Programming Models for Correct and Modular Distributed Systems

Shadaj Laddad

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-85
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-85.html

May 16, 2025

Programming Models for Correct and Modular Distributed Systems

by

Shadaj Ramnivas Laddad

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alvin Cheung, Co-chair
Professor Joseph M. Hellerstein, Co-chair
Heather C. Miller
Matei Zaharia

Spring 2025

Programming Models for Correct and Modular Distributed Systems

Abstract

Programming Models for Correct and Modular Distributed Systems

by

Shadaj Ramnivas Laddad

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alvin Cheung, Co-chair

Professor Joseph M. Hellerstein, Co-chair

Distributed systems are a fundamental part of modern computing, but they are notoriously difficult to program. Developers must reason about a wide variety of non-deterministic behaviors, including message reordering, retries, and failures, all while also having to deal with the inherent concurrency across machines. Existing programming languages focus on local semantics, and provide little help reasoning about the global behavior of the distributed system.

In this dissertation, we present a new foundation for distributed programming that offers richer correctness guarantees and enables new opportunities for modularity. We develop a generalized notion of asynchronous streams that capture distributed semantics in types and restrict downstream behavior to guarantee determinism. Our model makes it possible to write an entire distributed protocol in a single function, encapsulating the network and concurrency. These building blocks can be composed to form complex distributed systems with strong correctness guarantees.

We reify this model in a Rust framework called Hydro. Hydro leverages staged programming to expose a high-level streaming interface while compiling down to bare-metal binaries with performance matching handwritten systems. We then explore optimization techniques for distributed systems enabled by our semantics, including program synthesis and term rewriting. Our work demonstrates that it is possible to build a practical programming model for distributed systems that is both correct and efficient, enabling developers to build more reliable and scalable software.

To my parents, whose love made this dream possible.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation, and my PhD journey as a whole, would not have been possible without the support of my mentors, friends and family. I am grateful to my advisors, Alvin Cheung and Joe Hellerstein, for their guidance and support throughout my PhD. When I began my PhD, I had never taken a databases course, or worked on a formal verification paper. Alvin and Joe took a chance on me, co-advising for the first time, and their mentorship has been invaluable. They provided immense support setting a direction for my work, supported my exploratory efforts outside formal research, and helped me find my voice as an academic.

I would also like to thank my committee members, Heather Miller and Matei Zaharia, for their feedback and guidance as I developed this dissertation. The three of us share common roots in the Scala community, where I first learned about the power of programming languages and type systems. Heather and Matei were role models to me from a very early age—I still remember their presentations on Spores and Spark at Scala Days. I am honored to have them on my committee. I would also like to thank Koushik Sen, my undergraduate advisor, for teaching me operational semantics, helping me write my first research paper, and convincing me to pursue a PhD.

A very special thank you goes to Professor Mae Milano, who started as a postdoc in the Hydro group and became my mentor and friend. It is very clear to me that this PhD would not have happened without Mae's support. Before starting my PhD, I had never written a formal programming languages paper. I was convinced that "there's no way I will write a POPL paper." Mae ignored my doubts and encouraged me to take on the challenge, providing the perfect balance of hands-on guidance and independence. Mae's mentorship has left a lasting impact on my confidence as a researcher, and her future students will be very lucky to have her as a mentor.

During my PhD, I also had the privilege of working with many talented undergraduate students on research projects. I would like to thank Amrita Rajan, David Wei, Nick Jiang, Ryan Alameddine, and Tyler Hou for their contributions to the Hydro project. I strongly believe that Berkeley undergraduates are some of the best in the world, and these students have taught me far more than I could ever teach them. I am excited to see where their careers take them.

I was lucky to call the Sky Computing Lab home for the past four years. Thank you to Audrey Cheng, David Chu, Lily Liu, Mick Kittivorawong, and Neil Giridharan for making the "party cubicle" such a fun place to work. The database group and our weekly dinners have been a constant source of joy and laughter. Thank you to Micah Murray, Jiwon Park, Parth Asawa, and Shreya Shankar for being such great friends. I was lucky to go on many fun trips with the lab to Lake Tahoe, Disneyland, and Las Vegas. Thank you to Altan, Gabriel, Lisa, Manish, Sam, Shishir, Vivian, and the all the lab members for the great memories.

The Hydro group has had a profound impact on my PhD experience. This team has the most talented, hard-working, and kind people I have ever met. Every project in this dissertation resulted from deep collaboration with the entire Hydro group. Mingwei Samuel, the creator of DFIR, built many of the foundations for this dissertation. Lucky Katanahas brought a wealth of practical experience to the team; his insights have been invaluable. Tiemo Bang introduced me to the world of core databases research and prototyped use cases for Hydro. Chris Douglas has been a constant source of inspiration, taking on the most challenging problems in our group.

There are two fellow PhD students in the Hydro group who have had a particularly deep impact on me, both as a researcher and as a person. Conor Power and David Chu are my closest collaborators, and also my closest friends. From complaining about Material Design to playing games of Speedrunners, David made this experience so fun. He supported me through tough times with great advice and amazing meals. I was lucky to co-author several papers with Conor; our late nights working in lab are some of my fondest memories. I have grown so much as a person from spending time with him. Conor and David have been there for me through all the ups and downs, and I aspire to be as kind, caring, and thoughtful as they are. I would also like to thank Conor's wife Laura and David's fiancée Kate for their friendship. Laura and Kate make all the people around them feel special and loved. I've enjoyed our many dinners and trips together, and am grateful for all their life advice through the years.

I would also like to thank Samyu Yagati and Jean-Luc Watson for their friendship. Our Bollywood movie nights are some of my favorite memories from the PhD. Samyu has become like an older sister to me over the years, and I am grateful for her support and excellent advice. Jean-Luc has shown me the importance of balancing research with life outside of work, and how to be a great partner. I am grateful for all our fun times together, and look forward to many more.

I am lucky to have many close friends beyond the lab who have made this journey so enjoyable. Thank you to Rohan, Vikranth, and Bhuvan, who have been my friends since high school and roommates through the first year of my PhD. They have filled the past four years with laughter, joy, and many Fortnite games. They are always there to remind me to take a step back and enjoy life. Rohan and Vikranth are incredibly talented researchers in their own right, and I cannot wait for their dissertations. I am looking forward to many more victory royales together.

Gautam and Shreyas, who I met during my undergraduate years, have been my closest friends since then. While both chose to go to industry rather than pursue a PhD, they have been incredibly supportive and are part of my happiest memories. Shreyas was my roommate during the second year of the PhD, and I am grateful for all the late-night conversations about life, work, and k-dramas. Gautam pushed me to be a better researcher and person, convinced me to stay in the PhD when my research had stalled, and has pushed me to step outside my comfort zone.

I met Sindhu and Snigdha in my freshman year, and they have been my best friends ever since. From garba to movie nights and trips to Napa, they have been constant sources of encouragement and happiness. Sindhu is one of the strongest people I know, and has taught me the importance of being true to myself. Snigdha is a fierce advocate for her friends, and guided me through many difficult times. Along with Rohan, this group of friends has been my rock throughout the PhD. I am grateful for all the Facetime calls, trips, and spontaneous adventures we have had together.

A very special thank you goes to Nikhita Shanker, who has been a constant source of love, support, and happiness as I wrote this dissertation. Nikhita is the most dedicated and sincere person I know. Her passion for doing things *right* always inspires me to hold my work to a higher standard. Nikhita never says "good luck" when I have a big presentation or deadline; she always says "all the best", and reminds me that hard work is what matters. Nikhita makes every day brighter, and I always look forward to spending quality time together. I am grateful for all the evening walks, games of Exploding Kittens, and conversations about every topic under the sun. I cannot wait to take on more adventures together.

Finally, I would like to thank my family for their unwavering love and support. My Papa, Ramnivas Laddad, and Mummy, Kavita Laddad, are my best friends and biggest supporters. From day one, they have showered me with love and encouragement. They have pushed me to be my best self, and I am grateful for all the sacrifices they have made to give me this opportunity. My parents bring joy in the little moments, going to grab samosas after school as a Friday treat or spending time playing with our dog, Iroh. I love them with all my heart, and hope to someday be as good a parent as they are.

It is clear to me that my Papa should have gotten a PhD himself; he is a truly curious person and his intellectual spirit inspires me every day. He taught me the principles of software development that I apply every day, and encouraged the open-source work that shaped my journey to being a researcher. It is not surprising to go on a walk together and end up talking about Rust lifetimes. He has also taught me so much about balancing work and life. He is an incredibly talented musician, practicing every day without fail, and his drive to pursue his passions is inspiring.

My Mummy is the most caring, loving, and hard-working person I know. She has always been there for me whenever I needed her, and I am grateful for all the sacrifices she has made to support my education. Over summers when I was in middle school, she would sit down with me throughout the day and help me with my math homework. Throughout college and the PhD, she has always sent me back to Berkeley with bags full of amazing meals. She set the foundation that enabled me to pursue my dreams, and always encourages me to be my best self.

I am also grateful to my extended family, who have always supported me in every endeavor. Thank you to my grandparents, aunts, uncles, cousins, and everyone else who has been part of this journey. My Naniji always reminds me to eat well and exercise, and gives the best hugs when I visit her. Whenever I give a talk, my Nanaji always asks if I answered the questions correctly; with the signatures on this dissertation, I can finally say yes! I am grateful for all the love and support from my family, and look forward to spending much more time with them.

This work is supported in part by National Science Foundation GRFP Award DGE-2146752, 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 Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Microsoft, NVIDIA, Samsung SDS, SAP, and VMware.

Chapter 1

Introduction

Writing distributed software is *hard*. Distributed systems are the backbone of the modern Internet, involved in every website load, button click, and data transaction. But these systems face complex correctness challenges. Concurrency, network delays, and machine failures all lead to bugs with brutal consequences: data loss, security vulnerabilities, and denial of service.

Distributed systems are hard because they have *implicit non-determinism*. In sequential programs, the order of operations is fixed and sources of non-determinism are explicit: random-number generators, user input, etc. In distributed systems, operations across machines are concurrent and sources of non-determinism are implicit: packet reordering, retries, etc. This makes it difficult to reason about the correctness of distributed systems.

The challenges making distributed systems correct are well-known. Most solutions focus on verification *pre-implementation* [53, 164] and testing *post-implementation* [81]. These tools are valuable in the development process, but we have seen little innovation where developers spend the vast majority of effort: in languages and frameworks for implementing distributed systems.

The most popular model for writing distributed systems is one where the language is not even aware of distribution! Instead, developers string together several single-machine services via a network [54, 125]. Each service is implemented as an isolated program in a language like Java/C++/Rust, which treat the network as an opaque box. This is the root of the problem: the network is beyond the scope of the language, which makes the non-determinism *implicit*.

Just like functions can encapsulate an entire sequential algorithm, could we encapsulate distributed protocols in a way that captures the behavior of the network? This dissertation proposes a programming model that does just that: **stream-choreographic programming**. It features two key innovations: a way to annotate distributed code with **locations** [110, 132] that declare where logic is executed, and a type system that uses **stream markers** to track message interleaving, retries, and other sources of non-determinism.

This dissertation aspires to yield practical artifacts for developers. An explicit non-goal is to create a new programming language–doing so would miss decades of investment into libraries and tooling. Instead, we introduce a *Rust framework* called **Hydro**, which exposes stream-choreographic programming via standard Rust types. Hydro leverages staging [143] to compile an efficient native binary for *each* distributed location, containing only the relevant local logic.

1.1 Correctness for Distributed Systems

Distributed system correctness is not a new problem; there is much research focused on giving developers more confidence in their systems. Languages like TLA+ [164], P [53], and Alloy [73] have been used in industry [116] to model distributed protocols while they are being designed. These languages let developers implement protocols in high-level terms, and then use model checking [39] to verify that the protocol satisfies certain properties. Once a protocol is validated, the developer must then *re-implement* it in a language like C++ or Java. This translation is done by hand, where it is easy to make mistakes. Moreover, real-world systems evolve, which can lead to divergence from the model, and involve side effects such as logging.

Traditional distributed systems tools typically focus on high-level invariants and end-to-end properties. While this remains the north star of verification, it is often impractical due to the high developer overhead. Defining specifications is an excruciating task; a spec that is too weak will fail to catch bugs and a spec that is too strong will generate false bug reports. Rather than dealing with these burdens, most developers opt to develop distributed systems without any formal specifications, and therefore have little assistance when it comes to correctness.

On the other hand, there are many tools for verifying the correctness of distributed systems *after* they have been implemented. In recent years, there has been a surge of interest in *deterministic simulation testing* tools like Jepsen [81] and Antithesis [145]. These tools run a distributed system in a controlled environment where network delays and machine failures can be *deterministically* simulated to catch bugs. Simulation testing is widely applicable to real-world systems, but is expensive to operate and cannot *formally guarantee* any system properties.

This dissertation takes a different approach focused on compositional correctness, inspired by languages like Rust which aim to guarantee safety without sacrificing performance. Rust uses lifetimes to reason about memory management entirely at compile time. In a similar spirit, we develop a type system that can reason about networking and concurrency. Like Rust, we do not aim to offer *behavioral* guarantees on program outputs, but rather provide a strong foundation of safety without developer friction or runtime overhead.

This dissertation identifies *determinism* as a key correctness property that is critical to most distributed systems and often the root cause for bugs. Determinism is well-suited for a language approach because it requires no additional specification work (determinism has a well-known definition) and is compositional. This means that we can use the type system to *locally* reason about correctness and surface errors in a familiar way.

But many distributed protocols appear to have local non-determinism even if the protocol is globally deterministic. For example, a protocol may have a non-deterministic retry mechanism followed by idempotent logic that restores determinism. The insight of this dissertation is that there are several forms of *equivalence* that can be used to reason about distributed logic. For example, the internal messages of a protocol with retries are deterministic *modulo* the retry count, or messages in a MapReduce system are deterministic *modulo* the order of elements. These equivalences make it possible to write many protocols while staying within the type system.

1.2 Stream-Choreographic Programming

This dissertation uses stream programming as a foundation for general-purpose distributed systems, a decision which may be counterintuitive. Most existing frameworks for streaming focus on real-time analytical workloads, such as data pipelines and machine learning. At first, it seems strange to consider streaming as an alternative to RPC frameworks or actor programming.

There are two facets that span most existing streaming frameworks that limit their use for low-level distributed programming. First, most streaming frameworks insist on **monolithic guarantees** for fault tolerance and consistency. For example, RDDs in Spark [165] guarantee that the entire computation yields consistent results even when there are network or machine failures. While this is useful for high-level data pipelines where developers are willing to sacrifice performance for consistency, it is not suitable for low-level distributed programming where developers need to reason about the network and machine failures at a finer granularity.

Second, streaming frameworks provide limited support for **temporal co-incidence**, where a computation can combine atomic snapshots of multiple asynchronous inputs. This is critical for stateful services, where incoming requests represented as a stream must be "joined" with local state. In most streaming frameworks, the dataflow graph is automatically distributed across several machines, which prevents developers from atomically joining streams since they may not be co-located. In traditional RPC or actor-based systems, each unit of compute is placed on a single machine, which gives developers the flexibility to maintain consistent local state.

Our insight is that the streaming model can be extended to encompass a wider range of asynchronous programming patterns with three key ideas:

1. We **generalize streams** beyond ordered sequences to accommodate unordered multi-sets, "singletons" which are mutated in-place, and other asynchronously updated data types. These stream types allow us to *capture* network non-determinism in a way that is zero-cost; there are no additional protocols applied at runtime to enforce consistency.

2. We introduce **locations** to explicitly place streams on specific machines rather than leaving it to runtime. This allows developers to reason more precisely about fault tolerance guarantees and build systems with more predictable performance.

3. We let developers define **iterative nested graphs** that manipulate snapshots of local streams. This construct brings co-incidence to the streaming model, enabling implementation of complex stateful services without having to fall back to imperative programming.

The benefit of the streaming model is that it allows the type system to reason about distributed invariants without requiring proof effort from the developer. Stream programming explicitly captures data dependencies in the program structure, which gives the compiler a "birds-eye" view of the entire system. Furthermore, because streams represent all events over time, the type system can capture temporal invariants that would be implicit in imperative programming models.

1.3 The Hydro Framework

As much as this dissertation introduces new programming models and foundational semantics for reasoning about distributed systems, it is also focused on bringing these ideas to developers through a practical framework. Our goal is to provide a framework that is accessible, flexible, and performant. To this end, our design decisions are guided by three core principles:

1. Use an **existing language** that is widely adopted in industry for distributed systems. This allows us to meet developers where they already are, enables incremental adoption, and lets us leverage existing libraries and tooling.

2. Provide a familiar, **zero-cost interface** for expressing distributed logic. Our framework should use patterns common in the host language, support deployment to standard infrastructure, and never introduce implicit runtime overhead.

3. Offer **escape hatches** when the type system is too restrictive and provide an interface for **program optimization**. The framework should help developers reason about correctness while enabling performance that matches or exceeds handwritten systems.

These principles have led us to **Hydro**, a Rust framework for distributed systems. Rust has gained popularity throughout industry for implementing critical distributed infrastructure that needs strong correctness guarantees, so our distributed correctness goals align well with the Rust community. Hydro is a regular Rust library that can be used with the standard Rust compiler and can be mixed with existing libraries.

Hydro exposes a stream-choreographic interface that lets developers write distributed protocols as straight-line Rust code that spans several distributed locations. Programs written in Hydro can use standard constructs for abstraction; common logic can be extracted into Rust functions and shared interfaces into Rust traits. To ensure these abstractions do not introduce any runtime overhead, Hydro uses **staged programming** [143] to extract the logic of each machine into a separate binary. Staged programming hides this low-level compilation work from the developer, who sees a standard streaming interface similar to Rust iterators.

Hydro lets developers control low-level runtime behavior to achieve maximum performance. Like Rust, Hydro provides an **unsafe** construct that allows developers to opt-out of the type system when they need to reason about correctness manually. This makes Hydro an effective tool to write practical distributed systems, while still providing strong guarantees about correctness. Hydro has been used to implement high-performance distributed protocols, including Paxos and CRDTs, and has been leveraged across several research projects [37, 38].

Finally, Hydro is designed with **optimization** in mind, and exposes a rich internal representation that can be used to analyze and transform distributed programs. Hydro separates distributed structure from low-level logic, making it possible to apply synthesis techniques to search for optimizations with strong correctness guarantees. Hydro also supports fully automated rewrites, which can be used to implement optimizations such as incremental processing. While research in this area is ongoing, Hydro provides a solid foundation for optimizing distributed systems.

1.4 Dissertation Overview

This dissertation is organized into three parts corresponding to fundamental pillars of a programming language stack: formal semantics, language design, and optimization / verification. All together, these pieces form a practical foundation for writing distributed systems that are correct, modular, and performant.

In Part I, we present the theoretical foundations of stream-choreographic programming. Chapter 2 introduces Flo, a compositional metalanguage for asynchronous stream processing that guarantees determinism. Notably, Flo defines local semantics and does not reason about distribution in any form. In Chapter 3, we extend Flo to distributed systems by augmenting the type system with locations that track where each stream is materialized. This extended metalanguage, Gyatso, offers networking APIs that capture various concurrency and fault-tolerance semantics, and introduces new abstractions for scale-out distribution.

In Part II, we introduce the core Hydro framework, which reifies our distributed semantics as a practical Rust API. In Chapter 4, we show how Hydro exposes stream types in Rust and leverages staged programming to compile distributed programs with zero cost. Hydro enables unique opportunities for modularity, since entire distributed protocols can be implemented as a single Rust function. Hydro is powered by a staging mechanism that enables an intuitive interface while preserving zero-cost compilation. In Chapter 5, we step deeper into this technical infrastructure and introduce Stageleft, a Rust library for staged programming.

In Part III, we explore techniques for optimizing distributed programs based on the principles of Hydro. We begin in Chapter 6 with Katara, a system that uses verification techniques to synthesize conflict-free replicated data types (CRDTs), which enable coordination-free replication. In Chapter 7 we explore how to rewrite distributed programs *without* needing a verifier. In particular, we show how e-graphs can automatically derive incremental algorithms by applying a series of local rewrites over Hydro's intermediate representation. Together, these techniques lay out a landscape of optimization opportunities enabled by the Hydro framework.

Part I

Semantic Foundations

The hardest part of programming distributed systems is reasoning about their underlying semantics. Unlike single-threaded programs with clear sequential steps, distributed systems introduce several sources of non-determinism. At the network level, messages may be delayed or reordered. In a cloud computing environment, machines may face unpredictable failures. And across machines, global information such as clock times and cluster membership may not be synchronized. All of these factors can lead to unexpected behavior that is challenging to reason about.

Existing programming models for distributed systems mostly leave this reasoning to the programmer. Actor programming [65, 111, 125] serializes incoming messages into a mailbox, but there are no guarantees on *which* messages will arrive and their order. Models such as durable execution [25, 27] offer stronger guarantees, but come with a performance cost since they use synchronization protocols to simulate a single machine across a distributed system.

A promising direction towards a better distributed programming model is *choreographic programming* [59, 110, 132]. In choreographic languages, developers write sequential routines with each variable and statement tagged to run on a particular *location*. The choreographic program can then be *projected* to each location to derive software that can be run on each machine.

While this has clearer semantics while preserving explicit control over placement, it has major limitations when it comes to scalability. Programs written in choreographic languages execute sequentially, which means that they cannot take advantage of "horizontal scaling" where a set of machines run instances of the same program on different data. Furthermore, they do not support protocols where concurrent requests access shared state. Both of these patterns appear frequently in modern, high-throughput distributed systems.

In the next two chapters, we set up the *semantic foundations* for writing distributed systems using **asynchronous streams**. In Chapter 2, we begin with **Flo**, a metalanguage for asynchronous stream processing. Flo generalizes the notion of a "stream" to encapsulate singleton values, lattices, sets with retractions, and more. Flo guarantees a pair of strong correctness properties for stream programs: **determinism** in the face of asynchronous scheduling and **progress** to yield meaningful outputs when given partial inputs.

In Chapter 3, we then present **Stream-Choreographic Programming**, which extends Flo to distributed systems by letting developers explicitly place streams on choreographic locations. Stream-choreographic programming uses the same underlying dataflow structure as Flo, but introduces specially-designed stream types that capture message interleaving and duplication. Like Flo, stream-choreographic programming provides strong, compositional guarantees for determinism, now extended to a distributed context. This combined model, formalized in the **Gyatso** metalanguage, forms the foundation for the rest of this thesis.

Chapter 2

Flo: Compositional Semantics for Asynchronous Streams

2.1 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 [7], 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 [30], Spark [166], StreamIt [151], and many functional-reactive programming languages [122]. 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 [2, 14, 45, 56]. 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 they do not even agree on what constitutes a stream! They can be ordered sequences [30, 151], or sets [12], or even lattices [90] or Z-Sets [24]. These languages also vary in their semantics for state persistence, and offer a range of interfaces 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 chapter, 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 se-

quences, and supports semantics that are often unsupported [44], 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 [156]) 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 types can layered on underlying collections, such as Stream Types [44], 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 [12] processes batches with a single global loop, while Naiad [112] 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 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 [65], 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 stored in buffers throughout the dataflow graph. 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 [112], 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 streaming patterns, such as windowed aggregations, processing minibatches of a larger collection, 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 [30], LVars [90], and DBSP [24]—and show how existing semantic goals from each are instances of 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 2.3).
- We introduce constructs in Flo for **composing operators** into dataflow graphs and prove that they preserve our key properties (Section 2.4).
- We describe the semantics of **nested streams and graphs** in Flo and demonstrate how they integrate with streaming progress and eager execution (Section 2.5).
- We show how the essence and key capabilities of **existing streaming languages** map to Flo and its foundational properties (Section 2.6).

2.2 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**.

2.2.1 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, this failure case appears 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.

2.2.2 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 [30]. 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.

2.2.3 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 [90]. 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.

2.2.4 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.

2.3 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 [30], 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 [90], Bloom [10], and Temporel [129] 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 2.4, we will define a new family of languages L^G which include mechanisms to compose operators into a dataflow graph. Finally, in Section 2.5, 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.

2.3.1 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 2.1.

$O \leftarrow$ 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

Figure 2.1: 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 [30] and Stream Types [44], 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.

2.3.2 Collection Values, Expressions, and Types

Flo programs manipulate *collections*, which are concrete, finite values used to capture inputs, outputs, and (in Section 2.4) 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 $\varnothing \in C$ is identity on the RHS of $+\!\!+$.

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

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

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

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

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 [44] 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

[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 2.3.3)

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 multi-sets—as in streaming extensions to SQL [18]—or sets, as in Dedalus [12]—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 2.2. We will lay out detailed examples of concrete collection types in Section 2.6.

2.3.3 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 2.3.2), or if it may never become that. To capture this, we use a **boundedness flag** inspired by work in databases [156], 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 2.2. We will see stream types in action in Section 2.3.6.

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

REFLEXIVE-SUBTYPE
$$S \leq S$$

BOUND-SUBTYPE
$$(C, \mathbf{B}) \leq (C, \mathbf{U})$$

Figure 2.2: 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 2.2, where $\leq$ is a subtyping relationship we will use in the rules for composing operators.

2.3.4 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 2.4, 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:

- $L^C = (C, +\!\!+, E^C, T^C, [\![\,]\!]^C, \lfloor\rfloor^C, \text{type}^C, \mathit{fix})$: 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 2.3.1).

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 2.3.1 (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.

2.3.5 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 2.4, 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 2.3.1 (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.

2.3.6 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 2.3.2 (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`) 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 2.3.3 (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)

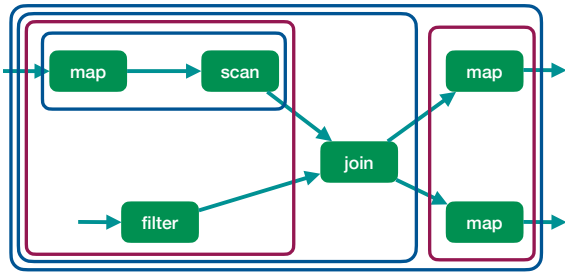
Any operator in an implementation of Flo must satisfy these properties. We will show in the next section that these properties are automatically preserved when composing operators into graphs, which alleviates any further proof burden for the implementation.

2.4 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 2.3.

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 2.4)




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

Figure 2.3: 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.

- $\vdash: e^G : (\tau^S \hookrightarrow \tau^S, \prec)$ 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 2.5)
- $e^g \rightarrow^\Delta (e^g, O)$, a small-step operational semantics where $O \in [C]$ and $e^g \in E^G$ (Figure 2.6)

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 2.4.

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

Figure 2.4: 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 2.5 and small-step operational semantics in Figure 2.6. In our semantics, we will use $\cdot$ to denote tuple concatenation, when dealing with types or values.

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

Lemma 2.4.1 (Graph Preservation). *Given a graph g of type $(I \hookrightarrow O, \prec)$, output state $S = (s_0 \ldots s_n)$, and updated output state $S' = (s'_0 \ldots s'_n)$ such that $O = ((T_0, \_) \ldots (T_n, \_))$ and*

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

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

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

Figure 2.5: Type semantics for graphs of a Flo program.

$\forall i.\ type^C(s_i) = T_i$, 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. □

2.4.1 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 2.3.1).

Lemma 2.4.2 (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 2.3.1.

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

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

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

SEQUENCE-LEFT

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

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

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

Figure 2.6: Small-step semantics for graphs of a Flo program.

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 both will eventually reach a stuck state, and their composition is a stuck state. □

2.4.2 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 2.4.1 (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 2.4.2), 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 2.4.2 (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 2.4.3. *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 2.3.1.

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 2.4.2, 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.

□

2.4.3 Streaming Progress

Lemma 2.4.4 (Streaming Progress for Graphs)**.** *Consider a well-typed graph g with type $\vdash g : (((I_0, B_{I,0}) \dots (I_n, B_{I,n})) \hookrightarrow ((O_0, B_{O,0}) \dots (O_m, B_{O,m})), \prec)$ such that inputs$(g) = i_0 \dots i_n$ and $B_{I,j} = \mathbf{B} \implies$ fixed(i_j). Consider all well-typed outputs O and $o'_0 \dots o'_m$ such that*

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

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

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

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

Base Case: A graph with a single operator. By Definition 2.3.3.

Inductive Step: Proof by cases:

Sequential Composition: We can apply Lemma 2.4.3 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. □

2.5 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.

2.5.1 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 2.7. 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.

$$
\begin{aligned}
[((C_1,B_1), \ldots, (C_m, B_m))] \triangleq \{\ & [(c_{1,1},\ \ldots), \ldots, (c_{n,1},\ \ldots)]\ \mid \\
& \forall i, j\ c_{i,j} \in C_j \land (i > 1 \land B_j = \mathbf{B}) \implies \mathit{fixed}(c_{i,j}) \\
\}\ \cup\ \{\ & [\otimes, (c_{1,1},\ \ldots), \ldots, (c_{n,1},\ \ldots)]\ \mid \\
& \forall i, j\ c_{i,j} \in C_j \land (B_j = \mathbf{B}) \implies \mathit{fixed}(c_{i,j})\}
\end{aligned}
$$

$$
\begin{aligned}
[\otimes, \ldots] +\!\!+ x &= [\otimes, \ldots] \\
[c_1, \ldots, c_n] +\!\!+ \otimes &= [\otimes, c_1, \ldots, c_n] \\
[c_1, \ldots, c_n] +\!\!+ ((v_1, \ldots, v_m), \mathit{true}) &= [(v_1, \ldots, v_n), c_1, \ldots, c_n] \\
[(v_1, \ldots, v_m), \ldots, c_n] +\!\!+ ((\delta_1, \ldots, \delta_m), \mathit{false}) &= [(v_1 +\!\!+ \delta_1, \ldots, v_m +\!\!+ \delta_m), \ldots, c_n]
\end{aligned}
$$

Figure 2.7: 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.

2.5.2 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, 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.

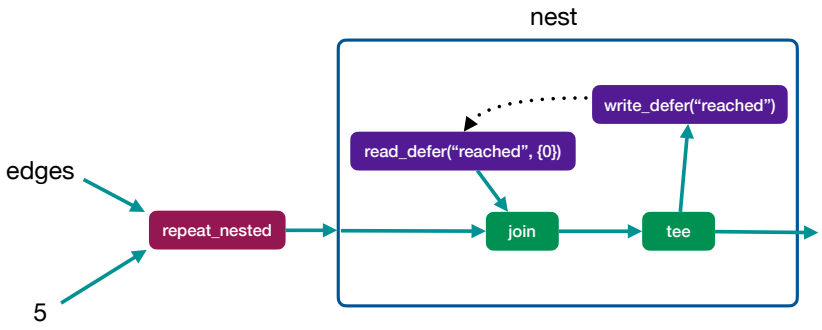


Figure 2.8: An example of identifying nodes within a fixed radius using nested graphs.

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 2.8. 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.

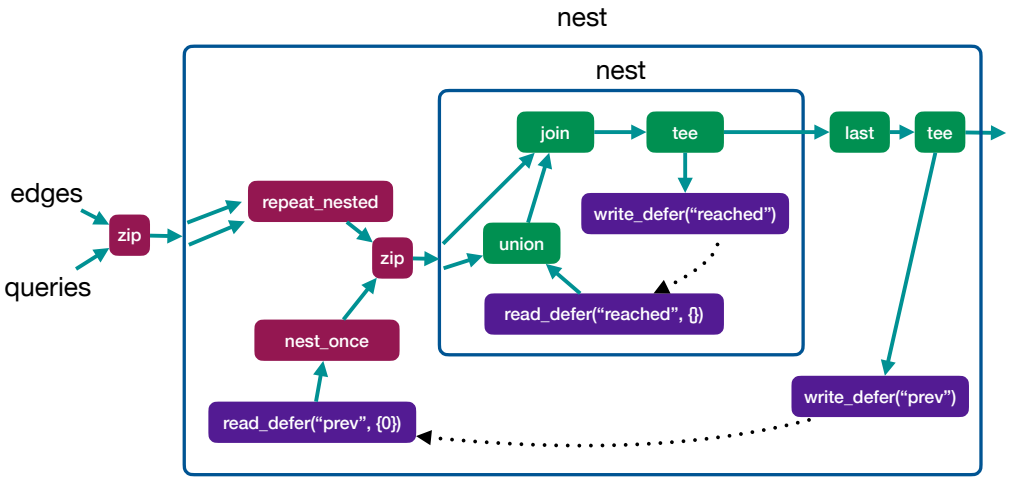


Figure 2.9: An example of graph reachability with a dynamic radius, using nested cycles.

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` to add the bootstrap nodes. Finally, we use `repeat_nested` as before to drive iterations of graph reachability.

2.5.3 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

[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.

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 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.

SEQUENCE

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

PAR

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

OPERATOR

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

READ-DEFER-VALUE-TYPE

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

READ-DEFER-NO-VALUE-TYPE

$$R, k : C; \emptyset \vdash^O \text{read\_defer(k)} : (() \hookrightarrow (C, \mathbf{B}), \emptyset)$$

WRITE-DEFER-TYPE

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

NEST-TYPE

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

NEST-WITH-COPY-TYPE

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

Figure 2.10: Type semantics with defer contexts, and for `read_defer`, `write_defer`, and `nest`.

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 core composition type semantics with these contexts as well as for `write_defer`, `read_defer`, and `nest` in Figure 2.10. 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.

2.5.4 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 2.11 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 2.12.

2.5.5 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

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

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

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

For `read_defer`, any operator expression *without* the value parameter is smaller, 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`.

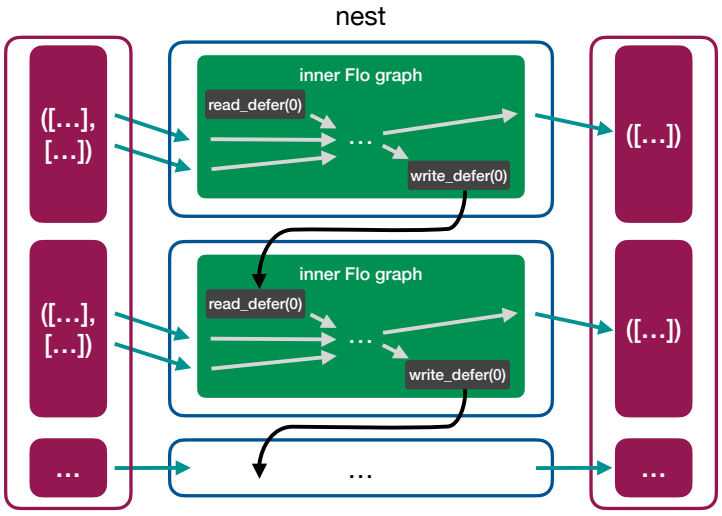


Figure 2.11: Visualization of the `nest`, `read_defer`, and `write_defer` operators, where the nested streams on the left and right have later elements lower.

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 2.3.1, 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 collection type. When we update an input/output, Lemma 2.4.1 guarantees that this is safe. In all our rules, the operator is only changed by setting the nested inputs, 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 2.4.2) and is deterministic (Lemma 2.4.3). □

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, by Lemma 2.4.3 introducing a delta before or after the step will result in the same final state, because introducing a delta to the

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

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

NEST-FIRST

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

NEST-FIRST-FIXED

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

NEST-RUN-GRAPH

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

NEST-RUN-STEP

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

NEST-RUN-FIXED

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

READ-DEFER-EMIT

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

Figure 2.12: Small-step semantics for the `nest` and `read_defer` operators.

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 2.4.4. 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. □

2.6 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 [30], a popular streaming framework that features windowed aggregation functions.
2. LVars [90], a language for parallel programming that uses lattices to ensure determinism.
3. DBSP [24], a system for incremental view maintenance that uses z-sets to model relations.

2.6.1 Flink

Flink [30] 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; we can model this in Flo by introducing a collection type that 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 2.13.

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

$$[v_1, \ldots, v_n] +\!\!+ [d_1, \ldots, d_m] = [d_1, \ldots, d_m, v_1, \ldots, v_n]$$

$$[\otimes, \ldots] +\!\!+ x = [\otimes, \ldots]$$

$$[v_1, \ldots, v_n] +\!\!+ \otimes = [\otimes, v_1, \ldots, v_n]$$

Figure 2.13: 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 2.14 (we omit partial orders for brevity).

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

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

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

Figure 2.14: Operational semantics for Flink operators in Flo.

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 uses windows, where the aggregation is run over blocks of data defined by timestamp intervals. This idea maps perfectly to the Flo model, where we convert an unbounded stream of timestamps-value pairs into a stream-of-streams (as in Section 2.5) 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 2.15 (again omitting the partial order for brevity).

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 2.5, 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.

2.6.2 LVars

LVars [90] is a language for deterministic parallel programming that uses lattice 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

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

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

Figure 2.15: Type and operational semantics for the `window` operator.

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.

$$\texttt{LVar<L>} \triangleq \{(v, \textit{true}) \mid v \in L\} \cup \{(v, \textit{false}) \mid v \in L\}$$

$$(v_1, \textit{false}) +\!\!+ v_2 = (v_1 \sqcup v_2, \textit{false})$$

$$(v_1, \textit{true}) +\!\!+ v_2 = (v_1, \textit{true})$$

$$(v_1, \textit{false}) +\!\!+ \otimes = (v_1, \textit{true})$$

Figure 2.16: 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 2.17.

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

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

Figure 2.17: 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 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 2.18 (omitting partial orders).

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

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

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

$$\textsc{lvar-threshold-terminated}\quad ((v, true), \text{thresh}(\ldots)) \rightarrow^{\delta} (\otimes, \text{thresh}(\ldots), \otimes)$$

Figure 2.18: Type and operational semantics for the `threshold` operator.

over collection values or any algebraic properties, still map very precisely to the approach taken in LVars to enable deterministic data processing.

2.6.3 DBSP

Another point in the streaming language design space comes from the database community. DBSP [24] 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 2.19. 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.

$$\text{Cardinality Maps: } M = \{k_1 : v_1, \ldots\} \text{ where } v_i \in \mathbb{Z},\ M[k] = 0 \text{ if } k \notin M$$

$$(M_1 + M_2)[k] = M_1[k] + M_2[k]$$

$$\text{ZSet} = \{(m, true) \mid m \in M\} \cup \{(m, false) \mid m \in M\}$$

$$(M_1, false) +\!\!+ M_2 = \{(M_1 + M_2, false)\}$$
$$(M, \_) +\!\!+ \otimes = \{(M, true)\}$$

Figure 2.19: Collection type and concatenation operator for z-sets in Flo.

In DBSP, inputs to the program are z-sets, and we do the same when mapping this to Flo. Next, we define operators over z-sets. Let us define `map`, a general version of projection, in Figure 2.20. We omit typing rules for brevity, but the output boundedness is the same as the input.

MAP-ZSET

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

MAP-ZSET-TERMINATED

$$((\{\}, \mathit{true}), \mathrm{map}(f)) \to^{\delta} (\otimes, \mathrm{map}(f), \otimes)$$

Figure 2.20: 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.

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

JOIN-ZSET

$$((M_1', s_1), (M_2', s_2), \bowtie (M_1, M_2)) \to^{\delta} (((\{\}, s_1), (\{\}, s_2), \bowtie (M_1 + M_1', M_2 + M_2'), (M_1 \bowtie M_2' + M_1' \bowtie M_2 + M_1' \bowtie M_2'))$$

JOIN-ZSET-TERMINATED

$$((\{\}, \mathit{true}), (\{\}, \mathit{true}), \bowtie (\_, \_)) \to^{\delta} (\otimes, \otimes, \bowtie, \otimes)$$

Figure 2.21: 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 multiplying the 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 2.21 (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 [44].

2.6.4 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.

2.7 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.

2.7.1 Stream Types and Deterministic Dataflow

The most closely related work to Flo recently is Stream Types [44], 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 computation with retractions, a key feature of DBSP that Flo can capture.

Other work defines streams as monoids [102, 103] 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.

2.7.2 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 [29], and highlights the foundational influence of CQL [14]. 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 [17, 31]".

The same tutorial also points out various constructs that stream query languages introduced for *tracking* progress, including punctuations [156], watermarks [7], heartbeats [138], slack [1], and frontiers [112]. 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 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 2.6.3, recent approaches [24, 112] 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.

2.7.3 Streaming Dataflow Systems

There has been much work on building *performant* streaming dataflow systems, particularly for use in analytical workloads. Systems like Samza [118], Storm [74], Flink [30], Heron [89], Beam [101], and Spark Streaming [166] all provide complete systems for stream dataflow. These systems 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 persistence of data on distributed nodes, an operational concern that we abstract away in Flo.

More recent work has applied batching to improve performance [88, 120], 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.

2.7.4 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 [33, 75, 76, 87, 122]. These programs can be formalized as streams, and are often compiled to a streaming dataflow represen-

tation similar to Flo. Of particular interest are papers which reason about avoiding space-time leaks [75, 87], requiring a property similar to our streaming progress condition.

Other work in this space has focused on the correspondence between LTL and FRP [33, 76, 122], or have focused on the incrementalization of functional programs [60, 163]. 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 [16, 22, 32, 114, 151]. While these present properties that are similar to eager execution and streaming progress, they are formalized with a syntax and semantics for a particular language. In contrast, Flo offers an abstract framework for streaming languages, with minimal constraints. We believe that Flo provides a basis to build such languages.

An incremental, streaming language of particular interest is Naiad [112], 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 [136]. 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.

2.8 Summary

In this chapter, 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.

Chapter 3

Stream-Choreographic Programming

3.1 Introduction

Choreographic programming [59, 110, 132] has recently emerged as a promising paradigm for distributed programming. Unlike architectures such as actors [65] and microservices [54], which require developers to split up their software according to network boundaries, choreographic programming allows developers to write "global" programs where each step in a function can be assigned to execute on a different networked "location". Instead of compiling to a single binary, choreographic programs are "projected" to multiple binaries–one for each location.

This approach has several key advantages when it comes to both correctness and modularity. In traditional architectures, each service is compiled independently, which means that the type checker is restricted to the scope of a single networked unit and cannot perform semantic analysis across machines. With a choreographic model, the entire system is compiled at once so the type system can propagate information across values materialized on different machines. Similarly, choreographic programming makes it possible to modularize shared logic such as consensus protocols as functions, which can be integrated into larger distributed systems.

Where current choreographic semantics struggle is *scalability* and *performance*. The semantics of choreographic programming are based on a synchronous execution model, which is not well-suited for modern distributed systems that often rely on asynchronous communication. In particular, choreographic programs do not offer mechanisms to model concurrent message arrival from several sources or out-of-order delivery. This means that we cannot use choreographic programming to write protocols that rely on replication or partitioning across a cluster, including fundamental ones like Paxos [93] and Raft [119].

What is missing from choreographic programming is the ability to reason about messages over *time*. Instead of writing sequential code that represents a single execution, we need semantics that capture concurrent executions across several machines and let developers program them together instead of one at a time. Stream and dataflow programming is a natural fit for this.

In stream programming languages, developers manipulate *streams* that represent many (possibly infinite) messages arriving over time. Instead of manipulating individual messages, develop-

ers use functional transformations like `map` and `filter`. This model has been widely adopted in distributed data processing frameworks like Spark Streaming [166] and Flink [30], which achieve high performance by abstracting away lower-level details such as batching and retries.

Most streaming frameworks today focus on high-throughput analytical workloads, where latency is not a primary concern. In these frameworks, streams have a *monolithic semantics* that guarantee exactly-once arrival and deterministic order. But to guarantee these semantics, streaming frameworks automatically insert checkpoints and retries, which reduce performance and take away control from developers. When implementing foundational infrastructure such as a cluster membership protocol or key-value store, developers need explicit control over every message, so any automatic fault tolerance is a non-starter.

In Chapter 2, we developed Flo, a metalanguage for stream processing that generalizes "streams" beyond ordered sequences. In Flo, a stream can represent an unordered collection of elements, or esoteric "collections" such as Z-Sets [24] or lattice values [90]. We can utilize this to precisely model concurrent distributed programs—applying specially designed stream types to *capture* concurrency and non-determinism instead of *hiding* it with runtime protocols.

Existing stream frameworks also perform automatic placement of compute across machines, which is not suitable for a model that offers low-level control over fault tolerance. This is where our inspiration from choreographic programming comes in. In our model, developers explicitly place streams on *locations* that represent the machines where the stream is materialized. A location can represent either a single machine or a dynamically scaled cluster of machines. By tracking locations in the type system, we know exactly where network boundaries exist and can reason about their semantics.

These two ideas–stream programming and choreographic locations–combine to form a new programming model that we call *stream-choreographic programming*. In this chapter, we present **Gyatso**, a metalanguage for stream-choreographic programming. Like Flo guarantees determinism, Gyatso guarantees two key properties: **eventual determinism** and **monotone outputs**. Gyatso guarantees that outputs will be *deterministic* even if they are subject to various network conditions, including reordering and temporary failures. And when machines crash, Gyatso guarantees that no unexpected side effects will occur. Together, these properties give developers an expressive and safe way to write high-performance distributed systems, from low-level protocols to high-level data analyses.

In summary, we make the following contributions:

- We present *Gyatso*, a metalanguage that extends Flo to distributed computations by adding **process and cluster location types** (Section 3.3)
- We extend Flo's core properties to distributed systems with **eventual determinism** and **monotone outputs**, and walk through their proofs and system assumptions (Section 3.4)
- We introduce **network operators** that move streams across locations while accurately modeling potential sources of non-determinism (Section 3.5)

3.2 Choreographies to Stream Choreographies

Choreographic programming is a paradigm that enables developers to write distributed programs as a single, unified specification. Instead of manually partitioning logic across networked components, choreographic programming allows developers to define "global" programs where each step is explicitly assigned to a networked "location".

At its core, choreographic programming relies on *projection*, where the global program is automatically sliced into multiple local programs, one for each location. Whenever data moves across locations, the compiler automatically adds networking and serialization logic on both sides. These local programs are guaranteed to coordinate in a way that preserves the sequential semantics of the global specification. By integrating the type system across locations, choreographic programming also provides strong guarantees about correctness, such as ensuring that serialization formats match across machines.

While choreographic programming provides a simple interface, it is significantly hampered by its underlying sequential semantics. Choreographic programs are typically written as functions where local variables and statements can span several locations. By compiling networking logic wherever a local variable needs to be sent across locations, it is easy to preserve the sequential semantics of the original implementation. But because choreographic programs focus on the semantics of a *single* request and do not provide any guardrails for interaction with concurrent logic, developers have little assistance reasoning about concurrency.

3.2.1 Stream Choreographies

Stream choreographies are a natural extension of choreographic programming that explicitly exposes *concurrency*. Instead of writing sequential code that represents a single execution, the streaming model lets developers write logic that operates over several concurrent requests.

From the choreographic model, we retain the concepts of *locations* and *projections* to allow developers to explicitly place streams on different machines for performance and fault-tolerance reasons. But unlike choreographic programs, the streaming inputs and outputs to a function represent *all requests over time* rather than a single request. This makes it possible for several requests to interact with shared state, and lets us more precisely reason about concurrency.

In choreographic programming, data can be sent over the network by writing a value to a "remote variable" at another location. The compiler handles this by adding networking logic to serialize the value and issue an RPC to continue execution on the recipient machine. In stream-choreographic programs, we introduce an equivalent construct that lets developers "move" a stream across locations. This is done by calling a network function that takes the original stream and a target location, and returns a new stream that will receive elements over the network.

Traditional choreographic programming guarantees that the distributed semantics map exactly to the sequential semantics if all locations were removed from the program. In stream-choreographic programming, we explicitly expose distributed semantics, so we need a richer set of proof guarantees. We take inspiration from our foundations in Flo, which guarantees *deter-*

minism in the face of non-deterministic operator execution and *streaming progress* to ensure that the program does not block on unbounded streams.

Stream-choreographic programming similarly focuses on **eventual determinism**, where the outputs of a program will eventually resolve to a deterministic value once all network messages have been processed. This property relies on liveness assumptions on the individual machines, and focuses on the "happy path" of typical execution. To reason about failures, we also guarantee **monotone outputs**, which ensures that even if machines irrecoverably fail, then the outputs of the program will be a subset of the intended output. This ensures that the distributed system will never release any unintended side effects, an important correctness concern.

3.2.2 Cluster Types

The other key innovation enabled by stream programming is the introduction of *cluster location types*. In traditional choreographic programming, each location represents a *single machine*, which maps directly to the sequential semantics. But in most distributed systems, developers need to scale out their applications to handle large workloads. This is typically done by partitioning data across several machines, or replicating data across several machines for fault tolerance.

While scale-out programming can be simulated in traditional choreographic programming by creating several locations, this requires the size of the cluster to be known at compile-time. What we need is a type of location that represents a set of machines with *unspecified size*. Our type system and semantics must then abstractly reason about concurrent execution so that the developer can arbitrarily scale out their application during deployment.

By embracing concurrency, the stream programming model lets us introduce a second type of location: *clusters*. When a piece of logic is placed on a cluster, the same logic will execute on *every* machine (à la Single-Program-Multiple-Data). In the compiler, we generate a single binary for the cluster, which is concurrently executed across several machines. When networking to and from a cluster, the developer is provided with dynamic identifiers to address specific members, even though they are not known at compile time. While this introduces new forms of network non-determinism, such as interleaving of messages from cluster members, our stream model allows us to precisely capture this behavior and guard the developer from correctness bugs.

3.3 Dataflow with Locations

Gyatso is an extension of Flo, a metalanguage for asynchronous stream processing. Flo provides a compositional dataflow model that lets developers string together **operators** that transform streaming data. These operators operate over input and output buffers containing concrete (finite) **collection** values. Because Flo is a metalanguage that does not have any built-in operators, developers bring their own collection types and operators that satisfy a core set of properties. Flo guarantees that any composition of such operators satisfies determinism and streaming progress.

We extend Flo with **locations** that represent the machines where streams are materialized. Locations are a first-class concept in Gyatso, and they are used to track the placement of streams

in the type system. This allows us to reason about the semantics of distributed programs and explicitly capture networking. In Gyatso, collections are identical to Flo; they must define a concatenation operator ⧺ that declares how new elements are added to the collection. What changes is the type system and execution semantics for operators. In this chapter, we begin with the type system changes needed for single-machine process locations. Then, we introduce cluster locations, which require extensions to both the type system and operational semantics to capture concurrent execution.

3.3.1 Location Types

The simplest type of location in Gyatso is a **process**, which represents a single machine where computations occur. A process (or cluster) location in Gyatso is associated with a **location tag**, which is a unique identifier for the location, such as **Leader** or **Worker**. This tag is used to identify the location in the type system and in the projection process, where one compiled program will be emitted for each unique location type in the program.

$\langle LTag \rangle$ ::= $\langle identifier \rangle$

$\langle L \rangle$ ::= Process[$\langle LTag \rangle$] | Cluster[$\langle LTag \rangle$]

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

REFLEXIVE-SUBTYPE
$S \leq S$

BOUND-SUBTYPE
$(C, \mathbf{B}, L) \leq (C, \mathbf{U}, L)$

Figure 3.1: The grammar for choreographic stream types, where $T \in T^C$, and the subtyping relationship for stream types.

To keep track of locations in Gyatso programs, we extend the type system in Flo to add a third element to each stream type containing the location where the stream will be materialized. Because each location represents a different set of machines, there is no subtyping relationship between streams with different locations; the existing relation for bounded and unbounded streams is preserved. We detail the updated stream type definitions in Figure 3.1, where T^C and the definitions of **B** and **U** are unchanged from Flo.

3.3.2 Choreographic Operators and Composition

With stream types augmented to capture locations, the next step is to define the semantics of operators in Gyatso and their composition into *global* dataflow graphs. Like Flo, Gyatso is a metalanguage that does not define any operators. This includes network operators; Gyatso only declares proof obligations, but we define some basic examples in Section 3.5.

We use the same small-step operational semantics as Flo to define the behavior of operators. Formally, Gyatso is parameterized on $(I, e^o) \rightarrow^{\delta} (I, e^o, O)$, a small-step operational semantics where $I, O \in [C]$ (the language of collection expressions) and $e^o \in E^O$ (the language of operator

expressions). Like Flo, Gyatso operators can consume their inputs in any way, but the outputs must be emitted as a "delta" collection that will be concatenated to the following output buffer.

Our main task is to define *where* each operator in a Gyatso program will be executed, since the program now spans several machines. Because the small step of an operator can atomically modify all the input buffers arbitrarily, we require that all the input streams be located at the same location, which is also where the logic of the operator will be executed.

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

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

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

Figure 3.2: Grammar and type semantics for graphs of a Gyatso program.

In Figure 3.2, we lay out the grammar and type semantics for composing operators into a global dataflow graph. Note that both composition rules (SEQUENCE and PAR) are unchanged from Flo; the only difference is that OPERATOR requires all inputs to be at the same process location with a tag L. To model cluster locations, where the same operator will be executed on different machines with independent inputs, we will extend our type system in Section 3.3.3.

Next, we show how existing Flo operators can be used in Gyatso. This can be performed via an "upgrade" process, where we derive a Gyatso type in $\vdash^O$ based on an existing Flo type in $\vdash^{O'}$. When we upgrade a Flo operator, we get a "local" Gyatso operator, where the inputs and outputs are at the same process location L (we will extend this to clusters in the next section). We detail the upgrade semantics in Figure 3.3. Other than network operators, all operators in Gyatso are local and satisfy the same properties as Flo operators.

UPGRADE-PROCESS

$$\frac{\vdash^{O'} op : (I \hookrightarrow O, \prec) \qquad I = ((S_{I_0}, B_{I_0}), \ldots) \qquad O = ((S_{O_0}, B_{O_0}), \ldots)}{\vdash^O op : (((S_{I_0}, B_{I_0}, \text{Process}[L]), \ldots) \hookrightarrow ((S_{O_0}, B_{O_0}, \text{Process}[L]), \ldots), \prec)}$$

Figure 3.3: Derivation for upgrading a Flo operator to a Gyatso operator.

3.3.3 Cluster Locations

A unique feature of Gyatso that distinguishes it from traditional choreographic programming is the notion of *cluster locations*. Unlike processes, which represent a single machine, cluster locations represent a set of machines whose size is not known at compile time. This allows developers to write programs that can scale out to handle large workloads, such as replicated key-value stores [162] or MapReduce workloads [50].

Following the same syntax as processes, clusters in Gyatso are denoted as $\text{Cluster}[L]$, where L is a location tag that uniquely identifies a set of machines. Note that if a process and cluster share the same type tag, they are still considered independent locations with no relationship to each other and the compiler will emit separate projections for both.

Clusters follow single-program-multiple-data (SPMD) semantics, where each operator is concurrently executed on several machines with different input data. To model this in our semantics, we introduce *multibuffers*, which keep track of several buffers for a single stream. We model concurrency in the operational semantics by non-deterministically picking which member of a cluster takes a step, and thus which pieces of the multibuffer are used.

We expose the entire multibuffer in the program syntax to simplify the operational semantics. Cluster multibuffers are denoted as a mapping from integer IDs (which uniquely identify a member of a cluster) to collection values, each of which tracks the local buffer on a single machine. Our type semantics do not inspect the size of multibuffers, which means that the same program can be run with an arbitrary number of machines in each cluster. We begin by formalizing the type semantics for an operator and its multibuffer inputs in Figure 3.4.

$$\frac{\vdash^{O} op : (I \hookrightarrow O, \prec) \qquad I = ((S_0, B_0, \text{Cluster}[L]), \ldots (S_n, B_n, \text{Cluster}[L])) \qquad \forall i, j.\ type^{C}(s_{i,j}) = S_j}{\vdash \{(\{id_0 : s_{0,0}, \ldots, id_m : s_{m,0}\}, \ldots, \{id_0 : s_{0,n}, \ldots, id_m : s_{m,n}\})\}[op] : (I \hookrightarrow O, (\prec))} \text{OPERATOR}$$

Figure 3.4: Type semantics for an operator with inputs placed on a cluster.

Just like we "upgraded" our type semantics from Flo to Gyatso, we need to upgrade our operational semantics to allow Flo operators to be placed on clusters. This process needs to capture the concurrency across cluster members, so we need to define a new small-step semantics that operates over multibuffers. We do this by non-deterministically delegating the small-step to one of the cluster members, which will then execute the operator on its own input buffer.

Since operators carry their own state, we define a syntax for cluster-located operators that use a similar key-value structure to store the state of each operator on each machine. To bring this into the Gyatso metalanguage, we define the type semantics for cluster-located operators in Figure 3.5. We follow the same structure as the upgrade to process locations, except that the operator state must have the same type across all cluster members. We also introduce a new

partial order $\prec_{\text{multiop}}$ that captures the consumption order over inputs, which delegates to the original partial order for each multibuffer element and operator state.

$$(\{\ldots, id_x : s_x, \ldots\}, \{\ldots, id_x : op_x, \ldots\}) \prec_{\text{multiop}} (\prec)(\{\ldots, id_x : s'_x, \ldots\}, \{\ldots, id_x : op'_x, \ldots\})$$
$$\text{if } (s_x, op_x) \prec (s'_x, op'_x)$$

$$\textsc{upgrade-cluster}\quad \frac{\forall_i \vdash^{O'} op_i : (I \hookrightarrow O, \prec) \qquad I = ((S_{I_0}, B_{I_0}), \ldots) \qquad O = ((S_{O_0}, B_{O_0}), \ldots)}{\vdash^{O} \{id_0 : op_0, \ldots\} : (((S_{I_0}, B_{I_0}, \text{Cluster}[L]), \ldots) \hookrightarrow ((S_{O_0}, B_{O_0}, \text{Cluster}[L]), \ldots), \prec_{\text{multiop}} (\prec))}$$

Figure 3.5: Type semantics for an operator placed across a cluster.

Next, we move on to the operational semantics. We start by defining a new concatenation operator for multibuffers, which executes pairwise over the cluster members; this is needed to inherit the $\rightarrow$ small-step that concatenates deltas to the output buffer. Any operator whose output will be materialized on a cluster must emit deltas in multibuffer form in order for the compositional semantics to be well-formed.

$$(\{id_0 : s_0, \ldots, id_n : s_n\} +\!\!+ \{\ldots, id_x : \delta_x, \ldots\})[id_i] = s_i \qquad \text{if } \delta_i \text{ is not defined}$$
$$(\{id_0 : s_0, \ldots, id_n : s_n\} +\!\!+ \{\ldots, id_x : \delta_x, \ldots\})[id_i] = s_i +\!\!+ \delta_i \qquad \text{if } \delta_i \text{ is defined}$$

$$\textsc{cluster-upgrade}\quad \frac{\{(s_{x,0}, \ldots, s_{x,n})\}[op_x] \rightarrow^{\Delta} (\{s'_{x,0}, \ldots, s'_{x,n}\}[op'_x], \delta_x)}{\begin{array}{c}\{(\{\ldots, id_x : s_{x,0}, \ldots\}, \ldots)\}[\{\ldots, id_x : op_x, \ldots\}] \rightarrow^{\Delta} \\ (\{(\{\ldots, id_x : s'_{x,0}, \ldots\}, \ldots)\}[\{\ldots, id_x : op'_x, \ldots\}], \{id_x : \delta_x\})\end{array}}$$

Figure 3.6: Small-step semantics for an operator with inputs placed on a cluster.

Then, we extend the small-step relation $\rightarrow^{\Delta}$ from Flo. This relation steps a Flo graph to yield an updated graph, as well as a delta that will be concatenated to the output buffer. The remaining graph composition rules, which are recursively defined in terms of $\rightarrow^{\Delta}$, are unchanged from Flo. We detail these semantics in Figure 3.6.

Because the small-step can non-deterministically pick which (x) cluster member takes a step, these simple semantics can precisely capture the various forms of concurrency that can occur in

a cluster. Note that because the operator handles each member independently, we still preserve the determinism and progress properties of Flo when considering the inputs and outputs across all members (the proof is straightforward).

This approach to modeling distributed semantics makes it easy to reason about behavior *across* a cluster. To reason about the local semantics for each member, we can project a cluster-located graph to extract the buffers and state for the specific member. The resulting graph, which will contain no multibuffers, is a standard Flo graph that can be analyzed with existing techniques.

3.4 Distributed Correctness

One of the key innovations of Flo is identifying two general but strong properties that all streaming programs should satisfy: **determinism** and **streaming progress**. In the context of Flo, determinism focuses on ensuring that outputs are not affected by non-deterministic operator scheduling, and progress focuses on ensuring that operators do not block on asynchronous events with unknown arrival.

In Gyatso, streaming progress is preserved as-is, and we do not modify its core definition. Local operators, which are inherited directly from Flo, already satisfy this property. For network operators, which send streams between locations, progress is trivially satisfied because we never need to block on an input stream before sending data.

Determinism is more complex. In Flo, determinism is modeled as "freezing" the inputs, letting the dataflow execute until no more small-steps can be taken, and then inspecting the fixed outputs. In a distributed system, we must tweak our formal definitions to account for arbitrary network latency between machines and failures that cause data loss.

To extend determinism to distributed systems, we introduce two new formal properties: **eventual determinism** and **monotone outputs**. Eventual determinism guarantees that when there are no machine failures and all network messages are processed, the outputs of a program will resolve to a deterministic value. We cannot guarantee determinism when machine failures occur, but instead introduce a weaker property called **monotone outputs**. This guarantees that the outputs will always be a "subset" of the intended output, even if the complete value is lost.

Like Flo, the properties in Gyatso are *compositional*, so the only proof burden is for local and network operators. In this section, we walk through how Flo properties on local operators translate to our distributed properties, and show how to adapt the Flo proof structure for graph composition to these new guarantees.

3.4.1 Distributed Concurrency

Flo is focused on modeling the semantics of *asynchronous* stream processing, and so already models many forms of concurrency that we care about in a distributed context. In Flo (and Gyatso) programs, the small-step rules for sequential and parallel composition non-deterministically pick one of the two subgraphs to execute. This means that whenever the global graph is stepped, a

single operator in the graph will take a step. In Gyatso, we can preserve these semantics as-is, as the non-deterministic operator scheduling already captures all forms of distributed concurrency[1].

The simplest form of distributed concurrency occurs inter-location, across two processes or clusters with different tags. In Gyatso, operators that are placed on two different locations will never share input or output buffers, because there must be a networking operator in between to move the stream across locations. This means that the small-step semantics of the operators will never interact directly with each other. Therefore, concurrent execution of operators on different locations is semantically equivalent to a small-step trace where one operator runs before the other without any network operators connecting the two taking a step.

The other form of distributed concurrency we are concerned with is across members of a cluster, where the operators are on the same location but distributed across several machines. Because we model cluster execution using multibuffers, we can similarly model concurrency via non-deterministic selection of which cluster member takes a step. Because cluster operators can only interact with a single member's state, concurrent execution across members is equivalent to taking a step with one member at a time. The power of this approach is that we can continue to reason about the semantics of the program as a sequence of small-steps taken one-by-one, and we do not need to model concurrent small-steps. This means that we can leverage many of the existing proofs in Flo as-is, since they rely on small-step semantics that are identical in Gyatso.

3.4.2 Machine Failures

The other major difference between Flo and Gyatso is the need for a fault model that exposes the effects of machine failures on the rest of the dataflow graph. In Flo, the program runs on a single machine and properties such as determinism rely on the assumption that the program can always be stepped until no-more small-steps are available (called the "stuck state"). In Gyatso, we cannot make this assumption because some locations may fail and stop processing data, while other locations may continue to run. This requires us to reason about the effects of machine failures on the program, and how they affect the properties we care about.

In Gyatso, we use a crash-stop fault model, where machines can fail and stop sending data, but they do not emit corrupted outputs or attempt to recover by re-launching the program. This is a common fault model in distributed systems, and it allows us to reason about machine failures without introducing additional complexity such as persistent state.

In our small-step semantics, the crash-stop model means that portions of the graph for a specific location may be frozen arbitrarily, at which point they will permanently cease to take small-steps. Similarly, for graphs on cluster locations, elements of the multibuffers for specific failed members may be frozen and the small-steps for those members will not execute. This approach broadly preserves the semantic structure in Flo, but we must update the proofs which assume the existence of a reachable stuck state.

[1] To keep our semantic model simple, we rule out semantics for true temporal concurrency (e.g. synchronized clocks and the ability to test equality of timestamps).

For determinism and progress, we introduce a precondition of *liveness*. Our definition follows from distributed systems literature, where a machine is considered live if it has the opportunity to take a small-step equally often. Consider an execution of a Gyatso program where all machines are live. If the inputs are frozen, the graph will eventually reach a stuck state because each operator will have the opportunity to take as many steps as are needed to deplete their inputs. Therefore, when all machines are live, the proofs of determinism and progress from Flo can be applied directly to Gyatso.

Note that this guarantee implicitly relies on the property that Gyatso graphs do not have cycles involving unbounded streams. If we were to introduce cycles across the network, we can no longer guarantee properties such as determinism, because there may not be a finite value that the output resolves to. But determinism still holds *with respect to* all inputs that are not involved in a cycle, and is still a useful property for developers.

3.4.3 Eventual Determinism and Eager Execution

Gyatso guarantees **eventual determinism**, which we formally define as follows:

Theorem 3.4.1 (Eventual Determinism)**.** *Consider a Gyatso program g that is not in a stuck-state. Assume that all machines used in g are live and all network messages are eventually delivered. For all inputs I and initial outputs O, there exists a state g' with inputs I' and outputs O' such that g' is in a stuck state, and every trace of small steps eventually reaches this later state.*

This is nearly identical to the determinism property in Flo. The only change is to account for unknown network latency, rather than guaranteeing that the outputs will resolve in a finite number of steps. In Section 3.5, we will model network operators so that each step always makes progress, and latency is modeled by delaying when that step is taken relative to other operators. By ensuring that network operators eventually make progress, we can preserve the proof structure from Flo that relies solely on the eager execution property. We omit this proof for brevity, but it is nearly identical to the proof of determinism in Flo.

Where Gyatso must do additional work is to accurately model the sources of non-determinism introduced by network operators. Instead of directly passing collection values as-is, reordering and interleaving may result in a resolved output different from what was sent. To tackle this, we will also introduce special collection types in Section 3.5 that account for this non-determinism by collapsing variants that differ *only* in order or cardinality. By using these collection types appropriately, each network operator will satisfy the standard eager execution property from Flo, and therefore will preserve eventual determinism.

3.4.4 Monotone Outputs and Concatenation

When machine failures *are* present, Gyatso provides guarantees that the program will never emit an *incorrect* output, even if that output is *incomplete*. This guarantee, inspired by work on the CALM theorem [64], is called **monotone outputs**. This property states that if a program is in a stuck state, even if due to failures, then the outputs will be a subset of the intended output.

The monotone output property is a powerful guarantee that gives developers confidence in the outputs of a Gyatso program even when failures occur. If the outputs are used to drive side effects such as issuing transactions or sending network messages, the monotone output property guarantees that any issued side effects are *intentional* and safe to perform.

First, we must define what it means for an output to be a "subset" of an intended output. To do this, we introduce the definition of a collection's *natural partial order*, which gives us a way to describe the relationship between collection values and how one can grow into the other. In Flo, the concatenation operator $+\!\!+$ does not need to obey any partial order, and can be defined arbitrarily. In Gyatso, collection types are still allowed to have an arbitrary concatenation operator, but can *optionally* define a partial order that is in alignment with concatenation in order to benefit from the monotone output property. We formally define this as follows:

Definition 3.4.1 (Collection Natural Order)**.** A collection type $C \in T^C$ may optionally define a natural partial order $\prec_C$, which is a reflexive, transitive relation over C such that for all $c_1, c_2, \delta \in C$, if $c_1 +\!\!+ \delta = c_2$ then $c_1 \prec_C c_2$.

As an example of a natural partial order, consider a set collection type $C = \mathcal{P}(T)$, where T is a type. The concatenation operator over sets is union ($c_1 +\!\!+ c_2 = c_1 \cup c_2$), so a valid natural partial order for sets is the subset relation ($\subseteq$). If we have an ordered sequence collection where new elements are concatenated to the end, the natural partial order is the prefix relation. This even extends to more esoteric collection values such as semilattices, where the concatenation operator ($c_1 +\!\!+ c_2 = c_1 \sqcup c_2$) induces a partial order ($c_1 \sqsubseteq c_2 \iff c_1 \sqcup c_2 = c_2$).

We can use this extension to collection types to formally define the monotone outputs property, which applies to any output of a Gyatso program which has a well-defined natural partial order. We define this as follows:

Theorem 3.4.2 (Monotone Outputs)**.** *Consider a Gyatso program g with inputs I and outputs O, even if it is stuck due to machine failures. By the determinism property Theorem 3.4.1, there exists some state g' with inputs I' and outputs O' such that g' is in a stuck state and if all machines were to be live, all executions of g would eventually reach this state. For each output $o_i \in O$ (and $o'_i \in O'$) whose collection type has a natural order $\prec_{C_i}$, we guarantee that $o_i \prec_{C_i} o'_i$.*

Proof. The trace of small-steps leading from g to g' consists of a series of small-steps for operators in the graph. The deltas emitted by $\rightarrow^{\Delta}$ are always concatenated to each of the output buffers o_i, and there are no other rules in our semantics that modify these buffers. Therefore, each updated output buffer o'_i consists of applying zero or more deltas to the original value ($o'_i = o_i +\!\!+ \delta_0 +\!\!+ \ldots$). By Definition 3.4.1, we know that if a natural partial order $\prec_{C_i}$ exists for output's collection type, then $o_i \prec_{C_i} o'_i$ because the concatenation operator must be in alignment with the natural order. □

Together, eventual determinism and monotone outputs provide a strong foundation for reasoning about the correctness of distributed programs in Gyatso. Eventual determinism ensures correctness in the happy path of failure-free execution, and monotone outputs ensure that the system does not go astray when failures occur.

3.5 Network Operators

So far, our formal semantics allow Gyatso programs to be placed on processes and clusters, but cannot span multiple locations because we have not yet defined operators that can facilitate communication between them. Instead of relying on implicit networking when a stream needs to move to another location, we instead introduce special *network operators* that facilitate the move while carefully modeling the non-deterministic network semantics. This approach gives developers the flexibility to choose between network operators that have strong guarantees but introduce overhead, or operators with weaker guarantees but higher raw performance.

Although network operators at runtime will use various system calls to transmit and receive data, in Gyatso we focus on modeling their high-level behavior rather than low-level details. As such, the operational semantics of each network operator need to precisely capture characteristics such as message reordering or duplication. In Gyatso, network operators follow the same structure as standard computational operators, fitting into the existing type system and expressing their operational semantics in terms of small-steps.

A key facet of the network that distinguishes it from local operators is that sending a piece of data involves potentially unbounded latency, whereas local operators are guaranteed to make progress in their computation whenever a small-step is taken. A naive way to model network latency is to allow network operators to take "no-op" steps where no progress is made, but this runs into issues with ensuring that the system *eventually* makes forward progress.

We note two insights of our semantics that allow us to simplify the modeling of network operators. First, we know that the compositional semantics of Flo and Gyatso can be captured as a linear trace of small-steps on one operator at a time. Second, because operators in Gyatso cannot observe how many steps have been taken in the global program, a no-op step taken by a network operator is not observable by any other logic.

This means that we can eliminate the no-op steps entirely, instead modeling the arrival of network messages *relative* to the execution of other operators (under a network liveness assumption). With these semantics, our network operators always make progress and benefit from the existing proofs of progress in Flo. In the remainder of this section, we walk through various network operators that facilitate communication between locations.

3.5.1 Process-to-Process Networking

The simplest network operator in Gyatso handles one-to-one networking between two process locations. We rely on standard TCP semantics, which guarantee in-order and exactly once delivery of messages. This means that the operator, which we call `network_o2o` can deterministically transfer ordered sequence streams between two locations.

To model the network semantics, our operator tracks messages in-transit via a buffer in its state. Whenever taking a step, the operator can either push a new element into this buffer (sending the data) or pop an element into the output (receiving the data). Because we use the ordered sequence collection, this process preserves the exact order and cardinality of the original stream. We formally define the type and operational semantics of the operator in Figure 3.7.

$$([i_0, \ldots], \text{network\_o2o}(S')) \prec_{\text{network\_o2o}} ([i_0, \ldots, h], \text{network\_o2o}(S))$$

$$([\ldots], \text{network\_o2o}([\ldots])) \prec_{\text{network\_o2o}} ([\ldots], \text{network\_o2o}([\ldots, h]))$$

NETWORK-O2O-TYPE

$$\frac{\forall_i \vdash v_i : T}{\vdash^O \text{network\_o2o}([v_0, \ldots]) : (([T], \mathbf{U}, \text{Process}[L_1]) \hookrightarrow ([T], \mathbf{U}, \text{Process}[L_2]), \prec_{\text{network\_o2o}})}$$

NETWORK-O2O-SEND

$$\frac{}{([i_0, \ldots, h], \text{network\_o2o}([n_0, \ldots])) \rightarrow^{\delta} ([i_0, \ldots], \text{network\_o2o}([h, n_0, \ldots]), [])}$$

NETWORK-O2O-RECV

$$\frac{}{([i_0, \ldots], \text{network\_o2o}([n_0, \ldots, h])) \rightarrow^{\delta} ([i_0, \ldots], \text{network\_o2o}([n_0, \ldots]), [h])}$$

Figure 3.7: Type and operational semantics for the network operator `network_o2o`.

As with operators in Flo, we must prove that this operator satisfies the local property of eager execution, by showing that its outputs are deterministic in the face of asynchronous input arrival, and well-formedness, by showing that each step makes progress. Our operator trivially satisfies eager execution, since it pushes through the input elements in FIFO order. The well-formedness property is also satisfied, since each step either consumes an element of the input buffer or shrinks the buffer of messages in-transit while emitting an output.

Note that the signature of this operator is an identity function from $[T]$ to $[T]$, and the operational semantics guarantee that the output collection is the same as the input. This means that (modulo effects on fault-tolerance), the operator is observationally equivalent to not having any operator at all. This means that we can insert this operator anywhere that a collection type $[T]$ is used to split the graph across multiple locations, which may yield performance benefits in line with recent work on optimizing distributed protocols [38].

3.5.2 Retries and Reordering

Our first network operator has semantics identical to those used by frameworks like Flink [30], which only expose ordered sequence streams. But unlike these existing frameworks, the meta-language approach allows us to use collection types beyond ordered sequences. We can use this to model network protocols with weaker guarantees, such as UDP-with-retries or unordered-QUIC [95]. These alternative network operators give developers the opportunity to squeeze more performance when their program does not rely on the strong guarantees of TCP.

A common pattern in distributed systems is to use a retry mechanism to ensure that messages are delivered even if the network is unreliable. To avoid having to coordinate with the downstream to de-duplicate messages, we can instead expose the retired messages directly to the downstream operator. First, we must define a new collection type that captures the equivalence between two streams that differ only in the message duplication. We will denote this collection type as $[T]^{\text{dup}}$, formalized in Figure 3.8.

$$[i_0, \ldots] \in [T]^{\text{dup}} \text{ if } \forall_x\ i_x \neq i_{x+1} \wedge i_x \in T$$
$$[\ldots] +\!\!+ [] = [\ldots]$$
$$[x, \ldots] +\!\!+ [\ldots, y, x] = [x, \ldots] +\!\!+ [\ldots, y]$$
$$[\ldots, x] +\!\!+ [y, z, \ldots] = [\ldots, x, y] +\!\!+ [z, \ldots] \text{ if } x \neq y$$

Figure 3.8: Collection type for ordered sequences with duplicates.

The concatenation operator for this collection type ignores incoming elements if they are duplicates of the last element in the stream, which lets us preserve order while ignoring duplicates. We can then define a new network operator, `network_o2o_retry`, which uses this collection type to model the retry semantics. Note that we do not need to explicitly model retries being delivered in our operational semantics, because the collection type already captures all possible retries (we assume liveness, that there are only finite retries per element). We define the type and operational semantics of this operator in Figure 3.9.

NETWORK-O2O-RETRY-TYPE

$$\frac{\forall_i \vdash v_i : T}{\vdash^O \text{network\_o2o\_retry}([v_0, \ldots]) : (([T]^{\text{dup}}, \mathbf{U}, \text{Process}[L_1]) \hookrightarrow ([T]^{\text{dup}}, \mathbf{U}, \text{Process}[L_2]), \prec_{\text{network\_o2o\_retry}})}$$

NETWORK-O2O-RETRY-SEND

$$([i_0, \ldots, h]^{\text{dup}}, \text{network\_o2o\_retry}([n_0, \ldots])) \rightarrow^{\delta} ([i_0, \ldots]^{\text{dup}}, \text{network\_o2o\_retry}([h, n_0, \ldots]), []^{\text{dup}})$$

NETWORK-O2O-RETRY-RECV

$$([i_0, \ldots]^{\text{dup}}, \text{network\_o2o\_retry}([n_0, \ldots, h])) \rightarrow^{\delta} ([i_0, \ldots]^{\text{dup}}, \text{network\_o2o\_retry}([n_0, \ldots]), [h]^{\text{dup}})$$

Figure 3.9: Type and operational semantics for the network operator `network_o2o_retry`.

The proof of eager execution for this operator relies on the fact that the concatenation operator for $[T]^{\text{dup}}$ is associative. The challenging case is when an element a is pushed onto the network, and then a arrives again as a delta on the input; because concatenation is associative down to individual elements, the deltas pushed to the output will resolve to the same collection even if the a is concatenated twice. The more complex proofs lie in the computational operators that consume this collection type, which we will discuss shortly.

$$\{i_0 : c_0, \ldots\} \in [T]^{\text{unord}} \text{ if } \forall_x\ i_x \in T \wedge c_x \in \mathbb{N} \wedge \forall_y\ (x \neq y \implies i_x \neq i_y)$$
$$(\{i_0 : c_0, \ldots, i_n : c_n\} +\!\!+ \{\ldots, i_x : \delta_x, \ldots\})[i_j] = s_j \text{ if } \delta_j \text{ is not defined}$$
$$(\{i_0 : c_0, \ldots, i_n : c_n\} +\!\!+ \{\ldots, i_x : \delta_x, \ldots\})[i_j] = s_j + \delta_j \text{ if } \delta_j \text{ is defined}$$

NETWORK-O2O-UNORD-TYPE

$$\frac{\forall_i \vdash v_i : T}{\vdash^O \text{network\_o2o\_unord}([v_0, \ldots]) : (([T]^{\text{unord}}, \mathbf{U}, \text{Process}[L_1]) \hookrightarrow ([T]^{\text{unord}}, \mathbf{U}, \text{Process}[L_2]), \prec_{\text{network\_o2o\_unord}})}$$

NETWORK-O2O-UNORD-SEND

$$\frac{c_h \geq 1}{(\{h : c_h, \ldots\}, \text{network\_o2o\_unord}([n_0, \ldots])) \rightarrow^{\delta} (\{h : c_h - 1, \ldots\}, \text{network\_o2o\_unord}([h, n_0, \ldots]), \{\})}$$

NETWORK-O2O-UNORD-RECV

$$\frac{}{(\{\ldots\}, \text{network\_o2o\_unord}([n_0, \ldots, h])) \rightarrow^{\delta} (\{\ldots\}, \text{network\_o2o\_unord}([n_0, \ldots]), \{h : 1\})}$$

Figure 3.10: Formal definition for T^{unord} and unordered network semantics.

We can similarly define a network operator that models reliable, unordered delivery of messages (which can be achieved using protocols like QUIC [95]). This operator, `network_o2o_unordered`, uses another new collection type, $[T]^{\text{unord}}$, which captures the equivalence between two streams that differ only in the order of the elements. This collection tracks elements as a multiset mapping element values to cardinalities, which allows us to ignore the order of the elements. When sending elements over the network, we will non-deterministically pick which element to send next, modeling the unordered protocol. We define this collection, as well as the type and operational semantics of the operator in Figure 3.10.

Our proof for eager execution is similar to retries, relying on the associativity of the concatenation operator down to an element with a cardinality of 1. Because every input element with cardinality n is eventually pushed onto the buffer n times, we know that it will also be popped into a delta n times. Therefore, because the cardinality of each element is independent, the final output collection will be the same regardless of how the input is broken up.

So far, our semantics are fairly standard, but things become interesting when we explore computational operators that can operate on our new collection types. Many classic operators such as `map` and `filter` can apply with standard behavior to these collection types, but standard implementations of `fold` or `reduce` will violate the eager execution property when applied to our collection types. In the typical semantics for these operators, the aggregation function is applied to the elements being received one-by-one; but if there are non-deterministic duplicates or reordering, the output of the operator may vary.

FOLD-IDEMPOTENT-TYPE

$$\frac{\vdash acc : U \qquad \vdash f : (U, T) \to U \qquad \forall_{s \in U, a \in T, i_0}\ f(f(s,a),a) = f(s,a)}{\vdash^{O} \text{fold\_idempotent}(acc, f) : (([T]^{\text{dup}}, X) \hookrightarrow (\text{Singleton}[U], X), \prec_{\text{fold\_idempotent}})}$$

FOLD-IDEMPOTENT

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

FOLD-COMMUTATIVE-TYPE

$$\frac{\vdash acc : U \qquad \vdash f : (U, T) \to U \qquad \forall_{s \in U, a \in T, b \in T}\ f(f(s,a),b) = f(f(s,b),a)}{\vdash^{O} \text{fold\_commutative}(acc, f) : (([T]^{\text{unord}}, X) \hookrightarrow (\text{Singleton}[U], X), \prec_{\text{fold\_commutative}})}$$

FOLD-COMMUTATIVE

$$\frac{f(acc, h) \Downarrow acc' \qquad c \geq 1}{(\{h : c, \ldots\}, \text{fold\_commutative}(acc, f)) \to^{\delta} (\{h : c-1, \ldots\}, \text{fold\_commutative}(acc', f), acc')}$$

Figure 3.11: Type and operational semantics for the operators `fold_idempotent` and `fold_commutative`.

In order to ensure determinism, we can introduce variants of `fold` called `fold_idempotent` and `fold_commutative`, which place algebraic restrictions on the aggregation function in order to make them immune to the upstream non-determinism. As the names suggest, `fold_idempotent` requires that the aggregation function is idempotent, meaning that applying it to the same value multiple times will yield the same result, and `fold_commutative` requires that the aggregation function is commutative, meaning that the order of the inputs does not matter. We formally define these operators in Figure 3.11.

To close off our discussion, we must prove that these variants indeed satisfy eager execution. In our semantics, we use a "Singleton" collection type to model the output of the operator, whose concatenation operator replaces the current value with what is concatenated ($s +\!\!+ x = x$). Therefore, we only need to prove that the final delta after all inputs are processed is deterministic regardless of when inputs arrive.

For `fold_idempotent`, the problematic case is if an element x is the last element of the initial input and then a delta introduces x as the first element of the input. If the concatenation was

done before processing, the x would be collapsed into a single element and f is called with it only once. When it is split across processing, f will be immediately called twice with x, which by our type constraints will yield the same state and result. Therefore, eager execution is satisfied.

For `fold_commutative`, the concatenation case is easy because we subtract one from the cardinality each step and concatenation across multisets is associative. The challenging piece is that our small-step is non-deterministic in which element is processed first. But due to the commutativity constraint on f, we know that the order of elements provided to f does not matter. Therefore, the final output will be the same, so eager execution is satisfied.

3.5.3 Networking with Clusters

So far, we have looked at network operators that can only send data between processes; operators that move data between clusters require special semantics to account for the parallelism across cluster members. In Gyatso, we need three network operators to model one-to-many, many-to-one, and many-to-many networking, each with special semantics that model message interleaving and partitioning across members. In this subsection, we will focus on operators that rely on TCP semantics, but these can be extended to support weaker protocol guarantees using the same techniques as earlier.

Let us begin with one-to-many communication from a process to a cluster. Unlike our previous networking operators, which consume a sequence of elements of type T, this operator takes in tuples of type $(\mathbb{Z}, T)$, where the first element is the ID of the cluster member to send the message to. This allows us to model "demux" behavior where each incoming message is appropriately routed to a specific cluster member (concepts like broadcast can be implemented by using a cross-product upstream). We define the type and operational semantics of this operator in Figure 3.12.

NETWORK-O2M-TYPE

$$\frac{\forall_i \vdash v_i : T \qquad \forall_i \vdash c_i \in \mathbb{Z}}{\vdash^O \text{network\_o2m}(\{c_0 : [v_0, \ldots], \ldots\}) : (([(\mathbb{Z}, T)], \mathbf{U}, \text{Process}[L_1]) \hookrightarrow ([T], \mathbf{U}, \text{Cluster}[L_2]), \prec_{\text{network\_o2m}})}$$

NETWORK-O2M-SEND

$$\frac{}{([i_0, \ldots, (c, h)], \text{network\_o2m}(\{\ldots, c : [v_0, \ldots], \ldots\})) \rightarrow^{\delta} ([i_0, \ldots], \text{network\_o2m}(\{\ldots, c : [h, v_0, \ldots], \ldots\}), \{\})}$$

NETWORK-O2M-RECV

$$\frac{}{([i_0, \ldots], \text{network\_o2m}(\{\ldots, c : [n_0, \ldots, h], \ldots\})) \rightarrow^{\delta} ([i_0, \ldots], \text{network\_o2m}(\{\ldots, c : [n_0, \ldots], \ldots\}), \{c : [h]\})}$$

Figure 3.12: Type and operational semantics for the network operator `network_o2m`.

This operator preserves strict ordering, since there is no interference of communication across cluster members. The only non-determinism introduced is when cluster members receive data relative to each other (since there is a separate network buffer for each member), but due to commutativity of multibuffer concatenation across cluster members we preserve eager execution.

A more interesting operator is many-to-one communication, which exposes non-determinism to the recipient since there is non-deterministic interleaving of messages from multiple cluster members. This operator takes in a multibuffer representing the input from each cluster member, and emits a single unordered output stream of tuples $(\mathbb{Z}, T)$, where the first element is the ID of the cluster member that sent the message. We define the type and operational semantics of this operator in Figure 3.13.

NETWORK-M2O-TYPE

$$\frac{\forall_i \vdash v_i : T \qquad \forall_i \vdash c_i \in \mathbb{Z}}{\vdash^O \text{network\_m2o}(\{c_0 : [v_0, \ldots], \ldots\}) : (([T]^{\text{unord}}, \mathbf{U}, \text{Cluster}[L_1]) \hookrightarrow ([(\mathbb{Z}, T)]^{\text{unord}}, \mathbf{U}, \text{Process}[L_2]), \prec_{\text{network\_m2o}})}$$

NETWORK-M2O-SEND

$$\frac{n \geq 1}{(\{\ldots, c : \{h : n, \ldots\}, \ldots\}, \text{network\_m2o}(\{\ldots, c : [v_0, \ldots], \ldots\})) \rightarrow^{\delta} (\{\ldots, c : \{h : n-1, \ldots\}, \ldots\}, \text{network\_m2o}(\{\ldots, c : [h, v_0, \ldots], \ldots\}), \{\})}$$

NETWORK-M2O-RECV

$$\frac{}{(\ldots, \text{network\_m2o}(\{\ldots, c : [n_0, \ldots, h], \ldots\})) \rightarrow^{\delta} (\ldots, \text{network\_m2o}(\{\ldots, c : [n_0, \ldots], \ldots\}), \{(c, h) : 1\})}$$

Figure 3.13: Type and operational semantics for the network operator `network_m2o`.

Because the output is unordered, we can rely on the knowledge that the concatenation operator for multisets is commutative. The only non-determinism in our semantics is picking which cluster member to receive the next element from, which results in unknown ordering, but the total cardinality of each element is deterministic. This means that after all concatenations are processed in the output, we get the same multiset and eager execution is satisfied.

The final network operator is for cluster-to-cluster communication, which combines the ID addressing approach of one-to-many with interleaving semantics of many-to-one. For brevity, we omit the formal definition of this operator, but its proofs of eager execution follow from the earlier proofs. Note that in cluster-to-cluster communication, unlike process-to-process, there are meaningful reasons to have the same source and destination cluster, for example to model gossip or re-partitioning within the existing members.

Altogether, these network operators allow developers to write *global* programs that combine local computation with networking across location. As we have shown, the semantics of these

operators are compositional and can be reasoned about using the same techniques as local operators, and preserve the end-to-end determinism properties of Gyatso. Furthermore, by defining special collection types that capture various forms of non-determinism, we enable low-level control over network protocols while accurately capturing their effects on downstream logic.

3.6 Related Work

3.6.1 Languages for Distributed Systems

There has been much work in research and industry developing programming languages and verification tooling for distributed systems. This work shares many of the same motivations as stream-choreographic programming: balancing performance with correctness and modularity.

One of the most widely adopted distributed programming models is actor programming [65]. In the actor model, distributed systems are composed of several single-threaded actors, which communicate with other actors via message passing. This model has been widely adopted in industry, with frameworks such as Akka [125] and Erlang [15] providing robust implementations. Because actors process incoming messages with sequential logic and directly observe network non-determinism, the actor model cannot provide strong guarantees about the global correctness about the system, and developers must reason carefully about concurrency across actors.

Noting this limitation, the P programming language [53] combines the actor model with a verification framework to provide stronger formal guarantees. Programs written in P follow the typical actor structure, but are augmented with formal specifications that can be verified using a model checker. With the appropriate specifications, P ensures correct global behavior. But the specification and proof effort required is often quite high, and can lead to significant overhead in development time. Rather than aiming to prove arbitrary global properties, stream-choreographic programming focuses on preventing non-determinism due to networking and concurrency without any additional specifications from the developer.

More recent work has explored continuations and async-await constructs as a way to write distributed software. The Unison language [150] allows developers to "fork" computation to a separate machine, returning the result as a future that can be await-ed in the original program. This results in a global programming experience similar to stream-choreographic programming, but has sequential semantics that are closer to traditional choreographic programming. This means that Unison programs can still run into concurrency bugs if forked logic attempts to interact with shared state. By taking a streaming approach, we are able to more precisely capture interactions between concurrent requests and guard developers from such bugs.

3.6.2 CALM and Monotonicity

A major line of work on distributed correctness, originating in the databases community, focuses on the Consistency as Logical Monotonicity (CALM) Theorem [64]. The CALM theorem states that the outputs of a distributed system are correct without coordination if and only if the dis-

tributed logic is monotone. This theorem provides a straightforward litmus test on whether a distributed program must use coordination mechanisms to avoid emitting incorrect results.

The properties laid out in Gyatso map directly to the CALM theorem, in particular the *monotone outputs* property. This property guarantees that for outputs where the concatenation operator follows a natural partial order, the outputs will always be lower in the partial order than the intended result even if failures occur. Under the CALM theorem, coordination-freeness means that there are no mechanisms to detect failures, which places us under the same preconditions. Just like the CALM theorem, Gyatso can only guarantee that any outputs are part of the full output, but there is no way to detect when that output is complete.

Rather than restricting programs to monotone operators, Gyatso allows for some program paths to be non-monotone as necessary. This extension aligns with recent work generalizing the CALM theorem beyond relational transducers [124]. For example, Gyatso supports Z-Set [24] collections which can be used to incrementally process insertions and removals to a database. Because the concatenation operator for Z-Sets is not monotone, Gyatso cannot make any guarantees on Z-Set outputs, but still permits them to be used in the program. To convert a Z-Set back to a monotone collection, a developer could use a coordination mechanism (beyond the scope of this chapter) such as a checkpoint to ensure that the Z-Set is complete.

3.6.3 Algebraic Properties for Distributed Systems

Many of the collection types in Gyatso are reminiscent of recent work focusing on algebraic properties critical to replica consistency in distributed systems. In particular, the work on CRDTs [131] and LVars [90] use semilattice data types, which satisfy associativity, commutativity, and idempotence in order to be resilient to network non-determinism. Similar work in the databases community has focused on these three algebraic properties as a new view on consistency [62].

Much of the focus in that work is on *state replication*, a key component of distributed systems, but not the only one that suffers from bugs due to the network and concurrency. Stream-choreographic programming relies on similar algebraic reasoning to enforce eventual determinism, covering not only CRDTs but also systems like MapReduce [50]. Algebraic properties show up in operators such as `fold`; commutativity is needed when the input stream is unordered and idempotence is needed when the input may have duplicates.

A key advantage of the stream-choreographic approach is that the type system can perform more granular reasoning on when certain algebraic properties are needed. For example, a program that uses strictly ordered streams via a protocol such as TCP need not worry about commutativity, and the developer is free to write sequential logic. But when implementing a protocol such as gossip replication with Gyatso, the non-determinism exposed by cluster networking operators will naturally demand the expected properties.

3.7 Summary

In this chapter, we presented Gyatso, a new programming model for distributed systems that combines ideas from stream and choreographic programming to offer strong global guarantees while preserving low-level control. Gyatso extends the dataflow semantics of Flo with location types, which allow developers to specify where streaming operators execute and when streams are moved across the network. Gyatso provides two strong guarantees on distributed execution: eventual determinism when machines are live and monotone outputs when failures are present. These properties are compositional, making it easy to bring operators from Flo and confidently use them to construct large distributed programs. With Gyatso, developers have a strong foundation to express large-scale distributed programs with compositional safety, high-performance, and end-to-end correctness.

Part II

Language Design and Implementation

Equipped with a semantic foundation for correct and performant distributed programming, our next task is to reify these semantics as a full-fledged language and compiler. A broad goal of this thesis is to improve the *accessibility* of distributed systems; to that end creating a brand-new programming language is impractical. Modern developers rely on ecosystems of libraries and tooling that are built up over *decades*, so it is unreasonable to expect them to adopt a language that lacks these comforts.

Instead, we leverage *staged programming* to expose stream-choreographic programming as a Rust framework called **Hydro**. Hydro exposes choreographic locations as regular Rust types, enabling full IDE support, while also supporting zero-cost endpoint projection by staging runtime logic. Hydro follows the semantic guidelines set by Gyatso, but must deal with the realities of practical systems such as timeouts and external APIs. Like Rust provides an unsafe construct for situations where the compiler cannot automatically guarantee memory safety, Hydro provides an unsafe escape hatch for developers to write arbitrary distributed logic.

Hydro offers a paradigm shift in how developers modularize distributed systems. Rust functions in Hydro can encapsulate entire distributed protocols such as two-phase commit and Paxos. Furthermore, common pieces across these protocols, such as quorums, can be shared in a zero-cost manner. In Chapter 4, we introduce the key components of the Hydro framework–its strongly-typed streaming API, staged endpoint projection, and integrated deployment infrastructure–and discuss its implications on distributed programming practices.

Staged programming itself is new to Rust, offering a fundamentally new approach to metaprogramming in the language. Rust has unique challenges for staging, since its macro system runs *before* typechecking which limits the availability of semantic information. We also must carefully consider lifetimes to ensure that staged code is in compliance with the borrow checker. In Chapter 5, we introduce Stageleft, a general-purpose library for staged programming in Rust. Beyond Hydro, Stageleft can be used to implement a variety of staged programming techniques such as partial evaluation and type-safe macro expansion.

Chapter 4

Hydro: a Rust Framework for Modular Distributed Systems

4.1 Introduction

Today's frameworks for distributed systems force a challenging tradeoff between performance, modularity, and correctness. Actor and RPC frameworks like Akka [125] and gRPC [72] provide a low-level interface for building distributed systems. In these frameworks, developers string together units of sequential logic using the network. These frameworks syntactically partition code according to where it is run; this makes it impossible to capture a protocol spanning multiple machines as a reusable function. Furthermore, there are no guarantees on the correctness of the distributed system as a whole—that reasoning is left to the developer.

On the other hand, frameworks like Durable Execution [27], Spark [165], and Flink [30] offer strong correctness guarantees. But they internally use implicit coordination protocols to hide faults and concurrency, without the developer's explicit consent. This leads to poor and unpredictable performance, and prevents developers from faithfully implementing fault-tolerant protocols like Paxos [93] or CRDT gossip [131].

This performance versus correctness gap is reminiscent of the motivations behind Rust [104]. Developers were forced to choose between languages like Java and Go, which guarantee memory safety but incur runtime overhead, or languages like C++, which provide low-level control but lack safety guarantees. Rust's approach is to provide type system mechanisms such as lifetimes [123] that enforce memory safety at compile time. Lifetimes in Rust enable *zero-cost abstractions*, which let developers extract reusable modules without any performance impact.

We apply the same principles to distributed systems. In this chapter, we present **Hydro**, a Rust framework for correct and modular distributed programming. Hydro exposes a *global programming model* where developers can write logic that spans several distributed locations as straight-line code in a single function. Hydro is a "zero-cost" framework that never introduces implicit runtime overhead; it instead relies on the type system to guard against sources of distributed non-determinism.

At its core, Hydro is a stream programming framework with semantic extensions that make it suitable for general-purpose distributed programming. Hydro's API is designed to be similar to existing frameworks like Spark [165] and Rust's built-in iterator API [148], but with semantics that explicitly expose distribution. The most important difference is that streams in Hydro represent *live, asynchronously updated* collections of data placed on *distributed locations*. These locations act as virtual identifiers that represent the machines where the stream will be materialized. Our approach has its semantic foundations in stream-choreographic programming (Chapter 3), where developers implement distributed systems as stream programs that span several machines.

Instead of using runtime protocols to enforce correctness guarantees, we use the type system to reason about network delays and concurrency across machines. These sources of non-determinism are captured in *stream markers*, which are Hydro's equivalent of Rust's lifetimes, but focusing on distributed safety rather than memory safety. By leaving correctness to types, Hydro offers a programming interface that allows low-level control over networking and placement while preserving high-level correctness guarantees.

Hydro is a *standard Rust library* and can be used alongside the existing Rust ecosystem. Key to our approach is the use of *staged programming* [143], which separates the compilation into two phases: one that constructs a global dataflow graph and another that compiles the logic for each distributed location into an optimized Rust binary. The first phase allows Rust's type system to reason about behavior across several machines, since the entire distributed system is captured in a single compilation unit. The second phase leverages Rust's traditional strengths to execute local logic in a safe and performant manner.

There are sometimes situations where the type system does not capture enough information to statically reason about safety properties. In Rust, this happens when interacting with external APIs or manually manipulating pointers, and in Hydro, this can happen when a protocol must intentionally observe non-determinism. Rust offers an *unsafe* mechanism to let developers step into a super-language with fewer memory-safety restrictions. Hydro offers an equivalent construct, which lets developers use non-deterministic operators. In unsafe Hydro logic, it is up to the developer to manually reason about correctness, but this is a one-time cost—once the logic is implemented and tested, it can be shared across systems without any runtime overhead.

While Hydro is exposed as a Rust library, its internal architecture is more similar to a traditional compiler. Programs written with Hydro generate a global dataflow graph that captures the entire system in a custom intermediate representation (IR), which can be further rewritten for instrumentation and optimization. This IR is then projected into individual Rust binaries for each location, which are compiled and launched to execute the computation. By slicing the global program into local units at compile time, we achieve *zero-cost projection*: each output binary contains only the low-level logical logic relevant to its associated distributed location, with performance matching traditional RPC services.

Hydro also explores a new frontier for choreographic programming: *co-locating distributed logic with cloud infrastructure*. Many developers use infrastructure-as-code [13, 61] systems to allocate cloud resources. Hydro takes this a step further, by offering a *type-safe* interface to specify how distributed locations should be mapped to cloud resources. Hydro handles all provisioning and networking configuration, so programs can be deployed in a matter of seconds.

In summary, we make the following contributions:

- We discuss the core architecture of the Hydro framework and show how developers can write global programs in straight-line Rust (Section 4.2)
- We explore how developers can implement distributed system logic by transforming asynchronous streams (Section 4.3)
- We introduce networking operators and show how Hydro guards against sources of nondeterminism through Rust types (Section 4.4)
- We show how developers can step outside Hydro's safety guarantees to implement custom low-level logic, including batched iterative computation (Section 4.5)
- We present a deployment system that lets developers map Hydro locations to cloud infrastructure in a type-safe manner (Section 4.6)

4.2 Architecture and Core Interfaces

Hydro is exposed as a Rust library that provides interfaces for defining distributed locations, writing streaming programs, and deploying them to cloud infrastructure. It can be added to the dependencies of a standard Rust project using the Cargo [147] build system and can be used alongside other Rust libraries. Unlike most libraries, however, Hydro uses a *multi-stage programming* approach. Instead of executing a Hydro program directly on distributed machines, the program is first compiled and run *on the developer's laptop* to capture an IR representing the entire system, which is then used to generate independent Rust binaries for each location.

4.2.1 Staged Programming

Hydro is built around a functional API where developers transform streams with classic operators such as `map`. But these methods do not immediately execute the passed closure when called. Instead, they "quote" the closure and add it to an IR that captures the global dataflow graph. This quoted code includes the Rust source of the closure as well as metadata on its context that allow the closure to be spliced into generated code; we will describe how quoting is achieved in Rust in Chapter 5. The execution of the Hydro program begins by "running" the distributed logic to collect the global dataflow graph, and then proceeds to a deployment phase.

At this point, the dataflow IR is projected onto each distributed location to extract the logic relevant to each machine. The low-level code is generated by "splicing" the closures for the IR nodes placed on that machine (which result in Rust sources for each closure) and stringing them together using DFIR [127], a highly-performant single node dataflow IR and runtime[1]. This results in one Rust source file for each location, which implements the low-level logic for each machine.

[1] DFIR is a component of the broader Hydro ecosystem and is tightly integrated with the high-level Hydro language.

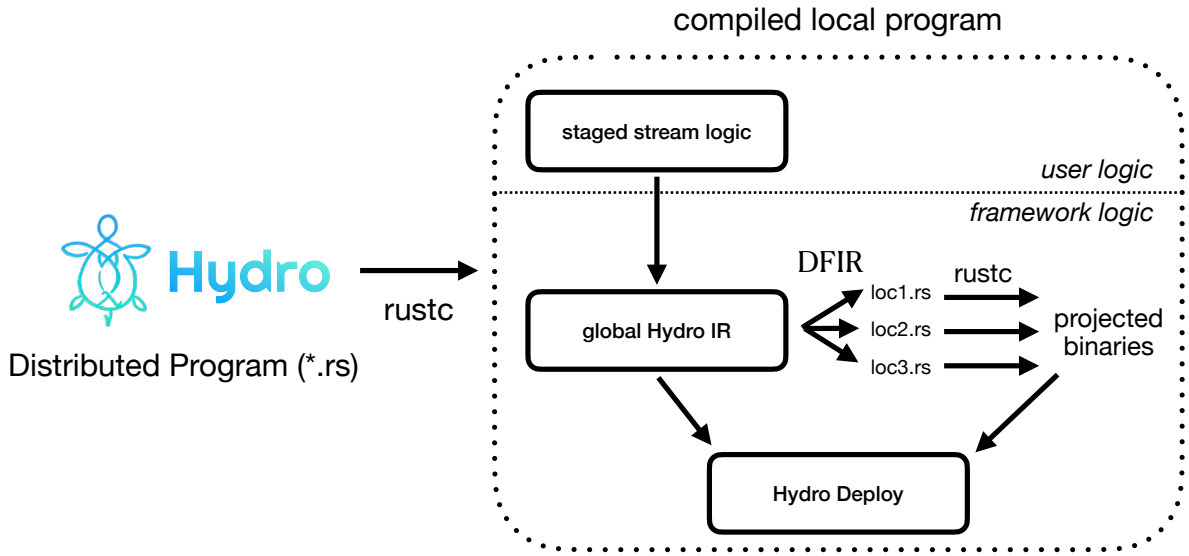


Figure 4.1: The Hydro compilation and deployment architecture. The Hydro program is compiled (represented by the dotted box) and executed locally (solid boxes) to generate the dataflow graph, project to individual Rust binaries, and deploy them.

Each of these Rust sources is then compiled and deployed to the target machines. By separating the final deployed binaries from the original Hydro program, we can offer a high-level API to developers while yielding bare-metal performance. We visualize this multi-stage compilation process in Figure 4.1.

Taking a step back, the staged programming approach means that the APIs exposed to the developer are completely independent of the DFIR code that is generated and executed on distributed machines. This makes it possible to modularize distributed systems with *zero-cost*; abstractions in Hydro logic do not affect the runtime DFIR. Distributed locations, streams, and the IR itself can all be manipulated as standard Rust values without having to jump through special syntax or type system hoops. This is a key difference from existing choreographic languages, which often require esoteric type annotations because the compiler only has a single stage.

Another advantage of the staged approach is it allows for developers to metaprogram the dataflow graph with local configuration. Because the Hydro program generates the global IR when it is executed, any conditionals in the program can be used to change the structure of the IR. While this stops short of true dynamic graphs, as data from the deployed program cannot be used when the IR is locally generated, it offers developers the ability to easily configure monitoring and debugging logic with standard control-flow constructs. This same approach can be used to implement optimizers over the IR (Chapter 7), which are executed in the staged local program.

4.2.2 Creating Locations

Hydro exposes a programming model where developers can place streams on *distributed locations*, which are an abstract representation of the physical machines where the streams will be materialized. Because Hydro is a staged library, locations can be passed around in Hydro programs as standard Rust values that act as opaque handles—they do not expose any information about the target machines.

In any Hydro program, the first step is to acquire a `Location` value, which can then be used to materialize a stream at that location. The `Location` type is a trait which is implemented by the various different kinds of locations in Hydro, such as `Process` which represents a single machine or `Cluster` which represents a cluster of machines. The broad semantics of these different location types follow those of Gyatso (Chapter 3).

To create a location, the developer must first have access to a `FlowBuilder`, which is where the global dataflow graph will be accumulated. When creating a location, the developer provides an arbitrary type parameter that is used as a "tag" to identify it in the type system. For example, calling `FlowBuilder::process<MyLocation>` yields a `Process<MyLocation>`, a Rust struct which can be passed around as usual.

This tag has no semantic meaning, as each location will also be assigned a unique integer identifier, but it helps flag mismatched locations during type checking. Although this approach permits false-negatives where two locations with the same type may have different identifiers, it enables richer composition. For example, a function that locally creates a location can be called multiple times, even though the same type tag will be used in each invocation. We show an example of creating a location in Figure 4.2 in the context of a typical Hydro program structure.

```
struct Leader;

// staged program entrypoint
fn main() {
    let flow = FlowBuilder::new();
    let leader: Process<Leader> = flow.process::<Leader>();

    distributed_logic(&leader);

    // deployment logic...
}

fn distributed_logic(leader: &Process<Leader>) { /* ... */ }
```

Figure 4.2: General structure of a Hydro program involving a process location.

Processes are the simplest type of location in Hydro. They represent a single thread of execution on a single machine, and are useful for roles such as a distinguished leader node, benchmark-

ing server, or a sink for a processing pipeline. For fault-tolerant, scale-out workloads, Hydro also supports *clusters*, which represent a set of machines that all run the same logic concurrently. To create a cluster, a developer can call a similar `FlowBuilder::cluster<T>` API. Clusters are useful for workloads that can be partitioned or replicated in Single-Program-Multiple-Data (SPMD) fashion. We will discuss the semantics of computation on clusters in more detail in Section 4.3.

4.2.3 Global IR Design

As a staged Hydro program (such as `distributed_logic`) is locally executed, the `FlowBuilder` accumulates a global dataflow graph capturing computation across all locations. This graph is stored in a custom intermediate representation (IR) that is designed to be easy to inspect, optimize, and project to runtime binaries.

The Hydro IR is capable of capturing dataflow graphs including cycles, similar to those in Flo (Chapter 2) and Gyatso (Chapter 3). In these formal languages, the streaming program is represented as an expression tree with sequential and parallel composition constructs. This makes the operational semantics easier to formalize, but is not an ideal structure to be manipulated by a compiler since parallel composition makes data dependencies hard to trace. Instead, Hydro uses an abstract data type (ADT) oriented IR design, where dataflow graphs are captured as a tree of operator expressions. We formally describe the core elements of the Hydro IR in Figure 4.3, omitting some operator nodes for brevity.

⟨*IR*⟩ ::= ⟨*Sink*⟩ ; ⟨*IR*⟩ | ⟨*Sink*⟩

⟨*Sink*⟩ ::= CycleSink(⟨*key*⟩, ⟨*Node*⟩) | TeeRoot(⟨*key*⟩, ⟨*Node*⟩) | ForEach(f: ⟨*quote*⟩, ⟨*Node*⟩)

⟨*Source*⟩ ::= CycleSource(⟨*key*⟩) | TeeRef(⟨*key*⟩) |
SourceIter(⟨*Location*⟩, ⟨*quote*⟩) | SourceExternal(⟨*Location*⟩)

⟨*Node*⟩ ::= ⟨*Source*⟩ |
Map(f: ⟨*quote*⟩, ⟨*Node*⟩) | Filter(f: ⟨*quote*⟩, ⟨*Node*⟩) |
Fold(init: ⟨*quote*⟩, acc: ⟨*quote*⟩, ⟨*Node*⟩) | Reduce(acc: ⟨*quote*⟩, ⟨*Node*⟩) |
Chain(⟨*Node*⟩, ⟨*Node*⟩) | Join(⟨*Node*⟩, ⟨*Node*⟩) |
Batch(⟨*Node*⟩, ⟨*Tick*⟩) | AllTicks(⟨*Node*⟩) | CrossSingleton(⟨*Node*⟩, ⟨*Node*⟩) |
Network(serialize: ⟨*quote*⟩, deserialize: ⟨*quote*⟩, to: ⟨*Location*⟩, ⟨*Node*⟩) |
...

⟨*Location*⟩ ::= Process(⟨*key*⟩) | Cluster(⟨*key*⟩) | ⟨*Tick*⟩

⟨*Tick*⟩ ::= Tick(⟨*key*⟩, ⟨*Location*⟩)

Figure 4.3: Grammar defining the core elements of the Hydro IR.

Most nodes in the Hydro IR are for computational operators such as `map` or `fold` which will be included in the projected Rust binaries. These IR nodes store quoted Rust code for the closures passed into the API, such as the transformation function in a `map` operator. The quoted code is stored as Rust AST using the syn library [153], which also provides mechanisms for splicing the tokens into the final generated code. In the IR, the quoted closures are stored as untyped tokens, but the first stage of type checking ensures that the closures will compile when spliced into DFIR.

Hydro exposes constructs for sharing a stream across separate paths (colloquially called diamonds) as well as for cycles spanning multiple locations. Neither of these can be natively represented in a tree structure, so Hydro introduces additional custom nodes that can be used to capture these constructs. The `TeeRef` node can be used to share a stream along multiple paths, with each path holding a reference-counted pointer to the `TeeRoot` IR for the shared stream.

Similarly, the Hydro IR includes `CycleSource` and `CycleSink` nodes which can be used to link the output of one expression tree to the input of another (including itself). The `FlowBuilder` accumulates a forest of Hydro IR nodes, each of which represents either an output of the program or an intermediate result that will be linked to a `CycleSource`. This allows Hydro to represent arbitrary cycles without requiring control-flow constructs such as a fixpoint operator.

The last (and perhaps most unique) node in our IR is the `Network` node. This node captures points in the program where a stream is *moved* across distributed locations. Notably, this is not a sink, and network nodes can be nested in further computation. This allows optimizers to reason about behavior across several machines. During projection, the IR is sliced at network boundaries to identify the nodes specific to each location. To accommodate moving arbitrary Rust types across the network, network nodes are parameterized on quoted snippets that capture serialization and deserialization logic.

```
ForEach(
    f: q!(|v| println!("{}", v)),
    Map(
        f: q!(|v| v * 2),
        Network(
            ..., to: Process("leader"),
            Filter(
                f: q!(|v| v > 2),
                SourceIter(Process("worker"), q!(0..5))))))
```

Figure 4.4: An example of a global program captured in Hydro IR.

To understand the implications of this design, let us walk through a simple distributed program represented in the Hydro IR. In Figure 4.4, we see a simple program that materializes a stream of integers at a worker location, filters them, and sends them to a leader location. The leader location then transforms and prints the values. This entire program can be represented as a single IR expression, with the `Network` node capturing the movement of data to the leader. We use `q!` (which is the Rust macro for quoting) to denote expressions that are quoted in the IR.

In traditional programming languages, we would have two independently compiled IRs (perhaps in LLVM [96]) with opaque system calls to send and receive data over the network. In contrast, our global IR gives us a "birds-eye view" of the entire distributed computation, allowing us to trace data dependencies across locations. This enables opportunities to visualize the distributed system as a whole, as well as to optimize the IR before projection. For example, under an assumption that the `Map` node is deterministic, we could commute it with the `Network` node to push the computation to the worker. We explore optimizations in this vein in Chapter 7.

4.2.4 Zero-Cost Projection

The final piece of the Hydro compilation process is projecting the global IR into Rust binaries for each location. Our goal is for the framework to have zero runtime overhead, which means that the global nature and abstractions of the original Hydro program should not leak into the final binaries. Our staged compiler approach allows us to achieve this by providing an entirely separate code generation process to generate Rust sources for each location, which are compiled as independent single-machine programs.

Each projected binary uses DFIR [127], which is a dataflow language and runtime for single-threaded stream processing. Under the hood, DFIR uses the Tokio asynchronous runtime [40], which enables high-performance networking and interoperability with existing Rust libraries. DFIR comes with its own compiler which lowers the dataflow to imperative Rust code, which processes incoming streams using microbatches. DFIR is based on the semantics of Flo (Chapter 2), which means that operators in the Hydro IR can be directly mapped to DFIR operators without any semantic differences.

```
// worker
node_0 = source_iter(0..5);
node_1 = node_0 -> filter(|v| v > 2);
node_1 -> map(serialize) -> dest_sink(conn_to_leader);

// leader
node_0 = source_stream(conn_from_worker) -> map(deserialize);
node_1 = node_0 -> map(|v| v * 2);
node_1 -> for_each(|v| println!("{}", v));
```

Figure 4.5: The DFIR code generated for the leader and worker locations in Figure 4.4.

We show an example of projecting the Hydro IR from earlier into DFIR programs for each location in Figure 4.5. Programs in DFIR consist of a series of named-pipeline statements, which let developers sequentially compose operators and name the output with a pipeline variable. These variables can then be referred in other pipelines, which enables a concise syntax for building dataflow graphs, including those with cycles.

To generate DFIR projections, we walk the Hydro IR bottom-up while keeping track of which location for each node (this only changes across `Network` nodes). Hydro generates all DFIR projections at once, so this traversal is done in a single pass. Each local IR node (`Map`, `Filter`, etc.) is compiled into a separate pipeline statement.

Network nodes require special logic, since they affect the DFIR generated for *two* locations: the sender and the receiver. On the sender side we use DFIR's `dest_sink` operator, which sends bytes to a network sink. On the receiver side, we use DFIR's `source_stream` operator, which reads incoming data from a Tokio stream. To ensure correct behavior, these operators must be paired at runtime using a network connection.

Unlike distributed frameworks like Spark or Flink, which dynamically place operators at runtime, Hydro offers *ahead-of-time placement*. This means that each projected DFIR program is compiled with *exactly* the operators needed for that location, which enables aggressive inlining across adjacent operators and LLVM optimizations such as auto-vectorization. Each projected binary is a *standard networked service*, which can run independently without any coordination with a global scheduler. Hydro programs can be deployed using standard infrastructure such as Kubernetes [23]; the only runtime component outside the scope of DFIR is the logic to acquire network connections, which we will discuss in Section 4.6.

4.3 Live Collections and Stream Types

Unlike most streaming frameworks, which only offer sequential streams and relations as their core data types, Hydro applies a broad definition of streams that encompasses many kinds of data. This includes sequential streams, unordered sets, and even singletons of single values. In Hydro, these asynchronously updated data types are called **live collections**. Each live collection is tied to a particular *location*, which specifies where the collection is materialized. This allows developers to write distributed programs by moving live collections between locations.

Hydro uses Gyatso (Chapter 3) as a semantic foundation for distributed correctness, so operators that transform live collections must satisfy several correctness properties. In particular, all *safe* operators in Hydro satisfy *eventual determinism*, which ensures that the output collection always resolves to the same value regardless of concurrency and network non-determinism.

In this section, we walk through the core collection types offered by Hydro and the APIs available to transform them. A key innovation in Hydro is the use of *stream markers*, which track sources of non-determinism and enforce safety guarantees through Rust's type system. Rather than relying on staged program analysis, Hydro surfaces distributed safety errors during type-checking, including in IDEs. Because Gyatso already covers formal semantics and DFIR provides operator implementations, we will focus on the API design rather than runtime behavior.

4.3.1 Sequential Streams

The most common live collection in Hydro is the `Stream<T, L, B>` type, which represents a sequential stream with elements of type T. All live collections in Hydro are parameterized

on the location L where they are materialized, which helps surface errors when streams are on mismatched locations. The third type parameter B is a *boundedness* marker (from Flo, Chapter 2) that tracks if the stream is `Bounded` or `Unbounded`. Bounded streams are finite and terminate in finite time, while an unbounded stream may never terminate.

The most straightforward way to create a stream is to use Hydro's `Location::source_iter` API, which reads elements from a potentially-infinite iterator (any Rust type which implements the `Iterator` trait). This API requires staged programming on the *developer-side*; the expression for the iterable must be *quoted* using the `q!` macro. The quoted code is not executed when the Hydro program is run locally to build the dataflow graph, instead it is spliced into the DFIR code for the location where the stream is being materialized.

Even though the quoted code is not run locally, we must ensure that it is typechecked when we compile the staged program. Hydro maintains a global invariant that if the Hydro program compiles, then each location's projected binary will also compile. This means that we must enforce any type constraints present in the projected code while typechecking the Hydro program.

When quoting a piece of code with the `q!` macro, the returned Rust value implements a trait called `QuotedWithContext`, which is parameterized on the type of the expression being quoted as well as a context parameter which is used to expose location-specific APIs. This first type parameter may be subject to additional constraints; in this case it must implement the `IntoIterator` trait so that it can be passed to the `source_iter` DFIR operator.

We show the type signature of the `source_iter` API in Figure 4.6, along with an example of using it to create a stream of integers. At compile time, the `E` type parameter will capture the type of the quoted expression, which lets us enforce the necessary type constraints. This is important for a good developer experience, as it allows IDEs to provide type hints and error messages inside the quoted code rather than treating it as an untyped set of tokens.

```
trait Location {
    fn source_iter<T, E: IntoIterator<Item = T>>(
        &self,
        e: impl QuotedWithContext<E, Self>,
    ) -> Stream<T, Self, Unbounded>;
}

fn distributed_logic(worker: &Process<Worker>) {
    let numbers = worker.source_iter(q!(0..5));
}
```

Figure 4.6: The type signature of the `source_iter` API and an example using it.

All live collections in Hydro are exposed to the developer as regular Rust values, which means they can be passed around in functions subject to the usual Rust ownership rules. Each instance of the `Stream` struct (as well as any live collection) stores a Hydro IR expression tree that captures

the dataflow graph whose output is represented by the stream. In the case of `source_iter`, this IR is simply a `SourceIter` node with the quoted code as its parameter.

Operators to transform streams are offered as methods on the `Stream` struct which return a new stream. *Computational* operators, such as `map` and `filter`, output a stream on the same location as the input; we will discuss networking operators in Section 4.4. Each operator in Hydro takes *ownership* of the input stream, which means that the input stream is consumed and cannot be used after the operator is called. The output stream is a new instance of the `Stream` struct, which contains a new IR tree that wraps the input IR with the additional operator.

Hydro offers classic functional operators on streams with standard type semantics, but the closures passed into these operators are quoted and stored in the IR. Like in Flo, we are able to pass down boundedness markers using Rust generics. We show an example of the type signature for `filter` in Figure 4.7, along with an example of using it and the Hydro IR of the result.

```
impl <T, L, B> Stream<T, L, B> {
    pub fn filter<F: Fn(&T) -> bool>(
        self, f: impl QuotedWithContext<F, L>
    ) -> Stream<T, L, B>;
}

let numbers = worker.source_iter(q!(0..5));
let filtered_numbers = numbers.filter(q!(|v| v > 2));

Filter(
    f: q!(|v| v > 2),
        SourceIter(Process("worker"), q!(0..5)))
```

Figure 4.7: The declaration for `Stream::filter`, an example of it, and the resulting Hydro IR.

Because each operator takes ownership of the previous stream, we can chain them together in a single expression and reduce the syntactic overhead of creating intermediate variables. Similarly, we can extract common logic into functions that take streams as parameters, including with generic type parameters that further generalize the shared behavior. This extends to the core Hydro API itself, where some operators are simply wrappers around other stream APIs. For example, the `flatten` operator simply calls `flat_map` under the hood.

Note that all these abstractions *do not* change the IR tree in any way, which means that the projected Rust code is identical regardless of abstractions and indirection in the original code. Our staged programming architecture enables such zero-cost abstractions, which let developers modularize their distributed systems without any performance cost.

In many distributed programs, we will want to share a stream across multiple downstream computations. This is *not* a zero-cost operator, since each element of the stream must be copied for each consumer. Furthermore, we cannot duplicate the IR tree, since that would represent independently re-calculating the stream rather than sharing a common result.

As we mentioned earlier, Hydro IR includes the `TeeRoot` and `TeeRef` IR nodes to facilitate subgraph sharing. The `TeeRoot` node is a sink that captures the output of a stream, while the `TeeRef` node stores a reference-counted pointer to the `TeeRoot` node. We expose this construct in Hydro using the standard `Clone` trait from Rust, which requires the elements of the stream to also implement `Clone`.

When a stream is cloned, we wrap its IR in a `TeeRoot` node and create a reference-counted shared pointer so that it is only compiled once. The original stream then replaces its local IR with a `TeeRef` node, and the returned stream also uses a `TeeRef`; both of these will be compiled to consume copies of elements from the shared stream. We show an example of this transformation in Figure 4.8.

```
let filtered_clone = filtered_numbers.clone();
// filtered_numbers can still be used

// before clone
Filter(
    f: q!(|v| v > 2),
    SourceIter(Process("worker"), q!(0..5)))

TeeRef(0xABC) // filtered_numbers and filtered_clone

0xABC: TeeRoot(Filter( // shared pointer
    f: q!(|v| v > 2),
    SourceIter(Process("worker"), q!(0..5))))
```

Figure 4.8: The IR for a stream before and after cloning it with the `Clone` trait.

To finish our discussion, we will explore how to use streams to output results. Streams in Hydro are lazily compiled and will be ignored if their outputs are not used. In particular, IR nodes will only be compiled into projected binaries if they are reachable from a root in the `FlowBuilder`. To create such a root, we must use a Hydro API like `for_each` or `dest_sink`, which consume the stream by performing side effects or emitting outputs over the network. When any of these APIs are called, the input stream is consumed, its IR tree is wrapped in a sink node, and the IR tree is added to the roots of the global `FlowBuilder` IR.

4.3.2 Stream Markers

The streams we have looked at so far have standard sequential semantics that guarantee exactly-once, in-order delivery of elements. This is a common assumption in many streaming frameworks, but it is not sufficient for all distributed applications. In particular, we need to be able to express streams with *retries* as well as *unordered* streams, which are useful for applications that provide custom fault tolerance behavior and can handle concurrent clients.

In Flo and Gyatso, these semantics are captured through alternate collection types that model various forms of non-determinism. Hydro takes a similar approach, but instead of introducing new Rust types for each stream variation, we add optional type parameters to the `Stream` type that track forms of non-determinism. These *stream markers* reduce duplication of APIs across Hydro collections and make it possible for developers to abstract over the collection type. We note that there is an inherent subtyping relationship between different stream markers—an ordered stream can always be passed to a function that tolerates streams with unordered elements. By using type parameters, we can express this relationship in Rust's type system.

Streams in Hydro have two optional type parameters that can be used to track non-determinism: one for ordering and one for retries. The ordering type parameter can have one of two values: `TotalOrder` indicates that the elements have a deterministic order (the default) and `NoOrder` means that the elements may be shuffled. Similarly, the retries type parameter can be `ExactlyOnce` for deterministic cardinality (default) or `AtLeastOnce` for non-deterministic duplication. The matrix of these markers captures a wide range of distributed semantics:

- **Strict (TotalOrder + ExactlyOnce)** streams are ordered and have exactly-once delivery semantics. This is the default stream type in Hydro.
- **Unordered (NoOrder + ExactlyOnce)** streams are unordered and have exactly-once delivery semantics. This is useful for applications that can tolerate out-of-order data arrival, such interleaved messages in relational queries.
- **Retry (TotalOrder + AtLeastOnce)** streams are ordered and have at-least-once delivery semantics, where elements are clustered with consecutive retries. This is useful for applications that can tolerate duplicate data, such as idempotent request handlers.
- **Unordered-Retry (NoOrder + AtLeastOnce)** streams are unordered and have at-least-once delivery semantics. This is useful for applications that can tolerate out-of-order data arrival and duplicates, such as in CRDT gossip [131].

Many stream operators in Hydro that transform elements one-by-one, such as `map` and `filter`, simply pass through their stream markers to the returned stream using generic type parameters in the function signature. We must take special care to ensure that these operators indeed preserve determinism modulo these markers, as in Flo. Other operators can constrain these markers in their type signature if necessary for determinism. For example `enumerate`, which assigns an index to each element, requires the input stream to be a strict stream since both ordering and retries affect the indices. We show an example of these signatures in Figure 4.9.

Because Rust does not support subtyping relationships between struct types (aside from lifetimes), we instead offer guaranteed-safe conversions between stream variants using the `Into` trait. Hydro permits conversion to a stream with strictly weaker markers, from `TotalOrder` to `NoOrder` and from `ExactlyOnce` to `AtLeastOnce`. Although this requires explicit conversions, Rust tooling can automatically suggest them when available to reduce developer friction.

```
impl <T, L, B, O, R> Stream<T, L, B, O, R> {
    pub fn map<F: Fn(T) -> U, U>(
        self, f: impl QuotedWithContext<F, L>
    ) -> Stream<U, L, B, O, R>;
}

impl <T, L, B> Stream<T, L, B, TotalOrder, ExactlyOnce> {
    pub fn enumerate(self) ->
        Stream<(usize, T), L, TotalOrder, ExactlyOnce>;
}
```

Figure 4.9: The type signatures of `map` and `enumerate` with stream markers.

4.3.3 Singletons and Optionals

A key feature of the formal semantic in Flo or Gyatso is extending the notion of streams to include many other data types, including those that do not have multiple elements but are still asynchronously updated. A key pair of collection types in Hydro that use this idea are the `Singleton<T, L, B>` and `Optional<T, L, B>` types.

A singleton in Hydro represents a single value of type T that is asynchronously updated. Any updates to the singleton replace the previous value, so this live collection acts as a mutable variable. Note that due to asynchronous processing semantics, the singleton may "skip" intermediate states—the eventual determinism properties from Gyatso only apply to the settled value. The singleton type is parameterized on the same location and boundedness markers as streams, but it does not have any markers since it only has one element.

The simplest way to create a singleton is to use the `Location::singleton` API, which takes a quoted expression and returns a singleton which will have a fixed value. This can be used to load constants into a Hydro program or to set up initial values for an iterative loop. But a singleton with a fixed value is not particularly interesting, since it is equivalent to a regular Rust value.

The real power of singletons comes from operators that create them from an asynchronously updated input. The most common operator that emits a changing singleton is `Stream::fold`, which accumulates a strict stream of values into the output singleton. This operator takes an initial value and a function that combines the current value with a new value, both of which must be quoted. As the operator processes incoming elements, it updates the singleton with the new value. We show the type signature of `Stream::fold` in Figure 4.10, along with an example of using it to sum a stream of integers.

Like streams (and all live collections), singletons have a boundedness type parameter which tracks if the singleton will resolve to a fixed value in finite time. Our declaration for `fold` passes through the boundedness of the input stream to the output singleton. If the input stream is bounded, the semantics of the output are clear: it will resolve to the final value of the accumulator once the stream terminates. But a less intuitive fact is that if the input stream is unbounded, the output singleton will also be unbounded and may never resolve to a final value.

```rust
impl <T, L, B> Stream<T, L, B> {
    pub fn fold<A, F: Fn(&mut A, T)>(
        self,
        init: impl IntoQuotedMut<'a, A, L>,
        comb: impl IntoQuotedMut<'a, F, L>,
    ) -> Singleton<A, L, B>;
}

let numbers = worker.source_iter(q!(0..5));
let sum = numbers.fold(q!(0), q!(|acc, v| *acc += v));
```

Figure 4.10: The type signature of the `Stream::fold` operator and an example using it.

What are unbounded singletons good for? They can be used to represent mutable variables that are updated by an infinite stream of values. For example, we can use an unbounded singleton to represent the current value of a counter that is incremented by a stream of events, or the current wall-clock time on a machine. *Observing* an unbounded singleton is disallowed under the core safe API of Hydro, since this is not deterministic. But in many cases developers can opt into an unsafe API to observe a snapshot of the singleton; we will discuss this further in Section 4.5.

A closely related type to singletons is the `Optional<T, L, B>` type, which represents a value of type T that may or may not be present. This is useful for representing optional values in a dataflow program and conditionally enabling or disabling operators. Like `Stream::fold`, an optional can be created with `Stream::reduce`. This operator does not take an initial value and instead combines elements from the input pairwise, so the output is empty until at least one element arrives in the input.

4.3.4 Algebraic Constraints

The aggregation operators we have discussed so far, `fold` and `reduce`, both require *strict streams* as input with deterministic ordering and no duplicates. In many cases, we will want to use these operators on streams that are not strict, such as unordered or retry streams. But the semantics of these operators are not safe over such streams, since the non-determinism in the input leaks into the output. For example, if we reduce an unordered stream of strings by concatenating them, the output is not guaranteed to resolve to a deterministic value.

To aggregate over non-strict streams, Hydro provides a set of operator variants that require the Rust closures passed to them to satisfy certain *algebraic properties*. These properties are used to ensure that the non-determinism present in the output does not affect the output. Algebraic properties bring Hydro into alignment with other work on distributed state management, such as CRDTs [131], and make it possible to safely implement patterns like MapReduce [50].

Let us begin with unordered streams, which have deterministic cardinality but not order. For a fold over an unordered stream to produce a deterministic result, the aggregation function passed to it must be *commutative*, so that reordering the elements does not affect the result.

Hydro exposes this through a separate `Stream::fold_commutative` operator, which requires the developer to pass a commutative closure. This variant is a *zero-cost abstraction*—the IR and generated DFIR are *identical* to `Stream::fold`, and the runtime logic for applying the aggregation does not change.

Aggregations over unordered streams are very common in bulk data processing systems. For example, we may have a cluster of parallel workers that produce a local aggregation that is sent over the network to a leader. The leader will receive these values in non-deterministic order, so Hydro's type system will enforce that the developer calls `fold_commutative` on the stream.

Similarly, if a stream may have non-deterministic duplicates, an attempt to `fold` it is not safe and will result in a type error. Consider a simple aggregation that counts the number of elements in a stream. If the input stream is a retry stream, the count may be non-deterministic since the same element may be counted multiple times. To ensure determinism, the aggregation function must be *idempotent*. Like commutativity, Hydro exposes a `Stream::fold_idempotent` operator that requires the developer to pass an idempotent closure.

In its current implementation, Hydro uses separate function names for each of these variants to make it clear to the developer which algebraic properties are required. It is up to the developer to ensure that their logic indeed satisfies these properties, typically through detailed inspection and code review. We plan to expand this API in the future to support automated verification of these properties to reduce developer effort and improve safety guarantees.

4.3.5 Collections on Clusters

So far, all our examples have used `Process` locations, which represent a single machine. A more interesting case is when we place a live collection on a `Cluster` location, which represents a *set of machines*. When this happens, each member of the cluster will compute an *independent* value, following single-program-multiple-data (SPMD) semantics. This means that the same code is run on each member of the cluster, but the input data may be different. When applying a local operator to a stream on a cluster, the operator is executed concurrently on each member of the cluster, and there are no guarantees on when each member will finish processing.

Hydro *does not* provide any guarantees on consistency *across* cluster members, but rather guarantees that each instance of a collection will resolve to an independent deterministic value. This low-level approach to cluster semantics gives developers a lot of flexibility to implement distributed protocols. For example, upstream data can be replicated or partitioned across cluster members, with each member performing an aggregation over its independent inputs. Likewise, in quorum protocols, each member of the cluster can make an independent vote decision.

4.4 Networking and Scaling

Equipped with a set of asynchronous collections and mechanisms for modeling sources of non-determinism, we can finally explore how to use Hydro to build *distributed* programs by sending data across locations. All the operators we have seen so far are *local* operators, which means

that their input and output streams have the same location. To allow Hydro programs to span several machines, we provide a set of *networking* operators that *move* streams between locations by using network channels.

Our semantics for networking operators are based on Gyatso (Chapter 3), which formalizes network communication as operators with non-deterministic semantics that model the behavior of the network. This means that network operators can be chained just like other local operators, even within a single function. This is a key difference from other distributed programming frameworks, which typically require each networked unit to be in a separate function or module. Hydro's approach makes it possible for developers to share logic that *includes* networking, such as entire protocols like Paxos [93], in a *single Rust function*.

4.4.1 Moving Streams

Let us begin with the simplest networking operator, moving a stream from one process to another. Given a strict stream of elements T, we can use the `Stream::send_bincode` API to move the stream to another location. This API internally uses the bincode [146] serialization library, and uses a standard TCP socket to preserve strict ordering without any duplication. Like all other operators in Hydro, `send_bincode` consumes the input stream as an owned value, so this API can be chained with downstream logic seamlessly. We show the type signature of `send_bincode` in Figure 4.11, along with an example of using it to create the distributed program in Figure 4.4.

```
impl <T: Serialize + DeserializeOwned, L, B> Stream<T, Process<L>, B> {
    pub fn send_bincode<L2>(
        self, dest: &Process<L2>,
    ) -> Stream<T, Process<L2>, Unbounded>;
}

fn distributed_logic(worker: &Process<Worker>, leader: &Process<Leader>) {
    worker.source_iter(q!(0..5))
          .filter(q!(|v| v > 2)) // Stream<usize, Process<Worker>, ...>
          .send_bincode(leader) // Stream<usize, Process<Leader>, ...>
          .map(q!(|v| v * 2))
          .for_each(q!(|v| println!("{}", v)));
}
```

Figure 4.11: The type signature of the `send_bincode` API and an example using it.

Because the Hydro IR natively captures dataflow programs that span locations, implementing this operator follows the same structure as the local operators. The `send_bincode` operator wraps the input IR in a `Network` node. Because the type T is known at compile time, we can generate the appropriate serialization code entirely during staged execution, so that the IR contains the

entire serialization logic. This is another example of a zero-cost abstraction; there is no runtime overhead to look up the appropriate serialization code, as it is compiled entirely ahead-of-time.

During projection, the `Network` node is compiled to a pair of DFIR operators on the sender and the recipient. The sender first serializes each element, and then performs a system call to send the data over a TCP socket. The receiver performs the inverse operation, deserializing the data and then inserting it into the stream. Because the compilation process for both sides is specified by a single IR node, we can ensure that the serialization and deserialization logic is paired correctly. This helps avoid common pitfalls in distributed programming such as mismatched serialization formats or data types.

4.4.2 Cluster Networking

Next, we introduce network operators that allow for communication with clusters, which represent sets of machines all running an identical program. Without clusters, all communication was *one-to-one*. But with clusters, we now have to consider the rest of the matrix: one-to-many, many-to-one, and many-to-many.

To let programs in Hydro decide which members of a cluster a value should be sent to, and identify which cluster member was a sender, we introduce the concept of **cluster IDs**. These are unique identifiers for each member of a cluster that are *only known at runtime*. Because Hydro programs are staged, this means that user programs cannot interact with cluster IDs at the top-level—their usage is restricted to inside the quoted expressions.

Cluster IDs in Hydro are exposed as a type-safe `ClusterId<L>` type, which is a wrapper around a `u32` value. At runtime, the struct is fully erasable to just the integer ID, so it does not have any impact on performance. But by using a wrapper type, we can avoid common mistakes such as using mismatched IDs from different clusters, as each ID has the same type tag as the cluster it belongs to. To support applications where an ID must be manually crafted, such as when deriving IDs from a hash function, we also provide a `ClusterId::from_raw` API that takes a `u32` value and returns a cluster ID. When using this API, the developer must ensure that the ID is valid; any messages sent to an invalid ID will be dropped at runtime.

When sending data to a cluster, the source stream must contain `(ClusterId<L>, T)` data pairs, where the type tag for the cluster ID matches the recipient cluster. At runtime, the sender will maintain independent connections to each cluster member and send each element to one of these connections according to the ID. When receiving data from a cluster, the receiver will interleave all incoming messages from cluster members into a single stream. Because cluster members execute concurrently, the order of this interleaving is non-deterministic, so the output stream will be *unordered*. Each element of this stream will be a `(ClusterId<L>, T)` pair, where the cluster ID is the ID of the sender.

Because cluster membership (and therefore the set of valid cluster IDs) is only known at runtime, Hydro provides a special API (`Cluster<L>::members()`) that returns a *quoted* set of cluster IDs. Developers are free to use this set in their quoted streaming logic, which makes it possible to implement many common distributed patterns without special support from Hydro. For example, we show in Figure 4.12 how to implement a broadcast by taking the cross-product

```rust
pub fn broadcast_elements<T: Clone, L1, L2>(
    data: Stream<T, Process<L1>, Unbounded>,
    cluster: &Cluster<L2>,
) -> Stream<T, Cluster<L2>, Unbounded> {
    let ids = cluster.members(); // impl Quoted<&Vec<ClusterId<L2>>, _>
    data.flat_map(q!(|v|
        (ids /* &Vec<ClusterId<L2> */)
            .iter().map(|id| (id, v.clone()))
    )).send_bincode(cluster)
}
```

Figure 4.12: A reusable function that broadcasts elements to all members of a cluster.

of elements with the set of members. This is not a core operator in Hydro, but rather a zero-cost abstraction that any developer could write.

```rust
fn reduce_from_cluster(
    data: Stream<usize, Cluster<Worker>, Unbounded>,
    leader: &Process<Leader>,
) -> Singleton<usize, Process<Leader>, Unbounded> {
    data
        .send_bincode(leader)
        // Stream<(ClusterId<Worker>, usize), Process<Leader>, ..., NoOrder>
        .map(q!(|v| v.1)) // drop the ID
        .fold_commutative(q!(0), q!(|acc, v| *acc += v))
}
```

Figure 4.13: A simple example of receiving a stream from a cluster and reducing it.

When sending data from a cluster to a process, we use the same `send_bincode` API, but the output is an unordered stream of `(ClusterId<L>, T)` values. The unordered marker requires developers to be more careful about their downstream logic. For example, we cannot use `fold`; instead we must use `fold_commutative` to ensure that the aggregation is deterministic. This aligns with systems like MapReduce [50], where the reducer must be commutative. We show a simple example of this in Figure 4.13. The `send_bincode` operator can also be used for many-to-many communication, where both the input and output have cluster IDs.

4.5 Unsafety and Nested Graphs

So far, all the Hydro programs we have looked at are strictly deterministic. Within the core, "safe" API provided by Hydro, all composition of operators is guaranteed to preserve end-to-end determinism, even in the face of networking logic. Although most pieces of complex distributed systems can be written in the safe language, there are times when developers must step out of these boundaries to implement custom low-level logic or opt into intentional non-determinism. This two-layer language is directly inspired by Rust, which has a core language that guarantees memory safety but additional APIs available in *unsafe* code that allow for logic that the compiler cannot reason about. In fact, Hydro uses the same `unsafe` keyword as Rust to denote its non-deterministic APIs.

In Rust, unsafe code can be written in two places: unsafe functions and unsafe blocks[2]. Unsafe functions can only be called from another unsafe context to ensure compositional safety. But unsafe blocks can be used inside safe functions when the developer knows that the overall function will always be safe. We show an example of both constructs in Figure 4.14.

```
// human asserted end-to-end safety
fn unsafe_inside_safe(...) {
    // ... (only safe APIs)
    unsafe {
        // ... (safe and unsafe APIs)
        unsafe_function(...);
    }
}

unsafe fn unsafe_function(...) {
    // ... (safe and unsafe APIs)
}
```

Figure 4.14: An example of using unsafe code in Hydro.

Hydro uses these same constructs to allow developers to opt into non-deterministic APIs. An unsafe function in Hydro captures a piece of logic whose outputs are not guaranteed to be eventually deterministic with respect to the inputs, or exposes non-deterministic side effects. An unsafe block inside a safe function, on the other hand, can be used to capture a piece of logic which the Hydro incorrectly flags as non-deterministic. In this case, the developer must manually assert that determinism is preserved. These constructs give Hydro developers more flexibility while preserving local, compositional guarantees.

[2] Recent versions of Rust encourage developers to use unsafe blocks even inside unsafe functions, in order to be more explicit about which pieces of the function are unsafe.

4.5.1 Unsafe Operators

Alongside the core API, live collections in Hydro also provide unsafe APIs that allow developers to perform non-deterministic transformations. Although these operators do not satisfy eager execution, the output collections can be used in a safe context with eventual determinism applying *from that stream onwards*.

This *localized unsafety* mirrors Rust and is a key advantage of Hydro's design, since it allows developers to tightly scope use of unsafe APIs and more carefully inspect them in code review. Furthermore, when applying techniques such as deterministic simulation testing [107, 145], we only need to inject non-determinism into unsafe parts, since the rest of the program is guaranteed to be deterministic regardless of scheduling.

As we noted earlier, there is no way to safely observe the value of an unbounded singleton, since it will continually change over time. One solution is to sample the singleton periodically, transforming it into an unbounded stream of sample values. This is clearly non-deterministic as the values pushed onto the output stream depend on when the samples are taken. But such an API can be useful for monitoring real-time analytics or benchmarking results.

Hydro offers a `Singleton::sample_every` API that samples an unbounded singleton at a fixed period. Because this API is unsafe, it can only be used inside unsafe functions or blocks. Because multiple samples may be taken without the singleton's value changing, the output is a retry stream with the `AtLeastOnce` marker. We show the function declaration for this API as well as an example of using it in Figure 4.15.

```
impl <T, L> Singleton<T, L, Unbounded> {
    pub unsafe fn sample_every(
        self, period: impl QuotedWithContext<Duration, L>
    ) -> Stream<T, L, Unbounded, TotalOrder, AtLeastOnce>;
}

unsafe fn live_sum(input: Stream<usize, Process<Leader>, Unbounded>) {
    let sum = input.fold(q!(0), q!(|acc, v| *acc += v));

    let samples = unsafe {
        sum
            .sample_every(q!(Duration::from_secs(1)))
            .assume_exactly_once() // duplicates are meaningful
    };

    samples.for_each(q!(|v| println!("Sum: {}", v)));
}
```

Figure 4.15: The type signature of the `sample_every` API and an example using it.

Beyond unsafe operators, Hydro also provides unsafe APIs for manipulating stream markers outside the existing subtyping. This can be helpful when details of the protocol logic result in an ordering that the Hydro type system cannot guarantee. For example, we can convert an unordered stream into a strict stream with the `Stream::assume_ordered` API. This API is unsafe because it requires the developer to manually assert that the input stream indeed has a deterministic order.

Similarly, we can convert a stream with non-deterministic retries into a strict stream with the `Stream::assume_exactly_once` API. We use this in the figure above, since `Stream::for_each` requires a strict stream to ensure deterministic side effects. These APIs are particularly useful when implementing coordination protocols like Paxos, whose job is to generate a strict stream of log elements from internal logic filled with non-deterministic ordering and retries.

4.5.2 Ticks and Temporal Co-incidence

The other key use of unsafety in Hydro is to allow for *atomic, iterative* processing of streams in a batched execution model. Contrary to typical use, batching is *not needed* for performance reasons. DFIR already uses a batched execution model, so regular Hydro operators will already compile down to a batched implementation. Instead, batching is used to enable *temporal co-incidence*, where incoming requests can be aligned with snapshots of other asynchronously updated state, and custom iterative computations. The former is particularly useful for consumers of eventually consistent state, which intentionally observe non-deterministic snapshots.

To create an iterative loop, Hydro provides a third location type called a `Tick`, which represents a single iteration of a local loop. Each process or cluster in Hydro can have multiple tick locations, each of which corresponds to an independent loop which is free to iterate asynchronously from other loops. Ticks are exposed with the `Location::tick` API, which returns a location for a newly created local loop.

Ticks implement the `Location` trait, so the existing APIs for creating streams and singletons can be used with them. The key difference is that the tick location is *not* a physical location, but rather a logical location that represents a single iteration of a loop. This means that the tick location does not have network semantics, and cannot be used to send or receive data.

Creating a tick is a safe API. But injecting data into a tick often involves unsafe logic. The most common way to move data into a tick is to use the `Stream::tick_batch` API, which takes a stream and a tick location and returns a new stream that is "batched" into the tick. On each tick, a non-deterministic number of elements from the input stream are sent to the tick location, where they can be processed as a *bounded* stream located on that tick. This API is unsafe because the boundaries between these batches are not guaranteed to be deterministic.

To show an example of temporal co-incidence, let us modify the `live_sum` function from before to allow queries to the current sum, instead of just printing this. Because the sum is an unbounded singleton, we cannot directly read from it when responding to queries since the value is asynchronously updated. Instead, we create a new tick where we will temporally align a batch of incoming queries with a snapshot of the sum.

To take a snapshot of the unbounded singleton, we use the `Singleton::tick_snapshot` API, which returns a new *bounded* singleton inside the tick that will have the most recent value of

the outer singleton. To respond to the queries, we use `Stream::cross_singleton`, which takes in a *bounded* singleton and pairs it with each value in the input stream. Finally, to emit the query results, we need to exit the loop. We do this with the `Stream::all_ticks` API, which converts a bounded stream inside a tick into an unbounded stream outside the tick by emitting each iteration's output in a streaming manner. We show this end-to-end example in Figure 4.16.

```
unsafe fn live_sum_query(
    leader: &Process<Leader>,
    input: Stream<usize, Process<Leader>, Unbounded>,
    queries: Stream<(), Process<Leader>, Unbounded>,
) -> Stream<usize, Process<Leader>, Unbounded> {
    let sum = input.fold(q!(0), q!(|acc, v| *acc += v));

    let tick = leader.tick();
    let sum_snapshot = unsafe { sum.tick_snapshot(&tick) };
    let queries_batch = unsafe { queries.tick_batch(&tick) };

    queries_batch
        .cross_singleton(sum_snapshot)
        .map(q!(|(_, v)| v)) // drop the query
        .all_ticks() // one output per query
}
```

Figure 4.16: An example of using a tick to align a batch of queries with a snapshot of the sum.

Note that *after* batching inputs into the tick, the remaining APIs are all safe because they produce deterministic outputs *with respect to their inputs*. This makes it easier for developers to focus on the sources of non-determinism, rather than having to reason about the entire program. For example, in an implementation of Paxos in Hydro, there are exactly 15 unsafe blocks, all of which correspond to critical invariants in the protocol.

The other use of ticks in Hydro is to enable custom iterative processing. Streams and singletons inside a tick can be sent to the next iteration of the loop by using the `Stream::defer_tick` API, which operates over bounded streams located on a tick. Although this API has an identity type signature, its output effectively represents the value of the input on the *previous tick*. This operator is useful for implementing low-level iterative logic such as counting quorums with garbage collection, where the existing votes are preserved across iterations.

Finally, developers can use ticks to ensure atomic processing of local state, critical for coordination protocols like Two-Phase Commit. In such systems, machines must commit data to their local state *before* acknowledging the request over the network. In asynchronous Hydro semantics, this is not possible since the logic to update the state may run after the acknowledgement is sent. But by using a tick, we can ensure that the local state is updated before the network logic is executed, since all results in a tick will be computed before any outputs are released.

4.6 Type-Safe Deployments

Implementing a distributed program is just one half of the story; deploying it is equally complex. Especially in the world of cloud computing providers and multi-cloud architectures, it is easy to make mistakes such as misconfiguring network ports or launching unnecessary virtual machines. We believe that deployments are *just as important* to the success of distributed programming as the computational layer, since developers need to confidently update subcomponents without worrying about their deployment logic going out-of-sync.

Hydro aims to solve this by providing an integrated layer for mapping the *virtual* locations in a dataflow program to *physical* deployment plans in the form of Terraform specifications [61]. By generating the deployment logic *from* the dataflow graph, developers can rest assured that their program will execute exactly as they intended, while preserving the ability to make decisions about machine types and datacenter selection.

Because Hydro uses a staged programming model, the deployment logic sits right next to distributed program and is run alongside the staged execution on the developer's machine. This means that developers benefit from IDE support such as jump-to-definition to know which logic will be run on deployed machines, and type-safety to ensure that the deployment logic appropriately covers all distributed locations. Post-deployment, Hydro goes a step further to provide type-safe mechanisms for interacting with launched nodes. This is a key difference from other programming models, which typically require developers to write deployment logic in a separate language or framework.

4.6.1 Hydro Deploy

The main interfaces for configuring cloud resources are provided by *Hydro Deploy*, a separate library that exposes Rust APIs for configuring virtual machines and networks on common cloud providers including Google Cloud Platform and Microsoft Azure. As a separate library, Hydro Deploy can be used independently of Hydro to deploy cloud applications from Rust. For example, it has been used in prior work to benchmark distributed protocols written in Dedalus [38].

Hydro Deploy manages the entire lifecycle of a distributed program, from creating the virtual machines to deploying the binaries and managing service discovery. The library can be used to monitor applications once they have been deployed and provides APIs for safe cleanup of resources. This makes it particularly useful for running benchmarking experiments, where various configurations of a distributed system can be deployed from a single Rust program.

The primary APIs in Hydro Deploy revolve around configuring virtual machines. Hydro Deploy provides *type-safe* APIs specific to each cloud provider that let developers configure the machine type, operating system, and region without leaving the Rust program. Once these machines are configured, the developer can provision them with a single call to `Deployment::deploy`. We show a simple example of using Hydro Deploy to launch a single virtual machine in Figure 4.17.

Under the hood, Hydro Deploy uses Terraform [61], an infrastructure-as-code (IaC) tool that uses a declarative JSON-like language to specify cloud resources. Hydro Deploy generates a Terraform file that describes the resources requested in Rust, which is then passed to the Terraform

```rust
let mut deployment = Deployment::new();
let vpc = Arc::new(RwLock::new(GcpNetwork::new("gcp-project", None)));
let vm = deployment
    .GcpComputeEngineHost()
    .machine_type("e2-micro")
    .image("debian-cloud/debian-11")
    .region("us-west1-a")
    .network(network.clone())
    .add();

deployment.deploy().await?;
```

Figure 4.17: A simple example of using Hydro Deploy to launch a single virtual machine.

CLI to handle the actual provisioning. By using Terraform, Hydro Deploy can take advantage of the existing ecosystem of Terraform providers and modules, making it easy to integrate with other cloud providers and stay up-to-date as the underlying cloud APIs change.

The next piece of Hydro Deploy is to manage network connections between services deployed to these machines. Developers can define a service by providing a path to a Rust binary, which Hydro Deploy will compile and upload to the configured virtual machine. Services in Hydro Deploy can be linked together with named network connections in a declarative manner, so that the underlying Terraform configuration will automatically create the necessary firewall rules and network interfaces. We show a simple example of this in Figure 4.18.

```rust
let service1 = deployment.add_service(RustCrate::new("my_service1", vm1));
let service2 = deployment.add_service(RustCrate::new("my_service2", vm2));

service1.get_port("out_port").send_to(service2.get_port("in_port"));

deployment.deploy().await?; // provision the machines
deployment.start().await?; // performs handshake and starts the services
```

Figure 4.18: An example of using Hydro Deploy to launch networked services.

Once the services are deployed, Hydro Deploy performs a simple service discovery step to establish network channels by performing a handshake over standard input/output[3]. This is a key difference from other IaC tools, which typically require developers to manually configure network connections between services.

[3] In the future, we plan to replace this mechanism with more robust infrastructure such as Kubernetes.

4.6.2 Location Fulfillment

The Hydro framework features a deep integration with Hydro Deploy to enable type-safe deployments. As we saw earlier, Hydro programs are built around the concept of locations, which represent the physical machines where the program will run. These locations are represented as Rust values, and each element of the Hydro IR will be compiled into a projected binary for the corresponding location. When streams are moved across locations, the IR captures both the source and destination locations, which means that Hydro is aware of the global network topology.

After building the global dataflow graph, Hydro offers a set of APIs to deploy a `FlowBuilder` to physical machines by mapping each location to a Hydro Deploy specification. The type system ensures that each process location is mapped to a single machine and each cluster is mapped to an array of machines. Once all locations are specified, Hydro walks the IR and automatically configures all necessary network channels between each deployed service. We show an example of deploying a simple distributed program in Figure 4.19.

```
async fn main() {
    let flow = FlowBuilder::new();
    let leader = flow.process::<Leader>();
    let workers = flow.cluster::<Worker>();

    distributed_logic(&leader, &workers);

    let mut deployment = hydro_deploy::Deployment::new();

    let launched = flow
        .with_process(leader, deployment
            .GcpComputeEngineHost()
            .machine_type("e2-micro")
            .region("us-west1-a"))
        .with_cluster(workers, vec![
            deployment.GcpComputeEngineHost()
                      .machine_type("e2-micro")
                      .region("us-west1-a"),
            deployment.GcpComputeEngineHost()
                      .machine_type("e2-micro")
                      .region("us-west1-b") // different region
        ])
        .deploy(&mut deployment).await?;
}
```

Figure 4.19: A simple example of using Hydro to deploy a distributed program.

By leveraging distributed locations in Hydro, developers can build distributed systems that respect modern constraints such as data sovereignty laws. For example, a Hydro program can assign distinct cluster locations to machines in the US and Europe, so that the global program respects data locality regulations. In the future, Hydro can place further constraints on the data types that can be sent between locations, so that developers can enforce security policies.

This integration also enables the ability to *optimize* deployments. For example, we can automatically choose which region each cloud machine is placed in based on the network connections between the corresponding locations. Similarly, members of a cluster can be automatically distributed across several cloud providers and regions to satisfy fault tolerance constraints. We believe that such techniques, combined with optimizations over the dataflow graph, point towards co-design of dataflow and deployment.

After deployment, developers can use the `FlowBuilder` API to interact with the launched services. Since processes and clusters have distinct Rust types, these APIs automatically switch between returning a single handle or a set of them. Developers can use machine handles for a variety of monitoring tasks, such as to stream logs or send configuration signals. By integrating deployment into the Hydro framework, developers can write programs that capture end-to-end behavior in a single language with strong type guarantees.

4.7 Related Work

4.7.1 Choreographic Languages

Hydro's global programming model is inspired by choreographic programming languages [59, 110, 132], which provide a global view of the entire distributed system. Choreographic languages allow developers to write distributed programs as a single unit, rather than splitting them into separate services. This makes it easier to reason about the overall behavior of the system.

Recent work has brought choreographic programming to Rust through a library called ChoRus [80]. ChoRus lets developers define distributed locations and write sequential Rust code that spans them. A key difference between this work and Hydro is the approach to projection. ChoRus performs projection entirely through Rust's type system, whereas Hydro uses a staged programming model. Like Hydro, ChoRus supports zero-cost projection, but relying on the Rust type system comes with many limitations. Locations in ChoRus must be statically defined and ChoRus does not generate an internal global IR, which means that developers cannot run optimizations over the global program or use the IR to generate deployment logic.

The other distinction from choreographic languages is our use of streaming semantics rather than a sequential model. As we saw in Gyatso (Chapter 3), stream programming makes it possible to capture a much wider range of distributed programs, including those that offer scale-out distribution with clusters. Hydro leverages this model to offer richer semantics for reasoning about concurrency and scaling, without sacrificing the end-to-end correctness guarantees.

There is also a line of research from the database community that is not formally choreographic but shares many similar themes. The Dedalus language [12] and its predecessors [3, 63]

allow developers to write distributed programs as Datalog rules that explicitly capture where facts are generated. These languages directly inspired the cluster location type in Hydro, which offers similar scale-out mechanisms. However, these languages focus on a restricted relational algebra, which limits their expressiveness in comparison to Hydro.

4.7.2 Distributed Programming Frameworks

Distributed programming frameworks have been around for decades, focusing on similar challenges in building distributed systems. These frameworks span a wide range of programming paradigms, correctness concerns, and performance goals. Hydro draws significant inspiration from this prior work, with a goal of extending their guarantees to global behavior.

A particularly prominent class of frameworks are those that use the actor model [65], which provide a message-passing abstraction for building distributed systems as a composition of lightweight single-threaded units called actors. Actor frameworks like Erlang [15] and Akka [125] use an underlying sequential execution model that focuses on atomic state management and fault tolerance, but do not make any guarantees about logic that spans several actors.

Other frameworks, such as Durable Execution [27] and Unison [150], model distributed systems as a set of asynchronous functions with internal message-passing. But these systems provide limited help reasoning about correctness when shared state is involved, since they do not model concurrent requests. Languages like Gallifrey [109] and Lasp [106] use the type system to enforce safe use of replicated data, but do not let developers control the replication algorithms themselves. Hydro's streaming model captures more fine-grained detail, including the effects of concurrent requests and network non-determinism.

Finally, there are a number of programming frameworks–such as Apache Spark [165], Apache Flink [30], and Apache Beam [101]–that leverage distribution to process large amounts of data. These frameworks provide a set of APIs for building distributed data processing pipelines, but they are intended for high-level data processing rather than general-purpose distributed programming. Hydro's focus on providing explicit control over networking and its APIs for performing atomic processing allow it to be used for a wider range of distributed applications.

4.8 Summary

In this chapter, we presented Hydro, a new framework for building correct, modular, and performant distributed systems. Hydro is built around a staged programming model that allows developers to write distributed programs as a single unit with type-checked guarantees on the end-to-end correctness of the system. Hydro offers a variety of live collection types which capture the semantics of distributed programming, including unbounded streams, asynchronously updated singletons, and stream variations that capture non-deterministic ordering and retries. With support for type-safe integrated deployment, Hydro provides a practical foundation for building modern distributed applications.

Chapter 5

Stageleft: Multi-Stage Programming in Standard Rust

5.1 Introduction

A major focus of modern languages is *zero-cost abstraction*, which aims to eliminate the performance impact of code reuse. Languages like Rust use specialization and compile-time memory management to enable such abstractions. But these techniques only go so far, as the structure of the source code still has an impact on the compiled binary—it determines function boundaries, execution order, and optimization opportunities. This leads to performance penalties when using abstractions that are not well aligned with low-level execution, such as dataflow programming.

Multi-stage programming [142, 144] is a powerful metaprogramming technique that aims to address this. Instead of directly compiling the source code to a runtime binary, staged programming compiles code using several (typically two) *stages*. When a program that uses staging is compiled and run, it does not directly execute the defined computation; instead, it generates a new source file containing the computation. This generated code is then compiled (the second stage) and executed to yield the final result.

Staged programming decouples the abstractions and interfaces exposed to developers from the structure of the compiled runtime code, all with minimal impact to the end-user APIs. Multi-stage programming has been used extensively in deep learning engines [4, 55, 121], query optimizers [128], and stream processing [82, 120] to extract bare-metal performance without impacting the high-level abstractions. In this dissertation, we saw how staging is used for zero-cost distributed programming in the Hydro framework (Chapter 4).

Some languages support staged programming out of the box with built-in constructs. For example, MetaOCaml [141] is a fork of the OCaml language that offers built-in staging mechanisms. Similarly, the Scala language has a rich metaprogramming system that enables staged programming [126, 139, 140]. These languages have deep integration with the compiler to make staging a first-class construct. Yet most languages, even those with metaprogramming capabilities, still do not support staged programming.

A prominent example of this is Rust, which has a powerful macro system that allows developers to write code generators, but does not support staged programming. The Rust macro system requires users to manipulate untyped blobs of tokens. While this is fine for Rust experts writing specialized code generators, it is impractical for developers to write application logic with such mechanisms. Staged programming is only practical when the high-level APIs have well-typed signatures and appear as regular functions from the perspective of the user.

In this chapter, we present *Stageleft*, a staged programming library for Rust that allows library authors to leverage staging to generate high-performance code while preserving a familiar API for users. Stageleft uses Rust's existing macro system under the hood, which means that it can be used in any Rust program without requiring special support from the compiler.

Stageleft provides strong static guarantees that the generated code is well-typed, and uses Rust's type system in novel ways to enable use across domains like distributed systems. Programs written in Stageleft can be consumed from other Rust code as macros, or can generate standalone Rust crates that can be compiled and run independently.

The primary constructs for staged programming are *quoting* and *splicing*, which have similar behavior to their typical metaprogramming forms but typically come with stronger typing guarantees. Stageleft provides these with a Rust macro that can be used to quote Rust expressions into a value that captures the type of the quoted code, which can then be spliced into another quoted expression to compose code blocks. Libraries are free to manipulate the quoted code using standard Rust metaprogramming libraries like syn [153] to generate runtime logic.

A key invariant of staged programming is that if the staged code compiles, the generated code will also compile. To achieve this, Stageleft must carefully manage the scope of symbols referred from quoted code and ensure that any external symbols are resolved correctly when the expression is spliced. Because Rust macros run before typechecking, we cannot rely on information from Rust to resolve symbols. Instead, Stageleft uses a variety of lightweight program analysis techniques to identify and resolve free variables in the quoted code.

A unique aspect of Stageleft is that it extensively leverages Rust traits to define interfaces for splicing values. While Stageleft provides built-in implementations for composing quotes and splicing constants, it also allows library authors to define implementations for their own types. This enables libraries to create special values that behave differently in a quoted context, such as for accessing metadata that is only available in the final generated code.

In summary, we make the following contributions:

- We present a construct for quoting and splicing code in standard Rust that preserves type safety and enables seamless use by libraries (Section 5.3)
- We explore methods for ensuring hygienic expansion without requiring special support from the Rust compiler (Section 5.4)
- We discuss how Stageleft allows quoted code to reference free variables and how library authors can customize their expansion (Section 5.5)
- We show how staged programs can be exposed as macros or standalone crates (Section 5.6)

5.2 Metaprogramming in Rust

Before we dive into Stageleft, we first need to understand the landscape of existing metaprogramming techniques in Rust. Rust offers a pair of built-in metaprogramming features: *declarative macros* and *procedural macros*. Declarative macros are a simple way to define code generation rules using a pattern-matching syntax. They are typically used for simple code generation tasks, such as generating boilerplate code or implementing common traits.

Procedural macros, on the other hand, are much more powerful and closer to the concept of staged programming. They allow developers to write custom code-generation logic using Rust itself, without any limitations on the macro logic or control flow. Procedural macros are defined as standard Rust functions that take a token stream as input and produce a token stream as output. Libraries can expose procedural macros using several syntactic constructs: as a function-like macro or as an attribute. In this chapter, we will focus on function-like macros, which consume and emit Rust expressions.

To write a procedural macro, developers must separate their code-generation logic into a separate crate, which will be dynamically loaded by the compiler. Because the Rust compiler provides a very minimal set of APIs for interacting with tokens, developers often use third-party libraries like `syn` [153] to parse the token stream into a Rust AST and manipulate it. When a macro is used in a program, the compiler invokes the macro implementation *before typechecking* the program. This offers significant flexibility, as the macro can introduce new types and functions, but it also means that the macro does not have access to types and symbol information.

The internal token representation of Rust code is also untyped, which is a significant difference from staged programming. With staging, quoted code may be stored as tokens under the hood, but the container includes the type of the quoted expression. When composing quoted snippets into larger expressions, this type information is used to ensure that the generated code is well-typed. In contrast, Rust macro authors are effectively performing string interpolation over Rust syntax, which means that they are responsible for ensuring that the generated code is syntactically valid and well-typed.

The other major distinction from staging is the symbols that can be referenced from quoted code. Rust procedural macros are *not hygienic*, and can reference any symbol in the scope of the macro invocation. A further effect is that quoted code in the macro implementation cannot reference symbols defined alongside the macro such as a helper function. Instead, the macro must generate all code under the assumption that it will be pasted into a different crate.

This is problematic for a library that wants to expose a staged interface, since developers often reference local symbols in closures passed to a library API. Stageleft addresses this by ensuring that quoted code is hygienic, and by providing ways for quoted code to safely reference local symbols. This is done by applying a lightweight program analysis to identify external symbols, and a trait system that generates the appropriate code to resolve these symbols in spliced code.

5.3 Quoting and Splicing

Developers using Stageleft will spend most of their time interacting with the `q!` macro, which is the primary interface for quoting a Rust expression in a type-safe manner. The `q!` macro is a function-like macro that takes a Rust expression as input and produces a quoted expression. The quoted expression can then be spliced into another quoted expression using the `q!` macro, or into a raw set of tokens using the `splice` method. In this section, we will discuss the basics of quoting and splicing in Stageleft, including how we built type-safe quoting on top of Rust's existing macro system.

5.3.1 Quoting Expressions

When quoting a piece of code in Stageleft, the quoted code is returned as a value that implements the `Quoted` trait, which captures the tokens in a type-safe container that tracks the type of the quoted expression. The only APIs available on this trait are for splicing the expression. Instead of using a struct type for quoted values, Stageleft uses traits to enable library authors to define their own data types with custom splicing behavior. We show the core definition of the `Quoted` trait in Figure 5.1. Our definitions vary slightly from the open-source implementation[1] for simplicity; we omit lifetimes that enable more use cases but do not affect overall correctness.

```
pub trait Quoted<T> {
    fn splice(self) -> TokenStream;
}
```

Figure 5.1: The `Quoted` trait, which is used to represent quoted expressions.

The primary way to create a quoted value is with the `q!` macro, a function-like procedural macro which takes a Rust expression as input and produces a quoted expression. Unlike the existing `quote` macro provided by Rust, the `q!` macro returns a value that captures the expression type. To achieve this, we need to expand the macro such that the provided expression is type-checked, but *not run*. For Rust's typechecker to infer the expression's type, the expression must show up as a return value. Our trick is to generate a closure whose return type will capture the quoted expression's type, but stores the tokens of the expression into a string as a side effect when it is invoked.

Consider a simple example of quoting an arithmetic Rust expression. The quoting macro stores the AST of the expression in a string, and then emits the expression itself for typechecking purposes in an unreachable branch. Because both branches of the conditional must return a value of the same type, we return an uninitialized value using `MaybeUninit`. Although this is unsafe in Rust, we never read the value returned and therefore preserve safety. Because the else branch has an underspecified type (`MaybeUninit` can be implemented for any Rust type), the type inference

[1] https://github.com/hydro-project/stageleft/tree/stageleft-v0.8.1

algorithm will use the unreachable branch to determine the inferred return type. We show an example of the invocation and expanded code in Figure 5.2.

```
q!(1 + 1) // expands to...
|set_output: &mut String| {
    *set_output = "1 + 1".to_string();
    if false {
        1 + 1 // drives type inference
    } else {
        unsafe {
            MaybeUninit::uninit().assume_init()
        }
    }
} // inferred type: impl Fn(&mut String) -> i32
```

Figure 5.2: Quoting a simple arithmetic expression, and the expanded result.

The inferred type of the quoted expression is a function that returns a value of the same type as the quoted expression. This is our core mechanism that enables type-safe metaprogramming, but so far this expression does not implement the `Quoted` trait. To do this, we need to provide a mechanism for splicing a quoted expression provided by a closure value. We show this implementation in Figure 5.3, which uses the syn library to parse the string of tokens.

```
impl <T, F: FnOnce(&mut String) -> T> Quoted<T> for F {
    fn splice(self) -> TokenStream {
        let mut output = String::new();
        let result = self(&mut output);
        std::mem::forget(output); // so that Drop doesn't run
        let expr = syn::parse_str::<Expr>(&output).unwrap();
        expr.into_token_stream()
    }
}
```

Figure 5.3: The implementation of the `Quoted` trait for quoted expressions.

Because the expanded closure returns an uninitialized value when invoked, we need to be careful with the value to avoid undefined behavior. When a Rust value goes out of scope, the compiler automatically inserts a call to the `Drop` trait, which can perform custom clean-up logic. Because the value is uninitialized, calling `Drop` on it would be undefined behavior, so we instead use `std::mem::forget` to prevent the destructor from running. The remaining logic simply converts the string containing the quoted expression into a token stream using the `syn` library, so that we can manipulate it with standard APIs for Rust ASTs.

5.3.2 Splicing and Code Generation

Once a piece of code has been quoted, a library author can splice it into generated code using the `splice` method. This method takes a quoted value and produces an untyped token stream capturing the expression. The spliced expression can then be used in any context where a Rust expression is expected, such as in a function body or as an argument to another function.

An important invariant to maintain while splicing is that the spliced code should *always* compile, because the quoted code is already typechecked. Maintaining syntactic correctness is straightforward, since the quoted code already successfully parsed. What is more challenging is ensuring that the spliced code typechecks, since the original typechecking environment may be different than the spliced context.

A particularly tricky case is when the quoted code is for a closure whose input and output types were inferred. When the expression was initially quoted, the type inference algorithm can use constraints on the quoted value to infer types for the closure. But when the quoted code is spliced into a new context, the type inference algorithm may choose different types or fail to infer types altogether. This is a common problem when writing dataflow-oriented libraries with Stageleft, whose APIs often take quoted closures as input.

```
// provided by stageleft
fn fn1_type_hint<T, U>(f: impl Fn(T) -> U) -> impl Fn(T) -> U {
    f
}

// user code
let quoted: impl Quoted<impl Fn(u32) -> u32> = q!(|x| x + 1);

quoted.splice_fn1() // generates the tokens:
"stageleft::fn1_type_hint::<u32, u32>(|x| x + 1);"
```

Figure 5.4: An example of splicing a quoted closure with type hints.

To address this, Stageleft provides mechanisms for a quoted expression to be spliced *along with* type hints that guide the type inference algorithm. To do this, we rely on Rust's `std::any::type_name` API, which allows us to obtain a string representation of a type. Because Rust specializes each generic function instance, the string representation will always be a fully qualified type path instead of a local type parameter. The string types can then be parsed back into a Rust AST and included in the tokens generated by the splicing process.

For example, when a library splices an expression using an API such as `splice_fn1` (there are many variants, one for each closure arity), we obtain the input and output types from the `Quoted<T>` trait and include them in the spliced code. When Rust type checks the spliced code, it will use these types as constraints when inferring the type of the closure. We show an example of this in Figure 5.4, which uses the type hints to preserve inferred types.

5.4 Scope Hygiene

When quoted code is a pure, local expression without any external references, it is easy to splice it into another context without changing the meaning of the code. When quoted code references external symbols, however, we need to be careful to ensure that the spliced code resolves the same symbols as the original code. This is a common problem in metaprogramming, and is known as the *hygiene* problem. An *unhygienic* macro allows the context of the spliced code to affect the meaning of it, which is the default behavior of Rust procedural macros. Stageleft aims to provide a hygienic quoting mechanism that lets developers use metaprogramming with confidence.

There are two major components to hygiene. The first deals with references to free variables, which are local values that were defined outside the scope of the quoted code; we will discuss this in Section 5.5. In this section, we focus on hygienic references to types and function symbols that require special care to re-resolve in spliced code.

5.4.1 Local Symbols

As we saw earlier, when quoting code with `q!`, the provided expression is emitted as-is to be typechecked. This means that quoted code is free to reference a type or function that is available in its scope. When the code is spliced, however, the context may be a different module or crate and the relative paths used in the quoted code may not resolve correctly.

The first step Stageleft takes to address this is to capture metadata about where an expression was quoted, so that we can attempt to replicate the original context at the splice site. Rust helpfully provides a macro called `module_path!` that returns the current module path as a string, which we can store in the quoted value next to the expression AST. Then, when the quoted code is spliced, we can import all symbols from that module so that they are available in the spliced code. We show an example in Figure 5.5, where the quoted code references a local function `helper`.

```
pub mod foo {
    pub fn helper() -> i32 { 42 }

    fn bar() {
        let quoted: impl Quoted<i32> = q!(helper()); // expands to:
        |set_output: &mut String, set_module: &mut &'static str| {
            *set_module = std::module_path!(); // "crate::foo"
            ...
        };
        quoted.splice() // "{ use crate::foo::*; { helper() } }"
    }
}
```

Figure 5.5: An example of quoting an expression that references a local function.

Note that this approach has a few limitations due to the Rust compiler. It cannot handle references to functions or types defined *inside* another function, since Rust does not provide a way to directly refer to such a symbol. Currently, Stageleft warns developers to avoid this pattern in documentation, but we hope to provide a more robust solution via custom lints or deeper integration with the compiler in the future.

Another challenge is that the quoted code may reference symbols that are not publicly visible from other modules, or symbols which are imported using a `use` statement. If the quoted code is spliced into another module, these symbols will not be available in that context. To address this, Stageleft applies a build-time step to generate mirrors of each module in the program that publicly exposes all local and imported symbols. Then, instead of importing symbols from the original module, the spliced code imports from the mirror module. We show an example of mirroring in Figure 5.6, where the quoted code references a symbol that is not publicly visible.

```
mod foo {
    use std::cmp::max;
    fn helper() -> i32 { 42 }

    pub fn bar() -> impl Quoted<i32> {
        q!(max(1, helper())).splice() // generates the tokens:
        "{ use crate::__staged::foo::*; { max(1, helper()) } }"
    }
}

mod __staged { // generated mirror module
    pub mod foo {
        pub use std::cmp::max;
        pub fn helper() -> i32 { 42 }
        pub use crate::foo::bar;
    }
}
```

Figure 5.6: An example the mirror module generated to resolve private symbols.

The mirror module simply duplicates every private declaration in each module, and re-exports any symbols that are already public. This means that some functions will be duplicated in the mirror module, but our generation process takes care that the behavior is unchanged. Overall, this approach is a *hack* to work around the privacy restrictions in Rust, but it works well in practice and avoids having to modify the compiler. In future work, these limitations could be addressed by mechanisms to bypass privacy restrictions for generated code or to specify a scope in which symbols should be resolved. These concerns have already been raised in the Rust community for procedural macros; we believe that staging provides a compelling use case for this.

5.4.2 Resolving Paths

The other set of references that need to be handled with care are *relative paths*, which can use keywords like `self`, `super`, and `crate` to refer to the current module, parent module, or crate root, respectively. If these symbols are passed through directly to the spliced code, they will not resolve correctly in the new context. To address this, Stageleft performs lightweight rewrites over the quoted AST to transform these relative paths into absolute ones.

Conveniently, paths in Rust are syntactically distinguishable from other expressions (unlike languages like Java or Scala), so we can traverse the AST of the quoted expression and rewrite any relative paths based on the known module path. For example, if the quoted code references `self::foo` and the module path is `crate::bar`, we rewrite the path to `crate::bar::foo`. When the quoted code needs to be spliced into a different crate than where it was quoted, we further rewrite the `crate` keyword to the name of the crate where the quoted code was defined.

By appropriately resolving local symbols, developers using libraries powered by Stageleft can easily pass in custom quoted code without worrying about restrictions about what can be quoted. In particular, developers are free to extract common runtime logic into helper functions and define custom types that are used in the quoted code. This makes it much more practical to introduce staging since it has limited effects on how developers write their application logic.

5.5 Free Variables

So far, we have only considered quoted code with external references to *symbols* such as function declarations and types. But many developers need their quoted code to *capture* local values with semantics similar to how a closure captures locals. This is a complex topic in staging, since quoted code cannot capture arbitrary local values[2] since they will not be available in the spliced context—a completely independent program!

References to locals come in many forms for various purposes. A quoted reference may refer to a constant value that should be inlined into a spliced code, a value that will be materialized elsewhere in the generated program at runtime, or even another quoted expression for composition. All of these cases require different semantics, but they all show up in quoted expressions in the same form: a *free variable*.

Free variables refer to any symbol that appears in a quoted expression but is not defined in the quoted code. Stageleft automatically detects free variables using a lightweight program analysis, and then uses a trait system to handle them appropriately based on their type. Handling free variables requires careful reasoning about Rust's borrow checker, to ensure that resolved references are valid in the spliced code. With free variables, Stageleft enables powerful composition techniques that allow developers to build abstractions around quoting.

[2] Some staging libraries [91, 126] allow capturing values of any type because the generated code is JIT compiled into the existing runtime. Because Rust is AOT compiled, Stageleft generates independent binaries, so we do not have the same flexibility. In future work, an FFI layer could allow capturing local values across this boundary.

5.5.1 Syntax-Driven Analysis

The first step in handling free variables is to identify them in the quoted code. Stageleft uses a syntax-driven analysis to identify free variables in the quoted expression, since the `q!` macro runs before typechecking and therefore does not have access to symbol information. To walk the AST, we leverage the `syn` library, which provides convenient visitor patterns for traversing and mutating Rust ASTs in-place. Visitors help ensure coverage of all Rust syntax, and allow us to focus on the relevant parts of the AST.

Our visitor works by maintaining a stack of scopes, which are used to track the current set of symbols that have been declared in the current block or its parents. When a new block is entered (such as a closure declaration of expression block), we push a new scope onto the stack since any declarations inside will not be visible outside. When a new symbol is introduced through a variable declaration or closure parameter, we add it to the current scope. As we walk the AST recursively, every time we encounter an identifier, we check if it is in any of the scopes on the stack; if it is not, we add it to the set of free variables.

A unique challenge when handling Rust ASTs is that the quoted code may itself use Rust macros. Because macros consume arbitrary tokens, the parsed AST stops at the invocation and does not include the macro expansion. This can result in the visitor missing free variables if the macro is invoked with an expression containing variable references. To tackle this, Stageleft attempts to parse each argument to the macro as a Rust expression, and if it succeeds we then walk the AST of the argument as usual. While this is not a perfect solution (in particular, it fails to handle formatting macros that use string interpolation syntax), it works well for common cases like `dbg!`. There have been proposals to allow Rust macros to change expansion order, which would allow us to handle this robustly, but they have not yet been accepted into the language.

5.5.2 Splicing Free Variables

Once we have identified the free variables, we need to appropriately determine what the type will be when spliced into the quoted expression and interpolate the spliced expression into the quoted code. Because different types of free variables may expand in different ways, Stageleft uses a trait called `FreeVariable`, which is implemented by all types that can be spliced.

There are two responsibilities of the `FreeVariable` trait. The first is to provide a method for splicing the free variable into the quoted code, which is done in `FreeVariable::to_tokens`. This method consumes the free variable as an owned value and returns a token stream that represents the free variable. The token stream will be used to compute the value of the free variable inside the quoted code. Because we take ownership of the free variable container, we can enforce borrow checking rules by preventing illegal aliasing.

The second is to provide the type of the expanded variable, which is done by using a *generic type* in the trait definition. By having the trait provide the type the free variable will expand to, we can ensure that the quoted expression is type checked with respect to the spliced value, rather than its container. The trait also provides an `uninitialized` method for obtaining a dummy value for typechecking purposes. We show the definition of the `FreeVariable` trait in Figure 5.7.

```rust
pub trait FreeVariable<O> {
    fn to_tokens(self) -> TokenStream;
    fn uninitialized(&self) -> O {
        unsafe {
            MaybeUninit::uninit().assume_init()
        }
    }
}
```

Figure 5.7: The core definition of the `FreeVariable` trait.

When a quoted expression references a free variable, the `q!` macro generates calls to the `FreeVariable` implementation for each referenced variable. First, we obtain dummy values for each free variable, which are stored in variables with the same identifier as the free variable (which shadows them for use in typechecking). Then, we generate the quoted tokens by interpolating the free variable tokens into the rest of the expression.

```rust
q!(x + 1) // expands to:
|set_output: &mut String, ...| {
    let x_shadow = FreeVariable::uninitialized(&x);
    let x_tokens = FreeVariable::to_tokens(x);
    *set_output = quote! {
        let x = #x_tokens;
        { x + 1 }
    }.to_string();

    if false {
        let x = x_shadow;
        { x + 1 }
    } else { ... }
}
```

Figure 5.8: An example of quoting an expression that references a free variable.

Notably, we *do not* insert the free variable tokens directly at the point where the variable was referenced. This is because the same free variable may be referenced multiple times in the quoted code, but to preserve semantics we cannot inline the free variable tokens twice as they may involve mutable references. Instead, we compute the value of the free variable at the top of the quoted expression, and then store the original tokens as-is. We show an example of this expansion in Figure 5.8, where the quoted code references a free variable `x`. The expanded code uses the `quote!` macro from syn, which provides a simple way to splice tokens.

5.5.3 Composing Quoted Values

The primary type of free variable that Stageleft supports is a quoted expression itself. By letting quoted expressions reference other ones, we can compose quoted code together to build more complex expressions. This is particularly useful for developing abstractions that need to augment user-provided quoted logic before passing it off to a lower-level API and for defining code generation logic in a type-safe manner.

A quoted expression of type `impl Quoted<T>` also implements `FreeVariable<T>`, which means that inside the quoted expression the free variable will appear to have type `T`. To generate the spliced tokens, we simply pass through the tokens of the quoted expression as-is. This is safe from a hygiene perspective because we guarantee that all free variables have been resolved before the quoted expression is spliced, so no symbols outside the spliced tokens will be used.

To see the free variable mechanism in action, let us walk through an end-to-end example composing two quoted snippets. Both perform a simple arithmetic computation, but the second relies on the value computed by the first. Because splicing a free variable consumes it by-value, we can no longer use it after it is captured. We show this composition in Figure 5.9.

```
let quoted_a = q!(1 + 1);
let quoted_b = q!(quoted_a * 2);

quoted_b.splice() // generates the tokens:
r#"{
    let quoted_a = { 1 + 1 };
    { quoted_a * 2 }
}"#
```

Figure 5.9: An example of composing quoted expressions.

While Rust traits are typically handled using static dispatch by specializing to the trait implementation, Rust also offers *trait objects*, which use dynamic dispatch to allow for dynamic fulfillment of a trait. We can use this in Stageleft to perform dynamic code generation, by passing around quoted trait objects across traditional control flow mechanisms.

This allows us to achieve a common use case for staged programming: partial evaluation. In situations where some inputs to a computation are known at stage-time, we can perform code generation that bakes in that input for higher performance. A classic example of this is raising an unknown base to a known power. Instead of relying on expensive instructions or runtime control-flow, we can instead generate an expression that uses a constant number of multiplications. By using staged programming, we can easily implement this optimization in a *type-safe manner*, guaranteeing that the expanded code will always compile.

The `Quoted` trait provides a helpful `boxed` API that turns a quoted expression into a heap-allocated trait object. This allows different branches in the recursion to return different quoted values, while the function still returns a single unified type. We also require the base to imple-

```rust
fn exp(base: impl FreeVariable<i32> + Copy, power: u32) -> impl Quoted<i32> {
    if power == 1 {
        q!(base).boxed()
    } else if power % 2 == 0 {
        let v = raise_to_power(base, power / 2);
        q!(v * v).boxed()
    } else {
        let v = raise_to_power(base, power / 2);
        q!((v * v) * base).boxed()
    }
}

exp(2, 5).splice() // expands to (simplified):
{
    let v = {
        let base = 2;
        let v = {
            let v = {
                let base = 2;
                { base }
            };
            { v * v } // 2^2
        };
        { (v * v) * base } // 2^5
    };
    { v * v } // 2^10
}
```

Figure 5.10: An example of partial evaluation for raising an unknown base to a fixed power.

ment `Copy`, allowing it to be passed into recursive calls spliced into multiple quoted expressions. While expressions quoted with `q!` do not implement this trait because they may capture mutable references, it can be implemented for constant literals as we will see shortly.

We show the full implementation and an example of expansion in Figure 5.10, where the expanded code more efficiently computes the power by breaking it down into squaring multiplications. Each intermediate value of `v` is computed by splicing the nested quoted expression, so the function recursively builds up the result. The boxing and dynamic dispatch across in staged programs is a *zero-cost abstraction*—the spliced code is not affected by the indirection and has the same raw performance.

5.5.4 Custom Free Variables and Ownership

In addition to composing quoted expressions, Stageleft also allows library authors to define their own container types that can be spliced into quoted code. These can be used to interact with the borrow checker in more complex ways by permitting aliased references to shared data or to enable access to data outside the scope of the spliced code.

A common special free variable type is for constant data available at stage-time that is referenced from quoted code. For example, if we have a variable storing an integer, we may want to reference it in a quoted expression with the integer value inlined as a constant literal. To do this, we need to implement the `FreeVariable` trait for primitive types such as integers, strings, and other data types which can be transferred into the quoted code as a literal. We show examples of these trait implementations in Figure 5.11, where we implement the trait for integers and strings.

```
impl FreeVariable<i32> for i32 {
    fn to_tokens(self) -> TokenStream {
        let expr = syn::parse_str::<Expr>(&self.to_string()).unwrap();
        expr.into_token_stream()
    }
}

impl FreeVariable<&'static str> for &str {
    fn to_tokens(self) -> TokenStream {
        let expr = syn::parse_str::<Expr>(&format!("\"{}\"", self)).unwrap();
        expr.into_token_stream()
    }
}
```

Figure 5.11: Implementations of the `FreeVariable` trait for integers and strings.

Because the container type is a primitive Rust value, such free variables also implement the `Copy` trait and can be used in multiple quoted expressions, because their spliced code is guaranteed to be pure and reference-free. We used this in Figure 5.10 to allow the base to be passed in as a regular Rust integer, which is then spliced into the quoted code as a constant literal.

Another custom free variable type provided by Stageleft allows for *unhygienic* access to an identifier declared outside of the quoted code. This is useful for referencing inputs to the program that will only be available at runtime, or for accessing global metadata in the generated program. To expose such a variable, Stageleft provides a `RuntimeData<T>` struct that captures a reference to an externally-declared variable of type `T`. This container simply stores the name of the variable under the hood and emits it when spliced.

A key component of Rust's borrow checker is setting restrictions on which types of variables can be referenced in multiple places, versus which can only be consumed once. An integer variable can be referenced many times, while a heap-allocated `Box` will be *moved* when it is used.

In Rust, these rules are captured by the `Copy` trait, which is implemented for types that can be transparently copied by value. In Stageleft, we lift these rules to the `RuntimeData` type, which implements `Clone` if the underlying type `T` implements `Copy`. This allows developers to explicitly create multiple references to the same variable, while ensuring that the ownership rules are respected. We show the definition and traits for `RuntimeData` type in Figure 5.12.

```
struct RuntimeData<T> {
    name: String,
}

impl<T> FreeVariable<T> for RuntimeData<T> {
    fn to_tokens(self) -> TokenStream {
        let expr = syn::parse_str::<Expr>(&self.name).unwrap();
        expr.into_token_stream()
    }
}
impl<T: Copy> Clone for RuntimeData<T> {
    fn clone(&self) -> Self {
        RuntimeData { name: self.name.clone() }
    }
}
```

Figure 5.12: The definition of the `RuntimeData` type and its trait implementations.

In Stageleft, Rust ownership rules are handled entirely through the `FreeVariable` trait; this enables a *general but precise* mechanism for staging that respects the borrow checker. In particular, variables storing shared pointers (`&T`) implement `Copy` and can be referenced multiple times. On the other hand, variables storing unique pointers (`&mut T`) can only be referenced once. While this is an overly restrictive condition, since mutable pointers can be used multiple places as long as only one instance is active at a time, we cannot enforce such restrictions in staging and take a more conservative approach.

Currently, instances of `RuntimeData` are manually created by mechanisms *outside* the quoted code and therefore Stageleft cannot guarantee that the variable will typecheck appropriately in the spliced code. In our quoting implementation, variable declarations inside quoted code cannot be referenced in nested quoted expressions, because those nested values must be materialized *before* they are spliced. In future work, we hope to support quoted expressions that explicitly reference free variables that will be declared in parent code, by automatically generating `RuntimeData` instances for these "parent variables".

5.6 Expansion and Entrypoints

The last piece of Stageleft is to provide a way to compile staged code into runnable binaries. To give library authors flexibility in code generation, Stageleft provides two ways to expose staged snippets: as a macro or as an independent Rust source file. The former is useful for developers who want to consume staged logic as an API in other Rust code, while the latter is useful when staged code needs to be wrapped with additional runtime logic and launched in a particular way.

5.6.1 Type-Safe Rust Macros

Staged logic written by Stageleft can be wrapped in a *macro entrypoint*, which is accessible from other Rust code as a *standard proc-macro*. The broad approach of exposing staging through macros is a standard staging technique used in languages like Scala [139], and turns Stageleft into a high-level API for writing macros that are *guaranteed* to typecheck when expanded.

Developers can create a macro entrypoint by defining a regular Rust function and annotating it with the `#[stageleft::entry]` macro. The function must have a return type of the form `impl Quoted<T>`; the quoted expression will be emitted as part of the macro expansion. The `stageleft::entry` macro wraps the function into a procedural macro definition with the same name which can be used in other Rust crates. This indirection, due to restrictions in the Rust compiler, means that any project using macro entrypoints must have at least two crates: one where the macro is defined, and one where it is used.

The parameters to the entrypoint function define inputs to the macro. Stageleft entrypoints cannot consume arbitrary tokens as inputs, as all inputs must be well-formed Rust expressions in order for them to be exposed as type safe staged values. When the macro is invoked, the input tokens are parsed into a list of Rust expression ASTs and then handled according to the corresponding type in the parameter list.

If the parameter is a `RuntimeData` type, the corresponding argument in the invocation will be evaluated and stored into a variable, and then handed off for opaque use by the staged logic. Like with free variables, the expression *will not* be inlined into the expanded code in order to preserve typical Rust evaluation semantics where function arguments are evaluated eagerly. This is useful for computational inputs to the staged code which will be computed at runtime.

If the parameter is any other type, it must implement a `ParseFromLiteral` trait defined by Stageleft. Each implementation of this trait defines logic to extract a value of type `T` given tokens representing a literal of that type (this is a dual of our previous `FreeVariable` implementations). Parsing literals is useful for cases where an integer or string will be passed to the macro as an inline literal, and allows the macro to generate code based on that input.

In order to guarantee type safety in the macro expansion, the quoted expression is wrapped into a function definition that takes in all the `RuntimeData` inputs as parameters. This ensures that if an expression passed into the macro invocation has the wrong type, the error will be isolated to that parameter instead of causing typechecking failures in the quoted code. We show an example of wrapping the exponentiation function in a macro entrypoint in Figure 5.13, where the macro takes in a base and power as parameters.

```rust
#[stageleft::entry]
pub fn exp_entry(base: RuntimeData<i32>, power: u32) -> impl Quoted<i32> {
    exp(base, power)
}

// expands to:
#[proc_macro]
pub fn exp_entry(input: TokenStream) -> TokenStream {
    let input_parsed: Vec<syn::Expr> = ...;
    let output_core = Quoted::splice(exp(
        RuntimeData {
            name: "value".to_string()
        },
        <u32 as ParseFromLiteral>::parse_from_literal(
            &input_parsed[1],
        )
    ));

    quote! {
        fn expand_staged(base: i32) -> i32 {
            #output_core
        }
        expand_staged(#(&input_parsed[0]))
    }
}

// invocation (in another crate):
let dynamic_input = ...;
let result = exp_entry!(dynamic_input, 5);
```

Figure 5.13: An example of a macro entrypoint for the exponentiation function.

Staged macros are an effective way to incrementally introduce staged programming to an existing Rust codebase. Because macro entrypoints use existing Rust mechanisms for code generation, they are easy to use and have strong support from tooling like IDEs—even features such as code completion work on arguments passed to the macro. Furthermore, the macro endpoint hides the implementation details from the end user, who does not need to know that staging is used under the hood. While not all macros can be written using staged programming, Stageleft is a great way to improve the correctness of macros that fit the staged model.

5.6.2 Independent Crates

For staged libraries that need to control how the staged code is compiled and run, Stageleft also provides a way to generate independent Rust binaries from staged code. While this approach requires more manual intervention from the library author, it allows the library to control the end-to-end staging process rather than relying on the developer to invoke a staged macro.

Our approach to independent compilation is based on the trybuild [154] library, which was originally intended to provide a way for Rust authors to write unit tests for the typechecking behavior of their libraries. Under the hood, trybuild generates an independent Rust crate containing the code to be tested, and then runs the Rust compiler on that crate to check the type errors. The trybuild library handles key components that we need in staging, such as efficient re-use of the existing compiler cache and propagating Rust compiler flags. Stageleft uses a fork of trybuild that preserves many of its core compilation features but allows library authors to plug in their own code generation logic.

To splice staged code into a new crate, library authors can use the existing `splice` API to obtain the tokens for a quoted expression. Then, they can wrap the generated code into entrypoints such as a `main` function or a `lib.rs` file. This file is then handed off to the trybuild library, which generates a new Rust crate with the provided code and compiler configuration derived from the original crate that is using the library.

Once the independent Rust crate is generated, the library can programmatically invoke the Cargo build tool to compile the staged code into a Rust binary, and can even invoke it. Although Stageleft does not yet provide mechanisms to dynamically load the compiled code into the existing process, we believe that this is a promising direction for future work to bring Stageleft closer in capability to staging libraries for other languages.

5.7 Related Work

Staged programming is a well-studied topic with implementations across many languages. Stageleft takes heavy inspiration from these implementations and mirrors many of their decisions, but focuses on supporting Rust without requiring changes to the compiler. In this section, we review some of the most relevant work in this space.

5.7.1 Staging-as-a-Library

While staging is now built into many modern programming languages, early implementations were often built as libraries on top of existing languages using their metaprogramming features. A particularly prominent example of this is Lightweight Modular Staging (LMS) [126], which is a Scala library that adds staging constructs without requiring special compiler support. LMS has been applied to main similar domains as Stageleft, including query processing [128] and streaming frameworks [82].

LMS uses Scala's macro system to provide a high-level API for quoting and splicing code, and uses the Scala compiler API to execute the generated code. Stageleft borrows much of its design

from LMS. Stageleft uses the `Quoted` type to capture expressions similar to the `Rep` type in LMS and inserts let bindings for references to free variables like LMS does when interpolating quoted code. Because Scala offers implicit conversions, LMS can *automatically* quote snippets of Scala code without the developer having to explicitly annotate them. Because Rust does not have a similar feature, Stageleft requires quoted code to be wrapped in the `q!` macro.

A common criticism of the LMS approach is that it incurs a latency penalty when compiling the generated code. When using Stageleft to generate an independent crate, we face a similar limitation as the Rust compiler must process the generated program. Recent work has focused on unifying macros and staging [139], which allows the staged code to be expanded at compile-time. Stageleft's macro endpoints feature enables a similar approach, but requires more manual separation of the staged code from the rest of the program. With support for dependent types on the horizon for Rust, it may soon be possible to further unify macros and staging in Rust by tracking staged snippets in the type system [91].

5.7.2 Metaprogramming in Rust

Today, Rust offers two vastly different macro systems: declarative macros and procedural macros. Declarative macros can be defined in the same crate as the code that uses them, but are limited to a simple pattern matching language. Procedural macros require low-level token manipulation and are defined in a separate crate, but allow for arbitrary code generation logic. Rust developers today face a challenging tradeoff between the two systems, which have vastly different semantics and limitations. Staging offers an opportunity to unify these systems by providing a mechanism that is higher-level than procedural macros but richer than declarative macros.

A key area of focus has been the *security* of macro systems in Rust [152]. Procedural macros in Rust can contain arbitrary Rust code, including logic that reads files and launches other programs. With growing concern about supply-chain attacks, the Rust community has been exploring ways to prevent rogue macro logic from exploiting development machines. A promising direction is to restrict procedural macros to *const functions* [149], a special function type that restricts the internal logic to be deterministic and have no side effects. Const functions are interpreted at compile-time, making them an excellent candidate for lifting staging into a first-class language feature that does not require the macro indirection.

5.8 Summary

In this chapter, we presented Stageleft, a Rust library for staged programming that allows libraries to leverage code generation logic in a type safe manner. Stageleft is designed to be easy to use and integrate into Rust libraries, while providing strong guarantees about the correctness of the generated code. Stageleft leverages traits in Rust to provide features such as referencing free variables and inlining constants without requiring support from the compiler. We showed how programs written in Stageleft can be exposed as macros or independent Rust binaries, enabling use across many application domains to improve performance with domain-specific compilation.

Part III

Optimization and Verification

A strong semantic foundation and a practical programming model give us an opportunity to explore new techniques for optimizing distributed systems. Traditionally, optimizations for distributed systems have centered around specific domains, such as analytical processing in Spark [165]. Because Hydro is a general-purpose distributed framework, optimizers over its IR can have an impact across a broad range of applications. In the next two chapters, we explore two promising directions: **program synthesis** and **term rewriting**.

Shared, mutable state is a fundamental challenge of distributed systems. In Hydro, state can be accumulated through operators such as `fold` or `reduce`, which collapse streamed elements into a single value which can be read by downstream logic. To ensure fault tolerance and scale reads, developers may want replicate this state across a cluster. But preserving the sequential semantics of an operator such as `fold` typically requires a coordination protocol such as Paxos [93], which comes with a significant performance cost.

Conflict-free replicated data types (CRDTs) [131] are a powerful solution to this problem. CRDTs allow for concurrent updates to shared state and support coordination-free, fault-tolerant reconciliation of state across machines. However, they are difficult to implement correctly, since they must satisfy complex mathematical properties of convergence. In Chapter 6, we present **Katara**, a system that uses verification techniques to synthesize CRDTs that mimic the behavior of a sequential data type, like one implemented in `fold`. Katara only requires the developer to define an ordering relation for how concurrent operations should be applied, and then synthesizes a CRDT that is guaranteed to be observationally equivalent to the sequential data type.

While program synthesis techniques can enable dramatic performance improvements, they are expensive and often require developer guidance. To build scalable optimizers for Hydro, we turn to *rewrite-oriented* optimization systems. This approach focuses on automatically applying low-level, compositional program transformations to discover complex, provably correct optimizations without involving a verifier.

In Chapter 7, we explore how rewrites can enable optimizations that use *incremental computation* [24, 163]. Our key innovation is "time-travelling rewrites", which use semantics inspired by nested graphs in Flo to reason about computation on historical inputs. However, this results in an *infinite* space of optimizations, since we can apply these rewrites at any point in the graph. To tackle this, we apply **e-graphs** [115, 117, 161], which efficiently capture infinite spaces of program rewrites. By composing simple, easy to verify rewrite rules, we can automatically derive classic incremental algorithms such as semi-naive evaluation.

Chapter 6

Katara: Optimizing Data Replication with CRDT Synthesis

6.1 Introduction

In today's interconnected world, there is an ever-growing need to write correct, scalable distributed programs that can serve users at any location with low latency. Many such applications rely on *distributed state*, which in turn is often *replicated* at multiple locations. Replication addresses many common concerns in distributed systems: it can lower latency by keeping a copy of data close to each client, improve availability by increasing the odds that some replica is on a reachable machine, and enhance the scalability of request handling by allowing the overall load of requests to be partitioned across replicas.

However, programmers are trained to write sequential programs and often struggle to write correct distributed programs that make concurrent updates to replicated state. As programs execute, nodes may update their replicas at different times or in different orders, which can cause replicated state to diverge and often results in erroneous application behavior. Our broad goal, first articulated in [35], is to get the best of both worlds: allow developers to *write familiar sequential code* and use *program synthesis to lift* that code to an efficient distributed implementation.

A classic solution to bridge the gap between sequential and distributed semantics is to introduce **coordination**. Coordination allows all replicas to agree upon the order of execution; as a result, each replica can delay the application of early-arriving operations, which, if applied eagerly, would lead to divergence. Protocols for coordination (e.g. Paxos [93], Raft [119], and Zookeeper [71]), offer a general-purpose solution to preserving sequential execution. However such approaches are prohibitively expensive for many applications—particularly at global scale [43, 64, 92]. Such coordination weakens the benefits of replication, as executing operations issued to single replicas now involves high-latency communication with other nodes. Thus, *avoiding* coordination has become a popular approach in the design of modern distributed systems [35, 51].

One of the most widely-adopted approaches for designing coordination-free programs is the use of **conflict-free replicated data types (CRDTs)** [131]. Rather than relying on coordination

to decide the order in which operations execute, CRDTs instead choose to limit their operations to those which *commute*; if all replicas of a CRDT see the same set of operations, then that CRDT will always *converge* to the same state, eliminating the threat of replica divergence. As a result, the system will *eventually* reach a *consistent* state at each replica. For applications that can accommodate this *eventual consistency* [157], CRDTs enable coordination-free state replication. CRDT interpretations of traditional sequential datatypes, including shopping carts, maps, sets, and logs, have found wide adoption in both academia [85, 159] and industry [66, 86, 125].

CRDTs provide a framework for coordination-free replication, but it is left to developers to design individual CRDTs that capture *application-specific* data models. For many, this is out of reach as ensuring convergence and semantic correctness is challenging even for experts [83]. Furthermore, specifying CRDTs is difficult, as the corresponding operations on sequential types are often *not* commutative. For example, one may wish to replicate a set that supports both insertions and removals, in which the order of insertions and removals is critical to the final state. The desire to use such data types has given rise to a class of CRDTs that *almost* match the behavior of a sequential data type. For example, remove operations will "appear" to evaluate before concurrent add operations in the Add-Wins Set CRDT, regardless of the order in which those operations arrive at each replica. These semantic differences make such CRDTs hard to verify [57]. Moreover, these CRDTs require complex logic to capture the effects of operations while ensuring that replicas ultimately converge.

In this chapter, we introduce Katara[1], an open-source system that automates the process of CRDT creation by leveraging **verified lifting** [6, 36, 79]. Using program synthesis techniques, we *lift* annotated implementations of sequential data types in languages like C/C++ to full implementations of nearly equivalent CRDTs. The sequential data types being lifted appear as standard data structures in traditional software, can include constructs such as branching and loops, and do not need to come equipped with custom convergence properties.

Users need only to annotate their sequential data types with a simple function that defines the order in which conflicting operations should *appear* to occur. For example, when lifting a set data structure, a user may choose to order removal operations before all concurrent addition operations, specifying the semantics of an Add-Wins Set. We then *automatically verify* synthesized CRDT candidates against a combination of the sequential semantics and user-specified conflict resolution policy. Crucially, such policies are easy to specify, requiring only a handful of lines in all our examples. By automating the process of verifying a candidate design, we can generate implementations of the CRDT's state, operations, and queries without any intervention.

CRDT designs can be split into two categories: op-based CRDTs and state-based CRDTs. In this chapter, we synthesize **state-based CRDTs (CvRDTs)**, as these can be deployed in more environments and can always be translated to op-based CRDTs if necessary [130]. State-based CRDTs are defined by a datatype representing their state, and an associative, commutative, and idempotent (ACI) *merge function*, which determines how the states of replicas are combined to reach convergence. The state type and merge function together form a join semilattice, with the merge function serving as the join.

[1] https://github.com/hydro-project/katara

Early CRDT work involves complex state structures, but it was subsequently observed that CRDTs can be assembled via composition of simple join-semilattices on sets, integers, or Booleans [42, 162]. We leverage this approach in the context of synthesis. By limiting the search space of state types to compositions of join-semilattices, we are able to achieve convergence—normally the most difficult property of CRDTs to verify—*entirely by construction.*

With this intuition, we introduce new algorithms for searching the space of possible CRDT implementations, including the state representation of the CRDT, operations defined on it, and the queries by which its state may be observed. This includes the design of grammars for runtime logic that we search with a Syntax-Guided Synthesis engine [8] and a parallelized enumerative search over compositions of semilattices for the state structure used within the CRDT. We apply multiple SMT encodings of our correctness conditions to quickly prune the CRDT search space and perform unbounded verification. Katara is able to automatically generate a variety of practical, provably correct CRDT designs for a wide range of specifications.

To summarize, we make the following contributions:

- We define a CRDT's correctness in terms of its operations and queries, and demonstrate how users can specify CRDTs by augmenting a sequential data type with lightweight ordering constraints that resolve conflicts between non-commutative operations (Section 6.3).
- We introduce an SMT encoding of our correctness conditions that enables automated verification of CRDTs against sequential data types with ordering constraints, along with a bounded variant that enables efficient pruning of the program search space (Section 6.4).
- We design a synthesis algorithm that efficiently searches semilattice compositions for the internal state of the CRDT, creates grammars for runtime components that guarantee convergence, and applies syntax-guided synthesis to generate both the core logic and invariants that prove correctness over unbounded executions (Section 6.5).
- We describe a practical implementation of Katara, including the details of how we automatically generate verification conditions from sequential data type implementations in C/C++ and optimize performance by synthesizing several candidate CRDTs in parallel (Section 6.6).
- We demonstrate how Katara can automatically lift sequential data types into practical CRDTs, and generate alternative designs with behavior equivalent to human-designed CRDTs in existing literature (Section 6.7).

6.2 Motivating Example

The distributed shopping carts problem, made popular by Amazon's Dynamo [51], is an essential business problem with a clever coordination-free solution. In the original formulation of this problem, the authors track a mapping of items to non-negative counts representing how many of that item are in the cart. Users can interact with the shopping cart by requesting insertions

and removals of items. The cart can also be queried to determine the count of each item during a checkout procedure. The goal is to replicate a single shopping cart across many distributed nodes to improve fault tolerance, while ensuring eventual consistency so that the accumulated states on any node can be used to query the complete cart.

Let us focus on a simplified version of this problem, where each item can only be in the cart at most once—effectively simplifying the cart to a set of items. A developer without a distributed systems background could attempt to implement this as a replicated data type by having the insertions, removals, and queries all operate on a standard hash-set. However, if we deploy this in a distributed setting, we will immediately begin to see issues.

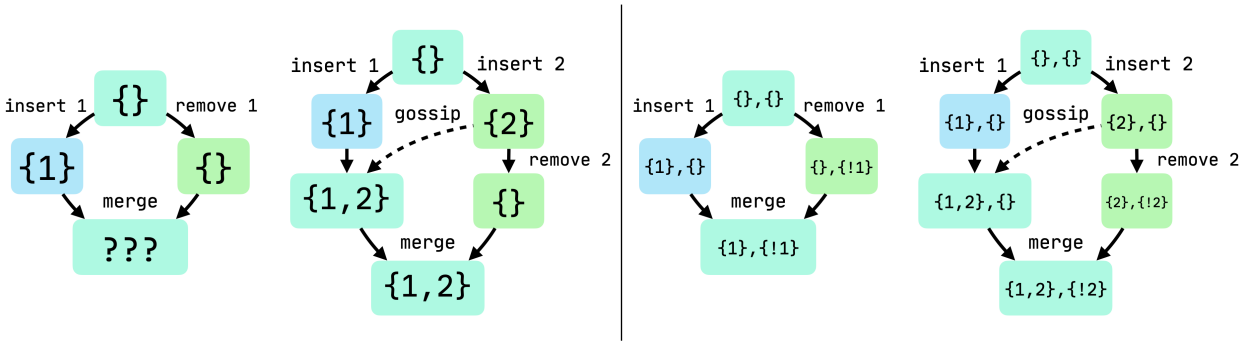


Figure 6.1: A sequential type (left) has consistency issues that are resolved by a CRDT (right).

Consider the leftmost scenario in Figure 6.1, where a shopping cart is replicated across two nodes. When a user sends operations to add and remove the same item, these requests are distributed between the nodes. The node that receives the insert operation will add the item to its local set, but the node that receives the remove will treat it as a no-op because its local set is currently empty. We may wish to merge the state at these nodes via set union, but this would fail to preserve the locally-ineffective remove operation; it is unclear if this matches user expectations.

If we periodically share state via gossip [52], new situations emerge where consistency is disrupted. On the left of Figure 6.1, we see an execution where one node processes an insert, and the other processes an insert and then remove of the same element. Even if we merge via set union, we still end up with non-deterministic results depending on when gossip takes place. If the state of the right node is gossiped to the left after the insert but before the remove, the left node will merge the new value into its local state. Even after the remove is processed, merging the two states together will still result in the element 2 being present. But if the gossip were not to take place, the merged state would only have the element 1.

Situations like these have severe correctness implications, and it can be challenging to reason about the many ways a distributed system can break sequential assumptions built into a data type. Furthermore, it is challenging for developers to fix such correctness issues, because doing so involves reasoning about all possible interleavings of operations and their possible effects. Our

work aims to tackle this issue by automatically synthesizing a CRDT, which satisfies the property of *convergence* and can be safely replicated in a distributed cluster without requiring coordination. Let's explore how a user would synthesize a shopping cart CRDT with Katara.

Our synthesis algorithm takes a sequential data type that defines the semantics of operations and queries the CRDT will support, and an ordering constraint that specifies how to resolve conflicts between non-commutative operations. We already have the first, since the user has implemented a sequential, single-node shopping cart. For the second, the non-commutative operations in the sequential type are inserts and removes, so the developer may decide that they want the CRDT to resolve conflicting operations by having the removes "win." This can be encoded as a simple pairwise ordering constraint (*opOrder*) that is passed into the synthesis algorithm.

Once Katara is given this specification, it searches potential state types and runtime logic that are both behaviorally correct and convergent. Along the way, our synthesis algorithm prunes out candidates by using bounded verification to quickly check correctness against short sequences of operations. Eventually, the synthesis engine will produce a provably correct CRDT. For our running example, we might get the CRDT in Figure 6.2 with an internal state of two sets $(s1, s2)$.

user input

```
set* init_state() { return set_create(); }

set* next_state(set* state, int add, int v) {
  if (add == 1) return set_insert(state, v);
  else return set_remove(state, v);
}

int query(set* state, int v) {
  return set_contains(state, v);
}
```

$opOrder(o_1, o_2)$: $o_{1,add} = 1 \vee o_{2,add} \neq 1$

synthesized design

crdt *ShoppingCart*
  initialState: $(\{\}, \{\})$
  merge $(a_1, a_2), (b_1, b_2)$
    return $(a_1 \cup b_1, a_2 \cup b_2)$
  operation $(s, add, value)$
    return $merge(s,$ **if**
      $add = 1$ **then**
        $(\{value\}, \{\})$
      else
        $(\{\}, \{value\})$
    $)$
  query $((s_1, s_2), value)$
    return $value \in (s_1 \setminus s_2)$

Figure 6.2: A user-provided sequential reference and the CRDT design synthesized by Katara.

Readers who are familiar with literature on CRDTs may recognize the implementation above as a Two-Phase Set [130], one of the classic CRDTs that mimics the behavior of a set while guaranteeing convergence in distributed execution. This synthesized implementation is provably correct for the given sequential specification with the operation reordering, so we can deploy it in our distributed application without having to worry about manually proving complex CRDT properties. We can see on the right of Figure 6.1 that our CRDT now consistently handles the execution graphs that had conflicts with the naive implementation. Without any baked-in knowledge of existing CRDTs, Katara is able to automatically generate such implementations that were previ-

ously only designed by distributed systems researchers, and can even generate new undiscovered designs by searching the wider space of CRDT structures.

6.3 Specifying CRDTs with Sequential Data Types

In verified lifting, a key piece of the puzzle is specifying correctness of the synthesized program in terms of the reference code. Past work has been focused on transpiling legacy functions into high-level DSLs, a domain in which correctness has a relatively simple definition: the synthesized logic must produce the same output as the original code for any valid input. When lifting CRDTs, however, our challenge is different: we are not just finding a function that produces the correct input/output pairs, but rather finding a *stateful* data structure that produces the correct outputs for *unbounded sequences* of method invocations—including mutations and queries. In this section, we develop a formal model for sequential data types and CRDTs and introduce the correctness conditions for a CRDT to match a sequential specification.

We model sequential data types as a combination of two functions: a state transition ($st(s, o)$) that takes in the current state and an operation (a tuple of client-provided parameters) and returns an updated state, and a query function ($query(s, q)$) that takes the current state and a query (similarly, a tuple of client-provided parameters) and returns some data of any type. Sequential types also define an initial state ($initialState$) that is updated as operations are processed. This model can handle a wide range of sequential data types, including those that do not separate updates from queries, since independent operation/query endpoints can be combined into a single $st/query$ function and we do not restrict the logic inside those functions.

On the CRDT side, we have a similar interface with one additional function to handle gossip from distributed nodes. In our discussion, we will distinguish the functions corresponding to a CRDT candidate from the sequential data type by attaching an asterisk to the CRDT names. Because the CRDT uses a different state type than the sequential structure, we also mark CRDT states with an asterisk (s^*). Just like before, we have an initial state ($initialState^*$), state transition ($st^*(s^*, o)$) returning a value of the CRDT state type, and query ($query^*(s^*, q)$) which together model how the CRDT processes requests over time. In addition, we introduce a merge function ($merge^*(s_1^*, s_2^*)$), which is used to merge a node's local state with gossip received from other nodes so that the replicated data converges.

CRDTs have two key components to their correctness: they must respond appropriately to operations and queries, and they must converge under eventual consistency. We guarantee the convergence by construction with grammar restrictions on the state (Section 6.5.1) and runtime logic (Section 6.5.2). In this section, we focus on specifying correct operations and queries by comparing the behavior of the CRDT to a reference sequential data type.

6.3.1 Specifying CRDTs with Operation Sequences

Because the sequential data type and the CRDT may use different internal state representations, we cannot directly compare instances of them by comparing their states. As a result, our def-

inition of correctness must reason about the *user-observable behavior* of the data type over *unbounded sequences of interactions*. Both the sequential type and the CRDT have two components a user can interact with: the state transition and the query. Since queries are the only way for users to observe data, we want to ensure that after processing any sequence of user interactions, the sequential type and the CRDT respond identically to any query. Because queries do not modify the state, we can simplify this condition: a sequential type and CRDT are equivalent if both return the same result to an arbitrary query after processing an arbitrary sequence of operations.

But we have to go a step further to justify this correctness definition, since CRDTs can be executed in a distributed system. When we replicate the data type, operations will nondeterministically arrive at nodes in different orders. In addition, the use of gossip protocols in the cluster results in additional state updates when a node merges its local state with a state received from another node. As a result, instead of having a totally-ordered sequence of operations taking the initial state to the final one, we instead have a partial order—a directed acyclic graph—that captures the many different orderings of operations that could be seen by different replicas.

Thankfully, operations on a CRDT that satisfies an eventual consistency policy are commutative: the CRDT state depends only on the *unordered set* of observed operations. Even for CRDTs with causally consistent semantics, operations can be made commutative by extracting sources of causality such as timestamps into operation parameters, making them commutative *modulo* the causality [113]. Therefore, as long as the provided CRDT is convergent and has commutative operations, we can reduce verification of a distributed CRDT execution to the sequential case via an arbitrary flattening of the execution graph.

6.3.2 Resolving Commutativity with Operation Orderings

So far, our correctness conditions require strict equivalence of the sequential data type and the synthesized CRDT. However, the additional requirement of operator commutativity on the CRDT means that many sequential types with non-commutative operations, including common ones like sets and maps, cannot directly correspond to an equivalent CRDT but instead are mapped to many popular CRDT variants that make different semantic compromises.

In Katara, users can define these semantic adjustments through **operation orderings**, which loosen the correctness requirements to only verify sequences of operations following a specific ordering constraint. This approach, reminiscent of past work on CRDT specification [28], minimizes user effort (by leveraging verified lifting) and enables automated verification (Section 6.4). Formally, a user can define a partial order $opOrder(o_1, o_2)$, which returns true when a call to o_2 is allowed to occur after a call to o_1.

As an example of the effect of this ordering, recall the shopping cart we lifted in the motivating example. In our sequential data type, inserts and removals do not commute, so no CRDT exists that strictly matches its semantics. However, we can resolve the conflict by introducing an ordering between the operations. If we specify that removes take place before inserts, we get a specification of a Grow-Only Set, which treats removes as no-ops. On the other hand, if we specify that removes take place after inserts, we get a specification of a Two-Phase Set, where elements can be inserted, then removed, but not inserted again.

Instead of having to consider all potential interleavings of operations in a distributed system, operation orderings make it possible for users to specify the distributed behavior in terms of a *sequential execution model*. This makes it possible for non-experts to use our synthesis algorithm, since they can reason about the operation ordering with the existing sequential data type.This general approach of ordering operations to resolve commutativity allows us to synthesize a wide variety of CRDTs by applying different orderings to simple sequential data types. Intuitively, ordering the non-commutative operations of the sequential type transforms the semantics to be effectively commutative, since any ordering of non-commutative operations will always be reordered into the same sequence by the *opOrder* constraint. Since the CRDTs we are verifying are already guaranteed to have commutative operations, it then suffices to verify correctness with sequences of operations that follow this order.

6.3.3 Operation Orderings with Time

A popular pattern when designing replicated data types is to place "distributed timestamps" on all operations, which makes it possible to introduce sequential semantics without losing convergence. For example, the Last-Writer-Wins Set is a classic CRDT that timestamps its operations; each operation supersedes any conflicting operations that have strictly earlier timestamps. Because distributed timestamps are only a partial order, there can be conflicts among operations that are incomparable in time, which are handled with remove- or add-wins semantics.

When a user provides a sequential data type and ordering specification, we allow them to enable a flag to introduce timestamps to each operation. This flag augments every operation with an integer timestamp o_t, which is computed by the local node at runtime using a source that can be mapped to an integer value. We use Lamport timestamps [94] as this source, which guarantees that causally ordered events will have accordingly ordered integer timestamps. We then augment the *opOrder* to order operations first by their timestamp, and then apply the user-defined ordering on operations with the same timestamp:

$$opOrder(o_1, o_2) \triangleq (o_{1,t} < o_{2,t}) \vee ((o_{1,t} = o_{2,t}) \wedge opOrder_{orig}(o_1, o_2))$$

When we introduce time, we also must introduce constraints on the operations we consider in our correctness conditions to avoid degenerate cases with illegal timestamps. We do this through an additional user-defined function $opPrecondition(o)$, which checks that an operation is valid. When timestamps are enabled, we define the precondition as $opPrecondition(o) = o_t > 0$ to ensure the operations we check have valid timestamps. Users can also add constraints to this precondition based on domain knowledge, such as if an operation parameter will always be positive.

6.4 Automated CRDT Verification

Now, we must encode this formal definition of CRDT correctness to enable the automated verification of candidate CRDT designs. We tackle this by encoding correctness in SMT logic, which allows us to use solvers like Z3 [46] and CVC5 [21] to automatically prove correctness or find

counterexamples. However, these solvers cannot directly reason about unbounded sequences of operations, so we must break down the correctness conditions into an inductive proof that reasons about individual state transitions.

6.4.1 State Equivalence

First, we focus on checking CRDT correctness without considering the ordering constraint. To build this inductive proof, we need a way to reason about the relationship between the states of the sequential data type and candidate CRDT after processing the same, arbitrary sequence of operations. To do this, we choose to relate the states of the CRDT and sequential data type implementations in the style of a bisimulation.

(1) $equivalent^*(initialState, initialState^*)$
(2) $\forall s, s^*, o : (equivalent^*(s, s^*) \land opPrecondition(o)) \implies equivalent^*(st(s, o), st^*(s^*, o))$
(3) $\forall s, s^*, q : equivalent^*(s, s^*) \implies query(s, q) = query^*(s^*, q)$

Figure 6.3: The verification rules that constrain CRDT synthesis to preserve the source semantics

We encode this proof by introducing the **state equivalence** function, which relates the states of the sequential reference and synthesized CRDT. Formally, if the reference and synthesized data types are in equivalent states, then after both process an arbitrary sequence of operations they will return the same result to any query. Intuitively, the equivalence function describes which states of the sequential data type correspond to states of the CRDT that capture the same queryable knowledge. Furthermore, the equivalence function captures invariants about the CRDT that filter unreachable states from the verification conditions. With this function available as the inductive invariant of our bisimulation, we can now encode our correctness conditions in SMT logic.

We begin our conditions in Figure 6.3 with rule (1), that the initial states of the sequential data type and CRDT must be equivalent. Next, we build the inductive proof that carries equivalence all the way to the final query. We start by encoding the query constraint in rule (3), where we query the reference and synthesized implementations in equivalent states. For this condition, the query results are of the same type so we can directly check for equality. Intuitively, this rule requires equivalence to be a guarantee that queries on the two states return the same result. However, since equivalence deals with not only the current state but also queries on the future states, the equivalence condition may need to be stronger. The condition that forces this strengthening is the inductive step of our proof in rule (2), which checks that if the two data types are in equivalent states, then after executing the same operation they should still be in equivalent states.

6.4.2 Enforcing Operation Orders with Invariant Synthesis

So far, our verification conditions ignore the presence of the user-defined operation ordering (*opOrder*), which specifies how the CRDT should handle conflicting non-commutative operations. In Katara, we implement two encodings of the constraints imposed by this ordering: one that supports unbounded verification but requires synthesizing additional invariants, and one that is only suitable for bounded verification but enables efficient exploration of the program space. In our end-to-end algorithm (Section 6.5.3), we use the bounded encoding first to quickly prune out candidate state structures.

To introduce the ordering constraint to the unbounded verification conditions, we take the approach of strengthening the inductive hypothesis by synthesizing additional invariants. The key insight in this approach is that CRDT states are accumulated by merging updates produced by each operation, so we can enforce orderings that use simple comparisons (such as equality or greater/less than) on the *accumulated state* instead of the individual operations in the history of the CRDT. For example, in the shopping cart scenario where inserts are ordered before removes, we know that an insert is in-order when the set of removed elements is empty.

Formally, we introduce another synthesized Boolean function $orderWithState^*(s^*, o)$, which returns true if the operation o satisfies the ordering constraint against the history of operations *implied* by the state of the CRDT. This function must be true when executing any operation in a correctly ordered sequence, which in turn ensures that the CRDT correctly handles all executions that satisfy the ordering constraint. By enforcing the user-provided ordering in terms of just the CRDT state, for which we already have inductive invariants, we are able to completely avoid the issue of separately reasoning about the history of operations that have been applied to the CRDT; we directly prove correctness in the unbounded case.

(1) $\forall o : equivalent^*(initialState, initialState^*) \wedge$
$(opPrecondition(o) \implies \boldsymbol{orderWithState^*(initialState^*, \mathbf{o})})$

(2) $\forall s, s^*, o_1, o_2 : (equivalent^*(s, s^*) \wedge opPrecondition(o_1) \wedge \boldsymbol{orderWithState^*(\mathbf{s}^*, \mathbf{o_1})})$
$\implies (equivalent^*(st(s, o_1), st^*(s^*, o_1)) \wedge ((opOrder(o_1, o_2) \wedge opPrecondition(o_2)) \implies$
$\boldsymbol{orderWithState^*(\mathbf{st}^*(\mathbf{s}^*, \mathbf{o_1}), \mathbf{o_2})}))$

(3) $\forall s, s^*, q : equivalent^*(s, s^*) \implies query(s, q) = query^*(s^*, q)$

Figure 6.4: The verification rules updated to use a synthesized invariant for ordering constraints

To start, we update the base case to require that any operation executed in the initial state must be in-order. Then, we update the inductive step to enforce the correctness of $orderWithState^*$ on *all* operations in an ordered sequence, by extending rule (2) to reason about pairs of adjacent operations. We introduce a precondition that checks if the first operation being executed is in-order with the state, and enforce the transitive property that o_2 be in-order with the state after

o_1 is processed if it is pairwise in-order after o_1. When these conditions are satisfied, we have a proof that our CRDT matches the sequential data type under the operation ordering for any unbounded execution.

6.4.3 Optimizing Verification with Quantified Queries

Because it is critical to verifying a CRDT candidate, the *equivalent** invariant must be synthesized alongside the other runtime logic. But with equivalence defined in terms of just the reference and synthesized states, synthesis can quickly become infeasible when the internal states involve large, unbounded structures such as sets and maps (Section 6.5.1), which would require the synthesizer to generate higher-order logic like reductions to compare the states. But we can significantly reduce this burden by noticing that beyond the inductive step, equivalence is only used as a precondition for checking that the sequential data type and CRDT return the same response for a *specific query*. Therefore, we can reduce the synthesis burden by introducing an additional parameter q to *equivalent** so that it is only responsible for checking that the states are observationally equivalent *for a given query*.

(1) $\forall o, \mathbf{q} : equivalent^*(initialState, initialState^*, \mathbf{q}) \wedge$
$(opPrecondition(o) \implies orderWithState^*(initialState^*, o))$
(2) $\forall s, s^*, o_1, o_2, \mathbf{q} : (equivalent^*(s, s^*, \mathbf{q}) \wedge opPrecondition(o_1) \wedge orderWithState^*(s^*, o_1))$
$\implies (equivalent^*(st(s, o_1), st^*(s^*, o_1), \mathbf{q}) \wedge ((opOrder(o_1, o_2) \wedge opPrecondition(o_2)) \implies$
$orderWithState^*(st^*(s^*, o_1), o_2)))$
(3) $\forall s, s^*, q : equivalent^*(s, s^*, \mathbf{q}) \implies query(s, q) = query^*(s^*, q)$

Figure 6.5: The verification conditions with the additional query parameter for equivalence

By giving the equivalence function a specific query, the synthesized logic can now focus on comparing the parts of each state that are relevant to that query. For example, when synthesizing a CRDT that uses maps, this can result in significant simplifications like only checking one key. We update the verification conditions by adding a new quantifier for the query to rules (1) and (2). In rule (3), we simply pass the existing query variable to the *equivalent** function. We define these updated conditions in Figure 6.5. An additional optimization this enables in Section 6.5.2 is to move the postcondition of rule (3) into the structure of *equivalent**, which further reduces the burden on the synthesizer since it would otherwise have to discover this condition by itself.

6.4.4 Solution Pruning with Bounded Operation Logs

The unbounded verification conditions, while necessary to prove correctness for the CRDT, have a large performance impact on synthesis since they require both the CRDT and *orderWithState** to

be synthesized simultaneously. To reduce this impact, we employ a two-phase synthesis approach where we synthesize the core logic of candidate CRDTs with verification conditions that check a bounded number of operations (and therefore do not require additional invariants), and separately synthesize the invariants to prove unbounded correctness.

There is one key modification to the base verification rules that we need to make: the state transition should only be checked when the operation being processed is in-order according the user-provided function. To encode this, we add an additional variable to the verification conditions (σ) that stores a bounded log of operations that have been processed. Because this operation log has a statically known bound, we can lower it to a fixed set of variables corresponding to each element of the list and avoid having to involve more complex theories. Note that the log cannot be used by any of the synthesized logic, since its only role is to aid verification.

$$\text{list in-order/valid} : lio(\sigma) \triangleq \forall i : (i < |\sigma|) \implies (opPrecondition(\sigma[i]) \wedge ((i < |\sigma| - 1) \implies opOrder(\sigma[i], \sigma[i+1])))$$

$$\text{list coherent} : lc(s^*, \sigma) \triangleq s^* = fold(\sigma, initialState^*, st^*)$$

(1) $\forall q : equivalent^*(initialState, initialState^*, q)$

(2) $\forall s, s^*, \boldsymbol{\sigma}, o, q : (equivalent^*(s, s^*, q) \wedge opPrecondition(o) \wedge$
$\boldsymbol{lio(\sigma) \wedge lc(s^*, \sigma) \wedge opOrder(\sigma[|\sigma| - 1], o)})$
$\implies equivalent^*(st(s, o), st^*(s^*, o), q)$

(3) $\forall s, s^*, q : equivalent^*(s, s^*, q) \implies query(s, q) = query^*(s^*, q)$

Figure 6.6: The verification rules updated to use operation logs for bounded ordering constraints

Then, we update the state transition verification rule to add a precondition that the operation log is in-order and coherent with the state of the CRDT. First, we check that every pair of operations in the log are in-order, since the operation log is a quantified variable in the SMT encoding and may have out of order values. Then, we verify that the synthesized state equals the result of folding over the log with the synthesized state transition. Since the operation log is bounded, this collapses into a bounded number of state transitions and can be efficiently verified by an SMT solver. Finally, we add a condition that the operation being applied in rule (2) is in-order with the existing log, which can be checked by comparing it against the last operation (since the ordering constraint is transitive).

Note that we do not need to introduce the ordering and coherence preconditions to the query verification conditions in rule (3) even though that rule operates on arbitrary input states. Because we still synthesize the *equivalent** function to relate the CRDT and sequential data type, we do not need to constrain *how* we reach the synthesized state being evaluated, just that any instance of

the sequential reference deemed equivalent will return the same response to the query. It is left to the synthesizer to introduce any invariants necessary to avoid checking queries on unreachable states. Together, these rule modifications are summarized in Figure 6.6. With bounded operation logs, this encoding enables efficient synthesis of the state transition and query functions.

6.5 CRDT Synthesis Algorithm

Now that we have a formal specification of correctness that can be verified by an SMT solver, we are ready to define the synthesis algorithm for CRDT implementations. Our end-to-end algorithm requires only two pieces of user input: the sequential data type written in a standard imperative language and the operation ordering that resolves conflicts. Our synthesis algorithm optionally takes a set of Boolean flags that enable synthesis of advanced designs, such as those that use timestamps (discussed in Section 6.3.3) or have non-idempotent operations (which we explore later in this section).

There are four core components to synthesize: the type of the internal state, the initial state, the state transition function, and the query function. We must also synthesize the *equivalent* and $orderWithState$ invariants from the previous section to enable verification. As discussed before, the synthesized CRDT may use a completely different state structure than the source, which adds a new layer of complexity because the choice of state type affects the search space for each synthesized function. Therefore, our synthesis algorithm uses multiple phases to generate candidates of runtime logic for a range of potential state types.

In Section 6.3, we explained that our verification conditions only check the user-observable behavior of the CRDT, but do not verify that the CRDT implementation meets the convergence properties. Instead of checking these properties through verification conditions [113], we craft our CRDTs in a way that satisfies these properties *by construction*. Inspired by past work on designing coordination-free distributed systems [42, 162], we synthesize CRDTs that use **semilattice compositions** for their internal state, which makes it straightforward to enforce monotonicity and commutativity since these are properties of the semilattice join.

6.5.1 State Synthesis

To explore candidate state structures for the CRDT, we use the classic synthesis approach of defining a grammar and iteratively processing deeper structures. Because we focus on compositions of semilattices, our grammar consists of simple rules for primitives, sets, maps, and tuples.

For primitive types, we include semilattice definitions based on Booleans and integers, which are sufficient to lift a wide variety of sequential data types. For Booleans, we have the `OrBool` lattice, which is a Boolean that has $\bot = false$ and is merged with $\vee$. For integers, we provide the `MaxInt` semilattice, which merges integers by taking the maximum. Beyond the primitives, we include a semilattice definition for `Set<T>`, which can have a non-lattice type `T` for elements; the only constraint on `T` is that it supports equality.

Our lattice definitions for composite data structures are more complex. First, we offer the `LexicalProduct<A, B>` semilattice, where A and B are themselves semilattices. In this semilattice, the first element has priority over the second when determining the ordering of two instances. This type is especially useful for CRDTs that use timestamps to have recent operations override older ones, but need to perform a merge over the underlying values when the effects of concurrent operations are combined. We define the lattice join for `LexicalProduct` as:

$$(a_1, b_1) \sqcup (a_2, b_2) = \begin{cases} (a_1, b_1) & a_1 > a_2 \\ (a_2, b_2) & a_2 > a_1 \\ (a_1 \sqcup a_2, b_1 \sqcup b_2) & \text{otherwise} \end{cases}$$

This definition respects the lattice axioms of associativity, commutativity, and idempotence. Furthermore, these tuples can be nested to form tuples of arbitrary arity. We also support the `FreeTuple<A,B>` lattice, which simply joins elements pairwise (i.e. $(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup a_2, b_1 \sqcup b_2)$) and can similarly be nested to form tuples of arbitrary arity.

In some cases we may not know the desired arity of a `FreeTuple` in advance, or we may not need all the "fields" of such a tuple in a given execution. To address this, we offer a `Map<K, V>` semilattice, where `K` can be any type that supports equality, and `V` is a semilattice. Our maps support common operations such as insertions with the same semantics as regular maps, except when inserting keys that are already present in the map. Instead of overwriting the value, we use the lattice join for the value type to combine the existing value with the one being inserted. This carries over to our definition of the lattice join for maps themselves, where we insert the entries of both maps, with keys that are present in both maps having their values merged according to their join:

$$m_1 \sqcup m_2 = \{k_i : \begin{cases} m_1[i] & (k_i \in m_1) \wedge (k_i \notin m_2) \\ m_2[i] & (k_i \notin m_1) \wedge (k_i \in m_2)\,)\} \\ m_1[i] \sqcup m_2[i] & (k_i \in m_1) \wedge (k_i \in m_2) \end{cases}$$

Again, this definition respects the standard lattice axioms. Given these semilattice types, we can construct the grammar in Figure 6.7 that defines the space of compositions to explore. Our grammar covers a large space of semantics, since the available types encode core capabilities such as free and lexicographically-ordered semilattice products (via maps and tupling) and general semilattice representations (sets). In our end-to-end synthesis algorithm, we explore types in this grammar with iteratively increasing depth bounds and attempt to synthesize the runtime component of the CRDT for each one. Note that we only include `FreeTuple` in the top-level *latticeList* for CRDTs that need multiple semilattices in their state.

6.5.2 Runtime Synthesis

With our state structure selected, we can now move on to synthesizing the runtime logic. Our algorithm for runtime synthesis proceeds in two phases: a first step that synthesizes the core

⟨*latticeList*⟩ ::= ⟨*latticeType*⟩ | FreeTuple(⟨*latticeType*⟩, ⟨*latticeList*⟩)

⟨*latticeType*⟩ ::= OrBool | NegBool | MaxInt
| Set(⟨*type*⟩) | Map(⟨*type*⟩, ⟨*latticeType*⟩)
| LexicalProduct(⟨*latticeType*⟩, ⟨*latticeType*⟩)

⟨*type*⟩ ::= Bool | Int

Figure 6.7: The grammar defining compositions of semilattices we explore during synthesis.

logic with the bounded operation log verification conditions, and a second that synthesizes the additional invariants required for unbounded verification. By using a two-phase approach, we are able to significantly improve the end-to-end synthesis performance of our algorithm by pruning out state structure candidates for which no runtime implementation satisfies even the bounded conditions. In addition, this approach reduces the number of invariants that must be synthesized simultaneously with the CRDT logic, which further improves efficiency.

Core Logic Synthesis

We derive significant power from our choice to implement the internal state of the CRDT via a semilattice. First, we observe that, as lattice join is compositional, we can define the merge function as the lattice join on the internal state; this in turn is derived directly from the state's constituent lattices. Next, we observe that we can also implement *operations* in terms of this lattice join: we define our state transition as $st^*(s^*, o) = merge^*(s^*, f^*(o))$, where f^*, which returns a lattice value of the same type as s^*, is the function that we actually synthesize. This choice grants us monotonicity, commutativity, associativity, and idempotence entirely for free, derived from the lattice join ($merge^*$) itself.

Along with the state transition, we synthesize the query function that is used to pull information out of the CRDT. There are no convergence restrictions on the query since it does not mutate the state, leaving only the sequential reference as a source of constraints on its synthesis. As a result, we do not need to craft the query function in any special way, and can let the synthesis engine drive the search of the query logic.

To support the inductive step of the verification conditions, we must synthesize the equivalence function. This function has two intuitive roles: (1) a *cross-state relation* that identifies which states of the sequential data type and the CRDT are observationally equivalent, and (2) a *CRDT state invariant* that is needed to strengthen the inductive hypothesis of the correctness proof. Following this intuition, we split the synthesis of the equivalence function into components for each role. As discussed when we introduced the query parameter to $equivalent^*$ in Section 6.4.1, we seed the equivalence function with a check that both states respond identically to the given query. This means that our equivalence function has the form $equivalent^*(s, s^*, q) \triangleq query(s, q) = query^*(s^*, q) \wedge relation^*(s, s^*) \wedge invariant^*(s^*)$, where $relation^*$ and $invariant^*$ are synthesized.

Because the embedded query comparison already filters out most states that immediately return different query responses, we can improve synthesis performance with a heuristic that bounds the maximum expression depth of $relation^*$ to one less than the other functions. Even with this optimization, we can synthesize complex $relation^*$ logic when necessary because the depth is iteratively increased. With the bounded operation log encoding, the $invariant^*$ component is unnecessary because we know that the CRDT state can be reached through the explicit log of operations. We will revisit the invariant in Section 6.5.2, when we synthesize the CRDT using the encoding for unbounded correctness that does not use a log.

$\forall T, U, O$

$\langle bool \rangle$ ::= false | true
| $\langle bool \rangle \wedge \langle bool \rangle \mid \langle bool \rangle \vee \langle bool \rangle$
| $\neg \langle bool \rangle$
| $\langle int \rangle > \langle int \rangle \mid \langle int \rangle \geq \langle int \rangle$
| $\langle T \rangle = \langle T \rangle$
| $\langle T \rangle \in \langle Set(T) \rangle \mid \langle Set(T) \rangle \subset \langle Set(T) \rangle$

$\langle int \rangle$ ::= 0 | 1 | $\langle int \rangle + \langle int \rangle \mid \langle int \rangle - \langle int \rangle$
| constants in the sequential source

$\langle Set(T) \rangle$ ::= {} | $\{\langle T \rangle\}$
| $\langle Set(T) \rangle \cup \{\langle T \rangle\} \mid \langle Set(T) \rangle \cup \langle Set(T) \rangle$
| $\langle Set(T) \rangle \setminus \langle Set(T) \rangle$

$\langle Map(T, U) \rangle$ ::= {} | $\{\langle T \rangle : \langle U \rangle\}$
| $\langle Map(T, U) \rangle \cup \langle Map(T, U) \rangle$

$\langle U \rangle$ ::= $\langle Map(T, U) \rangle[\langle T \rangle$, default=$\langle U \rangle]$
| $\langle Tuple(U, T) \rangle[0] \mid \langle Tuple(T, U) \rangle[1]$
| input of type U

if top-level or U is not a Set or Map:

$\langle U \rangle$ ::= if $\langle bool \rangle$ then $\langle U \rangle$ else $\langle U \rangle$

Figure 6.8: The core grammar used to synthesize the state transition and query functions.

The synthesized components of the state transition, query, and equivalence functions all use a common core grammar. Similar to past program synthesis work, we generate the grammar in Figure 6.8 based on the type constraints of supported operations and bound it by an iteratively increased depth, discussed further in Section 6.5.3. Our grammar features the core set of operations available on the types we support in our system, such as arithmetic, Boolean logic, and set/map operations. In addition, we include conditionals in our grammar to support branching inside the synthesized logic. Because branches that emit complex types such as sets or maps are expensive to synthesize, we restrict those to the top-level of the expression and seed the condition with any Boolean inputs and equality comparisons for integer inputs.

The astute reader may notice that this grammar does not enforce any of the ACI properties. But this is not a problem! Recall that we are synthesizing a function $f^*(o)$ that returns a lattice value to be passed into $merge^*$. Therefore, even though some operations in this grammar are not idempotent or commutative, the overall state transition function st^* remains associative, commutative and idempotent by construction. The semantics of the operations in our language

are largely standard, and we lower the operations directly to the corresponding logic in the SMT solver when possible.

Finally, we synthesize the initial state using a shallow grammar of constructors and relevant constants for each type. We include small integer literals, Boolean constants, and empty instances of sets and maps. In cases in which $\perp$ is defined, the initial state is often synthesized to just be the bottom value of the lattice, but occasionally we want the synthesizer to pick an alternate value to handle queries in the initial state. For example, when synthesizing a Boolean register where concurrent enables are ordered *after* disables, the natural lattice to represent the flag's state is a `LexicalPair<ClockInt, OrBool>` with $\perp = (0, false)$, but we need the initial state to be $(0, true)$ if the sequential data type starts in a enabled state. By synthesizing this value instead of fixing it to $\perp$, we can synthesize CRDT designs over semilattices that do not define bottom, or where the initial state starts higher in the semilattice order.

Synthesizing Non-Idempotent Operations

So far, the state transitions we can synthesize are always idempotent, which is not a requirement of CRDTs in general and prevents us from synthesizing designs such as counters. To resolve this limitation, we use a common trick in replicated distributed systems and relax the idempotence constraint while ensuring that certain state can only have a single writer via constraints on the state transition grammar. We introduce **node IDs**, which are unique integer identifiers for each node in the cluster that can be used as map keys to separate portions of the state that are tied to each node. With this separation of writable state, we can synthesize non-idempotent operations that update portions of the state that only the current node can write.

Users can introduce node IDs to the synthesis pipeline by enabling a single Boolean flag when the sequential data type has non-idempotent operations. Because synthesizing CRDTs with non-idempotent operators introduces additional variables and a larger grammar, which can impact performance, the flag is disabled by default and must be explicitly enabled by the user based on their knowledge of the sequential data type. In future work, we hope to automate this process by analyzing the sequential reference to automatically detect non-idempotence.

Enabling non-idempotent operators affects two components of the synthesis algorithm: the structure of the synthesized functions and the grammar used for runtime logic. The synthesized component of the state transition, which previously could only read the operation arguments to ensure idempotence and commutativity, is expanded to have access to the CRDT state as well as the current node ID. As a result, we must now synthesize a function with the form $f^*(o, s^*, \text{currentNodeID})$.

To synthesize CRDT logic that uses node IDs, we add a production rule so that the state transition can read from portions of the state that are owned by the current node, which are the values of maps keyed by a node ID. Similarly, we add a rule that allows the state transition to update portions of the state that the current node owns, by allowing insertions keyed by the current node ID. Finally, we introduce rules to the query grammar for performing reductions over the values of maps keyed by node IDs, which makes it possible to combine the state of each node into a global response to queries. We detail these additional grammar elements in Figure 6.9.

for the state transition:
$\forall V$

$\langle V \rangle$ ::= $\langle$*Map(NodeID,V)*$\rangle$[currentNodeID, default=$\langle V \rangle$]

$\langle$*Map(NodeID,V)*$\rangle$ ::= $\langle$*Map(NodeID,V)*$\rangle \sqcup$ {currentNodeID: $\langle V \rangle$}

for queries:

$\langle$*Int*$\rangle$ ::= reduce(values($\langle$*Map(NodeID,Int)*$\rangle$), $\lambda a.\lambda b.a + b$, 0)

$\langle$*Bool*$\rangle$ ::= reduce(values($\langle$*Map(NodeID,Bool)*$\rangle$), $\lambda a.\lambda b.a \vee b$, $false$)
| reduce(values($\langle$*Map(NodeID,Bool)*$\rangle$), $\lambda a.\lambda b.a \wedge b$, $true$)

Figure 6.9: The additional production rules required to support non-idempotent operations.

Although these changes to the construction of the state transition may allow it to be non-idempotent (and potentially non-commutative), the synthesized CRDT remains convergent because the only requirement for state-based CRDTs is that the merge function agrees with the state transition. Because our state transition still performs a lattice join with the previous state at the top level, and the non-idempotence/commutativity is restricted to portions of state, each owned by an individual node in the cluster, the merge function remains correct since a node can never receive new information about the portions of state it owns through gossip from other nodes.

Pruning Grammars with Specialized Types

While shallow instantiations of these grammars are sufficient to synthesize simple CRDTs, such as grow-only sets, they quickly grow to infeasible sizes when the state structure involves a larger number of nested data structures. Much of the grammar expansion comes from a conflation of types that can have distinct semantic meanings, resulting in production rules like arithmetic and comparison operations being unnecessarily introduced.

```
set* init_state() { return set_create(); }

set* st(set* s, int add, int v) { ... }

int query(set* state, int v) { ... }
```

```
stateTypeHint = Set(OpaqueInt())
opArgTypeHint = [EnumInt(), OpaqueInt()]
queryArgTypeHint = [OpaqueInt()]
queryRetTypeHint = EnumInt()
```

Figure 6.10: An example of how a sequential data type is annotated with specialized types.

To resolve this, we introduce *specialized* integer types, which represent distinct interpretations of integer values. In Katara, we have `OpaqueInt`, which represents an abstract value that

does not support arithmetic, `ClockInt`, which represents a positive timestamp that only supports comparison operations, and `EnumInt`, which represents values that only support equality. Users can then annotate the functions in their sequential data types, as shown in Figure 6.10, to mark types in the state and operation/query functions that conform to these specialized alternatives. When timestamps are enabled by the user to define richer operation orderings, we automatically add the necessary `ClockInt` annotations for those values.

By using distinct types for integer inputs, we can avoid searching expressions that, for example, compare timestamps to opaque values. We define grammar rules for these types in Figure 6.11. These types are also added to the state structure grammar, but for brevity we omit the changes here.

$$\begin{aligned}
&\langle bool \rangle ::= \langle opaque \rangle > \langle opaque \rangle \mid \langle opaque \rangle \geq \langle opaque \rangle \mid \langle opaque \rangle = \langle opaque \rangle \\
&\quad \mid \quad \langle clock \rangle > \langle clock \rangle \mid \langle clock \rangle \geq \langle clock \rangle \mid \langle clock \rangle = \langle clock \rangle \\
&\quad \mid \quad \langle enum \rangle = \langle enum \rangle \\
&\\
&\langle clock \rangle ::= 0 \\
&\\
&\langle enum \rangle ::= 0 \mid 1 \mid \text{constants in the sequential source} \\
&\\
&\\
&\forall T, U \\
&\\
&\langle U \rangle ::= \text{reduce}(\text{values}(\langle Map(T,U) \rangle), \lambda a.\lambda b.a \sqcup b, \bot)
\end{aligned}$$

Figure 6.11: The production rules for specialized integer types and semilattice reductions.

With the grammars defined for all three functions, we can apply a syntax-guided synthesis algorithm to explore the space of CRDT implementations and use the bounded operation log verification conditions to automatically verify candidates using an SMT solver. The bounds used in this phase start at very small values but are incrementally increased based on feedback from later phases of the synthesis algorithm, which we discuss in further detail in Section 6.5.3.

Invariant Synthesis for Unbounded Verification

After the first synthesis phase produces a CRDT design that passes bounded-log verification, we must synthesize additional invariants to check the CRDT against the unbounded verification conditions. We must re-synthesize the equivalence function with the CRDT state invariant included, since the unbounded conditions depend on the invariant to exclude unreachable CRDT states. In addition, we synthesize the *orderWithState** function so that the unbounded conditions can reason about operation orderings.

The CRDT state invariant only has access to the CRDT state, which helps reduce the size of the grammar generated. We seed the invariant with an explicit condition that checks if the state is valid according to the relevant semilattice definitions. Each semilattice in our state grammar

comes with logic for checking validity, such as that the integer values for clocks are at least zero. By automatically including these checks, we further reduce the burden on the synthesizer to discover properties needed for the inductive proof. The rest of the invariant is synthesized using the same type-based grammar as the other functions. Note that we do not need to synthesize the relation component of equivalence, since that was already synthesized in the bounded-log phase.

Synthesizing *orderWithState** is a bit more complex. Since the role of this function is to determine whether an operation is in-order while only having access to the CRDT state, this function often needs to combine information from large portions of the state rather than just manipulating data associated with specific keys. For example, when synthesizing a CRDT that uses clocks to order operations, *orderWithState** will need to check that the timestamp of the given operation is greater than all existing timestamps in the state. But it is challenging for syntax-guided synthesis engines to reason about arbitrary reductions, so we must reduce the complexity of the grammar.

We tackle this by noting that reductions (such as collecting the highest timestamp) use the semilattice join of the type being accumulated. This has intuitive backing as well, since we can check if a single value is at least as high in the semilattice order as several others through a single comparison against the semilattice join over those values. Based on this observation, we add a rule in Figure 6.11 to compute reductions using the lattice join for all relevant lattice values in the state. Note that we support reductions over map values, which are of a type in <*latticeType*>, but not sets because their elements may not be semilattices.

With these additional production rules, we can synthesize *invariant** and *orderWithState** for the candidate CRDT. With the invariant grammars configured and the existing *st** and *query** functions from the previous phase, we return to the synthesis engine with the unbounded verification conditions. At this point, we are verifying all scenarios the CRDT is expected to correctly handle, so if we successfully synthesize the invariants we have a provably correct CRDT design!

6.5.3 End-to-End Synthesis Algorithm

Now that we have the search space for state structures and runtime logic defined, we can synthesize the entire CRDT from scratch by simultaneously exploring both spaces. In our end-to-end algorithm, we apply multiple logic synthesis phases and verification modes to create provably correct CRDTs while also pruning the program space early in the synthesis algorithm.

At the top level of the algorithm, we iterate over candidate state structures generated from the grammar of semilattice compositions, bounded to the same *depth* as the runtime logic. For each of these, we then generate the appropriate runtime logic grammars and perform synthesis with the bounded-log verification conditions (with an initial $logBound = 2$). If we fail to synthesize, we can eliminate the candidate state structure from consideration, since there is no synthesizable logic even when the verification is relaxed.

If we successfully synthesize, we can move on to synthesizing the additional invariants for unbounded verification. We combine the synthesized code from the previous phase with the grammars for *invariant** and *orderWithState**, and call out to the synthesis engine again. If we fail to synthesize at this point, it means that either the bounded-log phase returned a buggy implementation or the grammar for invariants was too small. To address this, we return to the

```
function search(ref, opOrder)
    for depth ← (2..∞) do
        for s ← semilatticeCompositions(depth) do
            logBound ← 2
            p2Depth ← depth
            loop
                p1Synth ← synthBoundedLog(ref, s, opOrder, depth, logBound)
                if p1Synth = unsat then
                    break
                p2Synth ← synthUnbounded(ref, s, opOrder, p2Depth, p1Synth)
                if p2Synth = unsat then
                    logBound ← logBound + 1
                    p2Depth ← p2Depth + 1
                else
                    return p2Synth
```

Algorithm 1: The end-to-end algorithm for synthesizing a CRDT from scratch.

bounded-log phase and increment the operation log bound for verification and the grammar depth (*p2Depth*) for invariants. If we successfully synthesize the invariants, we have a provably correct CRDT that we can return to the user. We summarize this process in Algorithm 1.

6.6 Implementation

We implement Katara using an extended version of the framework in Casper [5], which allows us to automatically extract sequential data types written in C and C++ by first compiling them to LLVM and analyzing the IR to generate the equivalent SMT logic. Our implementation also includes wrappers around the Rosette synthesis engine [155] and CVC5 solver [21], which we use to perform synthesis and verification.

6.6.1 Supported Language Features

To synthesize a CRDT, Katara must first extract the semantics of the sequential data type provided by the user. To ensure that the logic implemented by the user can be accurately translated into the SMT logic used for verification, we define a space of programs that can be safely handled. Katara can handle basic LLVM operations, branches, integer/Boolean primitives, and list/set/map types.

Our analysis can accurately handle primitive types such as integers and Booleans, along with the corresponding arithmetic and logical operations on them. In addition, we provide a set of APIs for lists, sets, and maps that users can build on in their sequential data type. Our analysis automatically recognizes uses of these specialized APIs and lowers them to the corresponding

SMT theory. Our framework offers a modular approach to defining the semantics of these types, so it is straightforward to add support for richer data structures such as stacks.

In addition to analyzing the types and operations on them, our framework can extract branches found in the LLVM IR to conditionals in the generated SMT logic. Katara generates separate specifications of each basic block found in a function, and links them together to define the function as a whole. This approach allows us to handle nested conditionals and early returns without needing additional logic for these cases. In addition to branches, our analysis also handles user-defined functions by inlining them into the top-level function that is lifted.

6.6.2 Bounded Data Structure Verification

When synthesizing with Rosette, we face a limitation that the size of the symbolic state must be a constant, which means that we cannot define verification conditions that operate over unbounded data structures such as lists or sets. To address this limitation, we bound the size of these data structures to a fixed value while performing synthesis. After Rosette returns us a successfully synthesized CRDT, we then pass the result to CVC5, which can perform verification with unbounded data structures when a theory is defined for their behavior.

CVC5 natively supports a theory of sets, and we provide our own set of axioms that define tuples. When maps—which have not yet been modeled in CVC5—are involved in the synthesized CRDT, we fall back to using Rosette for verification with a large bound for the data structures. We hope to improve this in future work by providing a set of complete axioms that enable the solver to reason about unbounded map instances. In the meantime, the bounds we use for the fallback are sufficiently large to consistently produce correct synthesized results.

6.6.3 Parallel State Structure Exploration

Both Rosette, which uses Z3 under the hood, and CVC5 are single-threaded. If we were to naively implement the end-to-end algorithm, we would underutilize the multi-core capabilities of modern systems. But because we control the search of CRDT state structures, and the logic synthesis for each structure candidate is independent of the others, we can drastically speed up the synthesis algorithm by parallelizing across state structures.

Katara allows users to configure the number of state structures to synthesize logic for in parallel. Based on this parameter, we then instantiate a thread pool and spawn the logic synthesis algorithm for candidate structures on free threads. Our implementation simply returns the first successfully synthesized CRDT from any thread. By exploring more state structures and avoiding situations where synthesis is blocked on a state candidate that is particularly difficult to synthesize logic for, this technique drastically improves the end-to-end synthesis performance.

6.7 Evaluation

In our evaluation, we explore the capability of Katara to *correctly* and *efficiently* synthesize CRDTs for a variety of sequential data type and operation ordering specifications. We focus on answering the following research questions:

- **RQ1**: Can Katara produce practical CRDTs based on specifications sourced from literature on coordination-avoidance?
- **RQ2**: What is the effect of pruning structures with bounded-log verification on the overall synthesis performance?
- **RQ3**: Is the grammar of lattice composition sufficiently rich to produce CRDT designs that differ from the canonical implementations in the literature?

6.7.1 RQ1: Synthesizing Practical CRDTs

We begin by evaluating the ability of our synthesis algorithm to produce correct CRDTs from scratch for a variety of user-provided specifications. We sourced several sequential data types and operation orderings, summarized in Table 6.1, from existing literature on human-designed CRDTs [130] and coordination-avoidance [47]. For each of the benchmarks, we created a minimal implementation of the sequential type in C based on the specifications provided by the source and encoded the operation ordering using the IR provided by the synthesis system. All benchmarks were conducted on a AMD Ryzen 9 3900X processor with 12C/24T and 48 GB of memory, with our implementation configured to use up to 12 threads. We use LLVM 11 to compile and analyze the sequential types, as well as the latest versions of Rosette (4.1) and CVC5 (1.0.2).

Table 6.1: The set of CRDT specifications used to evaluate our synthesis algorithm.

Benchmark	Source	Specification Size	Timestamps	Non-Idempotent
Grow-Only Counter	Shapiro	21 LoC		✓
General Counter	Shapiro	20 LoC		✓
Enable-Wins Flag	De Porre	21 LoC	✓	
Disable-Wins Flag	De Porre	21 LoC	✓	
Last-Writer-Wins Register	Shapiro	16 LoC	✓	
Grow-Only Set	Shapiro	24 LoC		
Two-Phase Set	Shapiro	24 LoC		
Add-Wins Set	De Porre	24 LoC	✓	
Remove-Wins Set	De Porre	24 LoC	✓	

Our overall approach is designed to require minimal user intervention to produce a practical CRDT. As a proxy for this goal, we measured the amount of code required for a user to specify

both the sequential data type and the operation ordering that specifies the synthesized CRDT. For all our benchmarks, both of these components can be declared within 25 lines of C (for the data type) and Python (for the operation ordering). The Boolean flags to enable timestamps and non-idempotence are also provided to the system through a Python API. Most of the lines of specification code are for the sequential data type, which a developer using Katara will likely already have. The operation orderings, which are specific to Katara, could all be defined in 4 LoC or less of integer comparisons.

Table 6.2: The performance of synthesizing CRDTs for the benchmark specifications with Katara.

Benchmark	Synthesis Time (default)	Synthesis Time (no pruning)	Synthesized State Type
Grow-Only Counter	1m 46s ± 0.7s	52s ± 0.3s	`Map<NodeID, MaxInt<Int>>`
General Counter	11m 3s ± 4s	13m 21s ± 4s	`FreeTuple<Map<NodeID, MaxInt<Int>>, Map<NodeID, MaxInt<Int>>>`
Enable-Wins Flag	2m 4s ± 1s	2m 53s ± 8s	`LexicalProduct<MaxInt<ClockInt>, OrBool>`
Disable-Wins Flag	1m 46s ± 3s	3m 4s ± 2s	`LexicalProduct<MaxInt<ClockInt>, OrBool>`
Last-Writer-Wins Register	26s ± 0.2s	18s ± 0.8s	`LexicalProduct<MaxInt<ClockInt>, MaxInt<Opaque>>`
Grow-Only Set	30s ± 0.2s	23s ± 0.1s	`Set<Opaque>`
Two-Phase Set	58s ± 0.6s	1m 5s ± 0.8s	`Map<Opaque, OrBool>`
Add-Wins Set	21m 17s ± 11s	1hr 58m ± 1m	`FreeTuple<Map<Opaque, MaxInt<ClockInt>>, Map<Opaque, MaxInt<ClockInt>>>`
Remove-Wins Set	18m 40s ± 14s	1hr 52m ± 1m	`FreeTuple<Map<Opaque, MaxInt<ClockInt>>, Map<Opaque, MaxInt<ClockInt>>>`

When run with our collection of benchmarks, our synthesis algorithm is able to successfully generate designs that conform to all of the specifications, and it identifies the inductive invariants necessary to prove correctness of each CRDT over unbounded executions. We list the average time required to synthesize each CRDT (along with standard deviations) and the state structure of the synthesized result in Table 6.2. Simpler CRDTs, such as the Grow-Only/Two-Phase Set and LWW-Register, can be synthesized by Katara in a matter of seconds. More complex CRDTs, especially those that use timestamps to order operations such as the Add/Remove-Wins Set, can take on the order of tens of minutes to synthesize. These performance measurements indicate that the composition of multiple synthesis phases allows for many types of CRDTs to be synthesized in a reasonable amount of time.

Our synthesis results also show the algorithm discovering a variety of CRDT design techniques without any baked-in knowledge of CRDTs, such as using timestamps to guard data and

crdt *AddWinsSet*
  initialState: $(\{\}, \{\})$

  operation $(s, add, value, time)$
    return $s \sqcup$ **if** $add = 1$ **then**
      $(\{value : time\}, \{\})$
    else
      $(\{\}, \{value : time\})$

  query $((s_1, s_2), v)$
    $t_1 = s_1[v, default = 0]$
    $t_2 = s_2[v, default = 0]$
    return $t_1 \geq t_2 \wedge t_1 > 0$

crdt *GeneralCounter*
  initialState: $(\{\}, \{\})$

  operation $((s_1, s_2), inc, nodeID)$
    $cur_1 = s_1[nodeID, default = 0]$
    $cur_2 = s_2[nodeID, default = 0]$
    return $(s_1, s_2) \sqcup$ **if** $inc = 1$ **then**
      $(\{nodeID : cur_1 + 1\}, \{\})$
    else
      $(\{\}, \{nodeID : cur_1 + 1\})$

  query $((s_1, s_2))$
    $r_1 = reduce(values(s_1), \lambda a.\lambda b.a + b, 0)$
    $r_2 = reduce(values(s_2), \lambda a.\lambda b.a + b, 0)$
    return $r_1 - r_2$

Figure 6.12: The synthesized Add-Wins Set and General Counter CRDTs.

storing the effects of conflicting operations in separate parts of the state. For example, Katara synthesizes the CRDT on the left in Figure 6.12 for the Add-Wins Set, which supports repeated insertions and removals by using timestamps to have Adds only shadow Removes when they are concurrent. Similarly, on the right side of Figure 6.12, Katara is able to discover how to use node IDs to handle non-idempotent operations in a counter CRDT, using multiple reductions in the query to take the difference of the accumulated increments and decrements.

Although our work does not focus on the runtime performance of the synthesized code, all of our CRDTs have comparable theoretical performance to human designs in existing literature. In the case of the Two-Phase Set benchmark, our synthesis algorithm comes up with a more efficient state encoding (which we discuss in Section 6.7.3) that simplifies the state to a single integer-to-Boolean map rather than the typical two integer sets. Overall, our synthesis algorithm is able to produce practical, provably correct CRDTs for the variety of specifications in our benchmarks.

6.7.2 RQ2: Search Space Pruning

A key contribution in our runtime synthesis algorithm is the use of two SMT encodings of the correctness conditions: one that can quickly verify CRDT candidates with checks for bounded executions, and one that can prove unbounded correctness but requires the synthesizer to identify additional invariants. In this section, we explore how this two-phase synthesis approach improves the performance of the overall algorithm.

First, we can compare the time that our end-to-end algorithm takes to find CRDTs for each of the benchmarks with and without the pruning optimization. In Table 6.2, we list the synthesis times without the bounded-log phase under the "no pruning" column. Other than lighter benchmarks which synthesize in around minute and have relatively simple CRDT state types, all the benchmarks synthesize faster with the two-phase algorithm. The largest performance improvements come for the CRDTs with the most complex state structures: the General Counter and Add/Remove-Wins Set. For Sets, we see up to a 5x speedup by using the two-phase approach.

For a more nuanced exploration of *why* we see these speedups, we collect the time it takes each synthesis algorithm to either correctly synthesize or prune out each candidate data structure it considers for the Add-Wins Set benchmark. We compare the two algorithms in Figure 6.13, where we plot a distribution of the percent of candidates (out of 86 total for both) that can be evaluated within a given amount of time. With pruning, all candidate structures can be processed in under 15 minutes, allowing the end-to-end algorithm to quickly reach the state candidate that yields a verified CRDT. Without pruning, many state candidates take up to 20 minutes to be evaluated and there is a long tail of candidates that take up to an hour each. When the CRDT must have a complex state to support the specified semantics, these stragglers have a significant toll on synthesis performance since they block exploration of the program space.

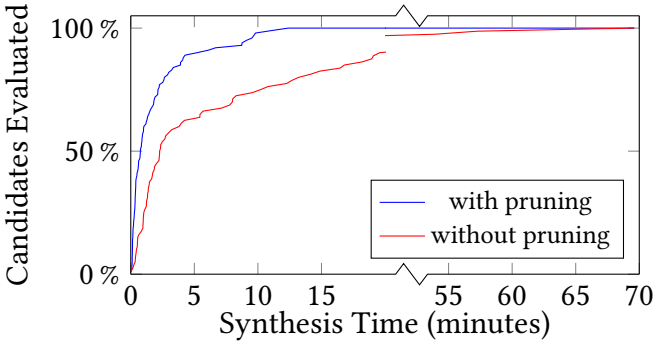


Figure 6.13: The time taken to evaluate candidates for Add-Wins Set.

6.7.3 RQ3: Alternative CRDT Synthesis

Finally, we evaluate the ability for our algorithm to produce *multiple* CRDT implementations for a *single* specification. Since our combination of a sequential type and operation ordering uniquely defines the user-observable behavior of a CRDT, any alternate designs will be functionally equivalent but may have different memory utilization and performance characteristics.

In our exploration of alternate CRDT designs, we focus on the Two-Phase Set, which has moderately complex semantics since its execution has multiple phases: allow inserts, then removes, but not inserts again. When we perform synthesis, the first CRDT that is generated is surprisingly **not** the 2P-Set from existing CRDT literature, but instead (to our knowledge) a novel design that uses a map to capture the phase of each element. If we continue searching, the algorithm eventually emits the classic 2P-Set, whose state structure is larger.

```
crdt TwoPhaseSet
    initialState: {}

    operation (m, add, value)
        return m ⊔ if add = 1
          then
              {value : false}
        else
              {value : true}

    query (m, v)
        return
            ¬m[v, default = true]
```

Figure 6.14: The novel CRDT synthesized for the Two-Phase Set benchmark.

We list the new design in Figure 6.14. There are several clever tricks that the synthesizer comes up with to match the specification while using a simpler state. First, the synthesizer realizes that the behavior of a Two-Phase Set "saturating" after removing an element matches how an `OrBool` saturates when it becomes true. Next, it discovers that by using a mapping from keys to these values, it can maintain a separate saturating value for each key.

But this still leaves a challenging situation for the initial state. Because we use the saturated *true* value to represent the element being removed, that means we have to set the value for a key to $false$ when it is inserted the first time. But since $\bot = false$ for an `OrBool`, that would leave us with no additional state. This is where the final trick is discovered by the synthesizer: to query the map with a default value of *true*. This effectively creates a third state for when the key is not even in the map. By automatically discovering this combination of CRDT design tricks, our synthesis algorithm is able to produce this novel encoding of a Two-Phase Set.

Our synthesis algorithm also produces alternate designs for many other benchmarks, such as using pairs of clocks instead of a `LexicalProduct` for the enable/disable-wins flag benchmarks. The presence of such alternatives paves the way for future work where we synthesize not only a *correct* CRDT, but a *performant* one according to a cost model that can compare CRDT candidates. Furthermore, the pool of alternative designs may be useful for incrementally re-synthesizing CRDTs as the sequential data type is updated, something we hope to explore in the future.

6.8 Related Work

6.8.1 Creating Replicated Objects from Sequential Specifications

There are a few lines of work that focus narrowly on the same problem we take aim at here: taking specifications of sequential datatypes and automatically creating equivalent replicated types. Where these projects differ from ours is primarily in our use of program synthesis: to our knowledge we are the first to utilize a search-based synthesis approach to generate state representations and runtime logic. Our other differences focus on how we chose to resolve conflicting operations, and our approach of searching semilattice compositions for the CRDT state.

Gallifrey [109], Indigo [20], and ECROs [47] focus on not just specifying replicated data types, but also in ensuring that applications which use them do not see inconsistent state—much as our work verifies correctness with respect to queries. Beyond our use of program synthesis, the main point of divergence is in their use of preconditions and postconditions as a verification tool to exactly match the behavior of synchronous objects. Other work also uses this pre/post-condition approach [48] or shares the goal of matching sequential behavior [135]. Our goal is not to exactly match sequential behavior; we let programmers tweak semantics with ordering constraints on conflicting operations. This allows us to lift all specified operations into CRDTs, rather than limiting ourselves to operations that already commute (as in Gallifrey) or resorting to explicit synchronization or deferred re-execution for conflicting events (as in Indigo and ECROs).

The MRDT line of work [78, 137] starts with a similar premise—creating replicated datatypes from annotated sequential specifications—but takes a radically different approach. MRDTs use a Git-inspired log of versioned data structures and are centered around annotations for abstraction and concretization. In contrast, our sorting-based annotations are simpler for non-experts to reason about, and our generated CRDTs require only a standard gossip protocol. Additionally, the merge function of MRDTs are generated using a rule-based approach, whereas our work takes the search-based synthesis approach.

6.8.2 Program Synthesis and Verified Lifting

The synthesis approach taken in this work is directly inspired by verified lifting [79], the approach at the heart of work such as Domino [134], Casper [5], and Dexter [6]. Our approach expands on this tradition primarily by our choice to synthesize entire data types, instead of just function implementations. This involves more complex conditions that check *behavioral equivalence* between the input code and the synthesized CRDT, rather than just checking equality of function outputs. Past work has explored the formal foundations of specifying CRDT correctness in terms of a sequential reference by layering constraints on how the effects of operations are applied [28]. Our introduction of pairwise ordering constraints, guided by this work, enables automated verification of CRDTs with lightweight annotations that are accessible.

Few previous systems have attempted to directly apply search-based program synthesis to the space of replication. Two that stand out are Hamsaz [69] and Hampa [99]. Hamsaz uses programmer-provided invariants to synthesize custom consistency protocols for the replication of shared data structures. While its analysis component is reminiscent of other work, such as Quelea, and the Indigo line [19, 20, 47, 133], its novel synthesis component is of particular interest to this work. Like our work, Hamsaz uses an SMT encoding of the programmer-specified semantics to search through potential replication strategies. Hampa [99], a similar work from the same research group, adds recency to the mix. However, both of these solutions are focused on identifying efficient coordination protocols, rather than *eliminating* coordination altogether. The CRDTs synthesized by our algorithm can be replicated without needing any coordination.

6.8.3 Verifying Replicated Data Types

Many systems focus on verifying CRDTs, checking both convergence properties and correctness with respect to a specification. Several of these require manual proof effort [57, 100, 167], which make them infeasible for program synthesis. Of particular note is recent work that explores automated verification of convergence properties via an SMT encoding [113]; we also use SMT to verify CRDT candidates. However, our work differs in *what* is being verified. Because our CRDTs are *convergent by construction*, we do not need to perform any convergence verification. Instead, we focus on the *user-observable behavior* of the CRDT, including checking the correctness of queries (which is beyond the scope of convergence).

Other lines of work focus on the general question of correct use of weak consistency [58, 158], which is a wider problem that is not specific to CRDTs. Furthermore, these lines of work focus on reasoning about how application invariants can be maintained when using weak consistency under the hood, rather than how different types of state can be replicated in an eventually consistent manner. Indeed, the literature around weak consistency is complementary, as they provide a path for developers to safely build applications on top of the CRDTs we synthesize.

There are certain CRDTs that challenge the current limitations of what we can specify in SMT for verification. These make for interesting future research directions. One example is the floating point comparison used in Logoot [159] which would require richer user-specified orderings and extensions to the query language. Another example is the Replicated Growable

Array (RGA), which requires tracking sequential data. Our query language does not currently support the types of iterative computations required to reason about sequences, but it has been shown that RGA can be represented as Datalog queries [84] over operations. We see this Datalog representation as a promising direction for enabling synthesis of such advanced CRDTs.

6.8.4 Making Replicated Objects Easier to Work With

Several papers attempt to make the process of programming against weak consistency tractable. Some do so by exposing weakly-consistent replicated objects with approachable semantics; indeed, the original CRDT work fits in this vein [130]. Other such approaches include the CloudTypes work from Microsoft [26] or work on allowing an application to safely mix consistency levels via MixT [108], Disciplined Inconsistency [67], CScript [49], or Red-Blue consistency [98]. Some work automatically chooses consistency levels for the programmer, driving the choice of mixed consistency via program invariants rather than explicit consistency annotations [77, 97, 133, 168, 169]. All work on mixing consistency shares the belief that programmers require stronger consistency for some operations; in contrast, we let users introduce semantic adjustments that eliminate the need for strong consistency.

Other work focuses on ensuring that application executions are convergent despite the inherent non-determinism of concurrency and weakly-consistent replication. These include a long line of work from Berkeley [9–11, 41, 42], and the Gallifrey, LASP, and LVars languages [90, 106, 109]. While we believe that whole-language approaches are valuable in this space, the CRDT literature typically does not analyze programs beyond the boundaries of the CRDT implementation.

6.9 Summary

The future of computing is distributed, so it is important to reduce the complexity of developing correct, efficient distributed programs. We believe that verified lifting can be a useful tool towards this goal, by automating much of the process of converting familiar sequential logic into scalable distributed code. In this chapter, we presented a first step in this direction with Katara, a system that automatically synthesizes CRDT designs from existing sequential data type implementations, requiring only simple annotations that are easy for developers to reason about.

We formalized the definition of *correctness* for CRDTs in terms of a sequential data type by introducing *operation orderings*, which allow users to define how the CRDT should handle conflicting non-commutative operations. To automate the verification process, we developed SMT encodings of this correctness definition that can be used to check CRDT candidates with a solver. Finally, we explored how compositions of semilattices can be naturally used as the state of a CRDT, and defined grammars for runtime logic and the invariants that enable unbounded verification of correctness. Our end-to-end algorithm efficiently synthesizes CRDTs for a variety of scenarios and produces novel alternatives to human-designed CRDTs. With Katara, we hope to further unlock the power of distributed systems by making it possible for any developer to automatically replicate their existing data types by synthesizing provably equivalent CRDTs.

Chapter 7

Time-Travelling Rewrites with Equality Saturation

7.1 Introduction

As applications scale to handle global real-time workloads, streaming dataflow systems have gained popularity as a way to enable low-latency computations on live data. Existing dataflow systems focus primarily on execution performance, utilizing incremental computation [14, 112] and operator fusion [120]. But these systems are intended for narrow domains such as analytical queries, and offer APIs with limited expressiveness.

The Hydro framework (Chapter 4) brings stream programming to general purpose distributed systems with a semantic model called stream-choreographic programming. This model gives developers precise control over placement and surfaces high-level correctness properties such as determinism without relying on expensive runtime protocols. Hydro exposes these semantics as a Rust library that lets developers apply stream transformations with arbitrary Rust closures.

Hydro uses a staged programming approach; the original Hydro program is *not* directly run on distributed machines. Instead, the Hydro program is locally executed to build an intermediate representation that captures the entire global program, which is then used to generate independent Rust sources for each location in the program. These sources are then compiled and deployed to the distributed machines. The global Hydro IR is a narrow waist in the Hydro stack, capturing all details of the distributed program in a machine-readable form. This is perfect point in the compilation process to *automatically* discover and apply optimizations.

In this chapter, we explore methods for optimizing Hydro programs through a rewrite-driven system. Hydro uses an expression-oriented IR, where each subtree represents a meaningful computation. We take advantage of this to define local rewrites that can be recursively applied across the IR. Rather than defining complex optimizations with even more complicated proofs of correctness, we instead focus on identifying a small set of rewrites necessary to derive effective optimizations through composition.

Although many Hydro operators involve arbitrary Rust closures, Hydro makes strong guaran-

tees on the determinism of each closure by limiting access to shared mutable state. Furthermore, many Hydro operators are reminiscent of relational algebra, which offers a rich set of optimization opportunities. These properties make it possible to rewrite Hydro IR without needing to reason about the semantics of the closures or involve verification techniques.

A unique feature of Hydro is its mechanism for defining iterative computation loops. Hydro programs typically run over unbounded asynchronous streams, but developers can also define local iterative loops using a model called "ticks". Ticks process finite batches of input streams and support operators to pass state across iterations. A particularly common use of ticks in Hydro is to repeatedly execute an algorithm over a growing prefix of an input. But this comes at a major performance cost, as the entire input must be reprocessed in each tick.

A common solution to this problem is *auto-incrementalization*, where the compiler automatically transforms the program to incrementally compute the updated results by relying on the results from the previous tick. This is a well-studied problem in the programming languages community [163] and database community [24]. But these solutions rely on handwritten incremental versions of operators such as joins. This approach requires significant effort to implement and verify the correctness of these incremental operators and are difficult to scale to complex IRs.

We instead approach incrementalization from first principles, using a set of fundamental rewrite rules that can be composed to derive the incremental version of a program. Our technique relies on two key insights. First, we extend Hydro IR with *time-travelling operators*, which allow a single expression tree to capture computations computed at different points in time. Then, we use e-graphs [115, 117] to explore the space of possible rewrites. E-graphs make it possible to capture *equivalence cycles*, which we use to capture inductive proofs of correctness over time.

Our rewrite rules are easy to verify, and can be applied in a general-purpose manner to any dataflow. In particular, our rewrite rules are sufficient for the optimizer to automatically discover classic patterns such as streaming joins. While there are some limitations of the e-graph approach, we believe that this model is a promising direction for further shifting the burden of correctness from axiomatic rewrite rules to the composition of local rewrites. In summary, we make the following contributions:

- We introduce time-travelling operators, which allow us to reason about the behavior of dataflow programs over time (Section 7.2)
- We develop a set of fundamental rewrite rules that reason about core properties of dataflow operators, such as associativity and determinism (Section 7.4)
- We show how equivalence cycles can be used to form inductive proofs and derive incremental computation (Section 7.4)
- We discuss the limitations of e-graphs, particularly in the context of DAGs with shared computations, and explore future work to address these limitations (Section 7.5)

7.2 Time-Travelling Operators

Before we dive into the optimizer, let us explore how developers use time-travelling operators in Hydro to build stateful streaming logic. Consider a simple chat application, where users can join a channel and receive all messages sent (including those before they joined). To keep things simple, we'll consider the case where there is only a single channel.

Hydro programs are written by transforming incoming streams with operators such as `map`, `filter`, or `join`. To allow more precise control over execution, Hydro allows developers to explicitly batch incoming streams into "ticks" using the `tick_batch` operator. Each batch is backed by a finite collection at runtime, and the Hydro runtime will continually process these batches in a loop. To yield elements from the loop, Hydro provides a `all_ticks` operator that will append each tick's outputs to an unbounded stream.

Our application has two streaming inputs, one for users requesting to join the channel (`add_member`), and one for messages being sent by users (`messages`). We batch both streams into a tick, and then take the cross product to broadcast messages. An initial attempt to implement this in Hydro using ticks[1] may look like the code in Figure 7.1.

```
let add_member = ...;
let messages = ...;
let tick = process.tick();
let broadcast = unsafe { // tick_batch is non-deterministic
  add_member.tick_batch(&tick)
    .cross_product(messages.tick_batch(&tick))
    .all_ticks()
};
```

Figure 7.1: Initial attempt at implementing a chat application in Hydro.

The logic of this program seems fairly reasonable, but this program will actually behave incorrectly! To understand why, we need to dive into the non-determinism of batching. At the beginning of each tick, Hydro collects any available network packets for each input channel into a batch. The streaming logic is then executed on these batches, and once the outputs are resolved we flush them to the output. Critically, all dataflow operators are *stateless* by default, so the state inside the cross-product is reset at the end of each tick.

In our example, this means that our program will only broadcast messages to users that joined *in the same tick*. This "catch" is by design—Hydro guides users to be mindful of the effects of non-deterministic batching on their programs (hence the use of `unsafe`). Indeed, there is nothing in our code that corresponds to showing *previous* messages to *newly* joined users. It is a *semantic* decision for the developer to decide whether to show previous messages or not, and Hydro avoids locking them into one or the other.

[1] Hydro also offers a `cross_product` operator that directly consumes unbounded streams, but it uses a handwritten incremental implementation so we will not use it here.

To fix this, we must use time-travelling operators in Hydro, which allow us to carry state across ticks in a principled manner. Hydro offers a *stateful* operator, `persist`, which consumes elements from some upstream source and emits the *entire history* of values it has received up to and including the current tick. With this operator, it is easy to get a program with semantics closer to our overall goal in Figure 7.2.

```
let broadcast = unsafe {
  add_member.tick_batch(&tick)
    .persist()
    .cross_product(messages.tick_batch(&tick).persist())
    .all_ticks()
};
```

Figure 7.2: Using `persist` to show previous messages to newly joined users.

What makes this *time-travelling* is the mathematical interpretation of `persist` under the property that operators in Hydro are deterministic. The traditional way to think about persist is in terms of its state, which will accumulate all elements in its history and re-emit them on each tick. Then, the output of the operator is a function of its input and state in the current iteration.

An alternative, more declarative way, is to assume that we have access to all inputs across all time steps. This allows us to define the semantics of this operator as a function of historical data, rather than reasoning about internal state. Formally, if at each iteration i the input to persist is a collection value c_i, then the output is $\sum_0^i c_i$ (where we use the concatenation operator from Flo, Chapter 2). This mathematical interpretation is key to defining our rewrite rules, which will reason about collections at different points in time with similar operators.

```
let broadcast = unsafe {
  add_member.tick_batch(&tick)
    .persist()
    .cross_product(messages.tick_batch(&tick).persist())
    .delta()
    .all_ticks()
};
```

Figure 7.3: Using `delta` to only show new messages to newly joined users.

Returning to our working example, there is still one last issue. Because `persist` replays the entire history of messages in every tick, clients will be sent repeated notifications for the same message. We can fix this by using the `delta` operation. This dataflow element consumes all values from an upstream source, but only emits the *new* values that did not appear in the previous tick. So our final, complete program looks like the snippet in Figure 7.3.

The only state for the `delta` operator is the input on the previous tick, but it is still useful to define its semantics in a time-travelling manner. Because Flo does not formally define an inverse for concatenation, this operator instead outputs a collection δ_i such that $c_{i-1} \pluscirc \delta_i = c_i$ (where c_i is the ith input to the operator). This operator is only valid for collection types where there is only one such δ_i (such as ordered sequences) to guarantee determinism. Comparing this to the semantics for `persist`, we see that `delta` is simply its inverse. We will use this property to identify incremental computations in our rewrite rules; our goal is to eliminate instances of `delta` that indicate redundant computations.

That's all! We now have a precise implementation of our specified program semantics. But this is not particularly efficient. In a naive execution of this dataflow, we will take the cross product with all messages in the history of the channel, only to later perform a delta that retains only the new messages and replays for newly joined members. Our goal is to preserve this clear model for computation while optimizing away the inefficiencies of naive state accumulation.

7.3 Fundamental Property Rewrites

To tackle the issue of inefficient stateful operators in dataflow, we turn to e-graphs. Our goal is to identify rewrite rules that optimize subgraphs while preserving *observational equivalence*, from the perspective of both users and downstream operators. For Hydro, we can rely on the formal semantics for collection values, which dictate when collections are considered equivalent and are used throughout the proofs of determinism in Flo.

Rather than creating specific rules for operations like cross-products, which have complex proofs and limited applicability, our goal is to define small, easy to prove rewrite rules that can be *composed* into complex optimizations. In our optimizer, each rewrite rule corresponds to a low-level property that would be used in a manual proof.

Before we can define our rewrite rules, we need to encode Hydro IR as expressions that can be processed by the e-graph engine. Because Hydro IR already uses a tree structure, we can directly map each Hydro IR node to an e-node (we omit Rust UDFs in our examples for brevity). We pass through `TeeRef` nodes, which are used to represent consumers of shared computation, as-is. This means that rewrites cannot cross a tee boundary; we will discuss this in more detail in Section 7.5. With our encoding, the Hydro IR in our motivating example can be expressed as an expression with LISP-like syntax in Figure 7.4.

```
(all_ticks (delta
  (cross_product
    (persist (tick_batch add_member))
    (persist (tick_batch messages)))))
```

Figure 7.4: Hydro IR for our motivating example translated to an e-graph expression.

After rewriting, the optimized expression can be translated back to Hydro IR by walking the expression tree and creating an IR node for each e-node. The alignment between the Hydro IR structure and e-graph expressions gives us significant power to define rewrite rules with proofs that fit into the existing Hydro semantic model.

7.3.1 Rewriting Persist

We will begin with equivalences revolving around the `persist` operator. In the previous section, we introduced the `delta` operators, which is an inverse for `persist`. We define a rewrite that matches instances of `persist` followed by a `delta` and eliminates both. This rewrite will always be the last step; we can eliminate a `delta` at the output if we can bubble a `persist` to the top.

In e-graphs, all equivalences are bidirectional. This means that we can also rewrite any subexpression to wrap it in a `persist` followed by a `delta`. In a traditional rewriting system, this would be problematic as it would create infinite possible expressions. But e-graphs can include cycles, which allow us to represent such equivalences without explicitly materializing them. We will use this property later to capture inductive proofs through rewrites.

Next, we develop rewrite rules for reasoning about the *semantics* of `persist` on its own, so that the optimizer understand its internal behavior. We discussed earlier that `persist` replays the messages it received in previous ticks, and also emits the values received from upstream in the current tick. A natural rewrite rule, then, is to make these semantics explicit so that our optimizer can reason about these two sources of values.

To do this, we introduce a new operator `old`, which behaves the same as `persist` *except* it does not emit the new values received from upstream. Formally, on tick i it emits $\sum_0^{i-1} c_i$ (where c_i is the input on the ith tick). We also introduce an operator `chain` which exposes the behavior of concatenation with ⧺ as an operator in our IR; it consumes two inputs and emits the concatenation of them. This operator allows us to split out the last step of a `persist`, by concatenating the output of `old` with the latest input.

With the rules so far, we can rewrite our working example to replace the `persist` operators with `old` and `chain`. We show the rewrite rules using syntax from the egg [161] engine in Figure 7.5, as well as the resulting expression.

```
(delta (persist ?a)) <=> ?a

(persist ?a) <=> (chain (old ?a) ?a)
```

```
(all_ticks (delta
  (cross_product
    (chain (old add_member) add_member)
    (chain (old messages) messages))))
```

Figure 7.5: Core rewrite rules for the `persist` operator, and our program after applying them.

7.3.2 Distributing Cross Products

A natural next step for our rewrite rules is to reason about cross products over chained input channels. The cross product operator is defined to output a stream with non-deterministic element ordering with multiset semantics. This means that we can distribute this operator over `chain`; doing so corresponds to splitting one of the nested loops into two halves.

Because `cross_product` is not commutative with respect to its input streams (swapping them changes the output tuples), we define separate rules for when the chain is in the first or second input. Applying both of these rules to our working example, we can shift both chain operators to the output of the cross product, which reveals how new and old data individually contribute to the final result. We show the rewrite rules and the transformed program in Figure 7.6.

```
(cross_product (chain ?a ?b) ?c) <=>
(chain (cross_product ?a ?c) (cross_product ?b ?c))

(cross_product ?a (chain ?b ?c)) <=>
(chain (cross_product ?a ?b) (cross_product ?a ?c))
```

```
1 (all_ticks (delta
2   (chain
3     (chain
4       (cross_product (old add_member) (old messages))
5       (cross_product (old add_member) messages))
6     (chain
7       (cross_product add_member (old messages))
8       (cross_product add_member messages)))))
```

Figure 7.6: Rewrite rules for distributing `cross_product` over `chain`.

In a general implementation beyond our example of cross products, we will also need to define rules for distributing other operators over `chain`. For example, `map` and `filter` can be distributed trivially because they are stateless operators and relational operators such as `join` are associative by definition. In the case of a stateful operator like `fold`, the equivalent distributive property is that a fold over chained input can be transformed into a nested pair of folds.

The rewrites for distributing cross products result in nested groups of chained expressions, but it is often helpful to regroup them with a different semantic objective. To do this, we define rewrites that capture the associativity of concatenation. Although Flo does not explicitly require ⧺ to be associative, this property is true for all collection types in Hydro.

Since *chain* simply applies the concatenation operator, it is associative as well, so we add a rewrite rule to capture this property. This rewrite rule allows us to isolate the cross product dealing with only old values, which we will need later on to make this computation incremental. We show the rewrite rule and the result of applying it in Figure 7.7.

```
(chain (chain ?a ?b) ?c) <=> (chain ?a (chain ?b ?c))
```

```
(all_ticks (delta
  (chain
    (cross_product (old add_member) (old messages))
    (chain
      (cross_product (old add_member) messages)
      (chain
        (cross_product add_member (old messages))
        (cross_product add_member messages))))))
```

Figure 7.7: Rewrite rule for associativity of `chain` and the resulting program.

These rewrites are critical to forming the *inductive step* of our incrementalization proof. They separate the computational steps that only operate on old data from the operators that consume data from the latest tick. With further rewrites, the former will match the inductive hypothesis, while the latter corresponds to the proof logic inside the inductive step.

7.3.3 Modeling Determinism

Our next insight is to capture the property of *determinism*, so that we can safely to re-use results from previous ticks. Flo requires all operators to be deterministic and there are no operators in Hydro that leak the current iteration number. Therefore, we know that all operators will produce the same results across ticks as long as the input collections are the same.

To formalize this, we start by introducing a *time-travelling* operator called `prev`. This operator simply yields the input from the previous ticks; the output is c_{i-1} in the ith tick. Determinism tells us that if an operator's inputs are all from the previous tick, we can shift the entire operator to run in the previous tick. We show the rewrites for `chain` and `cross_product` in Figure 7.8.

```
(chain (prev ?a) (prev ?b)) <=>
    (prev (chain ?a ?b))

(cross_product (prev ?a) (prev ?b)) <=>
    (prev (cross_product ?a ?b))
```

Figure 7.8: Determinism rules for `chain` and `cross_product`.

Because there are no other rewrites that involve `prev`, we cannot yet apply this operator to our working example. In the next section, we will introduce a time-unrolling rewrite that reveals the `prev` operator in our expression. This will allow us to apply the determinism rewrite and eliminate the `delta` operator.

7.4 Unrolling and Incrementalization

So far, all of our rewrites have reasoned about either an algebraic property of an operator (such as distribution) or its behavior across a single step of time (such as determinism). These are not sufficient to derive an incrementalization algorithm on their own. Each rewrite rule corresponds to a step in a proof of correctness, but we do not yet have any rules that correspond to inductive reasoning. We must introduce two new rewrite rules: one that unrolls time to create the inductive steps, and one that identifies a cycle in the e-graph to close the inductive proof.

7.4.1 Unrolling Ticks

Earlier in this chapter, we introduced the `old` operator, which is used to separate a `persist` node into the new data and everything up to the previous tick. But this `old` operator has a dead-end; we cannot continue walking back in time because there are no rules that split up the `old` operator.

Our insight is that the `prev` operator effectively updates the context of its children to execute as if the current tick is one less (`(let t (- t 1))`). Therefore, we see that `old` is equivalent to executing `persist` on the previous tick. This means that we can turn the `old` operator back into a `persist` by pushing it back in time by one tick. This effectively generalizes the rewrites in DBSP [24] to any collection in Flo (Chapter 2).

`(old ?a) <=> (prev (persist ?a))`

```
1 (persist a) => (chain
2   (prev (chain
3     (prev (chain ... a))
4     a
5   ))
6 a)
```

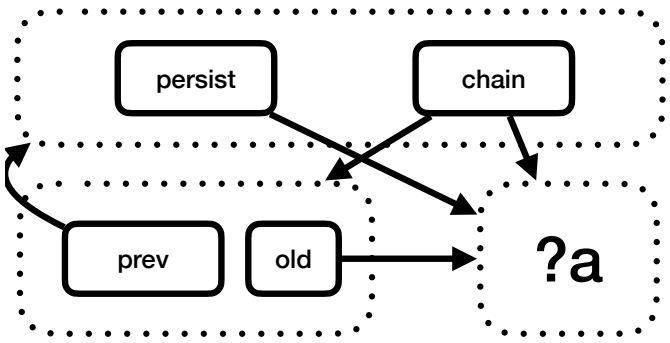


Figure 7.9: Unrolling `old` into `persist` and the resulting e-graph cycle.

This rule is simple and easy to prove correct, but it plays a role that is much more complex: allowing us to inductively reason about all ticks of a program. Because `old` can be converted into a `persist`, and `persist` can be rewritten to use `old` and `chain`, we can infinitely unroll the program by cycling between these two rewrites. Because we are using e-graphs, this does not result in infinite expansion of the e-graph. Instead, this rewriting is captured as a cycle in the e-graph, which allows us to reason about ticks of infinite depth without explicitly materializing them. We show the rewrite, an example of applying this unrolling, and a visualization of the e-graph cycle in Figure 7.9.

```
(all_ticks (delta (chain
  (prev (cross_product
    (persist add_member) (persist messages)))
  (chain
    (cross_product (old add_member) messages)
    (chain
      (cross_product add_member (old messages))
      (cross_product add_member messages))))))
```

Figure 7.10: The result of rewriting our working example with the unrolling rule.

E-graph cycles are common across several domains and important to efficiently capture bi-directional rewrites. But what is unique in our case is that the cycle involves use of the `prev` operator, which shifts the inner computation back one step in time. This means that a cycle in the e-graph corresponds to induction over *time*, since each cycle takes a single step back. As a result, we do not need to manually develop inductive proofs for our rewrites; the e-graph automatically discovers the inductive proof for us. If we apply this rewrite to our working example, we obtain the equivalence in Figure 7.10.

7.4.2 Induction with E-Graph Cycles

Although our e-graph now captures an inductive argument, we are not yet done. The final step is to close the inductive proof, transforming the cycle into an e-graph back into an acyclic expression that can be used to eliminate the `delta`. In Figure 7.10, our unrolling only went one way, turning a `persist` into an infinite chain across time steps. Our goal is to reverse this process, turning the `chain` into a `persist` that can cancel out the `delta`.

E-graphs remember their entire history of rewrites, and can be queried to determine if two expressions are in the same equivalence class. Looking at our working example, we notice that the rewritten expression inside the `delta` must be equivalent to the original expression: `(cross_product (persist add_member) (persist messages))`. An identical expression appears *within* our `chain`, but computed in the *previous* tick.

This is exactly the pattern we are looking for; repeatedly applying this equivalence would create an infinite unrolling of `chain` operators over ticks. If we reverse the unrolling rewrite from earlier, we can instead `persist` only the new elements contributed in each tick. This `persist` can then be used to cancel out the `delta` operator, which will yield an incremental computation.

Because egg does not have native syntax for pattern matching a cycle in the e-graph, we instead use a predicate to check if the root and the child of the `prev` operator are in the same equivalence class. We show this rewrite in Figure 7.11; note that it is a one directional rewrite because `?a` does not appear on the right-hand side.

```
(chain (prev ?a) ?b) => (persist ?b)
if eclass((chain (prev ?a) ?b)) = eclass(?a)
```

Figure 7.11: Reversing the unrolling rewrite to turn an e-graph cycle into a `persist` operation.

The proof of correctness for this rewrite relies on induction over ticks. In the base case, `(prev ?a)` is an empty stream, so `(chain (prev ?a) ?b) = ?b = (persist ?b)` because `persist` will have no state in the initial tick. In the inductive step, our hypothesis is that the rewritten expression is equivalent up to tick $t-1$. This is codified in our e-graph as the equivalence `(prev (persist ?b)) = (prev (chain (prev ?a) ?b)) = (prev ?a)`. We can wrap these expressions to get `(chain (prev ?a) ?b) = (chain (prev (persist ?b)) ?b)`. The latter can be further rewritten into `(chain (old ?b) ?b)` which is equivalent to `(persist ?b)`.

```
(all_ticks (chain
  (cross_product (old add_member) messages)
  (chain
    (cross_product add_member (old messages))
    (cross_product add_member messages))))
```

Figure 7.12: The final incremental result of our rewrite rules.

Applying this rewrite to our working example, we transform the `chain` into a `persist` operator, which we can then cancel out with the `delta` operator. We show the final result in Figure 7.12. This is an exciting result; we have identified an incremental way to compute the cross product by composing primitive rewrites rather than writing specialized rules. Indeed, we only have one rewrite that deals with incremental computation, the rest of the reasoning is left to the composition of fundamental properties.

7.4.3 Cost Model and Scalability

So far, we have discussed only the rewrite rules, but have not specified how we pick a single rewritten program from the expanded e-graph. We use a simple cost model that minimizes the number of nodes, with a higher weight for `delta` nodes because they indicate duplicated work. As a result, the e-graph engine will aim to find a rewritten program with as few `delta` nodes as possible; a fully incremental program will have none.

Because our rewrite rules are so general, we can easily derive incremental algorithms for other programs. For example, we can apply the same set of rewrite rules to a three-way cross product (between `add_member`, `messages`, and `platforms`), and discover an appropriate incremental algorithm with our simple cost model. This rewritten program is quite a bit more complex, and

would be much harder to reason about manually. But because the rewriting process also generates a proof, we can be confident in its correctness. We show the resulting program in Figure 7.13.

```
(all_ticks (chain
  (cross_product
    add_member
    (cross_product
      (old messages) (old platforms)))
  (cross_product
    (persist add_member)
    (chain
      (cross_product messages (old platforms))
      (cross_product (persist messages) platforms)))))
```

Figure 7.13: Incremental cross-product of three streams.

The rewrite rules which manipulate `delta`, `persist`, and `old` are general across collection types, and we have no rules specific to incrementally computing `cross_product`. If we define similar rules for distribution and determinism over the `join` operation, we can derive semi-naive Datalog evaluation from scratch! By using e-graphs to explore the search space of composed rewrites, we are able to easily support a large swath of programs with minimal effort needed to define rewrites and verify their correctness.

7.5 Diamonds are Hard to Crack

There is one limitation of our approach using e-graphs that is hard to ignore, yet leaves many exciting opportunities for future work in the wider e-graphs space. In Hydro, the data flowing out of a node can be used by several downstream paths through the `Stream::clone` API, which at runtime sends copies of each incoming value to each consuming operator. For example, we can use `clone` to compute users who should meet at the next Bay Area e-graph meetup in Figure 7.14.

In the cost model for an optimizer, it is critical to take into account that the computation before the `clone` is only performed once each tick, regardless of the number of consumers. But with our expression encoding, this is currently not possible because we can only extract computation *trees* rather than general DAGs. In particular, the dataflow structure that breaks our encoding is a **diamond**, a dataflow where a common computation is transformed in different ways that are eventually merged together (by interleaving their elements, joining on a key, etc). This can appear when using a common table expression (CTE) in SQL or a let-binding in functional languages.

In our current implementation, we simply flatten all diamonds by duplicating their shared subexpressions, and re-form diamonds after optimization by searching for identical expressions in the output. But in an ideal system, diamonds would be handled just like any other constructs in the optimizer. There are three key challenges in optimizing diamonds:

```
let add_member = ...;
let members = add_member.tick_batch(&tick).persist();
let meetup = members.clone()
  .map(with_school)
  .filter(berkeley)
  .cross_product(
      members.map(with_school).filter(stanford)
  )
  .all_ticks();
```

Figure 7.14: An example where a shared computation is used in a Hydro program.

1. When computing the cost function for a node, a common subexpression's cost should be counted only once even if is referenced multiple times.
2. We may want to shift logic from the common subexpression into its downstream consumers (inlining), to enable further optimizations.
3. The reverse of (2), after performing rewrites we may want to extract shared logic into a common subexpression to avoid duplicate computation.

7.5.1 Forming Diamonds with Zippers

In our early prototypes, we designed an explicit operator that captures the structure of a diamond. The `diamond` operator takes four parameters: the shared computation, two "edges" that describe the transformations being applied to data from the common source, and a merge node that defines how to combine the results from the two edges. This representation immediately solves challenge (1), since we precisely capture which computation is shared between multiple paths. For example, we can encode the earlier example with `diamond` on the left-side of Figure 7.14.

A key trick in this formulation is representing the "edges" of the diamond using a zipper [70] data structure. The nesting of operators is reversed between the halves of the zipper. In the first half, operator nodes have their inputs as children, but in the second half they have consumers as children. We use two special variables, `first` and `second`, to reference the values flowing out of both edges of the diamond.

What is powerful about zippers is that they make it possible to isolate either the first *or* last operator in a sequence by shifting the "cursor," the point where the two halves meet. In standard zipper implementations, this is implemented by popping an element from one half and pushing it to the other. For our encoding, we similarly remove the outermost operator from one half and wrap the other half with it. In our example, we can shift the cursors in both zippers to isolate one operator in each half, on the right-side of Figure 7.15.

After isolating the last operator of the edge in the second half of a zipper, we can apply another rewrite to inline the operator in the output, solving challenge (2). Thanks to the symmetry of the

```
(diamond
  (persist
    (tick_batch add_member))
  (zipper
    in
    (map with_school
      (filter berkeley out)))
  (zipper
    (filter stanford
      (map with_school in))
    out)
  (cross_product
    first second))
```

```
(diamond
  (persist
    (tick_batch add_member))
  (zipper
    (map with_school in)
    (filter berkeley out))
  (zipper
    (map with_school in)
    (filter stanford out))
  (cross_product
    first second))
```

Figure 7.15: Encoding a diamond as a zipper structure and shifting operators.

zipper, we can *also* solve challenge (3). If a single operator is isolated in the *first* half of each zipper, and is the same for both edges, it can be shared. Applying both rewrites, we can transform our example to the program in Figure 7.16.

```
(diamond
  (map with_school (persist (tick_batch add_member)))
  (zipper in out)
  (zipper in (filter stanford out))
  (cross_product (filter berkeley first) second))
```

Figure 7.16: The rewritten diamond with the shared operator extracted.

This encoding comes with a catch: there are dataflow graphs that *cannot* be encoded in terms of this diamond operator. Because our zippers only represent flat sequences, we cannot have any multi-input operators along an edge, unless those operators are part of a sub-diamond. In addition, manipulating zippers is very expensive, as we generate new e-classes for both halves whenever we perform a cursor shift, causing the e-graph to expand quite quickly.

We note that rewrite rules *consuming* zippers only care about the *isolated* first or last operator, so other intermediate states only exist for the shifting rule. In future work, we hope to explore ways to more efficiently represent zipper structures in an e-graph to take advantage of this domain-specific knowledge rather than naively using standard rewrite rules.

7.6 Related Work

7.6.1 Incremental Languages

Deriving incremental algorithms from a non-incremental program is a well-studied problem in both the programming languages and databases communities. Much of the foundational work in this area is focused on languages that capture incremental computation such as INC [163], which use a set of program transformations to compile to incremental algorithms. This approach avoids the cost of searching a large space of possible rewrites, but limits the core operators to those with known incremental implementations.

Similarly, work on functional reactive programming (FRP) identifies a set of reactive operators which can incrementally process changing inputs [33, 75, 76, 87, 122]. These operators are similar to the ones offered in Hydro, but restricted to only those that have manual incremental implementations with handwritten proofs. With our e-graphs approach, we automatically derive the incremental implementations, and therefore can support a wider range of operators.

The Stream Types [44] line of work focuses on a streaming language similar to Hydro. This work introduces formal requirements that limit the memory of each stream operator, forcing it to process inputs incrementally. While Hydro does not enforce these constraints, it cleanly separates stateful and stateless operators, which allows us to identify where state is being accumulated. In future work, formally separating these operators would make it possible to automatically derive an appropriate cost model for the optimizer.

7.6.2 Incremental View Maintenance

In the database community, there has been a long history of work on incremental view maintenance [34]. This work focuses on identifying efficient algorithms for keeping the results of a relational query up to date as the underlying data changes. Much work has focused on developing efficient systems that can perform IVM [56] using increasingly sophisticated semantics.

A particularly relevant piece of work is Naiad [105, 112], which supports incremental view maintenance for recursive queries using a linear model of time similar to ticks. Hydro itself borrows the concept of ticks from Dedalus [12], an extension to Datalog that lets developers explicitly reason about time. Our contribution is the simplicity of our rewrite rules; query engines rely on handwritten incremental operators with complex proofs of correctness, while we focus on foundational properties and leave incremental reasoning to the e-graph.

Recent work has revolved around the use of Z-Sets, which are multisets with integer cardinality, to represent the state of the database and propagate updates. Systems like DBSP [24] use this property to define query rewrites that result in an incremental algorithm. This work features many operators reminiscent of our time-travelling operators, such as differentiation (`delta`) and integration (`persist`). However, DBSP requires stream operators to satisfy properties such as bilinearity by assertion. We instead derive an equivalent result by composing distributivity and associativity. This simplifies our proof effort and allows us to support a wider range of operators.

7.7 Summary

The Hydro framework provides a powerful foundation for building distributed systems with a rich set of dataflow operators. However, the iterative ticks model makes it easy to write inefficient programs with redundant computations. Existing incremental systems rely on handwritten variants of built-in operators, which limits generality and can lead to correctness bugs. In this chapter, we explore a new direction that automatically derives incremental algorithms from first principles. Key to our approach is the use of e-graphs to capture equivalences and invariants across time, making it possible to compose primitive rewrites into complex optimizations. Our optimizer is able to derive classic incremental algorithms such as semi-naive joins without any human effort. This work demonstrates the feasibility of inductive reasoning within an e-graph engine, and paves the path for effective optimizers with simple proofs of correctness.

Chapter 8

Conclusion

With the rise of each new computing platform, we have seen a corresponding programming model that is *native* to that domain. Early computers and embedded devices led to languages with explicit memory management: C and C++. The rise of cloud computing led to high-level languages with garbage collection: Java and Go. The shift towards rich web applications led to declarative frameworks: React and Angular. The growth of machine learning led to tensor compilers: TensorFlow and PyTorch. Each of these models have unlocked opportunities for developers to build software that is more capable, reliable, and maintainable.

The past few years have seen a new movement focused on global interconnectedness, with the growth of applications that facilitate real-time connection across physical boundaries. This has led to a new generation of distributed systems that support low-latency applications with millions of users. These systems also face correctness concerns around security and privacy with increased urgency. What is the native programming model for this new generation of systems?

This dissertation proposes an answer: a programming model built around asynchronous streams that lets developers write correct and modular distributed systems with zero performance cost. While many streaming frameworks are already widely used, they apply expensive runtime protocols to guarantee that distribution does not affect the end-to-end correctness. Like the shift from runtime enforcement with garbage collection to static enforcement with borrow checking, our strategy is to capture distributed behavior in the type system and guarantee correctness properties with zero runtime cost.

The Hydro framework demonstrates that this programming model can be brought to life in a practical form. By leveraging the Rust ecosystem, we can provide a familiar programming model for developers while also creating new opportunities for modularity and optimization. Hydro's staged programming model enables us to expose distributed semantics without sacrificing performance, and its integration with existing libraries and tooling makes it easy to adopt. The benefits of our programming model go beyond correctness; they unveil new opportunities to share complex distributed logic as libraries of well-tested and performant protocols.

In the rest of this chapter, we walk through the broader insights from our research. We discuss the impact that Hydro has had beyond the scope of this dissertation, and we explore opportunities for future work.

8.1 Core Insights

Distributed systems have facets of correctness that are generally applicable. Much work in verifying and testing distributed systems focuses on definitions of correctness that are specific to a particular application domain. For example, many systems focus on *consistency* properties, which only apply to replicated state, or *safety* properties, which involve system-specific invariants. Our work identifies *determinism* as a key property that is relevant to all distributed systems. The non-determinism present in the network and concurrency show up in all distributed programs, and developers generally expect that their systems are immune to these perturbations. Unlike other correctness properties, determinism is compositional and requires no specification effort, which makes it ideal for a general-purpose programming model.

Stream programming need not be limited to ordered sequences. The traditional notion of a "stream" is a sequence of ordered elements, which is a natural fit for many applications. However, this is not the only way to think about streams. With Flo, we show that streams can be generalized to include sets, lattices, and even non-monotonic mutating state! This broader model focuses on the ability to asynchronously update collections and process partial inputs. Stream programming has largely been considered orthogonal to architectures such as actor programming, but our work shows that stateful actor-like logic can be meaningfully expressed as stream programs. This enables a new paradigm for modularity; entire distributed protocols can be captured in a single function, even when they span network boundaries.

Staged programming reduces the time to bring a new programming model to life. Historically, lines of research that introduce domain-specific programming models have required a new language and compiler, or suffer from performance penalties. This lengthens the timeline for adoption from a few years to decades; Rust was developed in 2006, but did not gain widespread adoptions until the 2020s. Staged programming allows us to expose a new programming model as a library in an existing language, while preserving fine-grained control over the compilation process. This makes it possible to bring a new programming model to life in a matter of months; the first downstream users of Hydro adopted it within the first year of its development. Staged programming has broad applicability beyond Hydro, and its integration into modern languages will make it easier to prototype and productionize new programming models.

Distributed systems are a rich domain for optimization. The field of distributed systems has a long history of optimization techniques, but they have largely focused on specific application domains such as query processing in SQL and analytical workloads in Flink. Hydro demonstrates that similar techniques can be applied to a much broader set of applications by using more advanced reasoning techniques. Hydro isolates arbitrary user logic to small Rust closures, which are amenable to verification and synthesis techniques. It also provides a rich compositional structure for distributed programs, which enable rewrite-oriented optimization. While our work in this area is still in early stages, Hydro demonstrates that optimization is feasible for general-purpose distributed systems.

8.2 Broader Impact: The Hydro Ecosystem

While this dissertation focuses on the core Hydro framework and formal research surrounding it, it is just one part of a much larger initiative led by the Hydro group that involves significant open-source contributions and research. This context has had a major impact on the design decisions behind Hydro, and includes the first practical applications of the Hydro framework.

The Hydro framework is available as an open-source library[1] with over 900 stars at the time of writing. This repository is structured as a monorepo, and contains not only the core Hydro framework, but also the DFIR runtime, Hydro Deploy, a standard library of protocols implemented in Hydro, and several research projects built on top of Hydro. Because all these systems are co-located, changes to Hydro are immediately checked against downstream projects, which ensures that the framework preserves stability and avoid regressions.

Hydro began with only DFIR (known at the time as Hydroflow [127]), with early research projects implemented directly in the low-level dataflow language. The first step towards the Hydro framework began with an implementation of Dedalus in DFIR[2] to support research exploring optimization techniques for distributed systems [38]. This was the first instance of using DFIR as a *target* for compilation, and resulted in an early version of the Hydro IR focused on the relational operators that appear in Datalog. Building this compiler also involved the creation of Rust Sitter[3], a new parsing library for Rust that was used to parse the Dedalus input. Rust Sitter has taken on a life of its own in the broader community with over 600 stars.

Shortly after the Dedalus implementation, we needed a deployment framework to evaluate the optimizer research project. At the time, the Hydro group used ad-hoc deployment scripts to launch DFIR programs, and stitching together multiple DFIR programs was a burdensome process. Thus, Hydro Deploy[4] was created to provide a simple, declarative way to launch several DFIR programs on a cluster. It was successfully used in the Compartmentalization project [38] as well as a (not yet published) novel gossip protocol. These projects stress tested Hydro Deploy, with deployments of hundreds of machines, and improved its robustness.

DFIR was never meant to be used by end-users, and lacked many compositional features such as modules. As research projects in the Hydro group grew in complexity, these limitations were starting to strain the development process. The Hydro framework began as a prototype to expose DFIR operators as a staged library in Rust so that we could use regular Rust functions to define reusable modules. Before we could develop this, we needed support for staging in Rust, and Stageleft[5] was born. From the beginning, Stageleft was designed to be a general-purpose staging library, with Hydro as a downstream user.

The earliest versions of Hydro yielded independent DFIR binaries, which would then be launched with a handwritten Hydro Deploy script. To reduce the deployment effort, Hydro took the leap to become a *distributed framework*, with locations that represent distributed machines

[1] https://github.com/hydro-project/hydro
[2] https://github.com/hydro-project/hydro/tree/dfir_datalog-v0.13.0/dfir_datalog
[3] https://github.com/hydro-project/rust-sitter
[4] https://github.com/hydro-project/hydro/tree/main/hydro_deploy
[5] https://github.com/hydro-project/stageleft

and network operators that move data between them. With these interfaces, Hydro could automatically derive the appropriate deployment configuration and control the end-to-end execution of the program. This was a major shift in the design of Hydro, and required significant changes to the compiler and runtime. Hydro is now a full-fledged distributed programming framework, with a rich set of operators and a strong type system.

The first major application of Hydro was a distributed implementation of Compartmentalized Paxos [160], a variant of the Paxos consensus algorithm focused on improving throughput (and developed by an alumnus of the same group that created Hydro). A key goal in this reimplementation, part of the Auto-Compartmentalization project [38], was to demonstrate that many components of this variant could be shared with the basic Paxos implementation. Achieving this resulted in the development of core programming patterns for Hydro, such as extracting quorums into a shared library and using Rust traits to define common interfaces over distributed protocols. This project also demonstrated the performance of Hydro, with throughput exceeding handwritten implementations in Scala and Dedalus (compiled to DFIR).

Hydro is an excellent example of co-design with downstream applications. Because other members of the group were actively using Hydro, we were able to identify pain points early and ensure our semantics could capture the challenges of real-world systems. For example, Paxos requires many conditional branches, which resulted in `Optional` being added as a stream type to Hydro. Similarly, feedback from downstream users led to iteration on interface naming in Hydro, such as the process / cluster distinction.

Beyond the core Hydro framework, our research on program optimization has also had a major open-source impact. The Katara project has been released as a standalone open-source library[6], which has been evaluated by users including researchers at Microsoft. Katara was the first project to leverage the Metalift framework[7] for verified lifting. As part of Katara, we contributed a total rewrite of the verification process in Metalift to support the new application domain.

In the e-graphs community, our work on time-travelling rewrites has inspired a new direction of research focused on applying e-graphs to database optimizers and distributed systems. These domains have unique challenges, such as diamonds, that are not well-supported by existing e-graph engines. Hydro is a compelling testbed for this research, since it has a rich set of potential rewrites and many opportunities to optimize real-world systems. For example, members of the Hydro group are exploring ways to add contextual reasoning to e-graphs [68], which would enable rewrites that change the stream type of a subexpression.

The wide range of projects in the Hydro ecosystem demonstrates the impact of a collaborative approach to programming language design. By building Hydro in tandem with downstream applications, we based our design decisions on feedback from real-world developers, rather than relying on estimations of what they might want. Hydro is a practical framework that has had a major open-source impact through the core framework and downstream projects. The Hydro ecosystem is a testament to the power of language/application co-design, and we hope that it will continue to grow and evolve in the future.

[6] https://github.com/hydro-project/katara
[7] https://github.com/metalift/metalift

8.3 Future Work

The work in this dissertation is only the beginning of a new programming paradigm for distributed systems. While Hydro has already been used in several research projects, there are many opportunities to extend its capabilities. To end this dissertation, we will discuss several promising areas of future work.

Beyond Determinism Our formal foundations in Flo and Gyatso focus on determinism as a key compositional property for distributed systems. However, there are many other relevant properties that are not yet captured in Hydro and can trigger unnecessary type errors. Each member of a cluster in Hydro is currently treated as an independent unit, and determinism only applies to *each machine*. This makes randomized partitioning unsafe, and does not guarantee any consistency properties across machines. Partitioning and replication have well-defined specifications, so Hydro should help developers reason about them. In future work, we believe that locations in Hydro can be extended to capture such invariants across machines in a cluster.

Algebraic Property Verification When introducing non-deterministic variants of streams that capture reordering and retries, we also had to develop variants of operators such as `fold` that require the closure to satisfy algebraic properties such as commutativity and idempotence. At the time, we left verifying these properties to the developer, but this is a major potential source of bugs. These closures are often quite small and involve few external dependencies, so it is feasible to use verification techniques to check that they satisfy the required properties. Automating this process would reduce the mental overhead for developers and likelihood of accidental bugs.

Deterministic Simulation Testing Developers using Hydro sometimes need to use the *unsafe* escape hatch to implement advanced protocol logic or work around limitations of the type system. In Rust, unsafe code is often verified using alternative testing techniques such as fuzzing. The equivalent tool for distributed systems is *deterministic simulation testing*, which locally simulates distributed execution with deterministic injection of network delays and concurrent execution. This allows developers to validate correctness properties beyond what is guaranteed by Hydro, including unsafe code. A particularly interesting area of future work is to fuzz test unsafe code for determinism, so that the code can be safely composed into a larger system.

Bridging Actors and Streams While this dissertation focuses on stream programming as a foundation for distributed systems, developers are already familiar with actor programming and there are many cases where they are a more natural fit. Existing research has shown how to compile sequential steps to dataflow programs; extending this to general actors would require compiling stateful logic. Bridging these paradigms would make it easier to build stateful services while preserving the end-to-end correctness guarantees of Hydro. It would also enable opportunities to incrementally migrate existing actor programs through compilation rather than rewriting them. This would be a major step towards bringing Hydro to a wider audience.

Bibliography

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

[2] Daniel J Abadi et al. "The design of the borealis stream processing engine." In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289.

[3] Martín Abadi and Boon Thau Loo. "Towards a Declarative Language and System for Secure Networking." In: *NetDB*. 2007.

[4] Martín Abadi et al. "TensorFlow: a system for large-scale machine learning." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. "Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1205–1220. DOI: 10.1145/3183713.3196891.

[6] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. "Automatically Translating Image Processing Libraries to Halide." In: *ACM Trans. Graph.* 38.6 (Nov. 2019). DOI: 10.1145/3355089.3356549.

[7] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803. DOI: 10.14778/2824032.2824076.

[8] Rajeev Alur et al. "Syntax-guided synthesis." In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[9] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. "Blazes: Coordination analysis for distributed programs." In: *2014 IEEE 30th International Conference on Data Engineering*. IEEE. 2014, pp. 52–63.

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

[11] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. "Dedalus: Datalog in time and space." In: *International Datalog 2.0 Workshop*. Springer. 2010, pp. 262–281.

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

[13] Amazon. *AWS Cloud Development Kit*. 2025. URL: https://aws.amazon.com/cdk.

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

[15] Joe Armstrong. "A history of Erlang." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 6˘1˘6–26. DOI: 10.1145/1238844.1238850.

[16] Michael Arntzenius and Neel Krishnaswami. "Seminaïve evaluation for a higher-order functional language." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371090.

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

[18] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams." In: *SIGMOD Rec.* 30.3 (Sept. 2001), pp. 109–120. DOI: 10.1145/603867.603884.

[19] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. "IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases." In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), pp. 404–418. DOI: 10.14778/3297753.3297760.

[20] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. "Putting Consistency Back into Eventual Consistency." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. DOI: 10.1145/2741948.2741972.

[21] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

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

[23] Eric A. Brewer. "Kubernetes and the path to cloud native." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: Association for Computing Machinery, 2015, p. 167. DOI: 10.1145/2806777.2809955.

[24] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. "DBSP: Automatic Incremental View Maintenance for Rich Query Languages." In: *Proc. VLDB Endow.* 16.7 (Mar. 2023), pp. 1601–1614. DOI: 10.14778/3587136.3587137.

[25] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. "Netherite: efficient execution of serverless workflows." In: *The VLDB Journal* 34.2 (Feb. 2025). DOI: 10.1007/s00778-024-00898-1.

[26] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. "Cloud Types for Eventual Consistency." In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 283–307.

[27] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. "Durable functions: semantics for stateful serverless." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485510.

[28] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. "Replicated data types: specification, verification, optimality." In: *ACM Sigplan Notices* 49.1 (2014), pp. 271–284.

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

[30] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache flink: Stream and batch processing in a single engine." In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

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

[32] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. "LUSTRE: a declarative language for real-time programming." In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188. DOI: 10.1145/41625.41641.

[33] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. "Fair reactive programming." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 361–372. DOI: 10.1145/2535838.2535881.

[34] Stefano Ceri and Jennifer Widom. "Deriving Production Rules for Incremental View Maintenance." In: *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, May 1999. eprint: https://direct.mit.edu/book/chapter-pdf/2305665/9780262287500_cbb.pdf. DOI: 10.7551/mitpress/4472.003.0035.

[35] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Mae Milano. "New directions in cloud programming." In: CIDR 2021. CIDR, 2021.

[36] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing database-backed applications with query synthesis." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 3–14.

[37] David C. Y. Chu, Chris Liu, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. "Bigger, not Badder: Safely Scaling BFT Protocols." In: *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '24. Athens, Greece: Association for Computing Machinery, 2024, pp. 30–36. DOI: 10.1145/3642976.3653033.

[38] David C. Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. "Optimizing Distributed Protocols with Query Rewrites." In: *Proc. ACM Manag. Data 2, N1 (SIGMOD)* (Feb. 2024). DOI: 10.1145/3639257.

[39] Edmund M. Clarke. "Model checking." In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Ramesh and G. Sivakumar. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56.

[40] The Tokio Contributors. *Tokio: A runtime for writing reliable asynchronous applications with Rust*. 2025. URL: https://github.com/tokio-rs/tokio.

[41] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M Hellerstein. "Edelweiss: Automatic storage reclamation for distributed programming." In: *Proceedings of the VLDB Endowment* 7.6 (2014), pp. 481–492.

[42] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. "Logic and Lattices for Distributed Programming." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. DOI: 10.1145/2391229.2391230.

[43] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. "TARDiS: A Branch-and-Merge Approach To Weak Consistency." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1615–1628. DOI: 10.1145/2882903.2882951.

[44] Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. "Stream Types." In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656434.

[45] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. "Adaptive Stream Processing using Dynamic Batch Sizing." In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: Association for Computing Machinery, 2014, pp. 1–13. DOI: 10.1145/2670979.2670995.

[46] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[47] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. "ECROs: Building Global Scale Systems from Sequential Code." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485484.

[48] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. "Putting Order in Strong Eventual Consistency." In: *Distributed Applications and Interoperable Systems*. Ed. by José Pereira and Laura Ricci. Cham: Springer International Publishing, 2019, pp. 36–56.

[49] Kevin De Porre, Florian Myter, Christophe Scholliers, and Elisa Gonzalez Boix. "CScript: A distributed programming language for building mixed-consistency applications." In: *Journal of Parallel and Distributed Computing* 144 (2020), pp. 109–123. DOI: https://doi.org/10.1016/j.jpdc.2020.05.010.

[50] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[51] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[52] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic Algorithms for Replicated Database Maintenance." In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '87. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1987, pp. 1–12. DOI: 10.1145/41840.41841.

[53] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. "P: safe asynchronous event-driven programming." In: *SIGPLAN Not.* 48.6 (June 2013), pp. 321–332. DOI: 10.1145/2499370.2462184.

[54] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow." In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[55] Roy Frostig, Matthew Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing." In: 2018. URL: https://mlsys.org/Conferences/doc/2018/146.pdf.

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

[57] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. "Verifying Strong Eventual Consistency in Distributed Systems." In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133933.

[58] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625.

[59] Eva Graversen, Fabrizio Montesi, and Marco Peressotti. *A Promising Future: Omission Failures in Choreographic Programming*. 2025. URL: https://arxiv.org/abs/1712.05465 arXiv: 1712.05465 [cs.PL].

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

[61] Hashicorp. *Terraform*. 2025. URL: https://developer.hashicorp.com/terraform.

[62] Pat Helland and David Campbell. *Building on Quicksand*. 2009. URL: https://arxiv.org/abs/0909.1788 arXiv: 0909.1788 [cs.DC].

[63] Joseph M Hellerstein. "The declarative imperative: experiences and conjectures in distributed logic." In: *ACM SIGMOD Record* 39.1 (2010), pp. 5–19.

[64] Joseph M Hellerstein and Peter Alvaro. "Keeping CALM: when distributed consistency is easy." In: *Communications of the ACM* 63.9 (2020), pp. 72–81.

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

[66] Todd Hoff. *How League Of Legends Scaled Chat To 70 Million Players - It Takes Lots Of Minions*. 2014. URL: http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.

[67] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. "Disciplined Inconsistency with Consistency Types." In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, pp. 279–293. DOI: 10.1145/2987550.2987559.

[68] Tyler Hou, Shadaj Laddad, and Joseph M. Hellerstein. "Towards Relational Contextual Equality Saturation." In: (2024).

[69] Farzin Houshmand and Mohsen Lesani. "Hamsaz: Replication Coordination Analysis and Synthesis." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290387.

[70] Gerard Huet. "Functional pearl: The zipper." In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[71] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems." In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, p. 11.

[72] Kasun Indrasiri and Danesh Kuruppu. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[73] Daniel Jackson. "Alloy: a language and tool for exploring software designs." In: *Commun. ACM* 62.9 (Aug. 2019), pp. 66–76. DOI: 10.1145/3338843.

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

[75] Alan Jeffrey. "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)*. CSL-LICS '14. Vienna, Austria: Association for Computing Machinery, 2014. DOI: 10.1145/2603088.2603106.

[76] Alan Jeffrey. "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*. PLPV '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 49–60. DOI: 10.1145/2103776.2103783.

[77] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. "Safe Replication through Bounded Concurrency Verification." In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276534.

[78] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. "Mergeable Replicated Data Types." In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360580.

[79] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. "Verified Lifting of Stencil Computations." In: *SIGPLAN Not.* 51.6 (June 2016), pp. 711–726. DOI: 10.1145/2980983.2908117.

[80] Shun Kashiwa, Gan Shen, Soroush Zare, and Lindsey Kuper. *Portable, Efficient, and Practical Library-Level Choreographic Programming*. 2023. URL: https://arxiv.org/abs/2311.11472 arXiv: 2311.11472 [cs.PL].

[81] Kyle Kingsbury. *Jepsen*. 2025. URL: https://github.com/jepsen-io/jepsen.

[82] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. "Stream fusion, to completeness." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 285–299. DOI: 10.1145/3009837.3009880.

[83] Martin Kleppmann. *Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode*. Tech. rep. University of Cambridge, Computer Laboratory, 2022.

[84] Martin Kleppmann. "Data structures as queries: Expressing CRDTs using Datalog." In: (2018). URL: https://martin.kleppmann.com/2018/02/26/dagstuhl-data-consistency.html.

[85] Martin Kleppmann and Alastair R. Beresford. "A Conflict-Free Replicated JSON Datatype." In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2733–2746. DOI: 10.1109/TPDS.2017.2697382.

[86] Rusty Klophaus. "Riak Core: Building Distributed Applications without Shared State." In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: Association for Computing Machinery, 2010. DOI: 10.1145/1900160.1900176.

[87] Neelakantan R. Krishnaswami. "Higher-order functional reactive programming without spacetime leaks." In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 221–232. DOI: 10.1145/2544174.2500588.

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

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

[90] Lindsey Kuper and Ryan R. Newton. "LVars: Lattice-Based Data Structures for Deterministic Parallelism." In: *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 71–84. DOI: 10.1145/2502323.2502326.

[91] Shadaj Laddad and Koushik Sen. "Fluid quotes: metaprogramming across abstraction boundaries with dependent types." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 98–110. DOI: 10.1145/3425898.3426953.

[92] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System." In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[93] Leslie Lamport. "The part-time parliament." In: *ACM Transactions on Computer Systems* 16.2 (May 1998), pp. 133–169. DOI: 10.1145/279227.279229.

[94] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. DOI: 10.1145/359545.359563.

[95] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. DOI: 10.1145/3098822.3098842.

[96] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75.

[97] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 281–292. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[98] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary." In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 265–278. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[99] Xiao Li, Farzin Houshmand, and Mohsen Lesani. "Hampa: Solver-Aided Recency-Aware Replication." In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 324–349.

[100] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell." In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428284.

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

[102] Konstantinos Mamouras. "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. 2020, pp. 394–427.

[103] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. "Data-trace types for distributed stream processing systems." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 670–685. DOI: 10.1145/3314221.3314580.

[104] Nicholas D. Matsakis and Felix S. Klock. "The rust language." In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. DOI: 10.1145/2663171.2663188.

[105] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow." In: *Proceedings of CIDR 2013*. Jan. 2013. URL: https://www.microsoft.com/en-us/research/publication/differential-dataflow/.

[106] Christopher Meiklejohn and Peter Van Roy. "Lasp: A Language for Distributed, Coordination-Free Programming." In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy: Association for Computing Machinery, 2015, pp. 184–195. DOI: 10.1145/2790449.2790525.

[107] Ruijie Meng, George Pirlea, Abhik Roychoudhury, and Ilya Sergey. "Greybox Fuzzing of Distributed Systems." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1615–1629. DOI: 10.1145/3576915.3623097.

[108] Mae Milano and Andrew C. Myers. "MixT: A Language for Mixing Consistency in Geodistributed Transactions." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 226–241. DOI: 10.1145/3192366.3192375.

[109] Matthew Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. "A Tour of Gallifrey, a Language for Geodistributed Programming." In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 136. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 11:1–11:19. DOI: 10.4230/LIPIcs.SNAPL.2019.11.

[110] Fabrizio Montesi. *Choreographic programming*. IT-Universitetet i København, 2014.

[111] Philipp Moritz et al. "Ray: A distributed framework for emerging {AI} applications." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[112] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455. DOI: 10.1145/2517349.2522738.

[113] Kartik Nagar and Suresh Jagannathan. *Automated Parameterized Verification of CRDTs*. 2019. DOI: 10.48550/ARXIV.1905.05684.

[114] David Navalho, Sérgio Duarte, Nuno Preguiça, and Marc Shapiro. "Incremental stream processing using computational conflict-free replicated data types." In: *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*. CloudDP '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 31–36. DOI: 10.1145/2460756.2460762.

[115] Charles Gregory Nelson. "Techniques for Program Verification." AAI8011683. PhD thesis. Stanford, CA, USA, 1980.

[116] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods." In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. DOI: 10.1145/2699417.

[117] Robert Nieuwenhuis and Albert Oliveras. "Proof-Producing Congruence Closure." In: *Proceedings of the 16th International Conference on Term Rewriting and Applications*. RTA'05. Nara, Japan: Springer-Verlag, 2005, pp. 453–468. DOI: 10.1007/978-3-540-32033-3_33.

[118] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Samza: stateful scalable stream processing at LinkedIn." In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645. DOI: 10.14778/3137765.3137770.

[119] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320.

[120] Shoumik Palkar et al. "Evaluating End-to-End Optimization for Data Analytics Applications in Weld." In: *Proc. VLDB Endow.* 11.9 (May 2018), pp. 1002–1015. DOI: 10.14778/3213880.3213890.

[121] Adam Paszke et al. "PyTorch: an imperative style, high-performance deep learning library." In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

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

[123] David J. Pearce. "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust." In: *ACM Trans. Program. Lang. Syst.* 43.1 (Apr. 2021). DOI: 10.1145/3443420.

[124] Conor Power, Paraschos Koutris, and Joseph M Hellerstein. *The Free Termination Property of Queries Over Time*. 2025. URL: https://arxiv.org/abs/2502.00222 arXiv: 2502.00222 [cs.DB].

[125] Raymond Roestenburg, Rob Williams, and Robertus Bakker. *Akka in action*. Simon and Schuster, 2016.

[126] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. GPCE '10. Eindhoven, The Netherlands: Association for Computing Machinery, 2010, pp. 127–136. DOI: 10.1145/1868294.1868314.

[127] Mingwei Samuel. "Hydroflow: A Model and Runtime for Distributed Systems Programming." MA thesis. EECS Department, University of California, Berkeley, Aug. 2021. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-201.html.

[128] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. "Building Efficient Query Engines in a High-Level Language." In: *ACM Trans. Database Syst.* 43.1 (Apr. 2018). DOI: 10.1145/3183653.

[129] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Ngo. "Optimizing Nested Recursive Queries." In: *Proc. ACM Manag. Data* 2.1 (Mar. 2024). DOI: 10.1145/3639271.

[130] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "A comprehensive study of convergent and commutative replicated data types." PhD thesis. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[131] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[132] Gan Shen, Shun Kashiwa, and Lindsey Kuper. "HasChor: Functional Choreographic Programming for All (Functional Pearl)." In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607849.

[133] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. "Declarative Programming over Eventually Consistent Data Stores." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 413–424. DOI: 10.1145/2737924.2737981.

[134] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet Transactions: High-Level Programming for Line-Rate Switches." In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 15–28. DOI: 10.1145/2934872.2934900.

[135] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. "Static Local Coordination Avoidance for Distributed Objects." In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 21–30. DOI: 10.1145/3358499.3361222.

[136] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. "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. 2010, pp. 507–528.

[137] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. "Certified Mergeable Replicated Data Types." In: *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022.

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

[139] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. "A practical unification of multi-stage programming and macros." In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 14–27. DOI: 10.1145/3278122.3278139.

[140] Nicolas Alexander Stucki. "Scalable Metaprogramming in Scala 3." en. PhD thesis. Lausanne: EPFL, 2023. DOI: 10.5075/epfl-thesis-8257.

[141] Walid Taha. "A Gentle Introduction to Multi-stage Programming." In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. DOI: 10.1007/978-3-540-25935-0_3.

[142] Walid Taha and Tim Sheard. "Multi-stage programming with explicit annotations." In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '97. Amsterdam, The Netherlands: Association for Computing Machinery, 1997, pp. 203–217. DOI: 10.1145/258993.259019.

[143] Walid Taha and Tim Sheard. "Multi-stage programming with explicit annotations." In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203–217. DOI: 10.1145/258994.259019.

[144] Walid Mohamed Taha. *Multistage programming: its theory and applications*. Oregon Graduate Institute of Science and Technology, 1999.

[145] The Antithesis Team. *Antithesis*. 2025. URL: https://antithesis.com.

[146] The Bincode Team. *Bincode: A binary encoder / decoder implementation in Rust*. 2025. URL: https://github.com/bincode-org/bincode.

[147] The Cargo Team. *Cargo: Rust's package manager*. 2025. URL: https://doc.rust-lang.org/cargo.

[148] The Rust Team. *Rust Iterators*. 2025. URL: https://doc.rust-lang.org/std/iter/trait.Iterator.html.

[149] The Rust Team. *Rust Reference: Constant evaluation*. 2025. URL: https://doc.rust-lang.org/reference/const_eval.html.

[150] The Unison Team. *Unison*. 2025. URL: https://www.unison-lang.org.

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

[152] David Tolnay. *Pre-RFC: Sandboxed, deterministic, reproducible, efficient Wasm compilation of proc macros*. 2025. URL: https://internals.rust-lang.org/t/pre-rfc-sandboxed-deterministic-reproducible-efficient-wasm-compilation-of-proc-macros/19359.

[153] David Tolnay. *Syn: Parser for Rust source code*. 2025. URL: https://github.com/dtolnay/syn.

[154] David Tolnay. *Trybuild: Test harness for ui tests of compiler diagnostics*. 2025. URL: https://github.com/dtolnay/trybuild.

[155] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 135–152. DOI: 10.1145/2509578.2509586.

[156] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. "Exploiting punctuation semantics in continuous data streams." In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 555–568.

[157] Werner Vogels. "Eventually Consistent." In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. DOI: 10.1145/1435417.1435432.

[158] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. "Replication-Aware Linearizability." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 980–993. DOI: 10.1145/3314221.3314617.

[159] Stephane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks." In: *2009 29th IEEE International Conference on Distributed Computing Systems*. 2009, pp. 404–412. DOI: 10.1109/ICDCS.2009.75.

[160] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. "Scaling replicated state machines with compartmentalization." In: *Proc. VLDB Endow.* 14.11 (July 2021), pp. 2203–2215. DOI: 10.14778/3476249.3476273.

[161] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "Egg: Fast and Extensible Equality Saturation." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304.

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

[163] Daniel M. Yellin and Robert E. Strom. "INC: a language for incremental computations." In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 211–236. DOI: 10.1145/103135.103137.

[164] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications." In: *Advanced research working conference on correct hardware design and verification methods*. Springer. 1999, pp. 54–66.

[165] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing." In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 2.

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

[167] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Formal Specification and Verification of CRDTs." In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 33–48.

[168] Xin Zhao and Philipp Haller. "Observable Atomic Consistency for CvRDTs." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 23–32. DOI: 10.1145/3281366.3281372.

[169] Xin Zhao and Philipp Haller. "Replicated data types that unify eventual consistency and observable atomic consistency." In: *Journal of Logical and Algebraic Methods in Programming* 114 (2020), p. 100561. DOI: https://doi.org/10.1016/j.jlamp.2020.100561.