A Universal Calculus for Stream Processing Languages

Robert Soulé[1], Martin Hirzel[2], Robert Grimm[1], Buğra Gedik[2], Henrique Andrade[2], Vibhore Kumar[2], and Kun-Lung Wu[2]

[1] New York University
{soule,rgrimm}@cs.nyu.edu
[2] IBM Research
{hirzel,bgedik,hcma,vibhorek,klwu}@us.ibm.com

Abstract. Stream processing applications such as algorithmic trading, MPEG processing, and web content analysis are ubiquitous and essential to business and entertainment. Language designers have developed numerous domain-specific languages that are both tailored to the needs of their applications, and optimized for performance on their particular target platforms. Unfortunately, the goals of generality and performance are frequently at odds, and prior work on the formal semantics of stream processing languages does not capture the details necessary for reasoning about implementations. This paper presents Brooklet, a core calculus for stream processing that allows us to reason about how to map languages to platforms and how to optimize stream programs. We translate from three representative languages, CQL, StreamIt, and Sawzall, to Brooklet, and show that the translations are correct. We formalize three popular and vital optimizations, data-parallel computation, operator fusion, and operator re-ordering, and show under which conditions they are correct. Language designers can use Brooklet to specify exactly how new features or languages behave. Language implementors can use Brooklet to show exactly under which circumstances new optimizations are correct. In ongoing work, we are developing an intermediate language for streaming that is based on Brooklet. We are implementing our intermediate language on System S, IBM's high-performance streaming middleware.

1 Introduction

Stream processing applications are everywhere. In finance, algorithmic trading programs federate live data feeds from independent exchanges to execute trade orders. Media players decode fixed-rate, MPEG-formatted byte streams, when viewers watch video streamed over the internet and digital television networks, or from DVD and Blu-ray discs. Search engines use large compute clusters to analyze snapshots of the web streamed from disk to construct the indices that enable fast information retrieval.

Informally, all such *streaming applications* are similar in that they require moving large amounts of data through several computational steps. These three

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 507–528, 2010.

examples illustrate the diversity of requirements for stream processing with respect to, among other things, program topology, data rate, and distributed execution. This diversity has led language designers to develop numerous domain-specific languages [1,3,4,9,18,20,24,26,28] that are both tailored to the needs of their particular applications, and optimized for performance on their particular target runtimes. Three prominent examples are CQL, StreamIt, and Sawzall:

- CQL [1] and other StreamSQL dialects [24] are popularly used for algorithmic trading. CQL extends SQL's 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.
- StreamIt [26], a synchronous data-flow language with stream abstractions, has been used for MPEG encoding and decoding [6]. The StreamIt compiler enforces static data transfer rates between user-defined operators with fixed topologies, and improves performance through operator fusion, fission, and pipelining [26].
- Sawzall [20], a scripting language for Google's MapReduce [5] platform, is used for web-related analysis. 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.

These three examples by no means comprise an exhaustive list of stream programming languages, but they are representative of the design space. In each case, language designers made difficult choices when considering the trade-offs between performance, usability, and generality. For example, StreamIt sacrifices generality for performance by restricting data transfer to fixed rates.

When considering these trade-offs, it is essential that language designers understand both how a language maps to its target platform, and how to optimize stream programs with respect to that mapping. Unfortunately, while streaming systems are well studied [2,14,15,16], prior work on the formal semantics of stream processing languages does not capture the details necessary for reasoning about implementation techniques. This paper presents Brooklet, a core calculus for stream programming languages that universally models any streaming language, and facilitates reasoning about program implementation[1].

The challenge in defining a calculus is deciding what parts of a language constitute the core concepts that need to be modeled in the formal semantics, and what details can be abstracted away. The two goals of understanding how a language maps to a platform, and how to optimize stream programs with respect to that mapping, dictate the requirements. First, to understand how a language maps to an execution environment, we need to understand how the state embodied in its operational building blocks is implemented on a distributed platform. Therefore, Brooklet makes state explicit as a core concept. Second, to understand how to

[1] Brooklet is so named because it is the essence of a stream, and is unrelated to the Brook language [3].

optimize stream programs, we need to understand how to enable language-level determinism on top of the inherent implementation-level non-determinism of a distributed system. Therefore, Brooklet exposes non-determinism as another core concept. Exposing non-determinism makes the machinery for achieving global determinism explicit, such as when implementing synchronous data flow. On the other hand, modeling local deterministic computations is well-understood, so our semantics treat local computations as opaque functions. Since our semantics are small-step, this abstraction loses none of the fine-grained interleaving effects of the distributed computation.

In this paper we make the following contributions:

- We define a core calculus for stream processing that is universal, and facilitates reasoning about program implementation by modeling state and non-determinism as core concepts.
- We translate CQL, StreamIt, and Sawzall to Brooklet, demonstrating the comprehensiveness of our calculus. This translation also defines the first formal semantics for Sawzall.
- We use our calculus to show the conditions that enable three vital optimizations data-parallel computation, operator fusion, and operator re-ordering.

This sets a foundation for an implementation of Brooklet, which can serve as a common intermediate language for stream processing with a rigorous formal semantics. We are in the process of exploring this implementation on System S [9], IBM's high-performance streaming middleware.

2 Notation

Throughout the paper, an over-bar, as in $\overline{q}$, denotes a finite sequence $q_1, \ldots, q_n$, and the i-th element in that sequence is written q_i, where $1 \leq i \leq n$. The lowercase 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 e's 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 $[\![P_b]\!]^p_z$ indicate translation. The subscripts $_{b,c,s,z}$ stand for Brooklet, CQL, StreamIt, and Sawzall, respectively.

3 Brooklet

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

Brooklet syntax:

P_b	::= *out in* $\overline{op}$	*Brooklet program*
out	::= `output` $\overline{q}$ `;`	*Output declaration*
in	::= `input` $\overline{q}$ `;`	*Input declaration*
op	::= `(` $\overline{q}$, $\overline{v}$ `)` ← *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: $F_b \vdash \langle V, Q\rangle \longrightarrow \langle V', Q'\rangle$

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

$$\frac{op = (\_, \overline{v}) \leftarrow f(\_, \_);}{updateV(op, V, \overline{d}) = [\overline{v} \mapsto \overline{d}]V} \text{ (E-UpdateV)}$$

$$\frac{\begin{array}{c} op = (\overline{q}, \_) \leftarrow f(\_, \_); \\ d_f, b_f = Q(q_f) \\ Q' = [q_f \mapsto b_f]Q \\ Q'' = [\forall q_i \in \overline{q} : q_i \mapsto Q(q_i), b_i]Q' \end{array}}{updateQ(op, Q, q_f, \overline{b}) = Q''} \text{ (E-UpdateQ)}$$

Fig. 1. Brooklet syntax and semantics

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-determinism as core concepts, and abstracts away local deterministic computations.

3.1 Brooklet Program Example: IBM Market Maker

As an example of a streaming program, we consider a hypothetical application that trades IBM stock. Data arrives on two input streams, `bids(symbol,price)` and `asks(symbol,price)`, and leaves on the `result(cnt,symbol,price)` output stream. Since 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 ask prices from the filtered streams to make trades. To keep the example simple, we assume that each sale is for exactly one share. The Brooklet program in the bottom left corner of Fig. 1 produces a stream of trades of IBM stock, along with a count of the number of trades.

3.2 Brooklet Syntax

A Brooklet program defines a directed, possibly cyclic, graph of *operators* containing pure *functions* connected by FIFO *queues*. It uses *variables* to explicitly thread state through operators. Data items on a queue model network packets in transit. Data items in variables model stored state; since data items may be lists, 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 [17,19],

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, since they are intended to thread state through a streaming application. In contrast, each queue must be defined once and used once. This restriction facilitates using our semantics for proofs and optimizations. The complete Brooklet grammar appears in Fig. 1.

3.3 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.

3.3.1 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, $F_b(\texttt{SelectIBM})$ 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).

3.3.2 Brooklet Execution Semantics

Computation proceeds in small steps. Each step fires Rule E-FireQueue from Fig. 1. To explain this rule, we illustrate each line rule one by one, starting with the following intermediate configuration of the market maker example:

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

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

$d, b = Q(q_i)$: 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.
E.g., $q_i = \texttt{ibmBids}$, $d = \langle \text{`IBM'}, 119\rangle$, $b = \big(\langle \text{`IBM'}, 124\rangle\big)$.

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

$(\overline{b}', \overline{d}') = F_b(f)(d, i, V(\overline{v}))$: 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.
E.g., $\overline{b}' = \Big(\big(\langle \text{`IBM'}, 119, 119\rangle\big)\Big)$, $\overline{d}' = \bullet$,
$d = \langle \text{`IBM'}, 119\rangle$, $i = 1$, $V(\overline{v}) = \langle \text{`IBM'},119\rangle$.

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

$Q' = \mathit{updateQ}(op, Q, q_i, \overline{b}')$: Update the queues: remove the popped data item from the firing queue, and for each output queue, push the corresponding list of output data items. The example has only one output queue and datum.
E.g., $Q' = \begin{bmatrix} \texttt{bids} \mapsto \bullet, & \texttt{ibmBids} \mapsto \big(\langle \text{`IBM'}, 124\rangle\big), \\ \texttt{asks} \mapsto \bullet, & \texttt{ibmAsks} \mapsto \bullet, \\ \texttt{ibmSales} \mapsto \big(\langle \text{`IBM'}, 119, 119\rangle\big), & \texttt{result} \mapsto \bullet \end{bmatrix}$

3.4 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. Due to non-determinism, the set may have more than one element. One possible output O_b of our running example is:

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

$$Q = \left[\begin{array}{ccc} \texttt{bids} \mapsto \bullet, & \texttt{asks} \mapsto \bullet, & \texttt{ibmSales} \mapsto \bullet, \\ \texttt{ibmBids} \mapsto \bullet, & \texttt{ibmAsks} \mapsto \bullet, & \texttt{result} \mapsto \{\langle 1, \text{'IBM'}, 119\rangle\} \end{array}\right]$$

The example illustrates the finite case. But in some 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.

3.5 Brooklet Summary

Brooklet is a core calculus for stream processing. We designed it to universally model any streaming language, 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 nondeterminism by *not* specifying which queue to fire for each step, thus permitting all interleavings possible in a distributed implementation.

4 Language Mappings

We demonstrate Brooklet's generality by mapping three streaming languages CQL, StreamIt, and Sawzall to it. Each translation exposes implicit uses of state as explicit variables; exposes a mechanism for implementing global determinism on top of an inherently non-deterministic runtime; and abstracts away local deterministic computations with higher-order wrappers that statically bind the original function and dynamically adapt the runtime arguments (thus preserving small step semantics).

4.1 CQL and Stream-Relational Algebra

CQL, the 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 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 [2].

4.1.1 CQL Program Example: Bargain Finder

A CQL program P_c is a query that computes a stream or relation from other streams or relations. The following hypothetical example uses CQL for algorithmic trading:

```
select IStream(*) from quotes[Now], history
  where quotes.ask <= history.low and quotes.ticker == history.ticker
```

This 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`. A *stream* in CQL is a bag of time-tagged tuples. The same

CQL syntax:

$P_c ::= P_{cr} \mid P_{cs}$ *CQL program*

$P_{cr} ::=$ *(Relation query)*

$RName$ *Relation name*

$\mid S2R(P_{cs})$ *Stream to relation*

$\mid R2R(\overline{P_{cr}})$ *Relation to relation*

$P_{cs} ::=$ *(Stream query)*

$SName$ *Stream name*

$\mid R2S(P_{cr})$ *Relation to stream*

$RName \mid SName ::= id$ *Input name*

$S2R \mid R2R \mid R2S ::= id$ *Operator name*

CQL example: Bargain finder.

```
IStream(BargainJoin(Now(quotes), history))
```

CQL program translation: $[\![F_c, P_c]\!]^p_c = \langle F_b, P_b \rangle$

$$\frac{[\![F_c, SName]\!]^p_c = \emptyset, \texttt{output}\, SName; \texttt{input}\, SName; \bullet}{} \quad (\mathrm{T}^p_c\text{-SName})$$

$$\frac{[\![F_c, RName]\!]^p_c = \emptyset, \texttt{output}\, RName; \texttt{input}\, RName; \bullet}{} \quad (\mathrm{T}^p_c\text{-RName})$$

$$\frac{\begin{array}{c} F_b, \texttt{output}\ q_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op} = [\![F_c, P_{cs}]\!]^p_c \\ q'_o = freshId() \qquad v = freshId() \\ F'_b = [S2R \mapsto wrapS2R(F_c(S2R))]F_b \\ \overline{op}' = \overline{op}, (q'_o, v) \leftarrow S2R(q_o, v); \end{array}}{[\![F_c, S2R(P_{cs})]\!]^p_c = F'_b, \texttt{output}\ q'_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op}'} \quad (\mathrm{T}^p_c\text{-S2R})$$

$$\frac{\begin{array}{c} F_b, \texttt{output}\ q_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op} = [\![F_c, P_{cr}]\!]^p_c \\ q'_o = freshId() \qquad v = freshId() \\ F'_b = [R2S \mapsto wrapR2S(F_c(R2S))]F_b \\ \overline{op}' = \overline{op}, (q'_o, v) \leftarrow R2S(q_o, v); \end{array}}{[\![F_c, R2S(P_{cr})]\!]^p_c = F'_b, \texttt{output}\ q'_o\texttt{;}\ \texttt{input}\ \overline{q}\texttt{;}\ \overline{op}'} \quad (\mathrm{T}^p_c\text{-R2S})$$

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

CQL domains:

$\tau \in \mathcal{T}$ *Time*

$e \in \mathcal{TP}$ *Tuple*

$\sigma \in \Sigma = \mathrm{bag}(\mathcal{TP})$ *Instantaneous relation*

$r \in \mathcal{R} = \mathcal{T} \to \Sigma$ *Time-varying relation*

$s \in \mathcal{S} = \mathrm{bag}(\mathcal{TP} \times \mathcal{T})$ *Time-varying stream*

CQL operator signatures:

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

CQL operator wrapper signatures:

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

CQL operator wrappers:

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

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

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

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

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

Fig. 2. CQL semantics on Brooklet

information can be more conveniently represented as a mapping from time stamps to bags of tuples. CQL calls such a mapping a *time-varying relation*, and each individual bag of tuples an *instantaneous relation*. In the example, input `history(ticker,low)` is the time-varying relation r_h:

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

The instantaneous relation $r_h(1)$ is $\{\langle \text{'IBM'}, 119\rangle, \langle \text{'XYZ'}, 38\rangle\}$. The CQL stream s_q represents the input `quotes(ticker,ask)`:

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

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

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

The next step of the query joins the quote relation r_q with the history relation r_h into a bargains relation r_b:

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

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

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

While CQL uses select-from-where syntax, the CQL semantics use an equivalent stream-relational algebra syntax (similar to relational algebra in databases):

```
IStream(BargainJoin(Now(quotes), history))
```

This algebraic notation makes the operator tree clearer. The leaves are stream name `quotes` and relation name `history`. CQL has three categories of operators. S2R operators turn a stream into a relation; e.g., `Now(quotes)` turns stream `quotes` into relation r_q. R2R operators turn one or more relations into a new relation; e.g., `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; e.g., `IStream(`r_b`)` turns relation r_b into the stream of its insertions. CQL has no S2S operators, because they would be redundant. CQL's R2R operators coincide with traditional database relational algebra.

The CQL grammar is in Fig. 2. A CQL 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.

4.1.2 CQL Implementation Issues

Before we translate CQL to Brooklet, let us discuss the two issues of state and non-determinism in CQL.

CQL state. CQL represents global state explicitly as named relations, such as the `history` relation from our running example. But in addition, all three kinds of CQL 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. 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 time stamps is available for all relations.

CQL non-determinism. CQL is deterministic in the sense that the output of a program is fully determined by the times and values of its inputs [2]. 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 time stamps. A CQL implementation might either assign time stamps upon receiving data, or use time stamps that are an inherent part of the input data, such as trading times. However, CQL implementations can permit

non-determinism to exploit parallelism. For example, the implementation need 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 time stamp.

Translation to Brooklet will make all state explicit, and will clarify how the implementation enforces determinism.

4.1.3 CQL Translation Example

Given the CQL example program from Fig. 2, the translation to Brooklet is the program P_b:

```
output qo;
input quotes, history;
(qq, $vn)       ⟵ wrapNow(quotes, $vn);
(qb, $vq, $vh) ⟵ wrapBargainJoin(qq, history, $vq, $vh);
(qo, $vo)       ⟵ wrapIStream(qb, $vo)
```

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 time stamp to one output instantaneous relation. Brooklet models the choice of interleavings by allowing either queue q_q or `history` to fire independently. Hence, the Brooklet operator processes one data item each time either queue fires. Assume a data item arrives on the first queue q_q. If there is already a data item with the same time stamp in the variable v_h associated with the second queue, Brooklet performs the join, which may yield data items for the output queue q_b. Otherwise, it simply stores the data item in v_q for later.

4.1.4 CQL Translation

Fig. 2 shows the translation from CQL 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 CQL functions. Working in concert, the rules enforce a global convention: the execution sends exactly one instantaneous relation on every queue at every time stamp. Operators retain historical data in variables, e.g., to implement windows.

4.1.5 CQL Discussion

CQL is an SQL dialect for streaming [1]. Arasu and Widom specify big-step denotational semantics for CQL [2]. We show how to translate CQL to Brooklet, thus giving an alternative semantics. As we will show below, both semantics define equivalent input/output behavior for CQL programs. Translations from

other languages can use similar techniques, i.e., make state explicit as variables; wrap computation in small-step firing functions; and define a global convention for how to achieve determinism.

4.2 StreamIt and Synchronous Data Flow

StreamIt [27,26] is a streaming language tailored for parallel implementations of applications such as MPEG decoding [6]. At its core, StreamIt is a synchronous data flow (SDF) language [16], 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 [6], which delivers control messages between operators and which could be modeled in Brooklet through shared variables.

4.2.1 StreamIt Program Example: MPEG Decoder

The following example StreamIt program P_s is based on a similar example by Drake et al. [6].

```
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(); }}
}
```

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. We omit the full syntax of Streamit for space reasons; the interested reader can find it in Appendix B of the extended technical report[22].

4.2.2 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, since 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.

StreamIt non-determinism. StreamIt, at the language level, is deterministic. Furthermore, since 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-, task-, and data-parallelism. This gives an implementation different scheduling choices, which Brooklet models by non-deterministically selecting a firing queue. Despite these non-deterministic choices, an implementation must ensure deterministic end-to-end behavior, which our translation makes explicit with buffering and synchronization.

4.2.3 StreamIt Translation Example

StreamIt program translation turns the StreamIt MPEG decoder P_s from earlier 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.

4.2.4 StreamIt Translation

For space reasons, we give only a high-level overview of the StreamIt translation here (the details are in Appendix B of the extended technical report[22]). 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. Fig. 3 shows the rules. The input to the splitter is a queue q_a, and the output is a list of queues $\overline{q}$; conversely, the input to the joiner is a list of queues $\overline{q}'$, and the output is a single queue q_z. Both the splitter

StreamIt program xlation excerpt:

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

$$\frac{\begin{array}{c} f = \mathit{freshId}() \\ \forall i \in 0 \ldots |\overline{q}'| : v_i = \mathit{freshId}() \\ F_b = [f \mapsto \mathit{wrapRRJoin}(|\overline{q}'|)] \\ \mathit{op} = (q_z, \overline{v}) \leftarrow f(\overline{q}', \overline{v}); \end{array}}{[\![F_s, \texttt{join roundrobin;}, q_z, \overline{q}']\!]^p_s = F_b, \mathit{op}} \text{(T}^p_s\text{-RR-Join)}$$

StreamIt operator wrappers excerpt:

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

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

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

Fig. 3. StreamIt round-robin split and join semantics on Brooklet

and the joiner maintain a cursor to keep track of the next queue in round-robin order. The joiner also stores one variable for each queue, to buffer data that arrives out-of-turn.

4.2.5 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 [26] 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 [27]. By modeling state, our Brooklet translation facilitates reasoning about optimizations.

4.3 Sawzall and MapReduce

Sawzall [20] is a scripting language for MapReduce [5], 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.

4.3.1 Sawzall Program Example: Query Log Analyzer

The example Sawzall program in Fig. 4 is based on a similar example in [20]. 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

Sawzall syntax:

$$
\begin{array}{lll}
P_z & ::= \overline{out}\ \texttt{in}\ \overline{emit} & \textit{Sawzall program} \\
out & ::= t\ \texttt{:}\ \texttt{table}\ f\texttt{;} & \textit{Output aggregator} \\
in & ::= q\ \texttt{:}\ \texttt{input;} & \textit{Input declaration} \\
emit & ::= \texttt{emit}\ t\texttt{[}f(q)\texttt{]} \leftarrow f(q)\texttt{;} & \textit{Emit statement} \\
q & ::= id & \textit{Queue name} \\
f & ::= id & \textit{Function name} \\
t & ::= id & \textit{Table name}
\end{array}
$$

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

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

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*	$O_z \in \mathcal{K}_2 \to \mathcal{X}_2$	*Output table*

Sawzall operator signatures:

$$
\begin{array}{l|l}
f_k : \mathcal{K}_1 \times \mathcal{X}_1 \to \mathcal{K}_2 & f_x : \mathcal{K}_1 \times \mathcal{X}_1 \to \mathcal{X}_2^* \\
f_a : \mathcal{X}_2 \times \mathcal{X}_2 \to \mathcal{X}_2 &
\end{array}
$$

Sawzall operator wrapper signatures:

$$
\begin{array}{l}
\text{Map} \quad : (\mathcal{K}_1 \times \mathcal{X}_1) \times \{1\} \to (\mathcal{T} \times \mathcal{K}_2 \times \mathcal{X}_2)^* \\
\text{Reduce: } (\mathcal{T} \times \mathcal{K}_2 \times \mathcal{X}_2) \times \{1\} \times O_z \to O_z
\end{array}
$$

Sawzall operator wrappers:

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

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

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

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

Fig. 4. Sawzall semantics on Brooklet

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:

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

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. Since the number 1 serves as the value associated with the key, the `sum` aggregators end up counting query log entries by key. Fig. 4 shows the Sawzall grammar.

4.3.2 Sawzall Implementation Issues

Sawzall has stateful and non-deterministic implementations.

Sawzall state. The map operator is stateless, whereas the reduce operator is stateful, using state to incrementalize its aggregation. The implementation in Pike et al.'s paper [20] 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.

4.3.3 Sawzall Translation Example

Given the Sawzall program P_z from earlier, 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 `$v1`...`$v4`.

4.3.4 Sawzall Translation

Fig. 4 specifies the program translation, domains, and operator wrappers. There is only one program translation rule T_z^p. The translation $[\![F_z, P_z, R]\!]_z^p$ takes the Sawzall function environment, the Sawzall program, and the number of reducer partitions as arguments. All the $\overline{emit}$ statements become part of the single map operator. The map operator wrapper uses a hash function to scatter its output over the reducer key space for load balancing. All the $\overline{out}$ declarations become part of each of the reduce operators. Each reducer's variable stores the mapping from each key in that reducer's partition to the latest reduction result for that key. If the key is new, rule W_z-REDUCE-$\emptyset$ fires and registers x_2 as the initial value. At the end of the run, the results in the variables are deterministic, because aggregators are associative and reducers work on disjoint parts of the key space.

4.3.5 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, but Lämmel studies MapReduce and Sawzall by implementing an emulation in Haskell [15]. 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 can serve as a basis for modeling teleport messaging [6].

4.4 Translation Correctness

We formulate correctness theorems for CQL and StreamIt with respect to their formal semantics [2,27]. The proofs are in an extended technical report [22]. We do not formulate a theorem for Sawzall, because it lacks 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 $[\![F_s, P_s]\!]_s^p$.*

5 Optimizations

The previous section used our calculus to understand how a language maps to an execution platform. This section uses our calculus to specify how to use three vital optimizations: data-parallel computation, operator fusion, and operator reordering. Each optimization comes with a correctness theorem; for space reasons, we leave the proofs to an extended technical report [22].

5.1 Data Parallelism

If an operation is commutative across data items, then the order in which the data items are processed is irrelevant. MapReduce uses this observation to exploit the collective computing power of a cluster for analyzing extremely large data sets [5]. The input data set is partitioned, and copies of the map operator process the partitions in parallel. In general, the challenge in exploiting such *data parallelism* is determining if an operator commutes. Sawzall and StreamIt solve this challenge by restricting the programming model. In Brooklet, commutativity analysis can be performed with a simple code inspection. Since a pure function always commutes[2], and all state in Brooklet is explicit in an operator's signature, a sufficient condition for introducing data-parallelism is that an operator does not access variables. The transformation must ensure that the output data is combined in the same order that the input data was partitioned. Brooklet

[2] At least in the mathematical sense; in systems, floating point operations do not always commute.

can use the round-robin splitter and joiner described in the StreamIt translation for this purpose. Thus, the operator `(out)⟵wrapMap-LatLong(q);` can be parallelized with $N = 3$ copies like this:

```
(q1, q2, q3, $sc)          ⟵ Split(q, $sc);
(q4)                       ⟵ wrapMap-LatLong(q1);
(q5)                       ⟵ wrapMap-LatLong(q2);
(q6)                       ⟵ wrapMap-LatLong(q3);
(out, $v4, $v5, $v6, $jc) ⟵ Join(q4, q5, q6, $v4, $v5, $v6, $jc);
```

The following rule describes how to create the new program with N duplicates of the parallelized operator.

$$\frac{\begin{array}{c} op = \texttt{(}q_{out}\texttt{)} \leftarrow f\texttt{(}q_{in}\texttt{)}; \\ \forall i \in 1 \ldots n : q_i = \mathit{freshId}() \qquad \forall i \in 1 \ldots n : q'_i = \mathit{freshId}() \\ F'_b, op_s = [\![\emptyset, \texttt{split roundrobin}, \overline{q}, q_{in}]\!]^p_s \\ \forall i \in 1 \ldots n : op_i = \texttt{(}q'_i\texttt{)} \leftarrow f\texttt{(}q_i\texttt{)}; \\ F''_b, op_j = [\![\emptyset, \texttt{join roundrobin}, q_{out}, \overline{q'}]\!]^p_s \end{array}}{\langle F_b, op \rangle \longrightarrow^N_{split} \langle F_b \cup F'_b \cup F''_b, op_s\ \overline{op}\ op_j \rangle} \quad (\mathrm{O}_b\text{-Split})$$

The precondition is that *op* does not refer to any state variables. The data parallelism optimization illustrates that Brooklet facilitates reasoning over shared state. The rules for round-robin split and join are in Fig. 3.

Making multiplexers explicit and fixing the degree of parallelism are important to faithfully model and reason about real-world systems. Possible implementation strategies for avoiding the limitation of a fixed degree of parallelism include using just-in-time compilation to do splitting online, or putting code on a larger number of machines and then in practice using only a subset as needed.

Theorem 3 (Correctness of O_b-Split). *For all function environments F_b, Brooklet programs P_b, and degrees of parallelism N, if rule* O_b-Split *yields* $\langle F_b, P_b \rangle \longrightarrow^N_{split} \langle F'_b, P'_b \rangle$, *then* $\rightarrow^*_b (F_b, P_b, I_b) = \rightarrow^*_b (F'_b, P'_b, I_b)$ *for all Brooklet inputs* I_b.

5.2 Operator Fusion

In practice, transmitting data between two operators can incur significant overhead. Data needs to be marshalled/unmarshalled, transferred over a network or written to a mutually accessible location, and buffered by the receiver, not to mention the expense of context switching. This overhead can be offset by *fusing* two operators into one. StreamIt applies this optimization to operators in a pipelined topology [26]. Operators may be fused if they meet two conditions. First, they appear in a simple pipeline. Brooklet makes this topology easy to validate because queues are defined and used exactly once. Second, the state used by the operators must not be modifiable anywhere else in the program. Again, because Brooklet requires an explicit declaration of all state, this condition can be verified with a simple code inspection. The following Brooklet program shows two steps in an MPEG decoder:

```
(q1,$v1)   ⟵ ZigZag(qin,$v1);
(qout,$v2) ⟵ IQuantization(q1,$v2);
```

The fused equivalent of the program is:

```
                    (qout,$v1,$v2) ⟵
Fused-ZigZag-IQuant(qin,$v1,$v2);
```

The following rule formalizes this optimization:

$$\frac{\begin{array}{c} op_1 = (q_1, v_1) \leftarrow f_1(q_{in}, v_1); \qquad (\exists op' = (\_, v_1) \leftarrow f'(\_,\_)) \Rightarrow op' = op_1 \\ op_2 = (q_{out}, v_2) \leftarrow f_2(q_1, v_2); \qquad (\exists op' = (\_, v_2) \leftarrow f'(\_,\_)) \Rightarrow op' = op_2 \\ f = freshId() \qquad F'_b = [f \mapsto fusedOperator(F_b, f_1, f_2)]F_b \end{array}}{F_b, op_1\ op_2 \longrightarrow F'_b, (q_{out}, v_1, v_2) \leftarrow f(q_{in}, v_1, v_2);} \text{ (O_b-FUSE)}$$

The preconditions guard against other operators writing variables v_1 or v_2. The following rule defines the new internal function:

$$\frac{(d_{temp}, d'_1) = F_b(f_1)(d_{in}, 1, d_1) \qquad (d_{out}, d'_2) = F_b(f_2)(d_{temp}, 1, d_2)}{fusedOperator(F_b, f_1, f_2)(d_{in}, \_, d_1, d_2) = (d_{out}, d'_1, d'_2)} \text{ (W_b-FUSE)}$$

In our example, this combines F_b(`ZigZag`) and F_b(`IQuantization`) into function F'_b(`Fused-ZigZag-IQuant`). The fusion optimization illustrates that Brooklet facilitates reasoning over topologies.

Theorem 4 (Correctness of O_b-Fuse). *For all function environments F_b and Brooklet programs P_b, if rule* O_b-FUSE *yields* $\langle F_b, P_b\rangle \longrightarrow_{Fuse} \langle F'_b, P'_b\rangle$, *then* $\rightarrow_b^* (F_b, P_b, I_b) = \rightarrow_b^* (F'_b, P'_b, I_b)$ *for all Brooklet inputs* I_b.

5.3 Reordering of Operators

A general rule of thumb for database query optimizations is that it is better to remove more tuples early in order to reduce downstream computations. The most popular example for this is hoisting a select operator, because a select reduces the tuple volume for operators it feeds into [1]. A select is said to *commute* with another operator if their output result is the same regardless of their execution order. The following program computes the commission on sales of IBM stock. The input is `sale(ticker, price)` and the output is `commission(ticker, cost)`. The commission is 2%.

```
output commission;
input sale;
(qt) ⟵ BrokerCommission(sale);
(commission) ⟵ Select-IBM(qt);
```

The functions for the two operators are:

$$F_b(\texttt{BrokerCommission})(d, \_) = \text{let } \langle \texttt{ticker}, \texttt{price}\rangle = d \text{ in } \langle \texttt{ticker}, 0.02 \cdot \texttt{price}\rangle$$
$$F_b(\texttt{Select-IBM})(d, \_) = \text{let } \langle \texttt{ticker}, \texttt{cost}\rangle = d \text{ in if } \texttt{ticker} = \text{`IBM' then } d \text{ else} \bullet$$

We can reorder the two operators for two reasons. First, the `BrokerCommission` operator is stateless, and therefore operates on each data item independently, so

its semantics do not change when it sees a filtered stream of data item. Second, the `Select-IBM` operator only reads the `ticker`, and `BrokerCommission` forwards the `ticker` unmodified. In other words, `Select-IBM` does not rely on any data modified by `BrokerCommission` and vice versa. The optimized program is:

```
output commission;
input sale;
(qt) ⟵ Select-IBM(sale);
(commission) ⟵ BrokerCommission(qt);
```

The following rule encodes the optimization:

$$\frac{\begin{array}{c} op_1 = (q_t) \leftarrow f_1(\overline{q}); \qquad op_2 = (q_{out}) \leftarrow f_2(q_t); \\ F_b(f_1)(d,i) = \text{let } \langle r, w\rangle = d \text{ in } \langle r, f_1(w,i)\rangle \\ F_b(f_2)(d,\_) = \text{let } \langle r,\_\rangle = d \text{ in if } f_2(r) \text{ then } d \text{ else } \bullet \\ \forall i \in 1 \ldots |\overline{q}| : q'_i = \mathit{freshId}() \\ op'_1 = (q_{out}) \leftarrow f_1(\overline{q'}); \qquad \forall i \in 1 \ldots |\overline{q}| : op_i = (q'_i) \leftarrow f_2(q_i); \end{array}}{F_b, op_1\ op_2 \longrightarrow F_b, \overline{op}\ op'_1} \text{ (O_b-HoistSelect)}$$

The first two preconditions restrict op_1 and op_2 to be stateless operators. The third precondition specifies that f_1 forwards a part r of the data item unmodified, and the fourth precondition specifies that f_2 is a select that only reads r, and forwards the entire data item unmodified. We have chosen in Brooklet to abstract away local deterministic computations into opaque functions, because their semantics are well-studied (e.g., [8,10,21]). We leverage this prior work by assuming that a static program analysis can determine the restrictions on the read and write sets of operator functions used for select hoisting.

Theorem 5 (Correctness of O_b-HoistSelect). *For all function environments F_b and Brooklet programs P_b, if $\langle F_b, P_b\rangle \longrightarrow_{HoistSelect} \langle F'_b, P'_b\rangle$ by rule O_b-HoistSelect, then $\rightarrow_b^* (F_b, P_b, I_b) {=\!\!\Rightarrow}_b^* (F'_b, P'_b, I_b)$ for all Brooklet inputs I_b.*

5.4 Optimizations Summary

We have used our calculus to understand how a language can apply three vital optimizations. The concise and straightforward formalization of the optimizations validates the design of Brooklet. There are many other streaming optimizations, including, to name just a few, sharing redundant subqueries in CQL [1]; pre-aggregating data on the workers performing the map phase of MapReduce [5]; or eliminating spurious synchronization in StreamIt [26]. Furthermore, there are stronger variants of the optimizations we sketched; for example, it is sometimes possible to introduce data parallelism even for stateful operators. We believe that the examples in this section are a useful first step towards formalizing optimizations for stream processing languages.

6 Related Work

Our approach to defining a core minimal language that allows us to reason about correctness is inspired by Featherweight Java [13].

There has been extensive prior work in the semantics of stream processing. Stephens [23] 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 process networks [14], such as Unix pipes, assume deterministic execution. Synchronous data flow [16] models, such as StreamIt, assume fixed buffer sizes and static communication patterns. Hoare's communicating sequential process [12] assumes no buffering, and synchronous communication. Gurevich et al. [11] recently studied streaming systems, but focused on their more theoretical aspects.

The database literature often refers to streaming applications as "continuous queries" [4,25]. 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 [10] and a study of object-oriented databases [7].

This paper uses CQL, Sawzall, and StreamIt as representative examples of streaming languages, but there are many more. Spade [9] is a streaming language for composing parallel and distributed flow graphs for System S, IBM's scalable data processing middleware. Pig Latin [18] is one of the languages designed to compose MapReduce or Hadoop jobs. DryadLinq [28] runs imperative code on local machines and uses integrated SQL to generate distributed queries.

7 Conclusion and Outlook

This paper presents Brooklet, a core calculus for stream processing. It represents stream processing applications as a graph of operators. Operators contain pure functions, thread all state through explicit variables, and trigger non-deterministically. Explicit state and non-deterministic execution are central concepts, capturing the reality of distributed implementations. We translate three representative languages, CQL, Sawzall, and StreamIt, to Brooklet, thus demonstrating its generality for language designers. We formalize three vital optimizations, data parallelism, operator fusion, and operator reordering, in Brooklet, thus demonstrating its usefulness for language implementors. Brooklet 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 Brooklet back into source languages, type systems work, exploration of time or space resource constraints, investigations of progress, fairness, and dead-lock, static analyses for establishing optimization preconditions, and specifications of additional optimizations. Brooklet also provides the foundation for a common intermediate language for stream processing. In ongoing work, we are implementing the translations from CQL, Sawzall, and StreamIt to Brooklet, the optimizations from Brooklet to Brooklet, and a translation from Brooklet to C++. The implementation uses System S [9] as a high-performance streaming runtime, which manages all processes across a cluster and their communications. The long-term goal of our work is to establish Brooklet as both a formal and practical foundation for stream processing.

Acknowledgements

The authors would like to thank the anonymous reviewers for their comments and suggestions. We would also 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. This material is based upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129.

References

1. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: Semantic foundations and query execution. VLDB Journal, 121–142 (2006)
2. Arasu, A., Widom, J.: A denotational semantics for continuous queries over streams and relations. In: SIGMOD Record, pp. 6–11 (2004)
3. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. In: TOG, pp. 777–786 (2004)
4. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: SIGMOD, pp. 379–390 (2000)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
6. Drake, M., Hoffmann, H., Rabbah, R., Amarasinghe, S.: MPEG-2 decoding in a stream programming language. In: IPDPS, pp. 86–95 (2006)
7. Fegaras, L.: Optimizing queries with object updates. In: JIIS, pp. 219–242 (1999)
8. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. In: TOPLAS, pp. 319–349 (1987)
9. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.: SPADE: The System S declarative stream processing engine. In: SIGMOD, pp. 1123–1134 (2008)
10. Ghelli, G., Onose, N., Rose, K., Siméon, J.: XML query optimization in the presence of side effects. In: SIGMOD, pp. 339–352 (2008)
11. Gurevich, Y., Leinders, D., den Bussche, J.V.: A theory of stream queries. In: DBLP, pp. 153–168 (2007)
12. Hoare, C.A.R.: Communicating sequential processes. In: CACM, pp. 666–677 (1978)
13. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java - a minimal core calculus for Java and GJ. In: TOPLAS, pp. 132–146 (1999)
14. Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP, pp. 471–475 (1974)
15. Lämmel, R.: Google's MapReduce Programming Model – Revisited. Science of Computer Programming Journal, 208–237 (2007)
16. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. In: Proc. IEEE, pp. 1235–1245 (1987)
17. Nielson, H.R., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons, Inc., Chichester (1992)
18. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A not-so-foreign language for data processing. In: SIGMOD, pp. 1099–1110 (2008)
19. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
20. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. In: Scientific Programming, pp. 277–298 (2005)

21. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis framework for parallelizing compilers. In: PLDI, pp. 54–67 (1996)
22. Soulé, R., Hirzel, M., Grimm, R., Gedik, B., Andrade, H., Kumar, V., Wu, K.-L.: A unified semantics for stream processing languages (extended). Technical Report 2010-924, New York University (2010)
23. Stephens, R.: A survey of stream processing. In: Acta Inf., pp. 491–541 (1997)
24. The StreamBase dialect of StreamSQL, `http://streamsql.org/`
25. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: SIGMOD, pp. 321–330 (1992)
26. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
27. Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffman, H., Brown, M., Amarasinghe, S.: StreamIt: A compiler for streaming applications. In: MIT Laboratory for Computer Science Technical Memo LCS-TM-622 (2001)
28. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI, pp. 1–14 (2008)