A Universal Calculus for
Stream Processing Languages (Extended)

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 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, 23, 25, 27] 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 [23] 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 [25], 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 [25].
- 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, 14–16], 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[3].

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 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.

---

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

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 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 $\{e : condition\}$ 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 $[\![P_b]\!]_z^p$ 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 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.

Brooklet syntax:

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

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

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

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

Fig. 1. Brooklet syntax and semantics.

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:

$$(ibmSales) \leftarrow SaleJoin(ibmBids, \$lastAsk);$$

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`:

$$(\$lastAsk) \leftarrow Window(ibmAsks);$$

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:

$$(result, \$cnt) \leftarrow Count(ibmSales, \$cnt);$$

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, $\texttt{\$cnt}$ is initialized to zero and $\texttt{\$lastAsk}$ is initialized to the tuple $\langle \text{'IBM'}, \infty \rangle$), 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:

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

$$Q = \left[\begin{array}{l} \texttt{bids} \mapsto \bullet, \ \texttt{ibmBids} \mapsto \Big(\langle \text{'IBM'}, 119 \rangle, \langle \text{'IBM'}, 124 \rangle \Big), \\ \texttt{asks} \mapsto \bullet, \ \texttt{ibmAsks} \mapsto \bullet, \\ \texttt{ibmSales} \mapsto \bullet, \ \ \texttt{result} \mapsto \bullet \end{array} \right]$$

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

$op = (\_,\_) \leftarrow f(\overline{q}, \overline{v});$: 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., $\overline{b}' = \Big(\big(\langle \text{'IBM'}, 119, 119 \rangle \big) \Big)$, $\overline{d}' = \bullet$,
$d = \langle \text{'IBM'}, 119 \rangle$, $i = 1$, $V(\overline{v}) = \langle \text{'IBM'}, 119 \rangle$.

$V' = updateV(op, V, \overline{d}')$: Update the variables using the output $\overline{d}'$.

E.g., in this example, $\overline{d}' = \bullet$, so $V' = V$.

$Q' = updateQ(op, Q, q_i, \overline{b}')$: 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.

$$\text{E.g., } Q' = \left[\begin{array}{ll} \texttt{bids} \mapsto \bullet, & \texttt{ibmBids} \mapsto \Big(\langle \text{'IBM'}, 124 \rangle \Big), \\ \texttt{asks} \mapsto \bullet, & \texttt{ibmAsks} \mapsto \bullet, \\ \texttt{ibmSales} \mapsto \Big(\langle \text{'IBM'}, 119, 119 \rangle \Big), & \texttt{result} \mapsto \bullet \end{array} \right]$$

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:

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

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

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 non-determinism 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 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:

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

CQL syntax:

$$P_c ::= P_{cr} \mid P_{cs} \qquad \text{\textit{CQL program}}$$
$$P_{cr} ::= \qquad\qquad\qquad \text{\textit{(Relation query)}}$$
$$RName \qquad\qquad \text{\textit{Relation name}}$$
$$\mid S2R(P_{cs}) \qquad \text{\textit{Stream to relation}}$$
$$\mid R2R(\overline{P_{cr}}) \qquad \text{\textit{Relation to relation}}$$
$$P_{cs} ::= \qquad\qquad\qquad \text{\textit{(Stream query)}}$$
$$SName \qquad\qquad \text{\textit{Stream name}}$$
$$\mid R2S(P_{cr}) \qquad \text{\textit{Relation to stream}}$$
$$RName \mid SName ::= id \qquad \text{\textit{Input name}}$$
$$S2R \mid R2R \mid R2S ::= id \qquad \text{\textit{Operator name}}$$

CQL example: Bargain finder.

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

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

$[\![\, F_c, SName \,]\!]_c^p = \emptyset, \text{output}\, SName; \text{input}\, SName; \bullet$
$$\text{(T}_c^p\text{-SName)}$$

$[\![\, F_c, RName \,]\!]_c^p = \emptyset, \text{output}\, RName; \text{input}\, RName; \bullet$
$$\text{(T}_c^p\text{-RName)}$$

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

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

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

CQL 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*

CQL operator signatures:

$$S2R : \mathcal{S} \times \mathcal{T} \to \Sigma$$
$$R2S : \Sigma \times \Sigma \to \Sigma$$
$$R2R : \Sigma^n \to \Sigma$$

CQL operator wrapper signatures:

$$S2R : (\Sigma \times \mathcal{T}) \times \{1\} \times \mathcal{S} \to (\Sigma \times \mathcal{T}) \times \mathcal{S}$$
$$R2S : (\Sigma \times \mathcal{T}) \times \{1\} \times \Sigma \to (\Sigma \times \mathcal{T}) \times \Sigma$$
$$R2R : (\Sigma \times \mathcal{T}) \times \{1 \ldots n\} \times (2^{\Sigma \times \mathcal{T}})^n$$
$$\to (\Sigma \times \mathcal{T}) \times (2^{\Sigma \times \mathcal{T}})^n$$

CQL operator wrappers:

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

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

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

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

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

Fig. 2. CQL semantics on Brooklet.

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)`:

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

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

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

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

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

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

$$s_o = \left\{ \langle \langle \text{‘IBM’}, 119, 119 \rangle, 1 \rangle, \langle \langle \text{‘XYZ’}, 35, 35 \rangle, 2 \rangle \right\}$$

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 q_o;
input quotes, history;
(q_q, $v_n)        ← wrapNow(quotes, $v_n);
(q_b, $v_q, $v_h) ← wrapBargainJoin(q_q, history, $v_q, $v_h);
(q_o, $v_o)        ← 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 $F_c(\texttt{BargainJoin})$ 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 [26, 25] 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.

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.

StreamIt program xlation excerpt:

$$f = \mathit{freshId}()$$
$$v = \mathit{freshId}()$$
$$F_b = [f \mapsto \mathit{wrapRRSplit}(|\overline{q}|)]$$
$$op = (\overline{q}, v) \leftarrow f(q_a, v);$$
$$\overline{\llbracket F_s, \texttt{split roundrobin};, \overline{q}, q_a \rrbracket_s^p = F_b, op}$$
$$(\text{T}_s^p\text{-RR-S}\textsc{plit})$$

$$f = \mathit{freshId}()$$
$$\forall i \in 0 \ldots |\overline{q}'| : v_i = \mathit{freshId}()$$
$$F_b = [f \mapsto \mathit{wrapRRJoin}(|\overline{q}'|)]$$
$$op = (q_z, \overline{v}) \leftarrow f(\overline{q}', \overline{v});$$
$$\overline{\llbracket F_s, \texttt{join roundrobin};, q_z, \overline{q}' \rrbracket_s^p = F_b, op}$$
$$(\text{T}_s^p\text{-RR-J}\textsc{oin})$$

StreamIt operator wrappers excerpt:

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

$$d_i' = d_{in}, d_i \qquad \forall j \neq i \in 1 \ldots N : d_j' = d_j$$
$$d_c'', d_{out} = d_c' \qquad \forall j \neq c \in 1 \ldots N : d_j'' = d_j'$$
$$\frac{b_{out}, c', \overline{d}''' = \mathit{wrapRRJoin}(N)(\bullet, i, c + 1 \bmod N, \overline{d}'')}{\mathit{wrapRRJoin}(N)(d_{in}, i, c, \overline{d}) = (b_{out}, d_{out}), c', \overline{d}'''}$$
$$(\text{W}_s\text{-RR-J}\textsc{oin}\text{-R}\textsc{eady})$$

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

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

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). 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 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 [25] 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.

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

Sawzall syntax:

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

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

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

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:

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

Sawzall operator wrappers:

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

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

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

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

Fig. 4. Sawzall semantics on Brooklet.

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:

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

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 v_1...v_4.

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, 26]. The proofs are in the Appendix. 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 re-ordering. Each optimization comes with a correctness theorem; for space reasons, we leave the proofs to the Appendix.

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[4], 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 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.

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

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.

---

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

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 [25]. 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,$v1)   ← ZigZag(q_in,$v1);
(q_out,$v2) ← IQuantization(q₁,$v2);
```

The fused equivalent of the program is:

$$(\texttt{q}_{\texttt{out}},\texttt{\$v1},\texttt{\$v2}) \leftarrow \texttt{Fused-ZigZag-IQuant}(\texttt{q}_{\texttt{in}},\texttt{\$v1},\texttt{\$v2});$$

The following rule formalizes this optimization:

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

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

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

In our example, this combines $F_b(\texttt{ZigZag})$ and $F_b(\texttt{IQuantization})$ into function $F'_b(\texttt{Fused-ZigZag-IQuant})$. 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:

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

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:

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

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 [25]. 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 [22] 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, 24]. 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 [27] 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

References

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

20. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, pp. 277–298, 2005.
21. M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis framework for parallelizing compilers. In *PLDI*, pp. 54–67, 1996.
22. R. Stephens. A survey of stream processing. In *Acta Inf.*, pp. 491–541, 1997.
23. The StreamBase dialect of StreamSQL. `http://streamsql.org/`.
24. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pp. 321–330, 1992.
25. W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *CC*, pp. 179–196, 2002.
26. W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications. In *MIT Laboratory for Computer Science Technical Memo LCS-TM-622*, 2001.
27. Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pp. 1–14, 2008.

Appendix

A   CQL Translation Correctness

This section proves Theorem 1.

A.1   Background on CQL Formal Semantics

Before we can prove that the semantics of CQL on Brooklet are equivalent to previously specified semantics of CQL, we recapitulate those semantics from [2].

A.1.1   CQL Function Environment. The CQL function environment maps names for stream-relational operators to functions. These functions are used both to define the CQL denotational semantics [2], and to define our semantics by translation to Brooklet. In [2] the CQL function environment is written $\mathcal{M}$, but we will write it as F_c here for consistency with the other languages in this paper.

As shown in Fig. 2, the signature of an S2R operator, a.k.a. a window, is $\mathcal{S} \times \mathcal{T} \to \Sigma$. Three common windows are:

$$F_c(\texttt{Now})(s, \tau) = \{e : \langle e, \tau \rangle \in s\}$$

(The `Now` window returns tuples from the current time stamp τ.)

$$F_c(\texttt{Range(T)})(s, \tau) = \{e : \langle e, \tau' \rangle \in s \text{ and } \max\{\tau - T, 0\} \leq \tau' \leq \tau\}$$

(The `Range(T)` window returns tuples from time stamps up to `T` in the past up to the current time stamp τ.)

$$F_c(\texttt{Rows(T)})(s, \tau) = \{e : \langle e, \tau' \rangle \in s \text{ and } \tau' \leq \tau \text{ and } N \geq |\{\langle e, \tau'' \rangle : \tau' \leq \tau'' \leq \tau\}|\}$$

(The `Rows(N)` window returns the last `N` tuples before the current time stamp τ.)

As shown in Fig. 2, the signature of an R2S operator is $\Sigma \times \Sigma \to \Sigma$. Three common R2S operators are:

$$F_c(\texttt{IStream})(\sigma_{\text{new}}, \sigma_{\text{old}}) = \{e : e \in \sigma_{\text{new}} \text{ and } e \notin \sigma_{\text{old}}\}$$

(The `IStream` operator monitors insertions into a relation.)

$$F_c(\texttt{DStream})(\sigma_{\text{new}}, \sigma_{\text{old}}) = \{e : e \notin \sigma_{\text{new}} \text{ and } e \in \sigma_{\text{old}}\}$$

(The `DStream` operator monitors deletions from a relation.)

$$F_c(\texttt{RStream})(\sigma_{\text{new}}, \_) = \sigma_{\text{new}}$$

(The `RStream` operator streams the current instantaneous relation.)

As shown in Fig. 2, the signature of an R2R operator is $\Sigma^n \to \Sigma$. Two common R2R operators are:

$$F_c(\texttt{Join(C)})(\sigma_{\text{lhs}}, \sigma_{\text{rhs}}) = \{e_{\text{lhs}} \oplus e_{\text{rhs}} : e_{\text{lhs}} \in \sigma_{\text{lhs}} \text{ and } e_{\text{rhs}} \in \sigma_{\text{rhs}} \text{ and } \texttt{C}(e_{\text{lhs}}, e_{\text{rhs}})\}$$

(The binary `Join(C)` operator joins tuples from two relations when they satisfy the join condition `C`. In the example in Section 4.1.1, the join condition `C` was `quotes.ask <= history.low`. In [2], the binary R2R operators are illustrated with a semi-join, but we chose a theta-join here, because we used it for the algorithmic trading example.)

$$F_c(\texttt{Select(C)})(\sigma) = \{e : e \in \sigma \text{ and } \texttt{C}(e)\}$$

(The unary `Select(C)` operator is a filter that returns only tuples that satisfy the selection condition `C`. In [2], this operator is called `Filter`, but we chose to call it `Select` here to match the terminology in the optimizations section.)

A.1.2 CQL Execution Semantics Function. The CQL execution semantics function is written as $\mathcal{M}$ in [2], but we will denote it as $\to_c^*$ here to avoid confusion with the function environment $\mathcal{M}$ and for consistency with the other languages in this paper. It takes as inputs a CQL function environment F_c, program P_c, and input I_c, and returns a CQL output O_c. The CQL semantics are big-step denotational, meaning that they define a mapping from an entire input to an entire output. Therefore, the CQL input and output domains are defined in a way that is easy to represent globally.

CQL domains for input and output:

$$
\begin{array}{lll}
\varrho & \in RName \to \mathcal{R} & Relations\ store \\
\varsigma & \in SName \to \mathcal{S} & Streams\ store \\
I_c & \in (RName \to \mathcal{R}) \times (SName \to \mathcal{S}) & CQL\ input \\
O_c & \in \mathcal{R} \mid (\mathcal{T} \to \mathcal{S}) & CQL\ output
\end{array}
$$

A CQL input I_c consists of two maps ϱ and ς, mapping relation names to time-varying relations and stream names to CQL streams, respectively. A CQL output is either a time-varying relation or a mapping from time stamps to streams, depending on whether the CQL program is a relation query or a stream query.

There are three kinds of relation queries, for which the semantics function returns a time-varying relation (domain $\mathcal{R}$):

$$\to_c^* (F_c, RName, I_c) = \text{let } \varrho, \varsigma = I_c \text{ in } \varrho(RName)$$

(The semantics of a relation name just retrieve the time-varying relation from the relation store part of the input.)

$$\to_c^* (F_c, S2R(P_{cs}), I_c)(\tau) = F_c(S2R)(\to_c^* (F_c, P_{cs}, I_c)(\tau), \tau)$$

(The semantics of a window recursively invokes $\to_c^*$ to obtain the subquery result, and invokes a window function from the function environment F_c. Recall from Fig. 2 that the non-terminal P_{cs} denotes a CQL query returning a stream.)

$$\to_c^* (F_c, R2R(\overline{P_{cr}}), I_c)(\tau) = \text{let } \forall i \in 1 \ldots |\overline{P_{cr}}| : r_i = \to_c^* (F_c, P_{cr_i}, I_c)$$
$$\text{in } F_c(R2R)(r_1(\tau), \ldots, r_{|\overline{P_{cr}}|}(\tau))$$

(The semantics of an $R2R$ operator uses the R2R function from the function environment F_c on all the instantaneous relations for the same time stamp.)

There are two kinds of stream queries, for which the semantics function returns a mapping from time stamps to stream contents up to and including that time stamp (domain $\mathcal{T} \to \mathcal{S}$).

$$\to_c^* (F_c, SName, I_c)(\tau) = \text{let } \varrho, \varsigma = I_c \text{ in } \{\langle e, \tau' \rangle : \langle e, \tau' \rangle \in \varsigma(SName) \text{ and } \tau' \leq \tau\}$$

(The semantics of a stream name takes a time stamp τ as a parameter, and returns the stream of all tuples with lesser or equal time stamps from the input mapping from stream names to streams).

$$\to_c^* (F_c, R2S(P_{cr}), I_c)(\tau) = \text{let } r = \to_c^* (F_c, P_{cr}, I_c)$$
$$\text{in } \cup \{F_c(R2S)(r(\tau'), r(\tau' - 1)) : 1 < \tau' \leq \tau\}$$

(The semantics of an $R2S$ operator recursively invoke $\to_c^*$ to obtain the subquery result, and then feed the instantaneous relation from the previous and the current time stamp into the operator function from the function environment F_c).

CQL input translation: $[\![\, I_c \,]\!]_c^i = \langle V, Q \rangle$

$$\frac{\varrho, \varsigma = I_c \qquad T = time(\varrho) \cup time(\varsigma)}{V = initVars() \qquad Q = [\![\, \varrho, T \,]\!]_c^i \cup [\![\, \varsigma, T \,]\!]_c^i}{[\![\, I_c \,]\!]_c^i = \langle V, Q \rangle} \qquad (\text{T}_c^i)$$

....

$$time([id \mapsto r]\varrho) = time(\varrho) \cup time(r) \qquad (\text{T}_c^i\text{-}\textsc{Time-}\varrho)$$

$$time(r) = \{\tau : r(\tau) \neq r(\tau - 1)\} \qquad (\text{T}_c^i\text{-}\textsc{Time-}r)$$

$$time([id \mapsto s]\varsigma) = time(\varsigma) \cup time(s) \qquad (\text{T}_c^i\text{-}\textsc{Time-}\varsigma)$$

$$time(s) = \{\tau : \exists e : \langle e, \tau \rangle \in s\} \qquad (\text{T}_c^i\text{-}\textsc{Time-}s)$$

$$time(\emptyset) = \emptyset \qquad (\text{T}_c^i\text{-}\textsc{Time-}\emptyset)$$

....

$$\frac{Q = [\![\, \varrho, T \,]\!]_c^i \qquad b = [\![\, r, T \,]\!]_c^i}{[\![\, [id \mapsto r]\varrho, T \,]\!]_c^i = [id \mapsto b]Q} \qquad (\text{T}_c^i\text{-}\textsc{Data-}\varrho)$$

$$\frac{\tau > \max T \qquad b = [\![\, r, T \,]\!]_c^i}{[\![\, r, \{\tau\} \cup T \,]\!]_c^i = b, \langle r(\tau), \tau \rangle} \qquad (\text{T}_c^i\text{-}\textsc{Data-}r)$$

$$\frac{Q = [\![\, \varsigma, T \,]\!]_c^i \qquad b = [\![\, s, T \,]\!]_c^i}{[\![\, [id \mapsto s]\varsigma, T \,]\!]_c^i = [id \mapsto b]Q} \qquad (\text{T}_c^i\text{-}\textsc{Data-}\varsigma)$$

$$\frac{\tau > \max T \qquad b = [\![\, s, T \,]\!]_c^i}{d = \langle \{e : \langle e, \tau \rangle \in s\}, \tau \rangle}{[\![\, s, \{\tau\} \cup T \,]\!]_c^i = b, d} \qquad (\text{T}_c^i\text{-}\textsc{Data-}s)$$

$$[\![\, \emptyset, \_ \,]\!]_c^i = \emptyset \qquad (\text{T}_c^i\text{-}\textsc{Data-}\emptyset_1)$$

$$[\![\, \_, \emptyset \,]\!]_c^i = \bullet \qquad (\text{T}_c^i\text{-}\textsc{Data-}\emptyset_2)$$

CQL output translation: $[\![\, V, Q \,]\!]_c^o = O_c$

$$\frac{[\![\, V, Q \,]\!]_c^o = [\![\, Q(q_{out}) \,]\!]_{cr}^o}{[\![\, V, Q \,]\!]_c^o = [\![\, Q(q_{out}) \,]\!]_{cs}^o} \qquad (\text{T}_c^o)$$

$$\frac{r = [\![\, b \,]\!]_{cr}^o}{r' = [\tau \mapsto \sigma]r}{[\![\, (b, \langle \sigma, \tau \rangle) \,]\!]_{cr}^o = r'} \qquad (\text{T}_c^o\text{-}\textsc{Relation})$$

$$[\![\, \bullet \,]\!]_{cr}^o = \emptyset \qquad (\text{T}_c^o\text{-}\textsc{Relation-}\bullet)$$

$$\frac{O_c = [\![\, b \,]\!]_{cs}^o}{O_c' = [\tau \mapsto stream((b, \langle \sigma, \tau \rangle))]O_c}{[\![\, (b, \langle \sigma, \tau \rangle) \,]\!]_{cs}^o = s'} \qquad (\text{T}_c^o\text{-}\textsc{Stream})$$

$$[\![\, \bullet \,]\!]_{cs}^o = \emptyset \qquad (\text{T}_c^o\text{-}\textsc{Stream-}\bullet)$$

$$\frac{s = stream(b)}{s' = s \cup \{\langle e, \tau \rangle : e \in \sigma\}}{stream(b, \langle \sigma, \tau \rangle) = s'} \qquad (\text{T}_c^o\text{-}\textsc{Stream-Aux})$$

$$stream(\bullet) = \emptyset \qquad (\text{T}_c^o\text{-}\textsc{Stream-Aux-}\bullet)$$

Fig. 5. CQL input and output translation.

A.1.3 CQL Input and Output Translation. The CQL domains are designed as global mathematical structures representing everything that ever happens in a program execution. This representation suits itself well to modeling with denotational semantics, but it is a mismatch for an actual implementation, where data arrives piecemeal and must be processed incrementally. It comes

therefore as no surprise that the representation is also a mismatch for the small-step operational semantics of Brooklet, since they are designed to facilitate reasoning about implementation.

Due to this representational mismatch, we need a couple of conversion functions that we shall use in our translation correctness proof. The function $[\![\, I_c \,]\!]_c^i$ converts a CQL input to a Brooklet input I_b, and the function $[\![\, O_b \,]\!]_c^o$ converts a Brooklet output back to CQL output O_c. Denoting them with semantic brackets $[\![\, \cdot \,]\!]$ is a slight abuse of notation, because the functions do not incorporate any deep semantics, but rather, are a mechanical (though tedious) conversion from one data format to another. However, since we denoted program translation as $[\![\, \cdot \,]\!]_c^p$, denoting input and output translation as $[\![\, \cdot \,]\!]_c^i$ and $[\![\, \cdot \,]\!]_c^o$ leads to more consistent notation. Note that besides the subscript $_c$ for CQL, we introduce the superscripts ^{p,i,o} for program, input, and output, respectively.

The left column of Figure 5 defines the input translation. It first obtains the set T of all "interesting" time stamps, at which either a relation changes, or a stream sends a tuple. Then, it creates Brooklet queue contents, which are sequences of data items where each data item is a tuple of an instantaneous relation and a time stamp in T. All variables are initialized to the empty set, and all input queues are initialized to queue contents.

The right column of Figure 5 defines the output translation. Depending on whether the CQL program is a relation query or a stream query, the output translation turns the contents of the Brooklet output queue either into a time-varying relation, or into a function from time stamps to the stream of all tuples seen up to that time stamp.

A.2   CQL Main Theorem and Proof
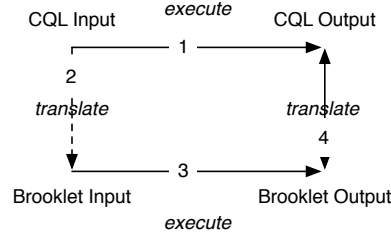
Fig. 6. CQL translation correctness, structural induction base.

Given the background definitions, we can re-state Theorem 1 more clearly:

Theorem 1 (CQL translation correctness). For all CQL function environments F_c, programs P_c, and inputs I_c:

$$\to_c^* (F_c, P_c, I_c) = [\![\, \to_b^* \,([\![\, F_c, P_c \,]\!]_c^p, [\![\, I_c \,]\!]_c^i) \,]\!]_c^o$$

In other words, executing under CQL semantics $(\to_c^*)$ yields the same result as translating the program and the input from CQL to Brooklet $([\![\, \cdot \,]\!]_c^{p,i})$, then executing under Brooklet semantics $(\to_b^*)$, and finally translating the output from Brooklet back to CQL $([\![\, \cdot \,]\!]_c^o)$. Fig. 6 illustrates this graphically.

Proof (Theorem 1). We use an outer structural induction over the query tree, with the two base cases SName (Lemma 2, stream identity) and RName (Lemma 3, relation identity), and the three recursive cases S2R, R2S, and R2R (Lemmas 4, 5, and 6). Each of the five cases does an inner induction over time stamps. Each inner base case is the empty set $\emptyset$ of time stamps. Each inner inductive step assumes that the translation is correct for all time stamps in a set T, and proves that the translation is correct when adding another time stamp $\tau > \max T$.   □

A.3 Detailed Inductive Proof of CQL Correctness

Lemma 1 (CQL Program Shape Correspondence). *Every CQL program P_c forms a logical query plan. For all $P_b = [\![\, P_c\,]\!]_c^p$,*

1. *P_b and P_c are trees with equivalent shapes.*
2. *Each node in P_b is a **Brooklet** operator that corresponds to exactly one CQL operator in P_c.*
3. *Each edge in P_c corresponds to exactly one queue in P_b.*
4. *Each node in P_b has a variable. No variables are shared.*
5. *There is a single root node in both P_b and P_c.*

Proof (Lemma 1). By construction. □

Lemma 2 (SName Translation Correctness). *For all CQL function environments F_c, inputs I_c, and primitive programs P_c consisting of just a single stream name SName, the first part of the lemma states:*

$$\to_c^* (F_c, SName, I_c) = [\![\, \to_b^* ([\![\, F_c, SName\,]\!]_c^i, [\![\, I_c\,]\!]_c^i)\,]\!]_c^o$$

This equality is a special case of Theorem 1. In terms of Fig. 6, it amounts to showing that $\to = \downarrow\,\llcorner\,\uparrow$ in the case where the program P_c consists of a single stream name SName.
The second part of the lemma states:

$$[\![\, \to_c^* (F_c, SName, I_c)\,]\!]_c^i = \to_b^* ([\![\, F_c, SName\,]\!]_c^i, [\![\, I_c\,]\!]_c^i)$$

This equality is used as the assumption for peer lemmas in the outer induction. In terms of Fig. 6, it amounts to showing that $\overrightarrow{\downarrow} = \llcorner$ in the case where the program P_c consists of a single stream name SName.

Proof (Lemma 2). We do an inner induction over the set T of time stamps.

I. Basis: $T = \emptyset$.
   $\overrightarrow{}$: $\to_c^* (Q_s, I_c)$
      $= \emptyset$
      by $\to_c^* (SName)$
   $\downarrow$: $[\![\, F_c, SName\,]\!]_c^i, [\![\, I_c\,]\!]_c^i$
      $= P_b, Q(SName) = \bullet$
      by rule T_c^i-DATA-S.
   $\llcorner$: $\to_b^* ([\![\, F_c, Q_s\,]\!]_c^i, [\![\, I_c\,]\!]_c^i)$
      $= Q(SName) = \bullet$
      because no **Brooklet** semantic rule fires.
   $\overrightarrow{\downarrow}$: $[\![\, \to_c^* (F_c, SName, I_c)\,]\!]_c^i$
      $= Q(SName) = \bullet$
      by rule T_c^i-DATA-S.
   $\llcorner\,\uparrow$: $[\![\, \to_b^* ([\![\, F_c, SName\,]\!]_c^i, [\![\, I_c\,]\!]_c^i)\,]\!]_c^o$
      $= \emptyset$
      by rule T_{cs}^o-$\bullet$

To show: ($\rightarrow$ = $\downarrow\uparrow$)
$\rightarrow_c^* (F_c, SName, I_c) = \emptyset$
$[\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o = \emptyset$
So, $\rightarrow_c^* (SName, I_c) = [\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$.

To show: ($\rightarrow\!\!\downarrow$ = $\downarrow\!\!\rightarrow$)
$[\![\rightarrow_c^* (F_c, SName, I_c)]\!]_c^i = \bullet$
$\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i) = \bullet$
So, $[\![\rightarrow_c^* (F_c, SName, I_c)]\!]_c^i = \rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)$

II. Inductive step: $T' = \{\tau\} \cup T$ where $\tau > \max T$.
  i. Assume for T:
    $\rightarrow$: $\rightarrow_c^* (Q_s, I_c)$
       $= s$
    $\downarrow$: $[\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i$
       $= P_b, Q(SName) = b_s,$
    $\downarrow\!\!\rightarrow$: $\rightarrow_b^* ([\![F_c, Q_s]\!]_c^i, [\![I_c]\!]_c^i)$
       $= Q(SName) = b_s$
    $\rightarrow\!\!\downarrow$: $[\![\rightarrow_c^* (F_c, SName, I_c)]\!]_c^i$
       $= Q(SName) = b_s$
    $\downarrow\uparrow$: $[\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$
       $= s$
  ii. Prove for T':
    $\rightarrow$: $\rightarrow_c^* (Q_s, I_c)$
       $= s \cup s'$
       where s is defined in the assumption as the stream of tuples at time T, and s' is the bag of additional tuples for time $\{\tau\}$, according to $\rightarrow_c^* (SName)$.
    $\downarrow$: $[\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i$
       $= P_b, Q(SName) = b_s, b_{s'}$
       by rule T_c^i-DATA-S.
    $\downarrow\!\!\rightarrow$: $\rightarrow_b^* ([\![F_c, Q_s]\!]_c^i, [\![I_c]\!]_c^i)$
       $= Q(SName) = b_s, b_{s'}$
       because no Brooklet semantic rule fires.
    $\rightarrow\!\!\downarrow$: $[\![\rightarrow_c^* (F_c, SName, I_c)]\!]_c^i$
       $= Q(SName) = b_s, b_{s'}$
       by rule T_c^i-DATA-S.
    $\downarrow\uparrow$: $[\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$
 $= s \cup s'$
       by T_c^o-STREAM

To show: ($\rightarrow$ = $\downarrow\uparrow$)
$\rightarrow_c^* (F_c, SName, I_c) = s \cup s'$
$[\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o = s \cup s'$
So, $\rightarrow_c^* (SName, I_c) = [\![\rightarrow_b^* ([\![F_c, SName]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$.

To show: ($\overset{\rightarrow}{\downarrow} = \overset{\downarrow}{\rightarrow}$)

$[\![\rightarrow^*_c (F_c, SName, I_c)]\!]^i_c = b_s, b_{s'}$

$\rightarrow^*_b ([\![F_c, SName]\!]^i_c, [\![I_c]\!]^i_c) = b_s, b_{s'}$

So, $[\![\rightarrow^*_c (F_c, SName, I_c)]\!]^i_c = \rightarrow^*_b ([\![F_c, SName]\!]^i_c, [\![I_c]\!]^i_c)$ □

Lemma 3 (RName Translation Correctness). *This is the other base case of the outer induction, and is formulated analogously to the SName case in Lemma 2.*

Proof (Lemma 3). We do an inner induction over the set T of time stamps, which is analogous to the proof for Lemma 2. □
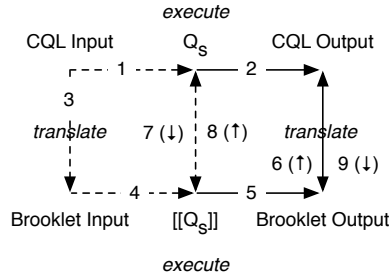
Fig. 7. CQL translation correctness, structural induction step.

Lemma 4 (S2R Translation Correctness). *For all CQL function environments F_c, CQL inputs I_c, CQL stream queries P_{cs}, and CQL S2R operators, assume:*

$$[\![\rightarrow^*_c (F_c, P_{cs}, I_c)]\!]^i_c = \rightarrow^*_b ([\![F_c, P_{cs}]\!]^i_c, [\![I_c]\!]^i_c)$$

This equality is the outer induction assumption, and is proven as part of the peer lemmas for any queries that return streams. In terms of Fig. 7, it amounts to assuming that $\overset{\rightarrow}{\downarrow} = \overset{\downarrow}{\rightarrow}$ for the left part of the diagram.

The first part of the lemma states:

$$\rightarrow^*_c (F_c, S2R(P_{cs}), I_c) = [\![\rightarrow^*_b ([\![F_c, S2R(P_{cs})]\!]^i_c, [\![I_c]\!]^i_c)]\!]^o_c$$

This equality is a special case of Theorem 1. In terms of Fig. 7, it amounts to showing that $\overset{\rightarrow\rightarrow}{} = \overset{\downarrow\rightarrow\uparrow}{}$ in the case where the program P_c has the shape $S2R(P_{cs})$.

The second part of the lemma states:

$$[\![\rightarrow^*_c (F_c, S2R(P_{cs}), I_c)]\!]^i_c = \rightarrow^*_b ([\![F_c, S2R(P_{cs})]\!]^i_c, [\![I_c]\!]^i_c)$$

This equality is used as the assumption for peer lemmas in the outer induction. In terms of Fig. 7, it amounts to showing that $\overset{\rightarrow\rightarrow}{\downarrow} = \overset{\downarrow}{\rightarrow}$ in the case where the program P_c has the shape $S2R(P_{cs})$.

Proof (Lemma 4). We do an inner induction over the set T of time stamps.

I. Basis: $T = \emptyset$.

$\overrightarrow{}$: $\rightarrow_c^* (P_{cs}, I_c)$
$= \emptyset$
by $\rightarrow_c^* (P_{cs})$

$\overrightarrow{}$: $\rightarrow_c^* (S2R(P_{cs}), I_c)$
$= \emptyset$
by $\rightarrow_c^* (S2R(P_{cs}))$

$\overrightarrow{}\downarrow$: $[\![\rightarrow_c^* (S2R(P_{cs}), I_c)]\!]_c^i$
$= \bullet$ by rule $\mathrm{T}_c^i\text{-}\textsc{Data-}\emptyset_2$

$\downarrow$: $[\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i$
$= P_b, V(v) = \emptyset, Q(q_o) = \bullet, Q(q_o') = \bullet$
by rule $\mathrm{T}_c^i\text{-}\textsc{Data-r}$.

$\hookrightarrow$: $\rightarrow_b^* ([\![F_c, P_{cs}]\!]_c^i, [\![I_c]\!]_c^i)$
$= V(v) = \emptyset, Q(q_o) = \bullet, Q(q_o') = \bullet$
because no Brooklet semantic rule fires.

$\hookrightarrow\!\!\!\rightarrow$: $\rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i)$
$= V(v) = \emptyset, Q(q_o) = \bullet, Q(q_o') = \bullet$
because no Brooklet semantic rule fires.

$\hookrightarrow\!\!\!\rightarrow\uparrow$: $[\![\rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$
$= \emptyset$
by rule $\mathrm{T}_{cr}^o\text{-}\bullet$

To show: ($\overrightarrow{} = \hookrightarrow\!\!\!\rightarrow\uparrow$)
$\rightarrow_c^* (S2R(P_{cs}), I_c) = \emptyset$
$[\![\rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o = \emptyset$
So, $\rightarrow_c^* (S2R(P_{cs}), I_c) = [\![\rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i)]\!]_c^o$.

To show: ($\overrightarrow{}\downarrow = \hookrightarrow\!\!\!\rightarrow$)
$[\![\rightarrow_c^* (S2R(P_{cs}), I_c)]\!]_c^i = \bullet$
$\rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i) = \bullet$
So,$[\![\rightarrow_c^* (S2R(P_{cs}), I_c)]\!]_c^i = \rightarrow_b^* ([\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i)$.

II. Inductive step: $T' = \{\tau\} \cup T$ where $\tau > max\ T$.
  i. Assume for T:
    $\overrightarrow{}$: $\rightarrow_c^* (P_{cs}, I_c)$
    $= s$
    $\overrightarrow{}$: $\rightarrow_c^* (S2R(P_{cs}), I_c)$
    $= r$
    $\overrightarrow{}\downarrow$: $[\![\rightarrow_c^* (S2R(P_{cs}), I_c)]\!]_c^i$
    $= b_r$
    $= \langle r(min\ T), min\ T \rangle \ldots \langle r(max\ T), max\ T \rangle$
    $\downarrow$: $[\![F_c, S2R(P_{cs})]\!]_c^i, [\![I_c]\!]_c^i$
    $= P_b, V, Q$
    $\hookrightarrow$: $\rightarrow_b^* ([\![F_c, P_{cs}]\!]_c^i, [\![I_c]\!]_c^i)$
    $V(v) = \emptyset$
    $Q(q_o) = b_s$
    $Q(q_o') = \bullet$

↳→→ : $\to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i)$
$V(v) = s$
$Q(q_o) = \bullet$
$Q(q_o') = b_r$

↳→→↴ : $\llbracket \to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i) \rrbracket_c^o$
$= r$

ii. Prove for T':

→ : $\to_c^* (P_{cs}, I_c)$
$= s \cup s'$

where s is defined in the assumption as the stream of tuples at time T, and s' is the bag of additional tuples for time $\{\tau\}$, according to $\to_c^* (P_{cs})$.

→→ : $\to_c^* (S2R(P_{cs}), I_c)$
$= F_c(S2R)(s \cup s', \tau)$
$= [\tau \mapsto F_c(S2R)(s \cup s', \tau)]r$
by $\to_c^* (S2R(P_{cs}))$ and assumption.

→→↴ : $\llbracket \to_c^* (S2R(P_{cs}), I_c) \rrbracket_c^i$
$= b_r, \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i$
by assumption and rule T_c^i-DATA-R.

↓ : $\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i$
$= P_b$
$V(v) = \emptyset$ by rule T_c^i
$Q(q_o) = \bullet$ by rule T_c^i-DATA-$\emptyset_2$
$Q(q_o') = \bullet$ by rule T_c^i-DATA-$\emptyset_2$

↳→ : $\to_b^* (\llbracket F_c, P_{cs} \rrbracket_c^i, \llbracket I_c \rrbracket_c^i)$
$V(v) = \emptyset$ because, by construction, nothing fires that modifies v.
$Q(q_o) = b_s, b_{s'}$ by assumption and E-FIREQUEUE.
$Q(q_o') = \bullet$

↳→→ : $\to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i)$
$V(v) = s \cup s'$ by rule W_c-S2R.
$Q(q_o) = \bullet$ by rule E-FIREQUEUE.
$Q(q_o') = b_r, \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i$ by assumption, E-FIREQUEUE, W_c-S2R, and T_c^i-DATA-R.

↳→→↴ : $\llbracket \to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i) \rrbracket_c^o$
$= \llbracket b_r, \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i \rrbracket_c^o$
$= [\tau \mapsto F_c(S2R)(s \cup s', \tau)]r$ because
$F_c(S2R)(s, \tau)$ is a curried version of the meaning function $\to_c^* (S2R)$ which returns the tuples for time parameter τ rather than all the time stamps in the time duration $\tau_1 \ldots \tau$.
$\llbracket b_r \rrbracket_c^o = r$ by assumption, and
$\llbracket \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i \rrbracket_c^o = F_c(S2R)(s \cup s', \tau)$ by T_c^o-RELATION and T_c^i-DATA-r

To show: (→→ = ↳→→↴)
$\to_c^* (S2R(P_{cs}), I_c) = [\tau \mapsto F_c(S2R)(s \cup s', \tau)]r$
$\llbracket \to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i) \rrbracket_c^o = [\tau \mapsto F_c(S2R)(s \cup s', \tau)]r$
So, $\to_c^* (S2R(P_{cs}), I_c) = \llbracket \to_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i) \rrbracket_c^o$.

To show: (→→↴ = ↳→→)
$\llbracket \to_c^* (S2R(P_{cs}), I_c) \rrbracket_c^i = b_r, \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i$

$\rightarrow_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i) = b_r, \llbracket F_c(S2R)(s \cup s', \tau) \rrbracket_c^i$
So,$\llbracket \rightarrow_c^* (S2R(P_{cs}), I_c) \rrbracket_c^i = \rightarrow_b^* (\llbracket F_c, S2R(P_{cs}) \rrbracket_c^i, \llbracket I_c \rrbracket_c^i).$ □

Lemma 5 (R2S Translation Correctness). *This is the second recursive case of the outer induction, and is formulated analogously to the S2R case in Lemma 4.*

Proof (Lemma 5). We do an inner induction over the set T of time stamps, which is analogous to the proof for Lemma 4. □

Lemma 6 (R2R Translation Correctness). *This is the third recursive case of the outer induction, and is formulated analogously to the S2R case in Lemma 4.*

Proof (Lemma 6). We do an inner induction over the set T of time stamps, which is analogous to the proof for Lemma 6. □

B StreamIt Mapping Details

Below is our syntax and translation for the SDF subset of StreamIt, which is similar to the "literal algebra in canonical form" in the earlier formalization of StreamIt semantics [26]. This section is organized analogously to Fig. 2 and Fig. 4, except that we moved out the program example into Section 4.2.1.

StreamIt syntax:

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

StreamIt program translation: $\llbracket F_s, P_s \rrbracket_s^p = \langle F_b, P_b \rangle$

$$\frac{q_{out} = freshId() \qquad q_{in} = freshId() \qquad F_b, \overline{op} = \llbracket F_s, P_s, q_{out}, q_{in} \rrbracket_s^p}{\llbracket F_s, P_s \rrbracket_s^p = F_b, \texttt{output} \; q_{out}; \; \texttt{input} \; q_{in}; \; \overline{op}} \qquad (\text{T}_s^p)$$

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

$$\frac{\begin{array}{c} n = |\overline{P_s}| \qquad \forall i \in 1 \ldots n-1 : q_i = freshId() \\ \forall i \in 1 \ldots n : F_{b_i}, \overline{op}_i = \llbracket F_s, P_{s_i}, q_i, q_{i-1} \rrbracket_s^p \end{array}}{\llbracket F_s, \texttt{pipeline}\{\overline{P_s}\}, q_n, q_0 \rrbracket_s^p = \cup \overline{F_b}, \overline{op}_1 \ldots \overline{op}_n} \qquad (\text{T}_s^p\text{-P}{\scriptstyle\text{L}})$$

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

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

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

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

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

StreamIt domains:

$$\begin{array}{ll} z \in \mathcal{Z} & \textit{Data item} \\ \ell \in \mathcal{Z}^* & \textit{List of data items} \\ x \in \mathbb{N} & \textit{Natural number (peek number)} \end{array}$$

. . . .

StreamIt operator signatures:

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

. . . .

StreamIt operator wrapper signatures:

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

StreamIt operator wrappers:

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

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

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

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

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

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

C   StreamIt Translation Correctness

This section proves Theorem 2.

C.1   Background on StreamIt Formal Semantics

Before we can prove that the semantics of StreamIt on Brooklet are equivalent to previously specified semantics of StreamIt, we recapitulate those semantics from [26]. The StreamIt semantics are specified over three algebras: the *literal algebra*, which is a subset of StreamIt; the *intermediate algebra*, which looks more like Lisp; and the *transform algebra*, which is a closed-form function. We will use the function from the transform algebra to define the StreamIt execution semantics function $\rightarrow_s^*$.

C.1.1   StreamIt Function Environment. The previous StreamIt semantics are defined on a simple SDF core of StreamIt, called *literal algebra* in [26]. It is mostly identical to the StreamIt syntax that we showed in Section B, except that it does not model state in filters. That means that the canonical form of a filter *ft* looks like this:

```
filter {
  work {
    (t₁, ..., t_{x_PUSH}) ← f(peek(0), ..., peek(x_PEEK − 1));
    push(t₁); ...; push(t_{x_PUSH});
    pop(); ...; pop(); /* x_POP times */
  }
}
```

Filters differ in the integers x_{PUSH}, x_{PEEK}, and x_{POP}, and in the name of the function f. The StreamIt semantics treat the function f as a black-box. We denote by F_s the StreamIt function store, which maps from function names to opaque functions. If a filter peeks and pushes x_{PEEK} and x_{PUSH} data items, respectively, and the filter uses the function name f, then the function $F_s(f)$ in the function environment has the following signature:

$$F_s(f) : \mathcal{Z}^{x_{\text{PEEK}}} \rightarrow \mathcal{Z}^{x_{\text{PUSH}}}$$

This is a pure function invoked each time the filter receives a data item. Our formalization resembles the formalization in [26] in that it uses black-box functions to abstract away local deterministic computation. Our formalization of the literal algebra differs from that in [26] in that we have a single function that returns multiple values to push, instead of having a separate function for each value to push. We adjust the remainder of this section to account for this difference. We also picked different letters in some cases, such as x for integers and $\mathcal{Z}$ for the domain of StreamIt data items.

C.1.2   StreamIt Intermediate Algebra. The StreamIt intermediate algebra (SIA) has a Lisp-like syntax:

StreamIt intermediate algebra syntax:

$$
\begin{aligned}
P_{SIA} &::= ft_{SIA} \mid pl_{SIA} \mid sj_{SIA} \mid fl_{SIA} & \textit{SIA program}\\
ft_{SIA} &::= (\texttt{filter}\ x\ x\ x\ \overline{f}) & \textit{SIA filter}\\
pl_{SIA} &::= (\texttt{pipeline}\ \overline{P_{SIA}}) & \textit{SIA pipeline}\\
sj_{SIA} &::= (\texttt{splitjoin}\ sp_{SIA}\ \overline{P_{SIA}}\ jn_{SIA}) & \textit{SIA split-join}\\
fl_{SIA} &::= (\texttt{feedbackloop}\ jn_{SIA}\ P_{SIA}\ P_{SIA}\ sp_{SIA}) & \textit{SIA feedback loop}\\
sp_{SIA} &::= \texttt{duplicate} \mid \texttt{roundrobin} & \textit{SIA split type}\\
jn_{SIA} &::= \texttt{roundrobin} & \textit{SIA join type}
\end{aligned}
$$

The formalization in [26] only sketches an incomplete translation from the literal algebra to the intermediate algebra. Here, we present the complete translation, filling in the missing details and expressing everything in a notation consistent with the rest of this paper. We will denote the translation as $[\![\,\cdot\,]\!]^p_{SIA}$. The input is a StreamIt function environment F_s and program P_s, and the output is an SIA function environment F_{SIA} and program P_{SIA}.

$$
\frac{\overline{z}' = f(z_{g_{\mathrm{local}}(0)}, \ldots, z_{g_{\mathrm{local}}(x_{\mathrm{PEEK}}-1)})}{wrap_{SIA}(f, i, x_{\mathrm{PEEK}})(g_{\mathrm{local}})(\overline{z}) = z'_i} \tag{W$_{SIA}$-F\textsc{t}}
$$

$$
\frac{
\begin{array}{c}
\forall i \in 1 \ldots |ps| : f_i = \mathit{freshId}()\\
F_{SIA} = [f_1 \mapsto wrap_{SIA}(F_s(f), 1, |pk|), \ldots, f_{|ps|} \mapsto wrap_{SIA}(F_s(f), |ps|, |pk|)]\\
P_{SIA} = (\texttt{filter}\ |ps|\ |pp|\ |pk|\ \overline{f})
\end{array}
}{
[\![\,F_s, \texttt{filter\{work\{}\overline{t}{\leftarrow}f(\overline{pk})\texttt{;}\overline{ps}\ \overline{pp}\texttt{\}\}}\,]\!]^p_{SIA} = \langle F_{SIA}, P_{SIA} \rangle
} \tag{T$^p_{SIA}$-F\textsc{t}}
$$

The function environment F_{SIA} of the intermediate algebra is populated with one function for each `push` statement in the literal syntax. These functions differ from the functions in the original environment in that they obtain their parameters directly from a local index transform g_{local} and a tape $\overline{z}$. An *index transform* is a function from integers to integers, and a tape is a sequence of data items. Therefore, the signature of a function $F_{SIA}(f)$ is:

$$
F_{SIA}(f) : (\mathbb{N} \to \mathbb{N}) \to (\mathcal{Z}^* \to \mathcal{Z})
$$

The local index transform $g_{\mathrm{local}} \in (\mathbb{N} \to \mathbb{N})$ maps the peek index $0 \le i < x_{\mathrm{PEEK}}$ to a tape index, and the tape $\overline{z} \in \mathcal{Z}^*$ is the sequence of all data items that ever travel on some stream. Hence, the indirect subscript $z_{g_{\mathrm{local}}(i)}$ reads a data item from the tape.

We just saw the $[\![\,\cdot\,]\!]^p_{SIA}$ translation rule for filters. The translation rules for recursive syntax (pipeline, split-join, and feedback loop) are fairly straightforward, since they only make superficial syntactic changes:

$$
\frac{
\begin{array}{c}
\forall i \in 1 \ldots |\overline{P_s}| : \langle F_{SIA_i}, P_{SIA_i} \rangle = [\![\,F_s, P_{s_i}\,]\!]^p_{SIA}\\
F_{SIA} = \cup \overline{F_{SIA}} \qquad P_{SIA} = (\texttt{pipeline}\ \overline{P_{SIA}})
\end{array}
}{
[\![\,F_s, \texttt{pipeline\{}\overline{P_s}\texttt{\}}\,]\!]^p_{SIA} = \langle F_{SIA}, P_{SIA} \rangle
} \tag{T$_{SIA}$-P\textsc{l}}
$$

$$
\frac{
\begin{array}{c}
\forall i \in 1 \ldots |\overline{P_s}| : \langle F_{SIA_i}, P_{SIA_i} \rangle = [\![\,F_s, P_{s_i}\,]\!]^p_{SIA}\\
F_{SIA} = \cup \overline{F_{SIA}} \qquad P_{SIA} = (\texttt{splitjoin}\ [\![\,sp\,]\!]^p_{SIA}\ \overline{P_{SIA}}\ [\![\,jn\,]\!]^p_{SIA})
\end{array}
}{
[\![\,F_s, \texttt{splitjoin\{}sp\ \overline{P_s}\ jn\texttt{\}}\,]\!]^p_{SIA} = \langle F_{SIA}, P_{SIA} \rangle
} \tag{T$_{SIA}$-S\textsc{j}}
$$

$$
\frac{
\begin{array}{c}
\langle F'_{SIA}, P'_{SIA} \rangle = [\![\,F_s, P'_s\,]\!]^p_{SIA} \qquad \langle F''_{SIA}, P'_{SIA} \rangle = [\![\,F_s, P''_s\,]\!]^p_{SIA}\\
F_{SIA} = F'_{SIA} \cup F''_{SIA} \qquad P_{SIA} = (\texttt{feedbackloop}\ [\![\,jn\,]\!]^p_{SIA}\ P'_{SIA}\ P''_{SIA}\ [\![\,sp\,]\!]^p_{SIA})
\end{array}
}{
[\![\,F_s, \texttt{feedbackloop\{}jn\ P'_s\ P''_s\ sp\texttt{\}}\,]\!]^p_{SIA} = \langle F_{SIA}, P_{SIA} \rangle
} \tag{T$_{SIA}$-F\textsc{l}}
$$

$$
\begin{aligned}
[\![\,\texttt{split duplicate}\,]\!]^p_{SIA} &= \texttt{duplicate}\\
[\![\,\texttt{split roundrobin}\,]\!]^p_{SIA} &= \texttt{roundrobin}
\end{aligned} \tag{T$_{SIA}$-S\textsc{p}}
$$

$$
[\![\,\texttt{join roundrobin}\,]\!]^p_{SIA} = \texttt{roundrobin} \tag{T$_{SIA}$-J\textsc{n}}
$$

C.1.3   StreamIt Execution Semantics Function. The previous formal semantics for StreamIt [26] is a denotational semantics with a meaning function $\mathcal{M}$. We denote the meaning function as $\rightarrow_s^*$ instead for consistency with the rest of the paper. It is defined by a translation $[\![\cdot]\!]_{STA}^p$, where STA stands for *StreamIt transform algebra*. Here is the definition:

$$\rightarrow_s^* (F_s, P_s, I_s) = \text{let } \langle F_{SIA}, P_{SIA} \rangle = [\![F_s, P_s]\!]_{SIA}^p \text{ in}$$
$$\text{let } f = [\![F_{SIA}, P_{SIA}]\!]_{STA}^p \text{ in}$$
$$f(g_{\text{id}})(I_s)$$

In other words, first use $[\![\cdot]\!]_{SIA}^p$ to transform from the literal algebra to the intermediate algebra, then use $[\![\cdot]\!]_{STA}^p$ to transform from the intermediate algebra to the transform algebra. The transform algebra result is a higher-order function f, which depends on a global stream transform for taking care of any index shifts incurred by splitters. Since we are at the outermost level, we pass in the identity index transform g_{id}. After applying the index transform, the resulting function f takes a sequence I_s of data items as a parameters, which is the tape that serves as the single input to the entire StreamIt program. In other words, the signature of $f = [\![\cdot]\!]_{STA}^p$ is:

$$f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathcal{Z}^* \rightarrow \mathcal{Z}^*)$$

Next, we look at the $[\![\cdot]\!]_{STA}^p$ rule for filters.
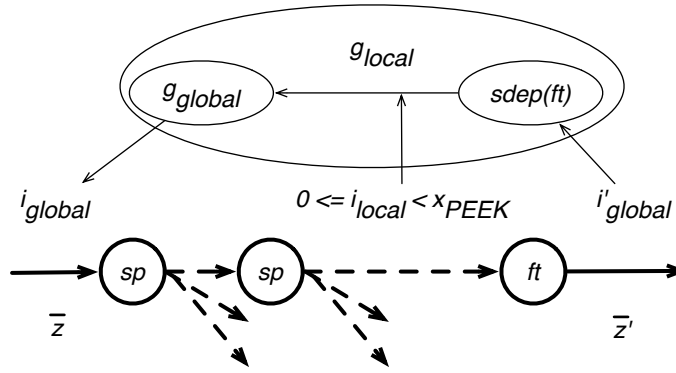
Fig. 8. StreamIt index transforms for a filter.

$$sdep((\texttt{filter } x_{\text{PUSH}} \, x_{\text{POP}} \, x_{\text{PEEK}} \, \_))(i'_{\text{global}}) = \lceil \tfrac{i'_{global}}{x_{\text{PUSH}}} \rceil \cdot x_{\text{POP}} + x_{\text{PEEK}} - x_{\text{POP}} \qquad (\text{W}_{STA}\text{-}\text{F}\text{T})$$

$$\frac{i_{\text{global}} = g_{\text{global}}(sdep(ft_{SIA})(i'_{\text{global}}) - i_{\text{local}})}{localIndexTransform(ft_{SIA}, g_{\text{global}}, i'_{\text{global}})(i_{\text{local}}) = i_{\text{global}}} \qquad (\text{W}_{STA}\text{-}\text{F}\text{T}\text{-}\text{L}\text{O}\text{C}\text{A}\text{L})$$

$$\frac{\begin{array}{c} g_{\text{local}} = localIndexTransform(ft_{SIA}, g_{\text{global}}, i'_{\text{global}}) \\ (\texttt{filter } x_{\text{PUSH}} \, \_ \, \_ \, \overline{f}) = ft_{SIA} \end{array}}{\left([\![F_{SIA}, ft_{SIA}]\!]_{STA}^p (g_{\text{global}})(\overline{z}) \right)_{i'_{\text{global}}} = F_{SIA}(f_{i'_{\text{global}} \bmod x_{\text{PUSH}}})(g_{\text{local}})(\overline{z})} \qquad (\text{T}_{STA}^p\text{-}\text{F}\text{T})$$

The local index $0 \leq i_{\text{local}} < x_{\text{PEEK}}$ is the peek number. The global index i'_{global} is the index on the output from the filter, and the global index i_{global} is the index on an upstream tape $\overline{z}$ that is separated from the filter by zero or more splitters. The local index transform g_{local} transforms the local index i_{local} to i_{global}. It uses a helper function $sdep(ft_{SIA})$ that turns a global index at a filter output into the global index at the filter input that it depends on. The $sdep(ft_{SIA})$ function is

computed based on the data rates x_{PUSH}, x_{POP}, and x_{PEEK} of the filter. The local index transform also subtracts the local index i_{local} to peek into the past, and finally uses the global index transform g_{global} to take care of any transformation imposed by the context of splitters. Figure 8 illustrates these transformations. Dashed lines represent intermediate streams that the index transforms skip past.

Next, we look at the $\llbracket \cdot \rrbracket_{STA}^p$ rule for pipelines.

$$\llbracket F_{SIA}, (\texttt{pipeline } P_{SIA}) \rrbracket_{STA}^p(g_{\text{global}}) = \llbracket F_{SIA}, P_{SIA} \rrbracket_{STA}^p(g_{\text{global}}) \tag{T_{STA}-PL-BASE}$$

$$\frac{\begin{array}{c} \overline{z}' = \llbracket F_{SIA}, (\texttt{pipeline } \overline{P_{SIA}}) \rrbracket_{STA}^p(g_{\text{global}})(\overline{z}) \\ \overline{z}'' = \llbracket F_{SIA}, P_{SIA} \rrbracket_{STA}^p(g_{\text{id}})(\overline{z}') \end{array}}{\llbracket F_{SIA}, (\texttt{pipeline } \overline{P_{SIA}}, P_{SIA}) \rrbracket_{STA}^p(g_{\text{global}})(\overline{z}) = \overline{z}''} \tag{T_{STA}-PL}$$

The base case of a pipeline with just one stage translates that stage in the context of the global index transform. The recursive case transforms the input sequence $\overline{z}$ of data items by first running it through all pipeline stages but one to get $\overline{z}'$, and then running it through the last pipeline stage to get $\overline{z}''$, in the context of an identity index transform g_{id} because there is no additional splitter before the last pipeline stage.

Before we can develop the $\llbracket \cdot \rrbracket_{STA}^p$ rule for split-joins, we need another *sdep* function that describes how to turn the global index at the output of a pipeline into the global index at its input. The function is defined using a simple recursion:

$$sdep((\texttt{pipeline } P_{SIA}))(i'_{\text{global}}) = sdep(P_{SIA})(i'_{\text{global}}) \tag{W_{STA}-PL-BASE}$$

$$sdep((\texttt{pipeline } \overline{P_{SIA}}, P_{SIA}))(i'_{\text{global}}) = sdep(P_{SIA})(sdep((\texttt{pipeline } \overline{P_{SIA}}))(i'_{\text{global}})) \tag{W_{STA}-PL}$$

To develop the *sdep* function for a split-join, we develop separate *sdep* functions for splitters and joiners.

$$sdepSplit(\texttt{duplicate}, \_, \_)(i_{\text{global}}) = i_{\text{global}} \tag{W_{STA}-SP-DUP}$$

$$sdepSplit(\texttt{roundrobin}, n, k)(i_{\text{global}}) = n \cdot i_{\text{global}} + k \tag{W_{STA}-SP-RR}$$

$$\frac{i_{\text{global}} \bmod n < k}{sdepJoin(\texttt{roundrobin}, n, k)(i_{\text{global}}) = \lfloor \frac{i_{\text{global}}}{n} \rfloor + 0} \tag{W_{STA}-JN-RR-0}$$

$$\frac{i_{\text{global}} \bmod n \geq k}{sdepJoin(\texttt{roundrobin}, n, k)(i_{\text{global}}) = \lfloor \frac{i_{\text{global}}}{n} \rfloor + 1} \tag{W_{STA}-JN-RR-1}$$

$$\frac{\begin{array}{ccc} n = |\overline{P_{SIA}}| & k = |\overline{P_{SIA}}| - 1 & i''_{\text{global}} = sdepJoin(jn_{SIA}, n, k)(i'''_{\text{global}}) \\ i'_{\text{global}} = sdep(P_{SIA_k})(i''_{\text{global}}) & i_{\text{global}} = sdepSplit(sp_{SIA}, n, k)(i'_{\text{global}}) \end{array}}{sdep((\texttt{splitjoin } sp_{SIA} \ \overline{P_{SIA}} \ jn_{SIA}))(i'''_{\text{global}}) = i_{\text{global}}} \tag{W_{STA}-SJ}$$

Now, we can formulate the translation $\llbracket \cdot \rrbracket_{STA}^p$ rule for split-joins.

$$\frac{\begin{array}{ccc} (\texttt{splitjoin } sp_{SIA} \ \overline{P_{SIA}} \ jn_{SIA}) = sj_{SIA} & n = |\overline{P_{SIA}}| & k = i''_{\text{global}} \bmod n \\ f_{STA} = \llbracket F_{SIA}, P_{SIA_k} \rrbracket_{STA}^p & i'_{\text{global}} = sdep(jn_{SIA}, n, k)(i''_{\text{global}}) \\ g'_{\text{global}} = g_{\text{global}} \circ sdep(sp_{SIA}, n, k) \circ sdep(P_{SIA_k}) \end{array}}{\left(\llbracket F_{SIA}, sj_{SIA} \rrbracket_{SIA}^p(g_{\text{global}})(\overline{z}) \right)_{i''_{\text{global}}} = \left(f_{STA}(g'_{\text{global}})(\overline{z}) \right)_{i'_{\text{global}}}} \tag{T_{STA}-SJ}$$

Fig. 9 illustrates the index transformations. The data item at index i''_{global} on the output from the split-join is the same as the data item at index i'_{global} on the output from the kth subprogram. The
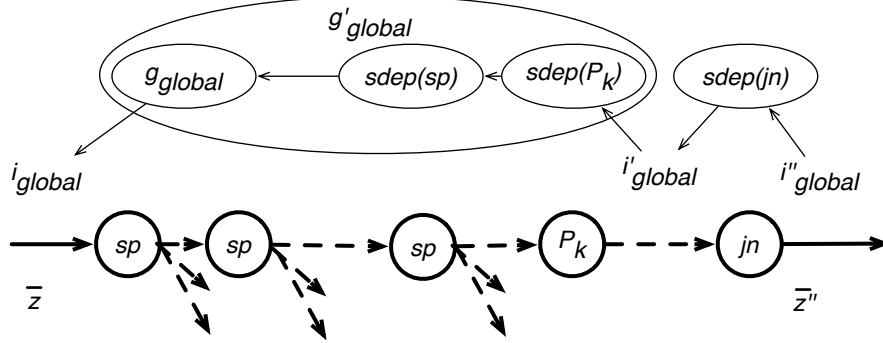
Fig. 9. StreamIt index transforms for a split-join.

kth subprogram operates in the context of an index transform that incorporates the context of the split-join, the index shift of the splitter, and the index shift of the subprogram itself.

We do not model feedback loops here, because they were missing from the transform algebra in [25].

C.1.4   StreamIt Input and Output Translation. In general, the StreamIt domains for input and output are as follows:

<div align="center">

StreamIt domains for input and output:

$I_s \in (id \rightarrow z) \times \mathcal{Z}^*$ *StreamIt input*

$O_s \in \mathcal{Z}^*$ *StreamIt output*

</div>

The input I_s consists of a variable store V in the domain $id \rightarrow z$ and a sequence $\overline{z}$ of data items $\overline{z} \in \mathcal{Z}^*$. The variable store contains initial contents of explicit state variables, and contains the data to kick-start feedback loops. However, since the StreamIt semantics in [25] do not model explicit variables or feedback loops, they only refer to the sequence $\overline{z}$ of input data items. The output O_s, in their case as well as ours, is a simple sequence $\overline{z}'$ of data items.

StreamIt input translation: $[\![I_s]\!]_s^i = \langle V, Q \rangle$

$$\frac{V, \overline{z} = I_s \qquad Q = [q_{in} \mapsto \overline{z}]}{[\![I_s]\!]_s^i = \langle V, Q \rangle} \tag{T_s^i}$$

StreamIt output translation: $[\![V, Q]\!]_s^o = O_s$

$$[\![V, Q]\!]_s^o = Q(q_{out}) \tag{T_s^o}$$

Fig. 10. StreamIt input and output translation.

The input translation $[\![I_s]\!]_s^i$ translates a StreamIt input I_s into a Brooklet input by copying the variable store and by initializing the input queue q_{in} with the sequence of data items. The output translation $[\![O_b]\!]_s^o$ translates a Brooklet output O_b into a StreamIt output by retrieving the sequence of data items from the output queue q_{out}. Note the superscripts ^{i,o} that distinguish input and output translation from program translation $[\![\cdot]\!]^p$. The translations are formalized in Fig. 10.

C.2   StreamIt Main Theorem and Proof

As we have seen in the background information, the previous denotational semantics for StreamIt [25] model neither stateful filters nor feedback loops. In our semantics by translation to Brooklet, on the other hand, we model both features. This was easy to do with our semantics, because it is small-step operational. But because the features are missing from the previous semantics, a proof can only
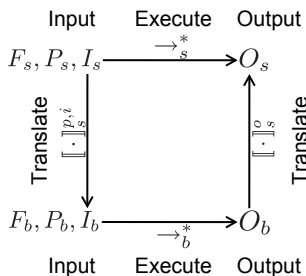
Fig. 11. StreamIt translation correctness.

show equivalence for programs that do not use them. With that in mind, we can re-state Theorem 2 more clearly:

Theorem 2 (StreamIt translation correctness. For all StreamIt function environments F_s, programs P_s, and inputs I_s, where the program P_s uses neither stateful filters nor feedback loops:

$$\to_s^* (F_s, P_s, I_s) = [\![\to_b^* ([\![F_s, P_s]\!]_s^p, [\![I_s]\!]_s^i)]\!]_s^o$$

In other words, executing under StreamIt semantics ($\to_s^*$) yields the same result as translating the program and the input from StreamIt to Brooklet ($[\![\cdot]\!]_c^{p,i}$), then executing under Brooklet semantics ($\to_b^*$), and finally translating the output from Brooklet back to StreamIt ($[\![\cdot]\!]_c^o$). Fig. 11 illustrates this graphically.

Proof (Theorem 2). We use an outer structural induction over the program topology, with the base case Filter (Lemma 7) and the two recursive cases Pipeline and Split-Join (Lemmas 8 and 9; there is no lemma for feedback loops, because they are missing in [25]). □

C.3 Detailed Inductive Proof of StreamIt Correctness

Lemma 7 (Filter Translation Correctness). *Theorem 2 holds for the special case where the StreamIt program P_s is a simple stateless filter.*

Proof (Lemma 7). Without loss of generality, we assume that the filter ft is in canonical form, repeated here for convenience:

```
filter { work {
    (t₁, ..., t_{x_PUSH}) ← f(peek(0), ..., peek(x_PEEK − 1));
    push(t₁); ...; push(t_{x_PUSH});
    pop(); ...; pop(); /* x_POP times */
} }
```

Let $\overline{z} = I_s$ be the StreamIt input and $\overline{z}' = O_s$ be the StreamIt output, and let i be an index in the program output. We will derive the result for z_i along the different edges in the commuting diagram in Fig. 11.

→ : Along this edge, we have $z_i = \left(\to_s^* (F_s, ft, \overline{z}) \right)_i$, which we can rewrite directly by expanding out the definition of $\to_s^*$. This rewriting eventually leads us to the following closed-form expression for z_i:

$$z_i = \left(\rightarrow_s^* (F_s, ft, \overline{z}) \right)_i$$

$$= \left([\![[\![F_s, ft]\!]_{SIA}^p]\!]_{STA}^p (g_{id})(\overline{z}) \right)_i$$

$$= \left([\![F_{SIA}, ft_{SIA}]\!]_{STA}^p (g_{id})(\overline{z}) \right)_i$$

$$= F_{SIA}(f_{i \bmod x_{\mathrm{PUSH}}})(localIndexTransform(ft_{SIA}, g_{id}, i))(\overline{z})$$

$$= F_{SIA}(f_{i \bmod x_{\mathrm{PUSH}}})(\lambda i_{local} : g_{id}(sdep(ft_{SIA})(i) - i_{local})(\overline{z})$$

$$= F_{SIA}(f_{i \bmod x_{\mathrm{PUSH}}})(\lambda i_{local} : sdep(ft_{SIA})(i) - i_{local}))(\overline{z})$$

$$= wrap_{SIA}(F_s(\mathtt{f}), i \bmod x_{PUSH}, x_{\mathrm{PEEK}})(\lambda i_{local} : sdep(ft_{SIA})(i) - i_{local}))(\overline{z})$$

$$= F_s(\mathtt{f})(z_{sdep(ft_{SIA})(i)-0}, \ldots, z_{sdep(ft_{SIA})(i)-x_{\mathrm{PEEK}}+1})(i \bmod x_{\mathrm{PUSH}})$$

$$= F_s(\mathtt{f})\left(z_{(\lceil \frac{i}{x_{PUSH}} \rceil \cdot x_{\mathrm{POP}} + x_{\mathrm{PEEK}} - x_{\mathrm{POP}} - 0)}, \ldots, z_{(\lceil \frac{i}{x_{PUSH}} \rceil \cdot x_{\mathrm{POP}} + x_{\mathrm{PEEK}} - x_{\mathrm{POP}} - x_{\mathrm{PEEK}} + 1)} \right)_{(i \bmod x_{\mathrm{PUSH}})}$$

$$= F_s(\mathtt{f})\left(z_{(\lceil \frac{i}{x_{PUSH}} \rceil \cdot x_{\mathrm{POP}} + x_{\mathrm{PEEK}} - x_{\mathrm{POP}})}, \ldots, z_{(\lceil \frac{i}{x_{PUSH}} \rceil \cdot x_{\mathrm{POP}} - x_{\mathrm{POP}} + 1)} \right)_{(i \bmod x_{\mathrm{PUSH}})}$$

$\downarrow$: This edge performs the program translation and input translation from StreamIt to Brooklet. We end up with the following Brooklet function environment F_b, program P_b, and input $I_b = \langle Q, V \rangle$:

$$F_b = [\mathtt{f} \mapsto wrapFilter(ft)]$$
$$P_b = \mathtt{output\ q_{out};\ input\ q_{in};\ (q_{out},v)\ \leftarrow\ f(q_{in},v);}$$
$$Q = [\mathtt{q_{in}} \mapsto \overline{z}, \mathtt{q_{out}} \mapsto \bullet]$$
$$V = [v \mapsto \bullet]$$

$\downarrow_{\blacktriangleright}$: Along this edge, we are determining the ith data item that gets pushed onto the output queue $\mathtt{q_{out}}$. This data item occurs when rule W_s-FILTER-READY has fired $\lceil \frac{i}{x_{\mathrm{PUSH}}} \rceil$ times. At that point, the contents of the variable $V(\mathtt{v})$ together with the last data item that triggered the firing consists of x_{PEEK} data items. These data items come from consecutive indices of $Q(\mathtt{q_{in}})$ by construction of the wrappers. The index of the last data item that triggered the firing is $\langle \frac{i}{x_{\mathrm{PUSH}}} \rangle + 1 - x_{\mathrm{POP}})$, because at each firing of the W_s-FILTER-READY rule, we pop x_{POP} inputs and push x_{PUSH} outputs. Hence, the data items that the wrapper function $wrapFilter(ft)$ passes to the wrappee function $F_s(\mathtt{f})$ are:

$$z_{(\langle \frac{i}{x_{\mathrm{PUSH}}} \rangle + x_{\mathrm{PEEK}} - x_{\mathrm{POP}})}, \ldots, z_{(\langle \frac{i}{x_{\mathrm{PUSH}}} \rangle + 1 - x_{\mathrm{POP}})}$$

In other words, we invoke the same function on the same parameters as in the $\rightarrow$ case. The result is the sequence of data items to push on the output, and we get the same result z_i as in the $\rightarrow$ case by subscripting with ($i \bmod x_{\mathrm{PUSH}}$).

$\downarrow_{\blacktriangleright}\uparrow$: Along this edge, we have:

$$([\![O_b]\!]_b^o)_i = (Q(\mathtt{q_{out}})_i)$$

which is the same data item computed by the $\downarrow_{\blacktriangleright}$ step.

Both of the calculations ($\rightarrow$ and $\downarrow_{\blacktriangleright}\uparrow$) result in the same data item for the ith position of the output. Since our argument holds for any i, the outputs are fully equal. $\square$

Lemma 8 (Pipeline Translation Correctness). *Assuming the translations for all sub-programs in the pipeline are correct, Theorem 2 holds for the special case where the StreamIt program P_s is a pipeline.*

Proof (Lemma 8). We use the assumption from the outer induction to equate the $\rightarrow$ and $\downarrow_{\blacktriangleright}\uparrow$ cases for each individual stage. We do an inner induction over the number of pipeline stages. In each stage, the output queue contents of the previous stage serve as inputs to the next stage. $\square$

Lemma 9 (Split-Join Translation Correctness). *Assuming the translations for all sub-programs in the split-join are correct, Theorem 2 holds for the special case where the StreamIt program P_s is a split-join.*

Proof (Lemma 9). We chose an arbitrary but fixed subprogram index. We express the contents of the input tape of that subprogram by using an index transform on the StreamIt side, but direct split operator execution on the Brooklet side. Then, we use the assumption from the outer induction to equate the $\rightarrow$ and $\downarrow\!\uparrow$ cases for that subprogram. Finally, we express the contents of the output tape of the entire split-join by using, again, an index transform on the StreamIt side and direct join operator execution on the Brooklet side.   □

D   Data Parallelism Optimization Correctness

This section sketches the proof for Theorem 3.

Let b_{in} and b_{out} be the sequences of all data items that ever appear on queues q_{in} and q_{out}, respectively. Because every Brooklet queue is defined only once and because op is stateless, b_{in} fully determines b_{out}. Since f commutes, b_{out} has the same contents in both P_b and P_b'. Since round-robin split-joins preserve order, b_{out} has the same order in both P_b and P_b'.   □

E   Fusion Optimization Correctness

This section sketches the proof for Theorem 4.

Let b_{in} and b_{out} be the sequences of all data items that ever appear on queues q_{in} and q_{out}, respectively. Because every queue is defined only once and because only op_1 and op_2 write v_1 and v_2, b_{in} fully determines b_{out}. We can show that b_{out} is the same for both P_b and P_b' by induction over b_{in}.   □

F   Selection Hoisting Optimization Correctness

This section sketches the proof for Theorem 5.

The input data on $\bar{q}$ fully determines the output data on q_{out}, because the operators are stateless. We can show by induction over the input data that the output data is the same for both P_b and P_b'. The proof relies on the fact that f_1 only reads data forwarded unmodified by f_2 and vice versa.   □