column "$\geq$ 1" shows that 33 composites have a non-zero number of parameters that are operators, in other words, 33 composites are higher-order. That accounts for 62% of the composites in the toolkit; none of the composites in the applications are higher-order. Row "OpInvokes in subgraph" column "None" shows that 3 composites have empty subgraphs: they are used as default values for the optional parameters of some of the higher-order composites, in order to implement optional subgraphs. A specific use-case for empty composites is optional workload generators. Some of the higher-order composites take as a parameter the workload generator to use, which by default is an empty composite, i.e., no workload is generated on-the-fly. Row "OpInvokes in subgraph: parameter" column "$\geq$ 1" shows that out of the 33 higher-order composites, only 9 directly invoke their operator parameters in their subgraph. The other higher-order composites merely pass operator parameters through to other composites.

After expansion, the Telco benchmark has a total of 11,866 composite operator instances. Their depths range from 0 for main composites to 6 for composites nested deeply in the expansion, with an average depth of 4.63. The average composite instance has 2.04 immediate children, with an average of 1.01 primitive and 1.03 composite immediate children. 762 composite instances have zero immediate children; those are instances of the 3 empty composite definitions. An example of a deep expansion chain is:

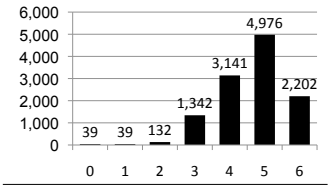


Table 2. Depth histogram of composite operator instances.

```
0 Main
1 CommonMainWithBSidedMergedChains
2 ChainContainerForBSidedMergedChainsForP1AndP3
3 ChainContainerForBSidedMergedChains
4 $coreB
5 OrderAndCompletenessDetector
6 OrderMessages
7 Custom
```

The expansion starts from the `Main` composite at depth 0. It passes the core application logic as operator parameters to generate the topology. One of the parameters gets passed through up to depth 4, where the operator parameter itself is instantiated. The core application logic is at depth 5; it invokes an operator from the toolkit at depth 6 to reuse common functionality. Finally, expansion ends with a primitive operator at depth 7.

5.4 Contracts

A contract on a composite specifies pre- and post-conditions to check during expansion. Section 3 already showed such a contract for the `Eddy` running example. This section discusses rates for synchronous data flow (SDF) [14] as another example. The SDF data rate of an operator is the number of data items it produces per data item it consumes. For example, a `Dup` operator might have a rate of 2, indicating that it produces 2 output items for each input item. The term *synchronous* is defined to mean that all the rates are constant and known at compile time. That is useful for compiler optimizations such as double-buffering.

Brooklet's contracts for composite operators can be used to compute and check rates. For example, the rate of a `Pipeline` composite is the product of the rates of its stages. As another example, in a `SplitMerge` composite, all parallel stages must have the same rate, which becomes the rate of the `SplitMerge` itself. If the stages have different rates, the compiler raises a contract-violation error. Figure 9 shows an example stream graph, expanded from a nested `Pipeline` in a `SplitMerge`, with rates.

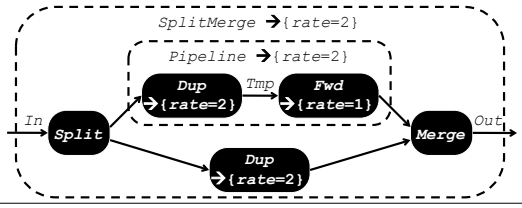


Figure 9. Stream graph with SDF data rates. The dashed ovals indicate expanded composite operators.

In SPL, contracts get used extensively for primitive operators, and can be quite sophisticated. For example, the contracts for database adapter operators in our library access the schema of the database instance, and check that the adapter is typed correctly. As another example, the contract for the relational `Join` operator in our library checks relationships between parameters for inner, left outer, or right outer joins. Besides user-defined contracts for primitive operators, the SPL compiler checks implicit structural contracts, such as operator arity, parameter names, and stream types.

6. Related Work

This paper is the first to use macros for implementing composites, making contributions in both areas.

6.1 Related Work on Composites

The StreamIt language allows users to define first-order composite operators [20]. StreamIt users cannot define their own higher-order composites. Instead, StreamIt comes with three built-in higher-order composites `Pipeline`, `SplitMerge`, and `FeedbackLoop`. In contrast, we allow users to define these and other higher-order composites by hand, using a single language feature. Instead of picking a fixed set of higher-order composites to build into the language, we leave this up to users and library writers, thus keeping the set of higher-order composites open. In addition, we provide contracts to make composites more robust. Other streaming languages with composite operators include LabView [1] and EventFlow [17]. Like

StreamIt and unlike our work, they restrict users to defining first-order composites.

Whereas StreamIt, LabView, and EventFlow follow streaming as the primary paradigm, other languages such as DryadLINQ [24] or FlumeJava [3] are primarily object-oriented with data-flow extensions. In those languages, the main program dynamically composes stream graphs. Instead of abstracting with composites, the user has access to host language abstraction features.

This paper describes composites as an extension to the Brooklet calculus [18]. This makes the description self-contained and helps focus on essential features that interact with hygiene and contracts. A minor difference to the original Brooklet is that our paper adds operator parameters and contracts. The key difference is that our paper adds composites.

6.2 Related Work on Macros and Contracts

Hygienic macros avoid accidental name capture [13]. In Scheme, macros are viewed as a set of extension "forms" over a base language. Macro expansion happens in multiple passes, until only base language code remains. *Macros that work* rename identifiers at the end after all expansion passes to achieve hygiene [4]. Subsequent work has further improved Scheme's macro facilities to the point where they can support sophisticated language extensions [21]. In contrast, our approach views the object language (OL) as a standalone language with its own syntax and scoping rules. The translator to intermediate language (IL) inserts calls to `freshId` before the IL interpreter expands macros. But the key difference is that we provide hygiene for composite operators in a streaming language.

Our IL is a novel composition of established concepts, one of them being the quote-and-escape syntax. Originating from Lisp, quote and escape have been adopted widely as a convenient notation to toggle between stages [19, 21, 22] or even languages [11]. Quote and escape can be arbitrarily nested and are implemented by grafting abstract syntax trees. Prior published literature on quote-and-escape syntax has not explored its interaction with higher-order contracts.

Our contracts for higher-order macros build upon prior work on contracts for higher-order functions [7]. The prior work explains the blame contra-variance issue, and how to solve it with wrapper functions. The key difference is that we combine contracts with macros to support composite operators. Contracts are a means to make macros more robust. There has been other work on making macros more robust by different means. Some systems, such as Java Mint, accomplish this by a combined type system for both the base language and the macro system [23]. Fortified macros embed restrictions on macro usage with dispatch patterns [5]. C++ concepts specify restrictions on C++ templates [9]. MorphJ checks naming constraints in code generation with Java generics [12]. Our contracts allow users to compute arbitrary predicates, which can even read external configuration files or database schemas.

7. Conclusions

The three main contributions of this paper are a design of higher-order composites for streaming, contracts for the composites, and an implementation of the composites via translation to an intermediate language with macros and contracts. Our implementation separates the general macro expansion technology from the domain-specific aspects of composites. We accomplish this by putting them into separate passes of a compilation pipeline, yielding a description that can be understood step by step. The steps of our approach could also be applied to macro-like features in other languages.

Acknowledgements. We are grateful to John Field, Robert Grimm, Byeongcheol Lee, Rodric Rabbah, Scott Schneider, and Robert Soulé for feedback on earlier drafts of this paper.

References

[1] http://www.ni.com/gettingstarted/labviewbasics/.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*, 2000.

[3] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Programming Language Design and Implementation (PLDI)*, 2010.

[4] W. D. Clinger and J. Rees. Macros that work. In *Principles of Programming Languages (POPL)*, 1991.

[5] R. Culpepper and M. Felleisen. Fortifying macros. In *International Conference on Functional Programming (ICFP)*, 2010.

[6] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *Transactions on Software Engineering (TSE)*, 2002.

[7] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[8] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Int. Conf. on Management of Data (SIGMOD)*, 1990.

[9] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[10] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. SPL Streams Processing Language Specification. Technical Report RC24897, IBM, 2009.

[11] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[12] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *Programming Language Design and Implementation (PLDI)*, 2008.

[13] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LISP and Functional Programming (LFP)*, 1986.

[14] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 1987.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford Digital Libraries Working Paper*, 1998.

[16] C. Ré, J. Siméon, and M. F. Fernàndez. A complete and efficient algebraic compiler for XQuery. In *International Conference on Data Engineering (ICDE)*, 2006.

[17] N. Seyfer, R. Tibbetts, and N. Mishkin. Capture fields: Modularity in a stream-relational event processing language. In *Conference on Distributed Event-Based Systems (DEBS)*, 2011.

[18] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, 2010.

[19] W. Taha and T. Sheard. Multi-stage programming with explicit annotation. *Partial Evaluation and Program Manipulation (PEPM)*, 1997.

[20] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction (CC)*, 2002.

[21] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Programming Language Design and Implementation (PLDI)*, 2011.

[22] D. Weise and R. Crew. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, 1993.

[23] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *Programming Language Design and Implementation (PLDI)*, 2010.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Operating Systems Design and Implementation (OSDI)*, 2008.