Hydroflow: A Model and Runtime for Distributed Systems Programming

Mingwei Samuel
Joseph M. Hellerstein, Ed.
Alvin Cheung, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-201
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-201.html

August 16, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Hydroflow: A Model and Runtime for Distributed Systems Programming

by Mingwei Samuel

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Alvin Cheung
Research Advisor

8/15/2021

(Date)

* * * * * * *

[illegible]

Professor Joseph M. Hellerstein
Second Reader

8/13/2021

(Date)

Hydroflow: A Model and Runtime for Distributed Systems Programming

Mingwei Samuel
mingwei@berkeley.edu
University of California, Berkeley

ABSTRACT

The cloud gives everyone the power of infinite computing resources, but programming distributed systems is hard. Current programming models require very precise and error-prone reasoning about network reliability and message handling, and frequent expensive coordination stages in which all nodes have to communicate to reach consensus.

In this paper we present our ongoing work on Hydroflow, a new *cloud* programming model used to create constructively correct distributed systems. The model is a refinement and unification of the existing *dataflow* and *reactive* programming models. Like dataflow, Hydroflow is based on an algebra of operators which execute in streaming fashion across multiple nodes. However in Hydroflow data passed between operators can be of any *lattice* type, represented using a compositional lattice sub-language. These features allow us to construct provably *monotonic* distributed programs which can always make forward progress without incurring the high cost of coordination.

Hydroflow is primarily a low-level compilation target for future declarative cloud programming languages, but developers can use it directly to precisely control program execution or fine-tune and debug compiled programs.

1 INTRODUCTION

The cloud presents the abstraction of infinite computational resources, but writing applications that can scale to harness this power is incredibly difficult. Conventional programming models are not well suited for writing general applications on a scaling, distributed platform. Application developers are left stringing together webs of expert-built black-box enterprise services, each with their own separate consistency requirements and guarantees—requirements and guarantees that don't optimize for program semantics and break when combined.

Monotonicity provides a powerful lens on distributed computing. It offers simple and efficient strategies to deal with complex distributed problems, letting us significantly speed up applications. This work builds on the CALM Theorem [12], which shows that all *monotonic* programs have a distributed implementation which is consistent and coordination-free. Anna, a partitioned multi-master key-value store (KVS), runs two orders of magnitude faster than state-of-the-art multiprocessor key-value stores by avoiding coordination using a design based on monotonic state [24].

Our goal is to bring the power of these concepts to general application development, through the Hydroflow programming model. Developers can use Hydroflow to build programs that are *constructively monotonic* and therefore can be safely and automatically distributed and scaled up or down. Hydroflow combines the existing *dataflow* and *reactive* programming models into a unified model that can ensure monotonicity through its use of lattice-based state.

A Hydroflow program specifies a directed graph of operators which are executed on single nodes or partitioned across multiple nodes. The operator algebra is designed so that we can determine many high-level properties by the composition of operators in a graph, most importantly monotonicity. Hydroflow is implemented in the Rust programming language, an increasingly popular choice known for its memory safety and speed [20]. Our implementation uses Rust's type system to represent properties of individual operators, and in turn these are used to ensure correctness and derive higher-level properties as operators are composed into a graph.

Hydroflow serves as the low-level compilation target for the declarative HydroLogic cloud programming language as part of the Hydro Project [6]. Hydroflow is designed so that we will be able to automatically and correctly transform a graph to optimize its performance and scalability during the compilation process. These transformations will include conventional relational algebra-style "logical" transformations as well as "physical" transformations to the ways that operators are distributed and partitioned across nodes.

Ensuring monotonicity is a primary goal of Hydroflow, however some programs require non-monotonicity. Our current implementation is of the main *monotonic* subset of Hydroflow, but in the future we will also provide the ability to write non-monotonic code which may require explicit coordination. This will make Hydroflow fast and safe-by-default while still allowing developers to use coordination when it is truly needed.

The following subsection 1.1 describes how this project relates to existing work. Section 2 reviews the theoretical background of monotonicity and the CALM Theorem, and connects these concepts to existing dataflow and reactive programming models. Section 3 explains how Hydroflow unifies those models and augments them with monotonicity guarantees. Section 4 describes how we actualized these concepts in our initial implementation of Hydroflow. Finally, Section 5 outlines ongoing and future work on this project.

1.1 Related Work

Existing languages meant for programming distributed systems include Bloom, Lasp, Gallifrey, and others. Bloom [2] is a high-level declarative programming language originating from Datalog [3]. Programs in Bloom opreator on unordered collections which can be though of as set or map lattices. Tools bundled with the Bloom interpreter let developers identify monotonic and non-monotonic parts of programs to determine *if* and *where* coordination is needed. Bloom^{L} [7] extends Bloom with compositional lattices-based datastructures in much the same style as we have in Hydroflow. Unlike Bloom, Hydroflow is imperative and low-level, and Bloom is interpreted while Hydroflow is compiled. We hope to eventually use Hydroflow as a compilation target to speed up Bloom-style programs.

CRDTs are distributed data structures used to synchronize particular variables in general programs in an eventually consistent way. *State-based* CRDTs are a sub-category which can often be thought of as semilattices [16, 21]. The Lasp programming language allows developers to compose and link CRDTs in a functional reactive programming-style [16].

Gallifrey [17] extends Java with the ability to replicate objects across nodes in a distributed setting. Gallifrey uses a rich type system to statically checks the correctness of operations at compile-time. Hydroflow also aims to be correct at compile-time and express similar properties through Rust's type system. We also use Gallifrey's *monotonic tests* which are boolean expressions which *trigger* actions once they become true, and afterwards always remain true.

LVars [13] is a programming model implemented as a Haskell library which is also based on lattices. Like Hydroflow, LVars ensures constructive monotonicity by representing state using lattices. However LVars is not meant for distributed systems; it allows consistent use of shared memory by finite-running applications in a single-machine setting. Hydroflow does not use shared memory and is intended for general long-running distributed programs.

Dataflow research builds on a long history of query processing over both stored and streaming data in database systems. This paper argues that dataflow can be framed in a monotonic way which explains its relative simplicity and success in distributed environments. Noteworthy distributed dataflow systems in production include MapReduce [9], Spark [26], and Beam [1]. Much of the existing work in these managed systems focuses on high-level execution details such as membership and error-retrying. Other high-level managed systems include Orleans [5], Dask [14], and Ray [18] provide actor, promise, and/or stream abstractions through their distributed platforms. The current implementation of Hydroflow only focuses on low-level single-node execution, and multi-node execution through explicit communication channels. We expect that monotonicity will simplify many of these high-level concerns when we reach them in the future.

The connections we make between dataflow, reactive programming, and *state* are related to *differential dataflow* [15] (used by Timely [19]) which frames dataflow in way which is incrementally-computable across a distributed system. A related concept is *partially-stateful* dataflow used by the Noria database [10]. These techniques are conventionally used to maintain up-to-date materialized views of database queries rather than represent general distributed applications.

2 THEORY

The CALM Theorem tells us: "a program has a consistent, coordination-free distributed implementation if and only if it is monotonic" [12]. A function $f : S \to T$ is *monotonic* if it preserves a partial ordering of its domain to a (possibly different) partial ordering of its codomain.

$$a \sqsubseteq_S b \quad \Longrightarrow \quad f(a) \sqsubseteq_T f(b) \qquad (\textit{monotonicity}) \; [8]$$

The original statement of the CALM Theorem defines monotonicity using the subset-orderings ($\subseteq$) of sets S and T; our definition is a straightforward generalization of that one. Intuitively, new inputs can never cause a monotonic function to retract previous conclusions, so computations can speed ahead without the need to backtrack or coordinate.

A *join semilattice* consists of a domain of elements and a binary *join* relation defined over those elements, also known as the *least upper bound (supremum)*. A *meet semilattice* consists of a domain and a binary *meet* relation or *greatest lower bound (infimum)*. As these two structures are dually equivalent, we use *lattice* to refer to both, and *merge* to refer to the binary relation.

A lattice merge $\sqcup$ on a domain S is defined as having the following three properties, for all $a, b, c \in S$:

$$\begin{aligned} a \sqcup (b \sqcup c) &= (a \sqcup b) \sqcup c && (\textit{associative}) \\ a \sqcup b &= b \sqcup a && (\textit{commutative}) \\ a \sqcup a &= a && (\textit{idempotent}) \end{aligned} \quad [8]$$

Together we refer to these as the *ACI* properties. These properties induce a partial ordering on S. Given $a, b \in S$, if the merge of a and b results in b then we can say "a precedes b"

or "b dominates a":

$$a \sqsubseteq b \quad \equiv \quad a \sqcup b = b \qquad (\textit{lattice partial order})\ [8]$$

If the merge of a and b results in a new third value then a and b are *incomparable*.

A function $f : S \longrightarrow T$ from lattice domain S to lattice codomain T is a *morphism*[1] if it structurally preserves merges, i.e. for all $a, b \in S$:

$$f(a \sqcup_S b) \quad = \quad f(a) \sqcup_T f(b) \qquad (\textit{morphism})\ [8]$$

The lattice partial order means morphisms are a special subset of monotonic functions over lattices. Importantly, morphisms are *differentially computable*[2]. For example, if we have some state $z := f(a \sqcup b \sqcup c)$ and we want to extend this computation to a fourth value $d \in S$, we can compute $z' := z \sqcup f(d)$ which avoids recomputation on a, b, c. Note that due to the ACI properties of merge $\sqcup$ this computation can process elements in any order and is unaffected by duplicates. Intuitively, this means that a system that represents state using lattices can easily be impervious to message reordering and duplication. It turns out that this framing of computation has direct connections to traditional dataflow programming.

2.1 Dataflow

Dataflow is a programming model where independent functions, called *operators*, are composed into a directed graph. Elements of data "flow" along the edges of the graph and are transformed by each operator. Operators such as *map* and *filter* operate on single elements of data, while *fold*[3] sits at the end of a chain of operators and combines all the arriving elements together into a single value.

This model of programming has a natural connection to lattice morphisms. Instead of thinking in terms of individual elements we can instead think of the *set* of all elements that pass through an operator over time. In this framing, each element operator represents a morphism over the set-union lattice, and each individually arriving element can be thought of as a singleton set, or a small *delta* to the overall set.

In Figure 1 we see a simple example dataflow on the bottom in green, and a *lifted* view of the dataflow in terms of sets above in blue. Given individual inputs y and z the dataflow computes $f(g(y))$ and $f(g(z))$ using element-wise functions f and g. In the lifted framing we view the corresponding functions big F and G as morphisms over sets of elements.

[1]Because both the domain and codomain are semilattice spaces, *semilattice homomorphism* is the most precise term.

[2]Differential computation is a generalization of incremental computation: incremental computation operates on a sequence of *changes* to a particular piece of data, avoiding redundant work but inducing a total order on the data's state. Differential computation works with independent partially ordered updates to the data which is essential in a distributed context [15].

[3]Also called *reduce*, *aggregate*, or *accumulate*.

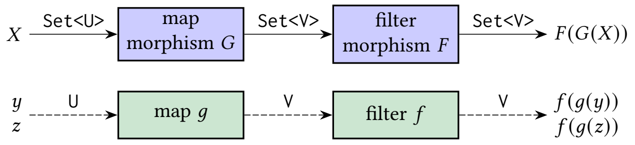


Figure 1: The bottom, in green, represents a simple dataflow program with a map g : U $\longrightarrow$ V and filter f on V. Both U and V are single-element types. The top, in blue, is a lifted view of the dataflow which can be thought of as operating on the *complete set* of U (or V) elements all at once.

If we view the dataflow output as a set as a set then these views are equivalent, where Y and Z are the singleton sets containing y and z respectively:

$$\{\, f(g(y)), f(g(z)) \,\} \;=\; F(G(Y)) \cup F(G(Z)) \;=\; F(G(Y \cup Z))$$

This is a simple example, but we can view many dataflow programs in this way, as a composition of monotonic morphisms over set-union lattices.

However, our morphism-centric perspective can be extended beyond just set-union lattices. We can *fold* the output elements of an operator together using any lattice merge function $\sqcup$ of our choosing, and the merge will be structurally preserved over preceding dataflow operations.

$$F(G(Y)) \sqcup F(G(Z)) \quad = \quad F(G(Y \sqcup Z))$$

This applies to any combinations of lattice types; any morphism can be represented and differentially computed in this streaming fashion.

This modelling only covers unary operators, but binary operators (e.g., joins, unions) which handle multiple input streams are essential in dataflow applications. However the obvious generalization of morphisms $f(a \sqcup b,\ y \sqcup z) = f(a, y) \sqcup f(b, z)$ is not useful; an f which computes Cartesian product, a differentially computable and monotonic operation, fails to meet the above as the right-hand side cannot provide pairs in (a, z) and (b, y). Monotonic modelling of binary operators is covered in section 3.2.

In the context of the CALM Theorem, this modelling provides a mathematical basis to justify why the dataflow model is so successful in distributed computing. Dataflow systems often require (or gain significant performance if) users specify associative and commutative *fold* functions [1, 9]. In this case the operators can be viewed as structure-preserving morphisms and this means the dataflow's execution can be coordination-free and highly scalable.

2.2 Reactive Programming

Reactive programming is a model where chains of operators are re-run when input values change. It is similar to

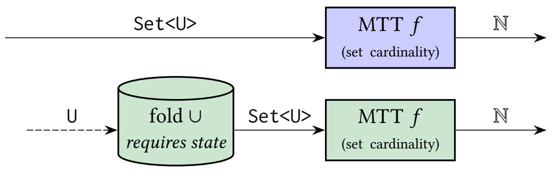


Figure 2: The pipeline before the fold (bottom left, in green) looks like dataflow. After the fold the pipeline looks like reactive programming. The fold does not have an effect on the lifted view (in blue), but after the fold the reactive pipeline and the lifted view match.

the dataflow model in that data moves through a graph of operations but unlike dataflow, computations are not run incrementally on independent elements (deltas) but instead are re-run on complete input values on updates. Traditionally, reactive programming is used to build responsive user interfaces which update in real time to changes in data [4]. However for our purposes it provides a model to handle non-morphism monotonic computations.

We refer to monotone functions that are *not* lattice morphisms as *monotone tricky*, abbreviated *MTT* . They are tricky because they are not differentially computable and do not preserve the ACI properties of lattices. A simple example of an MTT function is set cardinality: given a set, return the number of elements in the set. This is clearly monotonic; the cardinality of the set grows as elements are added to the input. But it is not idempotent: $card(\{a, b\}) + card(\{b, c\}) = 4$, while $card(\{a, b\} \cup \{b, c\}) = 3$. Operationally, we have the problem that if we try to compute set cardinality on the elements passing through a dataflow pipeline one at a time, we have no way to detect duplicated elements and enforce idempotence.

However, if we accumulate the entire set of elements then computing cardinality is trivial. Figure 2 shows this setup: to transition from a dataflow of individual elements to a whole set we can *fold* all the elements using set-union (∪). Then to compute cardinality we simply take in the entire set as input and count the elements it contains. When the set grows, we once again take in the entire *updated* set as input and once again count elements. This is *reactive programming*: we operate on the whole value (in our example, a set) repeatedly.

3 THE HYDROFLOW MODEL

In this section we describe the Hydroflow model, and how it builds upon dataflow and reactive programming. Like other streaming paradigms, Hydroflow is modelled as a directed graph where each vertex is an operator doing computation and data flows along the edges between operators. In the monotonic subset of Hydroflow, each edge has a specific lattice type and elements flowing through the edge are instances of that lattice.

The overall graph may be partitioned into subgraphs which are run on separate *nodes* or groups of nodes. A node corresponds to a single CPU thread, and nodes may be distributed between threads within a process, between multiple processes, or between multiple machines. To cross node boundaries, elements are sent through inter-thread or IPC channels, or across the network. We avoid using shared memory as it significantly inhibits multi-core scalability under contention [24].

Like with dataflow and reactive programming in Figures 1 and 2, all data that flows through an edge represents a single lifted value. Unlike dataflow, in Hydroflow the lattice type of the lifted value matches the lattice type of the edge. This is more powerful than dataflow: dataflow uses single elements which can be though of as singleton sets, whereas Hydroflow allows any lattice types. For the set-union lattice, elements can be singletons but also can be sets containing multiple items, which is a natural way to mathematically represent "batches" of data. Elements could also be max-int lattice points representing counters, or map-union lattice points representing data in a key-value store. The specific lattice types currently provided by Hydroflow are listed later in section 4.1.

3.1 Delta and Cumulative Edges

Edges in a Hydroflow graph are marked as either *delta* edges or *cumulative* edges. These types denote certain semantics and requirements for the sequence of lattice elements which flow along that edge. An operator is capable of outputting a stream of delta elements, cumulative elements, *or both* (which can be modelled as two output edges).

Semantically, a *delta* edge represents a stream of updates which ideally are small (in a physical representation sense) and non-redundant. However delta edges have no requirements for correctness other than that elements must be instances of the edge's lattice type; any stream of lattice elements is valid in a delta edge. Just as individual elements flowing through a dataflow graph can be seen in a lifted view as singletons representing a larger set (section 2.1), individual lattice elements flowing through delta edges are small lattice values which when merged together represent a larger lattice value.

To assemble the "whole" lattice value from a delta edge, we can merge $\sqcup$ together a stream of delta elements: when a new element arrives, we merge it into the existing state and produce a new cumulative value. These sequences of values are what cumulative edges carry. In Hydroflow a special fold operator called a `StateMergeOp` does exactly this; it takes in a delta edge, merges together each element successively, then outputs a cumulative edge which provides a view of the sequence of merged values to downstream operators.These cumulativevalues are "whole" in the sense that they are the cumulative merge of a history of delta values up to the current moment. Cumulative edges correspond to reactive programming as described in Section 2.2.

Example 1. *Consider a simple single-node key-value store. We use a* map-union lattice *to represent the KVS's state in a* `StateMergeOp`*. A map-union lattice is a table from a growing set of static keys to growing* lattice *values. To merge two map-union lattices, we union the entries and resolve conflicting keys by marging their corresponding values (section 4.1). Write operations are lattices to be merged into the* `StateMergeOp`*; each write is represented by a single-item map-union lattice element* `(key, value)`*. In this example the input stream of writes is a delta edge; each element is a small update to the overall state. Meanwhile the output of the* `StateMergeOp` *is a cumulative edge which provides all the stored keys and values.*

As they represent "whole" values, we can do more with a cumulative edge than with a delta edge. An operator which computes a monotone tricky function such as cardinality requires a cumulative edge as input. As the simple KVS in Example 1 does not support deleting keys, it would be fine to compute the cardinality (number of keys) in the KVS using the view provided by the cumulative edge, however it would of course be wrong to count keys by looking at individual writes on the delta input. This may seem obvious, but both the cumulative overall state and the individual delta writes are represented using the same map-union lattice type, so it's important to keep track of this distinction.

The uniformity between cumulative and delta edge elements means that the same mechanisms can handle both. A morphism can be computed in a streaming dataflow fashion on a delta edge, or in a reactive fashion over a cumulative edge.

From a type theory perspective, cumulative edges can be thought of as a subtype of delta edges: it would always be *correct*, but not generally *efficient*, to treat a cumulative edge as a delta edge. The reverse it not true, as cumulative edges have an important correctness requirement: each successive element must dominate the previous. A sequence of elements from a cumulative edge have a total ordering which follows a path through the lattice partial ordering: for a sequence of lattice elements $X_1, X_2, X_3, \ldots$ we have that $i < j$ implies $X_i \sqsubseteq X_j$.

Delta edges do not have any requirements on their lattice elements, but we have brought up the notion of "efficiency" of delta elements. Practically speaking, we of course want to avoid transmitting redundant information along edges, so a sequence of delta elements is efficient if the elements are small and contain minimal redundant information.

Example 2. *As an expository example for delta efficiency, we compare two ways to represent a set-union lattice* $\{3, 5, 8, 9\}$ *using delta values. A delta sequence* $\{3\}, \{8\}, \{5, 9\}$ *is efficient—the sets contains no redundant intersecting elements. But a sequence* $\{8\}, \{3, 5, 8\}, \{3, 8, 9\}, \{5, 9\}$ *which results in the exact same overall set is clearly inefficient—it contains redundant elements and data. In fact, the last value* $\{5, 9\}$ *is entirely redundant as it contains no new elements.*

Not all lattice types have efficient delta representations. Set-union lattices clearly do as they can be split into small disjoint subsets, however max-int lattices do not—the lattice is linear so there is no way to break the update into smaller pieces (representing the update as +1 would not use $\sqcup$ and wouldn't be idempotent).

Example 3. *Consider a set-union lattice representing elements 1 to 100. If we generate those elements in order, it's clearly cheap to to transmit 100 singleton integer sets* $\{1\}, \{2\}, \{3\}, \cdots$ *as deltas than to transmit the growing cumulative sets* $\{1\}, \{1, 2\}, \{1, 2, 3\}, \cdots$*. In contrast, consider a max-int lattice which grows from 1 to 100. To transmit this we send* $1, 2, 3, \cdots$ *as deltas. However, this is exactly identical to the cumulative representation—the deltas are no smaller and in fact are identical.*

Some composed lattices, such as a map-union lattice of max-int values, fall into a middle ground where some aspects of delta values can be broken up and some cannot. In general, the "compactness" of delta values depends on the specifics of the lattice and its representation in memory. Note that because lattice merges are idempotent, this notion of "efficiency" doesn't matter for correctness but is a practical concern.

As mentioned we can turn a delta edge into a cumulative edge with a `StateMergeOp`, but we can also use the stored state to remove redundant data from the delta elements, making them more efficient. Given a incoming element δx and existing lattice state x, we can find the smallest (in a practical memory-usage sense) element $\delta x'$ such that:

$$x \sqcup \delta x \;=\; x \sqcup \delta x'$$

and output these compact $\delta x'$ values as a new delta edge. For example in a set-union lattice we can use *set difference* to remove redundant (i.e. previously-seen) members in a set:

$$\delta x' \;\coloneqq\; \delta x \setminus x$$

This means a `StateMergeOp` has both a cumulative output edge and and a delta output edge, as shown in Figure 3. The `StateMergeOp` enforces that these two edges are in sync: a supplied cumulative element dominates all past delta elements. This is used to ensure new operators will not miss any elements if dynamically added, described in section 3.3.

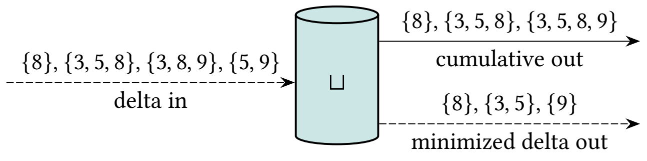


Figure 3: An example of how a `StateMergeOp` works using the set-union delta sequence from Example 2. The operator receives the set-union elements on the left, in listed order. The cumulative out edge provides the entire set of elements with each update. The delta out edge outputs elements that have been minimized to remove redundant elements. Note that the final input element $\{5, 9\}$ has no effect on the outputs since it is entirely dominated.

We have a few options to convert a cumulative edge back into a delta edge. First of all a cumulative edge can be directly viewed per-element as a delta edge, but this is clearly inefficient as each element will contain a lot of redundant information. However we can use the a `StateMergeOp` just like above to minimize redundant parts of each element, producing an efficient delta output stream.

Note that cumulative edges require order guarantees as each successive element must be dominate the previous. An unreliable network could violate this requirement by reordering elements. Conventional reactive programs either run on a single node or experience momentary "glitched" views of inconsistent data [4]. In Hydroflow we limit cumulative elements to single nodes: cumulative edges are not allowed to cross node boundaries . If we want to send a cumulative edge across nodes we need to use a `StateMergeOp` to convert it into deltas, send those over the boundary, then use another `StateMergeOp` to re-merge them together on the other side (in any order!).

3.2 Binary Operators and Joins

Binary operators like *join* are essential for expressing most applications. Database and dataflow systems usually use a handful of monolithic join operators that cannot be decomposed into smaller components. Because Hydroflow deals with general lattice types rather than just sets of tuples, we require more expressive binary operators. In this section we outline how Hydroflow can compute general monotonic binary functions.

We call a binary function $f : R \times S \longrightarrow T$ monotonic if it is monotonic in each of its inputs. This is equivalent to a unary monotonic function over the product domain of R and S. For $a, b \in R$ and $x, y \in S$:

$$a \sqsubseteq_R b, \quad x \sqsubseteq_S y \quad \Longrightarrow \quad f(a,x) \sqsubseteq_T f(b,y) \tag{\textit{monotonic binary function}}$$

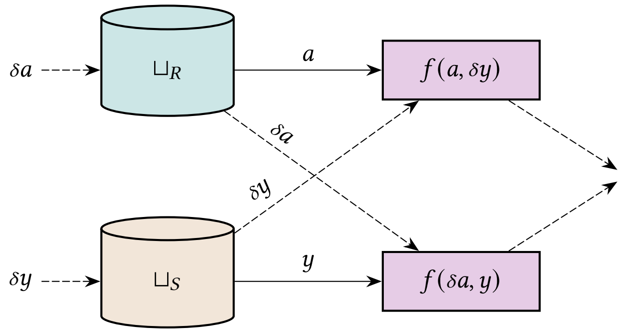


Figure 4: Differential computation of a *split binary morphism*. The two cylinders (teal and brown) are `StateMergeOps` which receive delta inputs and produce both delta and cumulative outputs. The two f nodes (violet) are identical halves of the same binary operator; they are split to reveal the two delta paths and to show that the structure is identical to symmetric hash join.

A simple way to evaluate a monotonic binary function is by taking both inputs through cumulative edges. Each time either input is updated, the function completely recomputes its output. This is correct for all monotonic binary functions but is inefficient for collection-like lattices.

We define another sub-class of monotonic binary functions which are efficiently computable. This class include joins and join-like functions. We call a function $f : R \times S \longrightarrow T$ a *split binary morphism* if it is a morphism in each of its arguments. For lattices $a,\ \delta a \in R$ and $y,\ \delta y \in S$:

$$\begin{aligned} f(a \sqcup_R \delta a,\ y) &= f(a, y) \sqcup_T f(\delta a, y) \\ f(a,\ y \sqcup_S \delta y) &= f(a, y) \sqcup_T f(a, \delta y) \end{aligned} \tag{\textit{split binary morphism}}$$

This construction is a differentially computable monotonic binary function. a and y are cumulative values, and $f(a, y)$ is the previously computed output. When a δa delta arrives, we can output $f(\delta a, y)$ as a delta, and symmetrically when a δy delta arrives, we output $f(a, \delta y)$.

As they require both cumulative and delta input edges for both arguments, split binary morphism will always be somewhere after `StateMergeOps`. Operators which preserve both cumulative and delta edges such as morphisms can exist between the `StateMergeOp` and split binary morphism. If R and S are collection-like lattices, then a join can be expressed simply with a function f which computes the join between each pair of lattice element instance as they arrive. In fact, this creates a structure identical to symmetric hash join, a fundamental join algorithm for highly-pipelined streaming joins [23, 25], as shown in Figure 4.

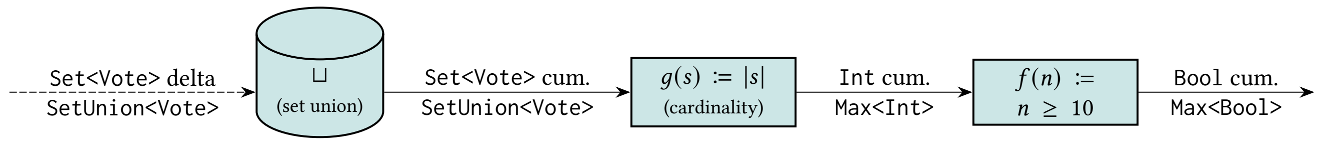


Figure 5: A chain of operators which counts votes. Each edge is labelled with its domain, edge type (cumulative or delta), and lattice type (as listed in Table 1). Once ten votes are counted, the final boolean lattice predicate becomes true, which may trigger other actions downstream. If this edge is followed by a dynamic split points we can mark it with a ⊤-stone. Otherwise we can remove the operators in reverse order, back to the nearest split.

3.3 Dynamic Graphs

For some applications we will need Hydroflow to express dynamic graphs. We can extend a Hydroflow graph by adding either new ingresses which send data to the graph, or new egresses which read data from the graph (or both). The former, sending additional data through new edges to the graph, is correct and monotonic in general. All elements are lattices so the merging of new data will always result in monotonic growth following the lattice order (or no change). As for the later, cumulative edges are the key to dynamic extension which receive data from the graph. Each element from a cumulative edge encompasses all past history which lets us "replay history" for a newly added operator. This means that new operators must be attached at some point after a `StateMergeOp`. Once attached, the new operator immediately receives past history via a cumulative element, then can continue to receive delta and/or cumulative elements as needed. Note that Hydroflow does not allow arbitrary attachment points for extensions, we instead mark specific edges as allowing them.

Removing operators from the graph breaks monotonicity in the general case, however there are specific situations where it *is* okay because an edge can no long have an effect on the system. Lattice types may have a final *top element*, denoted $\top$, which is a value that dominates all other values: $x \sqsubseteq \top$ for all x in the lattice domain [8]. For example, a `Max<Bool>` lattice can only switch from `false` to `true`, so `true` is $\top$. We can also augment other lattices types with a top, for example we can define a ceiling for a max-int lattice, or a finite domain for a set-union lattice. Once $\top$ is transmitted through an edge no new elements can affect its output. In this situation we can do a couple things depending on the graph's dynamic split points.

If a dynamic split point follows the edge then we must remember that the edge has reached $\top$ for any future extensions. To optimize for this case we can mark the edge with a single $\top$ bit, and then can remove any lattice state and mechanisms for handling updates. We call this technique $\top$-stoning ("top-stoning") since it is similar to using tombstones to delete values [12, 16, 21].

Furthermore, if there aren't dynamic split points following the $\top$-ed edge, we can completely remove that edge. Transitively, each preceding edge can also be removed as long as there are no other (dynamic or static) split points branching off. An example is shown in Figure 5.

3.4 Other Aspects of Hydroflow

This subsection covers aspects of the Hydroflow model that are still under development.

3.4.1 Non-Monotonicity. In Hydroflow non-monotonic operators are marked as "tainted." This taint comes from functions that make non-monotonic observations or depend on non-deterministic computations. These functions are clearly labelled for developer visibility and for code analysis tools. Simple rules determine how this taint spreads: each operator that consumes data from a non-monotonic operators is also tainted, creating a non-monotonic subgraph.

Using terminology from Bloom, these non-monotonic areas are unresolved *points of order* [2]. At these points inconsistent behavior arises due to a dependence on order that may not be enforceable in a distributed setting. Developers can deal with these points in several ways. The most straightforward option is to ensure order through explicit coordination. This will have a high cost if the coordination is invoked on each data element on a hot path. Another possibility is to designate a single node or group of coordinated nodes to handle the point of order, though this of course has liveness tradeoffs. A third option (presented in Bloom [2]) is to simply tolerate inconsistency in the style of Helland and Campbell's "memories, guesses and apologies" [11]. This encodes an amount of non-determinism into the application's semantics: operators may make reasonable "guesses" based on incomplete information. If later the guess turns out to be false an "apology" is issued in order to correct the error. It is an open problem to develop language support to guarantee properties of such design patterns.

3.4.2 Cyclic Graphs. Cyclic graphs are used in Datalog and recursive SQL queries, and in some dataflow systems like Timely [19]. There is no need to reach fixed-point if a Hydroflow graph contains a cycle as long as the entire cycle is monotonic. Note that it is also possible to construct monotonic cycles that have unbounded outputs,[4] such as a loop which continually increments a max-int. Currently it is up to the developer to avoid spinning infinitely this way, though in the future we may develop automated tools that can detect these situations

3.4.3 Partitioning and Bundles. Any lattice type X can be partitioned using a map-union lattice; just use X as the nested lattice for the values. We can transform operators in a similar way: take an operator on X and use a separate instance on each key in a map-union lattice. A pipeline created in this way acts as a "bundle" of pipelines, with a "strand" for each key.

Currently our implementation will see a bundle only as a single pipeline, so metadata and $\top$ information isn't tracked separately for each "strand" of the bundle, but in the future it might be useful to make bundles a first-class construct. Bundles may be useful not just for local grouping of data, but also for partitioning *in space*, across multiple nodes.

Interestingly, other lattice types can also be thought of as bundles. A dominating-pair lattice (described in section 4.1) of `(X,Y)` is similar to a "forgetful" map-union lattice of `Map<X,Y>` where keys `X` traverse a lattice order as time passes. In a sense, this is a special type of partitioning—*partitioning through time.* This is an interesting concept, but may or may not prove to be useful.

3.4.4 Garbage collection. Garbage collection is a major issue in distributed systems that cannot be brushed under the rug. Some monotonic mechanisms such as $\top$-stoning and dominating-pair lattices can be thought of as limited types of garbage collection, however in general garbage collection is non-monotonic and will require coordination.

Unlike coordination for explicit non-monotonicity, we can probably coordinate garbage collection in the background rather than on a hot data path and avoid a lot of its costs. This would allow monotonic operators to continue running while coordination for garbage-collection is happening. However actually implementing this is future work.

4 IMPLEMENTATION

HYDROFLOW is implemented as a Rust library[5] and provides abstractions for lattice composition, function types (including monotonic functions, unary morphisms, and split binary morphisms), operator construction, and graph execution. The implementation makes heavy use of Rust's type system to represent abstract properties of operators and ensure their correct composition.

[4]In Datalog this is known as an "unsafe rule."

[5]HYDROFLOW is available under the Apache 2 license at https://github.com/hydro-project/hydroflow/.

Table 1: Lattice Merge Functions

Merge	Domain	Merge
`Max`, `Min`	Totally-ordered domains, `Ord`.	Picks the larger, smaller element.
`Union`, `Intersect`	Collections.	Unions, intersects sets.
`MapUnion`*	Maps and pair collections.	Unions keys, sub-merges intersecting values.
`Pair`*†	Lattice pair tuples `(X,Y)`.	Sub-merges both `X` and `Y`.
`DominatingPair`*‡	Lattice pair tuples `(X,Y)`.	Picks the pair whose `X` dominates the other's. Otherwise sub-merges both `X` and `Y`.

*Higher-order lattices compose with one or more sub-lattice types.
†Also known as *coordinatewise order* [8].
‡`PairLattice` in ANNA [24], also known as *lexicographic order* [8].

4.1 Lattices

HYDROFLOW provides a library of composable lattice types, listed in Table 1. `Min` and `Max` express totally-ordered lattices, using Rust's standard `Ord` trait. Set-`Union` and `Intersection` provide monotonically growing and shrinking collections on static domains. The remaining lattice types are higher-order and compose with sub-lattice types. `MapUnion` provides a table from a growing set of static keys to growing *lattice* values. When we merge two `MapUnion` lattices we resolve any "conflicting" keys by simply merging their corresponding values.[6] A `Pair` lattice is a simple product pairing of two sub-lattices. Finally, a `DominatingPair` lattice is a pairing of two sub-lattices where the first can be though of as a "version" for the second. When merging if one version dominates the other then that element is picked. Otherwise if the two versions "conflict," either because they are equal or because they are incomparable, then both sub-lattices are merged.

Importantly, lattice types are not used directly as their own ad-hoc datastructures, but instead as flexible labels which can be paired with an underlying physical representation. This separation of *lattice type* and *physical representation* allows HYDROFLOW to represent both delta and cumulative collection-like lattice elements with no overhead relative to conventional dataflow. As listed in Table 2, the physical representation of a `Union<T>` lattice could be an actual set type

[6]Note that this is *not* equivalent to a set-union lattice of `(K,V)` pairs, as the set does not merge the values of intersecting keys

Table 2: Representations for Collection-like Lattices

Set<T>	Map<K,V>	Description
HashSet<T>	HashMap<K,V>	Any size; requires hashable keys.
BTreeSet<T>	BTreeMap<K,V>	Any size; requires totally-ordered keys.
Vec<T>	Vec<K,V>	Any size vector; a batch.
[T; N]	[(K,V); N]	Size N array; a batch.*
Single<T>	Single<(K,V)>	Size one; single item.
Option<T>	Option<(K,V)>	Size one or zero; filtered item.

Library-provided collection representations. Developers can also specify custom representations if needed.
*A bit-masked version of fixed-sized arrays is also provided but not listed.

such as HashSet<T> or BTreeSet<T>, or could be a batch represented via a low-overhead collection like Vec<T> or fixed-size array [T; N], or could be single elements via Single<T> or Option<T>. We have a lot of power to tailor a lattice's representation to our needs; for example, Single<T>s can be used as deltas to match dataflow's execution model of single elements streaming through operators, or fixed-size arrays can be used as explicit batches. Fixed-size arrays can also be used with SIMD or tensor processors to speed up compute-heavy workloads. Developers may specify custom representations if they need even more control, but are responsible for the correctness of their implementation.

LatticeRepr types combine a lattice type with a representation. LatticeRepr is a trait which provides ::Lattice and ::Repr associated types which can be used to get the lattice and representation types respectively. This arrangement allows us to compose lattice types and their representations in a natural structure which can be validated by the Rust type-checker. HYDROFLOW includes LatticeRepr types for each lattice type, and uses special TAG types as names to denote the underlying collection representation. These features are shown in Figure 6.

When provided to operators and user-defined functions, lattice element representations are wrapped in the Hide struct. This struct contains a direct representation of the underlying item, so it has no overhead, but tags the item as either an delta or cumulative at compile-time. Hide blocks access to non-monotonic properties and functions on the underlying value, as well as monotone tricky functions if the value is not marked as cumulative. For monotonic functions, Hide provides a modified version of the function such that the return type is also wrapped with Hide.

```
1  type KvsLatRepr = MapUnionRepr<TAG::HASH_MAP,
2      String,
3      DominatingPairRepr<MaxRepr<u32>, MaxRepr<String>>
4  >
5
6  // KvsLatRepr::Lattice
7  MapUnion<
8    String,
9    DominatingPair<Max<u32>, Max<String>>
10 >
11
12 // KvsLatRepr::Repr
13 HashMap<String, ( u32, String )>
```

Figure 6: An example of a composed lattice type for a hashmap-based key-value store with string keys and integer-timestamped string values, where newer values overwrite older ones. Lines 1-4 show how a lattice and representation are defined together in a LatticeRepr type. Lines 7-10 shows the derived lattice type and line 13 shows the derived physical representation.

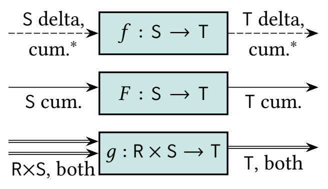
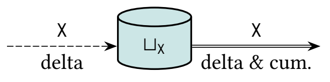
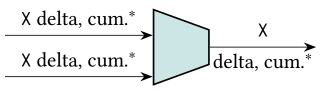
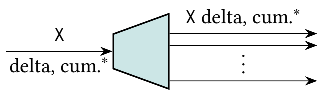
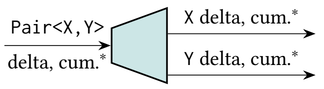
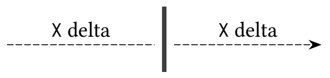
```
1  #[repr(transparent)]
2  pub struct Hide<Y: Qualifier, Lr: LatticeRepr>
3  {
4      value: Lr::Repr,
5  }
6
7  Hide<Cumulative, KvsLatRepr>
8  Hide<Delta, KvsLatRepr>
```

Figure 7: A simplified version of the Hide struct (lines 1-5). #[repr(transparent)] ensures the struct's layout exactly matches the underlying value's, so conversion is free. The Qualifier generic parameter tags an element with Cumulative or Delta. Lines 7 and 8 show how Hide is used with the lattice type from Figure 6.

```
pub trait Merge<Other: LatticeRepr>:
    LatticeRepr<Lattice = Delta::Lattice>
{
    /// Merge `other` into `this` in-place.
    fn merge(
        this: &mut Self::Repr,
        other: Other::Repr);
}
```

Figure 8: A simplified version of the Merge trait. If lattice representation A implements Merge<B> then a B can be merged into an A. The trait bound ensures the two have the same lattice type, but they may have different representations.

Table 3: Topologically Interesting Operators

Description	Diagram
`MorphismOp`, `MonotoneOp`, `SplitBinaryMorphismOp`: User-defined monotonic functions.	 S delta, cum.* $f : \mathsf{S} \rightarrow \mathsf{T}$ T delta, cum.* S cum. $F : \mathsf{S} \rightarrow \mathsf{T}$ T cum. R×S, both $g : \mathsf{R} \times \mathsf{S} \rightarrow \mathsf{T}$ T, both
`StateMergeOp`: Merges deltas, outputs cumulatives and minimized deltas.	
`MergeOp`: Merges two streams into one: interleaves elements and merges cumulatives.	
`SplitOp`: Splits a stream; copies input elements to each output.	
`SwitchOp`: Splits `Pair`s to two outputs without copying.	 Pair<X,Y> delta, cum.* X delta, cum.* Y delta, cum.*
`Channel`, network `Op`s: Sends deltas across thread, network boundaries.	

*These operators can have cumulative output if the input is cumulative.

The `Merge` trait implements a merge function between possibly different representations for a given lattice type. The function is asymmetric: an `other` value is merged in-place into `this`. Naturally, not all lattice representations can be merged into; for example you cannot merge into a `Single<T>`, `Option<T>`, or fixed-size array for a set-union lattice. There are similar traits for computing partial ordering of lattice elements, converting between lattice representations, minimizing delta elements, et cetera. This creates a flexible system where specific properties of lattices can be ensured at compile-time via traits.

4.2 Operators

Operators are the core building block of Hydroflow programs. Operators are another layer of abstraction above morphisms and monotonic functions. Certain operators simply take in user-defined monotonic functions and execute them,

```rust
pub trait Op {
    /// The output element type of this op.
    type LatRepr: LatticeRepr;
}

pub trait OpCumulative: Op {
    fn get_cumulative(&self)
        -> Hide<Lb, Self::LatRepr>;
}

pub trait OpDelta: Op {
    fn poll_delta(&self, waker: Waker)
        -> Poll<Option<Hide<Delta, Self::LatRepr>>>;

    /// Ordering metadata (example).
    type Ordering: Order;
}
```

Figure 9: A simplified version of the `Op` trait. Each implementer defines the possibly-generic associated `::LatRepr` type it outputs, and then separately implements `OpDelta` and/or `OpLb` (for outputting deltas and cumulatives respectively).

but in general operators can do much more than that. Operators can store state, control pipeline timing and execution, split or merge pipelines, or send data across thread and network boundaries. Some of the operators provided by Hydroflow are listed in Table 3.

Internally all Hydroflow operators implement the `Op` trait which is shown in Figure 9. Alone, the `Op` trait only defines an associated `::LatRepr` type to specify the operator's output lattice type and its representation. Usually this is not a single specific type but one defined through generic parameters constrained by trait bounds. `Op`s are then refined further: those which output delta values implement `OpDelta`, and those which output cumulative values implement `OpCumulative`. An operator can implement both or either, possibly conditionally based on trait bounds. The `Op` traits themselves do not define operator inputs, instead operators take ownership of upstream operator(s) and internally call the upstream operator's functions. In this structure each operator owns the previous operator, and therefore transitively owns the whole previous pipeline back to a node boundary or dynamic split point. This allows the Rust compiler to optimize pipelines of operators as single units through monomorphization—the specialization of polymorphic definitions to a specific type [22].

The `OpCumulative` trait requires a simple `get_cumulative` method which synchronously returns a cumulative value. This method usually involves propagating the call backwards to the nearest `LatticeStateOp` and then each operator makes any changes to the cumulative value before returning it back to the caller. This means that when new operators are added to the dataflow graph, we can immediately and synchronously invoke this method to replay all missed history and send it to the new operator without waiting.

The `OpDelta` trait is a little more complex, and is structured to work with streams from Rust's asynchronous framework. `poll_delta` returns a `Poll` algebraic data type which can be either `Ready(value)` or `Pending`. This setup gives Hydroflow easy user-thread scheduling and non-blocking execution. An async runtime keeps a list of all these *tasks* and polls them in a loop. To prevent unnecessary polling, a `Waker` is used to signal to the runtime when a task is ready to be polled. We currently use Tokio[7] as our async runtime and scheduler.

Metadata is represented using tag types associated with each operators. Figure 9 lists an `Ordering` associated type as an example of this; a marker type would be used to denote a particular ordering (or lack thereof) of delta elements, and this ordering can be checked when operators are composed. For example a `ZipOp` which pairs deltas from two operators together would require those operators have the same ordering (among other things) to ensure determinism. We expect to represent many different types of metadata in this way.

5 ONGOING & FUTURE WORK

Hydroflow is very much a work-in-progress. As a driving use case we are implementing a key-value store using Hydroflow; this work is briefly described in subsection 5.1. Subsection 5.2 describes current work to better implement dynamic graphs in Hydroflow. And finally we describe some practical improvements to Hydroflow: a surface syntax in subsection 5.3 and tooling in subsection 5.4.

5.1 Key-Value Store Prototype

We are implementing a basic key-value store in the style of Anna to test the expressiveness and speed of Hydroflow. Anna's state is phrased in a monotonic way at a high-level, but it does not directly ensure monotonicity in its C++ implementation. Additionally Anna does not attempt to model clients of the KVS as monotonic. In Hydroflow we would like to ensure as much of a system is monotonic as possible, however modelling one-off *reads* as monotonic has turned out to be very tricky.

A KVS can be modelled as taking in a stream of requests, splitting those into reads and writes, and then simply joining those streams together as shown in Figure 10. In this case, the write tuples are `(key, value)` pairs and the read tuples are `(key, reader)` pairs where `reader` identifies the requester of the read; these two streams are joined on `key` as shown in Figure 10. An equivalent design would be to

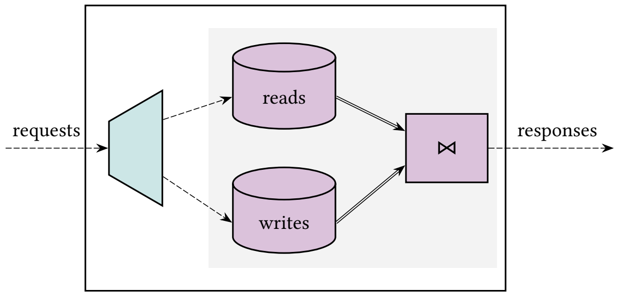


Figure 10: A key-value store expressed as a join between reads and writes through time. Requests are split into reads and writes, then stored in hash tables. Then each read request is joined with the stored writes using the data key. The highlighted violet operators form a symmetric hash join as in Figure 4.

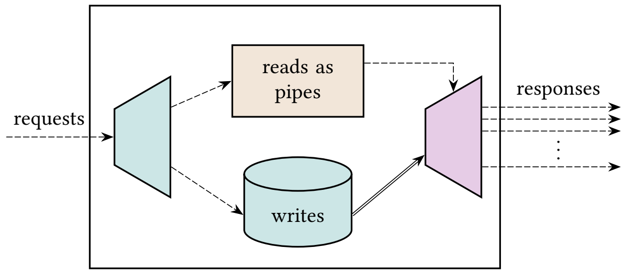


Figure 11: A key-value store expressed using "pipes in pipes." Each read is converted into a dynamic pipe which is then attached to the output of the write storage. This is done by the violet `DynamicSplitOp`. Each new pipe projects the write state to its read key and sends it to the requester.

create new response pipes for each read and dynamically connect those pipes to the writes as shown in Figure 11. Each new pipe probes the write cumulative view for a read key and then sends the value to the requester. However both of these designs have the same design flaw: these "reads" are actually subscriptions rather than just one-time reads, so this is effectively a pub-sub system rather than a key-value store. The join will continue outputting updated values after the first, and the new pipes will continue to carry updates as well.

The root cause of this problem is that one-off reads of monotonically-growing state are simply not monotonic. The state is growing through time, and when a read arrives it intercepts that growing state at an arbitrary point. Any network delays or reordering of reads with writes will cause a different value to be read, creating non-deterministic and therefore non-monotonic behavior. Although this problem shows that expressing programs in Hydroflow can be tricky,

[7]https://github.com/tokio-rs/tokio

it also shows how Hydroflow can ensure monotonicity in the face of developer error.

We are pretty close to finding a way to express one-off reads in Hydroflow, and the solution will probably involve clever use of $\top$ pipes and careful framing of monotonicity. Another option will be to limit monotonicity guarantees down to cover only the KVS state like Anna does,

We will eventually implement other features of Anna such as recovery, key repartitioning, and multi-node scaling. These high-level features will require non-monotonicity and coordination and will provide a driving use-case for Hydroflow.

5.2 Dynamic Graphs

The current implementation works well for static graphs. Operators take full ownership of their predecessors, and the Rust compiler optimizes pipelines of chained operators. However for dynamic graphs, this implementation is not very ergonomic. Dynamic additions naturally cannot take ownership or be optimized at compile time, so we currently have to use separate adaptors to link dynamic pipelines together. Additionally, using the Tokio async runtime is convenient, but creates some tricky cases where we have to wait for elements to finish flowing before we can modify the graph.

To fix this we can add another abstraction layer on top of operators to handle dynamic pipeline segments. This two-tiered approach may allow us to retain the benefits of compile-time optimizations and ownership while handling dynamic segments more easily. The lower layer will act in the same way as current operators.

Meanwhile, the new higher layer can treat each segment from the lower layer as a single unit or edge in a fully dynamic graph. Additionally since dynamic changes in the graph depends on state, we can also handle all `StateMergeOp` state at the higher layer. Similarly, we can replace Tokio runtime by handling asynchronous operators (such as time-based batching) at the higher layer as well.

5.3 Surface Syntax (DSL)

Hydroflow provides a model for cloud programming which is similar to dataflow. However because it is currently implemented as a Rust library, right now a Hydroflow program is simply Rust code. This means developers are forced to learn Rust—which is known for its steep learning curve [20]—in order to assemble Hydroflow graphs and setup the runtime. Additionally, general Rust code is difficult to interpret by automated tooling, and in the future we want to do a lot of static analysis and optimization on Hydroflow graphs.

To solve these problems, we are designing a “surface syntax” DSL as a layer above Rust. This language will be used to specify the important parts of Hydroflow programs, such as lattices, operators, graph topography, dynamic attachment points, useful metadata, and so on. All Rust-specific bootstrapping code is excluded. Tools will be able to analyse these specifications directly and easily. Operators and custom lattices will be able to contain Rust code, and later we can add support for other languages. For now we will use Rust’s procedural macro system to convert surface syntax programs directly into Rust, allowing us to handle Rust code directly without any special considerations.

5.4 Tooling

Custom tooling will be important to make understandable and correct Hydroflow programs. As a compilation target for the Hydro Project, easy-to-use tooling will be essential for debugging and optimizing HydroLogic programs. Hydroflow programs are structured as graphs so naturally we would like to visualize those graphs, similar to what Bloom [2] does. Many properties such as edge metadata, points of order, non-monotonicity, and so-on can be displayed in the generated graphs.

6 CONCLUSION

The cloud provides great opportunity for general distributed applications, but current programming models make reasoning about distributed systems incredibly difficult. Hydroflow uses the lens of monotonicity to provide a simplifying view of distributed computing. In this paper we provide a mathematical model which unifies dataflow and reactive programming. Hydroflow extends this mathematical foundation with compositional lattice types, binary operators, constructive monotonicity, and explicit non-monotonicity.

The current implementation of Hydroflow provides a solid foundation for future work. Our lattice composition system separates lattice types from their representations which lets us directly model lattices in code without compromising on performance. We are continuing work on implementing better dynamic graph handling, coordination mechanisms for non-monotonicity, and data partitioning in space across nodes. In the future we plan to add a custom surface syntax and implement useful tools for developers. Our goal is to make Hydroflow a fully-featured and usable programming framework for the cloud.

ACKNOWLEDGMENTS

My thanks goes to my advisors Prof. Joseph M. Hellerstein and Prof. Alvin Cheung, and to Dr. Mae Milano for their leadership and guidance in this project. And to my friends for supporting me through tough times.

REFERENCES

[1] 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

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. CIDR Conference, Asilomar, CA, USA, 249–260. https://people.ucsc.edu/~palvaro/cidr11.pdf

[3] 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.

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[5] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

[6] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Matthew Milano. 2021. New Directions in Cloud Programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. CIDR Conference, Asilomar, CA, USA. http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf

[7] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

[8] Brian A. Davey and Hilary Priestley. 2002. *Introduction to Lattices and Order.* Cambridge University Press, New York, NY, USA.

[9] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[10] 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

[11] Pat Helland and David Campbell. 2009. Building on Quicksand. CIDR Conference, Asilomar, CA, USA. https://dsf.berkeley.edu/cs286/papers/quicksand-cidr2009.pdf

[12] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM. *Commun. ACM* 63, 9 (Aug. 2020), 72–81. https://doi.org/10.1145/3369736

[13] 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

[14] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 126 – 132. https://doi.org/10.25080/Majora-7b98e3ed-013

[15] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *Proceedings of CIDR 2013*. CIDR Conference, Asilomar, CA, USA.

[16] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data* (Bordeaux, France) *(PaPoC '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/2745947.2745954

[17] Mae Milano, Rolph Recto, Tom Magrino, and A. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming. In *SNAPL*. https://drops.dagstuhl.de/opus/volltexte/2019/10554/pdf/LIPIcs-SNAPL-2019-11.pdf

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[19] 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

[20] Jeffrey M. Perkel. 2020. Why scientists are turning to Rust. *Nature* 588, 7836 (Dec. 2020), 185–186. https://doi.org/10.1038/d41586-020-03382-2

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[22] Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. 2018. Safe Low-level Code Generation in Coq Using Monomorphization and Monadification. *Journal of Information Processing* 26 (2018), 54–72. https://doi.org/10.2197/ipsjjip.26.54

[23] A.N. Wilschut and Peter Apers. 1991. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. 68–77. https://doi.org/10.1109/PDIS.1991.183069

[24] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044

[25] Junyi Xie and Jun Yang. 2007. *A Survey of Join Processing in Data Streams.* Springer US, Boston, MA, 209–236. https://doi.org/10.1007/978-0-387-47534-9_10

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.