SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2338

River: an intermediate language for stream processing

Robert Soulé[1,*,†], Martin Hirzel[2], Buğra Gedik[3] and Robert Grimm[4]

[1] *Faculty of Informatics, Università della Svizzera italiana, Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland*
[2] *IBM Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA*
[3] *Department of Computer Engineering, Bilkent University, Bilkent, Ankara 06800, Turkey*
[4] *Department of Computer Science, New York University, 715 Broadway Room 711, New York, NY 10003, USA*

SUMMARY

This paper presents both a calculus for stream processing, named Brooklet, and its realization as an intermediate language, named River. Because River is based on Brooklet, it has a formal semantics that enables reasoning about the correctness of source translations and optimizations. River builds on Brooklet by addressing the real-world details that the calculus elides. We evaluated our system by implementing front-ends for three streaming languages, and three important optimizations, and a back-end for the System S distributed streaming runtime. Overall, we significantly lower the barrier to entry for new stream-processing languages and thus grow the ecosystem of this crucial style of programming. Copyright © 2015 John Wiley & Sons, Ltd.

Received 19 June 2014; Revised 14 May 2015; Accepted 18 May 2015

KEY WORDS: stream processing; optimizations

1. INTRODUCTION

It is widely accepted that virtual execution environments help programming languages by decoupling them from the target platform and vice versa. At its core, a virtual execution environment provides a small interface with well-defined behavior, facilitating robust, portable, and economic language implementations. Similarly, a calculus is a formal system that mathematically defines the behavior of the essential features of a domain. This paper demonstrates how to use a calculus as the foundation for an execution environment for stream processing. Stream processing makes it convenient to exploit parallelism on multicores or even clusters. Streaming languages are diverse [1–5], because they address many real-world domains, including transportation, audio and video processing, network monitoring, telecommunications, health care, and finance.

The starting point for this paper is the Brooklet calculus [6]. A Brooklet application is a stream graph, where each edge is a conceptually infinite stream of data items and each vertex is an operator. Each time a data item arrives on an input stream of an operator, the operator fires, executing a pure function to compute data items for its output streams. Optionally, an operator may also maintain state consisting of variables to remember across operator firings. Prior work demonstrated that Brooklet is a natural abstraction for different streaming languages.

The finishing point for this paper is the River execution environment [7]. Extending a calculus into an execution environment is challenging. A calculus deliberately abstracts away features that are not relevant in theory, whereas an execution environment must add them back in to be practical. The question is how to do that while (i) maintaining the desirable properties of the calculus,

*Correspondence to: Robert Soulé, Faculty of Informatics, Università della Svizzera italiana, Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland.
†E-mail: robert.soule@usi.ch

(ii) making the source language development effort economic, and (iii) safely supporting common optimizations and reaching reasonable target-platform performance. The answers to these questions are the research contributions of this paper.

On the implementation side, we wrote front-ends for dialects of three very different streaming languages (Continuous Query Language (CQL) [1], Sawzall [4], and StreamIt [5]) on River. We wrote a back-end for River on System S [8], a high-performance distributed streaming runtime. And we wrote three high-level optimizations (placement, fusion, and fission) that work at the River level, decoupled from and thus reusable across front-ends. This is a significant advance over prior work, where source languages, optimizations, and target platforms are tightly coupled. For instance, because River's distributed target platform, System S, runs on a shared-nothing cluster, our implementation of the CQL front-end on River yields the first distributed implementation of the CQL.

This journal paper combines and extends our conference papers on the Brooklet calculus [6] and the River execution environment [7]. It goes beyond the conference papers by providing more details on CQL (source syntax, source semantics, and type system) and StreamIt (complete translation rules). It also offers more extensive explanations and examples to make the material more approachable than in the limited-page budget of the conference papers.

Overall, this paper shows how to obtain the best of both theory and practice for stream processing. Starting from a calculus supports formal proofs showing that front-ends realize the semantics of their source languages and that optimizations are safe. And finishing in an execution environment lowers the barrier to entry for new streaming language implementations and thus grows the ecosystem of this crucial style of programming.

The rest of this paper is organized as follows. Section 2 introduces the Brooklet calculus. Section 3 discusses the River execution environment. Section 4 presents detailed translations from CQL, Sawzall, and StreamIt to River. Section 5 discusses three optimizations implemented in River, operator fission, fusion, and placement, that can be reused across languages. Section 6 describes the interface between River's runtime support and a target streaming engine. Section 7 provides the details of our evaluation. Section 8 discusses related work. Section 9 identifies future work. Section 10 concludes.

2. A CALCULUS FOR STREAM PROCESSING

A stream-processing language is a language that hides the mechanics of stream processing; it notably has built-in support for moving data through computations and for composing the computations with each other. Brooklet is a core calculus for such stream-processing languages. It is designed to model any streaming language and to facilitate reasoning about language implementation. To achieve these goals, Brooklet models state and non-deterministic operator firings as core concepts and abstracts away local deterministic computations inside of operators.

2.1. Brooklet program example: IBM market maker

As an example of a streaming program, we consider a hypothetical application that trades IBM stock. Data arrive on two input streams, `bids(symbol,price)` and `asks(symbol,price)`, and leaves on the `result(cnt,symbol,price)` output stream. Because the application is only interested in trading IBM stock, it filters out all other stock symbols from the input. The application then matches bid and asks prices from the filtered streams to make trades.

To keep the example as straightforward as possible, we have made a few simplifications. First, we assume that each sale is for exactly one share. Second, we do not buffer bids. Rather, the `SaleJoin` operator that matches bids to ask prices performs a one-sided join, discarding a bid with no sale if a match cannot be made. Note that these simplifications are not necessary in practice. For example, as will be explained in the following, if a user wanted a two-sided join, Brooklet can support it with an additional state variable. The Brooklet program in the bottom-left corner of Figure 1 produces a stream of trades of IBM stock, along with a count of the number of trades.

Brooklet syntax:

P_b	$::=$	*out in* $\overline{op}$	*Brooklet program*
out	$::=$	`output` $\overline{q}$ `;`	*Output declaration*
in	$::=$	`input` $\overline{q}$ `;`	*Input declaration*
op	$::=$	`(` $\overline{q}, \overline{v}$ `)` $\leftarrow f$ `(` $\overline{q}, \overline{v}$ `);`	*Operator*
q	$::=$	*id*	*Queue identifier*
v	$::=$	`$` *id*	*Variable identifier*
f	$::=$	*id*	*Function identifier*

Brooklet example: IBM market maker.

```
output result;
input bids, asks;
(ibmBids) ⟵  SelectIBM(bids);
(ibmAsks) ⟵  SelectIBM(asks);
($lastAsk)⟵  Window(ibmAsks);
(ibmSales)⟵  SaleJoin(ibmBids,$lastAsk);
(result,$cnt) ⟵  Count(ibmSales,$cnt);
```

Brooklet semantics: $F_b \vdash \langle V, Q\rangle \longrightarrow \langle V', Q'\rangle$

$$\frac{\begin{array}{c} d, b = Q(q_i) \\ op = \ (\_,\_) \leftarrow f\ (\overline{q}, \overline{v})\ ; \\ (\overline{b}', \overline{d}') = F_b(f)(d, i, V(\overline{v})) \\ V' = \mathit{updateV}(op, V, \overline{d}') \\ Q' = \mathit{updateQ}(op, Q, q_i, \overline{b}') \end{array}}{F_b \vdash \langle V, Q\rangle \longrightarrow \langle V', Q'\rangle} \quad \text{(E-FIREQUEUE)}$$

$$\frac{\begin{array}{c} op = \ (\_,\overline{v}) \leftarrow f\ (\_,\_)\ ; \\ V' = [v_1 \mapsto d_1, \ldots, v_n \mapsto d_n]V \end{array}}{\mathit{updateV}(op, V, \overline{d}) = V'} \quad \text{(E-UPDATEV)}$$

$$\frac{\begin{array}{c} op = \ (\overline{q},\_) \leftarrow f\ (\overline{q'},\_)\ ; \\ d_f, b_f = Q(q'_f) \\ Q' = [q'_f \mapsto b_f]Q \\ Q'' = [q_1 \mapsto Q(q_1), b_1, \ldots, q_n \mapsto Q(q_n), b_n]Q' \end{array}}{\mathit{updateQ}(op, Q, q_f, \overline{b}) = Q''} \quad \text{(E-UPDATEQ)}$$

Figure 1. Brooklet syntax and semantics.

2.2. Brooklet syntax

A Brooklet program defines a directed, possibly cyclic, graph of *operators* containing pure *functions* connected by first-in, first-out *queues*. It uses *variables* to explicitly weave state through operators. Data items on a queue model network packets in transit. Data items in variables model stored state; because data items may be listed, a variable may store arbitrary amounts of historical data. The following line from the market maker application defines an operator:

```
(ibmSales) ⟵ SaleJoin(ibmBids, $lastAsk);
```

The operator reads data from input queue `ibmBids` and variable `$lastAsk`. It passes that data as parameters to the pure function `SaleJoin` and writes the result to the output queue `ibmSales`. Brooklet does not define the semantics of `SaleJoin`. Modeling local deterministic computations is well understood [9, 10], so Brooklet abstracts them away by encapsulating them in opaque functions. On the other hand, a Brooklet program does define explicit uses of state. In the example, the following line defines a window over the stream `ibmAsks`:

```
($lastAsk) ⟵ Window(ibmAsks);
```

The window contains a single tuple corresponding to the most recent ask for an IBM stock, and the tuple is stored in the variable `$lastAsk`. Both the `Window` and `SaleJoin` operators access `$lastAsk`.

The `Window` operator writes data to `$lastAsk`, but does not use the data stored in the variable in its internal computations. Operators that incrementally update state must both read and write the same variable, such as in the `Count` operator:

```
(result, $cnt) ⟵ Count(ibmSales, $cnt);
```

Queues that appear only as operator input, such as `bids` and `asks`, are program inputs, and queues that appear only as operator output, such as `result`, are program outputs. Brooklet's syntax uses the keywords `input` and `output` to declare a program's input and output queues. We say that a queue is *defined* if it is an operator output or a program input. We say that a queue is *used* if it is an operator input or a program output. Variables may be defined and used in several clauses, because they are intended to weave state through a streaming application. In contrast, each queue must be defined once and used once. This restriction facilitates using our calculus for proofs and optimizations. The complete Brooklet grammar appears in Figure 1.

2.3. Notation

Throughout the paper, an over-bar, as in $\overline{q}$, denotes a finite sequence $q_1, \ldots, q_n$, and the ith element in that sequence is written q_i, where $1 \leqslant i \leqslant n$. For notational convenience, we do not specify the separator character in the sequence. The lower-case letter b is reserved for lists, and $\bullet$ is an empty list. A comma indicates *cons* or *append*, depending on the context; for example, d, b is a list consed from the first item d and the remaining items b. A bag is a set with duplicates. The notation $\{e\ :\ \mathit{condition}\}$ denotes a bag comprehension: it specifies the bag of all es where the *condition* is true. The symbol $\emptyset$ stands for both an empty set and an empty bag. If E is a store, then the substitution $[v \mapsto d]E$ denotes the store that maps name v to value d and is otherwise identical to E. Angle brackets identify a tuple. For example, $\langle \sigma, \tau \rangle$ is a tuple that contains the elements σ and τ. In inference rules, an expression of the form $d, b = b'$ performs pattern matching; it succeeds if the list b' is non-empty, in which case, it binds d to the first element of b' and b to the remainder of b'. Pattern matching also works on other meta-syntax, such as tuple construction. An underscore character _ indicates a wildcard and matches anything. Semantics brackets such as $[\![P_b]\!]_z^p$ indicate translation. The subscripts $_{a,b,c,s,z}$ stand for (stream relational) Algebra, Brooklet, CQL, StreamIt, and Sawzall, respectively. The superscripts ^{p,i,o} stand for program, input, and output translation.

2.4. *Brooklet semantics*

A program operates on data items from a domain $\mathcal{D}$, where a data item is a general term for anything that can be stored in queues or variables, including tuples, bags of tuples, lists, or entire relations from persistent storage. Queue contents are represented by lists of data items. We assume that the transport network is lossless and order preserving but may have arbitrary delays, so queues support only *push*-to-back and *pop*-from-front operations.

Brooklet execution configuration. The function environment F_b maps function names to function implementations. This environment allows us to treat operator functions as opaque. For example, F_b(`SelectIBM`) would return a function that filters out data items whose stock symbol differs from IBM. At any given time during program execution, the configuration of the Brooklet program is defined as a pair $\langle V, Q \rangle$, where V is a store that maps variable names to data items (in the market maker example, `$cnt` is initialized to zero and `$lastAsk` is initialized to the tuple $\langle$'IBM', $\infty\rangle$), and Q is a store that maps queue names to lists of data items (initially, all queues except the input queues are empty).

Brooklet execution semantics. Computation proceeds in small steps. Each step consumes exactly one data item from one input queue of one operator, by firing Rule E-FIREQUEUE in Figure 1. To explain this rule, we illustrate each line one by one, starting with the following intermediate configuration of the market maker example:

$$V = [\texttt{\$lastAsk} \mapsto \langle\text{'IBM'}, 119\rangle, \texttt{\$cnt} \mapsto 0]$$

$$Q = \begin{bmatrix} \texttt{bids} \mapsto \bullet,\ \texttt{ibmBids} \mapsto (\langle\text{'IBM'}, 119\rangle, \langle\text{'IBM'}, 124\rangle)\,, \\ \texttt{asks} \mapsto \bullet,\ \texttt{ibmAsks} \mapsto \bullet, \\ \texttt{ibmSales} \mapsto \bullet,\ \ \texttt{result} \mapsto \bullet \end{bmatrix}$$

$d, b = Q(q_i)$: Non-deterministically select a firing queue q_i. For a queue to be eligible as a firing queue, it must satisfy two conditions: it must be non-empty (because we are binding d, b to its head and tail) and it must appear as an input to some operator (because we are executing that operator's firing function). This step can select any queue satisfying these two conditions.
For example, q_i = `ibmBids`, $d = \langle$'IBM', 119$\rangle$, $b = (\langle$'IBM', 124$\rangle)$.

$op = (\_, \_) \leftarrow f(\overline{q}, \overline{v})$;: Because of the single-use restriction, q_i uniquely identifies an operator.
For example, op = `(ibmSales)` $\leftarrow$ `SaleJoin(ibmBids, $lastAsk);`.

$(\overline{b}', \overline{d}') = F_b(f)(d, i, V(\overline{v}))$: Use the function name to look up the corresponding function from the environment. The function parameters are the data item popped from q_i, the index i relative to the operator's input list, and the current values of the variables in the operator's input list. For each output queue, the function returns a list b'_j of data items to append, and for each output variable, the function returns a single data item d'_j to store.

For example, $\overline{b}' = ((\langle\text{'IBM'}, 119, 119\rangle)), \overline{d}' = \bullet$,
$d = \langle\text{'IBM'}, 119\rangle, i = 1, V(\overline{v}) = (\langle\text{'IBM'},119\rangle)$.

$V' = \mathit{updateV}(op, V, \overline{d}')$: Update the variables using the output $\overline{d}'$.
In this example, $\overline{d}' = \bullet$, so $V' = V$.

$Q' = \mathit{updateQ}(op, Q, q_i, \overline{b}')$: Update the queues. Each time an operator fires, it pops one data item from the firing queue and produces a variable number of data items that are put on the output queues. The $d_f, b_f = Q(q'_f)$ pattern matches on the firing queue to bind the head and tail. The head, d_f, is popped from the firing queue, creating a new queue environment in $Q' = [q'_f \mapsto b_f]Q$. The data items produced by the operator firing, identified as $b_1, \ldots, b_n$, are pushed onto the appropriate output queues, $q_1, \ldots, q_n$, creating an updated queue environment, Q''. The example has only one output queue and datum.

For example, $Q' = \begin{bmatrix} \texttt{bids} \mapsto \bullet, & \texttt{ibmBids} \mapsto (\langle\text{'IBM'}, 124\rangle), \\ \texttt{asks} \mapsto \bullet, & \texttt{ibmAsks} \mapsto \bullet, \\ \texttt{ibmSales} \mapsto (\langle\text{'IBM'}, 119, 119\rangle), & \texttt{result} \mapsto \bullet \end{bmatrix}$

2.5. *Brooklet execution function*

We denote a program's input $\langle V, Q\rangle$ as I_b and an output $\langle V', Q'\rangle$ as O_b. Given a function environment F_b, program P_b, and input I_b, the function $\rightarrow_b^* (F_b, P_b, I_b)$ yields the set of all final outputs. An execution yields a final output when no queue is eligible to fire. Because of non-determinism, the set may have more than one element. One possible output O_b of our running example is

$$V = [\texttt{\$lastAsk} \mapsto \langle\text{'IBM'}, 119\rangle, \texttt{\$cnt} \mapsto 1]$$

$$Q = \begin{bmatrix} \texttt{bids} \mapsto \bullet, & \texttt{asks} \mapsto \bullet, & \texttt{ibmSales} \mapsto \bullet, \\ \texttt{ibmBids} \mapsto \bullet, & \texttt{ibmAsks} \mapsto \bullet, & \texttt{result} \mapsto (\langle 1, \text{'IBM'}, 119\rangle) \end{bmatrix}$$

The example illustrates the finite case. But in many application domains, streams are conceptually infinite. To use our semantics in that case, we use a theoretical result from prior work: if a stream program is computable, then one can generalize from all finite prefixes of an infinite stream to the infinite case [11]. If $\rightarrow_b^*$ yields the same result for all finite inputs to two programs, then we consider these two programs equivalent even on infinite inputs.

2.6. *Brooklet abstractions and their rationale*

The following is a list of simplifications in the Brooklet semantics, along with the insights behind them.

Atomic steps. Brooklet defines execution as a sequence of atomic steps. Being a small-step operational semantics makes it amenable to proofs. Each atomic step contains an entire operator firing. By not sub-dividing firings further, it avoids interleavings that unduly complicate the behavior. In particular, Brooklet does not require complex memory models.

Pure functions. Functions in Brooklet are pure, without side effects, and with repeatable results. This is possible because state is explicit and separate from operators. Keeping state separate also makes it possible to see right away which operators in an application are stateless or stateful, use local or shared state, and read or write state.

Opaque functions. Brooklet elides the definition of the functions for operator firings, because semantics for local sequential computation are well understood.

Non-determinism. Each step in a Brooklet execution non-deterministically picks a queue to fire. This non-deterministic choice abstracts away from concrete schedules. In fact, it even avoids the need for any centralized scheduler, thus enabling a distributed system without costly coordination. Note that determinism can be implemented on top of Brooklet with the appropriate protocols, as will be shown by translations from deterministic languages in Section 4.

No physical platform. Brooklet programs are independent from any actual machines they would run on.

Finite execution. Stream-processing applications run conceptually forever, but a Brooklet execution is a finite sequence of steps. As mentioned in the previous section, one can reason about an infinite execution by induction over each finite prefix.

2.7. Brooklet summary

Brooklet is a core calculus for stream processing. We designed it to be a general model for streaming languages and to facilitate reasoning about program implementation. Brooklet models state through explicit variables, thus making it clear where an implementation needs to store data. Brooklet captures inherent non-determinism by *not* specifying which queue to fire for each step, thus permitting all interleavings possible in a distributed implementation.

3. A VIRTUAL EXECUTION ENVIRONMENT FOR STREAM PROCESSING

On the one hand, Brooklet, being a calculus, makes abstractions. In other words, it removes irrelevant details to reduce stream processing to the features that are essential for formal reasoning. On the other hand, River, being an execution environment, has to take a stand on each of the abstracted-away details. This section describes these decisions and explains how the execution environment retains the benefits of the calculus.

3.1. River concretizations and their rationale

This section shows how the River execution environment fills in the holes left by the Brooklet calculus. For each of the abstractions from Section 2.6, it briefly explains how to concretize it and why. The details and correctness arguments for these points come in later sections.

Atomic steps. Whereas Brooklet executes firings one at a time, albeit in non-deterministic order, River executes them concurrently whenever it can guarantee that the end result is the same. This concurrency is crucial for performance. To guarantee the same end result, River uses a minimum of synchronization that keeps firings conceptually atomic. River shields the user from concerns of locking or memory models.

Pure functions. Both the calculus and the execution environment separate state from operators. However, whereas the calculus passes variables in and out of functions by copies, the execution environment uses pointers when the state is local to avoid the copying cost. For accessing distributed state, River calls the appropriate API used by the underlying runtime. Using the fact that state is explicit, River automates the appropriate locking discipline where necessary, thus relieving users from this burden. Furthermore, instead of returning data items to be pushed on output queues, functions in River directly invoke callbacks for the runtime library, thus avoiding copies and simplifying the function implementations.

Opaque functions. Functions for River are implemented in a traditional non-streaming language. They are separated from the River runtime library by a well-defined API. Because atomicity is preserved at the granularity of operator firings, River does not interfere with any local instruction-reordering optimizations that the low-level compiler or the hardware may want to perform.

Non-determinism. Being a virtual execution environment, River ultimately leaves the concrete schedule to the underlying platform. However, it reduces the flexibility for the scheduler by bounding queue sizes and by using back-pressure. Furthermore, it uses techniques to avoid deadlocks caused by waiting for available space in bounded queues.

No physical platform. River programs are target-platform independent. However, at deployment time, the optimizer takes target-platform characteristics into consideration for placement.

Finite execution. River applications run indefinitely and produce timely outputs along the way, fitting the purpose and intent of practical stream-processing applications.

3.2. River operator implementations

A River program consists of two parts: the stream graph and the operator implementations. For the stream graph, River simply reuses the topology language of Brooklet. For operators, on the other hand, River must go beyond Brooklet by supplying an implementation language. The primary requirements for this implementation language are that (i) the creation and decomposition of data items be convenient, to aid in operator implementation, and (ii) mutable state be easily identifiable, in keeping with the semantics. An explicit non-goal is a support for traditional compiler optimizations, which we leave to an off-the-shelf traditional compiler.

Typed functional languages clearly meet both requirements, and a lower-level traditional intermediate language (IL) such as the low-level virtual machine [12] can also meet them, given library support for higher-level language features such as pattern matching. In our current implementation, we rely on OCaml as River's implementation sublanguage. OCaml is a typed functional language that features a high-quality native code compiler and a simple foreign function interface (FFI), which facilitates integration with existing streaming runtimes written in C/C++.

3.3. Maximizing concurrency while upholding atomicity

This section gives the details for how River upholds the sequential semantics of Brooklet. In particular, River differs from Brooklet in how it handles state variables and data items pushed on output queues. These differences are motivated by performance goals: they avoid unnecessary copies and increase concurrency.

River requires that each operator instance is single-threaded and that individual queue push and pop operations are atomic. Additionally, if variables are shared between operator instances, each operator instance uses locks to enforce mutual exclusion. River's locking discipline follows established practice for deadlock prevention; that is, an operator instance first acquires all necessary locks in a standard order, then performs the actual work, and finally releases the locks in reverse order. Otherwise, River does not impose further ordering constraints. In particular, unless prevented by locks, operator instances may execute in parallel. They may also enqueue data items as they execute and update variables in place without differing in any observable way from Brooklet's call-by-value-result semantics. To explain how River's execution model achieves this, we first consider execution without shared state.

Operator instance firings without shared state behave as if atomic. In River, a downstream operator instance o_2 can fire on a data item, while the firing of the upstream operator instance o_1 that enqueued it is still in progress. The behavior is the same as if o_2 had waited for o_1 to complete before firing, because queues are one-to-one and queue operations are atomic. Furthermore, because each operator is single-threaded, there cannot be two firings of the same operator instance that are active simultaneously, so there are no race conditions on operator instance local state variables.

In the presence of shared state, River uses the lock assignment algorithm shown in Figure 2. The algorithm finds the minimal set of locks that covers the shared variables appropriately. The idea is that locks form equivalence classes over shared variables: every shared variable is protected by exactly one lock, and shared variables in the same equivalence class are protected by the same lock.

```
1.  $AllClasses.add(AllSharedVars)$
2.  **for all** $o \in OpInstances$ **do**
3.      $UsedByO = sharedVariablesUsedBy(o)$
4.      **for all** $v \in UsedByO$ **do**
5.          $EquivV = v.equivalenceClass$
6.          **if** $EquivV \not\subseteq UsedByO$ **then**
7.              $AllClasses.remove(EquivV)$
8.              $AllClasses.add(EquivV \cap UsedByO)$
9.              $v.equivalenceClass = EquivV \cap UsedByO$
10.             $AllClasses.add(EquivV \setminus UsedByO)$
```

Figure 2. Algorithm for assigning shared variables to equivalence classes, that is, locks.

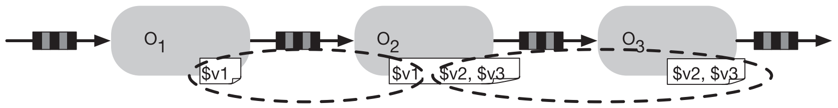


Figure 3. Variables are placed in equivalence classes, indicated by the dashed ovals.

Figure 3 illustrates the result of the assignment algorithm. In the example, there are three operators: o_1, o_2, and o_3. There are also three variables: $\$v_1$, $\$v_2$, and $\$v_3$. Operator o_1 uses variables $\$v_1$ and $\$v_2$; operator o_2 uses variables $\$v_1$, $\$v_2$, and $\$v_3$; and operator o_3 uses variables $\$v_2$ and $\$v_3$. In this example, River uses two locks for the three variables, indicated by the dashed ovals. Variable $\$v_1$ is in one equivalence class, while $\$v_2$ and $\$v_3$ are in a second equivalence class.

Two variables only have separate locks if there is an operator instance that uses one, but not the other. For example, in Figure 3, variables $\$v_2$ and $\$v_3$ remain in the same equivalence class, because no operator uses $\$v_2$, but not $\$v_3$ or vice versa. The algorithm starts with a single equivalence class (lock) containing all variables in line 1. The only way for variables to end up under different locks is by the split in lines 7–10. Without loss of generality, let v be in *EquivV* $\cap$ *UsedByO* and w be in *EquivV* $\setminus$ *UsedByO*. That means there is an operator instance o that uses *UsedByO*, which includes v but excludes w.

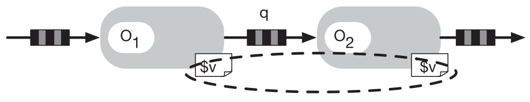
An operator instance only acquires locks for variables it actually uses. For example, in Figure 3, operator o_3 does not need to acquire the lock for variable $\$v_1$. Let us say operator instance o uses variable v, but not w. We need to show that v and w are under separate locks. If they are under the same lock, then the algorithm will arrive at a point where *UsedByO* contains v, but not w, and *EquivV* contains both v and w. That means that *EquivV* is not a subset of *UsedByO*, and lines 7–10 split it, with v and w in two separate parts of the split.

Shared state accesses behave as if atomic. An operator instance locks the equivalence classes of all the shared variables it accesses.

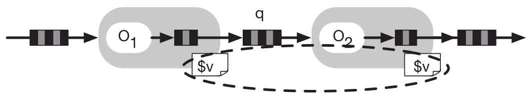
3.4. Bounding queue sizes

In Brooklet, communication queues are infinite, but real-world systems have limited buffer space, raising the question of how River should manage bounded queues. One option is to drop data items when queues are full. But this results in an unreliable communication model, which significantly complicates application development [13], wastes effort on data items that are dropped later on [14], and is inconsistent with Brooklet's semantics. A more attractive option is to automatically apply back-pressure through the operator graph.

A straightforward way to implement back-pressure, illustrated in Figure 4(a), is to let the enqueue operation performed by an operator instance block when the output queue is full. While easy to implement, this approach could deadlock in the presence of shared variables. To see this, consider an operator instance o_1 feeding another operator instance o_2 and assume that both operator instances access a common shared variable, $\$v$. Further assume that o_1 is blocked on an enqueue operation because of back-pressure. Because o_1 holds the lock on $\$v$ during its firing, o_2 cannot proceed,



(a) A deadlock-prone way to implement back-pressure.



(b) River back-pressure with intermediate buffer.

Figure 4. Back-pressure with and without an intermediate buffer.

```
1.  $process(dataItem, submitCallback, variables)$
2.      $lockSet = \{v.equivalenceClass()$ **for** $v \in variables\}$
3.      **for all** $lock \in lockSet.iterateInStandardOrder()$ **do**
4.          $lock.acquire()$
5.      $tmpBuffer = \bullet$
6.      $tmpCallback = \lambda d \Rightarrow tmpBuffer.push(d)$
7.      $opFire(dataItem, tmpCallback, variables)$
8.      **for all** $lock \in lockSet.iterateInReverseOrder()$ **do**
9.          $lock.release()$
10.     **while** $!tmpBuffer.isEmpty()$ **do**
11.         $submitCallback(tmpBuffer.pop())$
```

Figure 5. Algorithm for implementing back-pressure.

while o_1 is blocked on the enqueue operation. On the other hand, o_1 will not be able to unblock until o_2 makes progress to open up space in o_1s output queue. They are deadlocked.

Figure 4(b) illustrates River's implementation of back-pressure. The main difference is an addition of a dynamically sized intermediate buffer for each operator output. The pseudocode in Figure 5 presents River's solution to implementing back-pressure. It describes the *process* function, which is called by the underlying streaming runtime when data arrives at a River operator. The algorithm starts in line 2 with an operator's lock set. The lock set is the minimal set of locks needed to protect an operator's shared variables, as described in Section 3.3. Before an operator fires, it first must acquire all locks in its lock set, as shown in lines 3–4. Once all locks are held, the process function invokes the operator's *opFire* method, which contains the actual operator logic. The *opFire* does not directly enqueue its resultant data for transport by the runtime. Instead, it writes its results to a dynamically sized intermediate buffer, which is passed to *opFire* as a callback. Lines 6–7 show the callback and invocation of the operator logic. Next, lines 8–9 release all locks. Finally, lines 10–11 drain the temporary buffer, enqueuing each data item for transport by calling the streaming runtime's *submit* callback.

The key insight is that lines 10–11 might block if the downstream queue is full, but there is no deadlock because at this point, the algorithm has already released its shared-variable locks. Furthermore, *process* will only return after it has drained the temporary buffer, so it only requires enough space for a single firing. If *process* is blocked on a downstream queue, it may in turn block its own upstream queue. That is what is meant by back-pressure. The algorithm in Figure 5 restricts the scheduling of operator firings. In Brooklet, an operator instance can fire as long as there is at least one data item in one of its input queues. In River, an additional condition is that all intermediate buffers must be empty. This does not impact the semantics of the applications or the programming interface of the operators. It simply impacts the scheduling decisions of the runtime.

The algorithm described earlier avoids deadlocks in acyclic stream graphs. Many streaming languages, including CQL and Sawzall, do not allow cycles. So, for these languages, the earlier mechanism is sufficient. For languages that do allow cycles, there are various techniques to avoiding deadlocks that may be adopted for use with River. StreamIt, for example, relies on static information from the StreamIt source program to compute deadlock-free schedules [15]. River could use its annotations to pass this scheduling information to the target runtime. Other languages, such as Streams Processing Language (SPL), use runtime control ports [16] to break deadlocks at runtime, which is out of scope of the IL.

4. LANGUAGE MAPPINGS

Language designers have developed numerous domain-specific languages for stream processing. To explore the use of River as a target for streaming language translation, we have implemented translators for representative subsets of three prominent examples:

- CQL [1] is a member of the StreamSQL family of languages, which are popularly used for algorithmic trading. CQL extends Structured Query Language's (SQL) well-studied relational operators with a notion of windows over infinite streams of data and relies on classic query optimizations [1], such as moving a selection before a join. Just as SQL is based on the formal foundation of relational algebra (RA), CQL is based on a stream RA (SRA).
- Sawzall [4], a scripting language for Google's MapReduce [17] platform, is used for large-scale data analysis, such as processing web-server logs. The MapReduce framework streams data items through multiple copies of user-defined *map* operators and then aggregates the results through *reduce* operators on a cluster of workstations. We view Sawzall as a streaming language in the broader sense and address it in this paper to showcase the generality of our work.
- StreamIt [5], a synchronous data flow language with stream abstractions, has been used for signal processing applications, such as MPEG encoding and decoding [18]. The StreamIt compiler enforces static data transfer rates between user-defined operators with fixed topologies and improves performance through operator fusion, fission, and pipelining [19].

We chose these languages because they occupy three diverse points in the design space for streaming languages and nicely illustrate the generality of both the Brooklet calculus and the River IL.

4.1. *Language mappings overview*

For each of these three languages, we first provide a formal translation into Brooklet using sequent calculus notation. The translations share a common approach. Each translation exposes implicit state and communication as explicit variables and queues, respectively; exposes a mechanism for implementing global determinism on top of an inherently non-deterministic runtime; and reuses existing operator implementations by wrapping them with higher-order functions that statically bind the original function and dynamically adapt the runtime arguments (thus preserving small-step semantics).

For CQL, the translation to Brooklet has an additional step. We first translate CQL into SRA and then translate SRA to Brooklet. Much like RA provides the mathematical foundation for SQL, SRA provides the mathematical foundation for CQL, and the translation makes the operators in a query explicit. The formal semantics for CQL [20] are written in terms of SRA.

After describing the formalism, we address the pragmatics of translation to River. The River translations closely follow the Brooklet translations. Both make state and communication explicit. However, while Brooklet uses higher-order functions to wrap existing operator implementations, the River translators generate boiler-plate code that invokes the existing operator functions. To simplify the development of source languages, Brooklet builds on language composition techniques, allowing operator functions to be written with reusable grammars and templates. Because the River translations are based on Brooklet, a language implementor is able to reason about the correctness of the translations.

4.2. *CQL and SRA example*

Continuous Query Language is a member of the StreamSQL family of languages. StreamSQL gives developers who are familiar with SQL's select-from-where syntax an incremental learning path to

CQL example: Bargain finder.

```
quotes :=
  {string ticker, int ask} stream
history :=
  {string ticker, int low} relation
select istream(*)
  from quotes[now], history
  where quotes.ask <= history.low
  and quotes.ticker==history.ticker
```

CQL syntax:

decl	+= *id* `:` *strType* (= *query*)$^?$ `;`	(G-DECL)
τ	+= *strType*	
strType	::= `{` $\overline{\tau\ id}$ `}` `stream`	(G-STREAMTYPE)
select	+= `select` *r2sOp* `(` *selectList* `)`	(G-SELECTSTREAM)
r2sOp	::= `istream`	
	| `dstream`	
	| `rstream`	
fromItem	+= *id* `[` *s2rOp* `]`	(G-INIDWINDOW)
s2rOp	::= `now`	(G-NOWWINDOW)
	| `range` *int*	(G-TIMEWINDOW)
	| `rows` *int*	(G-COUNTWINDOW)

SQL syntax:

program	::= $\overline{decl}$	(G-PROGRAM)
decl	::= *id* `:` *relType* (= *query*)$^?$ `;`	(G-DECL)
τ	+= *relType*	
relType	::= `{` $\overline{id{:}\tau}$ `}` `relation`	(G-RELTYPE)
query	::= *select from where*$^?$	(G-QSELECT)
select	::= `select` *selectList*	(G-SELECTREL)
selectList	::= $\star$	(G-SELECT$\star$)
	| $\overline{projectItem}$	(G-PROJECTLIST)
	| $\overline{aggrItem}$	(G-PROJECTAGGR)
projectItem	::= *expr* `as` *id*	(G-OUTID)
	| *id* `.` *id*	(G-OUTATTRIB)
aggrItem	::= *aggr* `(` *id* `)`	(G-OUTAGGR)
aggr	::= `count`	(G-COUNT)
	| `avg`	(G-AVG)
	| `sum`	(G-SUM)
from	::= `from` $\overline{fromItem}$	(G-FROM)
fromItem	::= *id*	(G-INID)
where	::= `where` *expr*	(G-WHERE)

Embedded expression syntax:

expr	::= *expr* + *expr*	(G-PLUS)
	| *expr* == *expr*	(G-EQUALS)
	| *expr* `&&` *expr*	(G-AND)
	| `!` *expr*	(G-NEG)
	| *id*	(G-ID)
	| *expr* `(` $\overline{expr}$ `)`	(G-CALL)
	| *expr* `.` *id*	(G-ATTRIB)
	| *int*	(G-ILIT)
	| *string*	(G-SLIT)

Embedded type syntax:

τ	::= `unit`	(G-UNIT)
	| `bool`	(G-BOOL)
	| `int`	(G-ITYPE)
	| `string`	(G-STYPE)
	| `{` $\overline{id{:}\tau}$ `}`	(G-RECORD)
	| $\overline{\tau}$ `->` τ	(G-FUNCTION)

Figure 6. SQL, CQL, and embedded expression and type syntax.

stream programming. This paper uses CQL to represent the entire StreamSQL family, because it has a clean design, has made significant impact [1], and has a formal semantics [20].

The River implementation of CQL, River-CQL, differs from prior definitions in one respect. Previously published versions [1, 20] lacked a general-type system, and all example applications used a single implicit integer type. River-CQL adds a type system, which makes the language easier to use, for example, by reporting type errors at compile time. Moreover, because the types are preserved during translation, River-CQL avoids the time overhead of type inspection and conversion overheads, as well as the space overhead for wrapping.

4.2.1. CQL source code example. Figure 6 shows an example CQL query for a hypothetical algorithmic trading application. The program finds bargain quotes, whose ask price is lower than the historic low. The program has two inputs: a stream `quotes` and a time-varying relation `history`. The first two lines of the example declare the types, or schemas, of the two inputs. The stream `quotes` has two fields: the string `ticker` and integer `ask`. The relation `history` also has two fields: the string `ticker` and integer `low`. In CQL, a *time-varying relation* maps a timestamp to a bag of tuples, and each bag of tuples is called an *instantaneous relation*. In the example, input `history(ticker,low)` is the time-varying relation r_h:

$$r_h = [1 \mapsto \{\langle\text{‘IBM’}, 119\rangle, \langle\text{‘XYZ’}, 38\rangle\}, 2 \mapsto \{\langle\text{‘IBM’}, 119\rangle, \langle\text{‘XYZ’}, 35\rangle\}]$$

The instantaneous relation $r_h(1)$ is $\{\langle\text{‘IBM’}, 119\rangle, \langle\text{‘XYZ’}, 38\rangle\}$.

A *stream* is a bag of timestamp tuple pairs. An example is the stream s_q, which represents the input `quotes(ticker,ask)`:

$$s_q = \{\langle\langle\text{‘IBM’}, 119\rangle, 1\rangle, \langle\langle\text{‘IBM’}, 124\rangle, 1\rangle, \langle\langle\text{‘XYZ’}, 35\rangle, 2\rangle, \langle\langle\text{‘IBM’}, 119\rangle, 2\rangle\}$$

Streams and relations are transformed by operators. There are three classes of operators in CQL, as illustrated in Figure 7. A stream-to-relation (S2R) operator converts a stream into a relation. More informally, S2R operators represent windows of data on which operations can be performed. A relation-to-relation (R2R) operator performs the standard operations from SQL, including

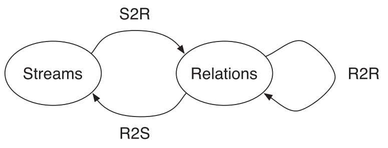


Figure 7. The three classes of operators in CQL.

SRA example: Bargain finder.

```
istream(
  bargainjoin(
    now(quotes),
    history))
```

SRA syntax:

P_a	$::=$	$P_{ar} \mid P_{as}$	*SRA program*
P_{ar}	$::=$		*(Relation query)*
		$RName$	*Relation name*
	$\mid$	$S2R(P_{as})$	*Stream to relation*
	$\mid$	$R2R(\overline{P_{ar}})$	*Relation to relation*
P_{as}	$::=$		*(Stream query)*
		$SName$	*Stream name*
	$\mid$	$R2S(P_{ar})$	*Relation to stream*
$RName \mid SName$	$::=$	id	*Input name*
$S2R \mid R2R \mid R2S$	$::=$	id	*Operator name*

Figure 8. Stream relational algebra (SRA) example query and syntax.

joins, projections, and selections. Finally, a relation-to-stream (R2S) operator converts a relation into a stream.

The subquery `quotes[Now]` uses the S2R operator `[Now]` to turn the `quotes` stream into a time-varying relation r_q:

$$r_q = [1 \mapsto \{\langle\text{'IBM'}, 119\rangle, \langle\text{'IBM'}, 124\rangle\}, 2 \mapsto \{\langle\text{'XYZ'}, 35\rangle, \langle\text{'IBM'}, 119\rangle\}]$$

The next step of the query applies the R2R operator, join, to the quote relation r_q with the history relation r_h into a bargains relation r_b. Note that this is a traditional join operator applied to one instantaneous relation at a time and therefore does not need to modify the timestamps:

$$r_b = [1 \mapsto \{\langle\text{'IBM'}, 119, 119\rangle\}, 2 \mapsto \{\langle\text{'XYZ'}, 35, 35\rangle, \langle\text{'IBM'}, 119, 119\rangle\}]$$

Finally, the R2S operator, `istream`, monitors insertions into relation r_b and emits them as output stream s_o of time-tagged tuples:

$$s_o = \{\langle\langle\text{'IBM'}, 119, 119\rangle, 1\rangle, \langle\langle\text{'XYZ'}, 35, 35\rangle, 2\rangle\}$$

As with SQL, the `*` symbol is used in the select-clause to indicate that the query returns all fields in the output stream or relation. The final result of the sample query is a stream with three fields of type: `string`, `int`, and `int`.

4.2.2. CQL to SRA translation example. While CQL uses select-from-where syntax, the CQL semantics use an equivalent SRA syntax (similar to RA in databases). The SRA translation of the CQL example query appears in Figure 8.

The algebraic notation makes the operator tree clearer. The leaves are stream name `quotes` and relation name `history`. SRA has the same three categories of operators. S2R operators turn a stream into a relation; for example, `Now(quotes)` turns stream `quotes` into relation r_q. R2R operators turn one or more relations into a new relation; for example, `BargainJoin(`r_q, r_h`)` turns relations r_q and r_h into the bargain relation r_b. Finally, R2S operators turn a relation into a stream; for example, `istream(`r_b`)` turns relation r_b into the stream of its insertions.

4.2.3. SRA to Brooklet translation example. Given the SRA example program in Figure 8, the translation to Brooklet is the program P_b:

```
output qo;
input quotes, history;
(qq, $vn)        ← wrapNow(quotes, $vn);
```

$(\mathtt{q_b}, \mathtt{\$v_q}, \mathtt{\$v_h}) \leftarrow \mathtt{wrapBargainJoin(q_q, history, \$v_q, \$v_h);}$
$(\mathtt{q_o}, \mathtt{\$v_o}) \leftarrow \mathtt{wrapIStream(q_b, \$v_o)}$

The leaves of the query tree serve as input queues; each subquery produces an intermediate queue, which the enclosing operator consumes; and the outermost query operator produces the program output queue. The translation to Brooklet makes the state of the operators explicit. The most interesting state is that of the `wrapBargainJoin` operator. Like each R2R operator, it has a function F_c(`BargainJoin`) that transforms one or more input instantaneous relations of the same timestamp to one output instantaneous relation. Brooklet models the choice of interleavings by allowing either queue $\mathtt{q_q}$ or `history` to fire independently. Hence, the Brooklet operator processes one data item each time in either queue fires. Assume a data item arrives on the first queue $\mathtt{q_q}$. If there is already a data item with the same timestamp in the variable $\mathtt{v_h}$ associated with the second queue, Brooklet performs the join, which may yield data items for the output queue $\mathtt{q_b}$. Otherwise, it simply stores the data item in $\mathtt{v_q}$ for later.

4.3. Economic language development

Having described one of the three source languages in detail, we now describe how River simplifies the task of language development. A language implementer who wants to create a new language translator needs to implement a parser, a type checker, and a code generator. We facilitate this task by decomposing each language into sublanguages and then reusing common sublanguage translator modules across languages. Principally, we follow the same approach as the Jeannie language [21], which composes C and Java. However, our work increases both the granularity of the components (by combining parts of languages) and the number of languages involved.

Modular parsers. The parsers use component grammars written as modules for the *Rats!* parser generator [22]. Each component grammar can either *modify* or *import* other grammar modules. For example, the CQL grammar consists of several modules: SQL's select-from-where clauses, streaming constructs modifying SQL to CQL, an imported expression sublanguage for operators like projection or selection, and an imported type sublanguage for schemas. The grammar modules for expressions and types are the same as in other River languages. The result of parsing is an abstract syntax tree (AST), which contains tree nodes drawn from each of the sublanguages.

Modular type checkers. Type checkers are also implemented in a compositional style. Type checkers for composite languages are written as groups of visitors. Each visitor is responsible for all AST nodes corresponding to a sublanguage. Each visitor can either dispatch to or inherit from other visitors, and all visitors share a common type representation and symbol table. For example, the CQL analyzer inherits from an SQL analyzer, which in turn dispatches to an analyzer for expressions and types. All three analyzers share a symbol table.
If there are type errors, the type analyzer reports those and exits. Otherwise, it populates the symbol table and decorates the AST with type annotations.

Modular code generators. The implementation language of the River IL allows language developers to write language-specific libraries of standard operators, such as select, project, split, join, and aggregate. However, the operator implementations need to be specialized for their concrete application. Consider, for example, an implementation for a selection operator:

```
Bag.filter (fun x -> #expr) inputs
```

where `#expr` stands for a predicate indicating the filter condition.

How best to support this specialization was an important design decision. One approach would be to rely on language support, that is, OCaml's support for generic functions and modules (i.e., *functors*) as reflected in the River IL. This approach is well understood and statically safe. But it also requires abstracting away any application-specific operations in callbacks, which can lead to unwieldy interfaces and performance overhead. Instead, we chose to implement common operators as IL templates, which are instantiated inline with appropriate types and expressions. Pattern

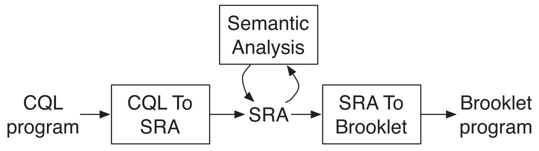


Figure 9. Overview of the complete translation from CQL to Brooklet.

variables (of form `#expr`) are replaced with concrete syntax at compile time. This eliminates the overhead of abstraction at the cost of code size. Note that the SPL language uses a similar approach [16].

The templates are actually parsed by grammars derived from the original language grammars. As a result, templates benefit from both the convenience of using concrete syntax and the robustness of static syntax checking. Code generation templates in River play the same role as currying in Brooklet; that is, they bind the function to its arguments.

Thus, code generation is also simplified by the use of language composition. The input to the code generator is the AST annotated with type information, and the output is a stream graph and a set of operator implementations. Our approach to producing this output is to first create the AST for the stream graph and each operator implementation and then pretty-print those ASTs. In the first step, we splice together subtrees obtained from the templates with subtrees obtained from the original source code. In the second step, we reuse pretty-printers that are shared across source language implementations. Overall, we found that the use of language composition led to a smaller, more consistent implementation with more reuse, making the changes to the source languages well worth it.

4.4. CQL to Brooklet

Figure 9 illustrates the steps for a complete translation from CQL to Brooklet. A CQL program is first translated to SRA, then the SRA program is type checked, and finally, the SRA program is translated to Brooklet. Each of these steps uses language composition to reduce the overall implementation effort.

4.4.1. CQL syntax. Figure 6 presents the formal syntax for the grammar modules used to develop the CQL parser: the embedded expression and type language, the core SQL language, and the streaming extensions to SQL. The expression language is fairly standard. We assume that the parser enforces the usual precedence and associativity rules in the usual way. In practice, there are more arithmetic and logic operators, omitted for brevity. Function calls are n-ary. Attribute access retrieves the value of an attribute from a record. The SQL grammar includes the standard select-from statement and optional where clause found in most language implementations. The use of composition is demonstrated in the CQL grammar, where the $+=$ symbol extends the standard Backus–Naur form notation to indicate that a grammar module adds more alternatives to a production. For example, CQL adds a stream declaration alternative to the *decl* production from SQL.

A CQL program P_c is a query that computes a stream or relation from other streams or relations. CQL code can declare both relations and streams and converts between them with R2S operators (*r2sOp*, e.g., `istream`) and S2R operators a.k.a. windows (*s2rOp*, e.g., `now`). Aside from the operators converting streams to relations and vice versa, all the ‘real’ work happens with the traditional relational operators. That is a major strength of CQL, because it leads to clear and simple semantics that are faithful to decades of relational database lore. Such fidelity is essential when evolving database applications to use streaming [23].

4.4.2. CQL typing. Figure 10 shows the typing rules for CQL. The rules are, again, arranged compositionally, with separate modules for expressions, SQL, and the CQL extensions to SQL.

The embedded expression type rules are standard. We provide them here for completeness, to make the SQL and CQL rules well founded. There is one rule for each alternative of each grammar

SQL typing:

$$\frac{\overline{id:\tau\ (=\ query)^?} = \overline{decl} \qquad n = |\overline{decl}| \\ \Gamma' = \{id_1:\tau_1,\ldots,id_n:\tau_n\} \\ \forall i \in 1\ldots n : \Gamma,\Gamma' \vdash decl_i : \texttt{unit}}{\Gamma \vdash \overline{decl}:\texttt{unit}} \quad (\textsc{Ty-Program})$$

$$\frac{\Gamma,\Gamma' \vdash query : \tau}{\Gamma,\Gamma' \vdash id:\tau\ (=\ query)^? : \texttt{unit}} \quad (\textsc{Ty-Decl})$$

$$\frac{\Gamma,\Gamma_B,\Gamma_\star \vdash select : \tau \\ \Gamma' \vdash from : \Gamma_B,\Gamma_\star \\ \Gamma\cup\Gamma_B\cup\Gamma_\star \vdash where : \texttt{unit}}{\Gamma,\Gamma' \vdash select\ from\ where^? : \tau} \quad (\textsc{Ty-QSelect})$$

$$\frac{\Gamma,\Gamma_B,\Gamma_\star \vdash selectList : \overline{id:\tau}}{\Gamma,\Gamma_B,\Gamma_\star \vdash \texttt{select}\ selectList : \{\overline{id:\tau}\}\ \texttt{relation}} \quad (\textsc{Ty-SelectRel})$$

$$\Gamma,\Gamma_B,\Gamma_\star \vdash \star : \Gamma_\star \quad (\textsc{Ty-Select}\star)$$

$$\frac{\Gamma'' = \Gamma\cup\Gamma_B\cup\Gamma_\star\cup\{\texttt{max}:\ldots,\texttt{count}:\ldots,\ldots\} \\ \forall i \in 1\ldots|\overline{projectItem}| : \Gamma'' \vdash out(projectItem_i) : (id_i : \tau_i)}{\Gamma,\Gamma_B,\Gamma_\star \vdash \overline{projectItem} : \overline{id : \tau}} \quad (\textsc{Ty-SelectList})$$

$$\frac{\Gamma \vdash id : \tau}{\Gamma \vdash out(id) : (id : \tau)} \quad (\textsc{Ty-OutID})$$

$$\frac{\Gamma \vdash id\,.\,id' : \tau}{\Gamma \vdash out(id\,.\,id') : (id' : \tau)} \quad (\textsc{Ty-OutAttrib})$$

$$\frac{\Gamma \vdash expr : \tau\texttt{->}\tau' \qquad \Gamma \vdash id : \tau}{\Gamma \vdash out(expr\,(id)\,) : (id : \tau')} \quad (\textsc{Ty-OutAggreg})$$

$$\frac{\Gamma \vdash expr : \tau}{\Gamma \vdash out(expr\ \texttt{as}\ id) : (id : \tau)} \quad (\textsc{Ty-OutAlias})$$

$$\frac{n = |\overline{fromItem}| \\ \forall i \in 1\ldots n : \Gamma \vdash in(fromItem_i) : \Gamma_{Bi},\Gamma_{\star i} \\ \forall i,j \in 1\ldots n : (\Gamma_{\star i} \vdash id : \tau) \wedge (\Gamma_{\star j} \vdash id : \tau') \Rightarrow \tau = \tau'}{\Gamma \vdash \texttt{from}\ \overline{fromItem} : \Gamma_{B1}\cup\ldots\cup\Gamma_{Bn},\Gamma_{\star 1}\cup\ldots\cup\Gamma_{\star n}} \quad (\textsc{Ty-From})$$

$$\frac{\Gamma \vdash id : \{\overline{id : \tau}\}\ \texttt{relation}}{\Gamma \vdash in(id) : (id : \{\overline{id : \tau}\}),\overline{\overline{id : \tau}}} \quad (\textsc{Ty-InID})$$

$$\frac{\Gamma \vdash expr : \texttt{bool}}{\Gamma \vdash \texttt{where}\ expr : \texttt{unit}} \quad (\textsc{Ty-Where})$$

Expression typing:

$$\frac{\Gamma \vdash expr : \tau \qquad \Gamma \vdash expr' : \tau}{\Gamma \vdash expr\ \texttt{+}\ expr' : \tau} \quad (\textsc{Ty-Plus})$$

$$\frac{\Gamma \vdash expr : \tau \qquad \Gamma \vdash expr' : \tau}{\Gamma \vdash expr\ \texttt{==}\ expr' : \texttt{bool}} \quad (\textsc{Ty-Equals})$$

$$\frac{\Gamma \vdash expr : \texttt{bool} \qquad \Gamma \vdash expr' : \texttt{bool}}{\Gamma \vdash expr\ \texttt{\&\&}\ expr' : \texttt{bool}} \quad (\textsc{Ty-And})$$

$$\frac{\Gamma \vdash expr : \texttt{bool}}{\Gamma \vdash \texttt{!}\,expr : \texttt{bool}} \quad (\textsc{Ty-Neg})$$

$$\frac{id : \tau \in \Gamma}{\Gamma \vdash id : \tau} \quad (\textsc{Ty-Id})$$

$$\frac{\Gamma \vdash expr : \overline{\tau}'\texttt{->}\tau \qquad \Gamma \vdash \overline{expr}' : \overline{\tau}'}{\Gamma \vdash expr\,(\overline{expr}'\,)\ : \tau} \quad (\textsc{Ty-Call})$$

$$\frac{\Gamma \vdash expr : \{\overline{id : \tau}\}}{\Gamma \vdash expr\,.\,id_i : \tau_i} \quad (\textsc{Ty-Attrib})$$

$$\Gamma \vdash int : \texttt{int} \quad (\textsc{Ty-ILit})$$

$$\Gamma \vdash STRING\_LIT : \texttt{string} \quad (\textsc{Ty-SLit})$$

CQL typing:

$$\frac{\Gamma,\Gamma_B,\Gamma_\star \vdash selectList : \overline{id:\tau}}{\Gamma,\Gamma_B,\Gamma_\star \vdash \texttt{select}\ r2sOp\,(selectList) : \{\overline{id:\tau}\}\ \texttt{stream}} \quad (\textsc{Ty-SelectStr})$$

$$\frac{\Gamma,\Gamma_B,\Gamma_\star \vdash selectList : \overline{id:\tau}}{\Gamma,\Gamma_B,\Gamma_\star \vdash \texttt{select}\ selectList : \{\overline{id:\tau}\}\ \texttt{stream}} \quad (\textsc{Ty-SelectImplicitIStream})$$

$$\frac{\Gamma \vdash id : \{\overline{id : \tau}\}\ \texttt{stream} \qquad \Gamma \vdash s2rOp : \texttt{unit}}{\Gamma \vdash in(id\,[\,s2rOp\,]\,) : (id : \{\overline{id : \tau}\}),\overline{id : \tau}} \quad (\textsc{Ty-InIDWindow})$$

$$\frac{\Gamma \vdash id : \{\overline{id : \tau}\}\ \texttt{stream}}{\Gamma \vdash in(id) : (id : \{\overline{id : \tau}\}),\overline{\overline{id : \tau}}} \quad (\textsc{Ty-InIDImplicitNow})$$

$$\frac{\Gamma \vdash id : \{\overline{id : \tau}\}\ \texttt{stream}}{\Gamma \vdash in(id\ \texttt{as}\ id') : (id' : \{\overline{id : \tau}\}),\overline{\overline{id : \tau}}} \quad (\textsc{Ty-InIDAsImplicitNow})$$

$$\Gamma \vdash \texttt{now} : \texttt{unit} \quad (\textsc{Ty-NowWindow})$$

$$\Gamma \vdash \texttt{range}\ int : \texttt{unit} \quad (\textsc{Ty-TimeWindow})$$

$$\Gamma \vdash \texttt{rows}\ int : \texttt{unit} \quad (\textsc{Ty-CountWindow})$$

Figure 10. Typing rules for SQL, CQL, and embedded expressions.

production, in the same order. The more interesting rules are T-ID, T-CALL, and T-ATTRIB, because they populate and access the type environment.

The SQL-type inference and type-checking rules are more complicated. The rules maintain four different type environments (which correspond to scopes in a compiler that uses a symbol table): in rule T-PROGRAM, Γ is the top-level environment with pre-defined functions, and Γ' is the environment for relation names in this program. Furthermore, in rule T-QUERYSELECT, Γ_B is the environment with base relation names from the from-clause, whereas $\Gamma_\star$ is the environment with attributes of those relations, which constitute the result of a $\star$ in the select-clause.

The CQL types use the same four type environments. CQL extends the SQL rules with R2S and S2R operators, as well as a rule for type-checking stream declarations.

4.4.3. SRA syntax. The complete SRA grammar is in Figure 8. An SRA program P_c can be either a relation query P_{cr} or a stream query P_{cs}, and queries are either simple identifiers RName or SName, or composed using operators from the categories S2R, R2R, or R2S. As a reminder, Figure 7 shows the three kinds of operators. An example of an R2R operator is a projection or a join. An example of an S2R operator is the `Now` window. An example of an R2S operator is `istream`.

Relational Algebra typing:

$$\frac{\begin{array}{c}\overline{id\!:\!\tau\ (=\ query)^{?}} = \overline{decl} \qquad n = |decl| \\ \Gamma_B = \{id_1\!:\!\tau_1,\ldots,id_n\!:\!\tau_n\} \\ \forall i \in 1\ldots n : (\Gamma,\Gamma_B) \vdash decl_i : \texttt{unit}\end{array}}{\Gamma \vdash \overline{decl}\!:\!\texttt{unit}} \quad (\textsc{Ty-Program})$$

$$\frac{\Gamma,\Gamma_B \vdash query : (id' : \tau)}{\Gamma,\Gamma_B \vdash id\!:\!\tau\ (=\ query)^{?} : \texttt{unit}} \quad (\textsc{Ty-Decl})$$

$$\frac{(id : \tau) \in \Gamma_B}{\Gamma,\Gamma_B : id : (id : \tau)} \quad (\textsc{Ty-InID})$$

$$\frac{\begin{array}{c}\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{relation}) \\ \Gamma, \overline{id : \tau} \vdash expr : \texttt{bool}\end{array}}{\Gamma,\Gamma_B \vdash \sigma\,(expr)\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-Select})$$

$$\frac{\begin{array}{c}\Gamma,\Gamma_B \vdash query : (id : \{\overline{id' : \tau'}\}\ \texttt{relation}) \\ \forall i \in 1\ldots|\overline{expr\ \texttt{as}\ id}| : \Gamma, \overline{id' : \tau'} \vdash expr_i : \tau_i\end{array}}{\Gamma,\Gamma_B \vdash \pi\,(\overline{expr\ \texttt{as}\ id})\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-Project})$$

$$\frac{\begin{array}{c}\Gamma,\Gamma_B \vdash query : (id : \{\overline{id'' : \tau''}\}\ \texttt{relation}) \\ \forall i \in 1\ldots|\overline{id}| : \overline{id'' : \tau''} \vdash id_i : \tau_i \\ \forall i \in 1\ldots|\overline{expr\ \texttt{as}\ id'}| : \Gamma,\Gamma_A, \overline{id'' : \tau''} \vdash expr_i : \tau'_i\end{array}}{\begin{array}{c}\Gamma,\Gamma_B \vdash \gamma\,(\overline{id}, \overline{expr\ \texttt{as}\ id'})\,(query) : \\ (id : \{\overline{id : \tau}, \overline{id' : \tau'}\}\ \texttt{relation})\end{array}} \quad (\textsc{Ty-Aggregate})$$

$$\frac{\begin{array}{c}\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{relation}) \\ \Gamma,\Gamma_B \vdash query' : (id' : \{\overline{id' : \tau'}\}\ \texttt{relation}) \\ \Gamma, id : \{\overline{id : \tau}\}, id' : \{\overline{id' : \tau'}\} \vdash expr : \texttt{bool} \\ \forall i \in 1\ldots|\overline{id : \tau}| : \forall j \in 1\ldots|\overline{id' : \tau'}| : id_i = id'_j \Rightarrow \tau_i = \tau'_j\end{array}}{\begin{array}{c}\Gamma,\Gamma_B \vdash\!\bowtie (expr)\,(query, query')\ \texttt{as}\ id'' : \\ (id'' : \{\overline{id : \tau}, \overline{id' : \tau'}\}\ \texttt{relation})\end{array}} \quad (\textsc{Ty-Join})$$

Stream Relational Algebra typing:

$$\frac{\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{relation})}{\Gamma,\Gamma_B \vdash r2sOp\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{stream})} \quad (\textsc{Ty-R2SOp})$$

$$\frac{\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{stream})}{\Gamma,\Gamma_B \vdash \texttt{now}\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-NowWindow})$$

$$\frac{\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{stream})}{\Gamma,\Gamma_B \vdash \texttt{range}[int]\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-TimeWindow})$$

$$\frac{\Gamma,\Gamma_B \vdash query : (id : \{\overline{id : \tau}\}\ \texttt{stream})}{\Gamma,\Gamma_B \vdash \texttt{rows}[int]\,(query) : (id : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-CountWindow})$$

$$\frac{\begin{array}{c}\Gamma,\Gamma_B \vdash query : (id' : \{\overline{id : \tau}\}\ \texttt{stream}) \\ \overline{id : \tau} \vdash id : \tau\end{array}}{\Gamma,\Gamma_B \vdash \texttt{rows}[id, int]\,(query) : (id' : \{\overline{id : \tau}\}\ \texttt{relation})} \quad (\textsc{Ty-PartitionWindow})$$

Figure 11. Typing rules for relational and stream relational algebras.

4.4.4. SRA typing. Figure 11 shows the typing rules for SRA. Just as the CQL-type rules extend the SQL rules, the SRA rules extend the RA rules. We use the standard symbols for RA operators: select σ, project π, aggregate γ, and join $\bowtie$. The SRA typing rules use two type environments: Γ is the top-level environment with pre-defined functions and Γ_B is the environment with base relation names.

4.4.5. CQL to SRA. Figure 12 shows the formal translation from CQL to SRA and the translation from SQL to RA. Both algebras (SRA and RA) represent programs as a tree of operators, which enables the syntax-driven translation strategy employed by the rules. Each rule always returns a valid subprogram and function environment. Rules at the top of the grammar hierarchy translate one part of the program (usually by adding a new operator to the root of the tree) and invoke rules lower in the hierarchy to provide the subtree.

These rules present basic translations for CQL and SQL. In practice, streaming systems and databases optimize the query plan according to standard algebra transformation rules that can be found in database textbooks [24].

4.4.6. SRA to Brooklet. Figure 13 shows the translation from SRA to Brooklet by recursion over the input program. Besides building up a program, the translation also builds up a function environment, which it populates with wrappers for the original functions. The translation introduces state, which the Brooklet wrappers maintain and consult to hand the right input to the wrapped SRA functions. Working in concert, the rules enforce a global convention: the execution sends exactly one instantaneous relation on every queue at every timestamp. Operators retain historical data in variables, for example, to implement windows.

SQL to RA: $\llbracket P_c \rrbracket_c^p = \langle F_a, P_a \rangle$

$$\frac{F_a = \emptyset}{\llbracket \mathit{id} \rrbracket_c^p = F_a, \mathit{id}} \quad (\text{T}_c^p\text{-Relation})$$

$$\frac{\begin{array}{c} n = |\overline{\mathit{fromItem}}| \\ \forall i \in 1 \ldots n : F_i, P_i = \llbracket \mathit{fromItem}_i \rrbracket_c^p \\ P_a = P_1 \times \ldots \times P_n \\ F_a = F_1 \cup \ldots \cup F_n \end{array}}{\llbracket \texttt{from}\ \overline{\mathit{fromItem}} \rrbracket_c^p = F_a, P_a} \quad (\text{T}_c^p\text{-From})$$

$$\frac{\begin{array}{c} F_a, P_a = \llbracket \mathit{from} \rrbracket \\ \mathit{id} = \mathit{freshId}() \\ F'_a = [\mathit{id} \mapsto \sigma(\mathit{expr})]F_a \\ P'_a = \mathit{id}\ (P_a) \end{array}}{\llbracket \mathit{from}\ \mathit{where} \rrbracket_c^p = F'_a, P'_a} \quad (\text{T}_c^p\text{-Where})$$

$$\frac{F_a, P_a = \llbracket \mathit{from}\ \mathit{where} \rrbracket_c^p}{\llbracket \texttt{select} * \mathit{from}\ \mathit{where} \rrbracket_c^p = F_a, P_a} \quad (\text{T}_c^p\text{-Select-*})$$

$$\frac{\begin{array}{c} F_a, P_a = \llbracket \mathit{from}\ \mathit{where} \rrbracket_c^p \\ \mathit{id} = \mathit{freshId}() \\ F'_a = [\mathit{id} \mapsto \pi(\overline{\mathit{projectItem}})]F_a \\ P'_a = \mathit{id}\ (P_a) \end{array}}{\llbracket \texttt{select}\ (\overline{\mathit{projectItem}})\ \mathit{from}\ \mathit{where} \rrbracket_c^p = F'_a, P'_a} \quad (\text{T}_c^p\text{-Select-Item})$$

$$\frac{\begin{array}{c} F_a, P_a = \llbracket \mathit{from}\ \mathit{where} \rrbracket_c^p \\ \mathit{id} = \mathit{freshId}() \\ F'_a = [\mathit{id} \mapsto \gamma(\overline{\mathit{aggrItem}})]F_a \\ P'_a = \mathit{id}\ (P_a) \end{array}}{\llbracket \texttt{select}\ (\overline{\mathit{aggrItem}})\ \mathit{from}\ \mathit{where} \rrbracket_c^p = F'_a, P'_a} \quad (\text{T}_c^p\text{-Select-Aggr})$$

CQL to SRA:

$$\frac{\begin{array}{c} \mathit{id}' = \mathit{freshId}() \\ F_a = [\mathit{id}' \mapsto \mathit{now}] \\ P_a = \mathit{id}'\ (\mathit{id}) \end{array}}{\llbracket \mathit{id}\,[\texttt{now}] \rrbracket_c^p = F_a, P_a} \quad (\text{T}_c^p\text{-NowWindow})$$

$$\frac{\begin{array}{c} \mathit{id}' = \mathit{freshId}() \\ F_a = [\mathit{id}' \mapsto \mathit{range}(\mathit{int})] \\ P_a = \mathit{id}'\ (\mathit{id}) \end{array}}{\llbracket \mathit{id}\,[\texttt{range}\ \mathit{int}] \rrbracket_c^p = F_a, P_a} \quad (\text{T}_c^p\text{-TimeWindow})$$

$$\frac{\begin{array}{c} \mathit{id}' = \mathit{freshId}() \\ F_a = [\mathit{id}' \mapsto \mathit{rows}(\mathit{int})] \\ P_a = \mathit{id}'\ (\mathit{id}) \end{array}}{\llbracket \mathit{id}\,[\texttt{rows}\ \mathit{int}] \rrbracket_c^p = F_a, P_a} \quad (\text{T}_c^p\text{-CountWindow})$$

$$\frac{\begin{array}{c} F_a, P_a = \llbracket \texttt{select}\ (\mathit{selectList})\ \mathit{from}\ \mathit{where} \rrbracket_c^p \\ \mathit{id} = \mathit{freshId}() \\ F'_a = [\mathit{id} \mapsto \mathit{r2sOp}]F_a \\ P'_a = \mathit{r2sOp}\ (P_a) \end{array}}{\llbracket \texttt{select}\ \mathit{r2sOp}\ (\mathit{selectList})\ \mathit{from}\ \mathit{where} \rrbracket_c^p = F'_a, P'_a} \quad (\text{T}_c^p\text{-Select-R2S})$$

Figure 12. Translation of CQL to the stream relational algebra (SRA).

SRA domains:

$\tau \in \mathcal{T}$	*Time*
$e \in \mathcal{TP}$	*Tuple*
$\sigma \in \Sigma = \text{bag}(\mathcal{TP})$	*Instantaneous relation*
$r \in \mathcal{R} = \mathcal{T} \to \Sigma$	*Time-varying relation*
$s \in \mathcal{S} = \text{bag}(\mathcal{TP} \times \mathcal{T})$	*Time-varying stream*

SRA program translation: $\llbracket F_a, P_a \rrbracket_a^p = \langle F_b, P_b \rangle$

$$\llbracket F_a, \mathit{SName} \rrbracket_a^p = \emptyset, \texttt{output}\,\mathit{SName}\texttt{;}\,\texttt{input}\,\mathit{SName}\texttt{;}\,\bullet \quad (\text{T}_a^p\text{-SName})$$

$$\llbracket F_a, \mathit{RName} \rrbracket_a^p = \emptyset, \texttt{output}\,\mathit{RName}\texttt{;}\,\texttt{input}\,\mathit{RName}\texttt{;}\,\bullet \quad (\text{T}_a^p\text{-RName})$$

$$\frac{\begin{array}{c} F_b, \texttt{output}\ q_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op} = \llbracket F_a, P_{as} \rrbracket_a^p \\ q'_o = \mathit{freshId}() \qquad v = \mathit{freshId}() \\ F'_b = [\mathit{S2R} \mapsto \mathit{wrapS2R}(F_a(\mathit{S2R}))]F_b \\ \overline{op}' = \overline{op},\ (q'_o, v) \leftarrow \mathit{S2R}\,(q_o, v)\,\texttt{;} \end{array}}{\llbracket F_a, \mathit{S2R}(P_{as}) \rrbracket_a^p = F'_b, \texttt{output}\ q'_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op}'} \quad (\text{T}_a^p\text{-S2R})$$

$$\frac{\begin{array}{c} F_b, \texttt{output}\ q_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op} = \llbracket F_a, P_{ar} \rrbracket_a^p \\ q'_o = \mathit{freshId}() \qquad v = \mathit{freshId}() \\ F'_b = [\mathit{R2S} \mapsto \mathit{wrapR2S}(F_a(\mathit{R2S}))]F_b \\ \overline{op}' = \overline{op},\ (q'_o, v) \leftarrow \mathit{R2S}\,(q_o, v)\,\texttt{;} \end{array}}{\llbracket F_a, \mathit{R2S}(P_{ar}) \rrbracket_a^p = F'_b, \texttt{output}\ q'_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op}'} \quad (\text{T}_a^p\text{-R2S})$$

$$\frac{\begin{array}{c} \overline{F_b, \texttt{output}\ q_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op}} = \overline{\llbracket F_a, P_{ar} \rrbracket_a^p} \\ n = |\overline{P_{ar}}| \qquad q'_o = \mathit{freshId}() \qquad \overline{q}' = \overline{q}_1, \ldots, \overline{q}_n \\ \forall i \in 1 \ldots n : v_i = \mathit{freshId}() \qquad \overline{op}' = \overline{op}_1, \ldots, \overline{op}_n \\ F'_b = [\mathit{R2R} \mapsto \mathit{wrapR2R}(F_a(\mathit{R2R}))](\cup \overline{F_b}) \\ \overline{op}'' = \overline{op}',\ (q'_o, \overline{v}) \leftarrow \mathit{R2R}\,(\overline{q_o}, \overline{v})\,\texttt{;} \end{array}}{\llbracket F_a, \mathit{R2R}(\overline{P_{ar}}) \rrbracket_a^p = F'_b, \texttt{output}\ q'_o\texttt{;}\,\texttt{input}\ \overline{q}'\texttt{;}\,\overline{op}''} \quad (\text{T}_a^p\text{-R2R})$$

SRA operator signatures:

$$\begin{array}{l} \text{S2R} : \mathcal{S} \times \mathcal{T} \to \Sigma \\ \text{R2S} : \Sigma \times \Sigma \to \Sigma \\ \text{R2R} : \Sigma^n \to \Sigma \end{array}$$

SRA operator wrapper signatures:

$$\begin{array}{l} \text{S2R} : (\Sigma \times \mathcal{T}) \times \{1\} \times \mathcal{S} \to (\Sigma \times \mathcal{T}) \times \mathcal{S} \\ \text{R2S} : (\Sigma \times \mathcal{T}) \times \{1\} \times \Sigma \to (\Sigma \times \mathcal{T}) \times \Sigma \\ \text{R2R} : (\Sigma \times \mathcal{T}) \times \{1 \ldots n\} \times (2^{\Sigma \times \mathcal{T}})^n \\ \qquad \to (\Sigma \times \mathcal{T}) \times (2^{\Sigma \times \mathcal{T}})^n \end{array}$$

SRA operator wrappers:

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad s = d_v \\ s' = s \cup \{\langle e, \tau \rangle : e \in \sigma\} \qquad \sigma' = f(s', \tau) \end{array}}{\mathit{wrapS2R}(f)(d_q, \_, d_v) = \langle \sigma', \tau \rangle, s'} \quad (\text{W}_a\text{-S2R})$$

$$\frac{\sigma, \tau = d_q \qquad \sigma' = d_v \qquad \sigma'' = f(\sigma, \sigma')}{\mathit{wrapR2S}(f)(d_q, \_, d_v) = \langle \sigma'', \tau \rangle, \sigma} \quad (\text{W}_a\text{-R2S})$$

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad d'_i = d_i \cup \{\langle \sigma, \tau \rangle\} \\ \forall j \neq i \in 1 \ldots n : d'_j = d_j \\ \exists j \in 1 \ldots n : \nexists \sigma : \langle \sigma, \tau \rangle \in d_j \end{array}}{\mathit{wrapR2R}(f)(d_q, i, \overline{d}) = \bullet, \overline{d}'} \quad (\text{W}_a\text{-R2R-Wait})$$

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad d'_i = d_i \cup \{\langle \sigma, \tau \rangle\} \\ \forall j \neq i \in 1 \ldots n : d'_j = d_j \\ \forall j \in 1 \ldots n : \sigma_j = \mathit{aux}(d_j, \tau) \end{array}}{\mathit{wrapR2R}(f)(d_q, i, \overline{d}) = \langle f(\overline{\sigma}), \tau \rangle, \overline{d}'} \quad (\text{W}_a\text{-R2R-Ready})$$

$$\frac{\langle \sigma, \tau \rangle \in d}{\mathit{aux}(d, \tau) = \sigma} \quad (\text{W}_a\text{-R2R-Aux})$$

Figure 13. Stream relational algebra (SRA) for CQL semantics on Brooklet.

SRA state. SRA represents global state explicitly as named relations, such as the `history` relation from our running example. But in addition, all three kinds of SRA operators implicitly maintain local state, referred to as 'synopses' in [1]. An S2R operator maintains the state of a window on a stream to produce a relation. Note that in the S2R operator wrappers rules in Figure 13, window state continuously increases. Although the Brooklet rules are semantically correct, the River implementation provides operators that are written to reclaim memory as early as possible. An R2S operator stores the previous state of the relation to compute the stream of differences. Finally, an R2R operator uses state to buffer data from whichever relation is available first, so it can be retrieved later to compute an output when data with matching timestamps is available for all relations.

SRA non-determinism. SRA is deterministic in the sense that the output of a program is fully determined by the times and values of its inputs [20]. Although a program can have independent inputs, for example, from a customer and from a stock exchange, any timing ambiguities outside the language are resolved by adding unambiguous timestamps. An SRA implementation might either assign timestamps upon receiving data or use timestamps that are an inherent part of the input data, such as trading times. However, SRA implementations can permit non-determinism to exploit parallelism. For example, the implementation needs not fully determine the order in which operators `Now` and `BargainJoin` process their data in `BargainJoin(Now(quotes), history)`. They can run in parallel as long as `BargainJoin` always waits for its two inputs to have the same timestamp.

Translation to Brooklet makes all state explicit and clarifies how the implementation enforces determinism. Arasu and Widom specify big-step denotational semantics for SRA [20]. We show how to translate SRA to Brooklet, thus giving an alternative semantics. As we will show in Section 4.7, both semantics define equivalent input/output behavior for SRA programs. Translations from other languages can use similar techniques, that is, make state explicit as variables, wrap computation in small-step firing functions, and define a global convention on how to achieve determinism.

Sawzall syntax:

P_z	$::= \overline{out}\ in\ \overline{emit}$	*Sawzall program*
out	$::= t$ `:` `table` f`;`	*Output aggregator*
in	$::= q$ `:` `input;`	*Input declaration*
emit	$::=$ `emit` t`[`f`(`q`)]` $\leftarrow$ f`(`q`);`	*Emit statement*
q	$::= id$	*Queue name*
f	$::= id$	*Function name*
t	$::= id$	*Table name*

Sawzall example: Query log analyzer.

```
queryOrigins : table sum;
queryTargets : table sum;
logRecord : input;
emit queryOrigins[getOrigin(logRecord)]⟵1;
emit queryTargets[getTarget(logRecord)]⟵1;
```

Sawzall program translation: $[\![F_z, P_z, R]\!]^p_z = \langle F_b, P_b\rangle$

$$\frac{\begin{array}{c}\overline{out}, q_{in}\texttt{ : input;}, \overline{emit} = P_z\\ \forall i \in 1\ldots R : q_i = \mathit{freshId}()\\ \forall i \in 1\ldots R : v_i = \mathit{freshId}()\\ f_{\texttt{Map}} = \mathit{wrapMap}(F_z, \overline{emit}, R)\\ f_{\texttt{Reduce}} = \mathit{wrapReduce}(F_z, \overline{out})\\ F_b = [\texttt{Map} \mapsto f_{\texttt{Map}},\ \texttt{Reduce} \mapsto f_{\texttt{Reduce}}]\\ op_m = \texttt{(}\overline{q}\texttt{)} \leftarrow \texttt{Map(}q_{in}\texttt{);}\\ \forall i \in 1\ldots R : op_i = \texttt{(}v_i\texttt{)} \leftarrow \texttt{Reduce(}q_i\texttt{,}v_i\texttt{);}\\ \overline{op}' = op_m, \overline{op}\end{array}}{[\![F_z, P_z, R]\!]^p_z = F_b, \texttt{output }\bullet\texttt{;input }q_{in}\texttt{;}\overline{op}'}\ (\mathrm{T}^p_z)$$

Sawzall domains:

$k_1 \in \mathcal{K}_1$	*Input key*	$k_2 \in \mathcal{K}_2$	*Output key*
$x_1 \in \mathcal{X}_1$	*Input value*	$x_2 \in \mathcal{X}_2$	*Output value*
$t \in \mathcal{T}$	*Aggregate name*	$O_z \in \mathcal{K}_2 \rightarrow \mathcal{X}_2$	*Output table*

Sawzall operator signatures:

$f_k : \mathcal{K}_1 \times \mathcal{X}_1 \rightarrow \mathcal{K}_2$	$f_x : \mathcal{K}_1 \times \mathcal{X}_1 \rightarrow \mathcal{X}_2^*$
$f_a : \mathcal{X}_2 \times \mathcal{X}_2 \rightarrow \mathcal{X}_2$	

Sawzall operator wrapper signatures:

Map $: (\mathcal{K}_1 \times \mathcal{X}_1) \times \{1\} \rightarrow (\mathcal{T} \times \mathcal{K}_2 \times \mathcal{X}_2)^*$
Reduce: $(\mathcal{T} \times \mathcal{K}_2 \times \mathcal{X}_2) \times \{1\} \times O_z \rightarrow O_z$

Sawzall operator wrappers:

$$\frac{\begin{array}{c}\texttt{emit }t\texttt{[}f_k\texttt{(\_)]} \leftarrow f_x\texttt{(\_);} = emit\\ \overline{b} = \mathit{wrapMap}(F_z, \overline{emit}, R)(d, 1)\\ k_1, x_1 = d \qquad k_2 = F_z(f_k)(k_1, x_1)\\ \overline{x_2} = F_z(f_x)(k_1, x_1) \qquad i = \mathit{hash}(k_2) \bmod R\\ b'_i = b_i, \langle t, k_2, x_{2_1}\rangle, \ldots, \langle t, k_2, x_{2_n}\rangle\\ \forall j \neq i \in 1\ldots R : b'_j = b_j\end{array}}{\mathit{wrapMap}(F_z, (emit, \overline{emit}), R)(d, \_) = \overline{b}'}\ (\mathrm{W}_z\text{-}\textsc{Map})$$

$$\frac{\forall i \in 1\ldots R : b_i = \bullet}{\mathit{wrapMap}(F_z, \bullet, R)(\_, \_) = \overline{b}}\ (\mathrm{W}_z\text{-}\textsc{Map}\text{-}\bullet)$$

$$\frac{\begin{array}{c}t, k_2, x_2 = d_q \qquad t\texttt{ : table }f_a\texttt{[];} \in \overline{out}\\ k_2 \in d_v \qquad x'_2 = F_z(f_a)(x_2, d_v(k_2))\\ d'_v = [k_2 \mapsto x'_2]d_v\end{array}}{\mathit{wrapReduce}(F_z, \overline{out})(d_q, \_, d_v) = d'_v}\ (\mathrm{W}_z\text{-}\textsc{Reduce})$$

$$\frac{\begin{array}{c}t, k_2, x_2 = d_q \qquad t\texttt{ : table }f_a\texttt{[];} \in \overline{out}\\ k_2 \notin d_v \qquad d'_v = [k_2 \mapsto x_2]d_v\end{array}}{\mathit{wrapReduce}(F_z, \overline{out})(d_q, \_, d_v) = d'_v}\ (\mathrm{W}_z\text{-}\textsc{Reduce}\text{-}\emptyset)$$

Figure 14. Sawzall semantics on Brooklet.

4.5. *Sawzall to Brooklet*

Sawzall [4] is a scripting language for MapReduce [17], which exploits cluster of workstations to analyze a massive but finite sequence of key/value pairs streamed from disk. In Sawzall, a stateless *map* operator transforms data one key/value pair at a time, feeding into a stateful *reduce* operator. The reduce operator works on separate keys separately, incrementally aggregating all values for a key into a single value. Although Sawzall programs are batch jobs, they use incremental operators to process large quantities of data in a single pass, and we therefore consider it a streaming language. Our translation provides the first formal semantics for Sawzall.

The example Sawzall program in Figure 14 is based on a similar example in [4]. The program analyzes a query log to count queries per latitude and longitude, which can then be plotted on a world map. This program specifies one invocation of the map operator and uses `table` clauses to specify `sum` as the reduce operator. The map operator transforms its input `logRecord` into two key/value pairs:

$$\begin{aligned}\langle k, x\rangle &= \langle \texttt{getOrigin}(\texttt{logRecord}), 1\rangle\\ \langle k', x'\rangle &= \langle \texttt{getTarget}(\texttt{logRecord}), 1\rangle\end{aligned}$$

Here, `getOrigin` and `getTarget` are pure functions that compute the latitude and longitude of the host issuing the query and the host serving the result, respectively. The latitude and longitude together serve as the key into the tables. Because the number 1 serves as the value associated with the key, the `sum` aggregators end up counting query log entries by key. Figure 14 shows the Sawzall grammar.

The Sawzall implementation maintains state and makes non-deterministic scheduling decisions.

Sawzall state. The map operator is stateless, whereas the reduce operator is stateful, using state to incrementalize its aggregation. The implementation in the paper of Pike *et al.* [4] partitions the reducer key space into R parts, where R is a command-line argument upon job submission. There are multiple instances of the reduce operator, one per partition. Because reduction works independently per key, each instance of the reduce operator can maintain the state for its assigned part of the key space independently.

Sawzall non-determinism. At the language level, Sawzall is deterministic. Sawzall is designed for MapReduce, and the strength of MapReduce is that at the implementation level, it runs on a cluster of workstations for scalability. To exploit the parallelism of the cluster, at the implementation level, MapReduce makes non-deterministic dynamic scheduling decisions. Reducers can start while map is still in process, and different reducers can work in parallel with each other. Different mappers can also work in parallel; we will use Brooklet to address this optimization later in the paper and describe a translation with a single map operator for now.

4.5.1. *Sawzall translation example.*

Given the Sawzall program P_z from earlier discussion, assuming $R = 4$ partitions, the Brooklet version P_b is

```
output; /*no output queue, outputs are in variables*/
input q_log;
(q_1, q_2, q_3, q_4) ← Map(q_log); /*getOrigin/getTarget*/
($v_1) ← Reduce(q_1, $v_1);
($v_2) ← Reduce(q_2, $v_2);
($v_3) ← Reduce(q_3, $v_3);
($v_4) ← Reduce(q_4, $v_4);
```

There is one reduce operator for each of the R partitions. Each reducer performs the work for both aggregators (`queryOrigins` and `queryTargets`) from the original Sawzall program. The final reduction results are in variables `$v_1 ... $v_4`.

4.5.2. Sawzall translation. Figure 14 specifies the program translation, domains, and operator wrappers. There is only one program translation rule T^{p}_{z}. The translation $[\![F_z, P_z, R]\!]^{p}_{z}$ takes the Sawzall function environment, the Sawzall program, and the number of reducer partitions as arguments. In detail, the translation proceeds as follows:

$\overline{out}, q_{in}$: $\texttt{input}\,\texttt{;}\,\overline{emit} = P_z$: Pattern matches on the entire Sawzall program P_z and binds names to the output aggregators $\overline{out}$, input queue q_{in}, and emit statements $\overline{emit}$.

$\forall i \in 1 \dots R : q_i = \mathit{freshId}()$: Creates R queue names, each of which will be used to connect the `Map` operator to a `Reduce` operator.

$\forall i \in 1 \dots R : v_i = \mathit{freshId}()$: Creates R variable names, each of which will store a result of a `Reduce` operator computation.

$f_{\texttt{Map}} = \mathit{wrapMap}(F_z, \overline{emit}, R)$: Wraps all of the functions from the $\overline{emit}$ statements into a single map operator.

$f_{\texttt{Reduce}} = \mathit{wrapReduce}(F_z, \overline{out})$: Wraps all of the $\overline{out}$ declarations into a single reduce operator, which will be replicated R times.

$F_b = [\texttt{Map} \mapsto f_{\texttt{Map}},\ \texttt{Reduce} \mapsto f_{\texttt{Reduce}}]$: Populates the Brooklet function environment with the wrapped functions.

$op_m = (\overline{q}) \leftarrow \texttt{Map}(q_{in});$: Declares a Brooklet operator for the `Map` function.

$\forall i \in 1 \dots R : op_i = (v_i) \leftarrow \texttt{Reduce}(q_i\texttt{,}\, v_i\texttt{)}\ \texttt{;}$: Declares R Brooklet operators for the `Reduce` functions.

$\overline{op}' = op_m, \overline{op}$: Appends the map operators and the sequence of reduce operators to the Brooklet sequence of operators.

In addition to the translation described earlier, there are four rules for wrapping the map and reduce functions:

W_z – MAP : This is a recursive rule for making a single Brooklet map operator from a list of *emit* statements. Each emit statement corresponds to a user-defined computation on the input data. The wrap function is similar to the one described for SRA, with the additional complexity that the map operator wrapper uses a hash function to scatter its output over the reducer key space for load balancing.
W_z – MAP – • : This rule is the base case for the recursive translation of *emit* statements, which is invoked when the list of statements is empty.
W_z – REDUCE : Each reducer's variable stores the mapping from each key in that reducer's partition to the latest reduction result for that key.
W_z – REDUCE – $\emptyset$: If the key is new, register x_2 as the key's initial value.

4.5.3. Sawzall discussion. The Sawzall translation is simpler than that of CQL or StreamIt, because each translated program uses the same simple topology. The translation hard-codes the data parallelism for the reducers but generates only one mapper, thus deferring data parallelism for mappers to a separate optimization step. There was no prior formal semantics for Sawzall. The closest is a rigorous description of the programming model by Lämmel, in which he produces an executable specification via an implementation in Haskell [25].

4.6. StreamIt to Brooklet

StreamIt [5, 26] is a streaming language tailored for parallel implementations of applications such as MPEG decoding [18]. At its core, StreamIt is a synchronous data flow (SDF) language [27], which means that each time an operator fires, it consumes a fixed number of data items and produces a fixed number of data items. In the MPEG example, data items are pictures. StreamIt distinguishes between primitive and composite operators. A primitive operator (*filter* in StreamIt terminology) has optional local state. A composite operator is either a pipeline, a split-join, or a feedback loop. A pipeline puts operators in sequence, a split-join puts them in parallel, and a feedback loop puts them in a cycle. The topology of a StreamIt program is restricted to well-nested compositions of these. All StreamIt operators and programs have exactly one input and one output. We only focus on StreamIt's SDF core here and encapsulate the local deterministic part of the computation in opaque pure functions while keeping the parts of the computation that are relevant to streaming. We omit non-core features such as teleport messaging [18], which delivers control messages between operators and which could be modeled in Brooklet through shared variables.

StreamIt syntax:

P_s	$::= ft \mid pl \mid sj \mid fl$	*StreamIt program*
ft	$::=$ `filter` $\{\ \overline{s}$ `work` $\{\ a\ \overline{ps}\ \overline{pp}\ \}\ \}$	*Filter*
a	$::= \overline{s}, \overline{t} \leftarrow f\ (\ \overline{s}, \overline{pk}\ )$;	*Assign*
pk	$::=$ `peek` $(\ x\ )$;	*Peek*
ps	$::=$ `push` $(\ t\ )$;	*Push*
pp	$::=$ `pop();`	*Pop*
pl	$::=$ `pipeline` $\{\ \overline{P_s}\ \}$	*Pipeline*
sj	$::=$ `splitjoin` $\{\ sp\ \overline{P_s}\ jn\ \}$	*SplitJoin*
fl	$::=$ `feedbackloop` $\{\ jn$ `body` P_s `loop` $P_s\ sp\ \}$	*Feedback*
sp	$::=$ `split` (`duplicate` | `roundrobin`) ;	*Split*
jn	$::=$ `join roundrobin;`	*Join*
$f \mid s \mid t$	$::= id$	*Function/state/temporary name*
x	$::= int$	*Number*

StreamIt example: MPEG decoder.

```
pipeline {
  splitjoin {
    split roundrobin;
    filter { work { tf ← FrequencyDecode(peek(1));
                    push(tf); pop(); } }
    filter { work { tm ← MotionVecDecode(peek(1));
                    push(tm); pop(); } }
    join roundrobin; }
  filter { s; work { s,tc ← MotionComp(s,peek(1));
                     push(tc); pop(); } } }
```

StreamIt program translation: $[\![\, F_s, P_s \,]\!]^p_s = \langle F_b, P_b \rangle$

$$\frac{q_{out} = \mathit{freshId}() \qquad q_{in} = \mathit{freshId}() \qquad F_b, \overline{op} = [\![\, F_s, P_s, q_{out}, q_{in} \,]\!]^p_s}{[\![\, F_s, P_s \,]\!]^p_s = F_b, \texttt{output}\ q_{out}\texttt{;}\ \texttt{input}\ q_{in}\texttt{;}\ \overline{op}} \quad (\mathrm{T}^p_s)$$

$$\frac{\begin{array}{c}\overline{s}, \overline{t} \leftarrow f\,(\overline{s}, \overline{pk}) = a \\ f_b = \mathit{freshId}() \qquad \forall i \in 1 \ldots |\overline{s}|+1 : v_i = \mathit{freshId}() \\ F_b = [f_b \mapsto \mathit{wrapFilter}(F_s, a, \overline{ps}, \overline{pp})] \\ op = (q_{out}, \overline{v}) \leftarrow f_b\,(q_{in}, \overline{v})\texttt{;}\end{array}}{[\![\, F_s, \texttt{filter}\{\overline{s}\ \texttt{work}\{a\ \overline{ps}\ \overline{pp}\}\}, q_{out}, q_{in} \,]\!]^p_s = F_b, op} \quad (\mathrm{T}^p_s\text{-Ft})$$

$$\frac{\begin{array}{c}n = |\overline{P_s}| \qquad \forall i \in 1 \ldots n-1 : q_i = \mathit{freshId}() \\ \forall i \in 1 \ldots n : F_{b_i}, \overline{op}_i = [\![\, F_s, P_{s_i}, q_i, q_{i-1} \,]\!]^p_s\end{array}}{[\![\, F_s, \texttt{pipeline}\{\overline{P_s}\}, q_n, q_0 \,]\!]^p_s = \cup\overline{F_b}, \overline{op}_1 \ldots \overline{op}_n} \quad (\mathrm{T}^p_s\text{-Pl})$$

$$\frac{\begin{array}{c}n = |\overline{P_s}| \qquad \forall i \in 1 \ldots n : q_i = \mathit{freshId}\texttt{()} \\ \forall i \in 1 \ldots n : q'_i = \mathit{freshId}\texttt{()} \\ F_{b_s}, op_s = [\![\, F_s, sp, \overline{q}, q_0 \,]\!]^p_s \\ \forall i \in 1 \ldots n : F_{b_i}, \overline{op_i} = [\![\, F_s, P_{s_i}, q'_i, q_i \,]\!]^p_s \\ F_{b_j}, op_j = [\![\, F_s, jn, q_{n+1}, \overline{q}' \,]\!]^p_s \\ F_b = F_{b_s} \cup (\cup\overline{F_b}) \cup F_{b_j} \qquad \overline{op} = op_s\ \overline{op}\ op_j\end{array}}{[\![\, F_s, \texttt{splitjoin}\ \{sp\ \overline{P_s}\ jn\}, q_{n+1}, q_0 \,]\!]^p_s = F_b, \overline{op}} \quad (\mathrm{T}^p_s\text{-Sj})$$

$$\frac{\begin{array}{c}\forall i \in 1 \ldots 4 : q_i = \mathit{freshId}\texttt{()} \\ F_{b_j}, op_j = [\![\, F_s, jn, q_1, (q_0, q_4) \,]\!]^p_s \\ F_{b_b}, \overline{op_b} = [\![\, F_s, P_s, q_2, q_1 \,]\!]^p_s \\ F_{b_l}, \overline{op_l} = [\![\, F_s, P'_s, q_4, q_3 \,]\!]^p_s \\ F_{b_s}, op_s = [\![\, F_s, sp, (q_3, q_5), q_2 \,]\!]^p_s \\ F_b = F_{b_j} \cup F_{b_b} \cup F_{b_s} \cup F_{b_l} \qquad \overline{op} = op_j\ \overline{op_b}\ op_s\ \overline{op_l}\end{array}}{[\![\, F_s, \texttt{feedbackloop}\{jn\ \texttt{body}\ P_s\ \texttt{loop}\ P'_s\ sp\}, q_5, q_0 \,]\!]^p_s = F_b, \overline{op}} \quad (\mathrm{T}^p_s\text{-Fl})$$

$$\frac{\begin{array}{c}f = \mathit{freshId}() \qquad op = (\overline{q}) \leftarrow f\,(q_0)\texttt{;} \\ F_b = [f \mapsto \mathit{wrapDupSplit}(|\overline{q}|)]\end{array}}{[\![\, F_s, \texttt{split duplicate;}, \overline{q}, q_0 \,]\!]^p_s = F_b, op} \quad (\mathrm{T}^p_s\text{-Dup-Split})$$

$$\frac{\begin{array}{c}f = \mathit{freshId}() \qquad v = \mathit{freshId}() \\ F_b = [f \mapsto \mathit{wrapRRSplit}(|\overline{q}|)] \\ op = (\overline{q}, v) \leftarrow f\,(q_0, v)\texttt{;}\end{array}}{[\![\, F_s, \texttt{split roundrobin;}, \overline{q}, q_0 \,]\!]^p_s = F_b, op} \quad (\mathrm{T}^p_s\text{-RR-Split})$$

$$\frac{\begin{array}{c}f = \mathit{freshId}() \qquad \forall i \in 0 \ldots |\overline{q}| : v_i = \mathit{freshId}() \\ F_b = [f \mapsto \mathit{wrapRRJoin}(|\overline{q}|)] \\ op = (q_o, \overline{v}) \leftarrow f\,(\overline{q}, \overline{v})\texttt{;}\end{array}}{[\![\, F_s, \texttt{join roundrobin;}, q_o, \overline{q} \,]\!]^p_s = F_b, op} \quad (\mathrm{T}^p_s\text{-RR-Join})$$

StreamIt domains:

z	$\in \mathcal{Z}$	*Data item*
ℓ	$\in \mathcal{Z}^*$	*List of data items*
x	$\in \mathbb{N}$	*Natural number (peek number)*
I_s	$\in (id \mapsto z) \times \mathcal{Z}^*$	*StreamIt input*
O_s	$\in \mathcal{Z}^*$	*StreamIt output*

StreamIt operator signatures:

$$\text{filter} : \mathcal{Z}^* \times \mathcal{Z}^* \rightarrow \mathcal{Z}^*$$

StreamIt operator wrapper signatures:

$$\begin{array}{ll}\text{wrapFilter} & : \mathcal{Z} \times \{1\} \times \mathcal{Z}^* \rightarrow \mathcal{Z}^* \times \mathcal{Z}^* \\ \text{wrapDupSplit} & : \mathcal{Z} \times \{1\} \rightarrow \mathcal{Z}^* \\ \text{wrapRRSplit} & : \mathcal{Z} \times \{1\} \times \mathbb{N} \rightarrow \mathcal{Z}^* \times \mathbb{N} \\ \text{wrapRRJoin} & : \mathcal{Z}^* \times \{1\} \times \mathbb{N} \rightarrow \mathcal{Z} \times \mathbb{N}\end{array}$$

StreamIt operator wrappers:

$$\frac{\begin{array}{c}\overline{s}, \overline{t} \leftarrow f(\overline{s}, \overline{\texttt{peek}\,(x)}) = a \\ \overline{z}, \ell = \overline{d}_v \qquad \ell' = \ell, d_{in} \\ |\ell'| \geq |\overline{pp}| \qquad \forall i \in 1 \ldots |\overline{x}| : |\ell'| \geq x_i \\ \forall i \in 1 \ldots |\overline{x}| : d_i = \ell'_{x_i} \\ \overline{z}', \overline{d}_q = f(\overline{z}, \overline{d}) \\ r = |\ell'| - |\overline{pp}| \qquad \ell'' = \ell'_{r+1} \cdots \ell'_{|\ell'|}\end{array}}{\mathit{wrapFilter}(a, \overline{ps}, \overline{pp})(d_{in}, \_, \overline{d}_v) = \overline{d}_q, \overline{z}', \ell''} \quad (\mathrm{W}_s\text{-Filter-Full})$$

$$\frac{\begin{array}{c}\overline{s}, \overline{t} \leftarrow f(\overline{s}, \overline{\texttt{peek}\,(x)}) = a \\ \overline{z}, \ell = \overline{d}_v \qquad \ell' = \ell, d_{in} \\ |\ell'| < |\overline{pp}| \text{ or } \exists i \in 1 \ldots |\overline{x}| : |\ell'| < x_i\end{array}}{\mathit{wrapFilter}(a, \overline{ps}, \overline{pp})(d_{in}, \_, \overline{d}_v) = \bullet, \overline{z}, \ell'} \quad (\mathrm{W}_s\text{-Filter-Wait})$$

$$\frac{\forall i \in 1 \ldots N : b_i = d_{in}}{\mathit{wrapDupSplit}(N)(d_{in}, \_) = \overline{b}} \quad (\mathrm{W}_s\text{-Dup-Split})$$

$$\frac{\begin{array}{c}c' = c + 1 \bmod N \qquad b_v = d_{in} \\ \forall i \in 1 \ldots N, i \neq c : b_i = \bullet\end{array}}{\mathit{wrapRRSplit}(N)(d_{in}, \_, c) = \overline{b}, c'} \quad (\mathrm{W}_s\text{-RR-Split})$$

$$\frac{\begin{array}{c}d'_i = d_{in}, d_i \qquad \forall j \neq i \in 1 \ldots N : d'_j = d_j \\ d''_c, d_{out} = d'_c \qquad \forall j \neq c \in 1 \ldots N : d''_j = d'_j \\ b_{out}, c', \overline{d}''' = \mathit{wrapRRJoin}(N)(\bullet, i, c+1 \bmod N, \overline{d}'')\end{array}}{\mathit{wrapRRJoin}(N)(d_{in}, i, c, \overline{d}) = (b_{out}, d_{out}), c', \overline{d}'''} \quad (\mathrm{W}_s\text{-RR-Join-Full})$$

$$\frac{\begin{array}{c}\forall j \neq i \in 1 \ldots N : d'_j = d_j \\ d'_i = d_{in}, d_i \qquad d_c = \bullet\end{array}}{\mathit{wrapRRJoin}(N)(d_{in}, i, c, \overline{d}) = \bullet, c, \overline{d}'} \quad (\mathrm{W}_s\text{-RR-Join-Wait})$$

Figure 15. StreamIt semantics on Brooklet.

4.6.1. StreamIt program example: MPEG decoder. The example StreamIt program P_s in Figure 15 is based on a similar example by Drake *et al.* [18]. It illustrates how the StreamIt language can be used to decode MPEG video. The example uses a pipeline and a split-join to compose three filters. Each filter has a work function, which peeks and pops from its predecessor stream, computes a temporary value, and pushes to its successor stream. In addition, the `MotionComp` filter also has an explicit state variable `s` for storing a reference picture between iterations.

4.6.2. StreamIt implementation issues. As before, we first discuss the intuition for the implementation before giving the details of the translation.

StreamIt state. Filters can have explicit state, such as `s` in the example. Furthermore, because Brooklet queues support only push and pop, but not peek, the translation of StreamIt will have to buffer data items in a state variable until enough are available to satisfy the maximum `peek()` argument in the work function. Round-robin splitters also need a state variable with a *cursor* that determines where to send the next data item. A cursor is simply an index relative to the splitter. It keeps track of which queue is next in round-robin order. Round-robin joiners also need a cursor, plus a buffer for any data items that arrive out of turn.

To improve performance, the StreamIt compiler aggressively parallelizes stateless filters to run on multiple cores [19]. The compiler can also parallelize implicitly stateful filters that peek, because the static scheduling information allows the runtime to preserve data output ordering. Although we have not implemented this optimization, River annotations could be used to pass the scheduling information to a streaming runtime. In a distributed setting, the scheduler could implement this optimization in a similar style to the hybrid static-dynamic scheduler described in our prior work [28].

StreamIt non-determinism. StreamIt, at the language level, is deterministic. Furthermore, because it is an SDF language, the number of data items peeked, popped, and pushed by each operator is constant. At the same time, StreamIt permits pipeline parallelism, task parallelism, and data parallelism. This gives an implementation different scheduling choices, in which Brooklet models by non-deterministically selecting a firing queue. Despite these non-deterministic choices, an implementation must ensure deterministic end-to-end behavior, in which our translation accomplishes with buffering and synchronization.

4.6.3. StreamIt translation example. StreamIt program translation turns the StreamIt MPEG decoder P_s in Figure 15 into a Brooklet program P_b:

```
output q_out;
input q_in;
(q_f, q_m, $sc)          ← wrapRRSplit-2(q_in, $sc);
(q_fd, $f)               ← wrapFilter-FrequencyDecode(q_f, $f);
(q_md, $m)               ← wrapFilter-MotionVecDecode(q_m, $m);
(q_d, $fd, $md, $jc)     ← wrapRRJoin-2(q_fd, q_md, $fd, $md, $jc);
(q_out, $s, $mc)         ← wrapFilter-MotionComp(q_d, $s, $mc);
```

Each StreamIt filter becomes a Brooklet operator. StreamIt composite operators are reflected in Brooklet's operator topology. StreamIt's SplitJoin yields separate Brooklet split and join operators. The stateful filter `MotionComp` has two variables: `$s` models its explicit state `s` and `$mc` models its implicit buffer.

Similarly to CQL, there are recursive translation rules, one for each language construct. The base case is the translation of filters, and the recursive cases compose larger topologies for pipelines, split-joins, and feedback loops. Feedback loops turn into cyclic Brooklet topologies. The most interesting aspect are the helper rules for split and join, because they use explicit Brooklet state to achieve StreamIt determinism. Figure 15 shows the rules, which are explained in detail as follows.

T_s^p : Every StreamIt program has a single input queue and a single output queue. This rule creates those queues for the Brooklet representation. The rest of the translation proceeds by recursion over the StreamIt topologies.

T_s^p – FT : This is the base case of the recursive translations. The most important part of the translation is the wrapping of the filter function. The remainder of the rule connects the appropriate input and output queues.

T_s^p – PL : This rule translates pipeline topologies. The bulk of this rule consists of setting up the appropriate queue connections and calling other rules to translate the component topologies.

T_s^p – SJ : This rule translates split-join topologies. Importantly, this rule will recursively invoke the helper rules for split and join.

T_s^p – FL : This rule translates feedback loops. Feedback loops turn into cyclic Brooklet topologies. Again, this rule creates and connects the appropriate queues and invokes the helper rules for join and split.

T_s^p – DUP-SPLIT : The input to the splitter is a queue q_0, and the output is a list of queues $\overline{q}$. The translation invokes the *wrapDupSplit* function to create the Brooklet function to duplicate data items.

T_s^p – RR-SPLIT : The round-robin splitter translation takes as input a queue q_0, and the output is a list of queues $\overline{q}$. This translation is slightly more complex than the translation of the duplicate splitter, because it needs to create a Brooklet variable, which is used as a cursor to keep track of the next queue in round-robin order.

T_s^p – RR-JOIN : The input to the joiner is a list of queues $\overline{q}'$, and the output is a single queue q_0. Like the splitter, the round-robin joiner creates a Brooklet variable to use a cursor for tracking queue order. The joiner also stores one variable for each queue, to buffer data that arrives out of turn.

In addition to the translation described earlier, the StreamIt-to-Brooklet translation uses the rules for wrapping functions described as follows. In general, there are four StreamIt functions that need to be wrapped: filter (e.g., a user-defined filter function), duplicate split, round-robin split, and join. Note that in StreamIt, filters may read more than one data item when firing. The StreamIt static scheduler ensures that there are enough data items on the input buffer each time a filter is executed. In contrast, the Brooklet implementation and semantics may execute a filter before there are enough data items on the input queue. To handle this, we define two wrapper functions each for filter and join: FULL handles the case where there are enough input items and WAIT handles the case when the operator needs to block until more data items arrive.

W_s – FILTER-FULL : The rule first checks if there are a sufficient number of data items for the StreamIt function. That number, x, is declared in the StreamIt program syntax. If there are a sufficient number of data items, the StreamIt function is invoked.

W_s – FILTER-WAIT : Like the FILTER-FULL version of the filter rule, the wrapper checks if there are a sufficient number of data items for the StreamIt function. If there are not, then the wrapper function does not invoke the StreamIt function and simply allows the data items to continue to buffer on the input buffer.

W_s – DUP-SPLIT : This wrapper function copies the input to each of its N outputs.

W_s – RR-SPLIT : The round-robin splitter uses a cursor, c, to keep track of the round-robin order. The wrapper function reads from its input queue and outputs the data item to the c'th output queue.

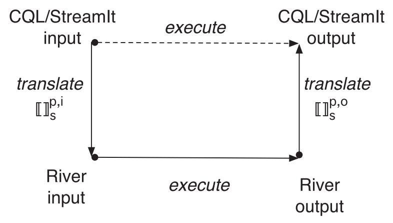


Figure 16. Illustration of correctness theorem for CQL and StreamIt.

W_s – RR-JOIN-FULL : Again, because Brooklet does not have the same static scheduling as StreamIt, the join wrapper function needs to check if data items have arrived on the designated input port. If a data item has arrived, then it can be passed to the output port.

W_s – RR-JOIN-WAIT : Alternatively, if no data item has arrived on the port designated by the cursor, then the join needs to block until items arrive, without incrementing its cursor.

4.6.4. StreamIt discussion. Our translation from StreamIt to Brooklet yields a program with maximum scheduling flexibility, allowing any interleavings as long as the end-to-end behavior matches the language semantics. This makes it amenable to distributed implementation. In contrast, StreamIt compilers [5] statically fix one schedule, which also determines where intermediate results are buffered. The buffering is implicit state, and StreamIt also has explicit state in filters. As we will see in Section 5, state affects the applicability of optimizations. Prior work on formal semantics for StreamIt does not model state [26]. By modeling state, our Brooklet translation facilitates reasoning about optimizations.

Now that we have seen how to translate three languages, it is clear that it is possible to model additional streaming languages or language features on Brooklet. For example, Brooklet could serve as a basis for modeling teleport messaging [18].

4.7. Translation correctness

We formulate correctness theorems for CQL and StreamIt with respect to their formal semantics [20, 26]. The proofs appear in a technical report [29]. We do not formulate a theorem for Sawzall, because it lacks prior formal semantics; our mapping to Brooklet provides the first formal semantics for Sawzall.

Theorem 1 (CQL translation correctness)
For all CQL function environments F_c, programs P_c, and inputs I_c, the results under CQL semantics are the same as the results under Brooklet semantics after translation $[\![F_c, P_c]\!]_c^p$.

Theorem 2 (StreamIt translation correctness)
For all StreamIt function environments F_s, programs P_s, and inputs I_s, the results under StreamIt semantics are the same as the results under Brooklet semantics after translation $[\![F_s, P_s]\!]_s^p$.

Informally, these theorems state that if you were to execute a CQL or StreamIt program according to their semantics, or if you were to translate that program to River and execute the River program, then the results would be the same, as shown in Figure 16.

5. OPTIMIZATIONS

One of the benefits of a virtual execution environment is that it can provide a single implementation of an optimization that applies to multiple source languages. In prior work on stream processing, each source language had to re-implement similar optimizations. The River execution environment, on the other hand, supports optimization reuse across languages. Here, we are primarily interested in optimizations from the streaming domain, which operate at the level of a stream graph, as opposed to traditional optimizations at the level of functional or imperative languages. By working at the level of a stream graph, River can optimize an entire distributed application. The three optimizations described in the following, operator fission, fusion, and placement, can be reused across our language implementations of CQL, Sawzall, and StreamIt. As with the other contributions of River, the Brooklet calculus provides a solid foundation, but new ideas are needed to build an execution environment upon it.

5.1. Brooklet treatment of optimizations

The Brooklet paper [6] decouples optimizations from their source languages. It specifies each optimization by a *safety guard* and a *rewrite rule*. The safety guard checks whether a subgraph satisfies the preconditions for applying the optimization. It exploits the one-to-one restriction on queues and the fact that state is explicit to establish these conditions. If a subgraph passes the safety guard, the rewrite rule replaces it by a transformed subgraph. The Brooklet paper then proceeds to prove that the optimizations leave the observable input/output behavior of the program unchanged.

The Brooklet paper discusses three specific optimizations: (i) *Fusion* replaces two operators by a single operator, thus reducing communication costs at the expense of pipeline parallelism. (ii) *Fission* replaces a single operator by a splitter, a number of data-parallel replicas of the operator, and a merger. The Brooklet paper only permits fission for stateless operators. (iii) *Selection hoisting* (also known as selection pushdown) rewrites a subgraph $A \rightarrow \sigma$ into a subgraph $\sigma \rightarrow A$, assuming that A is a stateless operator and σ is a selection operator that only relies on data fields unchanged by A. Selection hoisting can improve performance by reducing the number of data items that A has to process.

5.2. River optimization support

We made the observation that River's source languages are designed to make certain optimizations that are safe by construction, without requiring sophisticated analysis. For example, Sawzall provides a set of built-in aggregations that are known to be commutative and partitioned by a user-supplied key, thus enabling fission. Rather than losing safety information in translation, only to have to discover it again before optimization, we wanted to add it to River's IL. However, at the same time, we did not want to make the IL source language specific, which would jeopardize the reusability of optimizations and the generality of River.

We resolved this tension by adding extensible annotations to River's graph language. An annotation next to an operator specifies *policy* information, which encompasses safety and profitability. Safety policies are usually passed down by the translator from source language to IL, such as which operators to parallelize. Profitability policies usually require some knowledge of the execution platform, such as the number of machines to parallelize on. In this paper, we use simple heuristics for profitability; prior work has also explored more sophisticated analyses for this, which are beyond the scope of this paper [15]. Policy is separated from *mechanism*, which implements the actual code transformation that performs the optimization. River's annotation mechanism allows it to do more powerful optimizations than Brooklet. For example, fission in River works not just on stateless operators but also on stateful operators, as long as the state is keyed and the key fields are listed in annotations. Both CQL and Sawzall are designed explicitly to make the key evident from the source code, so all we needed to do is preserve that information through their translators.

Annotation	Description
`@Fuse(`*ID*`)`	Directive to fuse operators with the same *ID* in the same process.
`@Parallel()`	Directive to perform fission on an operator.
`@Commutative()`	Declares that an operator's function is commutative.
`@Keys(`$k_1, \ldots, k_n$`)`	Declares that an operator's state is partitionable by the key fields $k_1, \ldots, k_n$ in each data item.
`@Group(`*ID*`)`	Directive to place operators with the same *ID* on the same machine.

Figure 17. River annotations for optimization.

To keep annotations extensible, they share a common, simple syntax, inspired by Java. Each use of an operator is preceded by zero or more annotations. Each annotation is written as an at-sign (`@`), an identifier naming the annotation, and a comma-separated list of expressions serving as parameters. River currently makes use of the annotations in Figure 17. We anticipate adding more annotations as we implement more source languages and/or more optimizations. The translator from River IL to native code invokes the optimizers one by one, transforming the IL at each step. A specification passed to the translator determines the order in which the optimizations are applied.

5.3. Fusion optimizer

Intuition. Fusion combines multiple stream operators into a single stream operator, to avoid the overhead of data serialization and transport [19, 30].

Policy. The policy annotation is `@Fuse(`*ID*`)`. Operators with the same *ID* are fused. Applying fusion is a trade-off. It eliminates a queue, reducing communication cost, but it prohibits operators from executing in parallel. Hence, fusion is profitable if the savings in communication cost exceed the lost benefit of parallelism. As shown in the Brooklet calculus, a sufficient safety precondition for fusion is if the fused operators form a straight-line pipeline without side entries or exits.

Mechanism. The current implementation replaces internal queues by direct function calls. A possible future enhancement would be to allow fused operators to share the same process but run on different threads. This would reduce the cost for communication but still maintain the benefits of pipeline parallelism on multicores. Tang *et al.* [31] describe another possible implementation in which all operators are fused, and the runtime determines when it is profitable to spawn a separate thread.

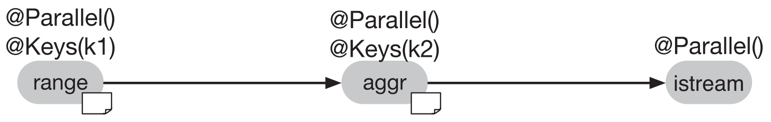
5.4. Fission optimizer

Intuition. Fission replicates an operator or a stream subgraph to introduce parallel computations on subsets of the data [17, 19, 32].

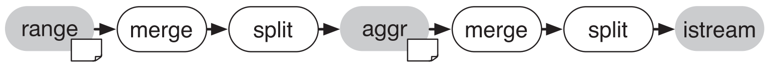
Policy. Fission uses three annotations. The `@Parallel()` annotation is a directive to parallelize an operator. The `@Commutative()` annotation declares that a given operator's function commutes. Finally, the `@Keys(`$k_1,\ldots,k_n$`)` annotation declares that an operator is stateful but that its state is keyed (i.e., partitioned) by the key in fields $k_1,\ldots,k_n$. Fission is profitable if the computation in the parallel segment is expensive enough to make up for the overhead of the inserted split and merge operators.

The safety conditions for fission depend on state and order. In terms of state, there must be no memory dependencies between replicas of the operator. This is trivially the case when the operator is stateless. The other way to accomplish this is if the state of the operator can be partitioned by key, such that each operator replica is responsible for a separate portion of the key space. In that case, the splitter routes data items by using a hash on their key fields. When a parallel segment consists of multiple operators, they must be either stateless or have the same key. To understand how order affects correctness, consider the following example. Assume that in the unoptimized program, the operator pushes data item d_1 before d_2 on its output queue. In the optimized program, d_1 and d_2 may be computed by different replicas of the operator, and depending on which replica is faster, d_2 may be pushed first. That would be unsafe if any downstream operator depends on order. That means that fission is safe with respect to order either if all downstream operators commute within each window boundary [33] or if the merger brings data items from different replicas back in the right order. Depending on the source language, the merger can use different ordering strategies: CQL embeds a logical timestamp in every data item that induces an ordering, Sawzall has commutative aggregations and can hence ignore order, and StreamIt only parallelizes stateless operators and can hence use round-robin order.

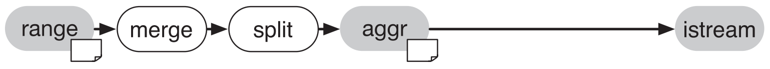
Mechanism. River's fission optimization consists of multiple steps. Consider the following example in which three operators appear in a pipeline. The first two operators, `range` and `aggr`, are stateful and keyed by k_1 and k_2, respectively. The third, `istream`, is stateless. The figures indicate stateful operators by a rectangle with a folded corner. All three operators have the `@Parallel()` annotation, indicating that fission should replicate them.



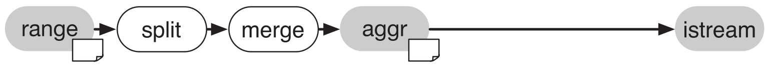
Step 1 adds split and merge operators around parallelizable operators. This trivially parallelizes each operator. At the same time, it introduces bottlenecks, as data streamed through adjacent mergers and splitters must pass through a single machine. Note that for source or sink operators, only merge or split, respectively, are needed. This is because the partitioning of input data and the combining of output data are assumed to occur outside of the system.



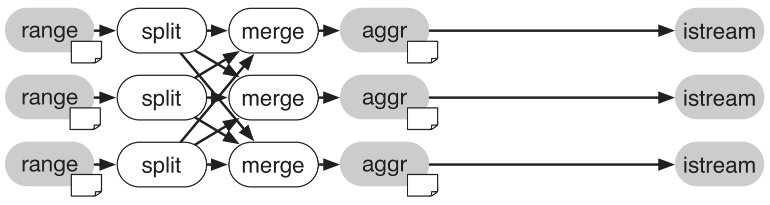
Step 2 removes the bottlenecks. There are two in the example; each calls for a different action. First, the merge and split between `aggr` and `istream` can be safely removed, because `istream` is stateless.



Next, the merge and split between `range` and `aggr` cannot be removed, because both operators partition state by different keys, k_1 and k_2. Instead, we apply a *rotation*. A rotation switches the order of the merge and split to remove the bottleneck. This is the same approach as the *shuffle* step in MapReduce [17].



Finally, step 3 replicates the operators to the desired degree of parallelism and inserts `@Group(`*ID*`)` annotations for the placement optimizer, to ensure that replicas actually reside on different machines.



In this example, all operators are parallelized. In other applications, only parts may be parallelized, so performance improvements will be subject to Amdahl's law: for example, if the unoptimized program spends one-fifth of its time in non-parallelizable operators, the optimized program can be no more than 5× faster.

5.5. Placement optimizer

While Brooklet explored selection hoisting, River explores placement instead, because it is of larger practical importance, and illustrates the need for the optimizer to be aware of the platform.

Intuition. Placement assigns operators to machines and cores to better utilize physical resources [34].

Policy. The policy annotation is `@Group(`*ID*`)`. Operators with the same *ID* are assigned to the same machine. Several operators can be assigned the same *ID* to take advantage of machines with multiple cores. Placement is profitable if it reduces network traffic by placing operators that communicate much on the same machine. In our current implementation, annotations are applied by hand. The annotations could be added automatically by leveraging prior work on determining optimal placement [34, 35].

Mechanism. The placement mechanism does not transform the IL but, rather, directs the runtime to assign the same machine to all operators that use the same identifier. Information such as the number of machines is only available at the level of the virtual execution environment, a trait that River shares with other language-based virtual machines. Placement is complicated if operators share state. In general, River could support sharing variables across machines but relies on the underlying runtime to support that functionality. Because our current back-end does not provide distributed shared state, the placement optimizer has an additional constraint. It ensures that all operators that access the same shared variable are placed on the same machine. Fortunately, none of the three exemplar languages, CQL, Sawzall, and StreamIt, need cross-machine shared state.

5.6. *When to optimize*

The Brooklet calculus abstracts away the timing of optimizations. The River execution environment performs optimizations once the available resources are known, just before running the program. In other words, the translations from source languages to IL happen ahead of time, but the translation from IL to native code for a specific target platform is delayed until the program is launched. That enables River to make more informed profitability decisions.

As just described, River implements *static* optimizations. In future work, we plan to address *dynamic* optimizations that make decisions at runtime. One approach for this is to change the data flow graph at runtime, as carried out, for instance, in Flextream [36] or Optimus [37]. But another promising approach to implementing dynamic optimizations is to statically create a more general graph and then adapt how data flows through it at runtime. A seminal example for this is the Eddy operator, which performs dynamic operator reordering without physically changing the graph at runtime [38]. River could use annotations to decide where an optimization applies and statically rewrite the graph to add in control operators that dynamically route data items for optimization.

6. RUNTIME SUPPORT

The main goal for River's runtime support is to insulate the IL and the runtime from each other. Figure 18 shows the architecture. It consists of a stack of layers, where each layer only knows about the layers immediately below and above it. Above the operating system (OS) is the streaming runtime, which provides the software infrastructure for process management, data transport, and distribution. Above that is the runtime adaptation layer, which provides the interface between River operators and the distributed runtime. At the highest level are the operator instances and their associated variables. River's clean separation of concerns ensures that it can be ported to additional streaming runtimes. The rest of this section describes each layer in detail.

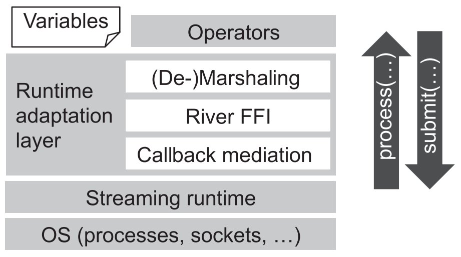


Figure 18. Stack of layers for executing River.

6.1. Streaming runtime

A distributed streaming runtime for River must satisfy the following requirements. It must launch the right processes on the right machines in the distributed system to host operators and variables. It must provide reliable transport with ordered delivery to implement queues. It must arbitrate resources and monitor system health for reliability and performance. Finally, our placement optimizer relies on placement support in the runtime. There is no strict latency requirement: the semantics tolerate an indefinite transit time.

Distributed state and synchronization are optional. If they are provided by the distributed runtime, River can exercise additional flexibility in placing operators that share state to machines. Otherwise, River uses placement support for co-locating operators that share state on the same machine.

The streaming runtime sits on top of an OS layer, which provides basic centralized services to the runtime layer, such as processes and sockets. However, building a high-performance streaming runtime satisfying the earlier-listed requirements is a significant engineering effort beyond the OS. Therefore, we reuse an existing runtime instead of building our own. We chose System S [16], a distributed streaming runtime developed at IBM that satisfies the requirements[‡] and that is available to universities under an academic license. While System S has its own streaming language, we bypassed that in our implementation, instead of interfacing with the runtime's C++ API.

6.2. Runtime adaptation

The runtime adaptation layer provides the interface between River operators and the distributed runtime. As shown in Figure 18, it consists of three sub-layers.

Callback mediation layer. This layer mediates between the runtime-specific APIs and callbacks, and the River-specific functions. The code in this layer peels off runtime-specific headers from a data item and then passes the data item to the layer above. Similarly, it adds the headers to data on its way down. If there are shared variables, this layer performs locking and implements output buffers for avoiding back-pressure-induced deadlock as described in Section 3.4. The callback mediation layer is linked into the streaming runtime.

River FFI layer. An FFI enables calls across programming languages. In this case, it enables C++ code from a lower layer to call a River function in an upper layer, and it enables River code in an upper layer to call a C++ function in a lower layer. The River FFI is the same as the OCaml FFI. Each OS process for River contains an instance of the OCaml runtime, which it launches during start-up.

(De-)marshaling layer. This layer converts between byte arrays and data structures in River's implementation language. It uses an off-the-shelf serialization module. The code in this layer is auto-generated by the River compiler. It consists of `process(...)` functions, which are called by the next layer down, demarshal the data, and call the next layer up, and of `submit(...)` functions, which are called by the next layer up, marshal the data, and call the next layer down. Because this layer is crucial for performance, we plan to optimize it further by specialized code generation.

Overall, implementing River's runtime support on System S required 1636 lines of Java code for the River-to-C++ translator. The River-to-C++ translator makes use of an additional 780 lines of boilerplate C++ code. Because that is only a moderate amount of code, we believe porting to a different platform would not be too much work.

6.3. Variables and operators

As described in Section 4.3, operators are generated from templates written by the language developer. Their implementation strikes a balance between the functional purity of operators in Brooklet

[‡]System S supports shared state and synchronization via the DPS toolkit [39].

and performance demands of an IL that needs to update data in place and make callbacks instead of returning values. Variables and operators are implemented in the implementation sublanguage of River. An operator firing takes the form of a function call from the next lower layer. If the operator accesses variables, then the call passes those as references so that the operator can perform in-place updates if needed. Instead of returning data as part of the function's return value, the operator invokes a callback for each data item it produces on an output queue. Note that this simple API effectively hides any lower-layer details from the variables and operators.

Operators handle all state allocations and memory maintenance for variables. The only time that state needs to pass from the operators to the lower-layer streaming runtime is for transport, in which case the data values are copied during serialization. In our OCaml implementation, all variables are allocated and garbage-collected by the OCaml runtime. When passing state for serialization, the OCaml runtime's FFI handles pinning the variables.

7. EVALUATION

We have built a proof-of-concept prototype of River, including front-ends for the three source languages, implementations of the three optimizations, and a back-end on the System S distributed streaming system. We have not yet tuned the absolute performance of our prototype; the goal of this paper was to show its feasibility and generality. Therefore, while this section presents some preliminary experimental results demonstrating that the system works and performs reasonably well, we leave further efforts on absolute performance to future work.

All performance experiments were run on a cluster of 16 machines. Each machine has two 4-core 64-bit Intel Xeon (X5365) processors running at 3 GHz, where each core has 32K L1i and 32K L1d caches of its own, and a 4-MB unified L2 cache that is shared with another core. The processors have a front side bus speed of 1333 MHz and are connected to 16 GB of uniform memory. Machines in the cluster are connected via 1-Gbit ethernet.

7.1. Support for existing languages

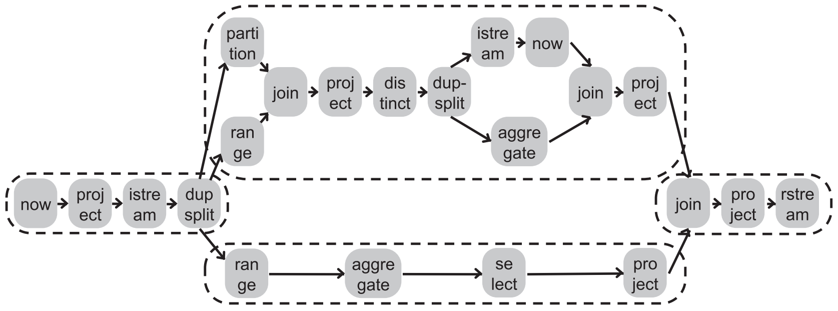
To verify that River is able to support a diversity of streaming languages, we implemented the language translators described in Section 4, as well as illustrative benchmark applications. The benchmarks exercise a significant portion of each language, demonstrating the expressivity of River. They are described as follows.

CQL Linear Road. Linear Road [1] is the running example in the CQL paper. It is a hypothetical application that computes tolls for vehicles traveling on the Linear Road highway. Figure 19(a) shows the operator graph of the application. Each vehicle is assigned a unique ID, and its location is specified by three attributes: speed, direction, and position. Each highway is also assigned an ID. Vehicles pay a toll when they drive on a congested highway. A highway is congested if the average speed of all vehicles on the highway over a 5-min span is less than 40 mph.

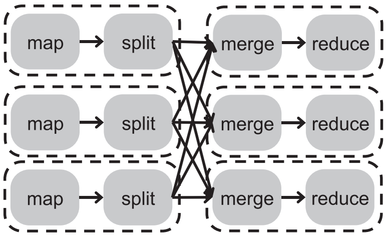
Sawzall batch log analyzer The example in Figure 14 shows this query, which is based on a similar query in Pike *et al.* [4]. The operator graph is shown in Figure 19(b). It is a batch job that analyzes a set of search query logs to count queries per origin based on IP address. The resulting aggregation could then be used to plot query origins on a world map.

StreamIt FM radio. This benchmark implements a multi-band equalizer [19]. As shown in Figure 19(c), the input passes through a demodulator to produce an audio signal and then an equalizer. The equalizer is implemented as a split-join with two band-pass filters; each band-pass filter is the difference of a pair of low-pass filters.

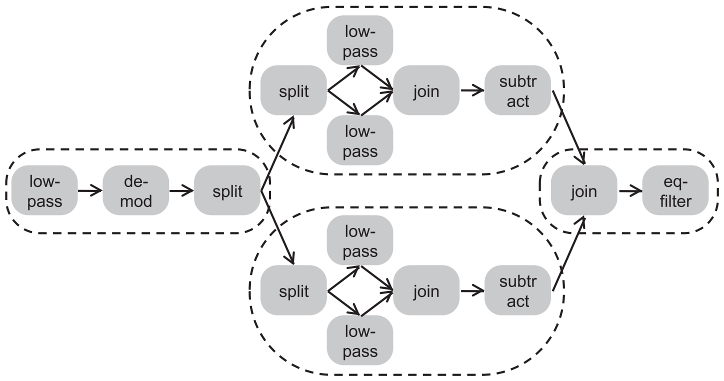
CQL continuous log analyzer. This is similar to the Sawzall log query analyzer, but it is a continuous query rather than a batch job. Its input comes from a server farm. Each server reports the origin of the requests it has received, and the analyzer performs an aggregation keyed by the origin over



(a) CQL's Linear Road in River.



(b) Sawzall Log Query Analyzer in River.



(c) StreamIt's FM Radio in River.

Figure 19. Structural view for the CQL, Sawzall, and StreamIt benchmarks. The dashed ovals group operators that are placed onto the same machine.

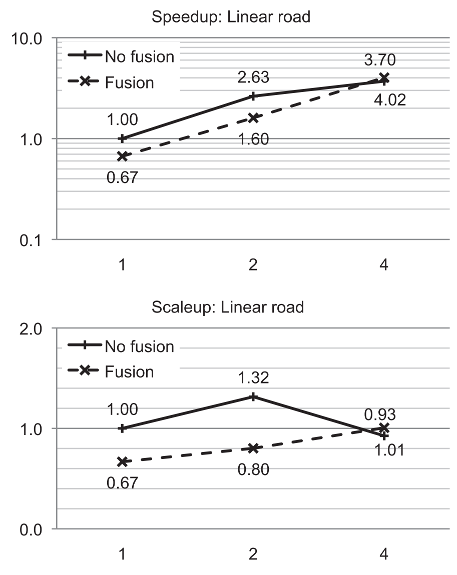
the most recent 5-min window. Note that the data are originally partitioned by the target (server) address, so the application must shuffle the data.

The three source languages, CQL, StreamIt, and Sawzall, occupy three diverse points in the design space for streaming languages, and the benchmark applications exercise significant portions of each language. This demonstrates that River is expressive enough to support a wide variety of streaming languages.

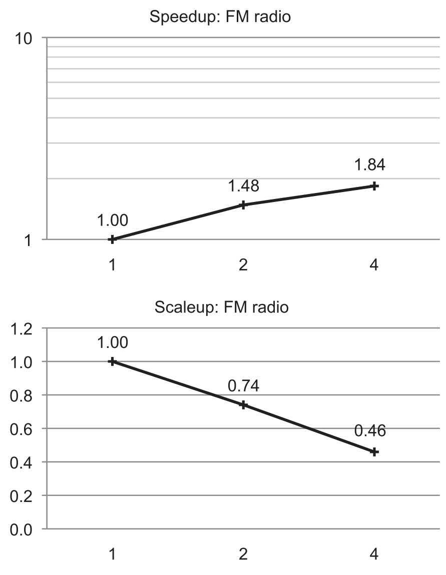
7.2. *Suitability for optimizations*

To verify that River is extensible enough to support a diverse set of streaming optimizations, we implemented each of the optimizations described in Section 5. We then applied the different optimizations to the benchmark applications from Section 7.1.

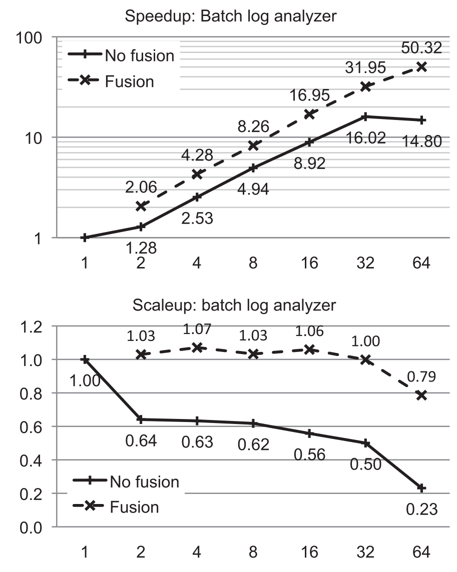
Placement. Our placement optimizer distributes an application across machines. Operators from each application were assigned to groups, and each group was executed on a different machine. As a first step, we used the simple heuristic of assigning operators to groups according to the branches of the top-level split–merge operators, although there has been extensive prior work on determining the optimal assignments [34]. In the non-fused version, each operator had its own process, and it was up to the OS to schedule processes to cores. Figure 20(a) and (b) shows the results of running both Linear Road and the FM radio applications on one, two, and four machines. Figure 19 shows the partitioning scheme for the four-machine case using dashed ovals. These results are particularly exciting because the original implementation of CQL was not distributed. Despite the fact that the Linear Road application shows only limited amounts of task and pipeline parallelism, the



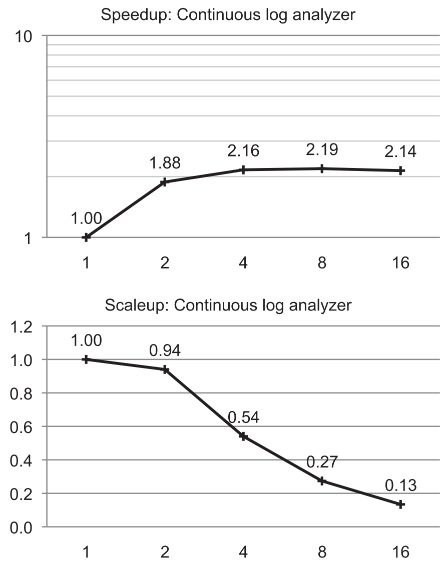
(a) Linear Road in CQL



(b) FM Radio in StreamIt



(c) Batch log analyzer in Sawzall



(d) Cont. log analyzer in CQL

Figure 20. Speedup is the throughput relative to single machine. Scaleup is the speedup divided by number of machines in (a) and (b), and the speedup divided by the degree of parallelism in (c) and (d).

first distributed CQL implementation achieves a 3.70× speedup by distributing execution on four machines. The FM Radio application exhibits a 1.84× speedup on four machines.

Fission. Our fission optimizer replicates operators and then distributes those replicas evenly across available machines. We tested two applications, the Sawzall batch log analyzer and the CQL continuous log analyzer, with increasing amounts of parallelism. In these experiments, the degree of parallelism corresponded to the number of available machines, from 1 to 16. We additionally ran the Sawzall query on 16 machines with 16, 32, 48, and 64 degrees of parallelism, distributing the computation across cores. The results are shown in Figure 20(c) and (d).

Fission adds split and merge operators to the stream graph. Therefore, in the non-fissioned case, there are fewer processing steps. In spite of this, the Sawzall program's throughput increased when the degree of parallelism went from 1 to 2. As the degree of parallelism and the number of machines increased from 2 to 16, the increase in throughput was linear, with an 8.92× speedup on 16 machines. When further distributed across cores, the Sawzall program also experiences a large performance increase. However, the 32-replicas case showed better performance than 64 replicas. This makes sense, because the unfused Sawzall query has five operators, each of which was replicated 64 times ($64 * 5 = 320$), while the total number of available cores across all machines was $16 * 8 = 128$.

The CQL continuous log analyzer saw a performance improvement with fission but only achieved at best a 2.19× speedup, with no benefit past six machines. Unlike Sawzall, all data items in CQL are timestamped. To maintain CQL's deterministic semantics, mergers must wait for messages from all input ports for a given timestamp. The lesson we learned was that when implementing a distributed, data-parallel CQL, we need to give more consideration on how to enforce deterministic execution efficiently. Unlike our River-CQL implementation, the original CQL did not do fission and thus did not encounter this issue.

Fusion. The Linear Road results in Figure 20(a) illustrate the trade-offs during fusion, which often comes at the expense of pipeline parallelism. The fusion optimization only improves performance in the four-machine case, where it achieves a 4.02× speedup, which is overall better than the four-machine results without fusion. Figure 20(c) shows that fusion is particularly beneficial to the Sawzall log query analyzer. In this case, fusion eliminates much of the per-data-item processing overhead and therefore allows the fission optimization to yield much better results. Fusion combines each mapper with its splitter and each reducer with its merger. With both fusion and fission, the Sawzall log query analyzer speeds up 50.32× on 16 machines with 64 degrees of parallelism. Note that in our implementation, when two operators are on the same computer, but not fused, the system serializes messages between them, adding unnecessary overhead. Our survey on streaming optimizations [40] describes alternative implementations, including simply passing the message data by reference.

Calibration to native implementation. The placement, fission, and fusion optimizations demonstrate that River can support a diverse set of streaming optimizations, thus validating our design. While we did not focus our implementation efforts on absolute performance metrics, we were interested to see how River's performance compared with a native language implementation. The Sawzall compiler translates Sawzall programs into MapReduce jobs. We therefore implemented the Sawzall web log analyzer query as a Hadoop job. We ran both Hadoop and River on a cluster of 16 machines, with an input set of 10^9 data items, around 24 GB. Hadoop was able to process this data in 96 s. Given that there are 16 machines with 8 cores each, we ran River with 64 mappers and 64 reducers ($16 * 8 = 64 + 64 = 128$). Under this scenario, River completed the computation in 115 s.

Despite the fact that Hadoop is a well-established system that has been heavily used and optimized, the River prototype ran about 83% as fast. There are two reasons for this performance difference. First, our current fission optimizer always replicates the same number of mappers as reducers, which is not an optimal configuration. Additional optimization work is needed to adjust the map-to-reduce ratio. Second, our implementation has some unnecessary serialization overheads.

A preliminary investigation suggests that eliminating those would give us performance on par with Hadoop.

7.3. Concurrency

This section explores the effectiveness of the locking algorithm in Figure 2. We compare River with two alternative locking algorithms: (i) *course grained*, in which there is a single global lock for all shared variables, and (ii) *fine grained*, in which each shared variable has its own lock.

Coarse-grained locking. For the coarse-grained locking experiment, we used the following setup, described in River's topology language:

```
(q2, $v1)         <- f1(q1, $v1      );
(q3, $v1, $v2)    <- f2(q2, $v1, $v2);
(q4,      $v2)    <- f3(q3,      $v2);
```

In this example, `f1` and `f3` are expensive (implemented by sleeping for a set time), whereas `f2` is cheap (implemented as the identity function). Operator instances `f1` and `f2` share the variable `$v1`, and `f2` and `f3` share the variable `$v2`. With coarse-grained locking, we would put all three variables under a single lock. That means that all three operator instances are mutually exclusive, and we would expect the total execution time to be approximately

$$cost(\texttt{f1}) + cost(\texttt{f2}) + cost(\texttt{f3})$$

On the other hand, with our smarter locking scheme, `f1` and `f3` have no locks in common and can therefore execute in parallel. Hence, you would expect an approximate total execution time of

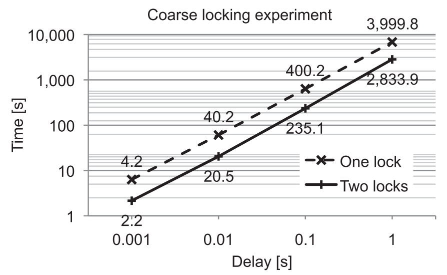
$$\max\{cost(\texttt{f1}), cost(\texttt{f3})\} + cost(\texttt{f2})$$

We tested this by varying the cost (i.e., delay) of the operators `f1` and `f3` over a range of 0.001 to 1 s. The results in Figure 21(a) behave as we expected, with our locking scheme performing approximately twice as fast as with the coarse-grained version.

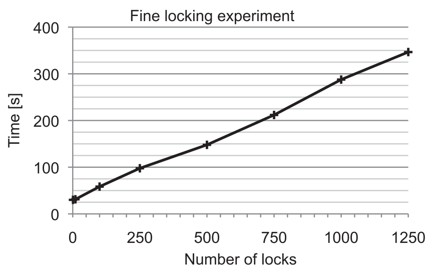
Fine-grained locking. As a second micro-benchmark for our locking algorithm, we wanted to quantify the overhead of locking on a fine-grained level. We used the following setup, described in River's topology language:

```
(q2, $v1, ..., $vn) <- f1(q1, $v1, ..., $vn);
(q3, $v1, ..., $vn) <- f2(q2, $v1, ..., $vn);
```

Operators `f1` and `f2` share variables `$v1` ... `$vn`, where we incremented `n` from 1 to 1250. With fine-grained locking, each variable would be protected by its own lock, as opposed to our locking scheme, which protects all `n` variables under a single lock. Figure 21(b) shows that the overhead of the fine-grained scheme grows linearly with the number of locks, just as expected.



(a) Coarse locking



(b) Fine locking

Figure 21. Locking experiments.

8. RELATED WORK

River is an *execution environment for streaming*, which runs on a distributed system and supports multiple source languages. SVM, the stream virtual machine, is a C framework for streaming on both CPU and graphics processing unit back-ends, but the paper does not describe any source language support [41]. MSL, the multicore streaming layer, executes only the StreamIt language on the cell architecture [42]. Erbium is a set of intrinsic functions for a C compiler for streaming on a $\times 86$ symmetric multiprocessing, but the paper describes no source language support [43]. And Lime is a new streaming language with three back-ends: Java, C, and field-programmable gate array bit-files [2]. None of SVM, MSL, Erbium, or Lime are distributed on a cluster, none of them are founded on a calculus, and they all have at most a single source language each. While we are not aware of prior work on execution environments for distributed streaming, there are various execution environments for distributed batch data flow. MapReduce [17] has emerged as a *de facto* execution environment for various batch-processing languages, including Sawzall [4], Pig Latin [44], and FlumeJava [45]. Dryad [46] is a batch execution environment that comes with its own language, Nebula, but is also targeted by DryadLINQ [47]. CIEL is a batch execution environment with its own language SkyWriting [48]. Like MapReduce, Dryad, and CIEL, River runs on a shared-nothing cluster, but in contrast, River is designed for continuous streaming and is derived from a calculus. Some recent distributed batch analytics systems have moved towards a streaming nature. Naiad uses multi-dimensional timestamps to coordinate both streaming inputs and iterative computation [49]. And Spark Streaming chops streams into small batches amenable for streaming use-cases that do not require low-latency processing [50].

Brooklet is a *calculus and semantics for stream processing*. It is inspired by other work on core minimal languages, such as Featherweight Java [51]. There has been extensive prior work in the semantics of stream processing. Stephens [52] provides a comprehensive survey, but it does not address recent language developments. Brooklet differs from prior work on streaming semantics because it models state and non-determinism as explicit core concepts. Kahn's process networks [53] assume deterministic execution. Synchronous data flow [27] models, such as StreamIt, assume fixed buffer sizes and static communication patterns. Hoare's communicating sequential processes [54] assume no buffering and synchronous communication. Gurevich *et al.* [11] studied streaming systems but focused on their more theoretical aspects.

The database literature often refers to streaming applications as *continuous queries*. Prominent examples of streaming systems from the database community include NiagaraCQ [55] and Tapestry [56]. Surprisingly, there is little work from the database community on optimizations of queries with side effects. Two exceptions are a study of XQuery with side effects [57] and a study of object-oriented databases [58].

Stream processing is closely related to *complex event processing* (CEP) [59–62], with the latter placing greater emphasis on the time intervals between the arrival of data items processed by an operator. Because CEP functionality can be encapsulated in a streaming operator [63], River should also be able to serve as a substrate for those languages.

River comes with an ecosystem for *economic source language development*. The LINQ (language integrated query) framework also facilitates front-end implementation, but using a different approach: LINQ embeds SQL-style query syntax in a host language such as C#, and targets different back-ends such as databases or Dryad [64]. Our approach follows more traditional compiler frameworks such as Polyglot [65] or MetaBorg [66]. We use the *Rats!* parser generator to modularize grammars [22]. Our approach to modularizing type checkers and code generators has some resemblance to Jeannie [21], although Jeannie is not related to stream processing, and River composes more language components at a finer granularity.

Several communities have come up with similar *streaming optimizations*, but unlike River, they do not decouple the optimizations from the source language translator and reuse them across different source languages [40]. In *parallel databases* [32], the IL is RA. Similarly to the fission optimization in this paper, parallel databases use hash-splits for parallel execution. But to do so, they rely on algebraic properties of a small set of built-in operators. In contrast, River supports an unbounded set of user-defined operators. There has been surprisingly little work on generalizing database optimizations to the more general case [57, 58], and that work is still limited to the database domain. The StreamIt compiler implements its own variant of fission [19]. It relies on fixed data rates and stateless operators for safety, and indeed, the StreamIt language is designed around making those easy to establish. Our fission is more general, because it can parallelize even in the presence of state. MapReduce has data parallelism hard-wired into its design. Safety hinges upon keys and commutativity, but those are up to the user or a higher-level language to establish [67]. River supports language-independent optimization by making such language-specific safety properties explicit in the IL.

9. FUTURE WORK

So far, the main focus of our implementation efforts was to provide a common substrate for optimization and to demonstrate the correctness and utility of the language-agnostic optimizations. An obvious next step for River is to improve absolute performance metrics. As discussed in Section 7, our prototype implementation could be improved in several ways. First, our fission optimization, which is the enabler for data parallelization, would benefit from being able to specify a differing number of mappers as reducers. Second, the serialization we used to implement River adds unnecessary overhead when sending data. Third, we have not studied the performance overhead when crossing language boundaries between C++ and OCaml. Fourth, we have not investigated how OCaml's garbage collection or its single-threaded runtime impacts performance.

River lays the ground work for a variety of future work, including formalization of additional languages, invention of new abstractions to expose and exploit parallelism, alternative translations for the languages we formalized, reverse translations from River back into source languages, type systems work, exploration of time or space resource constraints, investigations of progress, fairness, and deadlock, static analyses for establishing optimization preconditions, and specifications of additional optimizations [40].

10. CONCLUSION

This paper presents a calculus and execution environment for distributed stream processing. The Brooklet calculus provides a formal model for streaming languages, which allows for a rigorously defined semantics and enables reasoning about the correctness of translations and optimizations. The River execution environment extends the calculus to provide a common target for language translation, which both improves the portability of source languages and encourages reuse of streaming optimizations.

We have formally described the implementation of three language front-ends and three crucial optimizations targeting River. Additionally, we have demonstrated how to reuse common language components across diverse languages, thus limiting the effort of writing parsers, type checkers, and code generators to the unique features of each language. Finally, we have provided three streaming benchmarks and measured the effects of applying the common optimizations. As a byproduct of our work, we provided the first formal semantics for Sawzall and the first distributed implementation of CQL.

Overall, this work provides a formal foundation for the continued study of streaming language and optimization design while providing an intermediate language that significantly reduces the effort it takes to develop stream-processing languages, thus enabling further innovation.

ACKNOWLEDGEMENTS

We would like to thank John Field, Rodric Rabbah, and Martin Vechev for their feedback on earlier versions of this paper and Nagui Halim for his support of this project.

REFERENCES

1. Arasu A, Babu S, Widom J. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 2006; **15**(2):121–142.
2. Auerbach J, Bacon DF, Cheng P, Rabbah R. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Reno/Tahoe Nevada, USA, October 2010; 89–108.
3. Gedik B, Andrade H, Wu K-L, Yu PS, Doo M. SPADE: the System S declarative stream processing engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008; 1123–1134.
4. Pike R, Dorward S, Griesemer R, Quinlan S. Interpreting the data: parallel analysis with Sawzall. *Scientific Programming* 2005; **13**(4):277–298.
5. Thies W, Karczmarek M, Amarasinghe S. StreamIt: a language for streaming applications. *Proceedings of the 11th International Conference on Compiler Construction*, Lecture Notes in Computer Science, vol. 2304, Grenoble, France, April 2002; 179–196.
6. Soulé R, Hirzel M, Grimm R, Gedik B, Andrade H, Kumar V, Wu K-L. A universal calculus for stream processing languages. *Proceedings of the 19th European Symposium on Programming*, Lecture Notes in Computer Science, vol. 6012, Paphos, Cyprus, March 2010; 507–528.
7. Soulé R, Hirzel M, Gedik Buğra, Grimm R. From a calculus to an execution environment for stream processing. *Proceedings of the 6th International Conference on Distributed Event-Based Systems*, Berlin, Germany, July 2012; 20–31.
8. Amini L, Andrade H, Bhagwan R, Eskesen F, King R, Selo P, Park Y, Venkatramani C. SPC: a distributed, scalable platform for data mining. *Proceedings of the 4th International Workshop on Data Mining Standards, Services, and Platforms*, Philadelphia, PA, USA, August 2006; 27–37.
9. Nielson HR, Nielson F. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc.: New York, New York, 1992.
10. Pierce BC. *Types and Programming Languages*. MIT Press: Cambridge, Massachusetts, 2002.
11. Gurevich Y, Leinders D, Van Den Bussche J. A theory of stream queries. *Proceedings of the 11th International Conference on Database Programming Languages*, Lecture Notes in Computer Science, vol. 4797, Vienna, Austria, September 2007; 153–168.
12. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis and transformation. *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, San Jose, CA, USA, March 2004; 75–88.
13. Grimm R, Davis J, Lemar E, MacBeth A, Swanson S, Anderson T, Bershad B, Borriello G, Gribble S, Wetherall D. System support for pervasive applications. *ACM Transactions on Computer Systems* 2004; **22**(4):421–486.
14. Mogul JC, Ramakrishnan KK. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 1997; **15**(3):217–252.
15. Gordon MI, Thies W, Karczmarek M, Lin J, Meli AS, Lamb AA, Leger C, Wong J, Hoffmann H, Maze D, Amarasinghe S. A stream compiler for communication-exposed architectures. *Proceedings of the 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, December 2002; 291–303.
16. Hirzel M, Andrade H, Gedik B, Jacques-Silva G, Khandekar R, Kumar V, Mendell M, Nasgaard H, Schneider S, Soulé R, Wu K-L. IBM streams processing language: analyzing big data in motion. *IBM Journal of Research and Development* 2013; **57**(3/4):7:1–7:11.
17. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004; 137–150.
18. Drake M, Hoffmann H, Rabbah R, Amarasinghe S. MPEG-2 decoding in a stream programming language. *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006; 86–95.
19. Gordon MI, Thies W, Amarasinghe S. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 2006; 151–162.
20. Arasu A, Widom J. A denotational semantics for continuous queries over streams and relations. *ACM SIGMOD Record* 2004; **33**(3):6–11.
21. Hirzel M, Grimm R. Jeannie: granting Java native interface developers their wishes. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Montreal, Quebec, Canada, October 2007; 19–38.
22. Grimm R. Better extensibility through modular syntax. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Ottawa, Ontario, Canada, June 2006; 38–51.
23. Zou Q, Wang H, Soulé R, Hirzel M, Andrade H, Gedik B, Wu K-L. From a stream of relational queries to distributed stream processing. *Proceedings of the 36th International Conference on Very Large Data Bases*, Singapore, 2010; 1394–1405.
24. Garcia-Molina H, Ullman JD, Widom J. *Database Systems: The Complete Book* (2nd edn). Prentice Hall: Upper Saddle River, NJ, 2008.

25. Lämmel R. Google's MapReduce programming model — revisited. *Science of Computer Programming* 2007; **68**(3):208–237.
26. Thies W, Karczmarek M, Gordon M, Maze D, Wong J, Hoffmann H, Brown M, Amarasinghe S. StreamIt: a compiler for streaming applications. *Technical Report MIT-LCS-TM-622*, Massachusetts Institute of Technology: Cambridge, Massachusetts, 2001.
27. Lee EA, Messerschmitt DG. Synchronous data flow. *Proceedings of the IEEE* 1987; **75**(9):1235–1245.
28. Soulé R, Gordon MI, Amarasinghe S, Grimm R, Hirzel M. Dynamic expressivity with static optimization for streaming languages. *Proceedings of the 7th International Conference on Distributed Event-Based Systems*, Arlington, TX, USA, 2013; 159–170.
29. Soulé R, Hirzel M, Grimm R, Gedik B, Andrade H, Kumar V, Wu K-L. A universal calculus for stream processing languages. *Technical Report NYU-CS-TR2010-924*, New York University: New York, NY, March 2010.
30. Khandekar R, Hildrum I, Parekh S, Rajan D, Wolf J, Wu K-L, Andrade H, Gedik B. COLA: optimizing stream processing applications via graph partitioning. *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Lecture Notes in Computer Science, vol. 5896, Urbana Champaign, Illinois, USA, November 2009; 308–327.
31. Tang Y, Gedik B. Auto-pipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 2012; **10.1109**(333):2344–2354.
32. DeWitt D, Gray J. Parallel database systems: the future of high performance database systems. *Communications of the ACM* 1992; **35**(6):85–98.
33. Xu Z, Hirzel M, Rothermel G, Wu K-L. Testing properties of dataflow program operators. *Proceedings of the 28th IEEE/ACM Conference on Automated Software Engineering*, Palo Alto, CA, USA, November 2013; 103–113.
34. Wolf J, Bansal N, Hildrum K, Parekh S, Rajan D, Wagle R, Wu K-L, Fleischer L. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Lecture Notes in Computer Science, vol. 5346, Leuven, Belgium, December 2008; 306–325.
35. Pietzuch P, Ledlie J, Schneidman J, Roussopoulos M, Welsh M, Seltzer M. Network-aware operator placement for stream-processing systems. *Proceedings of the 22nd International Conference on Data Engineering*, Atlanta, GA, USA, April 2006; 49–61.
36. Hormati AH, Choi Y, Kudlur M, Rabbah R, Mudge T, Mahlke S. Flextream: adaptive compilation of streaming applications for heterogeneous architectures. *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Raleigh, North Carolina, USA, September 2009; 214–223.
37. Ke Q, Isard M, Yu Y. Optimus: a dynamic rewriting framework for data-parallel execution plans. *Proceedings of the 8th European Conference on Computer Systems*, Prague, Czech Republic, April 2013; 15–28.
38. Avnur R, Hellerstein JM. Eddies: continuously adaptive query processing. *ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, June 2000; 261–272.
39. Nathan S, Gedik B. Using InfoSphere streams with memcached and Redis: a large-scale in-memory distributed process store, October 2013. Available at: http://www.ibm.com/developerworks/library/bd-streamsmemcached [last accessed 22 October 2013].
40. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A catalog of stream processing optimizations. *ACM Computing Surveys* 2014; **46**(4):46:1–46:34.
41. Labonte F, Mattson P, Thies W, Buck I, Kozyrakis C, Horowitz M. The stream virtual machine. *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September/October 2004; 267–277.
42. Zhang D, Li QJ, Rabbah R, Amarasinghe S. A lightweight streaming layer for multicore execution. *ACM SIGARCH Computer Architecture News* 2008; **36**(2):18–27.
43. Miranda C, Pop A, Dumont P, Cohen A, Duranton M. Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Scottsdale, AZ, USA, October 2010; 11–20.
44. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: a not-so-foreign language for data processing. *Proceedings of the CM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 2008; 1099–1110.
45. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N. FlumeJava: easy, efficient data-parallel pipelines. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Toronto, ON, Canada, June 2010; 363–375.
46. Isard M, Yu MBY, Birrell A, Fetterly D. Dryad: distributed data-parallel program from sequential building blocks. *Proceedings of the 2nd European Conference on Computer Systems*, Lisbon, Portugal, March 2007; 59–72.
47. Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson Ú, Gunda PK, Currey J. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, California, USA, December 2008; 1–14.
48. Murray DG, Schwarzkopf M, Smowton C, Smith S, Madhavapeddy A, Hand S. CIEL: a universal execution engine for distributed data-flow computing. *Proceedings of the 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, March 2011; 113–126.
49. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: a timely dataflow system. *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, Pennsylvania, USA, November 2013; 439–445.

50. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: fault-tolerant streaming computation at scale. *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, Pennsylvania, USA, 2013; 423–438.
51. Igarashi A, Pierce BC, Wadler P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 2001; **23**(3):396–450.
52. Stephens R. A survey of stream processing. *Acta Informatica* 1997; **34**(7):491–541.
53. Kahn G. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, Rosenfeld JL (ed.). North Holland Publishing Company: Amsterdam, 1974; 471–475.
54. Hoare CAR. Communicating sequential processes. *Communications of the ACM* 1978; **21**(8):666–677.
55. Chen J, DeWitt DJ, Tian F, Wang Y. NiagaraCQ: a scalable continuous query system for internet databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, May 2000; 379–390.
56. Terry D, Goldberg D, Nichols D, Oki B. Continuous queries over append-only databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 1992; 321–330.
57. Ghelli G, Onose N, Rose K, Siméon J. XML query optimization in the presence of side effects. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008; 339–352.
58. Fegaras L. Optimizing queries with object updates. *Journal of Intelligent Information Systems* 1999; **12**(2–3): 219–242.
59. Adi A, Etzion O. Amit — the situation manager. *The VLDB Journal* 2004; **13**(2):177–203.
60. Barga RS, Goldstein J, Ali M, Hong M. Consistent streaming through time: a vision for event stream processing. *Proc. Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2007; 363–374.
61. Brenna L, Gehrke J, Johansen D, Hong M. Distributed event stream processing with non-deterministic finite automata. *Proceedings of the 3rd International Conference on Distributed Event-Based Systems*, Nashville, TN, USA, 2009. Article No. 3.
62. Wu E, Diao Y, Rizvi S. High-performance complex event processing over streams. *Proc. ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, 2006; 407–418.
63. Hirzel M. Partition and compose: parallel complex event processing. *Proceedings of the 6th International Conference on Distributed Event-Based Systems*, Berlin, Germany, 2012; 191–200.
64. Meijer E, Beckman B, Bierman G. LINQ: reconciling object, relations and XML in the .NET framework. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, June 2006; 706.
65. Nystrom N, Clarkson MR, Myers AC. Polyglot: an extensible compiler framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, vol. 2622, Warsaw, Poland, April 2003; 138–152.
66. Bravenboer M, Visser E. Concrete syntax for objects. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 2004; 365–383.
67. Xu Z, Hirzel M, Rothermel G. Semantic characterization of MapReduce workloads. *Proceedings of the International Symposium on Workload Characterization*, Portland, OR, USA, September 2013; 87–97.