Flo: A Semantic Foundation for Progressive Stream Processing

SHADAJ LADDAD, UC Berkeley, USA
ALVIN CHEUNG, UC Berkeley, USA
JOSEPH M. HELLERSTEIN, UC Berkeley, USA
MAE MILANO, Princeton University, USA

Streaming systems are present throughout modern applications, processing continuous data in real-time. Existing streaming languages have a variety of semantic models and guarantees that are often incompatible. Yet all these languages are considered "streaming"—what do they have in common? In this paper, we identify two general yet precise semantic properties: streaming progress and eager execution. Together, they ensure that streaming outputs are deterministic and kept fresh with respect to streaming inputs. We formally define these properties in the context of Flo, a parameterized streaming language that abstracts over dataflow operators and the underlying structure of streams. It leverages a lightweight type system to distinguish bounded streams, which allow operators to block on termination, from unbounded ones. Furthermore, Flo provides constructs for dataflow composition and nested graphs with cycles. To demonstrate the generality of our properties, we show how key ideas from representative streaming and incremental computation systems—Flink, LVars, and DBSP—have semantics that can be modeled in Flo and guarantees that map to our properties.

CCS Concepts: • **Software and its engineering → Specialized application languages**; • **Theory of computation → Streaming models**.

Additional Key Words and Phrases: stream processing, dataflow languages, incremental computation

ACM Reference Format:
Shadaj Laddad, Alvin Cheung, Joseph M. Hellerstein, and Mae Milano. 2025. Flo: A Semantic Foundation for Progressive Stream Processing. *Proc. ACM Program. Lang.* 9, POPL, Article 9 (January 2025), 30 pages. https://doi.org/10.1145/3704845

1 Introduction

Stream processing is an increasingly important component of modern applications, from real-time analytics to collaborative tools. These applications must respond with low latency to events as they arise and often process long streams of data. Furthermore, these applications often involve stateful processing, where the output of a computation depends on the history of the inputs.

Many streaming applications are expressed as dataflow programs [4], specified as a directed graph of operators. Each node is an operator that consumes and produces data elements, and the edges represent the flow of data between them. This model is used in systems like Apache Flink [15], Spark [45], StreamIt [42], and many functional-reactive programming languages [38]. Dataflow programs benefit from being written in a declarative manner that abstracts away from low-level details such as how operators are scheduled and where state in the system is accumulated [1, 7, 20,

Authors' Contact Information: Shadaj Laddad, UC Berkeley, USA, shadaj@cs.berkeley.edu; Alvin Cheung, UC Berkeley, USA, akcheung@cs.berkeley.edu; Joseph M. Hellerstein, UC Berkeley, USA, hellerstein@cs.berkeley.edu; Mae Milano, Princeton University, USA, mpmilano@cs.princeton.edu.

ACM 2475-1421/2025/1-ART9
https://doi.org/10.1145/3704845

21]. This makes it easy for compilers to optimize dataflow programs, since they can rearrange and transform operators within the graph without affecting the observable behavior of the program.

Existing streaming languages present a variety of semantics and aim to provide various guarantees. But several streaming languages do not even agree on what constitutes a stream! They can be ordered sequences [15, 42], or sets [6], or even lattices [30] or Z-Sets [13]. These languages also vary in their semantics for state persistence, and offer a range of approaches for concepts like windowed aggregations and batched execution. But they also have much in common: streaming languages tolerate changing inputs and aim to produce outputs as early as possible. Yet these ideas have remained fuzzy and tied to incompatible semantics.

In this paper, we distill these common traits into two key properties: **streaming progress** and **eager execution**. We formally define these properties in the context of **Flo**, a parameterized streaming language that accommodates a range of streaming semantics with compositional dataflow. Flo abstracts away from notions of underlying collection types, such as ordered sequences, and supports semantics that many streaming languages cannot reason about [19], such as retractions.

A key challenge in streaming systems is ensuring that the program makes *progress*. Unlike traditional languages, the definition of progress in streaming languages has long remained fuzzy and tied to very specific semantics. In Flo, we introduce a **general yet precise** formal definition called **streaming progress**, which uses *stream termination* (inspired by work from the databases community [43]) as a common semantic feature to make guarantees about streaming outputs. Streaming progress guarantees that a Flo program produces **as much output** as possible given its input, and that the program **will not block** on a stream that may never terminate.

To enforce streaming progress, we introduce a *lightweight* type system that differentiates between bounded and unbounded streams. Bounded streams are guaranteed to eventually terminate, while unbounded streams may never terminate. Operators can only block on bounded streams, and must always make progress with respect to unbounded streams. These lightweight types can be layered on arbitrary underlying collection types, such as Stream Types [19], sets, or even lattices.

Where streaming progress focuses on ensuring that outputs are produced in a timely fashion relative to inputs, **eager execution** ensures that the outputs are *deterministic*. Many streaming systems make strong assumptions about how operators are executed. For example, Dedalus [6] processes batches with a single global loop, while Naiad [34] processes messages one-by-one. In Flo, we generalize the requirement of deterministic processing into **eager execution**. This property enforces that Flo can **eagerly** execute downstream operators while their inputs are **still being updated**. This property allows for arbitrary execution schedules while arriving at a deterministic result, which gives low-level schedulers significant power to decide when operators should be run.

Flo is a declarative dataflow language that takes inspiration from the iterative processing of actors [23], but uses an event loop that maintains *several* independent input and output queues. Rather than process messages one by one, programs in Flo describe a dataflow that operates over concrete collections of data. In fact, these collections are *finite*, unlike models of streams such as co-inductive lists. To implement streaming applications, these concrete inputs can be extended, and the execution of the Flo program can be safely *resumed* over these new inputs.

Flo also supports **streams of streams**, which capture behavior such as batching. Inspired by ingress/egress nodes in Naiad [34], nested streams can be processed by **nested dataflow graphs**, which iteratively process chunks of data sourced from a larger stream with support for carrying state across iterations. This makes it possible to precisely implement a wide range of applications, such as windowed aggregations, processing data with minibatches, or iterative algorithms.

Flo is a **parameterized** family of languages which bring their own data types and operators. Our proofs of streaming progress and eager execution are compositional, reducing the proof burden to individual operators. This allows Flo to capture the essence of a wide range of streaming systems

under a single model, allowing for composition that spans these approaches. To demonstrate this generality, we show how Flo can be used to model key ideas from a representative variety of streaming languages and incremental computation systems—Flink [15], LVars [30], and DBSP [13]—and show how existing semantic goals from each map to streaming progress and eager execution.

In summary, we make the following contributions:

- We formally define **streaming progress** and **eager execution** in the context of Flo, and specify a type system that reasons about stream termination (Section 3).
- We introduce constructs in Flo for **composing operators** into dataflow graphs and prove that they preserve our key properties (Section 4).
- We describe the semantics of **nested streams and graphs** in Flo and demonstrate how they integrate with streaming progress and eager execution (Section 5).
- We show how the essence and key capabilities of **existing streaming languages** map to Flo and its foundational properties (Section 6).

2 Motivating Example

To understand why we need a model for streaming systems with strong semantic guarantees, let us walk through the challenges a developer may face while writing a simple program that sums up a stream of numbers.

We will accept a sequence of numbers from a streaming source, sum them up, and emit the resulting sum as the single fixed value in the output stream of our program. Streaming sources and sinks are modeled as inputs and outputs to a dataflow graph, so we will not have explicit operators for those. Instead, we can focus on just the core computation of summing up the numbers. A naive attempt may use a `fold` operator, which accumulates a value over a stream of data. In Rust:

```
output = input.fold(0, |acc, x| acc + x)
```

This program is simple, but it has a critical flaw: the `fold` operator is defined over a *fixed* input collection. Operationally this means it will continue processing without producing any output until the stream somehow explicitly terminates. This concern is not addressed in the specification. In a streaming system, this is a common mistake that can lead to programs that hang indefinitely while consuming resources.

We next envision a number of ways a programmer could recognize and address this issue by choosing alternative semantics for this program. We categorize them into strategies that motivate the key properties we aim to establish with Flo: **streaming progress** and **eager execution**.

2.1 Checking Boundedness Constraints

Our program above does work on a subset of input streams: those that are finitely **bounded**, i.e. where the "last" element of the input stream is guaranteed to arrive. Unfortunately, this program is not well-defined on unbounded streams since we may accumulate the aggregation forever. In our semantics, we will model this failure case as an operator that does not satisfy **streaming progress**.

To resolve this, we can imagine classifying input streams via a subtype that would capture the boundedness property. We could then declare that the semantics of the `fold` operator are defined (correct) on bounded input streams, but undefined (incorrect) on unbounded input streams. Boundedness annotations on streams and operators would allow us to statically analyze the program above as incorrect, and suggest a fix: find a way to ensure that `input` is bounded.

But what if the programmer's intent was to handle an unbounded `input` stream? Two natural variations to this specification are possible, as we discuss next.

2.2 Coercing to Bounded Streams

Many streaming applications and languages address the mismatch between unbounded streams and operators that require boundedness by introducing constructs for computing over finite batches or "windows" of the input stream [15]. Perhaps this is what our programmer intended: their use of `fold` was intended to be scoped to a finite substream of `input`.

To capture this idea, we can envision a program variant that uses a `batch` operator to emit a **stream of streams**, where each inner stream is a batch of the original input. There are many possible "windowing" semantics for such a `batch` operator, but let us assume that any such `batch` operator ensures that each inner stream is **bounded** by specification. In that case, it is correct to employ `fold` over the inner streams, even though the outer stream may be **unbounded**. We can specify how each inner stream is handled via a `nest` operator that allows us to define a nested dataflow graph to run for each of these inner streams:

```
output = input.batch().nest(|inner| {
    inner.fold(0, |acc, x| acc + x)
})
```

The output of this program is another stream of streams, where each inner stream is the (single) sum of a batch of the input. This avoids the semantic problem of our previous example: even if `input` is unbounded, each `inner` argument to `nest` is bounded, and hence can be passed into `fold`. Moreover, if `input` *is* bounded, this program can (with appropriate parameterization) produce the same result as our original program, by treating the whole input as a single batch. Hence in some sense we have not drifted too far from what seems to have been the programmer's original intent.

2.3 Embracing Streaming Operators

An alternative "fix" to the initial program would be to replace the `fold` operator with a streaming variant like `scan` that emits the "running" sum:

```
output = input.scan(0, |acc, x| acc + x)
```

On the positive side, this program works on both unbounded and bounded input streams (and it will satisfy our formal definition for streaming progress). However, it seems rather distant from our original program: in particular, there is no way to make it produce the same result as our original program if `input` is bounded.

Instead, we could imagine a streaming operator whose output is a singleton stream of one monotonically growing value. At each step, this aggregator computes an updated sum, but ignores the result if it is smaller than the previous aggregated result. We could then write a program consuming an unbounded input stream:

```
output = input.sum_lattice()
```

Once again, for a bounded input, this program will produce the same result as our original program. It is, however, a departure from traditional streaming systems: for an unbounded input, the output of the `sum_lattice` operator "grows" in the domain of natural numbers rather than in a domain of collections.

To get back to the domain of collections, such a "monotonic singleton" stream can be passed into a monotone function that emits an event upon reaching a threshold:

```
output = input.sum_lattice().event_when_above(100)
```

This is a common pattern in monitoring systems, and is a simplified version of the approach taken by LVars [30]. Why does the threshold need to be monotone? This boils down to our second formal property: **eager execution**. This requires that the overall program yields deterministic results even if we *eagerly* execute operators on partial inputs. If this threshold were not monotone,

there could be non-determinism due to when the threshold is evaluated. But **eager execution** is a more general property than monotonicity; we will show that it is equally meaningful in contexts where there is no natural ordering of values, such as in incremental computations with retractions.

2.4 Discussion

We started with a program that is ill-specified over unbounded streams. We saw various ways to "fix" this problem, inspired by salient design points of different streaming languages. What is key is that although these techniques were motivated by ideas from different languages, they all serve to satisfy two general properties of programs written in Flo: **streaming progress** and **eager execution**. In the following sections, we will walk through the formal semantics of Flo and show how we can precisely define these properties while retaining the flexibility to implement a wide range of streaming semantics found in the literature and used in practice.

3 Collections, Streams, Operators, and Core Properties

The Flo model is based on specifications of dataflow pipelines, where **collections** of data elements are transformed by **operators** such as `map`, `filter`, or `join`. This is inspired by existing systems such as Flink [15], but with a critical difference that Flo is *parameterized* over collection types and operators. This enables us to reason about a wide range of streaming paradigms and capture the essence of languages like LVars [30], Bloom [5], and Temporel [39] under a single model.

In this section, we define a family of collection languages L^C, operator languages L^O, and specify the formal properties that these languages must satisfy. In Section 4, we will define a new family of languages L^G which include mechanisms to compose operators into a dataflow graph. Finally, in Section 5, we will extend L^G with built-in operators for executing nested graphs. Our goal is to prove eager execution and streaming progress for all these languages.

3.1 The Flo Event Loop

Before we can dive into the semantics of these languages, we need to first discuss how Flo programs are executed. Flo deviates from classic streaming models in that it uses an actor-inspired event loop where messages are received, processed, and outputs are emitted. This means that the Flo program itself is always executing over concrete, finite collections of data rather than abstract streams. We describe a lightweight pseudocode for the event loop of a Flo program in Figure 1.

$O \leftarrow$ tuple of empty collections for each output
$G \leftarrow$ initial Flo program
loop
  $\Delta \leftarrow$ tuple of new data batches for each input
  $I \leftarrow$ inputs of G
  $G \leftarrow G$ with inputs set to $I +\!\!+ \Delta$
  $G, O \leftarrow G$ after running an arbitrary number of small-steps with initial output O
  $O \leftarrow$ remaining data after sending arbitrary part of O

Fig. 1. The event loop used to execute Flo programs.

Whenever a batch of new data is received, we use a **concatenation** operator $+\!\!+$ to add this to the existing inputs. In classical streaming systems, such as those proposed in Flink [15] and Stream Types [19], this corresponds to appending new elements to the end of the existing data. But in Flo, our formalization makes it possible for this concatenation operator to take many forms, including those that do not monotonically grow the collection.

The other key aspect to note is that we run an *arbitrary* number of small-steps of the program G in each iteration, rather than running it until there is nothing to be done. We also allow the event loop to arbitrarily choose which data is sent at the end of each iteration; the outputs need not be consumed according to concatenation order. Later in this section, we will introduce key properties that ensure that this loop will always make progress and yield deterministic results.

3.2 Collection Values, Expressions, and Types

Flo programs manipulate *collections*, which are concrete, finite values used to capture inputs, outputs, and (in Section 4) intermediate states of the program. Collection values can be updated as new data arrives or as an operator consumes data, but the way a collection value changes over time *does not need to follow a partial order*, making it possible for our semantics to capture applications such as incremental computation over relations.

We define a collection language $L^C = (C, +\!\!+, E^C, T^C, [\![ ]\!]^C, \lfloor \rfloor^C, \text{type}^C, \mathit{fix})$ as a tuple of:

- C: the set of collection values, which are mathematical objects
- $+\!\!+ : C \times C \to C$: a "concatenation" function on collections
- E^C: the set of collection expressions, which are syntactic objects
- $T^C \subseteq \mathcal{P}(C)$: the set of collection types, which are sets of collection values
- $[\![ ]\!]^C : E^C \to C$: a total denotational semantics that maps collection expressions to values
- $\lfloor \rfloor^C : C \rightharpoonup E^C$: a partial lowering function that maps collection values to expressions
- $\mathit{type}^C : E^C \to T^C$: a total typing function that maps collection expressions to types
- $\mathit{fix} : C \to C$, a transformation from a value into an equivalent[1] one that is *fixed*

We additionally define: $\mathit{fixed}(c) \triangleq \forall c' \in C.\ c +\!\!+ c' = c$ and $\emptyset \in C$ is identity on the RHS of $+\!\!+$.

We constrain L^C via the following well-formedness conditions:

$$\forall e \in E^C.\ [\![e]\!]^C \in \mathit{type}^C(e) \wedge \lfloor [\![e]\!]^C \rfloor^C = e$$

$$\forall c \in C.\ \mathit{fixed}(\mathit{fix}(c)) \wedge c +\!\!+ \emptyset = c$$

$$\forall \tau \in T^C, c \in \tau, c', c'' \in C.\ c +\!\!+ c' = c'' \implies c'' \in \tau$$

The language of collections involves both mathematical and syntactic representations. Our definition of collections is centered around collection **values**, which are the underlying mathematical objects being manipulated. At the syntax level, we represent these with collection **expressions**, which can be lifted to values via a denotational semantics, and then lowered back down to syntax using the $\lfloor \rfloor^C$ function. We also define a typing function type^C that maps collection expressions to types, which are simply sets of collection values.

A key difference between the Flo model and other streaming semantics [19] is that the concatenation function **does not** need to follow a partial order over collection types, or satisfy algebraic properties like commutativity or associativity. What *does* interest us is the question of when the concatenation function reaches a fixpoint. The *fixed* predicate identifies a collection value such that no more data can be added to it, which we will leverage to define streaming progress.

Collections can take on a variety of forms. A common collection in streaming systems is the *ordered sequence*, which captures an ordered list of elements. But collections could also be multisets—as in streaming extensions to SQL [11]—or sets, as in Dedalus [6]—where order often does not affect semantics. A "collection" can even be a single value where "concatenating" to the collection updates the value—as in our `lattice_sum` result in Section 2. We will lay out detailed examples of concrete collection types in Section 6.

[1]The definition of equivalence is up to the collection (for example, concatenating a stream terminator or setting a maximum size), and determines the guarantees provided by streaming progress (Definition 3.3)

3.3 Stream Types and Boundedness

Collections describe the values that are being processed by operators, but our discussion so far has been more reminiscent of batch processing than streaming. Our unique interest in streaming is the evolution of collections over time. In our motivation, we identified two key aspects of a streaming program's behavior with respect to time: **eager execution** makes it possible to correctly process newly-arrived data on an input to get an updated output, and **streaming progress** ensures that the program will not unexpectedly block on a collection becoming fixed.

To formally define streaming progress later in this section, we need to add a layer on top of collection types, which we call **stream types**. In our model, the key property we care about is whether a collection value will eventually become **fixed** (using the definition from Section 3.2), or if it may never become that. To capture this, we use a **boundedness flag** inspired by work in databases [43], which is either **B**ounded or **U**nbounded. We define a stream type as a pair of a collection type and a flag on the left of Figure 2. We will see stream types in action in Section 3.6.

$$\langle \textit{stream-type} \rangle ::= (\langle T \rangle, \mathbf{B} \mid \mathbf{U})$$

$$\textsc{reflexive-subtype}\quad S \leq S \qquad\qquad \textsc{bound-subtype}\quad (C, \mathbf{B}) \leq (C, \mathbf{U})$$

Fig. 2. The grammar for stream types, where $T \in T^C$, and the subtyping relationship for stream types.

Note that collection expressions are not typed directly to a stream type, instead stream types are used as markers on inputs and outputs of a Flo program. We also have a simple subtyping relationship, where a stream type that is declared as bounded can be used in an unbounded context, because an unbounded stream has no restrictions on how the collection value behaves over time. We list the typing rule for this relationship on the right of Figure 2, where $\leq$ is a subtyping relationship we will use in the rules for composing operators.

3.4 Operators

Flo programs transform input collections into output collections. This transformation is carried out by **operators** that consume data from several input collections to update output collections. In this section, we lay out the family of operator languages L^O, which captures Flo programs with a single operator. Because programs written in this language fit the general structure of the Flo event loop, we will use this language to lay out all the key properties we aim to prove about Flo. In Section 4, we will extend this language to L^G to capture the composition of operators into a dataflow graph.

We will use the notation $[C]$ to represent tuples whose elements are each in C (and similarly for $[E^C]$), which denotes having multiple inputs or outputs. We will also denote T^S to be the set of all stream types and $[T^S]$ to be a tuple of many stream types. Tuples of stream types follow an element-wise subtyping relationship.

We define an operator language $L^O = (L^C, E^O, \rightarrow^\delta, ORD^O, \vdash^O)$ as a tuple of:

- $L^C = (C, \nexists, E^C, T^C, [\![\,]\!]^C, \lfloor\rfloor^C, \text{type}^C, \textit{fix})$: a well-formed collection language
- E^O: a language of operator expressions, which are syntactic objects
- $(I, e^o) \rightarrow^\delta (I, e^o, O)$, a small-step operational semantics where $I, O \in [C]$ and $e^o \in E^O$
- $(I, e^o) \prec^O (I, e^o) \in ORD^O$, a set of partial orders on collections where $I \in [C]$ and $e^o \in E^O$ (for some operators, we will omit the operator expression in the partial order)
- $\vdash^O$: $e^O : (\tau^S \hookrightarrow \tau^S, \prec^O)$ a typing relation between elements $e^O \in E^O$, stream types $\tau^S \in [T^S]$, and partial orders $\prec^O \in ORD^O$

We augment this with the following definitions: Given $L^O = (L^C, E^O, \rightarrow^O, ORD^O, \vdash^O)$, we define:

- The set of operator types: $T^O = \{\tau_i \hookrightarrow \tau_o, \prec \mid \tau_i, \tau_o \in [T^S] \wedge \prec \in ORD^O\}$

- The small-step relation $\rightarrow^O = \{((I, e, O), (I', e', O +\!\!+ O')) \mid (I, e) \rightarrow^\delta (I', e', O')\}$
- The typing relation on small-step configurations:

$$\frac{\vdash^O e : ((\tau_i, B_i) \ldots \hookrightarrow (\tau_o \ldots, B_o), \prec^O) \qquad I \in (\tau_i \times \ldots) \qquad O \in (\tau_o \times \ldots)}{\vdash^{\rightarrow} : (I, e, O) : (\tau_i \ldots \hookrightarrow \tau_o \ldots, \prec^O)}$$

We further constrain L^O via the following well-formedness condition:

$$\forall \prec \in ORD^O.\ \prec \text{ is finite and downwards-closed}$$

We also require, $\forall e, e' \in E^O, \tau \in T^O, I, I', O, O' \in [C].\ \vdash^{\rightarrow} (I, e, O) : \tau \wedge (I, e, O) \rightarrow^O (I', e', O')$ (For all well-typed expressions which step):

- $\rightarrow^O$ must be confluent
- $\vdash^{\rightarrow} (I', e', O') : \tau$ (type preservation)
- $\tau = (\ldots, \prec) \implies (e', I') \prec (e, I)$ (steps reduce the operator or its inputs)

Let us break down the intuition behind these properties. Every operator has a type with several input stream types and output stream types. The semantics of each operator are defined by the small-step relation $\rightarrow^\delta$, where the input and operator expression (which may carry state) are used to produce an updated input, operator expression, and an output collection. The small-step relation $\rightarrow^O$ transforms this relation into a classic operational semantics form, where the output generated by $\rightarrow^\delta$ is concatenated to the existing output (this concatenated form will be key to Definition 3.1).

A key property of operators is the confluence of $\rightarrow^O$. In Flo, we **do not** require there to be a unique small step that can be taken for a given input and operator expression. For example, when processing a set of values, an operator may choose to process them in any order. But confluence guarantees that there exists some later state (I', e', O') which all traces of small steps starting from (I, e, O) will eventually reach. For operators that do have this non-determinism, proofs of this property typically involve a commutativity argument over the order of processing inputs.

Each operator also has a partial order over the operator expression and its inputs $\prec$, which is provided by the typing relation $\vdash^O$ and must be preserved across small-steps. We can use this to prove our first property on operators in L^O, that they always reach a stuck state in finite steps:

Lemma 3.1 (Operator Stuck State). *Given an operator op, for all input states I and output states O, there is a finite number of small steps that can be taken before no more small steps can be applied.*

Proof. We leverage the partial order for this operator $\prec$. Since there are a finite number of operator expressions and collection values smaller than the initial state, and each step reduces the expression or its input, and the order is preserved across steps, there must be a finite number of total steps that can be taken before either no step applies or there is no smaller operator or input in the partial order. □

Note that our definition for stuck state does not require the expression to be reduced to some terminating form, such as the inputs all being empty. We only require that no more steps can be taken, which allows us to further loosen the requirements for collections; there is no need to define a unique bottom value, for example. Combined with the confluence of small-steps, this implies that every operator will eventually reach a unique stuck state.

3.5 Eager Execution

Flo hinges on two key properties that enable safe and progressive execution over streaming inputs: **eager execution** and **streaming progress**. The first guarantees that if new data arrives *after* partial inputs have already been processed, then we can safely *resume* the execution of the Flo program while arriving at a deterministic result. The second guarantees the program will never

block on the fixedness of an input that may never become fixed. In Section 4, we will prove that both of these properties are true of **well-typed** graphs and Flo as a whole.

Eager execution avoids the situation where all input to an operator must be computed before the operator can begin execution. Instead, we require all operators to prove that they can begin processing partial inputs and receive additional data later via concatenation, while still producing the same result as if all the data was present from the start. This enables flexibility for scheduling and ensures that the outputs of a Flo program are deterministic even if an arbitrary number of small steps are run during each iteration of the event loop.

Definition 3.1 (Eager Execution)**.** Consider an operator $op \in E^O$. For all inputs $I \in [C]$, outputs $O \in [C]$, concatenated collection $\Delta \in [C]$, updated operator $op' \in E^O$, input collection, $I' \in [C]$, output collection $O' \in [C]$ such that

$$(I, op, O) \rightarrow^O (I', op', O') \text{ and } (I +\!\!+ \Delta, op, O) \rightarrow^O (I'', op'', O'')$$

there exists a stuck state (I''', op''', O''') such that

$$(I' +\!\!+ \Delta, op', O') \rightarrow^{O*} (I''', op''', O''')$$

and

$$(I'', op'', O'') \rightarrow^{O*} (I''', op''', O''')$$

Note that a simple inductive extension of this property tells us that we can introduce a single additional chunk of data of any size interleaved with executing small steps for the operator, and still end up in the same stuck state as if the data was present from the start. A further inductive argument says that if we have several chunks to concatenate, they can be introduced at any time interleaved with steps of the operator while still arriving at the same stuck state.

3.6 Streaming Progress

Streaming progress is a more challenging property to define. Unlike classic correctness properties such as determinism, streaming progress is focused on ensuring that outputs are kept *fresh* with respect to certain inputs. Let us first formally define *freshness* as **output maximality**.

Definition 3.2 (Output Maximality)**.** We are given a well-typed (according to $\vdash^{\rightarrow}$) small-step configuration $((i_0 \ldots i_n), op, O)$ and well-typed final outputs $o'_0 \ldots o'_m$ such that:
$((i_0, \ldots, i_n), op, O) \rightarrow^{O*} (I', op', (o'_0, \ldots, o'_m))$ and $(I', op', (o'_0, \ldots, o'_m))$ is stuck.
Then the given output $o'_0 \ldots o'_m$ is **maximal** if

$$((fix(i_0), \ldots, fix(i_n)), op, O) \rightarrow^{O*} ((i''_0, \ldots, i''_n), op'', (fix(o'_0), \ldots, fix(o'_m)))$$

and $((i''_0, \ldots, i''_n), op'', (fix(o'_0), \ldots, fix(o'_m)))$ is stuck.

Consider our motivating example. Some operators (`scan`) can satisfy Output Maximality for all inputs because at any point in the execution, we can reach a state where all outputs are released, and no more outputs would be released if the input became fixed. But other operators (`fold`) cannot satisfy Output Maximality for all inputs, because we never reach a state with any outputs released unless the input is fixed, at which point the output is released (and hence changes).

This is where the stream types we introduced earlier come in, which will allow us to define a property for streaming progress that works for all operators. Each operator annotates its inputs and outputs with boundedness flags. Intuitively, if an **input** is **unbounded**, we want to prevent the problem we have illustrated with `fold`: we do not want the operator to block until the input becomes fixed. By contrast, if an input is **bounded**, it may make sense for an operator (e.g., `fold`) to withhold some outputs until the input becomes fixed.

Output Maximality and stream types together enable us to ensure that an operator always keeps its outputs as *fresh* as possible: bounded inputs are guaranteed to produce outputs (after becoming fixed), as are unbounded inputs (since they do not block on fixedness).

Finally, to enable composition across multiple operators, we want to derive restrictions on the outputs from input properties. Once the **bounded inputs** are fixed, the **bounded outputs** must become fixed in a finite number of steps to avoid blocking downstream operators. With that intuition in place, we formally define streaming progress in terms of Output Maximality:

Definition 3.3 (Streaming Progress)**.** Consider a well-typed operator op with type $\vdash^O op : ((I_0, B_{I,0}) \dots (I_n, B_{I,n})) \hookrightarrow ((O_0, B_{O,0}) \dots (O_m, B_{O,m}))$. Consider all well-typed inputs $i_0 \dots i_n \in C$ such that $B_{I,j} = \mathbf{B} \implies \mathit{fixed}(i_j)$ (the bounded **inputs** are fixed).

Let us also consider all well-typed initial outputs O and final outputs $o'_0 \dots o'_m$, such that:

$$((i_0, \dots, i_n), op, O) \rightarrow^{O*} (I', op', (o'_0, \dots, o'_m))$$

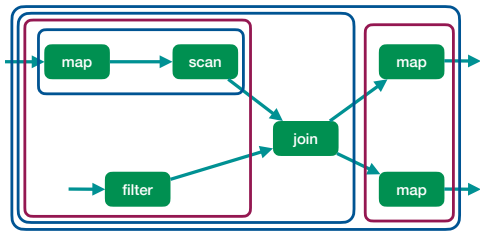
and $(I', op', (o'_0, \dots, o'_m))$ is stuck. Then the operator op satisfies streaming progress if:

- $o'_0 \dots o'_m$ are **maximal** for the operator op with inputs $i_0 \dots i_n$ and initial outputs O
- $\forall j.\ B_{O,j} = \mathbf{B} \implies \mathit{fixed}(o'_j)$ (the bounded **outputs** are fixed)

Any operator in an implementation of Flo must satisfy these properties. We will show in the next section that these properties are automatically preserved when composing operators into graphs, which alleviates any further proof burden for the implementation.

4 Composition with Graphs

Programs in Flo are formed by composing operators into a directed-acyclic graph, where each node is an operator and each edge captures an intermediate collection of data elements. In Flo, we express these directed acyclic graphs as expressions of L^G through recursive constructs for sequential and parallel composition, such as in Figure 3.




```
(
  (
    (({_} map); ({_} scan)) |
     ({_} filter));
   ({_} join));
(({_} map) | ({_} map))
```

Fig. 3. A dataflow graph and its decomposition into an expression in our language (with parentheses for clarity). Magenta boxes represent parallel composition and blue boxes represent sequential composition.

Unlike before, the graph language L^G is not parameterized on any new definitions, and can be directly layered on any instance of an operator language $L^O = (L^C, E^O, \rightarrow^\delta, ORD^O, \vdash^O)$. We layer on this language a few additional constructs:

- E^G: the language of graph expressions, which are syntactic objects (Figure 4)
- $\vdash: e^G : (\tau^S \hookrightarrow \tau^S, \prec)$ a typing relation between elements $e^G \in E^G$, stream types $\tau^S \in [T^S]$, and partial orders $\prec \in \bigcup_{n \in \mathbb{N}} (ORD^O)^n$ (Figure 5)
- $e^g \rightarrow^\Delta (e^g, O)$, a small-step operational semantics where $O \in [C]$ and $e^g \in E^G$ (Figure 6)

We will also augment this with the small-step relation: $\rightarrow = \{((g, O), (g', O +\!\!+ O')) | g \rightarrow^\Delta (g', O')\}$

Sequential composition passes the outputs of one subgraph into the inputs of the other, and is the primary way that operators can be chained together in a Flo program. Parallel composition makes

it possible to capture portions of the graph where several operators can be run independently on separate sets of inputs to produce separate outputs. We lay out the grammar for graphs in Figure 4.

$$e ::= e|e \;\mid\; e;e \;\mid\; \{S\}[O]$$

Fig. 4. The grammar for graphs of a Flo program, where $S \in [E^C]$ and $O \in E^O$.

Note that we include a state term S, which collects inputs to an operator. This term will be essential when formalizing our small-step semantics, which needs to reason about buffered inputs at an *arbitrary* position in a graph. Our type system models graphs in terms of their input and output stream types, and a partial order over inputs like for operators. We list the typing rules for graphs in Figure 5 and small-step operational semantics in Figure 6. In our semantics, we will use $\cdot$ to denote tuple concatenation, when dealing with types or values.

SEQUENCE

$$\frac{\vdash e_1 : (I_1 \hookrightarrow O_1, \prec_1) \qquad \vdash e_2 : (I_2 \hookrightarrow O_2, \prec_2) \qquad O_1 \leq I_2}{\vdash e_1; e_2 : (I_1 \hookrightarrow O_2, \prec_1)}$$

PAR

$$\frac{\vdash e_1 : (I_1 \hookrightarrow O_1, \prec_1) \qquad \vdash e_2 : (I_2 \hookrightarrow O_2, \prec_2)}{\vdash e_1 \mid e_2 : (I_1 \cdot I_2 \hookrightarrow O_1 \cdot O_2, \prec_1 \cdot \prec_2)}$$

OPERATOR

$$\frac{\vdash^O op : (I \hookrightarrow O, \prec) \qquad I = ((S_0, B_0), \ldots (S_n, B_n)) \qquad \forall i.\ type^C(s_i) = S_i}{\vdash \{(s_0, \ldots, s_n)\}[op] : (I \hookrightarrow O, (\prec))}$$

Fig. 5. Type semantics for graphs of a Flo program.

$$\begin{aligned} inputs(e_1; e_2) &\triangleq inputs(e_1) \\ inputs(e_1 \mid e_2) &\triangleq inputs(e_1) \cdot inputs(e_2) \\ inputs(\{I\}[op]) &\triangleq I \end{aligned}$$

$$\begin{aligned} setinput(e_1; e_2, I) &\triangleq setinput(e_1, I); e_2 \\ setinput(e_1 \mid e_2, I_1 \cdot I_2) &\triangleq setinput(e_1, I_1) \mid setinput(e_2, I_2) \\ setinput(\{I\}[op], I') &\triangleq \{I'\}[op] \qquad when\ |I| = |I'| \end{aligned}$$

SEQUENCE-LEFT

$$\frac{e_1 \rightarrow^\Delta (e_1', I')}{(e_1; e_2) \rightarrow^\Delta (e_1'; setinput(e_2, \lfloor [\![inputs(e_2)]\!]^C + I' \rfloor^C), \varnothing)}$$

SEQUENCE-RIGHT

$$\frac{e_2 \rightarrow^\Delta (e_2', O')}{(e_1; e_2) \rightarrow^\Delta (e_1; e_2', O')}$$

PAR-LEFT

$$\frac{e_1 \rightarrow^\Delta (e_1', O_1')}{(e_1 \mid e_2) \rightarrow^\Delta (e_1' \mid e_2, O_1', \varnothing)}$$

PAR-RIGHT

$$\frac{e_2 \rightarrow^\Delta (e_2', O_2')}{(e_1 \mid e_2) \rightarrow^\Delta (e_1 \mid e_2', \varnothing, O_2')}$$

OPERATOR

$$\frac{([\![I]\!]^C, op) \rightarrow^\delta (I', op', O')}{(\{I\}[op]) \rightarrow^\Delta (\{\lfloor I' \rfloor^C\}[op'], O')}$$

Fig. 6. Small-step semantics for graphs of a Flo program.

Before we continue, let us prove that graphs satisfy preservation.

Lemma 4.1 (Graph Preservation). *Given a graph g of type $(I \hookrightarrow O, \prec)$, output state $S = (s_0 \dots s_n)$, and updated output state $S' = (s'_0 \dots s'_n)$ such that $O = ((T_0, \_) \dots (T_n, \_))$ and $\forall i.\ type^C(s_i) = T_i$, if (g, S) takes a step to (g', S'), then g' is also of type $(I \hookrightarrow O, \prec)$ and $\forall i.\ type^C(s'_i) = T_i$.*

Proof. We can prove this by structural induction over the graph.

Base Case: A graph with a single operator. By operator preservation, we know that the type of I is the same as the type of I', that op' has the same type, and that O' has the same type as O. Therefore, the graph as a whole has the same type and the output is of the correct type.

Inductive Step: Proof by cases:

Sequential Composition: If we apply the sequence-left rule, then by induction we know that e_1 has the same type as e'_1, and I' has the same types as the inputs of e_2. Therefore, when we set the inputs of e_2 to I', we preserve the typing (due to well-formedness of the denotational lifting and syntactical lowering). Since the output is unchanged, we satisfy preservation.

If we apply the sequence-right rule, then by induction we know that e_2 has the same type as e'_2, and the output has the same type due to concatenation. Therefore, we satisfy preservation.

Parallel Composition: In both rules, we use induction to know the types of both sides are preserved. The typing rule for parallel simply composes these types, so we are done. □

4.1 Graph Stuck State

Now, let us extend the properties we require of operators to graphs as a whole. First, we will extend Operator Stuck State (Lemma 3.1).

Lemma 4.2 (Graph Stuck State). *Given a graph initialized with a fixed set of input collection values, running the graph will eventually reach a stuck state where no additional steps can be taken.*

Proof. We can prove this by structural induction over the graph.

Base Case: A graph with a single operator. By Lemma 3.1.

Inductive Step: A graph such that its subgraphs satisfy Graph Stuck State. Proof by cases:

Sequential Composition: There are only two small steps that can be taken at any point, for the left or right. If we only step one of the two subgraphs, by induction that side will eventually reach a stuck state. If the left side reaches a stuck state, then running the right side will never re-enable the left side by the definition of $\rightarrow$. If the right side reaches a stuck state, we may be able to run the left side which may re-enable the right side, but this will cycle back to the left and eventually the left side will be stuck. Therefore, the graph as a whole will reach a stuck state.

Parallel Composition: The two subgraphs are independent, and so by the inductive hypothesis we know that both will eventually reach a stuck state, and their composition is a stuck state. □

4.2 Determinism and Eager Execution

The most significant change between reasoning about operators in isolation and the composition of them is that at any point when executing a graph, there may be multiple small steps for each operator that can be taken. We need to prove we can non-deterministically execute these operators while arriving at the same output. To prove this for all graphs, we will also need to extend Eager Execution to graphs. These proofs are mutually recursive, so we will prove them simultaneously. Both our definitions look nearly identical to those for operators, just with the use of the general small step relation rather than only for operators.

A quick aside on notation. In this section, we will use the shorthand $\{I\}g$ to denote a graph g whose inputs are set to I, so $\{I\}g = setinput(g, I)$.

Definition 4.1 (Determinism)**.** Consider a graph g. For all inputs I and initial outputs O where a small step for $(\{I\}g, O)$ exists, there exists a later state g', inputs I', and outputs O' such that in every trace of small steps $(\{I\}g, O) \rightarrow^* (\{I'\}g', O')$ we eventually reach this later state.

Note that combined with stuck states (Lemma 4.2), this implies that every graph will eventually reach a **unique** stuck state. This is because we can always take a series of steps to arrive at the same later state, and eventually we will reach a point where no more steps can be taken.

Definition 4.2 (Eager Execution)**.** Consider a graph g. For all input streams I, output streams O, delta set Δ, updated graph g', input stream, I', and output stream O' such that

$$(\{I\}g, O) \rightarrow (\{I'\}g', O')$$

there exists a stuck state f such that

$$(\{I +\!\!+ \Delta\}g, O) \rightarrow^* f \text{ and } (\{I' +\!\!+ \Delta\}g', O') \rightarrow^* f$$

Lemma 4.3. *Consider any expression. It must satisfy:*

(1) Determinism
(2) Eager Execution

Proof. We can prove this by structural induction over the graph.
Base Case: A graph with a single operator.

(1) By confluence of $\rightarrow^O$.
(2) By Definition 3.1.

Inductive Step: A graph such that its subgraphs satisfy both (1) and (2). Proof by cases:
Sequential Composition: a graph of form $a; b$

(1) We know that there is at least one small-step that can be taken, and the only options are to recursively step a or b. Let us define an *execution trace* that captures an ordered sequence of small-steps to take. This trace will have the form "$(a_i|b)+$", with each element directing us to take the corresponding small step corresponding to the named subgraph, with the indices for a counting up from 0. Given a trace $t =$"$s_0 \ldots s_n$", we define $\rightarrow_t$ to take the steps in order. For each instance of a_i, the index lets us uniquely identify the small-step rule that will be applied to a. For b, the token represents taking any small-step on b. We will call a trace after which no more steps can be taken a *terminating trace*.
Next, let us define equivalence between a pair of traces t_1 and t_2. Two traces are equivalent if executing both on the same initial state results in the same final state, *even* with non-deterministic selection of which small-step to run for each b. We will prove that for any pair of terminating traces t_1 and t_2, the traces are equivalent.
Consider a trace of the form "*prefix* $b\ a_i\ \ldots\ a_j\ b^*$". The execution of this looks like

$$(\{I_a^p\}a^p; \{I_b^p\}b^p, O^p) \rightarrow_{prefix} (\{I_a\}a; \{I_b\}b, O) \rightarrow_b (\{I_a\}a; \{I_b'\}b', O')$$
$$\rightarrow_{a_{i\ldots j}} (\{I_a'\}a'; \{I_b''\}b', O') \rightarrow_b^* (\{I_a'\}a'; \{I_b'''\}b'', O'')$$

First, by the definition of $\rightarrow^\Delta$, we know that $I_b'' = I_b' +\!\!+ \Delta_i +\!\!+ \Delta_{i+1} \ldots$. Then, inductively Eager Execution applied to b lets us rewrite "$b\ a_i\ \ldots\ a_j\ b^*$" to "$a_i\ \ldots\ a_j\ b^*$" (note that the number of trailing b in the rewritten suffix may be arbitrary), because the execution of $a_i\ \ldots\ a_j$ simply introduces additional data for b to process. This results in the following execution

$$(\{I_a^p\}a^p; \{I_b^p\}b^p, O^p) \rightarrow_{prefix} (\{I_a\}a; \{I_b\}b, O)$$
$$\rightarrow_{a_{i\ldots j}} (\{I_a'\}a'; \{I_b +\!\!+ \Delta_i +\!\!+ \Delta_{i+1} \ldots\}b, O) \rightarrow_b^* (\{I_a'\}a'; \{I_b'''\}b'', O'')$$

Therefore, the trace *prefix* $b\ a_i\ \ldots\ a_j\ b^*$ is equivalent to *prefix* $a_i\ \ldots\ a_j\ b^*$.
If we repeatedly apply this rewrite to both traces to pull all a_i to the front, we will arrive at two traces of the form $a_0\ \ldots\ a_n\ b^*$ and $a_0\ \ldots\ a_m\ b^*$. We know that both original traces are terminating, therefore after running $a_0\ \ldots\ a_n$ and $a_0\ \ldots\ a_m$ even though the bs between the elements have been removed, there will be no more small steps that can be taken on a. By determinism from induction, since a has terminated the traces $a_0\ \ldots\ a_n$ and $a_0\ \ldots\ a_m$ result in the same state and are equivalent. Similarly, because our rewrites preserve equivalence, by determinism we know that after running b^* on both traces, we will reach the same final state. Therefore, the traces are equivalent and $a; b$ satisfies determinism.

(2) We can split into cases based on the small step that could be taken.
Case 1: The small step is on a. By Definition 4.2, we know that we can introduce the delta before or after the small step on a and then continue running small steps for a until reaching the common later state for a, which is also our overall later state f.
Case 2: The small step is on b. If we run the small step, then introduce the delta, let the state immediately after introducing the delta be f. If we instead first introduce the delta, then run b, the state after is also f because running the small step for b is unaffected by the introduction of the delta.

Parallel Composition: a graph of form $a|b$

(1) The small steps for a parallel composition just run the small steps for either side, which are independent. Therefore by induction both sides will step to a deterministic state.
(2) In parallel composition, the introduction of a delta results in independent chunks being added to both sides. If we step the graph first, that just steps one of the sides, so the inductive hypothesis holds on one of the sides and the other side is unaffected.

□

4.3 Streaming Progress

Lemma 4.4 (Streaming Progress for Graphs). *Consider a well-typed graph g with type* $\vdash g : (((I_0, B_{I,0}) \ldots (I_n, B_{I,n})) \hookrightarrow ((O_0, B_{O,0}) \ldots (O_m, B_{O,m})), \prec)$ *such that inputs*$(g) = i_0 \ldots i_n$ *and* $B_{I,j} = \mathbf{B} \implies$ *fixed*(i_j). *Consider all well-typed outputs O and $o'_0 \ldots o'_m$ such that*

$$(g, O) \rightarrow^* (g', (o'_0, \ldots, o'_m))$$

and $(g', (o'_0, \ldots, o'_m))$ is stuck. Then o'_j must be fixed if $B_{O,j} = \mathbf{B}$ and there must also be a stuck state

$$(\{(fix(i_0), \ldots, fix(i_n))\}g, O) \rightarrow^* (g'', (fix(o'_0), \ldots, fix(o'_m)))$$

Proof. We can prove this by structural induction over the graph.
Base Case: A graph with a single operator. By Definition 3.3.
Inductive Step: Proof by cases:
Sequential Composition: We can apply Lemma 4.3 to only focus on traces where we run the left half until stuck state and then the right half. First, we apply streaming progress to the left half, which tells us that we will output intermediate collections such that each output with a bounded stream type will have a fixed value. This satisfies the premise for induction on the right subgraph, so we can apply streaming progress again to know that each bounded output will be fixed. Using the same proof structure, we know that the intermediate collections will be maximal with respect to the unbounded inputs, so the final outputs will be maximal as well.
Parallel Composition: Because both sides are independent, we can simply use induction on each side. Because all bounded outputs will be fixed and all outputs are maximal with respect to the unbounded inputs, we satisfy streaming progress. □

5 Nested Streams and Graphs

So far, we have considered dataflow programs with a direct path of operators from each input to the outputs. But for many applications, it is necessary to perform stateful, iterative computations over an input stream. In Flo, we tackle this using constructs for **nested streams and graphs**.

Before we dive into formal semantics, let us lay out a high-level overview of our approach to nesting. First, we introduce nested streams, which are a specific type of stream that encapsulate several smaller streams. We define a set of restrictions for how operators must generate such nested streams, in particular how boundedness of the inner streams is enforced.

Once we have nested streams, we need an operator that can transform them. This is where the `nest` operator comes in, which makes it possible to transform a nested stream by defining a nested Flo graph that should be run on each inner stream. We introduce the `write_defer` and `read_defer` operators, which can be used to pass state across the iterations for each inner stream to enable iterative computation. We prove that these operators satisfy all the core operator properties, therefore preserving the high-level guarantees we have established for Flo.

5.1 Nested Streams

Our definition of Flo so far has dealt only with an abstract notion of collections and operators. But the `nest` operator is a concrete instance, and so we also need a concrete collection type for it to consume and produce. Furthermore, this collection type must store nested streams in a way that preserves boundedness properties and allows the inner graph to manipulate the inner streams.

To tackle this, we introduce the **ordered sequence of streams** in Figure 7. This collection type, denoted $[(S_0, \ldots S_n)]$ is parameterized over several inner stream types $S_i = (C_i, B_i)$. Values of this type are stored as a list of tuples $[(c_{0,0}, \ldots c_{0,n}), \ldots, (c_{m,0}, \ldots c_{m,n})]$, where each $c_{i,j}$ is a value of type C_j. The terminator symbol $\otimes$ indicates the end of a stream.

$$
\begin{aligned}
[((C_1,B_1),\ldots,(C_m,B_m))] \triangleq \{\ & [(c_{1,1},\ \ldots),\ldots,(c_{n,1},\ \ldots)] \mid \\
& \forall i, j\ c_{i,j} \in C_j \land (i > 1 \land B_j = \mathbf{B}) \implies \mathit{fixed}(c_{i,j}) \\
\} \cup \{\ & [\otimes, (c_{1,1},\ \ldots),\ldots,(c_{n,1},\ \ldots)] \mid \\
& \forall i, j\ c_{i,j} \in C_j \land (B_j = \mathbf{B}) \implies \mathit{fixed}(c_{i,j})\} \\
& [\otimes, \ldots] +\!\!+ x = [\otimes, \ldots] \\
& [c_1, \ldots, c_n] +\!\!+ \otimes = [\otimes, c_1, \ldots, c_n] \\
& [c_1, \ldots, c_n] +\!\!+ ((v_1, \ldots, v_m), \mathit{true}) = [(v_1, \ldots, v_n), c_1, \ldots, c_n] \\
& [(v_1, \ldots, v_m), \ldots, c_n] +\!\!+ ((\delta_1, \ldots, \delta_m), \mathit{false}) = [(v_1 +\!\!+ \delta_1, \ldots, v_m +\!\!+ \delta_m), \ldots, c_n]
\end{aligned}
$$

Fig. 7. The collection type and concatenation operator for the ordered sequence of streams.

The concatenation operator on this collection type takes an ordered sequence of streams and *either* the terminator $\otimes$, the tuple of the boolean true and a tuple of collections values matching the inner stream types, or a tuple of the boolean false and a tuple of concatenation values corresponding to the right-hand side accepted by $+\!\!+$ for each inner stream type. If the boolean flag is true, the concatenation operator extends the collection with the tuple used as the new leftmost value. If it is false, the operator uses the concatenation operator of each of the inner stream types to extend the existing leftmost collections with the new values.

There is another key concern we need to address. Once a new tuple of collections is pushed into the ordered sequence, none of the other tuples will ever grow through concatenation. We need to

ensure that these finalized tuples satisfy the restrictions of the inner stream types; in particular that they satisfy boundedness properties. To do this, we require that all tuples of collections after the leftmost one have fixed collections for each bounded stream type.

5.2 Nesting Graphs

The `nest` operator maps nested streams by transforming their inner streams one-by-one using an inner Flo graph. These inner graphs have special privileges: they can define *iterative computations* by passing data across executions on subsequent inner streams. To do this, developers use pairs of `read_defer` and `write_defer` operators with matching keys. Any data sent to a `write_defer` operator will be emitted by the corresponding `read_defer` operator when processing the next inner stream (for the first step, `read_defer` takes an initial value as a parameter).

Before we dive into the formal semantics of these operators, let us walk through a simple example to show how `nest`, `write_defer`, and `read_defer` can be combined to enable iterative computation. We will implement a classic iterative algorithm where we are given a set of directed edges and want to compute which nodes are reachable from a root within a fixed radius. Our algorithm starts with a single root node, and in a loop identifies the next "layer" of reachable nodes.

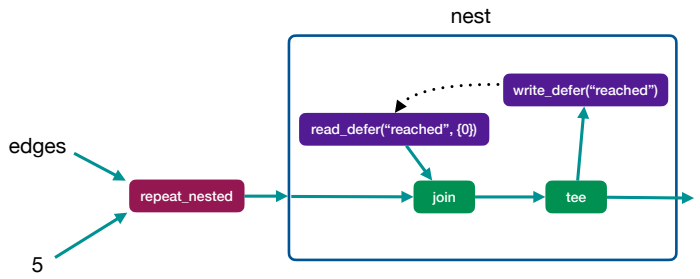


Fig. 8. An example of identifying nodes within a fixed radius using nested graphs.

First, we need a collection type for sets of nodes and sets of edges (using standard semantics), along with some operators inspired by relational algebra. We omit the detailed semantics for brevity, but these are straightforward to define. The `join` operator takes in a set of nodes and a set of edges, and identifies the destination of all edges originating at a node in the input set. The `tee` operator consumes a single stream and emits a pair of streams, each duplicating the input.

Next, we must generate a stream-of-streams that drives the nested graph. For graph reachability within a fixed radius n, we need to run n iterations of the inner graph. To achieve this, we introduce a `repeat_nested` operator which consumes a stream and a natural number singleton k, and emits a stream with k inner streams, each of which duplicates the contents of the input.

Putting these operators together, we show how to implement this algorithm in Figure 8. On every iteration, we first collect the nodes reached up to the previous iteration using `read_defer`, with an initial value of just the root node 0. Then, we emit the next layer of reachable nodes and also send them to `write_defer` to be used in the next iteration. In the output of this program, we will have a stream of sets of nodes, where each set contains the nodes reachable from the root with increasing radii up to the fixed limit.

In Flo, `nest` is a standard operator that satisfies all the proof obligations, so it can be... nested! This makes it possible to build arbitrarily complex nested cycles. For example, we can tweak the graph reachability example to allow recomputing the reachability analysis with extended radii. In this algorithm, we can use the output from a previous query to "bootstrap" the next query, and only run iterations to extend the radius rather than starting from scratch.

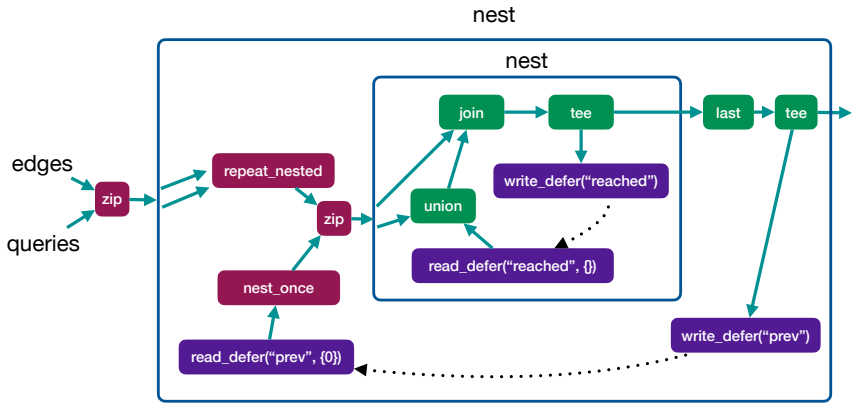


Fig. 9. An example of graph reachability with a dynamic radius, using nested cycles.

In this example program, we assume that the input edges have already been shaped into an unbounded stream-of-streams where each inner stream contains the full set of edges[2]. The queries, which represent expansions of the radius, are also a stream-of-streams where each inner stream is a singleton containing the amount to expand the radius by. We use a new `zip` operator to feed multiple nested streams into `nest` by tupling their inner streams pairwise.

We use a new `last` operator to extract the final set emitted by reachability, which we defer to bootstrap the next query. To inject these nodes, we use a new operator `nest_once` which generates an infinite stream-of-streams where the first inner stream contains the input and the rest are empty. Then, inside the reachability graph, we use `union` (which performs set union) to add the bootstrap nodes. Finally, we use `repeat_nested` as before to drive iterations of graph reachability.

5.3 Type Semantics

Now, we are ready to lay out the formal semantics for nested graphs, beginning with the type semantics. First, we define the `defer` operators: `write_defer` takes a key as a parameter and accumulates a bounded stream as input, and on the next iteration any matching `read_defer` with the same key will emit the accumulated collection. Type-safety for these operators is a bit more complex, since we need to ensure that there is a single `write_defer` for each key and that the stream types being written match the types being read.

To achieve this, we introduce a new pair of contexts R and W to our typing rules ($\vdash$ and $\vdash^O$) which each store a map from keys to stream types. We will use context W substructurally, admitting only exchange (but not weakening or contraction) on this context. When typing a nested graph, these contexts are set to (arbitrary) identical values, which enforces that the same types are written and read. On the write-side, we also enforce that each key is written exactly once by splitting the W keys at each composition until there is one key isolated to each `write_defer`. For `read_defer`, we have two variants because the optional second parameter stores a value to be emitted.

The `nest` operator takes a graph g of type $I \hookrightarrow O$ with partial order $\prec_g$. Each stream in O must be **bounded** so that the inner graph finishes in finite time. The operator itself takes a stream of streams and emits a stream of streams, where the inner types are I and O respectively. The boundedness of the outer output (denoted X) is the same as the outer input. We also include a variant of `nest` with an additional parameter that stores the initial graph for the next iteration. We re-define our

[2]We could also consume the set of edges only once and "persist" them across iterations of the nested graph by sending a copy across a `defer` cycle. But that adds complexity to this example that distracts from nested cycles.

$$\textsc{sequence}\quad \frac{R; W_1 \vdash e_1 : (I_1 \hookrightarrow O_1, \prec_1) \qquad R; W_2 \vdash e_2 : (I_2 \hookrightarrow O_2, \prec_2) \qquad O_1 \leq I_2}{R; W_1, W_2 \vdash e_1; e_2 : (I_1 \hookrightarrow O_2, \prec_1)}$$

$$\textsc{par}\quad \frac{R; W_1 \vdash e_1 : (I_1 \hookrightarrow O_1, \prec_1) \qquad R; W_2 \vdash e_2 : (I_2 \hookrightarrow O_2, \prec_2)}{R; W_1, W_2 \vdash e_1 \mid e_2 : (I_1 \cdot I_2 \hookrightarrow O_1 \cdot O_2, \prec_1 \cdot \prec_2)}$$

$$\textsc{operator}\quad \frac{R; W \vdash^O op : (I \hookrightarrow O, \prec) \qquad I = ((S_0, B_0), \ldots (S_n, B_n)) \qquad \forall i.\ type^C(s_i) = S_i}{R; W \vdash \{(s_0, \ldots, s_n)\}[op] : (I \hookrightarrow O, (\prec))}$$

$$\textsc{read-defer-value-type}\quad \frac{type^C(v) = C \qquad fixed([\![v]\!]^C)}{R, k : C; \emptyset \vdash^O \text{read\_defer(k, v)} : (() \hookrightarrow (C, \mathbf{B}), \emptyset)}$$

$$\textsc{read-defer-no-value-type}\quad R, k : C; \emptyset \vdash^O \text{read\_defer(k)} : (() \hookrightarrow (C, \mathbf{B}), \emptyset)$$

$$\textsc{write-defer-type}\quad R; k : C \vdash^O \text{write\_defer(k)} : ((C, \mathbf{B}) \hookrightarrow (), \emptyset)$$

$$\textsc{nest-type}\quad \frac{D; D \vdash g : (I \hookrightarrow (O_1, \ldots), \prec_g) \qquad \forall i.\ O_i = (C_i, \mathbf{B})}{R; \emptyset \vdash^O \text{nest}(g) : (([I], X) \hookrightarrow ([(O_1, \ldots)], X), \prec_{\text{nest}} (\prec_g))}$$

$$\textsc{nest-with-copy-type}\quad \frac{D; D \vdash g : (I \hookrightarrow (O_1, \ldots O_m), \prec_g) \qquad D; D \vdash g_o : (I \hookrightarrow (O_1, \ldots O_m), \prec_g) \qquad O_i = (C_i, \mathbf{B})}{R; \emptyset \vdash^O \text{nest}(g, g_o) : (([I], X) \hookrightarrow ([(O_1 \ldots O_m)], X), \prec_{\text{nest}} (\prec_g))}$$

Fig. 10. Type semantics with defer contexts, and for `read_defer`, `write_defer`, and `nest`.

core composition type semantics with these contexts as well as for `write_defer`, `read_defer`, and `nest` in Figure 10. Note that this requires a modification to the full type system; we do this in the usual way. In particular, note that as existing operators never have graphs as subterms, they will be lifted into our context-enhanced system with arbitrary R and empty W contexts.

5.4 Operational Semantics

The `nest` operator processes tuples of inner streams one-by-one, maintaining the current inner streams at the rightmost element of the input. It shifts to the next tuple of inner streams once the graph reaches a stuck state and all the outputs (including those to `write_defer`) are fixed. The `nest` operator first stores a copy of the initial graph as a second parameter (this variant is lower in the partial order for `nest`). To process an inner stream, we use *setinput* to set the inner graph inputs, step the inner graph, and then use *inputs* to propagate input consumption to the nested stream. Once the input only contains a terminator, the operator emits a terminator as well.

Note that `write_defer` has no small-step rules; its behavior is handled by the semantics for `nest`. The `read_defer` operator takes a single small-step, which emits its collection parameter. This collection parameter is either a default value (for the first tuple of inner streams) or a value from `write_defer`. When shifting to the next inner stream input, we use the *collect_defer* helper to accumulate the inputs to each `write_defer` into a map, and then use the *set_defer* helper to create a copy of the initial graph with the corresponding `read_defer` operators updated to use those collections. We visualize this behavior in Figure 11 where a stream-of-streams on the left, with later elements lower, is transformed into another stream-of-streams. We then lay out the formal operational semantics in Figure 12.

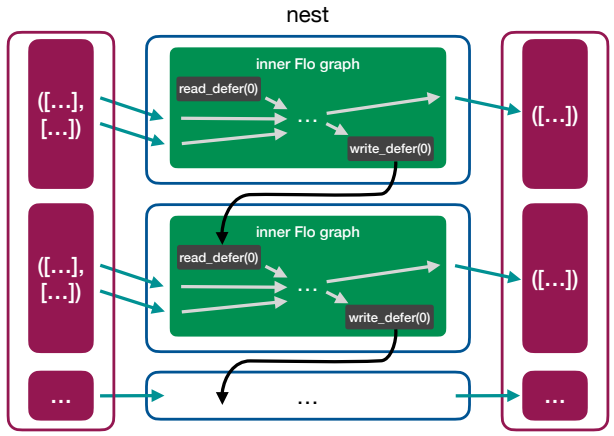


Fig. 11. Visualization of the `nest`, `read_defer`, and `write_defer` operators, where the nested streams on the left and right have later elements lower.

$$
\begin{aligned}
\mathit{collect\_defer}(e_1; e_2) &\triangleq \mathit{collect\_defer}(e_1) \cup \mathit{collect\_defer}(e_2) \\
\mathit{collect\_defer}(e_1 \mid e_2) &\triangleq \mathit{collect\_defer}(e_1) \cup \mathit{collect\_defer}(e_2) \\
\mathit{collect\_defer}(\{I\}[\mathit{write\_defer}(k)]) &\triangleq \{k : I\} \\
\mathit{collect\_defer}(\{I\}[op]) &\triangleq \otimes \qquad \text{when } op \neq \mathit{write\_defer}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{set\_defer}(e_1; e_2, M) &\triangleq \mathit{set\_defer}(e_1, M); \mathit{set\_defer}(e_2, M) \\
\mathit{set\_defer}(e_1 \mid e_2, M) &\triangleq \mathit{set\_defer}(e_1, M) \mid \mathit{set\_defer}(e_2, M) \\
\mathit{set\_defer}(\{\}[\mathit{read\_defer}(k, v)], M) &\triangleq \{\}[\mathit{read\_defer}(k, M[k])] \\
\mathit{set\_defer}(\{I\}[op], M) &\triangleq \{I\}[op] \qquad \text{when } op \neq \mathit{read\_defer}
\end{aligned}
$$

NEST-FIRST

$$\frac{I \neq \otimes}{([\ldots, I], \mathrm{nest}(g)) \to^{\delta} ([\ldots, I], \mathrm{nest}(g, g), ((\bot, \ldots, \bot), \mathit{true}))}$$

NEST-FIRST-FIXED

$$([\otimes], \mathrm{nest}(g)) \to^{\delta} ([\otimes], \mathrm{nest}(g, g), \otimes)$$

NEST-RUN-GRAPH

$$\frac{(\mathit{setinput}(g, \lfloor I \rfloor^C)) \to^{\Delta} (g', (O'_1, \ldots, O'_m))}{([\ldots, I], \mathrm{nest}(g, g_o)) \to^{\delta} ([\ldots, [\![\mathit{inputs}(g')]\!]^C], \mathrm{nest}(g', g_o), ((O'_1, \ldots, O'_m), \mathit{false}))}$$

NEST-RUN-STEP

$$\frac{(\mathit{setinput}(g, \lfloor I \rfloor^C), (O_1, \ldots, O_m)) \text{ is stuck} \qquad \forall m.\mathit{fixed}(O_m) \qquad \forall_{d \in \mathit{collect\_defer}(g)}\ \mathit{fixed}(d) \qquad I_{next} \neq \otimes}{([\ldots, I_{next}, I], \mathrm{nest}(g, g_o)) \to^{\delta} ([\ldots, I_{next}], \mathrm{nest}(\mathit{set\_defer}(g_o, \mathit{collect\_defer}(g)), g_o), ((\bot, \ldots, \bot), \mathit{true}))}$$

NEST-RUN-FIXED

$$\frac{(\mathit{setinput}(g, \lfloor I \rfloor^C), (O_1, \ldots, O_m)) \text{ is stuck} \qquad \forall m.\mathit{fixed}(O_m)}{([\otimes, I], \mathrm{nest}(g, g_o)) \to^{\delta} ([\otimes], \mathrm{nest}(g_o, g_o), \otimes)}$$

READ-DEFER-EMIT

$$((), \texttt{read\_defer}(k, v)) \to^{\delta} ((), \texttt{read\_defer}(k), v)$$

Fig. 12. Small-step semantics for the `nest` and `read_defer` operators.

5.5 Operator Properties

Because `nest` is a standard operator, it must satisfy all Flo's core operator properties. First, we define the partial order $\prec_{\texttt{nest}}\ (\prec_g)$, which is parameterized over the partial order for the inner graph. Our small step semantics either consume the rightmost input inner stream or reduce it according to the nested graph's partial order. So we have

$$[...] \prec_{\texttt{nest}}\ (\prec_g) [..., I]$$

$$[..., I'] \prec_{\texttt{nest}}\ (\prec_g) [..., I] \text{ if } I' \prec_g I$$

$$\otimes \prec_{\texttt{nest}}\ (\prec_g) [...]$$

For `read_defer`, any operator expression *without* the value parameter is smaller than any with it, so the step for `read_defer` reduces the operator expression. Since `write_defer` takes no steps, it satisfies our operator proof obligations trivially. We can now prove the properties of `nest`.

Operator Well-Formedness. When we step across an input (NEST-RUN-STEP and NEST-RUN-FIXED), the input is updated to a prefix, which satisfies our first case of the partial order. The only other rule that modifies inputs is NEST-RUN-GRAPH, which will only touch the inputs if it recursively steps a left-most operator that consumes those inputs. Because of Lemma 3.1, we know that running any of these operators will reduce the input along the partial order for the inner graph. □

Operator Preservation. There are only two ways we modify the inputs and outputs; either we push or pop an entire tuple of inner streams or update the rightmost input or leftmost output. In the first case, we only push $\perp$, and popping does not affect the type of the collection. When we update an input/output instead, Lemma 4.1 guarantees that this is safe. In all our rules, the operator is only changed by setting the inputs of the graph, which is safe because the input types are unchanged. □

Operator Determinism. First, NEST-FIRST or NEST-FIRST-FIXED will execute, then NEST-RUN-GRAPH will run until stuck state, then NEST-RUN-STEP will run, until the input stream is fixed and NEST-RUN-FIXED is run. In NEST-RUN-GRAPH, the only rule where we recursively apply a step, we know that the stuck state exists (Lemma 4.2) and is deterministic (Lemma 4.3). Therefore, `nest` is deterministic. □

Eager Execution. For NEST-FIRST-FIXED and NEST-RUN-FIXED, because the input collection is already fixed deltas have no effect. For NEST-FIRST, regardless of whether the delta is introduced before or after, the final state will be the same because we copy the input as-is and a concatenation will never affect $I \neq \otimes$ because an element can never be replaced by the terminator. Because NEST-RUN-GRAPH will run until the inner graph reaches a stuck state, we can apply Lemma 4.3 to know that introducing a delta before or after the step will result in the same final state, because introducing a delta to the nested stream will only affect the last element I. For NEST-RUN-STEP, the delta will never affect I, and any delta to I_{next} will be applied the same before or after the step. □

Streaming Progress. If the input is bounded, the output will become fixed because each iteration will finish in finite time by Lemma 4.4. If it is unbounded, the input sequence being fixed only affects NEST-FIRST-FIXED and NEST-RUN-FIXED rules, which simply concatenate a terminator to the output sequence without modifying it in any other way. □

6 Case Studies

Flo aims to provide strong guarantees that are meaningful across a range of applications while remaining sufficiently abstract to capture a variety of semantics. In this section, we demonstrate the expressiveness of Flo by using it to implement the key ideas found in existing streaming languages. Note that our goal is **not** to show how to implement these entire languages in Flo, rather that key ideas from them can be expressed and satisfy our properties. We focus on three existing languages:

(1) Flink [15], a popular streaming framework that features windowed aggregation functions.
(2) LVars [30], a language for parallel programming that uses lattices to ensure determinism.
(3) DBSP [13], a system for incremental view maintenance that uses z-sets to model relations.

6.1 Flink

Flink [15] is a classic example of a streaming dataflow language. Like Flo, Flink uses compositions of operators to describe computations over streams. A key technique from Flink is the use of *windows* to enable aggregations over fixed-time intervals of an infinite stream. We will show that this idea from Flink can be modeled in Flo as a timestamped collection type, where windowing operators generate streams-of-streams which can then be aggregated in a nested graph.

Flink uses ordered sequences as its primary semantics for streams. We can model this in Flo by introducing an ordered sequence collection, which simply stores a list of values where the newest items are on the left and the oldest elements on the right. We define this collection in Figure 13.

$$\texttt{S<V>} \triangleq \{[v_1, \ldots, v_n] \mid \forall i.\ v_i \in V\} \cup \{[\otimes, v_1 \ldots v_n] \mid \forall i.\ v_i \in V\}$$
$$[v_1, \ldots, v_n] +\!\!+ [d_1, \ldots, d_m] = [d_1, \ldots, d_m, v_1, \ldots, v_n]$$
$$[\otimes, \ldots] +\!\!+ x = [\otimes, \ldots]$$
$$[v_1, \ldots, v_n] +\!\!+ \otimes = [\otimes, v_1, \ldots, v_n]$$

Fig. 13. Collection type and concatenation operator for ordered sequences in Flo.

With a collection type for ordered sequences, we can define classic operators found in Flink such as `map`. We can also define semantics for `fold` that matches the Flink semantics of emitting a stream containing a single value, which is the result of the aggregation. We can define the type and operational semantics for these operators in Figure 14 (we omit partial orders for brevity).

MAP-TYPE
$$\frac{\vdash f : T \to U}{\vdash^O \mathrm{map}(f) : ((\texttt{S<T>}, X) \hookrightarrow (\texttt{S<U>}, X), \prec_{\mathrm{map}})}$$

MAP
$$\frac{f(h) \Downarrow u}{([\ldots, h], \mathrm{map}(f)) \to^\delta ([\ldots], \mathrm{map}(f), [u])}$$

MAP-TERMINATOR
$$([\otimes], \mathrm{map}(f)) \to^\delta (\otimes, \mathrm{map}(f), \otimes)$$

FOLD-TYPE
$$\frac{\vdash acc : U \qquad \vdash f : (U, T) \to U}{\vdash^O \mathrm{fold}(acc, f) : ((\texttt{S<T>}, B) \hookrightarrow (\texttt{S<U>}, B), \prec_{\mathrm{fold}})}$$

FOLD
$$\frac{f(acc, h) \Downarrow acc'}{([\ldots, h], \mathrm{fold}(acc, f)) \to^\delta ([\ldots], \mathrm{fold}(acc', f), [])}$$

FOLD-TERMINATOR
$$([\otimes], \mathrm{fold}(acc, f)) \to^\delta (\otimes, \mathrm{fold}(acc, f), [\otimes, acc])$$

Fig. 14. Operational semantics for Flink operators in Flo.

Map processes elements one by one and passes through the terminator, so it satisfies eager execution and streaming progress easily. But note that for the fold operator to satisfy streaming progress, its input must be bounded, otherwise the step that emits the aggregated value when the input becomes fixed would be illegal.

Now given an unbounded stream, how do we use fold? Flink's answer is to use windows, where the aggregation is run over blocks of data defined by timestamp intervals. This idea maps perfectly to the Flo model, where we can convert an unbounded stream of timestamps-value pairs into a stream-of-streams (as in Section 5) and then use a nested graph to aggregate over each window.

To implement this windowing operator, we will use the internal state of the operator to store the values corresponding to the next window. When a timestamp farther than the end of the current interval is received, we emit the accumulated window. Because the operator uses timestamp boundaries to determine when to emit inner streams, the inner streams are bounded even though the outer stream-of-streams is unbounded. We omit detailed proofs for brevity, but this operator also satisfies eager execution and streaming progress. We can sketch the type and operational semantics for this operator in Figure 15 (again omitting the partial order for brevity).

WINDOW-TYPE

$$\frac{\textit{interval} \text{ is an amount of time} \qquad T \text{ is a timestamp} \qquad \forall i\ t_i \text{ is a timestamp} \qquad \forall i\ \vdash v_i : D}{\vdash^{O} \text{window}(\textit{interval}, [(v_1, t_1), \ldots (v_n, t_n)]) : ((\texttt{S<(D, T)>}, X) \hookrightarrow ([(\texttt{S<D>}, B)], X), \prec_{\text{window}})}$$

WINDOW-FIRST

$$([\ldots (v_n, t_n)], \text{window}(\textit{interval}, [])) \rightarrow^{\delta} ([\ldots], \text{window}(\textit{interval}, [(v_n, t_n)]), [])$$

WINDOW

$$\frac{t_n - wt_m \leq \text{interval}}{([\ldots (v_n, t_n)], \text{window}(\textit{interval}, [(w_1, wt_1), \ldots, (w_m, wt_m)])) \rightarrow^{\delta} ([\ldots], \text{window}(\textit{interval}, [(v_n, t_n), (w_1, wt_1), \ldots, (w_m, wt_m)]), [])}$$

WINDOW-EMIT

$$\frac{t_n - wt_m > \text{interval}}{([\ldots (v_n, t_n)], \text{window}(\textit{interval}, [(w_1, wt_1), \ldots, (w_m, wt_m)])) \rightarrow^{\delta} ([\ldots], \text{window}(\textit{interval}, [(v_n, t_n)]), ([w_1, \ldots, w_m], \textit{true}))}$$

Fig. 15. Type and operational semantics for the `window` operator.

To complete our example of how patterns from Flink can be modeled in Flo, we can perform aggregations over these windows by using a nested graph. We can pass the result of the window operator into the `nest` operator defined in Section 5, and use the `fold` operator inside the nested graph. Because the nested stream is bounded, this will typecheck and the aggregation will be appropriately computed for each window.

6.2 LVars

LVars [30] is a language for deterministic parallel programming that uses lattice-based data structures to ensure determinism. A key insight of LVars is to leverage monotonicity to ensure determinism, by requiring that pieces of state are always updated monotonically, and restricting reads of the state to threshold queries that check if the state is larger than a given value. We will show that the essence of LVars can be modeled in Flo as a special collection type, where threshold queries can be used to safely read from lattice values that are derived from unbounded aggregations.

First, let us define the collection for an LVar. Consider a lattice defined by a set of values L, a bottom value $\bot$, and the lattice join operator $\sqcup$. We will define the LVar collection type a tuple of the lattice value and a boolean flag, where the boolean flag indicates whether the value is *fixed* or not. We will use the lattice join for concatenation, and the $\otimes$ terminator to terminate a collection.

$$\texttt{LVar<L>} \triangleq \{(v, \mathit{true}) \mid v \in L\} \cup \{(v, \mathit{false}) \mid v \in L\}$$
$$(v_1, \mathit{false}) +\!\!+ v_2 = (v_1 \sqcup v_2, \mathit{false})$$
$$(v_1, \mathit{true}) +\!\!+ v_2 = (v_1, \mathit{true})$$
$$(v_1, \mathit{false}) +\!\!+ \otimes = (v_1, \mathit{true})$$

Fig. 16. Type semantics and concatenation operator for LVars in Flo.

There are many operators that can produce an LVar from various input collection types. Let us use ordered sequences as an example. We can define a `fold_lattice` operator which transforms each value into a lattice and then applies the lattice join across the sequence. We define the type and operational semantics for this operator in Figure 17.

FOLD-LATTICE-TYPE
$$\frac{\vdash f : T \to U \qquad U \text{ is a lattice}}{\vdash \text{fold\_lattice}(f) : (\texttt{S<T>}, X) \hookrightarrow (\texttt{LVar<U>}, X)}$$

FOLD-LATTICE
$$\frac{v \neq \otimes \qquad f(v) \Downarrow l}{([..., v], \text{fold\_lattice}(f)) \to^{\delta} ([...], \text{fold\_lattice}(f), l)}$$

FOLD-LATTICE-TERMINATED
$$([\otimes], \text{fold\_lattice}(f)) \to^{\delta} (\otimes, \text{fold\_lattice}(f), \otimes)$$

Fig. 17. Type and operational semantics for the `fold_lattice` operator.

We omit detailed proofs of the core operator properties for brevity here, but note that the boundedness of the output is equal to the boundedness of the input. This is because we can guarantee a terminator on the output when the input will become fixed. In addition, we satisfy eager execution because we always consume elements from the rightmost side, and concatenation to the input can only introduce new elements on the left.

Consider a naive attempt to implement an operator that converts an `LVar<T>` back into an ordered sequence `[T]` by generating a stream containing a single value with that LVar:

$$((v, \_), \text{to\_sequence}) \to^{\delta} (\otimes, \text{to\_sequence}, [v])$$

This operator will be **illegal** because it does not satisfy eager execution. Recall that we are interested in convergence regardless of whether a delta is introduced before or after the step. If we introduce a delta that changes the lattice value, the output sequence would be different depending on this scheduling decision, making the operator non-deterministic. Let's make another attempt to implement this operator, where we wait for the LVar to be fixed first:

$$((v, \mathit{true}), \text{to\_sequence}) \to^{\delta} (\otimes, \text{to\_sequence}, [v])$$

This operator satisfies eager execution, but **now fails** to satisfy streaming progress when instantiated with an unbounded streaming input! If we run the operator on an unfixed input, the output will be an empty sequence. But if we terminate this input, the output will grow to include the lattice value, which is illegal because streaming progress mandates that the only change between these executions should be that the output also becomes fixed, without any changes to its contents.

A fix is to restrict the typing rules for the operator to only accept bounded inputs, so that the input is guaranteed to be eventually fixed.

What can we do with *unbounded* LVars? The fundamental properties of Flo and the original LVars paper come to the same conclusion: we must use a threshold query instead. We can define an operator that takes an LVar and a threshold value, and emits the threshold if the input exceeds it. We list the type and operational semantics for this operator in Figure 18 (omitting partial orders).

LVAR-THRESHOLD-TYPE

$$\frac{\forall i.\ t_i \in U \land \forall i, j\ \nexists k.\ i \neq j \land t_i \sqcup t_j = k}{\vdash_O \text{thresh}(t_1,\ \ldots) : ((\texttt{LVar<U>}, X) \hookrightarrow (\texttt{S<U>}, X), \prec_{\text{thresh}})}$$

LVAR-THRESHOLD

$$\frac{v \sqcup t_i = v}{((v, \_), \text{thresh}(t_1,\ \ldots)) \rightarrow^{\delta} (\otimes, \text{thresh}(t_1,\ \ldots), t_i)}$$

LVAR-THRESHOLD-TERMINATED

$$((v, \mathit{true}), \text{thresh}(\ldots)) \rightarrow^{\delta} (\otimes, \text{thresh}(\ldots), \otimes)$$

Fig. 18. Type and operational semantics for the `threshold` operator.

This operator satisfies **both** eager execution and streaming progress, making it safe to use in a Flo program. The more general properties required for Flo, which do not involve partial orders over collection values or any algebraic properties, still map very precisely to the approach taken in LVars to enable deterministic data processing.

6.3 DBSP

Another point in the streaming language design space comes from the database community. DBSP [13] introduces a formal model for relational operators that can be incrementally executed on live updating databases. A key insight of DBSP is that relations with incremental updates can be modeled as z-sets, where each element in the set has an integer cardinality, such that negative values correspond to retractions of data. We will show that the essence of DBSP can be modeled in Flo by using a special collection type for z-sets, where incremental operations over these correspond to satisfying eager execution.

First, let us define the collection for a z-set in Figure 19. We will define the z-set collection type as a map of keys to integer cardinalities as well as a boolean flag that indicates that the collection is fixed. The concatenation operator simply combines the two maps by adding the cardinalities of matching keys, and the $\otimes$ terminator makes the collection fixed.

$$\text{Cardinality Maps: } M = \{k_1 : v_1, \ldots\} \text{ where } v_i \in \mathbb{Z},\ M[k] = 0 \text{ if } k \notin M$$
$$(M_1 + M_2)[k] = M_1[k] + M_2[k]$$
$$\text{ZSet} = \{(m, \mathit{true}) \mid m \in M\} \cup \{(m, \mathit{false}) \mid m \in M\}$$
$$(M_1, \mathit{false}) +\!\!+ M_2 = \{(M_1 + M_2, \mathit{false})\}$$
$$(M, \_) +\!\!+ \otimes = \{(M, \mathit{true})\}$$

Fig. 19. Collection type and concatenation operator for z-sets in Flo.

In DBSP, inputs to the program are z-sets, and we will take the same approach when mapping this to Flo. Next, we define operators over z-sets. Let us define `map`, a general version of projection, in Figure 20. We omit typing rules for brevity, but the output boundedness is the same as the input.

$$\textsc{map-zset}\quad \frac{f(k_1, v_1) \Downarrow v'}{((\{k_1 : v_1, \ldots\}, \_), \mathrm{map}(f)) \to^{\delta} ((\{\ldots\}, \_), \mathrm{map}(f), (\{k_1 : v'\}, \_))}$$

$$\textsc{map-zset-terminated}\quad ((\{\}, true), \mathrm{map}(f)) \to^{\delta} (\otimes, \mathrm{map}(f), \otimes)$$

Fig. 20. Small-step semantics for the map operator.

This operator trivially satisfies streaming progress, because no outputs are gated on termination. In DBSP, the primary goal is incremental execution: we can introduce additional input and the output will be updated to the result on the full input. This is *exactly* the definition of **eager execution**. Our operators satisfy this property because they are distributive over the z-set. Consider processing a key k_1 with cardinality v_1 only to have it re-introduced by a delta with cardinality v_2. If the delta is applied before the operator, the operator will directly emit a value with cardinality $v_1 + v_2$. If the delta is applied after, cardinality v_1 will be emitted, and later the operator will emit v_2 which will be added together by concatenation.

$$(M_1 \bowtie M_2)[k] = M_1[k] \cdot M_2[k]$$

$$\textsc{join-zset}\quad ((M_1', s_1), (M_2', s_2), \bowtie (M_1, M_2)) \to^{\delta} ((\{\}, s_1), (\{\}, s_2), \bowtie (M_1 + M_1', M_2 + M_2'), (M_1 \bowtie M_2' + M_1' \bowtie M_2 + M_1' \bowtie M_2'))$$

$$\textsc{join-zset-terminated}\quad ((\{\}, true), (\{\}, true), \bowtie (\_, \_)) \to^{\delta} (\otimes, \otimes, \bowtie, \otimes)$$

Fig. 21. Operational semantics for the join operator.

A more interesting operator is the natural join ($\bowtie$), which takes two z-sets and produces a new z-set by joining on a key. First, we define a $\bowtie$ operator on z-sets which joins them by taking the product of cardinalities of matching keys. To perform an incremental join, we store the z-sets which have already been processed in the state of the operator. We can then apply the z-set property $(a + a') \bowtie (b + b') = a \bowtie b + a' \bowtie b + a \bowtie b' + a' \bowtie b'$. We use this in a sketch for the operational semantics in Figure 21 (again, omitting type semantics but using only unbounded streams).

Again, what is interesting here is that proving eager execution *aligns exactly* with the incremental computation goal in DBSP. In DBSP, proofs of correctness hinge on the join operator being bilinear, because $a \cdot (b + c) = a \cdot b + a \cdot c$. This is exactly the property we need to prove eager execution, because the operator must be distributive over concatenations to the z-set. This is a powerful demonstration of the flexibility of Flo, as it can precisely capture the semantics of incremental computation with retractions, a key limitation of approaches like Stream Types [19].

6.4 Putting It Together

What is particularly exciting is that all these case studies fit into the common model of Flo. In fact, we could unify all three into a single language, since the operators are all composable and can be used together. For example, we shared the ordered sequence collection between Flink and LVars, so the operators we defined in both could easily be mixed together to compute a threshold over windowed aggregates. This shows the power of the abstract approach taken by Flo; we can capture a wide range of semantics under one roof, while still providing strong guarantees about the behavior of the system as a whole.

7 Related Work

Flo builds on the vast bodies of work on streaming language design from both the programming languages and databases communities. We leverage the insights across both traditional stream processing and incremental computation to devise a new model for progressive streams.

7.1 Stream Types and Deterministic Dataflow

The most closely related work to Flo recently is Stream Types [19], which provides a rich type system that can precisely capture the structure of elements in a stream. Stream Types focus on capturing ordering invariants, such as the presence of certain elements within bracketing pairs. These fine-grained types make it possible to prove strong semantic guarantees about the *implementation of operators*, such as determinism when operating on prefixes of data.

These properties map well to the eager execution and streaming progress properties of Flo, which takes a more abstract compositional approach to stream semantics. In this way, Stream Types and Flo can be complementary, since Stream Types can be used to prove that operators in a Flo language satisfy the properties required by Flo. Flo's notion of streams, however, is more general than that of Stream Types; indeed, one of the key limitations of Stream Types is that they cannot model incremental computation with retractions, a key feature of DBSP that Flo can capture.

Other work defines streams as monoids [32, 33] and uses monotone operators to ensure determinism. We generalize this approach by relaxing their monotonicity requirements into eager execution, and by relying on a notion of concatenation that generalizes their monoidal structure. This enables Flo to be used to model retractions that the monoidal approach cannot capture.

7.2 Stream Query Languages

"Continuous" query languages over streams have been a topic of recurring interest in database research since the 1990s. A recent tutorial article overviews the history of that work [14], and highlights the foundational influence of CQL [8] on language semantics. CQL extends SQL with operators that map a family of timestamped stream collections (unbounded, in our terminology) to relations (bounded) and vice-versa; SQL is used as an inner language to map relations to relations. CQL assumes a totally ordered, timestepped model of execution in which all data for each timestep is known to be available when that timestep is processed. Like many stream query languages of its time, CQL does not address delay directly: "Our semantics does not dictate 'liveness' of continuous query output—that issue is relegated to latency management in the query processor [10, 16]".

The same tutorial also points out various constructs that stream query languages introduced for *tracking* progress, including punctuations [43], watermarks [3], heartbeats [41], slack [2], and frontiers [34]. While some of these are operational (e.g., timeout-based), many fit our framework in two places: families of collection types that admit reasoning about fixedness (e.g., mixing data and control messages), and language constructs for extracting bounded "inner" collections.

An additional recurring discussion in these systems relates to the practical issue of "late-arriving information" or "out of order processing," in which input values arrive that require a system to "compensate" for or "retract" previously-emitted output values. As illustrated in Section 6.3, recent approaches [13, 34] show how these concerns can be made orthogonal to our discussion here by lifting compensations and their handling into richer collection types and operator algebras.

7.3 Streaming Dataflow Systems

There has been much work on building *performant* streaming dataflow systems, particularly for use in analytical workloads. Systems like Samza [36], Storm [24], Flink [15], Heron [29], Beam [31], and Spark Streaming [45] all provide complete systems for stream dataflow. These systems are

highly performant, and as a result, they focus on the operational aspects of streaming systems, such as fault tolerance, scalability, and low-latency processing. As such, many of the contributions of these systems center on managing persistence of data on distributed nodes and preservation of deterministic outputs in the face of failures, an operational concern that we abstract away in Flo.

More recent work has focused on batching as a way to improve performance [28, 37], which can be modeled in Flo using nested streams. All these approaches, however, generally focus on ordered sequences as a global stream type, rather than allowing programs to mix and match collection types as in Flo. Although Flo is a theoretical foundation, we believe there is much work to be done in building a practical streaming system that can leverage the guarantees provided by Flo.

7.4 Reactive, Incremental, and Stream-Based Programming

Much work exists on functional reactive programming (FRP), a paradigm in which programs are continuously re-run (often incrementally) on ever-changing inputs [18, 25–27, 38]. These programs can be formalized as streams, and are often compiled to a streaming dataflow representation similar to those we explore in this paper. Of particular interest are papers which reason about avoiding space-time leaks [26, 27], requiring a property similar to our streaming progress condition.

Other work in this space has focused on the correspondence between LTL and FRP [18, 25, 38], or have focused on the incrementalization of functional programs [22, 44]. While our work also reasons about properties like equivalence under re-ordering, eventual termination, and avoiding space-time leaks, we choose a new, more general formalism both better-suited to our domain and less opinionated about the definitions of "streams" and "operators."

Many stream-based languages have precise ideas of how to define both streams and computations [9, 12, 17, 35, 42]. While much of this work is interested in properties similar to eager execution and streaming progress, all of it is formalized with a syntax and semantics for a particular language. In contrast, Flo offers an abstract, general framework for streaming languages, with only enough constraints to prove our core properties. We believe that Flo provides a basis to build such languages.

An incremental, streaming language of particular interest is Naiad [34], which uses a dataflow model that supports incremental execution of dataflow with cycles. Our model of nested streams is inspired by Naiad, which similarly uses special operators to describe how streams are fed into out of nested loops. In Flo, our collection type for ordered sequences of streams requires inner streams to be processed in-order, while Naiad allows for "time-travelling" with vector timestamps to allow modifications to already-processed streams. One could imagine implementing this in Flo using a specialized collection type and nesting operator for timestamped messages.

Other work in the streaming space focuses on a similar goal of unifying several streaming semantics under one language [40]. But this work makes limited guarantees about the behavior of the program, with respect to both correctness and liveness of outputs. Flo provides a similar general model, but supports compositional proofs of determinism and completeness of outputs.

8 Conclusion

In this paper, we introduced Flo, a parameterized streaming dataflow language that provides strong guarantees about the behavior of streaming computations. Flo identifies two key properties which are general yet necessary for streaming programs: **streaming progress** and **eager execution**. We formally model these properties and show that they are preserved across composition. Furthermore, we showed that Flo supports nested streams and graphs while maintaining the semantic guarantees of the language. To demonstrate the capabilities of Flo, we showed that Flo can capture a wide range of streaming semantics, from windowed aggregation in Flink, to monotone thresholds in LVars, and even incremental computation in DBSP. We believe that Flo provides a powerful foundation for building streaming systems that can be used to more strongly reason about their guarantees.

Acknowledgments

We thank our anonymous reviewers for their insightful feedback on this paper. This work is supported in part by National Science Foundation CISE Expeditions Award CCF-1730628, IIS-1955488, IIS-2027575, DOE award DE-SC0016260, ARO award W911NF2110339, and ONR award N00014-21-1-2724, and by gifts from Amazon Web Services, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, Scotiabank, and VMware. Shadaj Laddad is supported in part by the NSF Graduate Research Fellowship Program under Grant No. DGE 2146752. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *Cidr*, Vol. 5. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *the VLDB Journal* 12 (2003), 120–139.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[5] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. Citeseer, 249–260.

[6] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15 (2006), 121–142.

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15 (2006), 121–142.

[9] Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.* 4, POPL, Article 22 (dec 2019), 28 pages. https://doi.org/10.1145/3371090

[10] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. 2003. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 253–264.

[11] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *SIGMOD Rec.* 30, 3 (sep 2001), 109–120. https://doi.org/10.1145/603867.603884

[12] Gérard Berry and Laurent Cosserat. 1985. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency: Carnegie-Mellon University Pittsburgh, PA, July 9–11, 1984*. Springer, 389–448.

[13] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (mar 2023), 1601–1614. https://doi.org/10.14778/3587136.3587137

[14] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2651–2658.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[16] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator scheduling in a data stream manager. In *Proceedings 2003 VLDB Conference*. Elsevier, 838–849.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

[18] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair reactive programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/2535838.2535881

[19] Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2024. Stream Types. *Proc. ACM Program. Lang.* 8, PLDI, Article 204 (jun 2024), 25 pages. https://doi.org/10.1145/3656434

[20] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670995

[21] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[22] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 156–166. https://doi.org/10.1145/2594291.2594324

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[24] Ankit Jain. 2017. *Mastering apache storm: Real-time big data streaming using kafka, hbase and redis*. Packt Publishing Ltd.

[25] Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA) *(PLPV '12)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783

[26] Alan Jeffrey. 2014. Functional reactive types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. Association for Computing Machinery, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/2603088.2603106

[27] Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.* 48, 9 (sep 2013), 221–232. https://doi.org/10.1145/2544174.2500588

[28] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. 2019. Arc: an IR for batch and stream programming. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages* (Phoenix, AZ, USA) *(DBPL 2019)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/3315507.3330199

[29] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[30] Lindsey Kuper and Ryan R. Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

[31] Jan Lukavsky. 2022. *Building Big Data Pipelines with Apache Beam: Use a single programming model for both batch and stream data processing*. Packt Publishing Ltd.

[32] Konstantinos Mamouras. 2020. Semantic foundations for deterministic dataflow and stream processing. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings 29*. Springer International Publishing, 394–427.

[33] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language*

Design and Implementation (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 670–685. https://doi.org/10.1145/3314221.3314580

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[35] David Navalho, Sérgio Duarte, Nuno Preguiça, and Marc Shapiro. 2013. Incremental stream processing using computational conflict-free replicated data types. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms* (Prague, Czech Republic) *(CloudDP '13)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/2460756.2460762

[36] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[37] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (may 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[38] Jennifer Paykin, Neelakantan R Krishnaswami, and Steve Zdancewic. 2016. The essence of event-driven programming. *Leibniz, Leibniz International Proceedings in Informatics* (2016).

[39] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1, Article 16 (mar 2024), 27 pages. https://doi.org/10.1145/3639271

[40] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A universal calculus for stream processing languages. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 507–528.

[41] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 263–274.

[42] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 179–196.

[43] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[44] Daniel M. Yellin and Robert E. Strom. 1991. INC: a language for incremental computations. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 211–236. https://doi.org/10.1145/103135.103137

[45] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737