SAFE PROGRAMMING OVER DISTRIBUTED STREAMS

Caleb Stanford

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Zachary G. Ives, Adani President's Distinguished Professor of Computer and Information Science

Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science

Vincent Liu, Assistant Professor of Computer and Information Science

Margus Veanes, Principal Researcher, Microsoft Research Redmond

SAFE PROGRAMMING OVER DISTRIBUTED STREAMS

Dedicated to Michael Zhao

1995–2018

ACKNOWLEDGMENTS

Thank you to my wife, Ariana, for joining forces with me to complete our PhDs together over the last six years. I've had your boundless support during the most exciting days, the shared relaxing evenings, and the hardest nights. I'll miss exploring Philadelphia with you; I'm looking forward to the next adventure.

I am grateful to my advisor, Rajeev, for his exceptional patience, and for teaching me most of what I know about research. Following our first paper submission, when I came to Rajeev with a crazy idea for a better way to define automata for streaming, he believed in me and let me take the project in that direction. When I fixated on technical details, Rajeev never lost sight of the bigger picture, and that has been invaluable to my work.

I've been fortunate to have many additional mentors during my PhD. Thanks to Zack Ives, for his expertise in pragmatism; Benjamin Pierce, for teaching me how to think about programming languages and writing; Vincent Liu, for teaching me how to think about systems; and Val Tannen, who probably best understands what my research is trying to do (and always made the process of doing it thoroughly enjoyable). Thank you to Stephanie Weirich and Steve Zdancewic for their advice on faculty interviews, for the board game night invites, and for taking the time to attend TGIF with students. I also want to thank Kostas Mamouras, Scott Weinstein, and Sampath Kannan. Thanks to my advisor at Amazon, Pauline Bolignano, for designing an internship which gave me close insight into what really matters to security engineers while still generating interesting research ideas; and to my advisors at Microsoft, Margus Veanes and Nikolaj Bjørner, for being excited to talk about derivatives with me despite the pandemic, and for rapidly onboarding me on the Z3 project.

My life as a PhD student at Penn was made possible by the PLClub community. Thanks to Leo Lampropoulos, Antal Spector-Zabusky, Jennifer Paykin, Robert Rand, and Christine Rizkallah for making me feel included when I was most in doubt; to Konstantinos Kallas and Filip Nikšić, who made research noticeably more enjoyable after they joined, through many whiteboard conversations, dinners, and drinks; and to several other friends, but especially Suguman Bansal, Kishor Jothimurugan, Yao Li, Solomon Maina, Omar Navarro Leija, Li-yao Xia, and Anton Xue. More broadly, I have appreciated interacting with numerous members of the PLClub over the years: Aalok, Antoine, Arthur, Calvin, Clara, Elizabeth, Gautam, Halley, Haoxian, Hengchu, Jiani, Kenny, Lawrence, Lei,

Lucas, Mukund, Nicolas, Nimit, Pardis, Pritam, Rado, Richard, Steve, Sulekha, Xujie, Yannick, Yishuai, and Ziyang. And best of luck to the next generation of PLClub stars (though I know they don't need it); these certainly include at least Harry Goldstein, Phillip Hilliard, Stephen Mell, Irene Yoon, Lef Ioannidis, Aaditya Naik, Joe Cutler, Yiyun Liu, Jessica Shi, Joey Velez-Ginorio, and Chris Watson.

Besides PLClub, I owe a huge thanks to all of my fellow CISDA volunteers – especially Luke Valenta, Daphne Ippolito, Oshin Agarwal, Paul He, and Alyssa Hwang – for their efforts to make the department a better place. Not to mention Cheryl Hickey, for single-handedly holding the department together. My collaborators Nofel Yaseen, Liangcheng Yu, and Ryan Beckett have been very patient with my ignorance about networks. I've enjoyed friendly conversations with Edo Roth, Zach Schutzman, Kelly Shiptoski, and Jiali Xing. And thanks to Yu Chen for generously meeting with me to talk about online graph algorithms. I'm sure I've forgotten someone, and I apologize! Outside of Penn, I am grateful for the many conferences I have been able to attend, and the wonderful attendees I have met over the years.

Although my PhD has made too many of our interactions virtual, I can't overestimate the importance of my family and close friends. Thank you to Kei Nishimura-Gasparian, Russell Shelp, and Frank Chung; my siblings, siblings-in-law, and niece/nephews (MKZJTPT, JSSLCM, H, THL, BI, and E); my grandparents and extended family; Pinto and Calcifer; and lastly, Alex Spentzos, for never running out of fun things to argue about, and for planning our wedding. In addition to Alex, my deep appreciation goes to Ariana's family, who welcomed me with open arms: Robin, Kyriakos, Sharon, Eric, Jasmin, Nathaniel, Michelle, Daniel, Aaron, Brian, and Cinder.

Some have sparked my love of thinking from the beginning. Thank you, Mom and Dad (Kathleen and Joe); Adella Croft, Michael Young, and Hiram Golze; Canada/USA Mathcamp; and Michael Zhao, whose passion, intellect, and laughter I will always remember. Thank you to my undergraduate mentors: Marion Scheepers, for a summer that made me certain that I wanted to do research as a career; Tim Nelson, for getting me excited about formal methods for systems; and Anna Lysyanskaya, who encouraged me and helped me apply to PhD programs in computer science seven years ago.

ABSTRACT

SAFE PROGRAMMING OVER DISTRIBUTED STREAMS

Caleb Stanford

Rajeev Alur

The sheer scale of today's data processing needs has led to a new paradigm of software systems centered around requirements for high-throughput, distributed, low-latency computation. Despite their widespread adoption, existing solutions do not provide a programming model with safe semantics – and they disagree on basic design choices, in particular with their approach to parallelism. As a result, naïve programmers are easily led to introduce correctness and performance bugs.

This work proposes a reliable programming model for modern distributed stream processing, founded in a type system for *partially ordered data streams*. On top of the core type system, we propose language abstractions for working with streams – mechanisms to build stream operators with (1) type-safe compositionality, (2) deterministic distribution, (3) run-time testing, and (4) static performance bounds. Our thesis is that *viewing streams as partially ordered conveniently exposes parallelism without compromising safety or determinism.* The ideas contained in this work are implemented in a series of open source software projects, including the Flumina, DiffStream, and Data Transducers libraries.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iv

ABSTRACT vi

LIST OF FIGURES x

CHAPTER 1: Introduction 1

1.1 Motivation 1

1.2 A Programming Example: Values and Barriers 3

1.3 Our Approach 5

1.4 Contributions 6

1.5 Software 8

1.6 Attribution 8

CHAPTER 2: Background 9

2.1 Distributed Stream Processing Systems 9

2.2 Commonalities 11

2.3 Limitations 16

2.4 Summary: Our Viewpoint 17

CHAPTER 3: Related Work 18

3.1 Dataflow Programming 18

3.2 Correctness Support for Data-Parallel Programs 22

3.3 Partially Ordered Trace Theory 24

3.4 Other Paradigms 26

3.5 Selected Systems Challenges 29

CHAPTER 4: A Foundation for Streams 32

4.1 Stream Types 33

4.2 Views of Streams 35

4.3 Examples . . . 41
4.4 Isomorphism between Views . . . 43
4.5 Subtyping . . . 48
4.6 Monotonicity, Type Safety, and Determinism . . . 50
4.7 Outtakes . . . 59
4.8 Historical Notes . . . 65

CHAPTER 5: Compositionality . . . 74
5.1 Notational Differences . . . 75
5.2 Design Goals . . . 75
5.3 Syntax and Semantics . . . 77
5.4 Examples . . . 81
5.5 Monotonicity . . . 84

CHAPTER 6: Distribution . . . 87
6.1 Motivation . . . 87
6.2 Contributions . . . 88
6.3 System Architecture . . . 90
6.4 Programming Model: Dependency-Guided Synchronization . . . 92
6.5 Execution Model: Synchronization Plans . . . 100
6.6 Experimental Evaluation . . . 107
6.7 Case Studies . . . 115
6.8 Outtakes . . . 118
6.9 Discussion . . . 120

CHAPTER 7: Testing . . . 122
7.1 Motivation . . . 123
7.2 Contributions . . . 124
7.3 Example Use Cases . . . 126
7.4 DiffStream . . . 128
7.5 Writing Specifications in DiffStream . . . 129
7.6 Differential Testing Algorithm . . . 132
7.7 Practical Bounds on Space Usage . . . 143

7.8 Evaluation . . . 144
7.9 Discussion . . . 155

CHAPTER 8: Performance Bounds . . . 158
8.1 Motivation . . . 159
8.2 Contributions . . . 160
8.3 Data Transducers . . . 162
8.4 Examples . . . 167
8.5 Constructions . . . 170
8.6 The QRE-Past Monitoring Language . . . 185
8.7 Succinctness . . . 191
8.8 Discussion . . . 195

CHAPTER 9: Outlook . . . 196
9.1 Vision . . . 196
9.2 Short-Term Ideas . . . 196
9.3 Long-Term Ideas . . . 197

APPENDIX . . . 199
A.1 Word Clouds . . . 199
A.2 Epigraph Outtakes . . . 200
A.3 Special Mention . . . 201

PRIMARY REFERENCES . . . 201

BIBLIOGRAPHY . . . 202

LIST OF FIGURES

Page

FIGURE 1.1: Overview of the work included in this thesis. 7

FIGURE 2.1: A selection of major distributed stream processing systems. 10
FIGURE 2.2: Example dataflow graph. 12
FIGURE 2.3: Example dataflow graph after operator replication. 14

FIGURE 4.1: Illustrative partially ordered stream. 33
FIGURE 4.2: Example stream type for Figure 4.1 drawn as a tree. 34
FIGURE 4.3: Monotonicity, type safety, and determinism examples. 52
FIGURE 4.4: Monotonicity, type safety, and determinism for the remaining chapters. . . 59
FIGURE 4.5: Outtakes from Figure 4.3. 62

FIGURE 6.1: DGS system architecture. 92
FIGURE 6.2: DGS example program. 93
FIGURE 6.3: DGS semantics. 96
FIGURE 6.4: Example synchronization plan. 101
FIGURE 6.5: Flink and Timely throughput experiments. 110
FIGURE 6.6: Timely code snippet. 111
FIGURE 6.7: Flink S-Plan throughput experiment. 112
FIGURE 6.8: Flink code snippet. 113
FIGURE 6.9: Flumina (DGS) throughput experiment. 114
FIGURE 6.10: Table of development tradeoffs. 114
FIGURE 6.11: Example optimized synchronization plan. 119
FIGURE 6.12: Flumina (DGS) latency experiment. 120

FIGURE 7.1: DiffStream example use case 1. 126
FIGURE 7.2: DiffStream example use case 2. 127
FIGURE 7.3: DiffStream architecture. 129
FIGURE 7.4: DiffStream example specification 1. 130

FIGURE 7.5: DiffStream example specification 2. 130
FIGURE 7.6: DiffStream example specification 3. 131
FIGURE 7.7: DiffStream example specification 4. 131
FIGURE 7.8: DiffStream algorithm: checking equivalence of two streams. 132
FIGURE 7.9: Example logical order of a stream. 134
FIGURE 7.10: DiffStream example code. 146
FIGURE 7.11: Manual parallelism case study results. 148
FIGURE 7.12: MapReduce case study results. 150
FIGURE 7.13: Example MapReduce code. 151
FIGURE 7.14: Performance case study results (Yahoo streaming benchmark). 154

FIGURE 8.1: Data transducer semantics illustration. 165
FIGURE 8.2: Data transducer evaluation algorithm. 166
FIGURE 8.3: Data transducer example 1. 169
FIGURE 8.4: Data transducer example 2. 170
FIGURE 8.5: Data transducer example 3. 171
FIGURE 8.6: Summary of the QRE-Past language. 186

CHAPTER 1 : Introduction

Today, data is produced at an overwhelming rate that cannot be processed by traditional methods. For example, Cisco has estimated in its annual white paper that data produced by people, machines, and things is around 850 zettabytes, in contrast to a much smaller volume of data that can be feasibly stored [79]. Researchers and industry practice have accordingly recognized the demand for a new paradigm of computing where data is distributed (processed in parallel over many devices), transient (processed as it arrives and discarded), and temporally structured (considered with respect to time). This *stream processing* paradigm has given rise to an increasing number of modern data processing software frameworks.[1] Broadly construed, the stream processing paradigm is exemplified not only by these dedicated frameworks, but also by many other modern systems: these include microservices deployed in the cloud; IoT and other edge devices, which operate in response to sensor data [245, 36]; and programmable network switches, which can be used to push some of this expensive streaming computation into the network. Stream processing also has theoretical justification in the *streaming model of computation* [204], where items arrive one at a time and are processed as they arrive, ideally using a minimal amount of space and time per element. In this thesis, we reconsider the stream processing paradigm through the lens of *programming languages*, by investigating software abstractions which are type-safe, deterministic, and performant: that is, which save programmers from common mistakes and ensure that the deployed program meets their intent.

1.1. Motivation

Although research in stream processing can be traced back two decades from within the database community [14, 13, 31, 71, 33], and even earlier in programming languages and compilers [58, 250, 258], researchers have devoted inadequate attention to software correctness (see our short paper [11]). Today's systems are difficult to test for correctness and debug. In the context of Apache Spark, researchers found that bugs are time-consuming to diagnose due to a number of issues related to distributed deployment (e.g., a bug only shows up on a particular input item out of millions, in a particular distributed execution, or in the presence of faults) [141]. For streaming applications in Apache Flink, user studies have demonstrated that the state of practice is limited to unit and

[1] E.g.: Kinesis from Amazon [242], Timely [194] and Differential Dataflow [196] from Materialize, and Spark [118], Storm [119], and Flink [115] from the Apache Software Foundation. See Chapters 2 and 3.

integration testing [267]. For some use cases, building and deploying a correct application in today's systems can require either significant expertise in distributed systems or a good deal of experience with developing and debugging for the specific streaming framework in question. We posit that there is therefore both a need and a research opportunity for more sophisticated testing and formal correctness techniques. With the right abstractions and automated formal tools, programming correct distributed applications over data streams could be much easier for inexperienced users than it is today.

From a programming languages viewpoint, two problems stand out in particular. First, existing systems lack *safe semantics*. We will elaborate on what we mean by safety when describing our proposed model; in brief, existing systems lack *fine-grained type safety* with respect to differences in streams and how they are parallelized, and lack *determinism* with respect to all possible parallel or distributed executions. The requirement for determinism has been articulated since the early history of streaming (see [251], requirement 4) but remains absent in practice.

Second, existing systems disagree on basic details about how streams are parallelized. Points of disagreement include, but are not limited to: are stream items ordered or unordered? Are streams parallelized explicitly (via syntax) or implicitly (by the system)? Can parallel instances of a stream operator communicate via external state or communicate with external services? Can parallel instances communicate with each other, through mechanisms such as broadcasting a message to all other instances? We will discuss some of these differences in more detail in Chapter 2. In summary, *there is no common, agreed-upon semantics for stream processing.* The lack of a common language and semantics makes it difficult to design formal tool support, especially if we want the ideas to be applicable across systems.

Besides semantic bugs, misunderstanding the details of parallelism in the system is also an easy way to overlook performance bottlenecks. Performance is critical for stream processing systems, but is not formally guaranteed (see e.g., [183, 93, 151, 52]); it can be less predictable and reliable than performance at the level of hardware or operating systems.

Semantic language-level issues certainly do not constitute the only barriers to software development in today's stream processing systems. Many other user-facing problems are of critical importance – to name a few, debugging tool support, boilerplate code and configuration, input and output, and

interfacing with the operating system and with other services. We do not focus on these problems in this thesis. However, from personal experience programming in stream processing frameworks, we believe that the semantic language-level issues we focus on do have a significant programming impact.

1.2. A Programming Example: Values and Barriers

To further motivate our approach, consider the following concrete minimal example. Three streams arrive in the system: two parallel streams of *values*, which are integers (`int`), and a stream of *barriers*, which are of the unit type (`unit`). All of these events are timestamped when they arrive. The values are parallel in the sense that they arrive in the system in parallel at multiple nodes (in this case two, for simplicity). The barriers all arrive in one stream at a single node. Our task is to output the "sum of the values occurring between every two adjacent barriers." That is, whenever a barrier occurs, we want to output the sum of all the values with timestamp values since the previous barrier. This computation is sometimes known as an *event-based window* because the window of events to aggregate depends on the occurrence of certain events (in this case, the occurrence of a barrier) [252]. We assume in this scenario that barriers are much less frequent than values, and not parallelizable; i.e., they require global synchronization across all nodes.

It turns out that such a parallel computation is rather subtle to program in existing systems. High-level query libraries (e.g., based on SQL) typically don't provide event-based windows directly but require deriving them from expensive joins. Naïvely, one way to solve the problem is to send all the values to the same node as the barrier, but this results in a central bottleneck and does not exploit any parallelism. To solve the problem, one solution is to *broadcast* the barrier stream to all other parallel nodes; the broadcast primitive is provided in a few systems such as Flink and Timely Dataflow [156, 195]. Once the stream is visible from all parallel nodes, each of those streams then has to solve only a local windowing problem. The local event-based window can be achieved in multiple ways, depending on the system; at the core, it requires a re-timestamping operation to group value events with respect to the barrier-window that they fall into, where the window is either identified by a number in sequence (1 for the first window, 2 for the second, and so on) or by the timestamp of the barrier it corresponds to. After values are timestamped appropriately, they can be added

by-timestamp (a built-in operation in all systems) and then aggregated across parallel nodes (also standard) for the final output at each barrier.

The value-barrier example is simple, but emblematic of the broader problem: when parallel structure is not simply embarrassingly-parallel and requires some synchronization (in this case, synchronization based on the barrier stream), embarrassingly-parallel abstractions fail. We claim that there is a better way; that parallelism can be expressed in a semantically meaningful manner at a higher-level of abstraction. Going back to the features that we want to provide in existing systems:

- *Fine-grained type safety:* Fundamentally, the value stream and the barrier stream are quite different. The barrier stream arrives only at one node; the value stream arrives in parallel at many nodes. But some existing systems do not distinguish between these two kinds of streams at the typing level. So operations that interact between sequential streams and parallel streams are not type-safe: a consumer expecting a sequential stream may get a parallel one at many nodes resulting in a software bug.

- *Determinism:* In the "correct" computation, the output value at each barrier is a deterministic function of the inputs (including their timestamps). In our experience it is very easy to write this computation incorrectly and accidentally window values in a nondeterministic manner, e.g., if events are grouped as-they-arrive instead of by timestamp. Yet the only way to detect this sort of nondeterminism is to run the system and hope that an execution appears where the events arrive in a different order than the timestamps indicate, which is highly unlikely when the rate of events is sparse and only becomes likely under heavy input load. In practice this could become a difficult-to-detect error in production.

- *Performance:* Finally, it is very easy to write a version of this computation (such as the naïve version that sends everything to one node) that is inefficient. In an ideal world, systems would give some formal guarantees about performance through static information known at compile-time, and flag an error if parallelism is being ignored entirely or if there is a sequential bottleneck.

As the value-barrier example illustrates, much of the work in this thesis stems from a desire to go beyond primitive parallelism: where primitive parallelism includes "everything is unordered," "everything is ordered," or "events are partitioned by key and ordered for each specific key." Incorporating

only these three kinds of parallelism represents the state-of-the-art, but is ad hoc and does not fare well in examples such as the value-barrier where there is inter-dependent ordering between events.

1.3. Our Approach

This discussion and example illustrate that, fundamentally, existing systems over streams disagree on the semantics for parallelism. In fact, they disagree on a rather more foundational question: what is a *stream*? In this thesis, we investigate a view of *streams as partially ordered sets*. The above example is a typical case: values are unordered with respect to each other, but ordered with respect to barriers. We argue that viewing streams as partially ordered sets allows for safe abstractions which are type-safe, deterministic, and performant.

We approach the problem of *specifying* these partial orders type-theoretically: we begin in Chapter 4 by outlining a type system for streams which serves as a foundation for the rest of the thesis. Our type system is an abstraction over *dependence relations*, which are studied in concurrency theory going back to Mazurkiewicz [192]. Historically, we defined two typing disciplines for partially ordered streams: data-trace types [5, 9], and synchronization schemas [2]; the type system in Chapter 4 is a modification of synchronization schemas based on our current thinking. We define stream types and show how streams (elements of the types) can be viewed equivalently as structured data called *batches*, as finite sequences up to ordering equivalence called *linearizations*, or as labeled partially ordered sets (traditionally known as partially ordered multisets, or pomsets). We show that all of these views are formally isomorphic. A key theorem is that *subtyping* is decidable in quadratic time. As a prelude to the rest of the thesis, we formally define three key properties: *monotonicity*, *type safety*, and *determinism* for operators over streams. We also state and prove compositionality laws for these three properties.

In the remaining chapters, we show how to build safe abstractions over stream types. In Chapter 5 (based on material from [2]) we consider how to define operators over streams compositionally; this amounts to a form of type-safe programming, but does not directly address determinism or performance. In Chapter 6 (based on material from [8]), we consider how to automatically parallelize operators in a way that guarantees determinism. This work includes a programming model (Dependency-Guided Synchronization), a semantics and execution model, a compiler framework,

and a code generator; it is implemented in Flumina, a prototype streaming system in Erlang. In Chapter 7 (based on material from [6]), we consider how to test for determinism dynamically via differential testing [193]; this can be seen as a dynamic type-checking problem. Our core algorithm and tool, DiffStream, can also be used more generally to check equivalence assertions between streams at runtime.

Finally, in Chapter 8 (based on material from [4], see also [3, 1]), we address provable guarantees about performance. This is a challenging problem in general; we consider only the sequential case, without parallelism. In order to provide upper bounds on performance, we investigate Data Transducers, a finite-state model of stream processing operators which have streaming algorithms for their evaluation. To demonstrate how finite-state models are useful for streaming, we show how to compile high-level query languages on a single machine with space and time bounds [25, 187]. Concretely, a query of size $O(n)$ can be compiled to a data transducer which uses $O(n^2)$ time and space to process each element of the input stream, measured in number of data accesses and data operations.

We discuss related work in more detail in Chapter 3, but a few lines of work have been important enough to ours to mention in the introduction. The theory of Mazurkiewicz traces [192, 97] is the basis for partially ordered streams. Kahn Process Networks [164] pioneered deterministic concurrent dataflow programming. Among prior language proposals, StreamIt [258] constitutes a particularly principled past language design based on Synchronous Dataflow [176]; the SPL language at IBM has previously addressed questions of safe (deterministic) parallelism [147, 238, 148]; and the Continuous Query Language [32, 33] has been a source of inspiration in its simplicity. On the formal languages side, the theory of finite-state transducers in general [104, 27] and quantitative automata in particular [240, 99, 24] have played a direct role in the development of our performance-bounded model in Chapter 8.

1.4. Contributions

The structure of the thesis is displayed visually in Figure 1.1. In summary, our contributions are as follows:

- We propose a foundational type system for distributed streams based on partially ordered sets. We show that under our type system, streams can be equivalently viewed as structured

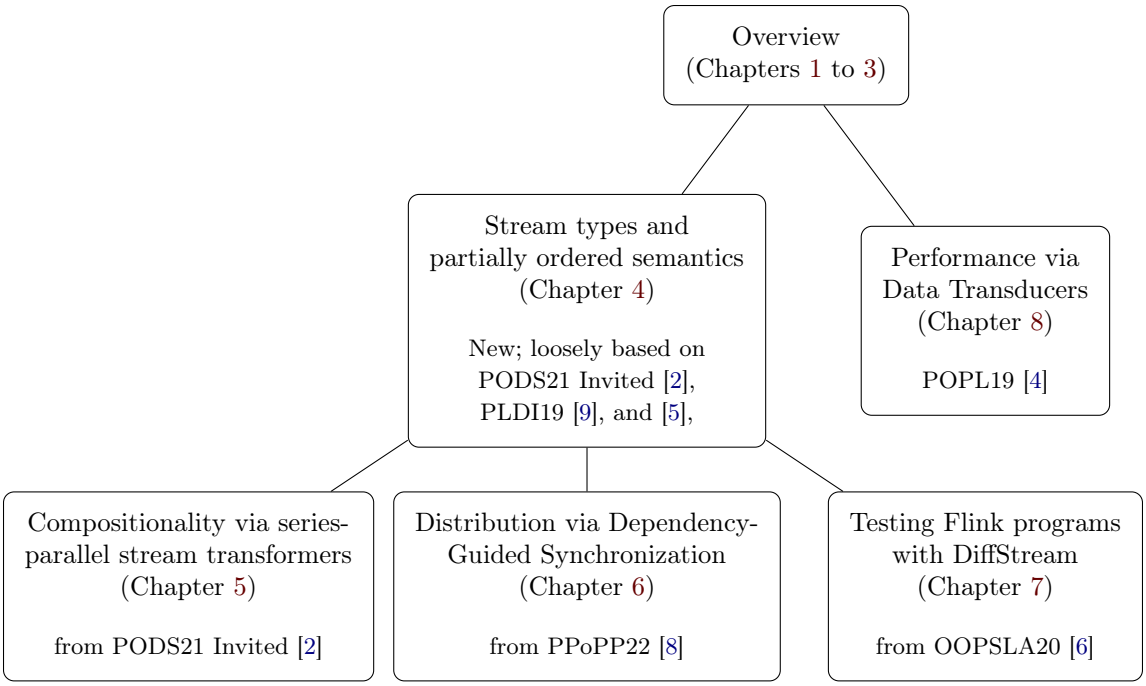


Figure 1.1: Overview of the work included in this thesis.

hierarchical data called *batches*, as finite sequences of events called *linearizations*, or as labeled partially ordered sets (traditionally known as pomsets). In the historical notes, we also discuss how these closely relate to our earlier synchronization schemas [2] and data-trace types [9]. (Chapter 4)

- We show how stream operators can be defined compositionally on top of our core type system. [2] (Chapter 5)

- We propose *dependency-guided synchronization*, a programming model and system for safe (deterministic and semantics-preserving) distribution [8]. (Chapter 6)

- To detect bugs due to nondeterminism in existing stream processing applications, we propose *DiffStream*, a differential testing tool [6]. In particular, we leverage the partial order viewpoint to specify ordering requirements, and we test for violations at runtime. (Chapter 7)

- Towards streaming applications with predictable performance, we propose *data transducers*, a monitoring formalism and state-machine based intermediate representation [4]. Our formalism is

compositional, enabling compilation of high-level monitoring queries with provable performance bounds. (Chapter 8)

Finally, Chapter 9 concludes with our outlook for the future of this work.

1.5. Software

The work in this thesis is implemented in a number of open-source tools available on GitHub.

- Flumina[2] is a parallel programming model for stream processing with safe distribution, written in Erlang, with experiments against Flink and Timely Dataflow.
- DiffStream[3] is a differential testing tool for Apache Flink.
- Data Transducers[4] is an intermediate representation for streaming with formal performance bounds, written in Rust.

1.6. Attribution

Most of the work presented in this thesis was done in close collaboration with my advisor, Rajeev Alur, and other coauthors: particularly Konstantinos Mamouras (for Chapter 8) and Konstantinos Kallas and Filip Nikšić (for Chapters 6 and 7). I wrote all the included material in Chapters 1, 2, 4, 5 and 9, the material on the programming model in Chapter 6, one of the case studies and other miscellaneous sections in Chapter 7, and almost all the material in Chapter 8. Chapter 2 incorporates material from my WPE-II written report [10] (of which I am the sole author), and Chapter 3 integrates some text from all of the papers included in the thesis.

The software repositories Flumina and DiffStream are shared projects with my collaborators, Konstantinos Kallas and Filip Nikšić. For Flumina, I contributed the experiments and infrastructure with Timely Dataflow in Rust, the Smart Home Power Prediction case study, and documentation. For DiffStream, I contributed the MapReduce case study and documentation. The Data Transducers development in Rust is solely my own work.

[2] https://github.com/angelhof/flumina
[3] https://github.com/fniksic/diffstream
[4] https://github.com/cdstanford/data-transducers

CHAPTER 2 : Background

The fourth requirement is that a stream processing engine must guarantee predictable and repeatable outcomes.

—Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik in "The 8 Requirements of Real-Time Stream Processing," 2005 [251]

2.1. Distributed Stream Processing Systems

Today's de facto programming solution for programming over distributed streams is found in *distributed stream processing systems (DSPSs)*. Popular modern DSPSs include Apache Flink [115, 62], Timely Dataflow [194, 203], and Apache Spark Streaming [118, 285]. A selection of these and other systems is included in Table 2.1. There are a vast number of DSPSs beyond what can be listed here, including many research prototypes as well as actively developed software products in widespread use. In this chapter, we review the primary defining properties common to DSPSs, with a particular focus on the programming model and semantics. We then discuss the limitations of the existing model and systems.

2.1.1. Comparison with Batch Processing

To understand the emergence of DSPSs and why traditional software infrastructure is not sufficient, note that traditional software is often based on the assumption that critical data can be stored and then processed later. For example, much of large-scale data analytics relies on processing data in large *batches* (e.g., training a machine learning model daily), such as via MapReduce jobs [94]. While batch processing does take advantage of distributed computing resources, it does not take advantage of the transient and temporal structure of data, and incurs data storage costs. In practice due to these costs, most data traveling over the internet and processed by cloud services is not harvested to its full potential. Additionally, the temporal structure of data is lost in batches, which divide data at arbitrary boundaries in time. For example, a machine learning model that might benefit from continuous updates is instead trained only on the data from yesterday. DSPSs offer a software framework for writing such programs, where data processing logic is defined in a platform-independent manner, then deployed as a distributed application over many nodes. DSPS performance is measured

System	Year	Stable Release	Active?	Questions on StackOverflow (as of 2022-06-07) [216]
Aurora	2003 [14]	2003 [262]	No	—
Borealis	2005 [13]	2008 [263]	No	—
	2011 [190]	2022 [119]	Yes	2564
 Apache Flink [115, 62]	2011	2022 [128]	Yes	6428
Google MillWheel	2013 [20]	—	No	—
 (Apache Spark Streaming)	2013 [285]	2022 [118]	Yes	5408
 [117, 211]	2013	2020 [121]	Yes	81
Timely Dataflow	2013 [203]	2021 [194]	Yes	—
 [116, 170]	2015	2021 [126]	Yes	43

Figure 2.1: A selection of major distributed stream processing systems.

in terms of *latency* and *throughput*, while batch processing performance is primarily measured in terms of throughput only. Concretely, DSPSs aim for latency in the milliseconds and throughput in tens of thousands of events per process per node.

2.2. Commonalities

While DSPSs differ in many substantial ways, all DSPSs in current use share the so-called *dataflow programming model*. This means that the programmer writes, in some form or another, a dataflow graph. In some systems, such as Apache Storm, the dataflow graph is written out explicitly, whereas in others, such as Apache Flink, the graph is implicit. Additionally, systems may offer high-level libraries for creating or composing dataflow graphs; in particular, these include libraries for complex windowing operations and for SQL- and CQL-based streaming queries. The dataflow programming model exposes task and pipeline parallelism; to expose additional data parallelism, DSPSs use *operator replication*. Similar to how a MapReduce [94] job is implicitly parallelized, all operators in a dataflow graph (unless configured otherwise) may be split into several copies; this is part of the programming model as well, and affects the semantics.

2.2.1. The Dataflow Programming Model

We introduce the programming model through a simplified example based on a real-time video analytics use case. Imagine a large-scale system of video cameras, perhaps located in several cities throughout a country. Each video camera produces a stream of video data, at a certain frame rate and image resolution. Suppose that we want to identify pedestrians and report to a central location the summary of all pedestrian activity in the last 10 minutes, i.e., where pedestrians are most active. To do so, we want to classify each image from each camera using an out-of-the-box classifier; then, to prevent noise and to summarize the total activity, we want to aggregate the data from all classifiers in the last 10 minutes in a particular location (e.g., one intersection or group of intersections). In the end, we report for each location the total amount of pedestrian activity. (One could imagine taking further steps, such as adding a smoothing filter which removes reports of pedestrian activity that last only for a single frame, assuming that these must be erroneous.)

A dataflow pipeline for this is given in Figure 2.2. The input data consists of raw video data. Notice that the pipeline contains only one operator at each stage; that is, we treat all input data

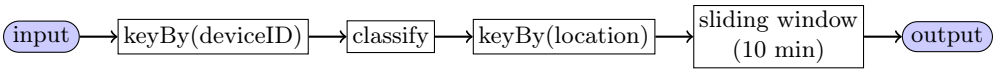


Figure 2.2: Example DSP dataflow graph for a program to classify video streams and report total pedestrian activity.

at *all* cameras as a single input stream; and we write transformations over that stream. The first transformation, `keyBy(deviceID)`, says to partition the stream into substreams for different keys, where each key is a device ID. Since the stream is already physically parallel by camera, this does not have any physical effect, but it makes the parallelism visible in the dataflow. In fact, depending on the system, this `keyBy` may be left implicit. The second transformation, `classify`, says to process each input data item, which is a video frame, and return the classification (1 for a pedestrian, 0 for no pedestrian). The third transformation is similar to the first, but this time we group by location, instead of by device ID. The final transformation is the sliding window which adds up all the values over the last 10 minutes. This is then output and could be displayed to the user as a real-time report of activity across all locations.

In general, streaming dataflow graphs are acyclic, although some systems support ways to provide feedback and/or support iterative (cyclic) computations. The input and output nodes in a dataflow are special, because they interact with the external system, and can usually be of several kinds: e.g., taking input from a distributed file system, taking input from a live stream of data, writing output to a file, or in general interacting with some external service which provides input or consumes output. It is generally preferred that *internal nodes do not consume input or produce output*; however, that does not mean they are pure; they are often stateful, and may also produce logs, interact with a stored database, etc. We summarize the following definition of a streaming dataflow graph that is (approximately) common to all DSPS programming models.

Definition 2.2.1 (Streaming Dataflow Graph)**.** An *acyclic streaming dataflow graph* (DSP dataflow graph) consists of a set of input nodes called *sources*, a set of intermediate nodes called *operators*, and a set of output nodes called *sinks*, connected by a set of directed edges which are *data streams*. The nodes and edges form a directed acyclic graph (DAG). Each source node may produce data items continuously, and each sink node consumes data items.

An *operator* is a possibly stateful streaming function that describes how to process an input item and when to produce output. It does not get to choose which of its input streams to read from; rather, it

has an input event handler which can be called on any event when it arrives. It may produce any number of outputs for one input item, in any combination of output streams.

The number of output items produced per input item is often called the operator's *selectivity*, which may be e.g., 1 (for a map), $0-1$ (for a filter), or more than 1 (for a copy). The fact that the selectivity is not constant is a major challenge that makes scheduling DSPS applications more difficult.

2.2.2. Auto-Parallelization (Operator Replication)

DSPSs rely on three types of parallelization to achieve scalability, especially to achieve high throughput. The first two are explicitly exposed in any acyclic streaming dataflow graph. *Pipeline parallelism*, which is visible in Figure 2.2, means that different operators in a sequential pipeline can be run by different workers, threads, or distributed nodes. *Task parallelism* is not shown in our example, but means that different operators in a parallel set of disjoint tasks (parallel nodes in the dataflow) can be run by different workers, threads, or distributed nodes. However, the arguably most important form of parallelism for huge data sources is *data parallelism*, where different data items in the same input stream are processed by different workers, threads, or distributed nodes. DSPSs use *auto-parallelization* (also known as *operator replication* or *sharding*) to accomplish data parallelism, and unlike the other two kinds of parallelism, it modifies the dataflow graph and potentially the semantics of the program.

In our example of Figure 2.2, we want to exploit data parallelism on video streams from different cameras. In the `classify` stage of the pipeline, we are classifying images from different cameras separately, so it should be able to be run in parallel. The problem with data parallelism is that it would be cumbersome for the programmer to expose on their own; they would be forced to explicitly write dozens or hundreds of copies of the `classify` operator, and manually divide the source into dozens or hundreds of different sources, so that each operator got its own subset of the input data. To avoid this, DSPSs automatically replicate operators in the dataflow graph into several parallel copies. Typically, the number of parallel copies can be configured by the programmer by setting the level of parallelism.

On our example, a possible auto-parallelized graph produced by the system is shown in Figure 2.3. Each operator is replicated, in this case into 3 copies. However, this does not fully describe what happens, because we have to understand the connections between operators. If there was an edge

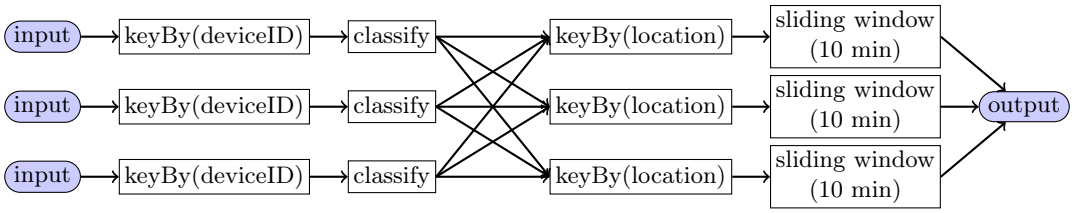


Figure 2.3: Example dataflow graph after operator replication.

before, now there are 9 possible edges, and the system must decide which of them to use, and how to send data along the edges.

Typically there are several possible strategies, and the system chooses one based on context and/or explicit user configuration. One strategy is *round-robin*, where outputs from one stage of the pipeline are sent to the inputs of the next stage in round-robin order (so for every 3 outputs from one stage, 1 gets sent to each of the 3 operators in the following stage). A second strategy is *key-based partitioning* where the operator copy that an item is sent to depends on (a hash of) a specified key. Finally, in cases where there are the same number of parallel copies from one stage to the next, it is common to *preserve the same partitioning* from one stage to the next.

The partition operator `keyBy` in our example dataflow graph has no effect except to force a particular strategy for connections between stages: key-based partitioning based on the specified key. In Figure 2.3, first we assume that the input arrives partitioned by device ID. The `keyBy` by device ID then preserves such a partitioning. All future operators preserve the same partitioning, except when there is a second `keyBy` by location – this operator re-partitions the data based on location instead of device ID.

We make no assumption about the different DSPS programming models and how they handle connections between stages, except that they should obey any constraints that are explicitly programmed by the user. We capture the allowable parallelization in an annotated DSP graph in the following definition.

Definition 2.2.2 (Annotated Dataflow Graph and Parallelization)**.** An *annotated dataflow graph* consists of a dataflow graph annotated with, for each edge, whether the connection between parallel operator replicas should be (1) round-robin, (2) on the basis of a particular key, (3) partition-

preserving, or (4) unspecified. Additionally, each vertex may be labeled with a level of parallelism that is allowed (1 for no parallelism).[1]

A *parallelization* of an annotated dataflow graph consists of a larger graph, without annotations, where: (i) each vertex is replicated to between 1 and i copies, where i is allowed level of parallelism; (ii) for each edge between v and v', if v has i copies and v' has i' copies, the connection must be consistent with the annotation. Specifically, (1) for round-robin items should be assigned to each operator in turn; (2) for key-based partitioning there should exist *some* partitioning of the keys such that that partition determines where a data item is sent next; (3) for partition-preserving, we must have $i = i'$ and the connections are one input to one output; (4) for unspecified, each output item from one stage may be sent to any of the operators for the next stage, as the system sees fit.

2.2.3. Streaming Runtime (Fault Tolerance and Scheduling)

Once given a parallelized dataflow graph, the primary job of a DSPS runtime is to *schedule* workers in a distributed cluster so as to execute all the operators, continuously and in parallel. The goal of the scheduler is to maximally utilize the available distributed resources, and to prevent one task from becoming a performance bottleneck (called a *straggler*). In the case of stragglers, there are common techniques for the system to respond, e.g., *throttling* and *back-pressure*.

The performance of the system is generally measured in terms of *latency* and *throughput*. Latency is the time it takes for an output item to be produced after the input item which triggered its production arrives in the system. Throughput is total number of input items successfully processed per unit time. Throughput is often measured as *max throughput*, the maximum input rate the system can handle before it breaks. Given fixed resources, there is always some limit to how much input the system can handle; so max throughput is always finite. Beyond the max throughput, DSPSs offer few guarantees about behavior, and will likely suffer increasing latency, drop data, or crash altogether.

Finally, DSPS runtimes try to provide all of this with a guarantee of *fault-tolerance*. Whenever a worker is assigned to process some set of items from the input stream, the worker may fail. If this occurs, the system must have a way to rewind back to a safe state and re-process those input items,

[1] Not all annotations are meaningful; for example, in the partition-preserving case, the level of parallelism should be the same from one vertex to the next.

or re-assign them to a new worker. It is challenging to accomplish this in a way that minimizes overhead and also minimizes the time to recovery when experiencing a fault (see [266]).

2.3. Limitations

- *Disagreeing semantics.* There exists no widely accepted common semantics for distributed stream processing. DSPS applications are always written as dataflow graphs, and there are other common elements, but beyond this different APIs make different choices. For example, Flink's API assumes that data arrives in-order per-key, whereas Timely's API does not offer this guarantee (i.e., all data may arrive out-of-order). Specific constructs then come with other differences, for example: whether watermarks (indicating stream progress) are explicitly available or implicit; and whether side effects are allowed in an operator or whether the system makes no guarantees in the presence of side effects.

- *Nondeterminism.* Unfortunately, auto-parallelization of stream processing applications is not semantics-preserving [9], which results in nondeterminism due to ordering of distributed events. If nondeterminism affects the output, it is usually undesirable as it can lead to bugs that are difficult to identify and reproduce.

  In practice, many standard operators are not affected by such reordering: e.g., commutative, associative reduce operations or stateless maps and filters. However, most DSPSs do not enforce annotations to determine when auto-parallelization is safe or not [238].

- *Low-level state management.* DSPS applications are built under the assumption that users should not have to write state management logic on their own, instead relying on predefined dataflow operators (e.g., maps, filters, aggregation, windowing, and SQL query libraries). In practice, however, some streaming operations require custom logic. Examples of more complex logic include interpolation (fill in missing input data in a temporally dependent manner), machine learning operators (aggregate and update a statistical model), and event-dependent windows (form a window with data-dependent start and closing times). Because of the ubiquity of such manual state management tasks, popular APIs (including Storm, Flink, and Timely) allow users to program operators manually, e.g., by providing a state type, an initial state, and an update function for each input tuple. However, such operators are difficult to program

because they must work under the auto-parallelization mentioned above. Additionally, if the operator requires interaction with an external service or has side effects (e.g., querying a database), state update logic has to be tolerant to unexpected behavior in case of node failures or network communication. In practice, these behaviors can go unobserved unless developers explicitly test for it, e.g., by introducing controlled node failures using a tool like Chaos Monkey [267].

- *Manual parallel programming.* Related to state management, DSPSs also promise a programming model where a user does not have to parallelize their application themselves. However, in practice, many applications are difficult to parallelize and require low-level constructs: for example, a *broadcast* construct is used to share important information with all other nodes. This and similar constructs are prone to correctness bugs.

- *Unpredictable performance.* Because operators and input data are partitioned by the system, users do not describe explicitly how to partition them. However, DSPSs do not offer any concrete guarantees about the throughput or latency of the runtime. In particular, unexpected performance bugs arise in the case that partitioning is not efficient. For instance, in an example application processing an input stream of webpage views, if most of the views come from the same website, partitioning by key fails and results in a performance bottleneck.

2.4. Summary: Our Viewpoint

Considered as systems, today's stream processing systems are usually quite effective – thanks to decades of engineering and research advances. They scale automatically across threads and distributed devices, they are high-throughput and meet microsecond-level latency requirements, and they seamlessly execute simple data processing tasks (e.g., windowing, mapping, grouping, and aggregating). Considered as programming languages, however, they fall short of today's standards. If adopted, tools for formal correctness properties (including fine-grained type safety and determinism) could improve the software development processes and toolchains in this space.

CHAPTER 3 : Related Work

The sequence is represented by a function called a stream*, which is a functional analog of a coroutine...*

—William H. Burge, 1975 [58]

Stream processing research, in particular the study of SPSs, can be traced back at least as far as the 1960s, although not always in a form that is immediately recognizable as such today.

—Robert Stephens, 1997 [250]

Of course, the notion of a stream as a programming abstraction has been around for decades...

—Thies, Karczmarek, and Amarasinghe, 2002 [258]

While the previous section discussed the primary programming paradigm in current use for distributed streams and its limitations, this section provides a more general survey of related work. We include dataflow programming paradigms, considered broadly; correctness support for both stream and batch data-processing; and the study of partially ordered traces in concurrency theory. We also survey other paradigms related to streaming (functional reactive programming, concurrent and distributed programming, state machines for streaming, and runtime monitoring). Finally, we include a non-exhaustive list of research on systems problems (distribution, parallelization, optimization, benchmarking, and profiling).

3.1. Dataflow Programming

3.1.1. Distributed Stream Processing Systems

Applications over streaming data can be implemented using high-performance, fault tolerant distributed stream processing systems (DSPSs), such as Apache Flink [62, 60, 115], Storm [119, 120], Spark Streaming [285, 118], Kafka [132], Samza [211], Heron [170, 116], and Beam [127]; Timely

Dataflow (Naiad) [203, 194] and Differential Dataflow [196]; Microsoft StreamInsight [21] and Trill [70]; IBM SPL [147]; Google MillWheel [20] (now replaced by Google Cloud Dataflow); Amazon Kinesis [242]; and early systems such as Aurora [14, 262], Borealis [13, 263], STREAM [31], and TelegraphCQ [71].[1] Stream processing is closely related to, and sometimes synonymous with, distributed event processing [206] and complex event processing [276]. See [37] for an early (2002) report on the status of the field. See also [250] for an even earlier (1997) report from the perspective of dataflow languages and reactive systems.

The core programming model for DSPSs is typically based on dataflow with auto-parallelization, as discussed in Chapter 2, and this is the primary point of comparison for this thesis. However, there are other programming models used for streaming, including high-level query languages (often based on SQL), extensions to the dataflow model, and extensions to stream parallelism and distribution.

3.1.2. High-level Query Languages

High-level query languages for streaming include CQL [32, 33], CACQ [186], CEDR [39], Streaming SQL [158, 41], SamzaSQL [222], Structured Streaming [34], and StreamQRE [187]. These languages have existed since the early history of streaming research from the databases community [251]. They offer the promise of convenience and clean semantics, but they can be expressively limited for some use cases (e.g., the value-barrier example discussed in the introduction), as we discuss in [8]. They typically offer type safety with respect to the relational schema of each stream and determinism with respect to processing order – though there are cases where determinism is not guaranteed, including some possible implementations of CQL's tuple-based windows where ties must be broken arbitrarily.[2] Traditional query languages view streams in the *sequence-of-relations* model popularized by CQL, which is limited in expressiveness compared to general partial orders. Inherently sequential operators, such as tuple-based windows and interpolation, are awkward in the sequence-of-relations model.

In modern systems, query operators (typically including maps, windows, filters, joins, and aggregates) are often implemented not as a separate stream management platform but as operators on top of the core dataflow programming model, which can also support custom stateful processing and where data need not conform to the relational schema viewpoint. The upshot of this two-layered design is

[1] Some relational database systems also support a limited form of streaming; for instance, streaming SQL in Apache Calcite [42, 129] and Streaming Columns in Apache Derby [114].

[2] Thank you to Phillip Hilliard for this observation.

that the parallelism present at the core dataflow programming model level is relevant even for query languages as: (i) it dictates the extent to which streams can be distributed and optimized, and (ii) it forces semantic requirements on the query in case determinism is required, as otherwise distribution will not be semantics-preserving.

3.1.3. Traditional Dataflow Networks and Synchronous Languages

Dataflow programming predates stream processing. Dataflow programming as a solution to parallel and distributed computing originated with Kahn Process Networks [164] (KPN), a deterministic dataflow model based on the restriction that channels are FIFO queues with blocking reads and non-blocking writes. Today's systems wish to offer on-demand processing and avoid arbitrary buffering, so they typically do not implement the KPN approach for streams. In addition to general KPNs, one restriction of KPNs has been particularly influential: synchronous dataflow [176], which further restricts the FIFO queues so that a fixed number of items are read and written on each cycle of an operator. StreamIt [258] shows that the synchronous dataflow restriction enables aggressive optimization and scheduling; it also somewhat alleviates the problem with blocking reads, because the presence or absence of input and output items is always known statically. Similar to StreamIt, the work on synchronous languages from the 90s including LUSTRE [142] and ESTEREL [46] (see [43] for an overview) benefits from the assumption of synchrony.

The problem with traditional synchronous dataflow languages is that they cannot implement operators which produce a non-static number of output items in response to an input – including the very most basic such operator, *filter*. (The filter operator applied to an input stream discards items that do not match a given predicate.) As a result, the synchronous model is generally considered too restrictive today for general streaming [238].

3.1.4. Modern Dataflow Languages

A more modern take on dataflow programming is MapReduce online [81]; this exemplifies the viewpoint that streaming dataflow graphs are like MapReduce operators chained together. Other works on streaming specifically focusing on the programming model include SPADE [133] and Brooklet [248]. SPADE supports a fixed list of useful operators for streams, like tuple-level transformations (map, flat-map, filter), aggregations, and streaming joins or barriers; it also incorporates some punctuation-related operators. Brooklet is more like imperative programming, and is extremely general: it allows

translations from CQL, StreamIt, and a (non-streaming) MapReduce like language called SawZall [227]. Another non-streaming, but popular, MapReduce-based dataflow language is FlumeJava [67].

3.1.5. Extensions to the Dataflow Model

The dataflow model is limited in its ability to express iterative or recursive queries and other queries with periodic synchronization between nodes, leading to various extensions focused on better expressiveness. Naiad [203, 194] proposes *timely dataflow* in order to support iterative computation. We compare closely with Timely, the implementation of timely dataflow in Rust, in Chapter 6. In brief, though Timely is very expressive it is also often quite low-level; as a result, it falls short of automatically scaling without high-level design sacrifices (exposing implementation details to the user). To avoid semantic issues with out-of-order data, Timely makes a simplifying assumption that all events are unordered. However, this also necessitates extraneous buffering, complicating the programming model in cases where order would be known between events.

As data processing applications are becoming more complex, evolving from data analytics to general event-driven applications, some stream processing and database systems are moving from dataflow programming to more general actor models [61, 45, 44, 243, 279]. For example, Flink has recently released Stateful Functions, an actor-based programming model running on top of Flink [19, 125]. Actor models are very expressive, but computations need to be implemented manually as message-passing protocols.

Other extensions to the dataflow model focus on enabling forms of synchronization between nodes or other concurrency control, or communication with external state. Communication between parallel nodes is disallowed in the usual formulation of dataflow auto-parallelization. For example, S-Store and TSpoon [197, 17] extend stream processing systems with online transaction processing, which can implement concurrency control, and Noria [136] and Nova [288] extend stream processing systems with abstractions for operators to access shared state. Another way to deal with the problem of communication between nodes is to use *broadcast state*, a low-level messaging mechanism present in Timely [195] and Flink [156] which lets one node broadcast a message to all other parallel nodes. We discuss our alternative solution to all of these issues related to concurrency and synchronization in Chapter 6. In brief, our model avoids sacrificing high-level design by making the distribution of a

program independent of the programming model; we guarantee that, subject to an assumption that the the program satisfies some consistency conditions, the implementation is deterministic.

3.1.6. Summary

In all of the existing works on dataflow surveyed, streams are typed at a coarse-grained level: typically, only using a construct such as `Stream<T>` for a stream of events of type `T`, and not encoding the possible parallelization. Some languages include a few variants, such as Flink's `KeyedStream` which is parallelized by key, and CQL's distinction between streams `Stream<T>` and time-series relations `Relation<T>` which are created from streams using windowing operators. Compared to coarse-grained stream types, fine-grained stream types record additional parallelization information and describe the possible parallelization at the system level, the dataflow model level, or at the query language level.

3.2. Correctness Support for Data-Parallel Programs

3.2.1. Testing

Many previous works focus on batch processing programs written in the MapReduce [94] framework [87, 280, 189, 76] (see also the recent survey [201]). Some work [281] goes beyond batch processing to study testing semantic properties of operators in general dataflow or stream-processing programs. One limitation of many of these works [87, 280, 281, 76] is that real-world MapReduce programs (and, by extension, aggregators in stream processing programs) can be non-commutative: the empirical study at Microsoft [277] reports that about 58% of 507 user-written reduce jobs are non-commutative, and that most of these are most likely not buggy. The previous work on testing would erroneously flag these programs as containing bugs due to nondeterminism (a false positive).

Differential testing [193, 139] is a well-established, lightweight, black-box method to detect bugs in complex programs by simply comparing two programs that are supposed to be equivalent. In Chapter 7 (DiffStream), we adopt differential testing to finding bugs due to parallelism with the goal of avoiding the false-positives mentioned in the previous paragraph; we do succeed in avoiding false positives in most cases, though not in a few others where nondeterminism is truly inherent to the computation.

Besides DiffStream, a few other dedicated testing tools for Flink now exist: Flinkspector [169] provides unit testing; FlinkCheck [106] uses temporal logic for property-based testing; and SPOT [283] uses symbolic execution to improve path coverage. See also [161] for further reading on this topic.

3.2.2. Static Verification

In addition to testing – a dynamic method of checking correctness – there has also been research on the static verification of data-parallel programs. Recent work focuses on the verification of parallel aggregators that are used in MapReduce programs; methods include automated verification and synthesis of *partial aggregators* given an aggregation function [181], or parallelizing user defined aggregators using symbolic execution [231].

We would be interested if similar ideas could be generalized to the abstractions in this thesis, rather than just MapReduce programs; i.e., we would like to verify and synthesize operators in streaming dataflow graphs. Note that streaming graphs are not always decomposable into aggregators, and their parallel and sequential implementations might have significant structural differences (see the Topic Count case study in Chapter 7), implying that the parallel implementation cannot be simply derived from the sequential implementation.

For general stream processing, [238] has proposed an approach to ensure correct parallelization (deterministic distribution) based on categorizing operators for properties such as statefulness and selectivity. This is very closely related to our work on guaranteeing type-safe, deterministic distribution; while it is practical, it could be considered ad hoc to enumerate operators into finitely many categories. Another complication is if considering streaming graphs that interact with external service (e.g., querying a Redis database) or complex extensions to the dataflow model including operators like broadcast-state.

Instead of verifying user-written streaming programs, one can instead consider the problem of correctness for systems, compilers, and implementations. Towards testing functional correctness of a stream processing system implementation, a framework has been proposed for Microsoft StreamInsight [230].

Overall, verifying streaming dataflows statically is an important problem for future work which is not yet easily within our reach, and which we discuss further in Chapter 9.

3.2.3. Empirical Studies and Debugging

Complementary to directly establishing the correctness of user-written programs, one can look at the problem of correctness from an empirical and engineering perspective. There are a number of empirical studies which aim to classify bugs in real-world stream- and batch-processing programs. Of these, most [239, 167, 180, 289] have primarily focused on sources of job failures (e.g., system crashes) or performance issues (e.g., memory use patterns and computational bottlenecks), which are orthogonal to semantic bugs which can be found by testing. The Microsoft study [277] is the only study we are aware of that classifies semantic bugs in user-written programs. In addition to these studies of data-processing programs, there have been some empirical studies which interview users about their testing and debugging needs. In [112], users of Spark are interviewed about tools that would be useful to them, but the study focuses on *human-computer interaction* needs such as data visualization and debugging tools. The more recent study [267] aims to determine how current specialists in data stream processing applications currently implement testing. Most specialist employ unit and integration testing, together with some techniques and tools for more sophisticated testing (e.g., inducing node failures). Our work is motivated by the need to go beyond these standard techniques to increase confidence in the *semantic correctness* of user-written programs, especially in the presence of parallelism and out-of-order data.

Empirical studies can motivate work on visualization and debugging. Visualization includes generating example inputs for dataflow programs showcasing typical semantic behavior [212]. Debugging includes, e.g., setting up breakpoints, stepping through computations, and determining crash culprits [141, 213].

3.3. Partially Ordered Trace Theory

3.3.1. Mazurkiewicz Traces

Our type system builds on foundational work in concurrency theory dating back to Mazurkiewicz [192], where partially ordered sets of events are called *Mazurkiewicz traces*. Mazurkiewicz traces have been studied from the viewpoints of algebra, combinatorics, formal languages and automata, and logic [97]. In practical applications to verification and testing of concurrent systems, they appear for example in relation to *partial order reduction* [137, 223], a technique for pruning the search space of possible execution sequences.

Mazurkiewicz traces correspond to the view of streams as linearizations and as labeled partially ordered sets (see Chapter 4), traditionally called *partially ordered multisets* (pomsets). Both of these views and the isomorphism between them are standard and well-known in this literature. Our type system, however, is an abstraction on top of Mazurkiewicz traces; it gives rise not to all dependence relations, but only to certain ones that have a series-parallel structure (see Proposition 4.4.3). We believe that the series-parallel streams constitute most useful use cases in practice, where there are usually only perhaps one or two levels of nesting in the type. Most streams consist of events of maybe one or two base types, with punctuation and other system events, and our types aim to cater towards these simpler use cases.

An important technical difference is that in the theory of Mazurkiewicz traces, one usually assumes a finite, symmetric, and reflexive dependence relation [97]. In contrast, in this thesis, we only require it to be symmetric; it is neither finite (due to arbitrary infinite base-types and key-based parallelism) nor reflexive (due to the relational base type). This is in order to support user-provided dependence relations over a possibly infinite data domain, which is necessary to model common patterns in the streaming setting. Patterns such as this one cannot be captured by a finite alphabet, and this limits the direct application of classical work on concurrency theory over a finite dependence relation.

3.3.2. Checking Properties of Traces

Much classical research has focused on deciding properties of traces such as serializability, linearizability, sequential consistency, and data race detection. Broadly speaking, these properties are search problems: the algorithm monitors an execution of events, and it must decide if there exists some possible equivalent execution that witnesses the desired property. For example, race detection involves deciding, given a sequence of events, if there is a valid reordering of the events, subject to the constraints imposed by synchronization events, in which two specific events (representing a potential race condition) get reordered. If there are an arbitrary number of threads then race detection is NP-hard [207, 208] (but it is easy to decide for, say, only two traces and can be done in a streaming manner). Similarly, checking sequential consistency of a given trace is NP-complete [134], as is checking linearizability in general [135]. Practical tools for testing correctness of traces (e.g., [237, 219, 241, 274, 56, 184]) are bound by these results and explore the trade-off between soundness, completeness, and tractability.

In our work we don't generally consider algorithmic problems on traces, with the exception of Chapter 7, where we consider the algorithmic problem of checking *equivalence* of two Mazurkiewicz traces (equality of streams up to re-ordering). As with our type system, we consider this problem for general infinite-alphabet traces, rather than just those over a finite alphabet. The problem we consider is in PTIME (for the offline variant), admits a space-optimal (though not space-bounded) online monitoring algorithm, and to our knowledge hasn't been explicitly articulated in existing work on Mazurkiewicz traces.

3.4. Other Paradigms

3.4.1. Functional Reactive Programming

In the functional programming community, researchers have long investigated *functional reactive programming* (FRP) [272, 210]. See also [38] for a survey and [48] for an introductory textbook. FRP is closely related to dataflow and is suitable for streaming. The earliest work on FRP dates back to the late 1990s, targeted for interactive graphics applications [103, 102], but FRP has persistently inspired new formalizations and implementations [86, 83, 113, 224]. In FRP, streams are mostly sequential objects which are processed incrementally; FRP abstracts both discrete-time and continuous-time signals. Some research has endeavored to make FRP distributed, for example by adding mechanisms for fault tolerance [225], type-safe clocks [59] and distributed actors [246].

3.4.2. Fork-Join Based Concurrency

Fork-join based concurrent programming [130, 174] constitutes a classical parallel programming paradigm, relevant to Chapter 6. Fork-join parallel programming models are typically expressive but low-level, and do not guarantee determinism or data-race freedom. Concurrent revisions [55] guarantees determinism in the presence of concurrent updates by allowing programmers to declare types to describe how parallel updates are merged on joins. More generally, a great many proposals exist to make concurrent programming safe (typically not fully deterministic, but at least data-race free); we do not attempt to survey them all here, but for a recent example and related work, see fearless concurrency [200].

3.4.3. Distributed Programming Models and Consistency

Monotonic lattice-based programming models, including Conflict-Free Replicated Data Types [244], Bloom^L [82], and LVars [171, 172], are designed for coordination-free distributed programming. These models guarantee strong eventual consistency, i.e., eventually all replicas will have the same state. Partially ordered sets are an important concept in this space because consistency for replicated data stores often relies on determining which events are unordered and can be safely executed without coordination, and which events require global synchronization between nodes. For some examples of this distinction, see RedBlue consistency [179], MixT [198], Gallifrey [199], Quelea [247], CISE [138], Carol [178], Hamband [155], and Quark [165], all of which support a mix of consistency guarantees on different operations, effectively inducing a partial order of data store operations.

3.4.4. State Machines for Data Processing

Returning from the distributed to the sequential setting, next we survey state-machine representations for performance-sensitive data stream processing, relevant to Chapter 8.

Deterministic and nondeterministic finite-state automata [228] are foundational in streaming as they correspond precisely to finite-memory, finite-time-per-element computations. However, they lack the ability to perform *quantitative* computations that aren't finite-state, such as e.g., simply counting the total number of input items. The simplest studied model of quantitative finite-state computation is *weighted automata* [22, 99], originally defined by Schützenberger [240]. Weighted automata extend nondeterministic finite-state automata by annotating transitions with *weights* (which are elements of an abstract semiring) and can be used for the computation of simple quantitative properties, such as counting or summing the input items. Extensions of weighted automata include *nested weighted automata* [73], which allows one level of nesting, and our related work [3], which generalizes this to arbitrary hierarchical nesting, and was a precursor to data transducers ([4] and Chapter 8). See [72, 74] for further discussion of quantitative models. Compared to data transducers, weighted automata support a limited set of operations (addition and multiplication in the semiring), rather than an arbitrary family of data types and operations on them. And while nested models recover some of this expressiveness, they can be cumbersome to work with.

A second approach to augment classical automata with quantitative features has been with the addition of *registers* that can store values from a potentially infinite set. These models are typically

varied in two aspects: by the choice of data types and operations that are allowed for register manipulation, and by the ability to perform tests on the registers for control flow. The literature on data words, data/register automata, and their associated logics [166, 209, 95, 47, 50] (and extensions such as register monitors [110]) studies words over an infinite alphabet, typically of the form $(\Sigma \times \mathbb{N})^*$, where Σ is a finite set of tags and $\mathbb{N}$ is the set of the natural numbers. They allow comparing data values for equality, and these equality tests can affect the control flow. The work on cost-register automata (CRA) [24] studies what happens when the control and data registers are kept separate by allowing write access to the registers but no testing. The register model in cost-register automata originated in streaming transducers [27, 28, 23]. As discussed in Section 8.7.1, data transducers are expressively equivalent to CRAs, but are exponentially more succinct. Expressively and in logical terms, both CRAs and data transducers recognize the class of *streamable regular transductions* [1], which can also be defined by monadic second order logic (MSO) or attribute grammars [105, 49].

A third extension of automata relevant for quantitative computation is symbolic automata [90, 264] and transducers [265, 89]; see [91] for an introduction. While symbolic automata do not allow all quantitative computations (e.g., adding up a data word of values), because transitions are predicates on input data, they can express some limited examples. One problem with weighted automata and register automata is that sometimes may lack closure under sequential composition, because sequential composition can be used to express non-regular properties (see [187], page 699); e.g., we can add up left and right parentheses to accept the Dyck language, or accept sequences of increasing numbers. Symbolic automata may be helpful to address this limitation.

3.4.5. Runtime Verification and Monitoring

Our work in Chapters 7 and 8 contributes to the large body of work on runtime verification [177, 143] (also known as *runtime monitoring*), a lightweight verification paradigm which aims to identify bugs in the output of a program as it is executed, using minimal computational resources. (The testing problem we consider is a runtime verification problem, and the data transducers work is a language that can be used for monitoring applications.) In typical work on runtime verification, the problem is to detect violations of a safety property written in a logical specification language. The specification is translated into a *monitor*, which executes along with the monitored system: it consumes system events in a streaming manner and outputs the satisfaction or falsification of the specification. *Linear Temporal Logic* (LTL) is the most widely used formalism for describing specifications for monitoring.

LTL is rather limited in expressiveness, but has been extended in various ways, including quantitative extensions. Metric Temporal Logic (MTL) has been used for monitoring real-time temporal properties [257]. Signal Temporal Logic (STL), which extends MTL with value comparisons, has been used for monitoring real-valued signals [96]. Computing statistical aggregates of LTL-defined properties, as in [111], is a limited form of quantitative monitoring. The Eagle specification language [40] can also express some quantitative monitoring properties, since it supports data-bindings. Quantitative regular expressions are suitable for quantitative and performance-sensitive monitoring [25, 187, 284, 15] and can be compiled to CRAs or data transducers. Finally, the synchronous languages [43] mentioned earlier can also be used for monitoring quantitative data streams. LOLA [88, 53] is a notable example of a synchronous language designed for runtime monitoring with close similarity to state-machine based monitors. RTLola [108] extends LOLA to the real-time monitoring setting, rather than synchronous monitoring.

In contrast to classical work in runtime verification and monitoring, our core type system in this thesis models program execution traces as partially rather than totally ordered; our ultimate goal is runtime verification abstractions which work on partially ordered streams, for which Chapter 7 is an initial proposal, handling the simplest case of equality checks between streams.

3.5. Selected Systems Challenges

On the other side of the programming model, researchers have proposed distribution, parallelization, optimization, benchmarking, and profiling strategies for stream programs. These can be relevant to the programming model as they affect the sort of semantic guarantees that the system can offer with respect to parallelism; however, the primary goal of these works is to enable efficient system implementations.

The literature in these areas is vast; we survey only a subset. We especially focus on papers relevant for *geo-distributed* performance optimization [10]: these aim to enable streaming applications over many distributed nodes that don't necessarily all reside in the same central cluster. This is especially relevant for IoT applications and edge/fog computing [236, 92, 245, 268] (see also programming models such as Mobile Fog [153]) and for applications where bandwidth is limited [270, 157, 287].

3.5.1. Performance Benchmarking and Profiling

Towards more fine-tuned optimization, researchers have proposed many techniques for analyzing the performance of existing systems. In particular, streaming benchmarks are invaluable for comparing across systems. The Yahoo Streaming Benchmark [78] is widely used, though outdated in some respects, and has motivated more modern benchmark suites [52, 183]. The DEBS Grand Challenge, an annual contest presented by the DEBS conference [206], is another useful source of more complex tasks and data. Performance profiling often centers on detecting performance bottlenecks, called stragglers [182, 168]. SnailTrail exemplifies a state-of-the-art system and technique for latency profiling and straggler detection [151].

3.5.2. Operator Placement

A fundamental problem in distribution of streaming operators, and closely related to the distribution problem considered in Chapter 6, is *operator placement* where the system determines what node to run a stream operator on. The work [63] uses constraint solving to optimize operator placement relative to network bandwidth and other constraints. Other than [63], there are several lines of work in job scheduling, operator placement, and optimization for DSPSs that try to be network-aware in some fashion. Early and influential works include [18] and [226]. The first [18] is probably the first to formalize the DSPS operator placement problem, and to explore (1) how network-awareness can lead to more efficient query evaluation, and (2) how there is a trade-off between latency and bandwidth use in this space. The second [226] presents a simple but effective heuristic algorithm which treats the geo-distributed physical nodes as a system of points in a combined latency-bandwidth space, and uses spring relaxation to find a good configuration. The work [140], similarly to [63], encodes operator placement as a mixed integer linear program. There are a number of other related papers on job scheduling [29, 278, 101, 275, 131], operator placement [51, 261, 233, 173], and resource elasticity [65, 150, 64, 93].

Researchers have also modified the programming model to allow the programmer to control operator placement. SpanEdge [234] is a primitive modification to the dataflow programming model with tasks that should be run globally versus locally. A system very related to SpanEdge is Geelytics [77]. It modifies the dataflow programming model with *scoped tasks*, which have a geographic granularity such as by site, by city, by district, or by section, and are similar to SpanEdge's local and global

tasks. Other than these, the paper [232] proposes a programming framework for stream processing in a geo-distributed (at the edge) fashion. The programming framework, however, is not based on dataflow, and is more focused on the communication mechanisms between nodes. There is also a large body of work on programming for wireless sensor networks; see the survey [202]. In general, the concerns in that domain have been more low-level, related to connections between sensors, mobility of sensors, communication from one sensor to another via short hops, and so on.

3.5.3. Stream Degradation

An even more aggressive technique for optimizing distributed execution is to degrade and approximate streams to reduce what needs to be sent over the network. The first stream processing system to incorporate general degradation of data streams was JetStream [229]. JetStream seeks to limit bandwidth use not just through degradation, but also "aggregation", which refers to a data model where data is saved and aggregated by geo-distributed nodes, and only sent when explicitly requested. AWStream [287] uses programming knobs to control the amount of degradation that occurs for video streams (e.g., frame rate reduction, resolution reduction, or some combination) to try to achieve a Pareto-optimal solution between accuracy and bandwidth use. Other systems in this space include WASP [163].

Load shedding [254, 253] can be seen as a primitive form of stream degradation. This refers to selectively dropping tuples in response to load that is beyond capacity, in order to maintain availability and good latency while hopefully not losing too much accuracy. For video data, for instance, load shedding would enable frame rate reduction but would not allow resolution reduction. It is well studied, but less flexible than what is offered by more modern systems like JetStream and AWStream.

CHAPTER 4 : A Foundation for Streams

The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.

—Edsger W. Dijkstra, 1972 [98]

This section lays the groundwork for the rest of the technical content of this dissertation: we present our core type system for distributed streams. Our stream types are an abstraction over Mazurkiewicz traces, studied in concurrency theory to model distributed sets of events [192, 97]. They are based on *synchronization schemas* [2], which in turn evolved from our earlier work on *data-trace types* [5, 9].

A stream type S describes the structure of events in the stream; semantically it will denote a collection of partially ordered traces. Formally, there are multiple ways to encode and view partially ordered traces. First, the global view as a structured batch $b : \mathsf{Batch}(S)$. This is a parsed structure: for example, if S is composed of two stream types in parallel, S_1 and S_2, then a batch of type S is a pair of a stream of S_1 and a stream of S_2. Second, S gives rise to a type for events $e : \mathsf{Event}(S)$, which are the possible individual elements in the stream. Third, S gives rise to a type for linearizations of the stream $l : \mathsf{Lin}(S)$ which are sequences of events. The type of linearizations is equipped with an *equivalence relation* which allows reordering of independent events. Finally, S gives rise to a type for labeled partially ordered sets $p : \mathsf{Poset}(S)$, where the poset is labeled with events in a way consistent with S.

Our main results include Propositions 4.4.2 and 4.4.5, which state that batches, linearizations and posets are all isomorphic for any type S. Proposition 4.4.3 characterizes the equivalence relations (on linearizations) that can arise from a stream type. Finally, Theorem 4.5.4 states that subtyping for stream types is decidable.

To connect these abstract results to real systems, we also consider a definition of *operators* (similar to the definition in Chapter 2), which are nondeterministic functions from linearizations to linearizations. We define three key properties for operators: monotonicity, type safety, and determinism. We connect these properties to the rest of the thesis in Figure 4.4. Propositions 4.6.4 and 4.7.1 state compositionality laws for these properties (under sequential and parallel composition). Lastly,

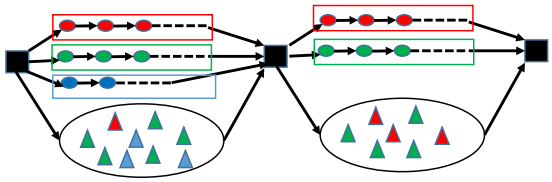


Figure 4.1: Illustrative partially ordered stream.

Theorem 4.6.2 captures why the properties matter: type-safe, deterministic operators give rise to well-defined stream transformations – well-typed functions on batches and labeled posets.

We choose to model streams as finite objects – in contrast to the traditional view as coinductive or infinite sequences. Though our results should adapt to the infinite setting, there are two reasons for this choice. First, infinite streams are not necessarily useful in practice, since real streams do eventually end – if nothing else, a DSPS will typically provide a way to take down the whole system and flush all output, e.g., via an end-of-stream punctuation event. Second, even in cases where we have in mind a truly infinite stream, relevant substreams still need to be finite. For example, in a stream of values separated by barriers, each block of values should be finite to ensure progress. The trade-off for using finite streams is that we need to recover monotonicity as a property (Section 4.6), whereas for infinite streams monotonicity would come "for free," since operators over infinite streams can only work in an item-by-item fashion.

4.1. Stream Types

Definition 4.1.1. Let T denote a base type in the following grammar. A *stream type* is a type defined syntactically by the following grammar:

$$S \quad ::= \quad \mathsf{Sync}(T, S) \mid \mathsf{Par}(S, S) \mid \mathsf{ParBy}(T, S) \mid \mathsf{Bag}(T) \mid \mathsf{Emp}$$

We also have the following abbreviation for a sequence of T:

$$\mathsf{Seq}(T) := \mathsf{Sync}(T, \mathsf{Emp}).$$

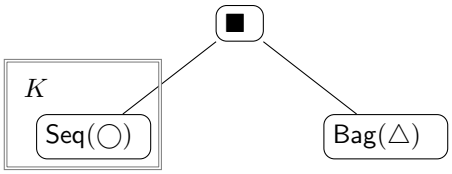


Figure 4.2: Example stream type for Figure 4.1 drawn as a tree.

The idea of this definition is to model partially ordered streams like the one visualized in Figure 4.1. This partial order consists of a sequence of black squares ■ with streams in between, which is described by the type $\mathsf{Sync}(■, S)$. The substream type S consists of circles and triangles combined in parallel: $\mathsf{Par}(S_1, S_2)$. The parallel substream of circles (type S_1) consists of a sequences of circles ◯, described by $\mathsf{ParBy}(K, \mathsf{Seq}(◯))$. The type K in the ParBy construct is a *key* field K that describes the index of the set of substreams (the key on which they are partitioned). Colors are used to indicate different values of the field K Second, the parallel substream of triangles △ is simply a bag, described by $\mathsf{Bag}(△)$. Here the colors indicate different values of the base type; note that equal values and unequal values are all parallel in a bag, unlike in ParBy.

Thus, overall, this illustration of a stream is described by the following type:

$$\mathsf{Sync}(■, \mathsf{Par}(\quad \mathsf{ParBy}(K, \mathsf{Seq}(◯)), \quad \mathsf{Bag}(△) \quad)).$$

Figure 4.2 shows this stream type visualized as a tree. Siblings correspond to the $\mathsf{Par}(S_1, S_2)$ constructor while the rectangular box, labeled with the key fields, corresponds to the $\mathsf{ParBy}(K, S)$ constructor. A parent node corresponds to the $\mathsf{Sync}(T, S)$ constructor, where T is the root and S is the forest (set of trees) of children.

This example is abstracted, but represents a practical use case such as the following example.

Example 4.1.2. Consider a stream of taxi events, where each is a GPS measurement, an indication of a taxi ride begin or a ride end, or an end-of-hour synchronization marker. GPS data for each taxi is a tuple type ◯ = `GPS(x: float, y: float, z: float)`, which indicates the type of a GPS reading using 3-dimensional coordinates. Completed ride data is a tuple type △ = `RideCompleted(rideID: int, passengerID: int, cost: int)`. Finally, end-of-hour events are used to synchronize in time; these are a tuple type ■ = `EndOfHour(date: date, hour: int)`.

The stream shown in Figure 4.1 applies to this example where squares, circles, and triangles are events of the corresponding tuple types as described above. The relationship between the different tuple types is described by the stream type shown in Figure 4.2. Described from the bottom up: first, the type $S_1 = \mathsf{ParBy}(\texttt{taxiID}, \mathsf{Seq}(\texttt{GPS}))$ denotes that `GPS` events are partitioned by the key `TaxiID` and are totally ordered for each taxi. Second, $S_2 = \mathsf{Bag}(\texttt{RideCompleted})$ denotes that `RideCompleted` events are unordered, and can be considered to be a bag. Finally, $S = \mathsf{Sync}(\texttt{EndOfHour}, \mathsf{Par}(S_1, S_2))$ denotes that `EndOfHour` events synchronize the events in S_1 and S_2, each of which can be processed in parallel as they are independent.

The partially ordered structures we have in mind (like the illustration of Figure 4.1) can be viewed in multiple ways, and we explore this formally in the next section. In particular, we will define what it means for a labeled partially ordered set to be a value of type S for a stream S in Section 4.2.4.

4.2. Views of Streams

4.2.1. Streams as Structured Batches

The following syntax defines concrete *structured* stream instances for each of the stream types, which we call *batches* because they represent data collected into a static bundle.[1] To define a concrete stream instance, one either defines a pair, a sequence, or a bag (unordered multiset).

$$B \quad ::= \quad (B, B) \mid [B, B, \ldots, B] \mid \{B, B, \ldots, B\} \mid t : T$$

Batches are typed using the following typing rules:

[1] Thanks to Joe Cutler for this suggestion.

$$\frac{b_0 : \mathsf{Batch}(S) \qquad t_i : T \text{ and } b_i : \mathsf{Batch}(S) \text{ for all } i = 1, \ldots, m}{[b_0, t_1, b_1, t_2, b_2, \ldots, t_m, b_m] : \mathsf{Batch}(\mathsf{Sync}(T, S))}\ \textsc{Synch}$$

$$\frac{b_1 : \mathsf{Batch}(S_1) \qquad b_2 : \mathsf{Batch}(S_2)}{(b_1, b_2) : \mathsf{Batch}(\mathsf{Par}(S_1, S_2))}\ \textsc{Par}$$

$$\frac{k_i : K \text{ for all } i \qquad k_i \neq k_j \text{ for all } i \neq j \qquad b_i : \mathsf{Batch}(S) \textit{ nonempty} \text{ for all } i}{\{(k_1, b_1), (k_2, b_2), \ldots, (k_n, b_n)\} : \mathsf{Batch}(\mathsf{ParBy}(K, S))}\ \textsc{ParBy}$$

$$\frac{t_i : T \text{ for all } i}{\{t_1, t_2, \ldots, t_n\} : \mathsf{Batch}(\mathsf{Bag}(T))}\ \textsc{Bag} \qquad \frac{}{[] : \mathsf{Batch}(\mathsf{Emp})}\ \textsc{Emp}$$

The *nonempty* requirement in the ParBy means that b_i must have at least one occurrence of $t : T$ for some base type T. We can define emptiness for batches inductively: (b_1, b_2) is empty iff b_1 and b_2 are empty; a list is empty iff all of its elements are empty; and a bag is empty iff all of its elements is empty. The batch $t : T$ is not empty.

In the Synch case, a batch is a list of odd length: it is formed from m synchronizing elements of type T and $m + 1$ batches of type S, for some $m \geq 0$. So the batch always contains at least one initial batch b_0, followed by zero or more alternating elements of T and batches of type S. The intuition is that a batch of type $\mathsf{Sync}(T, S)$ is like a string separated by commas (elements of T), and there is a (possibly empty) batch before the first comma and after the last comma. A possible generalization of $\mathsf{Sync}(T, S)$ would be to have $\mathsf{Sync}(S_1, S_2)$ for arbitrary stream types S_1 and S_2, but the typing rule for this gets a bit complicated, and we aren't aware of a good use case for this complexity in practice. On the other hand, it *would* be useful in practice to have a version of $\mathsf{Sync}(T, S)$ that requires the stream to be terminated by an element of T, as this avoids the off-by-one mismatch between elements of T and batches of S. But this requires a generalization of the type system, to add other constructs including singleton types; see Section 4.7.1.

4.2.2. Stream Events and Dependence Relation

Events. A stream can also be thought of as a type for individual events in isolation. Events can be either base types or pairs. Pairs are needed to encode tuples (using values of a key type K); similar to the ParBy case for batches, we just make the first element of the pair the value of a key type. Here is a grammar for events:

$$E \quad ::= \quad (E, E) \mid t : T$$

We can also talk about events specific to a particular stream type: $e : \mathsf{Event}(S)$ means that e is a valid event for a stream type S. Notice that there are no rules for $t : \mathsf{Event}(\mathsf{Emp})$ – there are no events of the empty stream type.

$$\frac{e : T}{e : \mathsf{Event}(\mathsf{Sync}(T, S))}\ \textsc{Synch-1} \qquad \frac{e : \mathsf{Event}(S)}{e : \mathsf{Event}(\mathsf{Sync}(T, S))}\ \textsc{Synch-2}$$

$$\frac{e : \mathsf{Event}(S_1)}{e : \mathsf{Event}(\mathsf{Par}(S_1, S_2))}\ \textsc{Par-1} \qquad \frac{e : \mathsf{Event}(S_2)}{e : \mathsf{Event}(\mathsf{Par}(S_1, S_2))}\ \textsc{Par-2}$$

$$\frac{k : K \qquad e : \mathsf{Event}(S)}{(k, e) : \mathsf{Event}(\mathsf{ParBy}(K, S))}\ \textsc{ParBy} \qquad \frac{e : T}{e : \mathsf{Event}(\mathsf{Bag}(T))}\ \textsc{Bag}$$

Groundedness. We say that a stream type S is *grounded* if all base types T occurring in S are nonempty and disjoint. This is useful because, for example, if we have $e : \mathsf{Event}(\mathsf{Par}(S_1, S_2))$, it allows us to deduce that either $e : \mathsf{Event}(S_1)$ or $e : \mathsf{Event}(S_2)$, but not both. Similarly, in $\mathsf{Sync}(T, S)$, it implies that elements $t : T$ are not also events of type S. We generally have grounded types in mind for the remainder of the chapter, but we will still explicitly state where this assumption is used. Groundedness was not needed for the view of streams as batches in the last section, but it is often needed for the other views.

Dependence relation. Of course, a stream type is more than just its type of events. In addition, a stream defines a *dependence relation*, a symmetric binary relation on pairs of events. The relation indicates whether the events should be considered ordered with respect to each other. For example,

in Figure 4.1, red circles are dependent with each other but independent of green circles; all circles are dependent with black squares. In the figure, the arrows denote a partial order on events, where x is less than y if there is a path left-to-right along arrows. The dependence relation gives rise to the arrows, and the partial order is the transitive closure of the arrows. One quirk to notice is that red circles on the left are ordered with green circles on the right, though they are not dependent; the ordering between them is forced by transitivity.

The dependence relation $e \mathrel{\mathcal{D}} e' \bmod S$ means that events e and e' are dependent with respect to the stream type S (the events should in particular satisfy $e, e' : \mathsf{Event}(S)$). This is defined by the following rules. Notice that there are no rules for Emp (because it has no events) nor for $\mathsf{Bag}()$ (because all events in a bag are independent).

$$\frac{t : T \qquad e : \mathsf{Event}(\mathsf{Sync}(T, S))}{e \mathrel{\mathcal{D}} t \bmod \mathsf{Sync}(T, S) \qquad t \mathrel{\mathcal{D}} e \bmod \mathsf{Sync}(T, S)}\ \textsc{Synch} \qquad \frac{e \mathrel{\mathcal{D}} e' \bmod S}{e \mathrel{\mathcal{D}} e' \bmod \mathsf{Sync}(T, S)}\ \textsc{Sub}$$

$$\frac{e \mathrel{\mathcal{D}} e' \bmod S_1}{e \mathrel{\mathcal{D}} e' \bmod \mathsf{Par}(S_1, S_2)}\ \textsc{Par-1} \qquad \frac{e \mathrel{\mathcal{D}} e' \bmod S_2}{e \mathrel{\mathcal{D}} e' \bmod \mathsf{Par}(S_1, S_2)}\ \textsc{Par-2}$$

$$\frac{k : K \qquad e \mathrel{\mathcal{D}} e' \bmod S}{(k, e) \mathrel{\mathcal{D}} (k, e') \bmod \mathsf{ParBy}(K, S)}\ \textsc{ParBy}$$

The dependence relation is symmetric, i.e., $e \mathrel{\mathcal{D}} e' \bmod S$ iff $e' \mathrel{\mathcal{D}} e \bmod S$, by an easy induction on the typing judgment. If two events $e, e' : \mathsf{Event}(S)$ are *not* dependent, we say they are independent and write $e \mathrel{\mathcal{I}} e' \bmod S$. Dependence is constructive and decidable via the above rules, so we could have alternatively given a typing judgment for $e \mathrel{\mathcal{I}} e' \bmod S$. For completeness, such a typing judgment is shown in Section 4.7.5 for grounded stream types. The following proposition is a sanity check:

Proposition 4.2.1. Let S be a grounded stream type. Then for any $e, e' : \mathsf{Event}(S)$, exactly one of $e \mathrel{\mathcal{D}} e' \bmod S$ and $e \mathrel{\mathcal{I}} e' \bmod S$ holds.

Proof sketch. By induction on S. Groundedness is used to subdivide each inductive step into cases: For example in the case $S = \mathsf{Sync}(T, S_0)$, there are three case either $e, e' : T$, $e : T$ and $e' : \mathsf{Event}(S_0)$, or $e, e' : \mathsf{Event}(S)$. Each of these three cases matches exactly one rule either for $\mathcal{D}$ or for $\mathcal{I}$. □

4.2.3. Stream Linearizations and Equivalence Relation

A *linearization* is a sequence of events:

$$L \quad ::= \quad [E, E, \ldots, E]$$

Notice that each event in the sequence may be different (they may even all have different types). Linearizations support list concatenation (denoted $\cdot$) and the interleaving relation $\mathsf{inter}(l; l_1, l_2, \ldots, l_k)$, meaning that l consists of $l_1, l_2, \ldots, l_k$ interleaved in some order.

Linearizations are typed using the rule that *a linearization has type S if all its elements are events of S*:

$$\frac{l = [e_1, e_2, \ldots, e_n] \qquad e_i : \mathsf{Event}(S) \text{ for all } i}{l : \mathsf{Lin}(S)} \textsc{Lin}$$

The dependence relation also gives rise to an equivalence relation on linearizations. This equivalence relation is derived as follows:

$$\frac{e \mathrel{\mathcal{I}} e' \bmod S}{[e, e'] \equiv [e', e] \bmod S} \textsc{Indep} \qquad \frac{l_1 \equiv l_1' \bmod S \qquad l_2 \equiv l_2' \bmod S}{l_1 \cdot l_2 \equiv l_1' \cdot l_2' \bmod S} \textsc{Concat}$$

$$\frac{l : \mathsf{Lin}(S)}{l \equiv l \bmod S} \textsc{Refl} \qquad \frac{l \equiv l' \bmod S \qquad l' \equiv l'' \bmod S}{l \equiv l'' \bmod S} \textsc{Trans}$$

We should immediately check that equivalence respects our typing judgment (i.e., equivalence is only defined between well-typed linearizations):

Proposition 4.2.2. If $l \equiv l' \bmod S$ then $l : \mathsf{Lin}(S)$ and $l' : \mathsf{Lin}(S)$.

Proof. By induction on the typing judgment for $l \equiv l' \bmod S$. The base case Refl and inductive case Trans are immediate. For the base case Indep, the independence precondition is only defined for events $e, e' : \mathsf{Event}(S)$. Finally for Concat, we observe that linearizations of type S are closed under concatenation by the definition Lin of $l : \mathsf{Lin}(S)$, since it states a condition on each element of the sequence individually. □

The other important thing to check immediately is that concatenation respects equivalence, but we don't need to prove this as it is baked into the Concat rule. The rule forces that equivalent linearizations concatenate to get equivalent linearizations.

4.2.4. Streams as Labeled Posets

For our last view of a stream, we define the concept of a *labeled poset*. This concept is traditionally called a *pomset*, for partially ordered multiset; however, this terminology can be confusing, because the ordering relation is on the underlying set, not on the elements of the multiset.[2]

A *partially ordered set* $(s, \leq)$ is a set s together with a binary relation $\leq$ on pairs of elements of s that is reflexive, transitive, and antisymmetric.

A *labeled poset* over X is a partially ordered set $(s, \leq)$ together with a labeling function $\ell : s \to X$. We denote this $(s, \leq, \ell)$. (A labeled poset over X different from a poset in X because multiple elements may be labeled with the same element of X.) Two labeled posets are *equivalent* if the underlying posets are isomorphic and the isomorphism preserves the labeling ℓ.

Now we can define labeled posets of type S. We write

$$(s, \leq, \ell) : \mathsf{Poset}(S)$$

if $\ell : s \to \mathsf{Event}(S)$ such that the order is consistent with the dependence relation in the following way:

(i) If $\ell(x) \mathcal{D} \ell(y) \bmod S$ then $x \leq y$ or $y \leq x$; and

[2] A better term might be *polset* (for partially ordered, labeled set).

(ii) No strictly smaller ordering (strict subset of $\leq$ that is still a partial order) satisfies (i).

For example, in Figure 4.1, (i) states that there must be a path between dependent events: there must be a path between every pair of red circles, or between every triangle and black square. And (ii) states that $\leq$ is minimal among orderings satisfying these conditions. For instance, there are no arrows in each of the two bags of triangles, because these are not required by condition (i).

4.3. Examples

We illustrate the different views of streams with the example of values and barriers from Section 1.2. The stream type corresponding to this example is

$$S = \mathsf{Sync}(\#, \mathsf{Bag}(\texttt{int}))$$

where # denotes a barrier and `int` denotes a value (integer). Intuitively, this creates several streams of integers in parallel, synchronized by # markers. The stream views for this example are as follows:

- A *batch* of type S in this case is an odd-length list, where every other element is a barrier event. For example:

$$[\{\}] : \mathsf{Batch}(S)$$
$$[\{1, 1, 2\}] : \mathsf{Batch}(S)$$
$$[\{1\}, \#, \{2\}] : \mathsf{Batch}(S)$$
$$[\{1\}, \#, \{1, 1, 2\}, \#, \{\}] : \mathsf{Batch}(S)$$

- An *event* of type S is either a value or a barrier. Anything is dependent with barrier, but values are not dependent with each other. For example:

$$1 : \mathsf{Event}(S)$$

$$2 : \mathsf{Event}(S)$$

$$\# : \mathsf{Event}(S)$$

$$1 \mathrel{\mathcal{D}} \# \bmod S$$

$$\# \mathrel{\mathcal{D}} \# \bmod S$$

$$1 \mathrel{\mathcal{I}} 2 \bmod S$$

  For example, the dependence $\# \mathrel{\mathcal{D}} \# \bmod S$ is derived using the SYNCH rule where we take $e = t = \#$.

- A *linearization* of type S is a sequence of values and barriers. Two linearizations are equivalent ($\equiv$) mod S if they are the same up to reordering values between adjacent barriers. For example:

$$[] : \mathsf{Lin}(S)$$

$$[1, 3, 2, 1] : \mathsf{Lin}(S)$$

$$[1, \#, 2, \#, 3] : \mathsf{Lin}(S)$$

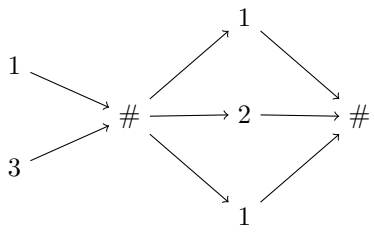
$$[1, \#, \#, \#, 2, 1] : \mathsf{Lin}(S)$$

$$[1, 2] \equiv [2, 1] \bmod S$$

$$[\#, 1, 3, 2, 1, \#] \equiv [\#, 1, 1, 2, 3, \#] \bmod S$$

  We would *not*, however, have equivalence between $[1, \#]$ and $[\#, 1]$, since 1 and # are dependent (don't commute).

- Finally, a *partial order* of type S consists of a partially ordered set labeled with values and barriers in a consistent way:



  Notice that the barrier # events are ordered with everything, and the value events are unordered as much as possible (except for their order with #). Since partial orders are transitive, this also implies some orderings on values; for example, the 1 and 3 in the first column are ordered less than the $1, 2, 1$ in the third column.

4.4. Isomorphism between Views

This section shows that all three views are isomorphic. The types are isomorphic in the sense that there is a one-to-one correspondence between their elements which also preserves the structure; in particular, we show that it preserves a concatenation operation. To define an isomorphism between batches and linearizations, we need a *flattening* relation which gives, for each batch, at least one (and possibly more than one) linearization corresponding to that batch.

4.4.1. Isomorphism between Batches and Linearizations

Definition 4.4.1 (Flattening)**.** Let S be a stream type, and let $b : \mathsf{Batch}(S)$. A *flattening* l of b is any linearization defined inductively on S as follows:

- If $S = \mathsf{Emp}$, then $b = []$ and l is a flattening of b iff $l = []$.
- If $S = \mathsf{Bag}(\mathcal{H})$, then b is a multiset, and l is a flattening of b iff the multiset of events in l equals b. (That is, l contains exactly the same events as b in some order. Assuming b has at least two distinct elements, there will be multiple such flattenings.)

- In case $S = \mathsf{Sync}(T, S')$, we have $b = [b_0, t_1, b_1, t_2, b_2, \cdots, t_m, b_m]$ for some batches $b_i : \mathsf{Batch}(S')$. Then l is a flattening of b iff $l = l_0 \cdot [t_1] \cdot l_1 \cdot [t_2] \cdot l_2 \cdot \ldots \cdot [t_m] \cdot l_m$, where l_i is a flattening of b_i for all i.

- In case $S = \mathsf{Par}(S_1, S_2)$, we have $b = (b_1, b_2)$. Then l is a flattening of b iff l is an interleaving of some l_1, l_2 where l_1 is a flattening of b_1 and l_2 is a flattening of b_2. That is, $\mathsf{inter}(l; l_1, l_2)$.

- Finally, in case $S = \mathsf{ParBy}(K, S')$, we have b is a set with finitely many entries (v_i, b_i) for $i = 1, \ldots, m$. Then l is a flattening of b iff l is an interleaving of the sequences $l_1, l_2, \ldots, l_m$ where l_i is a flattening of t_i for each i. That is, $\mathsf{inter}(l; l_1, l_2, \ldots, l_m)$.

Proposition 4.4.2. Let S be a grounded stream type.

(1) For every linearization $l : \mathsf{Lin}(S)$, there exists a *unique* batch b such that $b : \mathsf{Batch}(S)$ and l is a flattening of b. Call this batch $b = \mathsf{parse}_S(l)$.

(2) For every batch $b : \mathsf{Batch}(S)$, every flattening l of b satisfies $l : \mathsf{Lin}(S)$.

(3) For every batch $b : \mathsf{Batch}(S)$, there exists at least one flattening l of b.

(4) For two linearizations $l_1, l_2 : \mathsf{Lin}(S)$ we have $l_1 \equiv l_2 \bmod S$ iff $\mathsf{parse}_S(l_1) = \mathsf{parse}_S(l_2)$

(5) Finally, if $b : \mathsf{Batch}(S)$, then for any two flattenings l_1, l_2 of b, $l_1 \equiv l_2 \bmod S$.

Proof. Adapted from Appendix B of [2] (Proof of Proposition 14). By induction on S. For $\mathsf{Bag}(T)$, all five conditions state a standard correspondence between a multiset of items and its linearizations. For $\mathsf{Par}(S_1, S_2)$ and for $\mathsf{ParBy}(K, S_1)$, we observe that sequences over events $e : \mathsf{Event}(S)$ are interleavings of events each from a subtype, and all such interleavings are equivalent with respect to $\equiv$, by the rule Indep. Conversely $\equiv$ only holds between different interleavings of the same two or more sequences up to equivalence, i.e., for parallel composition, if $l \equiv l'$ and l is an interleaving of l_1 and l_2 and l'_1 is an interleaving of l'_2, then $l_1 \equiv l'_1$ and $l_2 \equiv l'_2$. This uses groundedness and can be proven inductively on $l \equiv l' \bmod S$.

The most interesting case is $S = \mathsf{Sync}(T, S')$. Here, we essentially apply the idea that $(a \cup b)^* = (a^* b)^* a^*$ for languages: in this context a is the type $\mathsf{Event}(S')$ and b is the type T. So a sequence of $\mathsf{Event}(S)$ decomposes into a sequence of subsequences over S' delineated by T events, where there is

one more subsequence than the number of T events. Since T events are fully dependent on everything else (rule SYNCH), this decomposition is not changed by $\equiv$, which can thus be identified with equality on the sequence of T events together with equivalence on each $\mathsf{Event}(S')$ substream. The definition of flattening reflects this decomposition exactly. □

It is worth noting that *not all* binary relations on pairs of $\mathsf{Event}(S)$ arise as a dependence relation $e \mathrel{\mathcal{D}} e' \mathsf{\ mod\ } S$. In particular, the (symmetric reflexive closure of) the relations $\{(a,b),(b,c),(c,d)\}$ and $\{(a,b),(b,c),(c,d),(d,a)\}$ do not have a hierarchical structure; basically, this is because none of a,b,c,d can act as synchronizing events for the other events as each event is only dependent with some, but not all, of the others. The following proposition characterizes exactly the dependence relations arising from stream types, based on these two examples.

Proposition 4.4.3. For any stream type S, the relation $\mathcal{D}$ ($e \mathrel{\mathcal{D}} e' \mathsf{\ mod\ } S$) is symmetric and reflexive. It additionally satisfies the following restriction: $\mathcal{D}$ does not contain the path graph $P_4 = \{(a,b),(b,c),(c,d)\}$ or the cycle graph $C_4 = \{(a,b),(b,c),(c,d),(d,a)\}$ when restricted to any four events a,b,c,d.

Proof. Symmetry and reflexivity are by construction in each type constructor for $\mathcal{D}$.

Suppose that we introduce a cycle C_4 or path P_4. It cannot have been introduced in the base case $\mathsf{Bag}(T)$, nor in parallel composition since C_4 and P_4 are connected; nor in $\mathsf{ParBy}(K,S)$ since there are no dependencies across keys. So it must have been introduced by the $\mathsf{Sync}(T,S)$ construct. But for either C_4 or P_4, there is no way to partition (cut) the vertices into those in T and those in S (with at least one vertex in each partition) such that every event in the first is dependent on every event in the second. □

4.4.2. Concatenation on Batches

Because batches are isomorphic to linearizations up to equivalence (as just shown in Proposition 4.4.2), we can define concatenation on batches. In particular, if l_1 and l_2 are linearizations of type S, their concatenation $l = l_1 \cdot l_2$ is a linearization of type S, so it has a unique parsing $\mathsf{parse}_S(l) : \mathsf{Batch}(S)$. To show that this lifts to an operation on batches $b_1 \circ b_2$, it remains to show that this is well-defined up to equivalence on linearizations:

Proposition 4.4.4. Let S be a grounded stream type. Then batch concatenation is well-defined and associative: $(b_1 \circ b_2) \circ b_3 = b_1 \circ (b_2 \circ b_3)$.

Proof. By the CONCAT rule as remarked earlier, if $l_1 \equiv l_1' \bmod S$ and $l_2 \equiv l_2' \bmod S$ then $l_1 \cdot l_2 \equiv l_1' \cdot l_2' \bmod S$. Then consider flattenings l_1 and l_2 of b_1 and b_2, respectively; by Proposition 4.4.2 (3), (5), l_1 and l_2 are unique up to equivalence; by the CONCAT rule the concatenation $l_1 \cdot l_2$ is unique up to equivalence; and by Proposition 4.4.2 (4) this means $\mathsf{parse}_S(l_1 \cdot l_2)$ is the same regardless of the choice of l_1 and l_2.

Associativity follows by associativity of concatenation on sequences, since concatenation respects equivalence. □

4.4.3. Isomorphism between Linearizations and Labeled Posets

Given any stream type S, any linearization $l : \mathsf{Lin}(S)$ gives rise to a labeled poset, which we call $\mathsf{poset}_S(l)$, as follows. If l has length n, then we let $s = \{1, 2, \ldots, n\}$, and we define the labeling function $\ell(i) = l[i]$. Then we define the ordering as follows: $i <_{l,S} j$ iff there is some sequence of indices

$$i = i_0 < i_1 < \ldots < i_m = j$$

such that each pair adjacent in the sequence is dependent:

$$\begin{aligned} &l[i_0] \mathrel{\mathcal{D}} l[i_1] \bmod S \\ &l[i_1] \mathrel{\mathcal{D}} l[i_2] \bmod S \\ &\ldots \\ &l[i_{m-1}] \mathrel{\mathcal{D}} l[i_m] \bmod S. \end{aligned}$$

This definition captures the property that $l[i]$ and $l[j]$ are ordered through some transitive closure of dependencies between events in the linearization. And when this holds, it is impossible to reorder the events ($\equiv$) to get $l[j]$ before $l[i]$. The following proposition justifies this formally:

Proposition 4.4.5. Let S be a grounded stream type.

(1) Let $l_1 : \mathsf{Lin}(S)$ and $l_2 : \mathsf{Lin}(S)$ be two linearizations. Then $l_1 \equiv l_2 \ \mathsf{mod}\ S$ iff $\mathsf{poset}_{S_1}(l_1)$ and $\mathsf{poset}_{S_2}(l_2)$ are isomorphic.

(2) Let $l : \mathsf{Lin}(S)$ be a linearization; then $\mathsf{poset}_S(l) : \mathsf{Poset}(S)$.

(3) Let $p : \mathsf{Poset}(S)$; then there exists a linearization l such that p is isomorphic to $\mathsf{poset}_S(l)$.

Proof. For (1) in the forward direction, what we need to show is that the rules for $\equiv$ preserve poset isomorphism. For the rule INDEP the isomorphism switches e and e' in the linearization; this preserves the order because e and e' are not ordered under $<_{l,S}$. REFL and TRANS hold because poset isomorphism is transitive. For CONCAT, we construct an isomorphism between the posets on $l_1 \cdot l_2$ and $l_1' \cdot l_2'$ by combining the isomorphisms on the first half and second half. The idea is then that any ordering in $<_{l_1 \cdot l_2, S}$ defined by $i = i_0 < i_1 < \ldots < i_m = j$ is derived from a first half of orderings in l_1, followed by a second half of orderings in l_2, which is isomorphic to a segment of orderings in l_1' followed by a segment of orderings in l_2', which implies ordering in $<_{l_1' \cdot l_2', S}$. The reverse direction requires decomposing any isomorphism by induction into a matching between the elements of the individual linearizations such that the matching can be formed by repeated consecutive swaps, implying equivalence via INDEP, CONCAT, and TRANS; this is the same as for data-trace types and a full proof can be found in [9] and [5].

For (2), the crux of the statement is that our definition of $<_{l,S}$ precisely captures the smallest ordering consistent with dependence $\mathcal{D}$ as described in the definition of a poset. Since $l[i]\ \mathcal{D}\ l[j]\ \mathsf{mod}\ S$ implies that $i <_{l,S} j$ or vice versa, $<_{l,S}$ is an ordering consistent with $\mathcal{D}$, it remains to see why there is no weaker ordering. But note that any weaker ordering would have to remove $i <_{l,S} j$ for some $i < j$ as these are the atomic constraints generating the ordering, and then it would not be consistent with $\mathcal{D}$.

For (3), for an arbitrary finite labeled poset p of size n, pick any topological ordering of its elements $1, 2, 3, \ldots, n$, and consider the linearization $l : \mathsf{Lin}(S)$ defined by elements $1, 2, 3, \ldots, n$ in that order. The ordering arising from p must contain all pairs $i <_{l,S} j$ where $l[i]\ \mathcal{D}\ l[j]\ \mathsf{mod}\ S$ because otherwise i and j would be independent, violating consistency with $\mathcal{D}$; and it must be the smallest ordering containing these pairs by minimality of p. □

4.5. Subtyping

Types are most useful when they can be algorithmically checked. In this section, we define a form of semantic subtyping: S_1 is a subtype of S_2, denoted $S_1 \lesssim S_2$, means that a stream of type S_1 can be interpreted as a stream of type S_2 if the partial order is relaxed. For example, a sequence can be interpreted as a bag, but not vice versa. Also, the type S_2 may have additional possible events; for example, S_1 should be a subtype of $\mathsf{Par}(S_1, S_2)$. The following picture captures the notion we want to define:

$$\begin{array}{ccc} \mathsf{Seq}(\mathtt{int}) & \lesssim & \mathsf{Bag}(\mathtt{int}) \\ \wr\!\wedge & & \wr\!\wedge \\ \mathsf{Sync}(\#, \mathsf{Seq}(\mathtt{int})) & \lesssim & \mathsf{Sync}(\#, \mathsf{Bag}(\mathtt{int})) \end{array}$$

An embedding that witnesses the subtyping from $\mathsf{Seq}(\mathtt{int})$ to $\mathsf{Sync}(\#, \mathsf{Seq}(\mathtt{int}))$ is the function which returns the same stream, without adding any # events. Going the other way, from $\mathsf{Sync}(\#, \mathsf{Seq}(\mathtt{int}))$ to $\mathsf{Seq}(\mathtt{int})$, is also possible – by throwing away all # events in the stream – but we don't allow this direction of subtyping, as having subtyping in both directions would be semantics-breaking. So we adopt the position that subtyping should only extend the set of possible events, not restrict it.

The following are adapted from Section 2.5 and Appendix B of [2]. The definition uses the following subtyping relation on events: $\mathsf{Event}(S_1)$ is a subtype of $\mathsf{Event}(S_2)$ if for all $e : \mathsf{Event}(S_1)$, $e : \mathsf{Event}(S_2)$.

Definition 4.5.1 (Subtyping)**.** For stream types S_1 and S_2, S_1 is a *subtype* of S_2, written $S_1 \lesssim S_2$, if:

(i) For every base type T occurring in S_1, T occurs in S_2;

(ii) For all events e, if $e : \mathsf{Event}(S_1)$, then $e : \mathsf{Event}(S_2)$; and

(iii) For all tuples $x, y : \mathsf{Event}(S_1)$, if $x \mathcal{D} y \bmod S_2$ then $x \mathcal{D} y \bmod S_1$.

If $S_1 \lesssim S_2$, we also say that S_2 is a *relaxation* of S_1.

Two stream types S_1 and S_2 are *order-equivalent*, denoted $S_1 \sim S_2$, if both $S_1 \lesssim S_2$ and $S_2 \lesssim S_1$.

Proposition 4.5.2. Let S, S' be stream types such that $S \lesssim S'$. Then: (1) For all l, if $l : \mathsf{Lin}(S)$ then $l : \mathsf{Lin}(S')$. (2) For all l_1, l_2, if $l_1 \equiv l_2 \bmod S$ then $l_1 \equiv l_2 \bmod S'$.

Proof. Statement (1) is direct from the fact that $\mathsf{Event}(S)$ is a subtype of $\mathsf{Event}(S')$ and linearizations are sequences of events. For (2), all typing rules for $l_1 \equiv l_2 \bmod S'$ are the same as for $l_1 \equiv l_2 \bmod S$ except INDEP, which introduces strictly more equivalences by condition (iii) of subtyping. □

This brings us to the subtyping problem for stream types. The problem is to determine, on input two grounded stream types S_1 and S_2, whether $S_1 \lesssim S_2$.

$$\textsc{Subtyping} = \{\langle S_1, S_2\rangle \ : \ S_1, S_2 \text{ are grounded stream types and } S_1 \lesssim S_2.\}$$

To decide the above problem, technically we also need to check whether S_1 and S_2 are grounded. To do this, we need to assume that deciding whether two base types are disjoint is decidable. (This would be true, for example, if base types consist of things like `int`, `bool`, and `float`, or if they are named records). We first state the groundedness result, then we get to the main theorem, Theorem 4.5.4. Define

$$\textsc{Groundedness} = \{\langle S\rangle \ : \ S \text{ is a grounded stream type.}\}$$

Proposition 4.5.3. Assume that checking whether two base types are disjoint is decidable in $O(1)$ time. Then GROUNDEDNESS is decidable in quadratic time.

Proof. Enumerate the base types in S: these are instances of T for $\mathsf{Sync}(T, S)$ or instances of K for $\mathsf{ParBy}(K, S)$. There are linearly many. For each type T, check whether it empty by checking is disjoint from itself. For each pair of base types, check whether they are disjoint. □

Theorem 4.5.4. Assume that groundedness is decidable in $T(n)$ time. Then SUBTYPING is decidable in quadratic time plus $2T(n)$.

Proof. On input S_1 and S_2, first check if they are grounded; if not, reject. Then write $\mathsf{Event}(S_1)$ as a disjoint union of finitely many tuple types; each member of the union is a nested tuple of key fields, followed by a payload, e.g.:

$$(K_1, (K_2, (K_3, T))).$$

(Here, T occurs in some $\mathsf{Sync}(T, S)$ subterm, and K_1, K_2, and K_3 are all partition-by keys in the surrounding context for T). We check conditions (i) and (ii) for each part of the union separately. For (i), we know K_1, K_2, K_3, and T should occur somewhere in S_2; if not, we reject. For (ii), we write S_2 as a disjoint union similarly to S_1. If K_1, K_2, K_3, and T do not occur in the exactly the same form, then by groundedness of S_2, elements of the type $(K_1, (K_2, (K_3, T)))$ cannot be elements of S_2. Additionally there is at least one such element, since groundedness also requires base types to be nonempty. So we can reject.

At this point, we have S_1 and S_2 as disjoint unions where each case in the union for S_1 is in one-to-one correspondence with some case in the union for S_2. We can ignore cases in the union for S_2 that are not in S_1. Let $C_1, C_2, \ldots, C_m$ be the shared types so that $\mathsf{Event}(S_1)$ is the disjoint union of $C_1, \ldots, C_m$. Then for each C_i, C_j, we build a formula $\phi^1_{C_i,C_j}$ whose free variables are elements of C_i and elements of C_j or fields of C_i and fields of C_j in case they are tuple types, which is true exactly when $x \mathrel{\mathcal{D}} y \bmod S_1$ for $x : C_i$ and $y : C_j$. This is done by expanding out the cases for the dependence relation D. In all cases, the formula built is either true or false, or derived from a subcase, except in the $\mathsf{ParBy}(K, S)$ case where we get an equality constraint as an atomic formula: specifically, for $(k, e) \mathrel{\mathcal{D}} (k', e') \bmod \mathsf{ParBy}(K, S)$ we get the constraint $k = k'$ (in conjunction with the recursive formula for $e \mathrel{\mathcal{D}} S \bmod S'$). Altogether, each $\phi^1_{C_i,C_j}$ is just an atomic formula over the language of equality. We do the same for S_2 to get formulas $\phi^2_{C_i,C_j}$.

Condition (iii) now becomes checking a set of implications of atomic formulas over the language of equality, which is decidable by checking each implication $\phi^2_{C_i,C_j} \to \phi^1_{C_i,C_j}$ in turn. The complexity is quadratic because there are quadratically many pairs i, j. □

As a corollary of Theorem 4.5.4, checking order-equivalence is decidable in quadratic time as it is sufficient to check subtyping in both directions.

4.6. Monotonicity, Type Safety, and Determinism

In this section we define three key properties that are relevant to the rest of the thesis: monotonicity, type safety, and determinism for operators over streams. First, we need an abstract model of an operator in a stream processing system.

Definition 4.6.1. An *operator* from type X to type Y is a relation $F \subseteq \mathsf{List}(X) \times \mathsf{List}(Y)$ such that each input is associated with at least one output: for every $l : \mathsf{List}(X)$, there exists $l' : \mathsf{List}(Y)$ such that $(l, l') \in F$.

Modeling the operator as a relation is necessary because it allows for nondeterministic behavior; that is, this is a model of a concurrent system with multiple possible execution traces. The definition only requires that there is at least one possible behavior on any given input stream.

A bit of foreshadowing: though they aren't directly expressed as such, operators are basically the key objects studied in Chapters 6 to 8. Chapter 5 is not quite eligible for this distinction as it only defines functions on batches, but these can also be seen as operators by applying our isomorphism from Section 4.4; see Theorem 4.6.3.

For the following properties, let S be an input stream type, S' an output stream type, and F an operator from $\mathsf{Event}(S)$ to $\mathsf{Event}(S')$.

- **Monotonicity:** F is *monotone* if, for all $(l, l') \in F$, if l is a prefix of m, then there exists m' such that l' is a prefix of m' and $(m, m') \in F$.
- **Type safety:** F is *type-safe* if, for all $(l_1, l_1') \in F$ and $l_1 \equiv l_2 \bmod S$, there exists l_2' such that $l_1' \equiv l_2' \bmod S'$ and $(l_2, l_2') \in F$.
- **Determinism:** F is *deterministic up to output equivalence*, or simply *deterministic*, if whenever $(l, l_1') \in F$ and $(l, l_2') \in F$, $l_1' \equiv l_2' \bmod S'$.

4.6.1. Examples

To illustrate monotonicity, consider the function Sum (encoded as an operator as a set of ordered pairs), which takes in a sequence or bag of integers and outputs their sum. The input stream type is $\mathsf{Seq}(\texttt{int})$ or $\mathsf{Bag}(\texttt{int})$ (either works) and the output type is $\mathsf{Seq}(\texttt{int})$ (though there is only be one output, so it is really a singleton). Sum is not monotonic because on input $[1, 2, 3]$ it produces $[6]$ and on input $[1, 2, 3, 1]$ it produces $[7]$, but $[6]$ is not a prefix of $[7]$. To make it monotonic, we have to add a special symbol to trigger output: the correct function to consider is the one which maps $[1, 2, 3]$ to $[]$, but maps $[1, 2, 3, \#]$ to 6. Specifically, we define SumAt which maps maps $[a_1, a_2, a_3, \ldots, a_k, \#] + t$ to $[a_1 + a_2 + \cdots + a_k]$ for any integers $a_1, \ldots, a_k$ and any tail list t, but maps input lists with no

Name	Example I/O	Input type S	Output type S'	Monotone?	Type-safe?	Deterministic?
Sum	$[1,2,3] \mapsto [6]$	Seq(int) or Bag(int)	Seq(int)	No	Yes	Yes
SumAt	$[1,2,3] \mapsto []$ $[1,2,3,\#] \mapsto [6]$	Sync(#, Bag(int))	Seq(int)	Yes	Yes	Yes
Iden	$[1,2,3] \mapsto [1,2,3]$	Seq(int)	Seq(int) or Bag(int)	Yes	Yes	Yes
		Bag(int)	Seq(int)	Yes	No	Yes
		Bag(int)	Bag(int)	Yes	Yes	Yes
Reorder	$[1,2,3] \mapsto [3,1,2]$ $[1,2,3] \mapsto [1,2,3]$	Seq(int) or Bag(int)	Seq(int)	Yes	Yes	No
		Seq(int) or Bag(int)	Bag(int)	Yes	Yes	Yes
First	$[1,2,3] \mapsto [1]$ $[] \mapsto []$	Seq(int)	Seq(int) or Bag(int)	Yes	Yes	Yes
		Bag(int)	Seq(int) or Bag(int)	Yes	No	Yes
Last	$[1,2,3] \mapsto [3]$ $[] \mapsto []$	Seq(int)	Seq(int) or Bag(int)	No	Yes	Yes
		Bag(int)	Seq(int) or Bag(int)	No	No	Yes
Drop	$[1,2,3] \mapsto [1,3]$ $[1,2,3] \mapsto [2]$	Seq(int)	Seq(int) or Bag(int)	Yes	Yes	No
		Bag(int)	Seq(int)	Yes	No	No
		Bag(int)	Bag(int)	Yes	Yes	No
Reverse	$[1,2,3] \mapsto [3,2,1]$ $[1] \mapsto [1]$	Seq(int)	Seq(int) or Bag(int)	No	Yes	Yes
		Bag(int)	Seq(int)	No	No	Yes
		Bag(int)	Bag(int)	No	Yes	Yes
Swap	$[1,2,3] \mapsto [1,3,2]$ $[1,2,3] \mapsto [2,1,3]$	Seq(int)	Seq(int)	Yes	Yes	No
		Seq(int) or Bag(int)	Bag(int)	Yes	Yes	Yes
		Bag(int)	Seq(int)	Yes	No	No
Star	$[1,2] \mapsto [1,2,1,2]$ $[1,2] \mapsto [1,2]$	Seq(int)	Seq(int) or Bag(int)	No	Yes	No
		Bag(int)	Seq(int)	No	No	No
		Bag(int)	Bag(int)	No	Yes	No

Figure 4.3: Examples of monotonicity, type safety, and determinism for a selection of stream operators and input and output types.

instance of # to []. (We could also choose to trigger output on every #, not just the first one. Either way, this function is monotonic.) The function SumAt is best described using the input type $\mathsf{Sync}(\#, \mathsf{Bag}(\mathtt{int}))$ (a sequence of bags separated by #). Its output type is $\mathsf{Seq}(\mathtt{int})$.

To illustrate type safety, consider the function Iden, which consists of all pairs (l, l), e.g., it maps $[1, 2, 3]$ to $[1, 2, 3]$. Suppose $S = \mathsf{Bag}(\mathtt{int})$ so that it allows input reorderings, that is $[1, 2, 3] \equiv [1, 3, 2] \bmod S$. Then type safety says that there should be an output on $[1, 3, 2]$ that is equivalent to $[1, 2, 3]$ under S'. This holds iff $[1, 2, 3] \equiv [1, 3, 2] \bmod S'$. So this identity operator is type-safe iff S' allows output reorderings. In particular, it is type-safe for $S' = \mathsf{Bag}(\mathtt{int})$ but *not* for $S' = \mathsf{Seq}(\mathtt{int})$. If instead $S = \mathsf{Seq}(\mathtt{int})$, it is type-safe either way.

To illustrate determinism, it is immediate that Sum and Iden are deterministic because they are functions. But for a less trivial example, consider the relation Reorder, which maps a list l to all its possible reorderings: in particular, for $l = [1, 2, 3]$, Reorder contains all six pairs

$$(l, l), (l, [1, 3, 2]), (l, [2, 1, 3]), (l, [2, 3, 1]), (l, [3, 1, 2]), (l, [3, 2, 1]).$$

Then Reorder is deterministic iff $l_1 \equiv l_2 \bmod S'$ for all reorderings (permutations) l_1 and l_2. Concretely, it is deterministic for output type $S' = \mathsf{Bag}(\mathtt{int})$ but not for output type $S' = \mathsf{Seq}(\mathtt{int})$. In short, determinism allows for multiple outputs as long as those outputs are equivalent.

It is possible for an operator to be type-safe, but not deterministic. For example, if $S' = \mathsf{Seq}(\mathtt{int})$, then Reorder is not deterministic, but it *is* still type-safe because while there are many (inequivalent) possible outputs for each input, this set of possibilities doesn't depend on the input.

Conversely, it is possible for an operator to be deterministic, but not type-safe. For example, consider the first-element function First that on $[1, 2, 3]$ returns $[1]$ and on $[2, 1, 3]$ returns $[2]$. This is deterministic because it is a function, whether the input type S is $\mathsf{Seq}(\mathtt{int})$ or $\mathsf{Bag}(\mathtt{int})$. But if we have $S = \mathsf{Bag}(\mathtt{int})$, so that $[1, 2, 3] \equiv [2, 1, 3] \bmod S$, then it is not type-safe, because $[1]$ and $[2]$ are not equivalent outputs (not even as bags). The function Last, returning the last element instead of the first, has the same properties except that it is also not monotone.

Finally, it is possible to be neither deterministic nor type-safe. An interesting example is Drop, which takes the input sequence l and nondeterministically drops some number of items. Formally,

$(l, l') \in$ DROP if l' is any subsequence of l. For the types $S = \mathsf{Bag}(\mathtt{int})$ and $S' = \mathsf{Seq}(\mathtt{int})$, DROP is not type-safe. This is because, for example, $[1, 2, 3]$ may produce output $[1, 3]$, but it is equivalent as a bag to $[3, 2, 1]$ which has no corresponding equivalent output. DROP is also not deterministic for any S and S'.

All of these examples, and a few extras, are summarized in Figure 4.3. In the table, we write $l \mapsto l'$ for $(l, l') \in F$. The operation REVERSE reverses the input list. Notice that it is not monotone, even in the bag-to-bag case; in this case it is essentially the identity on bags, but our monotonicity requirement is stronger and requires that the sequential output be monotone, not just the bag output. The operation SWAP is like REORDER in that it nondeterministically disrupts the input, but only minimally so: it may make *at most one swap* of two adjacent elements. Unlike REORDER, it is not type-safe as a bag-to-sequence operation because the output behavior depends on the input reordering. It is monotone because once a swap point is chosen, the input can always be extended with more items.

The last item in the table, STAR, is the simplest example we can think of that is neither monotone nor deterministic, nor necessarily type-safe. It takes the input list and repeats it some nonzero number of times (like Kleene star). As with many other operations in the table, it fails to be type-safe in the sequence-to-bag case, but is type-safe in the others. With STAR included, note that all 8 possible choices for the 3 properties are possible, i.e., the 3 properties are fully independent. We include a few more examples, omitted from this section, in Section 4.7.3.

4.6.2. Notes and Discussion

Why do we call the second property "type safety"? By analogy, if one defines a function on an input type of integers modulo 5, the output should not depend on the exact choice of residue class; $f(4)$ and $f(9)$ should be the same (up to output equivalence). Or to take another example, if Booleans are represented as integers where False is 0 and True is any nonzero integer, Boolean operations are only type-safe if they are not defined with respect to the particular choice of representation of True. In our case, S and S' define types with a custom equality relation $\equiv$ on linearizations, and the property encodes the fact that the equalities in the input transfer over to the output. This captures exactly the property necessary for F to be interpretable as a relation between input batches and output batches, or between input labeled posets and output labeled posets – see the following subsection.

A second reason is that it corresponds precisely to type safety on batch types, as we will see in the next subsection. For example, batches of the stream type $\mathsf{Par}(S_1, S_2)$ are ordered pairs of a batch from S_1 and a batch from S_2. So a function defined on sequences – interleavings of arbitrary events from S_1 and S_2 – is *not necessarily a well typed function on ordered pairs.* From this perspective, the input to an operator – a list of events – contains more information than just an element of a type; it also includes an arbitrary representation of that element, which an operator should really be oblivious to.

It is important to us that determinism is strictly weaker than saying the relation is functional, because we want to allow implementations which benefit from parallelism. That is, we do *not* require that for any l there is exactly one l' such that $(l, l') \in F$. Instead, the output l' only must be unique up to output equivalence: there can be multiple (l, l'_1) and (l, l'_2), but only if l'_1 and l'_2 are equivalent.

Alternate definitions of monotonicity are possible. The REVERSE and SWAP examples reveal this. Considering REVERSE, it would be reasonable to define monotonicity only up to reordering, meaning that REVERSE would be monotonic as a bag-to-bag operator. To do this, in the definition of monotonicity, we would replace sequence prefix (l is a prefix of m) with batch prefix: $\mathsf{parse}_S(l)$ is a prefix of $\mathsf{parse}_S(m)$ (under batch concatenation). For SWAP, note that it is monotonic, but displays the following odd behavior: $[1, 2, 3] \mapsto [1, 3, 2]$ is possible, but $[1, 2]$ does not map to any of the prefixes of $[1, 3, 2]$. So this behavior on input $[1, 2, 3]$ appears "out of thin air" and not in an incremental way. Put another way, if we imagine a black-box system implementing SWAP and give it elements only one at a time, it will never truly display swapping behavior. Perhaps a better definition of SWAP is that it can either swap two elements, or drop the last element: that way $[1, 2] \mapsto [1]$ is possible, removing the "out of thin air" new behavior after feeding in the third input item. However, these issues come up only in considering nondeterministic functions, and do not really arise in the deterministic case.

4.6.3. Functional Characterization

If F is both type-safe and deterministic, then the following theorem says it is a *well-defined function* from input streams up to equivalence to output streams up to equivalence. Equivalently, it defines a function from input batches to output batches and a function from input labeled posets to output labeled posets.

Theorem 4.6.2. Let $F \subseteq \mathsf{List}(\mathsf{Event}(S)) \times \mathsf{List}(\mathsf{Event}(S'))$ be type-safe and deterministic. Then there exist functions $f : \mathsf{Batch}(S) \to \mathsf{Batch}(S')$ and $g : \mathsf{Poset}(S) \to \mathsf{Poset}(S')$ such that

(1) $(l, l') \in F$ implies $f(\mathsf{parse}_S(l)) = \mathsf{parse}_{S'}(l')$.

(2) $(l, l') \in F$ implies $g(\mathsf{poset}_S(l)) = \mathsf{poset}_{S'}(l')$.

Proof. (1) is an application of the isomorphism between batches and equivalence classes of linearizations (Proposition 4.4.2) and (2) is an application of the isomorphism between linearizations and posets (Proposition 4.4.5). For (1), to define $f(b)$ we consider any flattening l of b; by definition of an operator F there exists some $(l, l') \in F$, and we let $f(b) = \mathsf{parse}_{S'}(l')$. By determinism, this definition does not depend on the choice of l'; by type safety it does not depend on the choice of flattening l. The argument for (2) is identical except for $g(p)$ we define l to be any total extension of p, and we let $g(p) = \mathsf{poset}_{S'}(l')$. □

A technical note: unfortunately (1) and (2) in Theorem 4.6.2 above are only forward implications. The reverse implication might not hold if F isn't closed under equivalence. For example, if F is the identity function (the set of ordered pairs (l, l)) where $S = S' = \mathsf{Bag}(T)$. Then F is type-safe and deterministic and the f and g arising in the theorem are the identity functions on batches and posets, respectively. But the implication $(l, l') \in F$ implies $f(\mathsf{parse}_S(l)) = \mathsf{parse}_{S'}(l')$ only goes one way; equality of sequences implies equivalence, but not vice versa. We allow this discrepancy because we want the identity function F to be well-typed for *any* S and S'.

The converse of Theorem 4.6.2 is also true: any well-defined function on batches or labeled posets is, when interpreted as a function on linearizations, both type-safe and deterministic. We state the result for batches; the result for posets is analogous.

Theorem 4.6.3. Let $f : \mathsf{Batch}(S) \to \mathsf{Batch}(S')$ be any function, and define F to be the set of pairs (l, l') such that $f(\mathsf{parse}_S(l)) = \mathsf{parse}_{S'}(l')$. Then F is a type-safe, deterministic operator.

Proof. Another application of Proposition 4.4.2. For type safety, if $l_1 \equiv l_2 \bmod S$ then $\mathsf{parse}_S(l_1) = \mathsf{parse}_S(l_2)$; therefore, $(l_1, l') \in F$ iff $(l_2, l') \in F$ for any l_1, l_2. For determinism, if $(l, l'_1) \in F$ and $(l, l'_2) \in F$ then $\mathsf{parse}_{S'}(l'_1) = \mathsf{parse}_{S'}(l'_2)$ (since f is a function) hence $l'_1 \equiv l'_2 \bmod S'$. □

The above consequences only mention type safety and determinism. The consequence of monotonicity is that the operator not only implements a well-defined function from input batches or posets to output batches or posets, but also is *streamable* in the sense that the output can be produced incrementally. We discuss incremental operators a bit more in Section 4.7.6.

4.6.4. Compositionality Laws

In addition to the functional characterization, one way to investigate the usefulness of the three properties is to consider how they behave when functions are composed. To cut to the chase: monotonicity and type safety are compositional. Determinism + type safety together are compositional, but determinism alone is not. This limitation of determinism as a property is because of the relaxed requirement on determinism in the output, where it only need to be the same up to equivalence, yet there is no reference to equivalence in the input. So it is arguable that *determinism + type safety* is a more important property than plain determinism. However, defining determinism on its own allows us to explore the rich space of different possibilities for example operators as in Figure 4.3, where some operators are deterministic, but not type-safe.

For operators F and F', define $F \cdot F'$ to be relational composition:

$$F \cdot F' := \big\{(l, l'') \mid \exists l' : (l, l') \in F \text{ and } (l', l'') \in F'\big\}.$$

The composition of operators is an operator because it is a total relation (for every input there is at least one output). For the following theorem, we say that l' in the definition above is the *intermediate witness* for $(l, l'') \in F$.

Proposition 4.6.4. Let S, S', S'' be stream types, F an operator from S to S', and F' an operator from S' to S''.

1. If F is monotone and F' is monotone, then $F \cdot F'$ is monotone.
2. If F is type-safe and F' is type-safe, then $F \cdot F'$ is type-safe.
3. If F is deterministic and F' is deterministic *and type-safe*, then $F \cdot F'$ is deterministic.

Proof. For (1), let $(l, l'') \in F \cdot F'$ and let l' be the intermediate witness to the composition. Suppose m extends l (l is a prefix of m). By monotonicity, we can find $(m, m') \in F$ such that m' extends l'. Then in turn, we can find $(m', m'') \in F'$ such that m'' extends l''.

For (2), let $(l_1, l_1'') \in F \cdot F'$ with intermediate witness l_1'. Now suppose $l_1 \equiv l_2 \bmod S$. By type safety, we can find l_2' with $l_1' \equiv l_2' \bmod S'$. By type safety again, we can find l_2'' with $l_1'' \equiv l_2'' \bmod S''$.

For (3), to show determinism let $(l, l_1'') \in F \cdot F'$ with intermediate witness l_1'. Now suppose that there is l_2'' such that $(l, l_2'') \in F \cdot F'$, and let the intermediate witness for this be l_2'. Here is where we need to apply type safety: determinism of F gives us that $l_1' \equiv l_2' \bmod S'$, but not their equality. Type safety of F' pushes this through to an equivalence in S'': applied to (l_1', l_1''), it gives us l_3'' such that $(l_2', l_3'') \in F'$ and $l_1'' \equiv l_3'' \bmod S''$. Now we have both $(l_2', l_2'') \in F'$ and $(l_2', l_3'') \in F'$ so we can apply determinism of F' to get $l_2'' \equiv l_3'' \bmod S''$. Applying transitivity of equivalence, we get $l_1'' \equiv l_2'' \bmod S''$ as required. □

Additional compositionality laws can be given for other kinds of composition, in particular parallel composition; see Section 4.7.2.

4.6.5. In the Context of the Remaining Chapters

Figure 4.4 shows how the properties monotonicity, type safety, and determinism relate to the remaining chapters of the thesis. Each chapter can be thought of as considering a particular definition of operators F, which is either a domain-specific language (DSL) or a black-box program in the case of Chapter 7. (Note that the chapters do not define an operator by Definition 4.6.1 directly, so the table should be understood as an informal picture, to the best of our understanding, of which formal properties would hold *if* the chapter were to give an instance of Definition 4.6.1. There are likely to be minor discrepancies if this were fleshed out formally.)

Among the different DSLs, Chapter 6 is relatively expressive as a language (in particular, it allows arbitrary sequential programs), whereas the others are relatively restricted. If we think of an operator as having an input type S and output type S', some chapters only consider restricted classes of output types S'. Monotonicity holds for every chapter. Type safety and determinism are enforced statically for Chapters 5 and 6 and checked dynamically for a specific concurrent execution for Chapter 7. Chapter 8 is a deterministic model but does not consider out-of-order input streams so does not

Chapter	Operator DSL (F)	Output (S')	Monotonicity?	Type safety?	Determinism?
5	Restricted (SPST)	Case-specific	Yes Thm. 5.5.1	Yes (By construction)	Yes (By construction)
6	General (DGS program)	$\mathsf{Bag}(T)$	Yes	Yes (If program is consistent) Thms. 6.4.4, 6.5.5	Yes (If program is consistent) Thms. 6.4.4, 6.5.5
7	General (Black-box program)	Any	Yes	Partial (For a specific runtime trace) Thm. 7.6.8	Partial (For a specific runtime trace) Thm. 7.6.8
8	Restricted (DT or QRE-Past)	$\mathsf{Seq}(T)$	Yes Thm. 8.3.1	—	Yes

Figure 4.4: Monotonicity, type safety, and determinism for operators defined in the remaining chapters. Here, F is an operator from S to S' defined in a domain-specific language (DSL) specific to each chapter (or any black-box program for Chapter 7). The entry — indicates that the property is not studied for the DSL in question.

consider type safety, but the necessary requirements to ensure type safety could be considered in future work.

4.7. Outtakes

4.7.1. Singleton Types

I originally wanted to include a construct for a singleton stream $\mathsf{Single}(T)$, denoting a stream with one element of type T. The problem with such a construct is it breaks the isomorphism described in Section 4.4 because singleton streams are *not closed under concatenation*. In particular this means that a linearization of type S isn't just a sequence over $\mathsf{Event}(S)$, but rather a sequence with some additional constraints. I would like to give a treatment of this kind of stream type in future work, since these constraints are useful in practice. For example, when processing the output of an aggregation operator it is often useful to know that the result has only one element. Writing

further operations on the result of an aggregation, such as dividing a sum and a total count to get an average (`let avg = sum / count`), is awkward when the arguments are streams and not known to be singletons.

Singletons can also be used to define relations that are set types, rather than bag types. The following abbreviation accomplishes this:

$$\mathsf{Set}(T) := \mathsf{ParBy}(T, \mathsf{Single}(\bullet))$$

where $\bullet$ denotes the unit type. Currently, we use types $\mathsf{Bag}(T)$ because they are, like all other constructs, closed under concatenation.

Singleton types would be typed using rules like the following:

$$\frac{t : T}{t : \mathsf{Batch}(\mathsf{Single}(T)) \qquad t : \mathsf{Event}(\mathsf{Single}(T)) \qquad [t] : \mathsf{Lin}(\mathsf{Single}(T))}\ \textsc{Singleton}$$

This would require a change to to $l : \mathsf{Lin}(S)$ to make typing judgments specific to each stream construct. It is not immediately clear whether we should have $t \mathcal{D} t \bmod \mathsf{Single}(T)$ or not; this choice is likely immaterial because a singleton stream only ever has one item, never two that could be dependent or independent.

Singleton stream types may require some dynamic enforcement. In particular, if an input stream to the system is typed as a singleton, this has to be checked at runtime.

4.7.2. Additional Compositionality Laws

In Section 4.6.4, we considered only compositionality under relational composition, a.k.a. sequential composition. This is enough to get the picture that monotonicity and type safety are generally compositional, while determinism is only compositional with an additional type safety assumption. It is possible to investigate other compositionality laws too, which are interesting in their own right.

The most immediate of these is the parallel, rather than sequential, composition of two operators. Let F_1 be an operator from S_1 to S_1', and F_2 an operator from S_2 to S_2'. Then we can define the

parallel composition $F_1 \| F_2$ as an operator from $\mathsf{Par}(S_1, S_2)$ to $\mathsf{Par}(S_1', S_2')$. It is defined as the set of pairs l, l' such that there exist $(l_1, l_1') \in F_1$ and $(l_2, l_2') \in F_2$ such that (i) l is an interleaving of l_1 and l_2 and (ii) l' is an interleaving of l_1' and l_2'. (See also [5], Section 3, which defines sequential and parallel composition operators for data trace types.) This is an operator because there is at least one output on every input, found by projecting out the input to F_1 and to F_2. For this definition of parallel composition, we get the following compositionality laws; in fact, they are even stronger than for sequential composition, because determinism is compositional as well.

Proposition 4.7.1. Let F_1 be an operator from S_1 to S_1', and let F_2 be an operator from S_2 to S_2'. Additionally assume that $\mathsf{Par}(S_1, S_2)$ and $\mathsf{Par}(S_1', S_2')$ are grounded. Then:

1. If F_1 and F_2 are monotone, then $F_1 \| F_2$ is monotone.

2. If F_1 and F_2 are type-safe, then $F_1 \| F_2$ is type-safe.

3. If F_1 and F_2 are deterministic, then $F_1 \| F_2$ is deterministic.

Proof sketch. Let π_1, π_2, π_1', and π_2' denote the projection of a list to S_1, S_2, S_1', and S_2' respectively. A basic fact about the type $\mathsf{Par}(S_1, S_2)$ is that any two interleavings of a list in S_1 and a list in S_2 are equivalent. For (1), if we have an extension of the input list then applying π_1 and π_2 we also get extensions of each coordinate; these correspond to extensions in the output by monotonicity. The additional output produced by F_1 and F_2 can be interleaved arbitrarily. For (2), for a specific output list o, the type equivalence $l \equiv m \bmod \mathsf{Par}(S_1, S_2)$ can be deconstructed into a sequence of swapping adjacent elements; we push this sequence to construct a type equivalence on $\pi_1(o)$ and on $\pi_2(o)$. Specifically, each swap is either a swap of an element in S_1 and an element in S_2, which we ignore in the output, or a swap between two elements in S_1, which we push to the corresponding projection $\pi_i(o)$. Then at the end we construct any interleaving we want from the outputs in S_1' and in S_2'. For (3), we fix an input l, which is an interleaving of some l_1 and l_2, and suppose it has two different outputs. The first output must be an interleaving of some l_1' and l_2' and the second an interleaving of some m_1' and m_2', by definition of the parallel composition operator, where l_1', m_1' are outputs of F_1 on l_1 and similarly for l_2', m_2'. Then we apply determinism in each case, and lift type equivalence to the interleaving by the rules of $\mathsf{Par}(S_1', S_2')$. □

Name	Example I/O	Input type S	Output type S'	Monotone?	Type-safe?	Deterministic?
Max	$[4,0,4] \mapsto [4]$	Seq(int) or Bag(int)	Seq(int)	No	Yes	Yes
ArgMax	$[4,0,4] \mapsto [0]$ $[4,0,4] \mapsto [2]$	Seq(int)	Seq(int)	No	Yes	No
		Bag(int)	Seq(int)	No	No	No
ArgZero	$[0,4,0] \mapsto [0]$ $[0,4,0] \mapsto [2]$	Seq(int)	Seq(int)	Yes	Yes	No
		Bag(int)	Seq(int)	Yes	No	No
Fresh	$[1,4] \mapsto [3]$ $[1,4] \mapsto [5]$	Seq(int) or Bag(int)	Seq(int)	No	Yes	No
Any	$[1,4] \mapsto [1]$ $[1,4] \mapsto [4]$	Seq(int) or Bag(int)	Seq(int)	Yes	Yes	No
Unique	$[1,2,3] \mapsto [3]$ $[1,2,1] \mapsto [2]$	Seq(int) or Bag(int)	Seq(int)	No	Yes	Yes
DeDup	$[1,2,3] \mapsto [1,2,3]$ $[1,2,1] \mapsto [1,2]$	Seq(int)	Seq(int) or Bag(int)	Yes	Yes	Yes
		Bag(int)	Seq(int)	Yes	No	Yes
		Bag(int)	Bag(int)	Yes	Yes	Yes
Dup	$[1,2] \mapsto [1,1,2]$ $[1,2] \mapsto [1,1,2,2]$	Seq(int)	Seq(int) or Bag(int)	Yes	Yes	No
		Bag(int)	Seq(int)	Yes	No	No
		Bag(int)	Bag(int)	Yes	Yes	No

Figure 4.5: Outtakes from Figure 4.3: additional examples of monotonicity, type safety, and determinism for stream operators.

4.7.3. Additional Operator Examples

Figure 4.5 shows some additional outtakes to add to the examples listed in Figure 4.3. Max is the usual maximum function. ArgMax returns the index of the maximum element, or any index if there are multiple maximums. ArgZero similarly returns the index of a zero, rather than of the maximum. Fresh takes an input stream and returns any unique new element (not seen in the input). Any is the dual of Fresh, and returns an element seen in the input. Unique is the famous count-distinct function, returning the number of unique elements in the input. DeDup removes all instances of each input element after the first, and Dup nondeterministically duplicates some number of the input items (at-least-once behavior).

4.7.4. Synchronization Relation

Stream linearizations in this chapter use a symmetric dependence relation $e_1 \mathcal{D} e_2 \bmod S$ between pairs of events. It is also possible to formalize linearizations using a transitive, asymmetric relation called *synchronizes*: "a synchronizes b" means that a is either at the same level in the hierarchy or at a greater level; or, that a is an ancestor of b in the tree visualization of the stream type (Figure 4.2). For example, in the figure, ■ synchronizes ■, ◯, and △; ◯ synchronizes ◯ of the same key only; and △ doesn't synchronize anything.

To formalize this relation, only a single change to the rules in Section 4.2.2 is required, in the SYNCH case, to make it asymmetric. The modified rule is as follows:

$$\frac{t : T \qquad e : \mathsf{Event}(\mathsf{Sync}(T, S))}{t \mathcal{D} e \bmod \mathsf{Sync}(T, S)} \textsc{Synch-asymmetric}$$

The symmetric version of the dependence relation can be derived from the synchronization relation as its symmetric closure. What is interesting about the synchronization relation is that it satisfies stronger mathematical properties, in particular it satisfies the following definition:

Definition 4.7.2. A *synchronization relation* is a binary relation $\preceq$ subject to two conditions:

1. Transitivity: if $a \preceq b$ and $b \preceq c$ then $a \preceq c$.
2. If $a \preceq b$ and $a' \preceq b$, then either $a \preceq a'$ or $a' \preceq a$.

We chose not to include the idea of synchronization relations in the chapter because the rules for linearization equivalence only use the relation up to symmetry (see the rule INDEP), so the synchronization relation as a concept is not needed directly, and probably only serves to confuse the reader. However, the synchronization relation may be useful for other purposes, and developing a theory of streams with a synchronization relation could be an interesting direction for future work.

4.7.5. Omitted Typing Rules

Below we include the typing rules for $e \mathcal{I} e' \bmod S$, omitted from Section 4.2.2. This is simply the complement of the relation $e \mathcal{D} e' \bmod S$ where $e, e' : \mathsf{Event}(S)$.

$$\frac{e \mathrel{\mathcal{I}} e' \text{ mod } S}{e \mathrel{\mathcal{I}} e' \text{ mod } \mathsf{Sync}(T, S)}\ \textsc{Sub-I} \qquad \frac{e_1 : \mathsf{Event}(S_1) \qquad e_2 : \mathsf{Event}(S_2)}{e_1 \mathrel{\mathcal{I}} e_2 \text{ mod } \mathsf{Par}(S_1, S_2)}\ \textsc{Par-I}$$

$$\frac{e \mathrel{\mathcal{I}} e' \text{ mod } S_1}{e \mathrel{\mathcal{I}} e' \text{ mod } \mathsf{Par}(S_1, S_2)}\ \textsc{Par-I1} \qquad \frac{e \mathrel{\mathcal{I}} e' \text{ mod } S_2}{e \mathrel{\mathcal{I}} e' \text{ mod } \mathsf{Par}(S_1, S_2)}\ \textsc{Par-I2}$$

$$\frac{k : K \qquad e \mathrel{\mathcal{D}} e' \text{ mod } S}{(k, e) \mathrel{\mathcal{D}} (k, e') \text{ mod } \mathsf{ParBy}(K, S)}\ \textsc{ParBy-I1}$$

$$\frac{k : K \qquad k' : K \qquad k \neq k' \qquad e : S \qquad e' : S}{(k, e) \mathrel{\mathcal{I}} (k', e') \text{ mod } \mathsf{ParBy}(K, S)}\ \textsc{ParBy-I2}$$

$$\frac{t : T \qquad t' : T}{t \mathrel{\mathcal{I}} t' \text{ mod } \mathsf{Bag}(T)}\ \textsc{Bag}$$

4.7.6. Incrementality

Viewing streams as static objects (batches, linearizations, and posets) has the disadvantage that it makes it not obvious whether a function on these objects is suitable for incremental processing. We somewhat recovered this suitability through the *monotonicity* requirement on operators F. Formally, one can show that if F is monotone, then an implementation can produce output on a prefix of the input stream safely; when getting some later items, we know that the new output will be a suffix of the old output, so we can produce the difference as new output. For example, the map-plus-one function is monotone: mapping $[1, 2, 3]$ to $[2, 3, 4]$. When receiving the input $[1, 2]$, we can safely produce the output $[2, 3]$; when 3 later arrives, we produce the new output $[4]$.

It is also useful, though, to define explicitly incremental objects; stream producers and consumers which are monotonic *by definition*. For example, here is a definition of a generator object which

produces events of a stream type:

$$\frac{\texttt{State} : \texttt{Type} \qquad f : \texttt{State} \to \texttt{Option<Event}(S) \times \texttt{State>}}{\texttt{Generator}(f) : S}\ \textsc{Generator}$$

The later chapters will consider incremental computation more explicitly, and we generally adopt an event-by-event view of streams for the remainder of the thesis.

4.8. Historical Notes

The stream types defined in this chapter differ from what we called *synchronization schemas* [2] in a few ways. Mostly these are not conceptually fundamental differences, but they nevertheless have some consequences. We survey each difference individually and also discuss the relationship with *data-trace types* [9].

4.8.1. Base Types vs. Tuple Types

First, synchronization schemas enforce that the base types are record types (also known as named tuples or headers). Here is the definition of *header* from [2], which is standard in any textbook on relational databases:

Definition 4.8.1 (Headers and tuples)**.** A *header* H consists of a unique *header name* α and *fields* $\langle \alpha_i : \tau_i \rangle$, for $1 \leq i \leq n$, where each α_i is a *field name* and τ_i is a *field type*. A *tuple* x of type H, denoted $x : H$, is of the form $x = (x_1, x_2, \ldots, x_n)$, where each $x_i : \tau_i$.

In fact, synchronization schemas used a *set* of headers (not just a single header). For a set $\mathcal{H}$ of headers, we write $x : \mathcal{H}$ if $x : H$ for some $H \in \mathcal{H}$. In the constructs $\mathsf{Bag}(T)$ and $\mathsf{Sync}(T, S)$, T is a set of headers.

4.8.2. Treatment of Key Fields in Key-Based Parallelism

There is also a difference in the ParBy construct $\mathsf{ParBy}(K, S)$. In the present treatment, we assume that the ParBy construct explicitly adds a key of type K to the data of all events in the stream: so note that in the definition of events $e : \mathsf{Event}(\mathsf{ParBy}(K, S))$, we have that e is an ordered pair (k, e') where $k : K$. In synchronization schemas, instead K is required to be one of the fields of all header

base types. In particular, one advantage of this is that $\mathsf{ParBy}(K, \mathsf{ParBy}(K, S))$ is type-isomorphic to $\mathsf{ParBy}(K, S)$ for synchronization schemas (while they are not type-isomorphic for our stream types). One disadvantage is that synchronization schemas must satisfy a *well-formedness* condition, meaning that within $\mathsf{ParBy}(K, S)$, all tuple types (headers) present in S must include the field K. For reference, here is the definition of a well-formed schema from [2]:

Definition 4.8.2 (Well-formed schema)**.** First, the partitioning construct $\mathsf{ParBy}(K, S)$ naturally gives rise to a concept of scope for partition keys: for each partition key $k \in K$ and for each header H appearing in the schema S, we say that H is *in the scope of* k.

A synchronization schema S is *well-formed* if the following conditions hold: (1) no header H appears in S twice, (2) if a header H is in the scope of a partition key k, then k is a field of H, i.e., $k \in H$, and (3) if a partition schema $\mathsf{ParBy}(K, S)$ is in the scope of a partition key k, then $k \notin K$.

The first condition (1) is necessary for unambiguous parsing, while (2) and (3) ensure that splitting on a key field is meaningful in a given context. Note that it is straightforward to check the conditions necessary for a schema to be well-formed.

In our development, at least for the batch view, our stream types don't need to satisfy any well-formedness conditions; for equivalence with linearizations though (Proposition 4.4.2), we do need a condition analogous to (1) (groundedness, i.e. that base types in a schema are disjoint). We don't need conditions (2) and (3).

4.8.3. Batches vs. Series-Parallel Streams

The batches we define ($b : \mathsf{Batch}(S)$) are analogous to what were called *series-parallel streams* (or SPSs) in [2], the type inhabitants for synchronization schemas. SPSs are an inductive representation of streams that are values of these types. Due to the different treatment of ParBy, SPSs have to be defined with a particular valuation of key fields in mind: the type $S[v]$ is elements of the stream S where each tuple has key values v, where v assigns a value to each key type.

Series-parallel streams: Consider the sequence of `GPS` events corresponding to the red taxi. The type of this sequence is $\mathsf{Seq}(\texttt{GPS})[\texttt{taxiID} = \texttt{red}]$. Such a type can be viewed as a *refinement type* of the schema type $\mathsf{Seq}(\texttt{GPS})$. This type is more specialized than $\mathsf{Seq}(\texttt{GPS})$, since all these events share a common value of the field `taxiID`.

If $d : H$ and F is a subset of the fields of H, we write $d|_F$ for the restriction of d to contain only those fields in F. For a set K of partition keys, K can also be considered to be a header containing these keys as its only fields. Then, for a particular tuple v of such a header type K, for a schema S, we use $S[v]$ to denote the refinement of schema S to an instance where all the tuples are required to have key values as specified by v.

Following [2], we use the following syntactic constructs to capture the structure of the desired series-parallel streams: $[x_1, x_2, \ldots, x_n]$ for a sequence (list), $\{x_1, x_2, \ldots, x_n\}$ for a bag (with standard bag equality semantics), $\langle x_1, x_2 \rangle$ for a pair of x_1 and x_2 where x_1 and x_2 are thought of as parallel instead of sequential, and $v \mapsto x$ to represent a key-indexed value.

Definition 4.8.3 (Series-Parallel Streams)**.** Let S be a synchronization schema. A *series-parallel stream* (SPS) $t : S[v]$ for a specific instantiation of key values $v : K$ is inductively defined as follows:

- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \ldots, m$, then $t = \{d_1, \ldots d_m\}$ is an SPS of type $\mathsf{Bag}(\mathcal{H})[v]$.
- If $t_1 : S_1[v]$ and $t_2 : S_2[v]$, then $t = \langle t_1, t_2 \rangle$ is an SPS of type $\mathsf{Par}(S_1, S_2)[v]$.
- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \ldots, m$, and if $t_i : S'[v]$ for $i = 0, 1, \ldots, m$, then $t = [t_0, d_1, t_1, \ldots, d_m, t_m]$ is an SPS of type $\mathsf{Sync}(\mathcal{H}, S')[v]$.
- Suppose that K' is a set of partition keys disjoint from K, and that $v'_1, v'_2, \ldots, v'_m : K'$ are *distinct* instances of key values for K'. Suppose $t_1, t_2, \ldots, t_m$ are *nonempty* streams such that $t_i : S'[v_i]$, and let $v_i : K \cup K'$ to the unique valuations such that $v_i|_K = v$ and $v_i|_{K'} = v'_i$, i.e., the extension of v'_i with the key values in v. Then $t = \{v'_1 \mapsto t_1, v'_2 \mapsto t_2, \ldots, v'_m \mapsto t_m\}$ is an SPS of type $\mathsf{ParBy}(K', S')[v]$.

We write $t : S$ when $K = \varnothing$ for $t : S[()]$, where $()$ is the empty tuple of type K. When $S[v]$ is clear from the context, we write $\bot : S[v]$ for the empty SPS: this abbreviates $\{\}$ for $S = \mathsf{Bag}(\mathcal{H})$, $\langle \bot, \bot \rangle$ for $S = \mathsf{Par}(S_1, S_2)$, $[\bot]$ for $S = \mathsf{Sync}(\mathcal{H}, S')$, and $\{\}$ for $S = \mathsf{ParBy}(K', S')$.

Additionally, [2] defined *concatenation* of series-parallel streams, denoted $\circ$. In our setting we instead derive $\circ$ from concatenation on linearizations rather than defining it inductively.

Definition 4.8.4 (Concatenation and Prefix Ordering for SPSs)**.** Let $t, u : S[v]$ be series-parallel streams over the same schema S and key valuation v. The *concatenation* $t \circ u$ is defined inductively on the structure of S:

- If $S = \mathsf{Bag}(\mathcal{H})$, $t = \{d_1, \ldots, d_m\}$, and $u = \{e_1, \ldots, e_n\}$, then

$$t \circ u = \{d_1, \ldots, d_m, e_1, \ldots, e_n\}.$$

- If we have $S = \mathsf{Sync}(\mathcal{H}, S')$, $t = [t_0, d_1, t_1, \ldots, d_m, t_m]$, and $u = [u_0, e_1, u_1, \ldots, e_n, u_n]$, then

$$t \circ u = [t_0, d_1, t_1, \cdots, d_m, (t_m \circ u_0), e_1, u_1, \cdots, e_n, u_n].$$

- If $S = \mathsf{ParBy}(K, S')$, then let the overlapping key values between t and u be $v_1, v_2, \ldots, v_l$, with additional keys $v'_1, v'_2, \ldots, v'_m$ in t only and $v''_1, v''_2, \ldots, v''_n$ in u only. If $t = \{v_1 \mapsto t_1, \ldots, v_l \mapsto t_l, v'_1 \mapsto t'_1, \ldots, v'_m \mapsto t'_m\}$ and $u = \{v_1 \mapsto u_1, \ldots, v_l \mapsto u_l, v''_1 \mapsto u'_1, \ldots, v''_n \mapsto u'_n\}$, then

$$\begin{aligned} t \circ u = \{\ & v_1 \mapsto t_1 \circ u_1, \ldots, v_l \mapsto t_l \circ u_l, \\ & v'_1 \mapsto t'_1, \ldots, v'_m \mapsto t'_m, \\ & v''_1 \mapsto u'_1, \ldots, v''_n \mapsto u'_n. \ \} \end{aligned}$$

- If $S = \mathsf{Par}(S_1, S_2)$, $t = \langle t_1, t_2 \rangle$, and $u = \langle u_1, u_2 \rangle$, then

$$t \circ u = \langle t_1 \circ u_1, t_2 \circ u_2 \rangle.$$

For $t, u : S[v]$, t is said to be a *prefix* of u, written $t \preceq u$, if there exists a series-parallel stream $t' : S[v]$ such that $t \circ t'$ equals u.

The following proposition was stated in [2]. It straightforwardly follows from viewing stream concatenation as derived from concatenation of linearizations instead of as a separate operation.

Proposition 4.8.5. For each type $S[v]$ and for all $t, t', t'' : S[v]$, the following hold: (1) $t \circ \bot = \bot \circ t = t$. (2) $(t \circ t') \circ t'' = t \circ (t' \circ t'')$. (3) $t \preceq t$. (4) If $t \preceq t'$ and $t' \preceq t$, then $t = t'$. (5) If $t \preceq t'$ and $t' \preceq t''$, then $t \preceq t''$.

4.8.4. Reflexivity of the Dependence Relation

Just as with our streams, synchronization schemas give rise to a dependence relation on events. This is not substantially different, but does differ in one important way: it is guaranteed in [2] to be reflexive, that is $e \mathcal{D} e \bmod S$ for all events $e : \mathsf{Event}(S)$. As remarked in [2] and included below (see Proposition 4.8.8), this has some advantages. However, we find it counterintuitive because in the relation case, it means that events of the same value are ordered. So we have dropped it for our stream types. The definition of dependence relation developed for synchronization schemas follows:

Definition 4.8.6 (Dependence Relation)**.** Let S be a synchronization schema and let $\mathbf{headers}(S)$ be all the headers appearing in S. The *dependence relation* is a binary relation on tuples of $\mathbf{headers}(S)$, written $x \, D_S \, y$ for $x, y : \mathbf{headers}(S)$, and defined inductively as follows: (i) if $S = \mathsf{Bag}(\mathcal{H})$, then D_S is the empty set; (ii) if $S = \mathsf{Sync}(\mathcal{H}, S')$, then D_S is $\{ (x, y) \mid x : \mathcal{H} \text{ or } y : \mathcal{H} \text{ or } x \, D_{S'} \, y \}$; (iii) if $S = \mathsf{ParBy}(K, S')$, then D_S is $\{ (x, y) \mid (x \, D_{S'} \, y) \text{ and } x|_K = y|_K \}$; and (iv) if $S = \mathsf{Par}(S_1, S_2)$, then D_S is $D_{S_1} \cup D_{S_2}$.

The dependence relation D_S over the set X of tuples then gives rise to the following equivalence relation on sequences s, s' over X, just as we defined $l \equiv l' \bmod S$.

Definition 4.8.7 (Equivalent sequences)**.** Let $D \subseteq X \times X$ be a symmetric relation. The equivalence relation $\equiv_D$ over sequences over X is the smallest equivalence relation (i.e., reflexive, symmetric, and transitive) such that (1) commuting independent items: for all $x, y \in X$, if *not* $x \, D \, y$, then $xy \equiv_D yx$; and (2) closure under (sequence) concatenation: for $s_1, s_1', s_2, s_2' \in X^*$, if $s_1 \equiv_D s_1'$ and $s_2 \equiv_D s_2'$ then $s_1 s_2 \equiv_D s_1' s_2'$. For a schema S, two sequences s, s' are *equivalent with respect to* S, written $s \equiv_S s'$, if $s \equiv_{D_S} s'$.

The reflexivity of D_S is not strictly necessary (we could drop it and have the empty relation in the base case of $\mathsf{Bag}(\mathcal{H})$), but is convenient, as it means that the dependence relation contains no extraneous information other than what it implies about event ordering. In particular, we have the following

proposition: for dependence relations D_1 and D_2, $D_1 = D_2$ iff $\forall s, s' \in X^*, s \equiv_{D_1} s' \iff s \equiv_{D_2} s'$. This means that if S_1 and S_2 are synchronization schemas, $D_{S_1} = D_{S_2}$ iff $\equiv_{S_1}$ and $\equiv_{S_2}$ are the same.

Proposition 4.8.8. Let D_1 and D_2 be two dependence relations on the same set of tuples X, arising from synchronization schema S_1 and S_2. Then $D_1 = D_2$ iff

$$\forall s, s' \in X^*, s \equiv_{D_1} s' \iff s \equiv_{D_2} s'.$$

In particular, if S_1 and S_2 are synchronization schemas, this implies $D_{S_1} = D_{S_2}$ iff $\equiv_{S_1}$ and $\equiv_{S_2}$ are the same equivalence relation on sequences.

Proof. The forward direction is immediate, and the backward direction follows taking any two distinct tuples x_1, x_2 and considering the sequence $x_1 x_2$. □

Finally, [2] also defined a tight correspondence (isomorphism) between sequences and series-parallel streams via dependence relations, which is analogous to our development in Proposition 4.4.2. Below is Proposition 14 of [2].

Proposition 4.8.9. Let S be a synchronization schema. (1) For every sequence s of tuples of type $\mathbf{headers}(S)$, there exists a unique (up to equality) $t : S$ such that s is a flattening of t. (2) For all sequences s_1, s_2 of tuples of type $\mathbf{headers}(S)$ and $t : S$, (a) if $s_1 \equiv_S s_2$ and s_1 is a flattening of t then s_2 is a flattening of t also, and (b) if s_1 and s_2 are both flattenings of t then $s_1 \equiv_S s_2$.

Proof. By induction on S. For $\mathsf{Bag}(\mathcal{H})$, all three conditions follow by the correspondence between a multiset of items and its linearizations. For $\mathsf{Par}(S_1, S_2)$ and for $\mathsf{ParBy}(K, S_1)$, we observe that sequences over tuples of $\mathbf{headers}(S)$ are interleavings of events each from a subschema, and all such interleavings are equivalent with respect to $\equiv_S$; conversely $\equiv_S$ only holds between different interleavings of the same two or more sequences up to equivalence. I.e., for parallel composition, if $s \equiv_S s'$ and s is an interleaving of s_1 and s_2 and s_1' is an interleaving of s_2', then $s_1 \equiv_{S_1} s_1'$ and $s_2 \equiv_{S_2} s_2'$. The most interesting case is $\mathsf{Sync}(\mathcal{H}, S_1)$. Here, we essentially apply the idea that $(a \cup b)^* = (a^* b)^* a^*$ for languages: in this context a is tuples of $\mathbf{headers}(S')$ and b is tuples of $\mathcal{H}$. So sequences over tuples of $\mathbf{headers}(S)$ decompose into a sequence of subsequences over S' delineated by $\mathcal{H}$ events, where there is one more subsequence than the number of $\mathcal{H}$ events. Since $\mathcal{H}$ events

are fully dependent on everything else, this decomposition is not changed by $\equiv$, which can thus be identified with equality on the sequence of $\mathcal{H}$ events together with equivalence on each $\mathbf{headers}(S')$ substream. The definition of flattening reflects this decomposition exactly. □

4.8.5. Relationship to Data-Trace Types

Data-trace types [9] essentially correspond to the linearizations view: a data-trace type is a dependence relation $\mathcal{D}$ on pairs of events, with a derived equivalence on linearizations $l \equiv l' \bmod S$ just as we defined it, except it depends only on $\mathcal{D}$ and not on S. So stream types S are an abstraction over dependence relations and thus an abstraction over data trace types. Proposition 4.4.3 shows that in fact the abstraction loses some dependence relations and so is less general; but we have not found the pathological cases it excludes (such as the graphs C_4 and P_4) to be useful in practice.

Here is the definition of a data type, dependence relation, and equivalence relation from [9], where the latter bears close resemblance to equivalence on linearizations.

Definition 4.8.10. A *data type* $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ consists of a potentially infinite *tag alphabet* Σ and a value type T_σ for every tag $\sigma \in \Sigma$. The set of *elements* of type A, or *data items*, is equal to $\{(\sigma, d) \mid \sigma \in \Sigma \text{ and } d \in T_\sigma\}$, which we will also denote by A. The set of *sequences* over A is denoted as A^*.

A *dependence relation* on a tag alphabet Σ is a symmetric binary relation on Σ. We say that the tags σ, τ are *independent* (w.r.t. a dependence relation D) if $(\sigma, \tau) \notin D$. For a data type $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ and a dependence relation D on Σ, we define the dependence relation that is induced on A by D as $\{((\sigma, d), (\sigma', d')) \in A \times A \mid (\sigma, \sigma') \in D\}$, which we will also denote by D. Define $\equiv_D$ to be the smallest congruence (w.r.t. sequence concatenation) on A^* containing $\{(ab, ba) \in A^* \times A^* \mid (a, b) \notin D\}$. Informally, two sequences are equivalent w.r.t. $\equiv_D$ if one can be obtained from the other by repeatedly commuting adjacent items with independent tags.

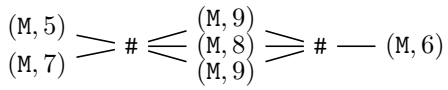
Definition 4.8.11. A *data-trace type* is a pair $X = (A, D)$, where A is a data type and D is a dependence relation on the tag alphabet of A. A *data trace* of type X is a congruence class of the relation $\equiv_D$. We also write X to denote the set of data traces of type X. Since the equivalence $\equiv_D$ is a congruence w.r.t. sequence concatenation, the operation of concatenation is also well-defined on data traces: $[u] \cdot [v] = [uv]$ for sequences u and v, where $[u]$ is the congruence class of u. We define

the relation $\leq$ on the data traces of X as a generalization of the prefix partial order on sequences: for data traces $\mathbf{u}$ and $\mathbf{v}$ of type X, $\mathbf{u} \leq \mathbf{v}$ iff there are $u \in \mathbf{u}$ and $v \in \mathbf{v}$ s.t. $u \leq v$ (i.e., u is a prefix of v). The relation $\leq$ on data traces of a fixed type is a partial order.

Example 4.8.12. Suppose we want to process a stream that consists of sensor measurements and special symbols that indicate the end of a one-second interval. The data type for this input stream involves the tags $\Sigma = \{\mathtt{M}, \mathtt{\#}\}$, where `M` indicates a sensor measurement and `#` is an end-of-second marker. The value sets for these tags are $T_{\mathtt{M}} = \mathbb{N}$ (the natural numbers), and $T_{\mathtt{\#}} = \mathtt{Ut}$ is the unit type (singleton). So, the data type $A = (\Sigma, T_{\mathtt{M}}, T_{\mathtt{\#}})$ contains measurements $(\mathtt{M}, d)$, where d is a natural number, and the end-of-second symbol `#`.

The dependence relation $D = \{(\mathtt{M}, \mathtt{\#}), (\mathtt{\#}, \mathtt{M}), (\mathtt{\#}, \mathtt{\#})\}$ says that the tag `M` is independent of itself, and therefore consecutive `M`-tagged items are considered unordered. For example, $(\mathtt{M}, 5)$ $(\mathtt{M}, 5)$ $(\mathtt{M}, 8)$ # $(\mathtt{M}, 9)$ and $(\mathtt{M}, 8)$ $(\mathtt{M}, 5)$ $(\mathtt{M}, 5)$ # $(\mathtt{M}, 9)$ are equivalent w.r.t. $\equiv_D$.

A data trace of X can be represented as a sequence of multisets (bags) of natural numbers and visualized as a partial order on that multiset. The trace corresponding to the sequence of data items $(\mathtt{M}, 5)$ $(\mathtt{M}, 7)$ # $(\mathtt{M}, 9)$ $(\mathtt{M}, 8)$ $(\mathtt{M}, 9)$ # $(\mathtt{M}, 6)$ is visualized as:



In the above picture, a line from left to right indicates that the item on the right must occur after the item on the left. The end-of-second markers # separate multisets of natural numbers. So, the set of data traces of X has an isomorphic representation as the set $\mathrm{Bag}(\mathbb{N})^+$ of nonempty sequences of multisets of natural numbers. In particular, the empty sequence ϵ is represented as $\emptyset$ and the single-element sequence # is represented as $\emptyset$ $\emptyset$.

Key properties described in the data trace types work bear close resemblance to the properties in Section 4.6. The notion of a *data-string transduction* is similar to a monotone operator; it is a monotone function $f : A^* \to B^*$. (So to be precise, a data-string transduction is a monotone functional operator.) Going further, the definition of *consistency*, which describes when a data-string

transduction gives rise to a data-trace type, bears close resemblance to type safety and determinism of operators:

Definition 4.8.13 (Consistency)**.** Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. We say that a data-string transduction $f : A^* \to B^*$ is (X, Y)-*consistent* if $u \equiv_D v$ implies that $\bar{f}(u) \equiv_E \bar{f}(v)$ for all $u, v \in A^*$.

CHAPTER 5 : Compositionality

The meaning of a compound expression is a function of the meanings of its parts and of the syntactic rule by which they are combined.

—Barbara Partee, 1984 [220], as formulated in [221]

In this section, we define *SPS-transformers* (SPSTs), a programming language for computations over series-parallel streams. If synchronization schemas are provided as types for the input and output streams of a computation, then an SPST is a (type-safe and deterministic) function from input batches to output batches, which respects the structure given by the types. This material was originally presented in [2] (some of it was delegated to the appendix).

The primary goal of SPSTs is to be *compositional*: they allow for defining operators over a stream in a compositional way with respect to the type of that stream. Per the goals of our introduction, every SPST should satisfy a type safety property. Formally, it should be a deterministic function from input batches to output batches. By Theorem 4.6.3 this implies that a corresponding *operator*, a low-level function from lists of input to lists of output, can be defined that is type-safe and deterministic in a more realistic event-by-event processing sense. We will define the semantics of an SPST inductively on the input structure; the typing judgments for this semantics in each case demonstrate type safety.

An interesting aspect of the material in this section – which we have found helpful as a way to think about future work in this space – is the approach to achieve incremental processing by distinguishing *open* and *closed* semantics. For each transformer, the open semantics represents a monotonically-increasing function from input streams to output streams, i.e., an operator that satisfies *monotonicity*. The *closed* semantics represents the semantics after the input stream is terminated, so that the stream operator may produce some final output that is not incremental. For example, using this distinction, the function Sum described in Figure 4.3 can be represented as a function which produces no output in its open semantics, but produces the total sum in its closed semantics. We describe this distinction further in Section 5.2.

5.1. Notational Differences

The material in this section uses the definitions, notation, and terminology from [2], rather than those from Chapter 4. Stream types are called *synchronization schemas*, and batches are called *series-parallel streams*, or SPSs for short. In synchronization schemas, instead of base types T, there are sets of headers $\mathcal{H}$, representing the possible tuples that might occur for an event (See Section 4.8). So if $T = \mathcal{H} = (H_1, H_2, H_3)$ for example, then H_1, H_2, and H_3 are tuple types and the events of type T are either tuples of type H_1, or tuples of type H_2, or tuples of type H_3. It uses the notation $\preceq$ for stream prefix. This is defined using concatenation: for batches b_1 and b_2, we say $b_1 \preceq b_2$ if there is some b_1' such that $b_2 = b_1 \circ b_1'$.

5.2. Design Goals

In addition to satisfying **monotonicity**, **type safety**, and **determinism** as already discussed, SPSTs should satisfy the following additional design goals. First, they should respect the **parallelism** in the input and output: parallel input events should be processed in parallel, and parallel input threads should produce parallel output events. For example, given an input which is two streams in parallel, the computation should be written in such a way that the two streams are processed separately, and outputs corresponding to them should be unordered. Second, to allow for the specification of potentially complex computations, we additionally want our language to be **compositional**: it should be natural to construct a computation by combining sub-computations. For example, processing a stream of the hierarchical type $\mathsf{Sync}(\mathcal{H}, S)$ should be definable, both syntactically and semantically, in terms of existing computations defined over sub-streams of type S. Finally, any computation in our language should be **incremental** (or *streamable*): it should process the input in one pass, producing output incrementally.

To satisfy these design goals, we make the following technical choices. First, to satisfy the parallelism goal, we define SPSTs to have SPSs as input and output rather than sequential objects. The input being an SPS allows us to specify the computation to exploit parallelism, and the output being an SPS requires that we respect parallelism when producing output events.

To understand the challenges in defining the semantics due to the interplay between streamability and compositionality, consider a transformer P processing hierarchical streams of type $S = \mathsf{Sync}(\mathcal{H}, S')$

that we would like define in terms of a transformer P' processing streams of type S'. Consider an input stream $t = [\bot, d, t']$ of type S for an $\mathcal{H}$-tuple d and stream t' of type S'. Suppose we want to extend the input stream t with a tuple d'. If the type of d' is one of the headers appearing in S', then it really extends the sub-stream t', and should be processed by the transformer P'. For incrementality, we want to make sure that, while processing d', P' extends the output stream only by adding new items. Formally, this means that the output of P' on the input stream t' should be a prefix of its output on the stream $t' \circ d$. With this motivation, we define such a semantics, which we call *open* semantics, for transformers as functions from input to output streams, and ensure that it is *monotonic* with respect to prefix ordering (see Theorem 5.5.1). But now suppose that the item d' is an $\mathcal{H}$-tuple that acts as a synchronization marker for the events in the sub-stream t'. Then to process it, the transformer P' should return, and let the top-level transformer P process the item d'. During this return, the transformer P' can do additional computation and produce additional output items even though the stream it has processed is still t'. This is a typical case when S' corresponds to key-based partitioning, and the arrival of the synchronization marker d' triggers the reduce operation that aggregates the results of the computations of the key-indexed sub-streams of t'. This though requires us to define another semantics of the transformer P' on the input stream t' that extends the open semantics and includes the results of the computation upon return. We call it *closed* semantics to indicate that it is applicable when the current stream is being closed. Note that the result of computation of P on the stream $[\bot, d, t', d', \bot]$ can be described by relying on the closed semantics of P' on the stream t'. In terms of existing work on punctuation, the closed semantics can be thought of as the stream output on a stream terminated by an *end-of-stream* marker.

Finally, an SPST is a function on pairs: it takes an initial value and an input SPS to a final value and an output SPS. We need this for compositionality: without the initial value as input, an SPS-transformer on a sub-stream of the input could not be initialized based on the surrounding context. Similarly, the final value (separate from the series of output items produced) can be used to describe a summary of the input stream to be used in the surrounding context when the computation finishes.

We summarize all of these choices in the following definition of the *interface* for an SPST. We also define subtyping for the interface, where the output is relaxed. Each of the language constructs will

then implement this interface. In Section 5.4, we give an extended example to illustrate the formal definitions in this section.

Definition 5.2.1 (SPS-transformer interface)**.** An SPS-transformer (SPST) P has:

- A *type* denoted (X, S, X', S'), where X is the type for the initialization value, S is an input synchronization schema, X' is a type for the final return value, and S' is an output synchronization schema. We write $P : (X, S, X', S')$.
- An *open semantics* denoted $[\![P]\!]_O(x, t) = t'$, where $x : X$ is the initial value, $t : S$ is the input SPS, and $t' : S'$ is the incrementally produced output SPS.
- A *closed semantics* denoted $[\![P]\!]_C(x, t) = (x', t')$, where $x' : X'$ is the initial value, $t : S$ is the input SPS, $x' : X'$ is the final value, and $t' : S'$ is the output SPS. We additionally enforce that the open semantics is a prefix of the closed semantics: $[\![P]\!]_O(x, t) \preceq t'$.

Definition 5.2.2. If $S_1' \lesssim S_2'$ (Definition 4.5.1), then (X, S, X', S_1') is a *subtype* of (X, S, X', S_2'). If $P : (X, S, X', S_1')$ then we also write $P : (X, S, X', S_2')$. The open and closed semantics are derived as the unique output stream given by Proposition 4.5.2.

In the remainder of the section, we give one language construct corresponding to each constructor of the input SPS. Some additional notation: for a set of headers $\mathcal{H}$, we write $\mathbf{tup}(\mathcal{H})$ for the set of tuples $x : \mathcal{H}$. For a synchronization schema S, we write $\mathbf{sps}(S)$ for the set of SPSs $t : S$. We write $\mathbf{bag}(X)$ for the set of bags (multisets) of items of type X.

5.3. Syntax and Semantics

5.3.1. Relational SPST

We start with the relational SPST, which represents a standard relational operator that can be used to process a bag of items, producing another bag of items. Relational operators are well studied and are commonly defined using SQL and its extensions. Our design choice here is to not impose a particular relational base language or SQL variant; instead, the relational operator is given as two *black-box* functions, which define the open and closed semantics, respectively. We only require that

these are functions on bags (i.e., independent of the input order), and that the open semantics is monotone and a prefix of the closed semantics.

Definition 5.3.1 (Relational SPST)**.** A relational SPST

$$P : (X, \mathsf{Bag}(\mathcal{H}), X', \mathsf{Bag}(\mathcal{H}'))$$

consists of two fields:

$$P.\mathsf{open} : X \times \mathbf{sps}(\mathsf{Bag}(\mathcal{H})) \to \mathbf{sps}(\mathsf{Bag}(\mathcal{H}'))$$
$$\text{and} \quad P.\mathsf{closed} : X \times \mathbf{sps}(\mathsf{Bag}(\mathcal{H})) \to X' \times \mathbf{sps}(\mathsf{Bag}(\mathcal{H}')).$$

such that (1) $P.\mathsf{open}$ is *monotone*: if $r_1 \preceq r_2$, then $P.\mathsf{open}(x, r_1) \preceq P.\mathsf{open}(x, r_2)$; and (2) $P.\mathsf{open}$ is a prefix of $P.\mathsf{closed}$: if $P.\mathsf{closed}(x, r) = (x', r')$ then $P.\mathsf{open}(x, r) \preceq r'$. The semantics of P is defined as $[\![P]\!]_O(x, r) = P.\mathsf{open}(x, r)$ and $[\![P]\!]_C(x, r) = P.\mathsf{closed}(x, r)$.

5.3.2. Parallel SPST

We now define the inductive SPSTs. An SPST processing inputs of type $\mathsf{Par}(S_1, S_2)$ is composed of two SPSTs running in parallel independently. The question here is, can the components SPSTs produce tuples of the same type? The answer is yes, provided such tuples, since they get produced independently, are summarized using a schema $\mathsf{Bag}(\mathcal{O})$, where $\mathcal{O}$ is a set of output headers. So the output schema for the parallel SPST will be $\mathsf{Par}(S_1', S_2', \mathsf{Bag}(\mathcal{O}))$.

Definition 5.3.2 (Parallel SPST)**.** Let S_1, S_2, S_1', S_2' be schemas. A parallel SPST

$$P : (X, \mathsf{Par}(S_1, S_2), X', \mathsf{Par}(S_1', S_2', \mathsf{Bag}(\mathcal{O}')))$$

consists of internal types X_1, X_2, X_1', X_2' and four fields:

$$P.\mathsf{left} : (X_1, S_1, X_1', \mathsf{Par}(S_1', \mathsf{Bag}(\mathcal{O}'))),$$
$$P.\mathsf{right} : (X_2, S_2, X_2', \mathsf{Par}(S_2', \mathsf{Bag}(\mathcal{O}'))),$$
$$P.\mathsf{init} : X \to X_1 \times X_2, \quad \text{and} \quad P.\mathsf{fin} : X_1' \times X_2' \to X'.$$

The semantics of P is as follows: if we have that $P.\mathsf{init}(x) = (x_1, x_2)$, $[\![P.\mathsf{left}]\!]_O(x_1, t_1) = \langle t_1', r_1' \rangle$, and $[\![P.\mathsf{right}]\!]_O(x_2, t_2) = \langle t_2', r_2' \rangle$, where $r_1', r_2' : \mathsf{Bag}(\mathcal{O}')$, and additionally $[\![P.\mathsf{left}]\!]_C(x_1, t_1) = (x_1', \langle t_1'', r_1'' \rangle)$ and $[\![P.\mathsf{right}]\!]_C(x_2, t_2) = (x_2', \langle t_2'', r_2'' \rangle)$, then

$$[\![P]\!]_O(x_1, t_1) = \langle \langle t_1', t_2' \rangle, r_1' \cup r_2' \rangle$$
$$[\![P]\!]_C(x_1, t_1) = (P.\mathsf{fin}(x_1', x_2'), \langle \langle t_1'', t_2'' \rangle, r_1'' \cup r_2'' \rangle).$$

5.3.3. Hierarchical SPST

When the input schema is $S = \mathsf{Sync}(\mathcal{H}, S_1)$, we want to define the corresponding SPST P parameterized by a sub-SPST from S_1 to S_1'. The SPST P maintains its own state that gets updated sequentially whenever any $\mathcal{H}$-tuple is processed, is passed to the sub-SPST when called, and is updated when the sub-SPST returns. The output schema of P has the same structure as the input: it is divided into synchronizing events and non-synchronizing events. On input synchronization events, any output tuple may be produced, including a synchronization event; but on input sub-stream events, it would be incorrect to produce an output synchronizing event, as this would not be produced in a consistent order. The distinction between closed and open semantics plays a key role here: synchronizing events, when processed by P, "close" the computation of the sub-SPST. To formalize this inductively, we introduce an auxiliary semantics $[\![P]\!]_{Aux}(y, t)$ where the output is an internal state (rather than a final value), and in which the input stream ends with a d_i event, i.e., the final t_i is $\perp$.

Definition 5.3.3 (Hierarchical SPST)**.** Let S_1 and S_1' be schemas, and $\mathcal{H}$ and $\mathcal{H}'$ be a set of input and output headers, respectively. Let $S' = \mathsf{Sync}(\mathcal{H}', S_1')$. A hierarchical SPST

$$P : (X, \mathsf{Sync}(\mathcal{H}, S_1), X', \mathsf{Sync}(\mathcal{H}', S_1'))$$

consists of internal types X_1, X_1', Y and six fields:

$$P.\mathsf{sub} : (X_1, S_1, X_1', S_1'),$$
$$P.\mathsf{update} : Y \times \mathbf{tup}(\mathcal{H}) \to Y \times \mathbf{sps}(S'),$$
$$P.\mathsf{call} : Y \to X_1, \qquad P.\mathsf{return} : Y \times X_1' \to Y,$$
$$P.\mathsf{init} : X \to Y, \qquad \text{and} \quad P.\mathsf{fin} : Y \to X' \times \mathbf{sps}(S').$$

The auxiliary semantics of P is denoted $[\![P]\!]_{Aux}(y,t) = (y',t')$, where $y, y' : Y$, and defined inductively *only* for t of the form $[t_0, d_1, t_1, \ldots, d_m, \bot]$. For the base case, $[\![P]\!]_{Aux}(y, \bot) = (y, \bot)$. Then inductively, if $[\![P]\!]_{Aux}(y,t) = (y',t')$, $t_1 : S_1$, and $d : \mathcal{H}$, and if we have $P.\mathsf{call}(y') = x_1$, $[\![P.\mathsf{sub}]\!]_C(x_1, t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{update}(y'', d) = (y''', t'')$, then $[\![P]\!]_{Aux}(y, t \circ [t_1, d, \bot]) = (y''', t' \circ t_1' \circ t'')$. Given the auxiliary semantics, we define the semantics of P on a trace decomposed as $t \circ [t_1]$, where the last list item of t is an empty sub-trace. Let $P.\mathsf{init}(x) = y$, $[\![P]\!]_{Aux}(y,t) = (y',t')$, and $P.\mathsf{call}(y') = x_1$. Additionally, let $[\![P.\mathsf{sub}]\!]_C(x_1,t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{fin}(y'') = (x', t'')$. Then:

$$[\![P]\!]_O(x,t) = t' \circ [\![P.\mathsf{sub}]\!]_O(x_1, t_1)$$
$$[\![P]\!]_C(x,t) = (x', t' \circ t_1' \circ t'').$$

5.3.4. Partitioned SPST

Finally, we define SPST for the partition-by case. The idea here is analogous to the parallel composition $\mathsf{Par}(S_1, S_2)$ case: each sub-stream corresponding to a different key value may produce output corresponding to that key value, *or* produce output corresponding to a common bag of tuples $\mathcal{O}'$. The partitioned SPST initializes the state of $P.\mathsf{sub}$ for each key with a nonempty SPS and runs the child SPST for each (non-empty) key in parallel. We additionally need an aggregation stage (applicable to the closed semantics only), in which we combine all of the partitioned states using a black-box relational operator $P.\mathsf{agg}$, similar to what was done in the relational SPST base case.

Definition 5.3.4 (Partitioned SPST)**.** Let $S = \mathsf{ParBy}(K, S_1)$ and $S' = \mathsf{ParBy}(K, S_1')$ be schemas, and $\mathcal{O}'$ a set of headers. A partitioned SPST

$$P : (X, \mathsf{ParBy}(K, S_1), X', \mathsf{Par}(\mathsf{ParBy}(K, S_1'), \mathsf{Bag}(\mathcal{O}'))$$

consists of internal types X_1, X_1' and three fields:

$$\begin{aligned} & P.\mathsf{sub} : (X_1, S_1, X_1', \mathsf{Par}(S_1', \mathsf{Bag}(\mathcal{O}'))), \\ & P.\mathsf{init} : X \times \mathbf{tup}(K) \to X_1, \\ \text{and} \quad & P.\mathsf{agg} : X \times \mathbf{bag}((\mathbf{tup}(K) \times X_1') \to X' \times \mathbf{bag}(\mathbf{tup}(\mathcal{O}')). \end{aligned}$$

For the semantics, suppose $t = \{v_1 \mapsto t_1, \ldots, v_m \mapsto t_m\}$, and for $i = 1, \ldots, m$, $P.\mathsf{init}(x, v_i) = x_i$, $[\![P.\mathsf{sub}]\!]_C(x_i, t_i) = (x_i', \langle t_i', r_i' \rangle)$, $P.\mathsf{agg}(x, \{(v_1, x_1), \ldots, (v_m, x_m)\}) = (x', r_0')$, and $[\![P.\mathsf{sub}]\!]_O(x_i, t_i) = \langle t_i'', r_i'' \rangle$. Then

$$\begin{aligned} [\![P]\!]_C(x, t) &= (x', \langle \{v_1 \mapsto t_1', \ldots, v_m \mapsto t_m'\}, r_0' \cup r_1' \cup \cdots \cup r_m' \rangle \\ [\![P]\!]_O(x, t) &= \langle \{v_1 \mapsto t_1'', \ldots, v_m \mapsto t_m''\}, r_1'' \cup \cdots \cup r_m'' \rangle. \end{aligned}$$

5.4. Examples

To illustrate the definition of the various SPST constructs, we continue the example schema from Figure 4.2 as the input type. For the output, suppose we want to produce two kinds of events: `EndOfHour`, representing end-of-hour summaries, and `Outlier`, representing outlier events that should be logged for further investigation. We describe building an SPST with this input and output, building it bottom-up from the structure of the input schema.

We begin with an example of a relational SPST. We describe the transformation on `RideCompleted` events which computes the sum of the costs of all completed hours. The interface of this SPST is $P_1 : ((), \mathsf{Bag}(\texttt{RideCompleted}), \texttt{float}, \emptyset)$. As it consumes a bag of `RideCompleted` events, it does not produce any output tuples, but instead we aggregate the sum of the return costs as a single `float`. For this relational base case, the computation can be written using an aggregator in a base

relational language such as SQL. Formally, in our framework P_1 is defined by two black-box functions: $P.\mathsf{open}((), r) = \bot$ and $P.\mathsf{closed}((), \{x_1, \ldots, x_m\}) = (x_1 + \cdots + x_m, \bot)$. The former component $P.\mathsf{open}$ indicates that in this case no events are produced incrementally (as the input stream is processed). The latter component $P.\mathsf{closed}$ indicates that the final result of the computation (after the entire input stream is seen) is the sum of all tuples in the input relation.

Next, we describe a simple sequential SPST which processes a linear sequence of `GPS` events. Recall that $\mathsf{Seq}(\mathcal{H})$ is a useful special case of hierarchical synchronization schemas that denote simple sequences, i.e., it is the schema $\mathsf{Sync}(\mathcal{H}, \mathsf{Bag}(\varnothing))$. Suppose we want to compute the distance traveled for a specific taxi given its `GPS` tuples; additionally, suppose we want to produce as output *outlier* `GPS` tuples, rather than including them in the aggregation.

$$P_2 : ((), \mathsf{Seq}(\texttt{GPS}), \texttt{float}, \mathsf{Seq}(\texttt{Outlier}))$$

P_2 keeps the last known location for the taxi and the current distance traveled as its state, and each time it processes a new `GPS` tuple, it updates both. Additionally, if the last known location is too far from the current one (> 1 below) instead of updating state it produces the tuple as output.

$$\begin{aligned}
P_2.\mathsf{update}((\bot, 0), gps) &= ((gps.loc, 0), \bot) \\
P_2.\mathsf{update}((loc, d), gps) &= ((gps.loc, d + dist(gps.loc, loc)) \\
&\qquad \text{if } dist(gps.loc, loc) \leq 1 \\
P_2.\mathsf{update}((loc, d), gps) &= ((loc, d), \texttt{Outlier}(loc)) \\
&\qquad \text{otherwise}
\end{aligned}$$

Because this is a sequential base case (a special case of hierarchical), $P_2.\mathsf{sub}$, $P_2.\mathsf{call}$, and $P_2.\mathsf{return}$ are trivial with no effect on the state. Finally, $P_2.\mathsf{init}() = (\bot, 0)$, and $P_2.\mathsf{fin}(loc, d) = (d, \bot)$.

Next, we define the partitioned SPST that computes the total distance traveled by all taxis (according to the taxi example described in Figure 4.2). The interface of the SPST is

$$P_3 : ((), \mathsf{ParBy}(\texttt{taxiID}, \mathsf{Seq}(\texttt{GPS})), \texttt{float}, \mathsf{Bag}(\texttt{Outlier}))$$

since it returns the total distance traveled by all taxis in miles. The child SPST is P_2, i.e., $P_3.\mathsf{sub} = P_2$, However, notice that instead of a sequential output, here the output outliers are a bag: this is because there are multiple keys (taxi IDs), so different key outputs may be unordered. Implicitly, we are relaxing the output of P_2 to be a bag instead of a sequence: this illustrates SPST *subtyping* (Definition 5.2.2), in which ordered output events may be reinterpreted as unordered. The interface of our sequential SPST is now $P_2 : \mathsf{Seq}(\texttt{GPS}), \texttt{float}, \mathsf{Seq}(\texttt{Outlier}))$. To fit the SPST definition exactly, we would additionally relax to $\mathsf{Par}(\varnothing, \mathsf{Seq}(\texttt{Outlier}))$ (to allow both keyed and bag outputs), but we leave this off for presentation; it is just another application of subtyping since the schemas are equivalent. To complete the definition of P_3, the aggregation produces a sum of the distances:

$$P_3.\mathsf{agg}(\_, ds) = (sum(\{d \mid (\_, d) \in ds\}), \bot)$$

and $P_3.\mathsf{init}$ initializes all child SPSTs with the unit value.

At this point, we have a partitioned SPST P_3 for processing the key-partitioned `GPS` stream, and we have a relational SPST P_1 for processing the `RideCompleted` events. In order to combine these into an overall query which also processes the `EndOfHour` synchronizing events, we first need to combine these two streams in parallel. We define an SPST P_4 which divides the aggregate cost by the aggregate distance. Let $S_1 = \mathsf{ParBy}(\texttt{taxiID}, \mathsf{Seq}(\texttt{GPS}))$ and $S_2 = \mathsf{Bag}(\texttt{RideCompleted})$. Then the interface of P_4 is

$$P_4 : ((), \mathsf{Par}(S_1, S_2), \texttt{float}, \mathsf{Bag}(\texttt{Outlier})).$$

The SPST calls the underlying SPSTs P_1 and P_3: $P_4.\mathsf{left} = P_3$ and $P_4.\mathsf{right} = P_1$, which return the total distance covered by all taxis and the total cost of all completed rides in that hour, and then simply divides to return the `float` ride cost per traveled mile, i.e., $P.\mathsf{fin}(dist, cost) = cost/dist$.

Notice that the average value in P_4 is only computed on finalization (after the entire stream is processed). In order to produce the same averages in a streaming manner, we need *synchronization events*, and this leads us to our final step: we complete the input schema in Figure 4.2 and the example by constructing a hierarchical schema which also processes the `EndOfHour` synchronization events. The schema P_5 which outputs the cost per distance traveled at the end of each hour has the

following interface:

$$P_5 : ((), \mathsf{Sync}(\texttt{EndOfHour}, \mathsf{Par}(S_1, S_2)),$$
$$(), \mathsf{Sync}(\texttt{CostPerMile}, \mathsf{Bag}(\texttt{Outlier})))$$

The SPST calls the underlying SPST P_4, i.e., $P_5.\mathsf{sub} = P_4$, which returns the cost per mile in the last hour as a `float`. P_4 also produces the `Outlier` output events. The internal state Y is the cost per mile from the last substream. The function $P_5.\mathsf{call}$ does not pass anything to P_4, but $P_5.\mathsf{return}$ does consume the final `float` and stores it in the state. Then P_5 simply outputs the `float` when processing an `EndOfHour` tuple:

$$P_5.\mathsf{update}(cpm, \_) = (cpm, \texttt{CostPerMile}(cpm)).$$

5.5. Monotonicity

The following is the major technical result of this chapter: for any SPST, the open semantics is monotone in the prefix relation on partially ordered streams. This ensures that event-by-event incremental processing is possible, though it does not define the event-by-event logic explicitly.

Theorem 5.5.1. Let $P : (X, S, X', S')$ be an SPST. Then P is monotone in the following sense: for any $x : X$ and $t, u : S$, if $t \preceq u$, then $[\![P]\!]_O(x, t) \preceq [\![P]\!]_O(x, u)$.

Proof. The proof is by induction on P. We strengthen the hypothesis to additionally show that the open semantics is a prefix of the closed semantics: if $[\![P]\!]_C(x, t) = (x', t')$ then $[\![P]\!]_O(x, t) \preceq t'$. In addition to the definition of concatenation $\circ$ and prefix $\preceq$, we use that $\preceq$ is a partial order (Proposition 4.8.5). One of the inductive cases is subtyping as given by Definition 5.2.2.

- In the relational case, $P.\mathsf{open}$ is monotonic and a subset relation of $P.\mathsf{closed}$ by assumption.
- In the parallel case, let $t = \langle t_1, t_2 \rangle$ and $u = \langle u_1, u_2 \rangle$, and suppose that we have $[\![P.\mathsf{left}]\!]_O(x_1, t_1) = t_1'$, $[\![P.\mathsf{left}]\!]_O(x_1, u_1) = u_1'$, $[\![P.\mathsf{right}]\!]_O(x_2, t_2) = t_2'$, and $[\![P.\mathsf{right}]\!]_O(x_2, u_2) = u_2'$. Applying the inductive hypothesis, what we need to show is that if $t_1' \preceq u_1'$ and $t_2' \preceq u_2'$, then $\langle t_1', t_2' \rangle \preceq \langle u_1', u_2' \rangle$. This follows by unfolding the definitions of prefix and underlying concatenation, which works

component-wise on $\langle t'_1, t'_2 \rangle$. The same reasoning applies to comparing the open and closed semantics.

- In the hierarchical case, our first step is to prove that the auxiliary semantics is monotonic. For this, we only consider when t and u each end in an empty sub-trace $t_m = \bot$ and $u_m = \bot$. This then follows by induction on the trace directly since the output is a sequence and produced one item at a time from the closed semantics of the sub-SPST, using transitivity of $\preceq$. Next for the general case, we observe the following: for any trace ending in an empty subtrace t, and any subtraces t_1, u_1 with $t_1 \preceq u_1$, the auxiliary semantics on t is a prefix of the open semantics on $t \circ [t_1]$ (by definition), which is a prefix of the open semantics on $t \circ [u_1]$ (by IH), which is a prefix of the auxiliary semantics on t concatenated with closed sub-SPST semantics on u_1 (by definition, IH, and associativity of concatenation), which is a prefix of the auxiliary semantics on $t \circ [u_1, d, \bot]$ for any d (by definition). This chain of prefix relations implies the general monotonicity for $t \preceq u$, using transitivity of $\preceq$. Also the auxiliary semantics on t concatenated with closed sub-SPST semantics on u_1 is a prefix of the closed semantics on $t \circ [u_1]$ (by definition), which gives that the open semantics is a prefix of closed.

- Next we consider the partition case. For the open semantics, $P.\mathsf{agg}$ does not factor in. We consider the output on $t \circ u$ and t in two parts: first the keyed output, and second the relational output. (i) For the keyed output, we need to show that the output on t is a prefix of the output on $t \circ u$. There are three cases here: the key is present in both t and u, present in only t, and present in only u. If present in both, the prefix relation holds by induction hypothesis. If only in t, the output on t and on $t \circ u$ are the same as these SPS are the same for this particular key value. If only in u, the output on t does not contain this particular key value, and so is a prefix of the output on $t \circ u$ taking u' to be the output on u for that key. (ii) For the relational output, we consider the set of key values in t: for each such value, the output on t and on $t \circ u$ produces a relation. We can ignore key values not in t (in $t \circ u$ only) as they only extend the output relation for $t \circ u$. Now we need to show that the relational output on $t \circ u$ is a superset of the relational output on t for each of these keys, which is true by induction hypothesis.

- Finally, we consider the case of subtyping (output schema relaxation). This requires careful application of Definition 13 from [2], Proposition 4.8.9 and Proposition 4.5.2. Using these we derive the following lemma: given a schema, $t' \preceq u'$ is equivalent to the following statement:

every flattening of t' can be extended to a flattening of u', and every flattening of u' is equivalent to a extension of a flattening of t'.

Given this lemma, let S and S' be the input and output schemas, and $S'' \lesssim S'$. Let t', u', t'', u'' be the output schemas for t and u: the definition of t'' and u'' is that all flattenings of t' are flattenings of t'', and all flattenings of u' are flattenings of u''. We also know by IH that $t' \preceq u'$, which we interpret in terms of flattenings by the lemma. Considering any flattening of t'', first we know it only contains events in $\mathbf{headers}(S')$ (because the original schema output was S'), and we can additionally show it is equivalent under S'' to some flattening of t'; this t' then can be extended to a flattening of u', so the flattening of t'' can be extended with the same extension to a flattening of u'', which by the lemma implies $t'' \preceq u''$. $\square$

CHAPTER 6 : Distribution

Deterministic ends should be accomplished with deterministic means.

—Edward A. Lee, 2006 [175]

This section considers the problem of automatic distribution of streaming operators. Rather than the structured representation of Chapter 5, distribution requires a more low-level representation because it requires knowing how an operator can be parallelized. In the chapter, we will define a programming model over streams called *dependency guided synchronization* and show how it can be used to parallelize stream operators automatically. The material in this section was originally published in [8].

Tying this into determinism are two key theorems for the chapter: Theorem 6.4.4, which states that for the programming model, *consistency implies determinism*; and Theorem 6.5.5, which states that the end-to-end system is correct and in particular, distribution is semantics-preserving. Together, these two results achieve type safety and determinism defined in Section 4.6, because they state formally that for all equivalent distributed input streams to the system, *and* for all possible choices of a distributed implementation, *and* for all possible executions of that distributed implementation, the output is the same up to reordering.

6.1. Motivation

The success of stream processing APIs based on the dataflow model can be attributed to their ability to simplify the task of parallel programming. To accomplish this (as described in Section 2.2.2), most APIs expose a simple but effective model of data-parallelism called *sharding* (*auto-parallelization*), in which nodes in the dataflow graph are replicated into many parallel instances, each of which will process a different partition of the input events. However, while sharding is intuitive for programmers, it also implicitly limits the scope of parallel patterns that can be expressed. Specifically, it prevents arbitrary *synchronization across parallel instances* since it disallows communication between them. This is limiting in modern applications such as video processing [157] and distributed machine learning [214], since they require both synchronization between nodes and high throughput and could therefore

benefit from parallelization. Further evidence that sharding is limiting in practice can be found in a collection of feature requests in state-of-the-art stream processing systems [122, 124, 123], asking either for state management that goes beyond replication or for some form of communication between shards. To address these needs, system developers have introduced extensions to the dataflow model to enable specific use cases such as *message broadcasting* and *iterative dataflows*. However, existing solutions do not generalize, as we demonstrate experimentally in Section 6.6.2. For the remainder of applications, users are left with two unsatisfying solutions: either ignore parallelization potential, implementing their application with limited parallelism; or circumvent the stream processing APIs using low-level external mechanisms to achieve synchronization between parallel instances.

For example, consider a fraud detection application where the input is a distributed set of streams of bank transaction events. Suppose we want to build an unsupervised online machine learning model over these events which classifies events as fraudulent based on a combination of *local* (stream-specific) and *global* (across-streams) statistical summaries. The problem with the traditional approach is that when classifying a new event, we need access to both the local and the global summaries; but this cannot be achieved using sharding since by default shards do not have access to a global summary. One extension to the dataflow model, implemented in some systems [115, 194] is the *broadcast* pattern, which allows the operator computing the global summary to broadcast to all other nodes. However, broadcasting is restricted since it does not allow bidirectional communication; the global summary needs to be both broadcast to all shards, but also updated by all shards. Cyclic dataflows are another partial solution, but do not always solve the problem, as we show in Section 6.6.2. In practice, applications like this one with complex synchronization requirements opt to manually implement the required synchronization using external mechanisms (e.g. querying a separate key-value store with strong consistency guarantees). This is error prone and, more importantly, violates the requirements of many streaming APIs that operators need to be effect-free so that the underlying system can provide exactly-once execution guarantees in the presence of faults.

6.2. Contributions

To address the need to combine parallelism with synchronization, we make two contributions. First, we propose *synchronization plans*, a tree-based execution model which is a restricted form of communicating sequential processes [149]. Synchronization plans are hierarchical structures that

represent concurrent computation in which parallel nodes are not completely independent, but communicate with their ancestors on special synchronizing events. While this solves the problem of being able to express synchronizing parallel computations, we still need a streaming API which exposes such parallelism implicitly rather than explicitly. For this purpose, we propose *dependency-guided synchronization* (DGS), a parallel programming model which can be mapped automatically to synchronization plans.

A DGS program consists of three components. First, the user provides a *sequential implementation* of the computation; this serves to define the semantics of what they want to compute assuming the input is processed as a sequence of events. Second, the user indicates which input events can be processed in parallel and which require synchronization by providing a *dependence relation* on input events. This relation induces a partial order on the input stream. For example, if events can be processed completely in parallel without any synchronization, then all input events can be specified to be independent. Third, the user provides a mechanism for parallelizing state when the input stream contains independent events: *parallelization primitives* called *fork* and *join*. This model is inspired by classical parallel programming, but has a streaming-specific semantics which describes how a partially ordered input stream is decomposed for parallel processing.

Given a DGS program, the main technical challenge is to generate a synchronization plan, which corresponds to a concrete implementation, that is both *correct* and *efficient*. More precisely, the challenge lies in ensuring that a derived implementation correctly enforces the specified input dependence relation. To achieve correctness, we formalize: (i) a set of conditions that ensure that a program is consistent, and (ii) a notion of P-*valid* synchronization plans, i.e., plans that are well-typed with respect to a given program P. To achieve efficiency, we design the framework so that correctness is independent of *which* synchronization plan is chosen—as long as it is P-valid. The idea of this separation is to enable future work on optimized query execution, in which an optimizing component searches for an efficient synchronization plan maximizing a desired cost metric without jeopardizing correctness. We tie everything together by proving that the end-to-end system is correct, that is, any concrete implementation that corresponds to a P-valid plan is equivalent to a program P that satisfies the consistency conditions.

In order to evaluate DGS, we perform a set of experiments to investigate the data parallelism limitations of Flink [62]—a representative high-performance stream processing system—and Timely

Dataflow [203]—a representative system with iterative computation. We show that these limits can be partly overcome by manually implementing synchronization. However, this comes at a cost: the code has to be specialized to the number of parallel nodes and similar implementation details, forcing the user to sacrifice the desirable benefit of *platform independence*. We then develop Flumina, an end-to-end prototype that implements DGS in Erlang [35, 269], and show that it can automatically produce scalable implementations (through generating synchronization plans from the program) independent of parallelism. In Section 6.7, we also evaluate programmability via two real-world case studies. In particular, we demonstrate that the effort required—as measured by lines of code—to achieve parallelism is minimal compared to the sequential implementation.

In summary, we make the following contributions:

- *DGS:* a novel programming model for parallel streaming computations that require synchronization, which allows viewing input streams as partially ordered sets of events. (Section 6.4)
- *Synchronization plans:* a tree-based execution model for parallel streaming computations that require synchronization, a framework for generating a synchronization plan given a DGS program, a prototype implementation, and an end-to-end proof of correctness. (Section 6.5)
- An evaluation that demonstrates: (i) the throughput limits of automatically scaling computations on examples which require synchronization in Flink and Timely; (ii) the throughput and scalability benefits achieved by synchronization plans over such automatically scaling computations; and (iii) the programmability benefits of DGS for synchronization-centered applications (Section 6.6).

Some of the material in this section (Sections 6.7 and 6.8) was published only in the extended version of the original paper [7], as an appendix. Flumina, our implementation of DGS, is open-source and available on GitHub[1].

6.3. System Architecture

Our solution can achieve data parallelism through the architecture summarized in Figure 6.1. The two primary abstractions (shown in blue) encode the required complex synchronization requirements

[1] https://github.com/angelhof/flumina

at different levels of abstractions: the DGS specification describes the computation and input dependencies in a platform-independent manner, and synchronization plans express the synchronization between processes at the implementation level, as communications between a hierarchically structured tree of processes.

The DGS specification is split in three parts. First, the user needs to provide a sequential implementation of the program, where the input is assumed to arrive in order and one event at a time. The sequential implementation consists of a stateful update function that can output events and update its state every time an input event is processed. For the fraud detection example, the update function would process bank transactions by checking if they are fraudulent and by constructing a sketch of the previously seen transactions, and fraud detection rules by using the sketch of previously seen transactions and the new rule to update the statistical fraud model. Second, the user provides a dependence relation that indicates the input events for which the processing order must be preserved, inducing a partial order on input events. For the current example, the user would simply indicate that fraud detection rule events depend on all other events. The final part of a specification consists of primitives that describe how to *fork* the state into two independent copies to allow for parallel processing and how to *join* two states when synchronization is required. These primitives abstractly represent splitting the computation into independent computations and merging the results, and are not tied to a specific implementation.

Given a DGS specification, the mapping to the synchronization plan in our architecture is given by a pluggable optimization component, which picks a synchronization plan based on information about the target execution environment, e.g., the number of processing workers and the location of the input streams. All of the induced plans are shown to be correct with respect to the sequential specification, so the optimizer is free to pick any of them without endangering correctness. As a starting point, we have developed a simple optimizer that tries to minimize the number of messages exchanged between different workers using information about the execution environment and the input streams. As a final step, the synchronization plan abstraction is deployed by the runtime system, which among other implementation details enforces the ordering of input events based on input dependencies, and is implemented in our DGSStream prototype.

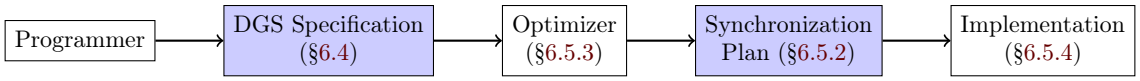


Figure 6.1: DGS system architecture.

6.4. Programming Model: Dependency-Guided Synchronization

A DGS program consists of three components: a *sequential implementation*, a *dependence relation* on input events to enforce synchronization, and *fork* and *join* parallelization primitives. In Section 6.4.3 we define program *consistency*, which consists of requirements on the *fork* and *join* functions to ensure that any parallel implementation generated from the program is equivalent to the sequential one.

6.4.1. DGS Programs

For a simple but illustrative example, suppose that we want to implement a stream processing application that simulates a map from keys to counters, in which there are two types of input events: *increment* events, denoted $\texttt{i}(k)$, and *read-reset* events, denoted $\texttt{r}(k)$, where each event has an associated key k. On each increment event, the counter associated with that key should be increased by one, and on each read-reset event, the current value of the counter should be produced as output, and then the counter should be reset to zero.

Sequential implementation. In our programming model, the user first provides a sequential implementation of the desired computation. A pseudocode version of the sequential implementation for the example above is shown in Figure 6.2 (left); Erlang syntax has been edited for readability, and we use `s[k]` as shorthand for the value associated with the key k in the map *or* the default value 0 if it is not present. The pseudocode consists of (i) the state type `State`, i.e. the map from keys to counters, (ii) the initial value of the state `init`, i.e. an empty map with no keys, and (iii) a function `update`, which contains the logic for processing input events. Conceptually, the sequential implementation describes how to process the data assuming it was all combined into a single sequential stream (e.g., sorted by system timestamp). For example, if the input stream consists of the events $\texttt{i}(1), \texttt{i}(2), \texttt{r}(1), \texttt{i}(2), \texttt{r}(1)$, then the output would be 1 followed by 0, produced by the two $\texttt{r}(1)$ (read-reset) events.

Dependence relation. To parallelize a sequential computation, the user needs to provide a dependence relation which encodes which events are independent, and thus can be processed in parallel, and which events are dependent, and therefore require synchronization. The dependence

```
// Types
Key = Integer
Event = i(Key) | r(Key)
State = Map(Key, Integer)
Pred = Event -> Bool

// Sequential Code
init: () -> State
init() =
    return emptyMap()

update: (State, Event)
        -> State
update(s, (i(k), ())) =
    s[k] = s[k] + 1;
    return s
update(s, (r(k), ())) =
    output s[k];
    s[k] = 0;
    return s
```

```
// Fork and Join
fork: (State, Pred, Pred)
        -> (State, State)
fork(s, pred1, pred2) =
    // two forked states
    s1 = init(); s2 = init()
    for k in keys(s):
        if pred1(r(k)):
            s1[k] = s[k]
        else:
            // pred2(r(k)) OR
            //   r(k) in neither
            s2[k] = s[k]
    return (s1, s2)

join: (State, State) -> State
join(s1, s2) =
    for k in keys(s2):
        s1[k] = s1[k] + s2[k]
    return s1
```

```
// Dependence Relation
depends: (Event, Event) -> Bool
depends(r(k1), r(k2)) = k1 == k2
depends(r(k1), i(k2)) = k1 == k2
depends(i(k1), r(k2)) = k1 == k2
depends(i(k1), i(k2)) = false
```

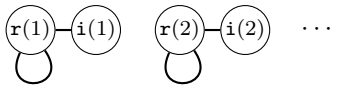


Figure 6.2: DGS program implementing a map from keys to counters. The `depends` relation is visualized as a graph with two keys shown; edges indicate synchronization, while non-edges indicate opportunities for parallelism.

relation abstractly captures all the dependency patterns that appear in an application, inducing a partial order on input events. In this example, there are two forms of independence we want to expose. To begin with, *parallelization by key* is possible: the counter map could be partitioned so that events corresponding to different sets of keys are processed independently. Moreover, each event is processed atomically in our model, and therefore *parallelizing increments* on the counter of the same key is also possible. In particular, different sets of increments for the same key can be processed independently; we only need to aggregate the independent counts when a read-reset operation arrives. On the other hand, read-reset events are synchronizing for a particular key; their output is affected by the processing of increments as well as other read-reset events of that key.

We capture this combination of parallelization and synchronization requirements by defining the dependence relation `depends` in Figure 6.2 (also visualized as a graph) (see Section 6.4.2 for a formal definition). In the program, the set of events may be *symbolic* (infinite): here `Event` is parameterized by an integer `Key`. To allow for this, the dependence relation is formally a *predicate* on pairs of events, and is given programmatically as a function from pairs of `Event` to `Bool`. For example, `depends(r(k1), r(k2))` (one of four cases) is given symbolically as equality comparison of keys, `k1 == k2`. The dependence relation should also be *symmetric*, i.e. `e1` is in `depends(e2)` iff `e2` is in `depends(e1)`; the intuition is that `e1` can be processed in parallel with `e2` iff `e2` can be processed in parallel with `e1`.

Parallelization primitives: *fork* and *join*. While the dependence relation indicates the possibility of parallelization, it does not provide a mechanism for parallelizing state. The parallelization is specified using a pair of functions to `fork` one state into two, and to `join` two states into one. The fork function additionally takes as input two predicates on events, such that the two predicates are *independent* (but not necessarily disjoint): every event satisfying `pred1` is independent of every event satisfying `pred2`. The contract is that after the state is forked into two independent states, each state will then *only be updated using events satisfying the given predicate.* A fork-join pair for our example is shown in Figure 6.2. The `join` function simply adds up the counts for each key to form the combined state. The `fork` function has to decide, for each key, which forked state to partition the count to. Since read-reset operations `r(k)` are synchronizing, i.e., depend on all events of the same key, and require knowing the total count, it partitions by checking which of the two forked states is responsible for processing read-reset operations, if any.

The programming model *exposes* parallelism, but the implementation (Section 6.5) determines when to call forks and joins. To do this, the implementation instantiates a synchronization plan: a tree structure where each node is a stateful worker with a predicate indicating the set of events that it is responsible for. Nodes that do not have an ancestor-descendant relationship process independent but not necessarily disjoint sets of events. When a node with children needs to process an event, it first uses `join` to merge the states of its children, and then it `fork`s back its children states using the combined predicates of its descendants, `pred1` for the left subtree, and `pred2` for the right subtree. The implementation can therefore instantiate synchronization plans with different shapes and predicates to enable different kinds of parallelism. For example, to indicate *parallelization by key*, the left child with `pred1` might contain all events of key 1 and the right child with `pred2` might contain all events of key

2. On the other hand, to indicate *parallelization on increments*, `pred1` and `pred2` might both contain `i(3)`, and in this case neither would contain `r(3)` (to satisfy the independence requirement). The latter example also emphasizes that `pred1` and `pred2` need not be disjoint, nor need they collectively cover all events. For the events not covered, in this case `r(3)`, a join would need to be called before an `r(3)` event can be processed. Parallelization can also be done repeatedly; the fork function can be called again on a forked state to fork it into two sub-states, and each time the predicates `pred1` and `pred2` will be even further restricted.

6.4.2. Formal Definition

A DGS program can be more general than we have discussed so far, because we allow for multiple *state types*, instead of just one. The initial state must be of a certain type, but forks and joins can convert from one state type to another: for example, forking a pair into its two components. Additionally, each state type can come with a *predicate* which restricts the allowed events processed by a state of that type. The complete programming model is summarized in the following definition.

Definition 6.4.1 (DGS program)**.** Let `Pred(T)` be a given type of *predicates* on a type `T`, where predicates can be evaluated as functions `T -> Bool`. A program consists of the following components:

1. A type of input events `Event`.
2. The dependence relation `depends: Pred(Event, Event)`, which is symmetric: `depends(e1, e2)` iff `depends(e2, e1)`.
3. A type for output events `Out`.
4. Finitely many state types `State_0`, `State_1`, etc.
5. For each state type `State_i`, a predicate which specifies which input values this type of state can process, denoted `pred_i: Pred(Event)`. We require `pred_0` = `true`.
6. A sequential implementation, consisting of a single initial state `init: State_0` and for each state type `State_i`, a function `update_i: (State_i, Event) -> State_i`. The update also produces zero or more outputs, given by a function `out_i: (State_i, Event) -> List(Out)`.

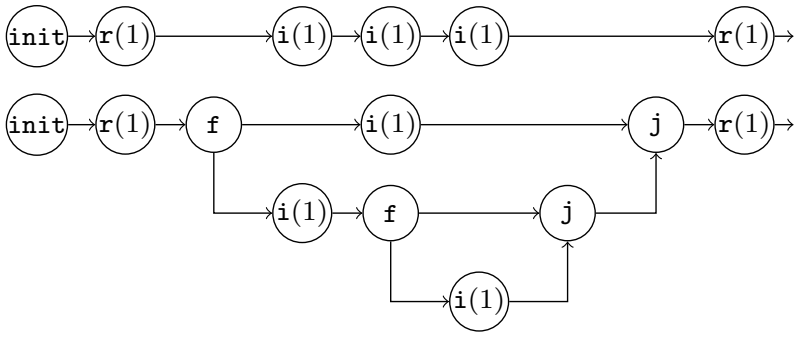


Figure 6.3: Example of a sequential (top) and parallel (bottom) execution of the program in Figure 6.2 on the input stream $\mathtt{r}(1), \mathtt{i}(1), \mathtt{i}(1), \mathtt{i}(1), \mathtt{r}(1)$ (`f` and `j` denote forks and joins).

7. A set of parallelization primitives, where each is either a *fork* or a *join*. A fork has type

```
(State_i, Pred(Event), Pred(Event)) -> (State_j, State_k),
```

   and a join has type `(State_j, State_k) -> State_i`, for some i, j, and k.

Semantics. The semantics of a program can be visualized using wire diagrams, as in Figure 6.3. Computation proceeds from left to right. Each wire is associated with (i) a state (of type `State_i` for some i) and (ii) a predicate (of type `Pred(Event)`) which restricts the input events that this wire can process. Input events are processed as updates to the state, which means they take one input wire and produce one output wire, while forks take one input wire and produce two, and joins take two input wires and produce one. Notice that the same updates are present in both sequential and parallel executions. It is guaranteed in the parallel execution that *fork and join come in pairs*, like matched parentheses. Each predicate that is given as input to the `fork` function indicates the set of input events that can be processed along one of the outgoing wires. Additionally, we require that updates on parallel wires must be on independent events. In the example, the wire is forked into two parts and then forked again, and all three resulting wires process $\mathtt{i}(1)$ events. Note that $\mathtt{r}(1)$ events cannot be processed at that time because they are dependent on $\mathtt{i}(1)$ events. More specifically, we require that the predicate at each wire of type `State_i` implies `pred_i`, and that after each `fork` call, the predicates at each resulting wire denote independent sets of events. This semantics is formalized in the following definition.

Definition 6.4.2 (DGS Semantics)**.** A *wire* is a triple using the notation $\langle \mathtt{State\_i}, \mathtt{pred}, \mathtt{s} \rangle$, where `State_i` is a state type, `s: State_i`, and `pred: Pred(Event)` is a predicate such that `pred` implies `pred_i`.

We give the semantics of a program through an inductively defined relation, which we denote $\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle$, where $\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle$ and $\langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle$ are the starting and ending wires (with the same state type and predicate), `u: List(Event)` is an input stream, and `v: List(Out)` is an output stream. Let `l1 + l2` be list concatenation and let `inter(l, l1, l2)` be list interleaving, as defined in Chapter 4. For `e1, e2: Event`, let `indep(e1, e2)` denote that `e1` and `e2` are not dependent, i.e. `not(depends(e1,e2))`. There are two base cases and two inductive cases.

(1) For any `State`, `pred`, `s`,

$$\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{[]}]{\texttt{[]}} \langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle.$$

(2) For any `State`, `pred`, `s`, and any `e: Event`, if `e` satisfies `pred` then

$$\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{out(s, e)}]{\texttt{[e]}} \langle \texttt{State}, \texttt{pred}, \texttt{update(s, e)} \rangle.$$

(3) For any `State`, `pred`, `s`, `s'`, `s''`, `u`, `v`, `u'`, and `v'`, if $\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle$ and $\langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle \xrightarrow[\texttt{v'}]{\texttt{u'}} \langle \texttt{State}, \texttt{pred}, \texttt{s''} \rangle$, then

$$\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{v + v'}]{\texttt{u + u'}} \langle \texttt{State}, \texttt{pred}, \texttt{s''} \rangle.$$

(4) Lastly, for any instances of `State`, `State1`, `State2`, `pred`, `pred1`, `pred2`, `s`, `s1'`, `s2'`, `u`, `u1`, `u2`, `v`, `v1`, `v2`, `fork`, and `join`, suppose that (the conjunction) `pred1(e1)` and `pred2(e2)` implies `indep(e1, e2)`, `pred1` implies `pred`, and `pred2` implies `pred`. Let `fork(s, pred1, pred2) = (s1, s2)` and `join(s1', s2') = s'`. If we have `inter(u, u1, u2)`, `inter(v, v1, v2)`, $\langle \texttt{State1}, \texttt{pred1}, \texttt{s1} \rangle \xrightarrow[\texttt{v1}]{\texttt{u1}} \langle \texttt{State1}, \texttt{pred1}, \texttt{s1'} \rangle$, and $\langle \texttt{State2}, \texttt{pred2}, \texttt{s2} \rangle \xrightarrow[\texttt{v2}]{\texttt{u2}} \langle \texttt{State2}, \texttt{pred2}, \texttt{s2'} \rangle$, then

$$\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle.$$

Finally, the *semantics* $[\![P]\!]$ of the program P is the set of pairs $(\texttt{u}, \texttt{v})$ of an input stream `u` and an output stream `v` such that $\langle \texttt{State\_0}, \texttt{true}, \texttt{init} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State\_0}, \texttt{true}, \texttt{s'} \rangle$ for some `s'`.

Representing predicates. In the running example, a predicate on a type `T` was represented as a function `T -> Bool`, but note that the programming model above allows other representation of

predicates, for example using logical formulas. The tradeoff here is that a more general representation allows more dependence relations to be expressible, but also complicates the implementation of an appropriate `fork` function as it must accept as input more general input predicates. In our implementation (see Section 6.5), we assume that an event consists of a pair of a `Tag` (relevant for parallelization) and a `Payload` (used only for processing), where predicates are given as sets of tags (or pairs of tags, for `depends`). This allows simpler logic in the fork function whose input predicates are then `Tag -> Bool` and don't depend on the irrelevant payload. In our example, $\texttt{i}(k)$ or $\texttt{r}(k)$ would be tags (not payload) as they are relevant for parallelization.

6.4.3. Consistency Conditions

Any parallel execution is guaranteed to preserve the sequential semantics, i.e. processing all input events *in order* using the `update` function, as long as the following *consistency conditions* are satisfied. The sufficiency of these conditions is shown in Theorem 6.4.4, which states that *consistency implies determinism up to output reordering.* This is a key step in the end-to-end proof of correctness in Section 6.5.5. Consistency can be thought of as analogous to the commutativity and associativity requirements for a MapReduce program to have deterministic output [94]: just as with MapReduce programs, the implementation does *not* assume the conditions are satisfied, but if not the semantics will be dependent on how the computation is parallelized.

Definition 6.4.3 (Consistency)**.** A program is *consistent* if the following equations always hold.

$$\texttt{join(update(s1,e),s2)} = \texttt{update(join(s1,s2),e)} \tag{C1}$$

$$\texttt{join(fork(s,pred1,pred2))} = \texttt{s} \tag{C2}$$

$$\texttt{update(update(s,e1),e2))} = \texttt{update(update(s,e2),e1))} \tag{C3}$$

subject to the following additional qualifications. First, equation (C1) is over all the join functions `join: (State_j, State_k) -> State_i`, events `e: Event` such that `pred_i(e)` and `pred_j(e)`, and states `s1: State_j`, `s2: State_k`, where `update` denotes the update function on the appropriate type. Additionally the corresponding output on both sides must be the same:

$$\texttt{out(s1, e)} = \texttt{out(join(s1, s2), e).} \tag{C4}$$

Equation (C2) is over all fork functions `fork: (State_i,Pred(Event),Pred(Event)) -> (State_j, State_k)`, all joins `join: (State_j, State_k) -> State_i`, states `s: State_i`, and predicates `pred1` and `pred2`. Equation (C3) is over all state types `State_i`, states `s: State_i`, and pairs of *independent* events `indep(e1, e2)` such that `pred_i(e1)` and `pred_i(e2)`. As with (C1), we also require that the outputs on both sides agree:

$$\texttt{out(s, e1) + out(update(s, e1), e2)} \;=\; \texttt{out(update(s, e2), e1) + out(s, e2).} \tag{C5}$$

Let us illustrate the consistency conditions for our running example (Figure 6.2). If `e` is an increment event, then condition (C1) captures the fact that counting can be done in parallel: it reduces to `(s1[k] + s2[k]) + 1 = (s1[k] + 1) + s2[k]`. Condition (C2) captures the fact that we preserve total count across states when forking: it reduces to `s[k] + 0 = s[k]`. Condition (C3) would not be valid for *general* events `e1`, `e2`, because a read-reset event does not commute with an increment of the same key (`s[k] + 1` $\neq$ `s[k]`), hence the restriction that `indep(e1, e2)`. Finally, one might think that a variant of (C1) should hold for `fork` in addition to `join`, but this turns out not to be the case: for example, starting from `s[k] = 100`, an increment followed by a fork might yield the pair of counts $(101, 0)$, while a fork followed by an increment might yield $(100, 1)$. It turns out however that commutativity only with joins, and not with forks, is enough to imply Theorem 6.4.4.

Theorem 6.4.4. If P is consistent, then P is deterministic up to output reordering. That is, for all $(\texttt{u}, \texttt{v}) \in [\![P]\!]$, the multiset of events in stream `v` is equal to the multiset of events in `spec(u)` where `spec` is the semantics of the sequential implementation.

Proof. We show by induction on the semantics in Definition 6.4.2 that every wire diagram is equivalent (up to output reordering) to the sequential sequence of updates. The sequential inductive step (3) is direct by associativity of function composition on the left and right sequence of updates (no commutativity of updates is required). For the parallel inductive step (4), we replace the two parallel wires with sequential wires, then apply (C1) repeatedly on the last output to move it outside of the parallel wires, then finally apply (C2) to reduce the now trivial parallel wires to a single wire. □

6.5. Execution Model: Synchronization Plans

In this section we describe *synchronization plans*, which represent streaming program implementations, and our framework for generating them from the given DGS program in Section 6.4. Generation of an implementation can be conceptually split in two parts, the first ensuring correctness and the second affecting performance. First a program P induces a set of P-*valid*, i.e. correct with respect to it, synchronization plans. Choosing one of those plans is then an independent optimization problem that does not affect correctness and can be delegated to a separate optimization component (Section 6.5.3). Finally, the workers in synchronization plans need to process some incoming events in order while some can be processed out of order (depending on the dependence relation). We propose a selective reordering technique (Section 6.5.4) that can be used in tandem with heartbeats to address this ordering issue. We tie everything together by providing an end-to-end proof that the implementation is correct with respect to a consistent program P (and importantly, independent of the synchronization plan chosen as long as it is P-valid) in Section 6.5.5. Before describing the separate framework components, we first articulate the necessary underlying assumptions about input streams in Section 6.5.1.

6.5.1. Preliminaries

In our model the input is partitioned in some number of input streams that could be distributed, i.e. produced at different locations. We assume that the implementation has access to *some* ordering relation $\mathcal{O}$ on pairs of input events (also denoted $<_{\mathcal{O}}$), and the order of events is increasing along each input stream. This is necessary for cases where the *user-written* program requires that events arriving in different streams are dependent, since it allows the implementation to progress by processing these dependent events in order. Concretely, in our setting $\mathcal{O}$ is implemented using *event timestamps*. Note that these timestamps do not need to correspond to real time, if this is not required by the application. In cases where real-time timestamps are required, this can be achieved with well-synchronized clocks, as has been done in other systems, e.g. Google Spanner [84].

Each event in each input stream is given by a quadruple $\langle tg, id, ts, v \rangle$, where tg is a *tag* used for parallelization, id is a unique identifier of the input stream, ts is a *timestamp*, and v is a *payload*. Of these, only the tag and payload are visible to the programming model in Section 6.4, and only the tag is used in predicates and in the dependence relation. Our implementation currently requires that

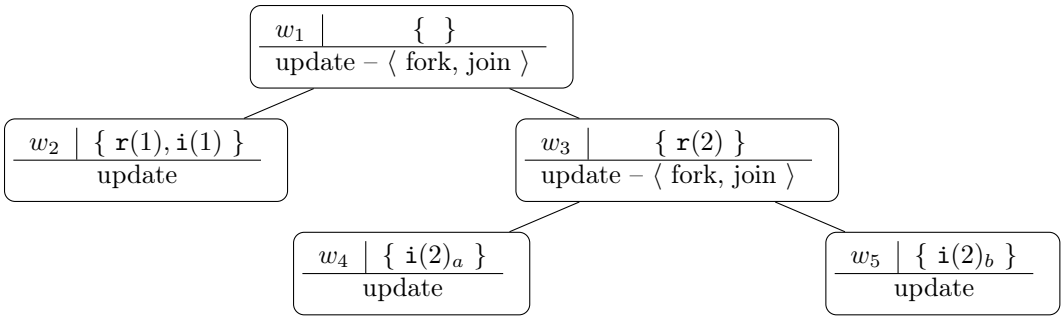


Figure 6.4: Example synchronization plan derived from the program in Figure 6.2 for two keys $k = 2$ and five input streams $\mathtt{r}(1), \mathtt{i}(1), \mathtt{r}(2), \mathtt{i}(2)_a, \mathtt{i}(2)_b$. Implementation tags $\mathtt{i}(2)_a, \mathtt{i}(2)_b$ both correspond to $\mathtt{i}(2)$ events but are separate because they arrive in different input streams.

the number of possible tags tg is finite (e.g. up to some maximum key) as well as the number of identifiers id.

For the rest of this section, we write events as $\langle \sigma, ts, v \rangle$ where the pair $\sigma = \langle tg, id \rangle$ is called the *implementation tag*. This is a useful distinction because at the implementation level, these are the two components that are used for parallelization. The relation `depends: (Tag, Tag) -> Bool` in the program straightforwardly lifts to predicates over tags and to implementation tags.

6.5.2. Synchronization Plans

Synchronization plans are binary tree structures that encode (i) parallelism: each node of the tree represents a sequential thread of computation that processes input events; and (ii) synchronization: parents have to synchronize with their children to process an event. Synchronization plans are inspired by prior work on concurrency models including fork-join concurrency [130, 174] and CSP [149]. An example synchronization plan for the program in Figure 6.2 is shown in Figure 6.4. Each node has an id w_i, contains the set of implementation tags that it is responsible for, a state type (which is omitted here since there is only one state type `State`), and a triple of update, fork, join functions. Note that a node is responsible to process events from its set of implementation tags, but can potentially handle all the implementation tags of its children. The leaves of the tree can process events independently without blocking, while parent nodes can only process an input event if their children states are joined. Nodes without a ancestor-descendant relationship do not directly communicate, but instead learn about each other when their common ancestor joins and forks back the state.

Definition 6.5.1 (Synchronization Plans)**.** Given a program P, a synchronization plan is a pair $(\overline{w}, \text{par})$, which includes a set of workers $\overline{w} = \{w_1, \ldots, w_N\}$, together with a parent relation $\text{par} \subseteq \overline{w} \times \overline{w}$, the transitive closure of which is an ancestor relation denoted as $\text{anc} \subseteq \overline{w} \times \overline{w}$. Workers have three components: (i) a state type $w.\mathsf{state}$ which references one of the state types of P, (ii) a set of implementation tags $w.\mathsf{itags}$ that the worker is responsible for, and (iii) an update $w.\mathsf{update}$ and possibly a fork-join pair $w.\mathsf{fork}$ and $w.\mathsf{join}$ if it has children.

We now define what it means for a synchronization plan to be *P-valid*. Intuitively, an P-valid plan is well-typed with respect to program P, and the workers that do not have an ancestor-descendant relationship should handle independent and disjoint implementation tags. P-validity is checked syntactically by our framework and is a necessary requirement to ensure that the generated implementation is correct (see Section 6.5.5).

Definition 6.5.2 (P-valid)**.** Formally, a P-valid plan has to satisfy the following syntactic properties: (V1) The state `State_i` $= w.\mathsf{state}$ of each worker w should be consistent with its update-fork-join triple and its implementation tags. The update must be defined on the node state type, i.e., $w.\mathsf{update}$: `(State_i, Event) -> State_i, State_i` should be able to handle the tags corresponding to $w.\mathsf{itags}$, and the fork-join pair should be defined for the state types of the node and its children. (V2) Each pair of nodes that do not have an ancestor-descendant relation, should handle pairwise independent and disjoint implementation tag sets, i.e., $\forall w, w' \notin \text{anc}(w, w'), w.\mathsf{itags} \cap w'.\mathsf{itags} = \emptyset \wedge \text{indep}(w.\mathsf{itags}, w'.\mathsf{itags})$.

As an example, the synchronization plan shown in Figure 6.4 satisfies both properties; there is only one state type that handles all tags and implementation tag sets are disjoint for ancestor-descendants. The second property (V2) represents the main idea behind our execution model; independent events can be processed by different workers without communication. Intuitively, in the example in Figure 6.4, by assigning the responsibility for handling tag $\mathtt{r}(2)$ to node w_3, its children can independently process tags $\mathtt{i}(2)_a, \mathtt{i}(2)_b$ that are dependent on $\mathtt{r}(2)$.

6.5.3. Optimization Problem

As described in the previous section, a set of P-valid synchronization plans can be derived from a DGS program P. This decouples the optimization problem of finding a well-performing implementation, allowing it to be addressed by an independent optimizer, which takes as input a description of the

available computer nodes and the input streams. This design means that different optimizers could be implemented for different performance metrics (e.g. throughput, latency, network load, energy consumption). The design space for optimizers is vast and thoroughly exploring it is outside of the scope of this work. For evaluation purposes, we have implemented a few simple optimizers, one of which tries to minimize communication between workers by placing them close to their inputs. Intuitively, it searches for divisions of the event tags into two sets such that those sets are "maximally" independent, using those sets of tags for the initial fork, and then recursing on each independent subset. Its design is described in more detail in Section 6.8.1.

6.5.4. Implementation

Each node of the synchronization plan can be separated into two components: an event-processing component responsible for executing `update`, `fork`, and `join` calls; and a mailbox component responsible for enforcing ordering requirements and synchronization.

Event processing. The worker processes execute the functions (`update`, `fork`, and `join`) associated with the tree node. Whenever a worker is handed a message by its mailbox, it first checks if it has any active children, and if so, it sends them a join request and waits until it receives their responses. After receiving these responses, it executes the join function to combine their states, executes the update function on the received event, and then executes the fork function on the new state, and sends the resulting states to its children. In contrast, a leaf worker just executes the update function on the received event.

Event reordering. The mailbox of each worker ensures that it processes incoming dependent events in the correct order by implementing the following selective reordering procedure. Each mailbox contains an event buffer and a timer for each implementation tag. The buffer holds the events of a specific tag in increasing order of timestamps and the timer indicates the latest timestamp that has been encountered for each tag. When a mailbox receives an event $\langle \sigma, ts, v \rangle$ (or a join request), it follows the procedure described below. It first inserts it in the corresponding buffer and updates the timer for σ to the new timestamp ts. It then initiates a cascading process of releasing events with tags σ' that depend on σ. During that process all dependent tags σ' are added to a dependent tag workset, and the buffer of each tag in the workset is checked for potential events to release. An event $e = \langle \sigma, ts, v \rangle$ can be released to the worker process if two conditions hold. The timers of its dependent tags are higher than the timestamp ts of the event (which means that the mailbox has

already seen all dependent events up to ts, making it safe to release e), and the earliest event in each buffer that σ depends on should have a timestamp $ts' > ts$ (so that events are processed in order). Whenever an event with tag σ is released, all its dependent tags are added to the workset and this process recurses until the tag workset is empty.

Heartbeats. As discussed in Section 6.5.1, a dependence between two implementation tags σ_1 and σ_2 requires the implementation to process any event $\langle \sigma_1, t_i, v_i \rangle$ after processing all events $\langle \sigma_2, t_j, v_j \rangle$ with $t_j \leq t_i$. However, with the current assumptions on the input streams, a mailbox has to wait until it receives the earliest event $\langle \sigma_2, t_j, v_j \rangle$ with $t_j > t_i$, which could arbitrarily delay event processing. We address this issue by periodically generating *heartbeat* events at each producer, which are system events that represent the absence of events on a stream. Heartbeats are interleaved together with standard events of input streams. When a heartbeat event $\langle \sigma, t \rangle$ first enters the system, it is broadcast to all the worker processes that are descendants of the worker that is responsible for tag σ. Each mailbox that receives the heartbeat updates its timers and clears its buffers as if it has received an event of $\langle \sigma, t, v \rangle$ without adding the heartbeat to the buffer to be released to the worker process. Similar mechanisms are often used in other stream processing systems under various names, e.g. heartbeats [162], punctuation [260], watermarks [62], or pulses [238].

In our experience, heartbeat rates are successful in improving the latency of the system unless they are configured to be very large or very low values. For a wide range of heartbeat values (about 10-1000 per synchronization event), the performance of the system is stable and exhibits minor variance (see more details in Section 6.8.2.

6.5.5. Proof of Correctness

We show that *any* implementation produced by the end-to-end framework is correct according to the semantics of the programming model (Theorem 6.5.5). First, Definition 6.5.3 formalizes the assumptions about the input streams outlined in Section 6.5.1, and Definition 6.5.4 defines what it means for an implementation to be correct with respect to a sequential specification. Our definition is inspired by the classical definitions of distributed correctness based on observational trace semantics (e.g., [185]). However, we focus on how to interpret the independent input streams as a sequential input, in order to model possibly synchronizing and order-dependent stream processing computations.

Definition 6.5.3. A *valid input instance* consists of k input streams (finite sequences) `u_1`, `u_2`, ..., `u_k` of type `List(Event | Heartbeat)`, and an order relation $\mathcal{O}$ on input events and heartbeats, with the following properties. (1) *Monotonicity:* for all i, `u_i` is in strictly increasing order according to $<_{\mathcal{O}}$. (2) *Progress:* for all i, for each non-heartbeat input event `x` in `u_i`, for every other stream j there exists an event or heartbeat `y` in `u_j` such that `x` $<_{\mathcal{O}}$ `y`.

Given a DGS program P, the *sequential specification* is a function `spec: List(Event) -> List(Out)`, derived from the sequential implementation by applying only `update` and no `fork` and `join` calls. The output specified by `spec` is produced incrementally (or *monotonically*): if `u` is a prefix of `u'`, then `spec(u)` is a subset of `spec(u')`. Define the *sort* function $\texttt{sort}_{\mathcal{O}}$: `List(List(Event | Heartbeat)) -> List(Event)` which takes k sorted input event streams and sorts them into one sequential stream, according to the total order relation $\mathcal{O}$, and drops heartbeat events.

Definition 6.5.4. A distributed implementation is *correct* with respect to a given sequential specification `spec: List(Event) -> List(Out)`, if for every valid input instance $\mathcal{O}$, `u_1`, ..., `u_k`, the set of outputs produced by the implementation is equal to $\texttt{set}(\texttt{spec}(\texttt{sort}_{\mathcal{O}}(\texttt{u\_1}, \ldots, \texttt{u\_k})))$.

Theorem 6.5.5 (implementation correctness)**.** Any implementation produced by our framework is correct according to Definition 6.5.4.

To prove Theorem 6.5.5, the key assumptions used are:

(1) The program is consistent, as defined in Section 6.4.3.

(2) The input streams constitute a valid input instance, as defined in Definition 6.5.3.

(3) The synchronization plan that is chosen by the optimizer is valid, as defined in Section 6.5.2.

(4) Messages that are sent between two processes in the system arrive in order and are always received. This last assumption is ensured by the Erlang runtime.

(5) The underlying scheduler is fair, i.e. all processes get scheduled infinitely often.

The proof decomposes the implementation into the mailbox implementations and the worker implementations, which are both sets of processes that given a set of input streams produce a set of output streams. We first show that the mailboxes transform a valid input instance to a worker input

that is correct up to reordering of independent events. Then we show that given a valid worker input, the workers processes' collective output is correct. The second step relies on Theorem 6.4.4 as well as Lemma 6.5.8 which ties this to the mailbox implementations, showing that the implementation produces a valid wire diagram according to the formal semantics of the program P. Given a valid input instance u and a computation program P that contains in particular a sequential specification `spec: List(Event) -> List(Out)`, we want to show that any implementation f that is produced by our framework is correct according to Definition 6.5.4.

Definition 6.5.6. For a valid input instance u, a worker input $m_u = (m_1, ..., m_N)$ is *valid* with respect to u if $m_i \in \text{reorder}_{D_I}(\text{filter}(\text{rec}_{w_i}, \text{sort}_O(u)))$, where reorder_{D_I} is the set of reorderings that preserve the order of dependent events, $\text{rec}_w(\langle \sigma, t, p \rangle) = \sigma \in \text{atags}(w) \cup w.\mathsf{itags}$ is a predicate of the messages that each worker w receives, and atags is the set of implementation tags that are handled by a workers ancestors, and is given by $\text{atags}_w = \{w'.\mathsf{itags} : \forall w' \in \text{anc}(w', w)\}$.

Lemma 6.5.7. Given a valid input instance u, any worker input m produced by the mailbox implementations (u, m) in f is valid with respect to u.

Proof. By induction on the input events of one mailbox and using assumptions (2)-(5). □

Each worker w runs the update function on each event $e = \langle \sigma, t, p \rangle$ on its stream that it is responsible for $\sigma \in w.\mathsf{itags}$ possibly producing output `o: List(Out)`. For all events in the stream that one of its ancestors is responsible for, it sends its state to its parent worker and waits until it receives an updated one. The following key lemma states that this corresponds to a particular wire diagram in the semantics of the program.

Lemma 6.5.8. Let m be the worker input to f for program P on input `u`, and let $\texttt{o}_i$: `List(Out)` be the stream of output events produced by worker i on input m_i. Then there exists `v: List(Out)` such that $\texttt{inter}(\texttt{v}, \texttt{o}_1, \texttt{o}_2, \ldots, \texttt{o}_N)$ and $(\texttt{u}, \texttt{v}) \in [\![S]\!]$.

Proof. By induction on the worker input and using assumption (3), in particular validity condition (V1), we first show that the worker input corresponds to a wire diagram, in particular we show that $\langle \texttt{State\_0}, \texttt{true}, \texttt{s} \rangle \xrightarrow[\texttt{v}]{\texttt{u}'} \langle \texttt{State\_0}, \texttt{true}, \texttt{s'} \rangle$ where `v` is an interleaving of $\texttt{o}_1, \ldots, \texttt{o}_N$ and $\texttt{u}'$ is *any* interleaving of the events $\texttt{u}'_i$ processed by each mailbox, namely $\text{filter}(\text{rec}_{w_i}, w_i.\mathsf{itags})$. Applying Lemma 6.5.7,

$\mathtt{u}$ is one possible interleaving of the events $\mathtt{u}'_i$ and hence we conclude that $\langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s} \rangle \xrightarrow[\mathtt{v}]{\mathtt{u}} \langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s'} \rangle$, thus $(\mathtt{u}, \mathtt{v}) \in [\![S]\!]$. □

Combining Lemma 6.5.8 and Theorem 6.4.4 then yields the end-to-end correctness theorem Theorem 6.5.5.

6.6. Experimental Evaluation

In this section we conduct a set of experiments to investigate tradeoffs between data parallelism and *platform independence* in stream processing APIs. That is, we want to distinguish between parallelism that is achieved automatically and parallelism that is achieved manually at the cost of portability when the details of the underlying platform change. To frame this discussion, we identify a set of platform independence principles (PIP) with which to evaluate this tradeoff:

PIP1: ***parallelism independence.*** Is the program developed without regard to the number of parallel instances, or does the program use the number of parallel instances in a nontrivial way?

PIP2: ***partition independence.*** Is the program developed without regard to the correspondence between input data and parallel instances, or does it require knowledge of how input streams are partitioned to be correct?

PIP3: ***API compliance.*** Does the program violate any assumptions made by the stream processing API?

Having identified these principles, the following questions guide our evaluation:

Q1 For computations requiring synchronization, what are the throughput limits of automatic parallelism exposed by existing stream processing APIs?

Q2 Can *manual* parallel implementations, i.e., implementations that may sacrifice **(PIP1–3)** above, that emulate synchronization plans achieve *absolute* throughput improvements in existing stream processing systems?

Q3 What is the throughput scalability of the synchronization plans that are generated automatically by our framework?

Q4 In summary, for each method of achieving data parallelism with synchronization, what platform independence tradeoffs are made?

In order to study these questions, we design three applications with synchronization requirements in Section 6.6.1. Our investigation compares three systems at different points in the implementation space with varying APIs and performance characteristics. First, Apache Flink [115, 62] represents a well-engineered mainstream streaming system with an expressive API. Second, the Rust implementation of Timely Dataflow [194, 203] (Timely) represents a system with support for iterative computation. Third, Flumina is a prototype implementation of our end-to-end framework that supports the communication patterns observed in arbitrary synchronization plans, some of which are not supported by the execution models of Flink and Timely.

Experimental setup. We conduct all the experiments in this section in a distributed execution environment consisting of AWS EC2 instances. To account for the fact that AWS instances can introduce variability in results, we chose m6g medium (1 core @2.5GHz, 4 GB) instances, which do not use burst performance like free-tier instances. We use instances in the same region (us-east-2) and we increase the number of instances for the scalability experiments. Communication between nodes is managed by each respective system (the system runtime for Flink and Timely, and Erlang for Flumina).

We configure Flink to be in true streaming mode by disabling batching (setting buffer-timeout to 0), checkpointing, and dynamic adaptation. For Timely, it is inherent to the computational model that events are batched by logical timestamp, and the system is not designed for event-by-event streaming, so our data generators follow this paradigm. This results in significantly higher throughputs for Timely, but note that these throughputs are *not* comparable with Flink and Flumina due to the batching differences. Because the purpose of our evaluation is not to compare absolute performance differences due to batching *across systems*, we focus on relative speedups *on the same system* and how they relate to platform independence **(PIP1–3)**.

6.6.1. Applications Requiring Synchronization

We consider three applications that require different forms of synchronization. All three of the applications do not perform CPU-heavy computation for each event so as to expose communication and underlying system costs in the measurements. The conclusions that we draw remain valid, since

a computation-heavy application that would exhibit similar synchronization requirements would scale even better with the addition of more processing nodes. The input for all three applications is synthetically generated.

Event-based windowing. An *event-based window* is a window whose start and end is defined by the occurrence of certain events. This results in a simple synchronization pattern where parallel nodes must synchronize at the end of each window. For this application, we generate an input consisting of several streams of integer *values* and a single (separate) stream of *barriers*. The task is to produce an aggregate of the values between every two consecutive barriers, where *between* is defined based on event timestamps. We take the aggregation to be the sum of the values. The computation is parallelizable if there are sufficiently more value events than barrier events. In the input to our experiments, there are 10K events in each value stream between two barriers.

Page-view join. The second application is an example of a streaming join. The input contains 2 types of events: *page-view* events that represent visits of users to websites, and *update-page-info* events that update the metadata of a particular website and also output the old metadata when processed by the program. All of these events contain a unique identifier identifying the website and the goal is to join page-view events with the latest metadata of the visited page to augment them for a later analysis. An additional assumption is that the input is not uniformly distributed among websites, but a small number of them receive most of the page-views. To simulate this behavior in the inputs used in our experiments, all views are distributed between two pages.

Fraud detection. Finally, the third application is a version of the ML fraud detection application mentioned in the introduction, where the synchronization requirements are the same but the computation is simplified. The input contains *transaction* events and *rule* events both of which are integer values. On receiving a rule, the program outputs an aggregate of the transactions since the last rule and a transaction is considered fraudulent if it is equivalent modulo 1000 to the sum of the previous aggregate (simulating model retraining) and the last rule event. As in event-based windowing, we generate 10K transaction events between every two rule events.

6.6.2. Implementations in Flink and Timely

In this section we investigate how the Flink and Timely APIs can produce scalable parallel implementations for the aforementioned applications. We iterated on different implementations resulting in a best-effort attempt to achieve good scalability. These implementations are summarized below. For

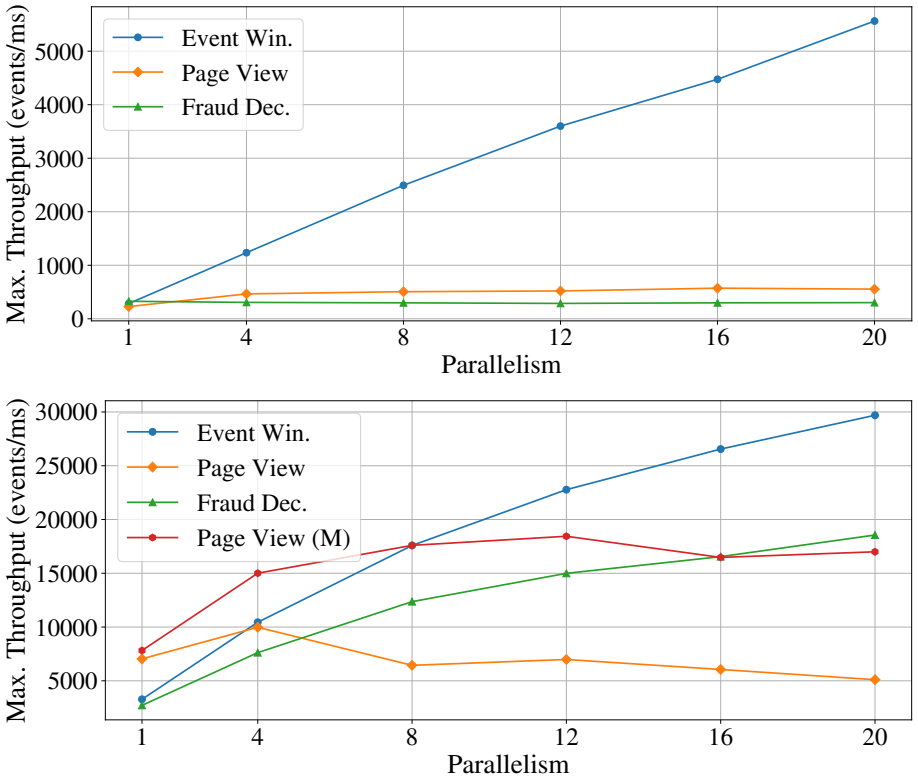


Figure 6.5: Flink (top) and Timely (bottom) maximum throughput increase with increasing parallel nodes for the three applications that require synchronization. For the page-view example in Timely, two implementations are shown: Page View uses automatic parallelism while Page View (M) is the manual parallel implementation in Figure 6.6.

each of the implementations, we then ran an experiment where we increased the number of distributed nodes and measured the maximum throughput of the resulting implementation (by increasing the input rate until throughput stabilizes or the system crashes). The results are shown in Figure 6.5.

Event-based windowing. Flink's API supports a `broadcast` construct that can send barrier events to all parallel instances of the implementation, therefore being able to scale with an increasing parallel input load. Note that Flink does not guarantee that the broadcast messages are synchronized with the other stream events, and therefore the user-written code has to ensure that it processes them in order. By transforming these barriers to Flink watermarks that indicate window boundaries, we can then aggregate values of each window in parallel, and finally merge all windows to get the global aggregate. Similarly, Timely includes a `broadcast` operator on streams, which sends all barrier events to all parallel instances; then the `reclock` operator is used to match values with corresponding barriers and aggregate them. Both the Flink and Timely implementations scale because the values are

```
updates.broadcast().filter(move |x| {
    x.name == page_partition_fun(NUM_PAGES, worker_index)
});
```

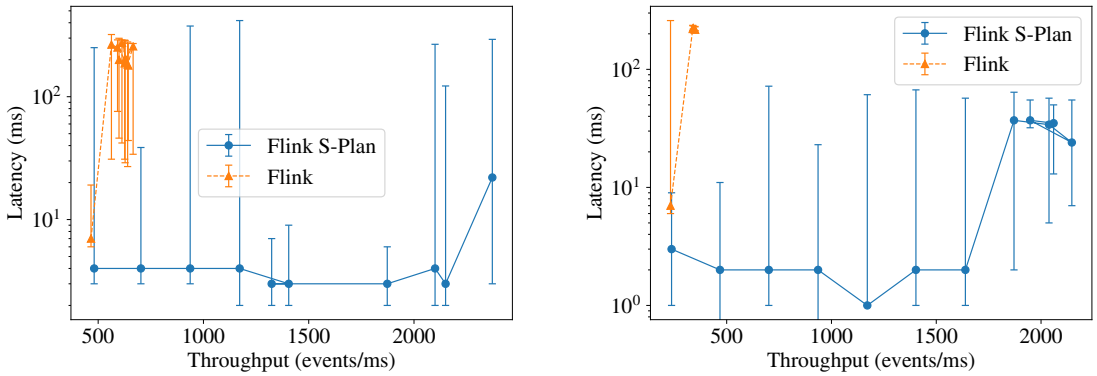
Figure 6.6: Snippet from the Timely manual (M) implementation of the page-view join example, satisfying **PIP1** and **PIP3** but not **PIP2**.

much more frequent than barriers, i.e., a barrier every 10K events, and are processed in a distributed manner.

Page-view join. The input of this application allows for data parallelism across keys, in this case websites, but also for the same key since some keys receive most of the events. First, for both Flink and Timely, we implemented this application using a standard keyed join, ensuring that the resulting implementation will be parallel with respect to keys. As shown by the scalability evaluation, this does not scale to beyond 4 nodes in the case that there are a small number of keys receiving most or all of the events.

We wanted to investigate whether it was possible to go beyond the automatic implementations and scale throughput for events *of the same key*. To study this, we provide a "manual" (M) implementation in Timely (Figure 6.6 and Figure 6.5, bottom). Here, we broadcast *update-page-info* events, then filter to only those relevant to each node, i.e. corresponding to *page-view*s that it is responsible for processing. A similar implementation would be possible in Flink. Unfortunately, our implementation sacrifices **PIP2**, i.e., the assignment of events to parallel instances becomes part of the application logic—there are explicit references to the physical partitioning of input streams (`page_partition_fun`) and the the worker that processes each stream (`worker_index`). Additionally, the implementation broadcasts *all* update events to *all* sharded nodes (not just the ones that are responsible for them), introducing a linear synchronization overhead with the increase of the number of nodes. An alternative choice would have been to not only broadcast events, but also keep state for *every* page at every sharded replica: this would satisfy **PIP2** because nodes no longer need to be aware of which events they process, but it does not avoid the broadcasting issue and thus we would expect performance overheads. Overall, we observe inability to automatically scale this application without sacrificing platform independence in both Flink and Timely.

Fraud detection. The standard dataflow streaming API cannot support cross-instance synchronization, and therefore we can only develop a sequential implementation of this application using Flink's API. Timely offers a more expressive API with *iterative* computation, and this allows for an



(a) Page-view join.

(b) Fraud detection.

Figure 6.7: Throughput (x-axis) and 10th, 50th, 90th percentile latencies on the y-axis for increasing input rates (from left to right) and 12 parallel nodes. Flink (orange) is the parallel implementation produced automatically, and Flink S-Plan (blue) is the synchronization plan implementation.

automatically scaling implementation: after aggregating local state to globally updated the learning model, we have a cyclic loop which then sends the state back to all the nodes to process further events. The results show that this implementation scales almost as well as the event-based window. This effectively demonstrates the value of iterative computation in machine learning and complex stateful workflows.

Take-away (Q1): The streaming APIs of Flink and Timely cannot automatically produce implementations that scale throughput for all applications that have synchronization requirements without sacrificing platform independence.

6.6.3. Manual Synchronization

To address Q2, we next investigate whether synchronization, implemented manually and possibly sacrificing **PIP1–3**, can offer concrete throughput speedups. We focus this implementation in Flink, and consider the two applications that Flink cannot produce parallel implementations for, namely *page-view join* and *fraud detection*. We write a DGS program for these applications and we use our generation framework to produce a synchronization plan for a specific parallelism level (12 nodes). We then manually implement the communication pattern for these synchronization plans in Flink, and we measure their throughput and latency compared to the parallel implementations that the systems produced in Section 6.6.2. The results for both applications are shown in Figure 6.7. Flink

```
public Integer joinChild(
    final int subtaskId,
    final Integer state
) {
    final int parentId = subtaskId / pageViewParallelism;
    final int childId = subtaskId % pageViewParallelism;
    joinSemaphores.get(parentId).get(childId).release();
    forkSemaphores.get(parentId).get(childId)
        .acquireUninterruptibly();
    return zipCode.get(parentId);
}
```

Figure 6.8: Snippet from the implementation of the manual synchronization `join` in Flink. This implementation does not satisfy **PIP1**–**3**.

does not achieve adequate parallelism and therefore cannot handle the increasing input rate (low throughput and high latency).

Page-view join. The synchronization plan that we implement for this application is a forest containing a tree for each key (website) and each of these trees has leaves that process page-view events. Each time an update event needs to be processed for a specific key, the responsible tree is joined, processes the event, and then forks the new state back.

Fraud detection. The synchronization plan that we implement for this application is a tree that processes rule events at its root and transactions at all of its leaves. The tree is joined in order to process rules and is then forked back to keep processing transactions.

Implementation in Flink. In order to implement the synchronization plans in Flink we need to introduce communication across parallel instances. We accomplish this by using a centralized service that can be accessed by the instances using Java RMI. Synchronization between a parent and its children happens using two sets of semaphores, J and F. A child releases its J semaphore and acquires its F semaphore when it is ready to join, and a parent acquires its children's J semaphores, performs the event processing, and then releases their F semaphores (Figure 6.8). This implementation of manual synchronization sacrifices all three platform independence principles **PIP1**–**3**. For **PIP1** and **PIP2**, it refers explicitly to the number of parallel instances and the partitioning (`pageViewParallelism`, `subtaskId`, etc.). For **PIP3**, it is not API-compliant because it uses an external service (semaphores) to implement synchronization, whereas Flink's documentation requires that operators lack side effects. This requirement is imposed because, among other considerations,

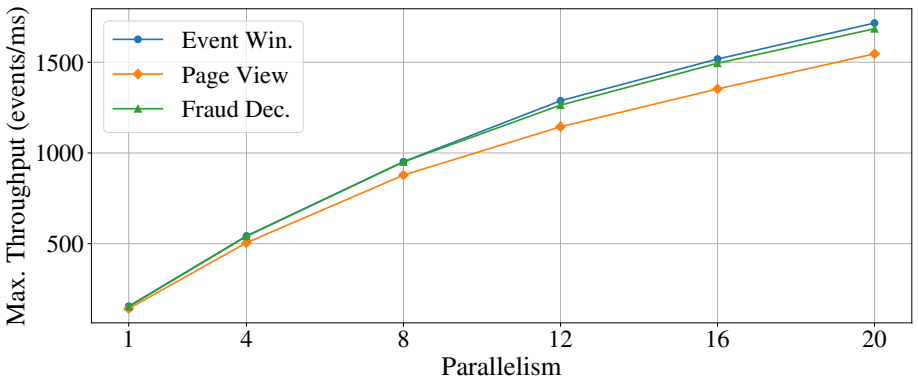


Figure 6.9: Flumina (DGS) maximum throughput increase with increasing parallel nodes for the three applications.

	Event window			Page-view join					Fraud detection			
Development tradeoff	F	TD	DGS	F	FM	TD	TDM	DGS	F	FM	TD	DGS
(PIP1) Parallelism independence	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
(PIP2) Partition independence	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓
(PIP3) API compliance	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
Scaling	10x	8x	8x	2x	9x	1x	2x	8x	1x	9x	6x	8x

Figure 6.10: Development tradeoffs for each program, together with throughput scaling for 12 nodes, in Flink (F), Flink with manual synchronization (FM), Timely (TD), Timely with manual partitioning (TDM), and our system (DGS).

the use of semaphores might cause the program to fail in cases where work is interrupted and/or repeated after a node failure.

Take-away (Q2): Synchronization plans achieve higher throughputs (4-8x for 12 parallel nodes) over the automatic parallel implementations produced by Flink's API.

6.6.4. Implementation in Flumina

To answer Q3, we implement Flumina, a prototype of our end-to-end framework that can automatically achieve parallelism given a DGS program. Flumina receives a DGS program written in Erlang, uses the generation framework that was described in Section 6.5 to generate a correct and efficient synchronization plan, and then implements the plan according to the description in Section 6.5.4. We implemented all three applications in Flumina, and measured the maximum throughput increase with the addition of parallel nodes (Figure 6.9).

Event-based windowing and fraud detection. The DGS program for event-based windowing contains: (i) a sequential update function that adds incoming value events to the state, and outputs the value of the state when processing a barrier event, (ii) a dependence relation that indicates that all events depend on barrier events, and (iii) a `fork` that splits the current state in half, together with a `join` that adds two states to aggregate them. The DGS program for fraud detection is the same with the addition that the `fork` also duplicates the sum of the previous transaction and last rule modulo 1000.

Page-view join. In addition to the sequential update function, the program indicates that events referring to different keys are independent, and that page-view events of the same key are also independent. The `fork` and `join` are very similar to the ones in Figure 6.2 and just separate the state with respect to the keys.

Take-away (Q3): Across all three examples, Flumina produces parallel implementations that scale throughput without sacrificing platform independence.

6.6.5. Summary: Development Tradeoffs

Finally, regarding Q4, Figure 6.10 shows all the tradeoffs that need to be made for each of the programs in this section together with the throughput increase for 12 nodes. Note that throughput scaling comparison is only relevant for the same system (each of which is denoted with a different color) and not across systems due to differing sequential baselines. Of all the implementations in Section 6.6.2, the Timely manual page-view example sacrifices **PIP2**. The manually synchronizing Flink implementations in Section 6.6.3 show good throughput scaling at the cost of **PIP1**–**3**.

Take-away (Q4): Of the three APIs studied, only DGS can achieve scalable implementations across all examples without sacrificing either parallelism independence, partition independence, or API compliance.

6.7. Case Studies

In the next couple of subsections, we evaluate if DGS can be used for realistic application workloads. We consider the following questions:

1. How does the performance of Flumina compare with handcrafted implementations?

2. What is the additional programming effort necessary to achieve automatic parallelization when using the DGS programming model?

To evaluate these questions we describe two case studies on applications taken from the literature that have existing high-performance implementations for comparison. The first is a state-of-the-art algorithm for statistical outlier detection, and the second is a smart home power prediction task from the DEBS Grand Challenge competition. For question (1), the two case studies are posed in the literature targeting different performance metrics: throughput scalability for outlier detection and latency for the smart home task. We are able to achieve performance comparable to the existing handcrafted implementations in both cases with respect to the targeted metric. For question (2), this performance is achieved while putting minimal additional effort into parallelization: compared to 200-700 LoC for the sequential task, writing the fork and join primitives requires only an additional 50-60 LoC. These results support the feasibility of using DGS for practical workloads, with minimal additional programmer effort to accomplish parallelism.

6.7.1. Statistical Outlier Detection

RELOADED [214] is a state-of-the-art distributed streaming algorithm for outlier detection in mixed attribute datasets. It is a structurally similar to the fraud-detection example from the experimental evaluation. The algorithm assumes a set of input streams that contain events drawn from the same distribution. Each input stream is processed independently by a different worker with the goal of identifying outlier events. Each worker constructs a local model of the input distribution and uses that to flag some events as potential outliers. Whenever a user (or some other event) requests the current outliers, the individual workers merge their states to construct a global model of the input distribution and use that to flag the potential outlier events as definitive outliers.

We executed a network intrusion detection task from the original paper [146]. The goal of the task is to distinguish malicious connections from a streaming dataset of simulated connections on a typical U.S. Air Force LAN. Each connection is represented as an input event. In the experiment we varied the number of nodes from 1-8 and we measured the execution time speedup. We executed this experiment on a local server (Intel Xeon Gold 6154, 18 cores @3GHz, 384 GB) with the NS3 [66] network simulator to have comparable network latency with the original paper that used a small local cluster.

Programmability. The sequential implementation of the algorithm in DGS consists of approximately 700 lines of Erlang code—most of which is boilerplate to manage the data structures of the algorithm. Compared to the sequential implementation, the programming effort to achieve a parallel implementation is a straightforward pair of `fork` and fljoin primitives, which amounts to 50 LoC.

Performance. DGS and synchronization plans can capture the complex synchronization pattern that was proposed for RELOADED, achieving comparable speedup to that reported in the original paper: almost linear (7.3× for 8 nodes), compared to 7.7× for 8 nodes by the handcrafted C++ cluster evaluation reported in the paper.

6.7.2. IoT Power Prediction

The DEBS Grand Challenge is an annual competition to evaluate how research solutions in distributed event-based systems can perform on real applications and workloads. The 2014 challenge [159, 160] involves an IoT task of power prediction for smart home plugs. As with the previous case study, our goal is to see if the performance and programmability of our model and framework can be used on a task where there are state-of-the-art results we can compare to. The problem (query 1 of the challenge) is to predict the load of a power system in multiple granularities (per plug, per house, per household) using a combination of real-time and historic data. We developed a solution that follows the suggested prediction method, that involves using a weighted combination of averages from the hour and historic average loads by the time of day. This task involves inherent synchronization: while parallelization is possible at each of the different granularities, synchronization is then required to bring together the historic data for future predictions. For example, if state is parallelized by plug, then state needs to be joined in order to calculate a historic average load by household. Our program is conceptually similar to the map from keys to counters in Section 6.4, where we maintain a map of historical totals for various keys (plugs, houses, and households). The challenge input contains 29GB of synthetic load measurements for 2125 plugs distributed across 40 houses, during the period of one month. We executed our implementation on a subset of 20 of the 40 houses, which we sped up by a factor of 360 so that the whole experiment took about 2 hours to complete. To compare with submissions to the challenge which were evaluated on one node, we ran a parallelized computation on one server node. To simulate the network and measure network load, we then used NS3 [66].

In the state, we maintain a set of maps of recent and historical averages for house, household, and plug. Then, we set different houses, $\texttt{house}_k$ (for k between 1 and 20) to be different tags, and we

add an end-timeslice event # at the end of every hour. The dependence relation is that end-timeslice is dependent on everything (this is when output is produced), and that $\texttt{house}_k$ is dependent on itself for every k, but independent of other houses. The *fork* function splits each map by house ID, and the *join* function merges maps back together.

Programmability. In total, the sequential code of our solution amounted to about 200 LoC, and the parallelization primitives (fork, join, dependence relation) were 60 LoC. We conclude that the overhead to enable parallelization is modest compared to the sequential code.

Performance. Latency varied between 44ms (10th percentile), 51ms (median), and 75ms (90th), and the average throughput was 104 events/ms. These results are on par with the ones reported by that year's grand challenge winner [205]: $6.9ms$ (10th) $22.5ms$ (median) 41.3 (90th) and 131 events/ms throughput. Note that although the dataset is the same, the power prediction method used was different in some solutions. In this application domain, our optimizer has the advantage of enabling edge processing: we measure only 362 MB of data sent over the network, in contrast to the 29 GB of total processed data.

6.8. Outtakes

6.8.1. Communication Optimizer

In this section we describe one of the synchronization plan optimizers that we have developed, namely one that is based on a simple heuristic: it tries to generate a synchronization plan with a separate worker for each input stream, and then tries to place these workers in a way that minimizes communication between them. This optimizer assumes a network of computer nodes and takes as input estimates of the input rates at each computer node. It searches for an P-valid synchronization plan that maximizes the number of events that are processed by leaves; since leaves can process events independently without blocking. The optimizer first uses a greedy algorithm that generates a graph of implementation tags (where the edges are between dependent tags) and iteratively removes the implementation tags with the lowest rate until it ends up with at least two disconnected components.

Example 6.8.1. For an example optimization run, consider Figure 6.4, and suppose that $\texttt{r}(2)$ has a rate of 10 and arrives at node E_0, $\texttt{r}(1)$, $\texttt{i}(1)$ have rates of 15 and 100 respectively and arrive at node E_1, $\texttt{i}(2)_a$ has rate 200 and arrives at node E_2, and $\texttt{i}(2)_b$ has rate 300 and arrives at node E_3.

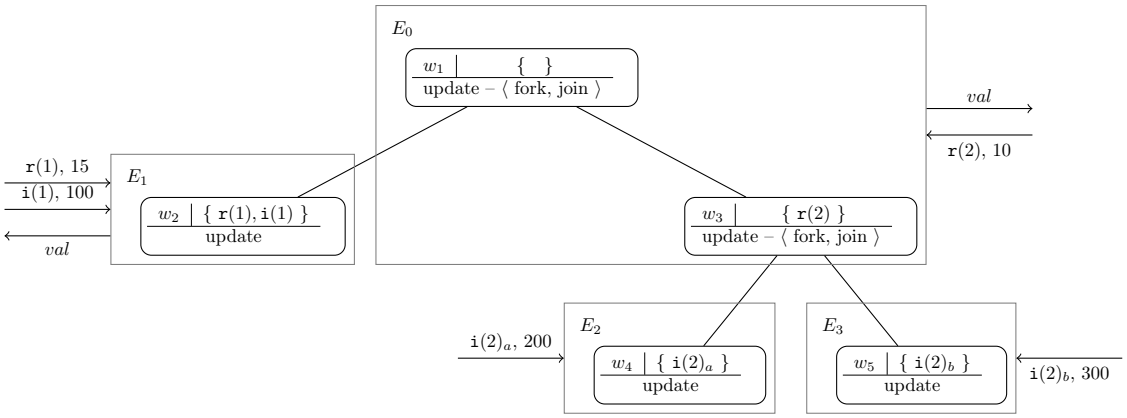


Figure 6.11: Example synchronization plan generated in Example 6.8.1. The large gray rectangles E_0, E_1, E_2, E_3 represent physical nodes and the incoming arrows represent input streams and their relative rates.

Since events of different keys are independent, there are two connected components in the initial graph—one for each key. The optimizer starts by separating them into two subtrees. It then recurses on each disconnected component, until there is no implementation tag left, ending up with the tree structure shown in Figure 6.4. Finally, the optimizer exhaustively tries to match this implementation tag tree, with a sequence of forks, in order to produce a valid synchronization plan annotated with state types, updates, forks, and joins. This generates the implementation in Figure 6.11.

6.8.2. Flumina Synchronization Latency

We also conducted an experiment to evaluate the latency of synchronization plans. We studied three factors that affect latency: (i) the depth of the synchronization plan, (ii) the rate of events that are processed at non-leaf nodes of the plan, and (iii) the heartbeat rate. We ran the event-based windowing application with various configurations and the results are shown in Figure 6.12. Latency increases linearly with the number of workers, since the more workers there are, the more messages have to be exchanged when a barrier event occurs. Note that the latencies are higher for lower vb-ratios since every time a barrier event occurs, all of the nodes need to synchronize. In particular, the system cannot handle beyond 22 workers for vb-ratio of 100 (i.e. 100 forks-joins of the whole synchronization plan per second). On the right, we see that if the heartbeat rate is too low latency increases since worker mailboxes cannot release events quickly and therefore get filled up, only releasing events in big batches whenever a barrier occurs.

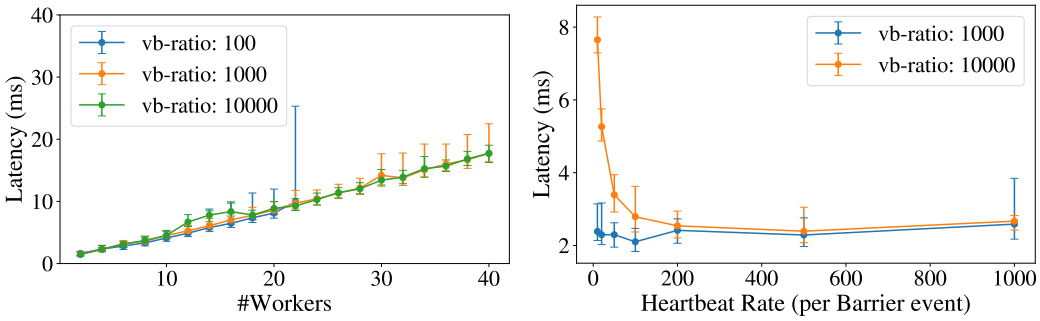


Figure 6.12: Flumina latency (10/50/90 percentile) on the event-based windowing application for various configurations: Synchronization impact on latency (10/50/90 percentile) for various configurations: **(a)** Varying ratios of value to barrier events (different lines) and number of parallel nodes (x-axis). Heartbeat ratio is 1/100 the vb-ratio. The blue line stops after 22 workers. **(b)** Varying ratios of value to barrier events (different lines) and heartbeat rates (x-axis). Number of parallel nodes is fixed to 5.

6.8.3. Flumina State Checkpoints

Flumina provides a simple state checkpointing mechanism for resuming a computation in case of faults. In contrast to other distributed stream processing systems, where the main challenge for checkpointing state is the acquisition of consistent distributed snapshots [203, 60], performing a state checkpoint is straightforward when the root node has joined the states of its descendants. The joined state at that point in time (which corresponds to the timestamp of the message that triggered the join request) is a consistent snapshot of the distributed state of the system, and therefore we get a non-pausing snapshot acquiring mechanism for free. We have exploited this property of Flumina to implement a programmable checkpoint mechanism that is given to the system as an option when initializing worker processes. The checkpoint mechanism can be instantiated to save a snapshot of the state every time the state is joined on the root node, or less frequently, depending on an arbitrary user defined predicate. We implemented a checkpoint mechanism that produces a checkpoint every time the root node joins its children states.

6.9. Discussion

There are a wealth of problems that we left open in this work. One problem that is not yet adequately explored in our framework is optimization: given a DGS program, how to select a valid synchronization plan which is most efficient according to a desired cost metric. Traditional optimization algorithms for stream processing systems cannot be directly applied to the complex

tree structure of synchronization plans. There are also possibilities for dynamic optimization, in which the synchronization plan is modified online in response to profiling data from the system. Besides optimization, the implementation of synchronization plans needs to address a plethora of systems related issues, such as (i) the efficient management and communication of forked state in a distributed environment, (ii) execution guarantees in the presence of faults, and (iii) supporting performance optimizations such as batching and backpressure.

From the perspective of the structured types and type safety in Chapter 4, an important limitation is that we only defined output up to reordering: essentially, this means that our type safety theorem for DGS programs applies to any input type S, but only to output types $S' = \mathsf{Bag}(T)$ for some T. I don't believe this is fundamental; it was done to simplify the model, which already involved a fair amount of complexity, so we didn't want to worry about output orderings. This also means that DGS can't currently do sequential composition, where we feed the output of one program to another. To explore this one needs to define the requirements on the programming model more carefully to track not just a dependence relation on the input, but also a dependence relation on the output.

The core programming model of Section 6.4 is elegant, and one thing we have wondered is whether it can be used to get consistency guarantees in distributed programming. Related and recent work in consistent replicated distributed systems (e.g., RedBlue consistency [179], MixT [198], Gallifrey [199], Hamband [155], and Quark [165]) suggests a starting point.

CHAPTER 7 : Testing

The return on investment from random testing is good. Our rough estimate—including faculty, staff, and student salaries, machines purchased, and university overhead—is that each of the more than 325 bugs we reported cost less than $1,000 to find.

—Yang, Chen, Eide, and Regehr, 2011 [282]

Given the typing discipline in Chapter 4, Chapters 5 and 6 have considered ways to write well-typed programs: programs defined compositionally, and for which they can be parallelized in a semantics-preserving way that ensures determinism.

This section considers the problem from a different angle: suppose we have an application that is already defined, or an input arriving from an external source; can we *test* empirically whether it is well-typed and deterministic with respect to a given input and output stream type? This problem is not easy. One issue is that the property we are interested in can be thought of as "on all executions, the ordering of the stream is the same up to equivalence" which is a property that requires running the program multiple times (i.e. a "hyperproperty"). We consider *differential testing* as a way to address this and test at runtime whether a program satisfies its stream typing requirements.

Specifically, we consider differential testing of two output streams, up to equivalence defined by a stream type S'. This may seem more restricted than testing whether a program is well-typed with input type S and output type S', but it turns out to be only more specific in some aspects, and more general in others. To solve the type safety and determinism problem, we have to generate two example inputs that are equivalent, feed them each to the same program P, and apply the differential testing algorithm on the two resulting outputs. So if we also have an input generation procedure, then we can solve the general type safety and determinism problem. But we can also apply differential output testing to two *different* programs P_1 and P_2 (typically, a sequential version and a parallel version), which should be equivalent, and compare their outputs.

7.1. Motivation

Beyond the context of this thesis, the problem of ensuring correctness of distributed programs is long established, as can be seen by the significant amount of past and ongoing work on the correctness of distributed protocols (e.g., [68, 16, 218]), concurrent data structures (e.g., [145, 57]), and distributed systems (e.g., [217, 273, 144]). In general, because this body of work addresses the challenges of verification of distributed protocols and low-level primitives, it targets experts who design and implement complex distributed systems. However, data-parallel programming frameworks, such as stream processing systems, aim to offer a simplified view of distributed programming where low level coordination and protocols are hidden from the programmer. This has the advantage of bringing distributed programming to a wider audience of end-users, and at the same time it requires new tools that *can be used by such end-users* (rather than just experts) to automatically test for correctness.

Unfortunately, there is limited work in testing for stream processing programs; in fact, the state of the art in practice is unit and integration testing [267]. In order to bridge this gap and provide support for checking correctness to end-users, we focus on the problem of *differential testing* for distributed stream processing systems, that is, checking whether the outputs produced by two implementations in response to an input stream are equivalent. Differential testing [193, 139, 107], allows for a simple specification of the correct program behavior, in contrast to more primitive testing techniques, where the specification is either very coarse (i.e. the application doesn't crash) or very limited (i.e. on a given input, the application should produce a specific output). More precisely, differential testing allows for a reference implementation to be the specification. This is especially useful in the context of distributed stream processing systems, since bugs introduced due to distribution can be caught by comparing a sequential and a distributed implementation. In addition, having a reference implementation as a specification allows testing with random inputs, since there is no need to specify the expected output.

We identify two critical challenges in testing stream processing programs. The first challenge is dealing with output events that are out-of-order due to parallelism. In particular, for differential testing, two implementations might produce events at different rates, asynchronously, and out-of-order. However, the order between specific output events might not affect the downstream consumer (e.g. events with timestamp less than t can arrive in any order, as long as they arrive before the watermark t),

therefore requiring our notion of equivalent streams which allows for out-of-order events. In fact, lack of order is often desirable, since it enables parallelism. Importantly on the other hand, *not all* output events are unordered, because operators in streaming dataflow graphs (in contrast to in batch processing and MapReduce settings) often require some order to preserved (see Section 7.3). Because of this, not all operators simply decompose into commutative/associative aggregators, and prior solutions on testing [87, 280, 189, 76] and static verification [181, 231] for MapReduce-like programs cannot be directly applied.

The second challenge is that stream processing systems, in contrast to batch processing systems, are designed to process input data that would not fit in memory. As best practice, it is recommended that applications written in these systems are tested under heavy load for long periods of time, to match the conditions that are expected after deployment [267]. Achieving this requires that the testing framework itself is an online algorithm, in the sense that it processes output events as they arrive, and that the computational overhead is minimal.

7.2. Contributions

We propose a matching algorithm that incrementally compares two streams for equivalence. Following the approach of [9], in our solution, ordering requirements between pairs of events are abstracted in a *dependence relation* that indicates when the ordering of two specific events is of significance to the output consumer. Given *any* dependence relation provided by the user, the algorithm determines, in an online fashion, whether the streams are equivalent up to the reorderings allowed by the dependence relation. We show that the algorithm is correct and that it reaches a negative verdict at the earliest possible time (Theorem 7.6.8). We also prove that the algorithm is optimal, in the sense that it uses a minimal amount of space: any correct online algorithm must store at least as much information (Theorem 7.6.9).

We have implemented DiffStream, a differential testing library for Apache Flink that incorporates our algorithm. DiffStream is implemented in Java and can be used alongside existing testing frameworks such as JUnit [255] or in stand-alone Flink programs. In order to evaluate the effectiveness and usability of the proposed testing framework, we have conducted a series of case studies.

First, we evaluate the effectiveness of the framework on a set of nondeterministic MapReduce programs from [277], adapted to the streaming setting. For some of these programs nondeterminism constitutes a bug, while for others it is acceptable, depending on input assumptions and application requirements. Using our framework, we demonstrate that tests can be written to successfully detect 5 out of 5 bugs (true positives), and to avoid flagging 5 out of 7 bug-free programs (false positives). This improves on previous work [281], which would generally flag all nondeterministic programs as buggy, thus suffering from false positives.

Second, we design two specific use cases to illustrate the benefits of using DiffStream to design and implement parallel Flink applications. We consider a difficult-to-parallelize application which requires event-based windowing: we show that it is significantly more difficult (requiring twice as many lines of code) to effectively parallelize this application using Flink, and we show how our framework can be used to test and correctly implement such an application. We also evaluate the effort needed to write tests for an example computation with a subtle bug. The specific programs we consider are explained in more detail in Section 7.3.

Finally, we demonstrate that the matching algorithm is efficient in practice, and can be used in an online monitoring setting, by monitoring two implementations of the Yahoo Streaming Benchmark [78] over the span of two hours and measuring the impact on performance. The overhead of testing is a modest 5% reduction in maximum possible throughput, and the memory usage remains stable at less than 500 unmatched items out of 30K items per second, reflecting the theoretical optimality of the algorithm for this particular application.

In total, the main contributions of this work are:

- A new testing methodology for specifying ordering requirements in stream processing programs. (Section 7.5)
- An optimal online matching algorithm for differential testing of stream processing programs which uniformly handles data with differing ordering requirements. (Section 7.6)
- DiffStream, a differential testing library for testing Apache Flink applications based on the online matching algorithm, together with a series of case studies to evaluate its usability and effectiveness. (Section 7.8)

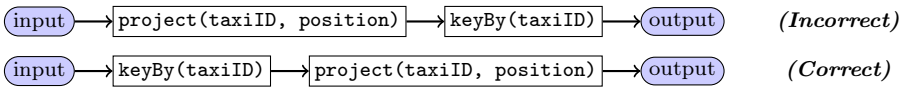


Figure 7.1: A subtle consequence of implicit parallelism over an input stream containing taxi location data.

DiffStream is available as an open-source repository on GitHub[1].

7.3. Example Use Cases

Programs written in distributed stream processing frameworks exhibit implicit parallelism, which can lead to subtle bugs. Programs in such frameworks are usually written as *dataflow graphs*, where the edges are data streams and the nodes are streaming operators or transformations. Common operators include stateless transformations (*map*), and operations that group events based on the value of some field (*key-by*). For example, suppose that we have a single input stream which contains information about rides of a taxi service: each input event ($\texttt{id}, \texttt{pos}, \texttt{meta}$) consists of a taxi identifier, the taxi position, and some metadata. In the first stage, we want to discard the metadata component (*map*) and partition the data by taxi ID (*key-by*). In the second stage, we want to aggregate this data to report the total distance traveled by each taxi. Notice that the second stage is order-dependent (events for each taxi need to arrive in order), so it is important that the first stage does not disrupt the ordering of events for a particular taxi ID.

To make a program for the first stage of this computation in a distributed stream processing framework such as Flink or Storm, we need to build a dataflow graph representing a sequence of transformations on data streams. A first (natural) attempt to write the program is given in Figure 7.1 (top). Here, the `project` node projects the data to only the fields we are interested in; in this case, `taxiID` and `position`. And `keyBy` (also known as "group by" in SQL-like languages, or the concept of a "stream grouping" in Storm) partitions the data stream into substreams by `taxiID`. Although written as an operator, here `keyBy` can be thought of as modifying the stream to give it a certain property (namely, if it is parallelized, streams should be grouped by the given key).

The first attempt is incorrect, however, because it fails to preserve the order of data for a particular key (taxi ID), which is required for the second stage of the computation. The problem is that dataflow

[1] https://github.com/fniksic/diffstream

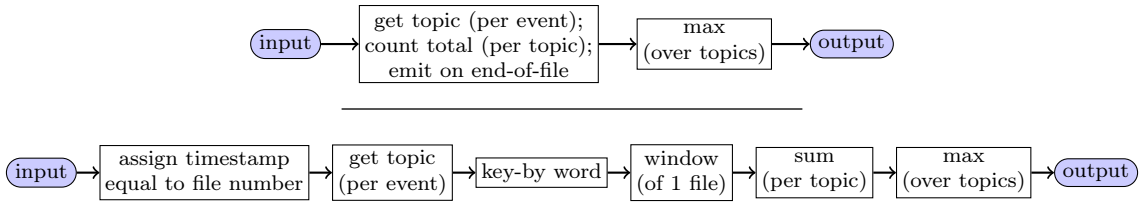


Figure 7.2: A difficult-to-parallelize sequential program (top), and the correct parallel version (bottom) over an input set of documents which arrive concatenated in a single stream.

graph operators are implicitly parallelized—here, the stateless map `project` is internally replicated into several copies, and the events of the input stream are divided among the copies. Because input events of the same key may get split across substreams, when the operator `keyBy` reassigns each item to a new partition based on its key, if items of a particular key were previously split up, then they might get reassembled in the wrong order.

This issue can be addressed by ensuring that parallelization is done only on the basis of `taxiID` *from the beginning of the pipeline*. This can typically be accomplished by simply by reversing the `project` and `keyBy` transformations, as in Figure 7.1 (bottom). (For example, this is done explicitly in Flink, and the concept is the same in Storm, except that instead of an explicit `keyBy` operator we implicitly construct it by setting the input stream to be grouped by key.) Although the two programs are equivalent when the `project` operation is not parallelized, the second lacks the undesirable behavior in the presence of parallelism: assuming the `project` operation has the same level of parallelism as `keyBy`, most systems will continue to use the same partition of the stream to compute the projection, so data for each key will be kept in-order. In particular, this works in any framework which guarantees that the same key-based partitioning is used between stages.

We have seen that even simple programs can exhibit counterintuitive behavior. In practice, programs written to exploit parallelism are often much more complex. To illustrate this, consider a single input stream consisting of very large documents, where we want to assign a topic to each document. The documents are streamed word by word and delineated by end-of-file markers. The topic of each word is specified in a precomputed database, and the topic of a document is defined to be the most frequent topic among words in that document.

In this second example querying the database is a costly operation, so it is desirable to parallelize by partitioning the words within each document into substreams. However, the challenge is to do so in

a way that allows for the end-of-file markers to act as *barriers*, so that we re-group and output the summary at the end of each document. Although a sequential solution for this problem is easy, the simplest solution we have found in Flink that exploits parallelism uses about twice as many lines of code (Figure 7.2). The source of the complexity is that we must first use the end-of-file events to assign a unique timestamp to each document (ignoring the usual timestamps on events used by Flink). After these timestamps are assigned, only then is it safe to parallelize, because windowing by timestamp later recovers the original file (set of events with a given timestamp). We also consulted with Flink users on the Flink mailing list, and we were not able to come up with a simpler solution. The additional complexity in developing the parallel solution, which requires changing the dataflow structure and not simply tuning some parameter, further motivates the need for differential testing.

7.4. DiffStream

These examples motivate the need for some form of testing to determine the correctness of distributed stream processing applications. We propose *differential testing* of the sequential and parallel versions. As the parallel solution might be much more involved, this helps validate that parallelization was done correctly and did not introduce bugs.

In the example of Figure 7.1, the programmer begins with either the correct program P_1 (bottom), or the incorrect program P_1' (top), and wishes to test it for correctness. To do so, they write a correct reference implementation P_2; this can be done by explicitly disallowing parallelism. Most frameworks allow the level of parallelism to be customized; e.g. in Flink, it can be disabled by calling `.setParallelism(1)` on the stream. The program P_1 or P_2 is then viewed as a black-box reactive system: a function from its input streams to a single *output stream* of events that are produced by the program in response to input events.

However, the specification of P_1 and P_2 alone is not enough, because we need to know whether the output data produced by either program should be considered unordered, ordered, or a mixture of both. A naive differential testing algorithm might assume that output streams are out-of-order, checking for multiset equivalence after both programs finish; but in this case, the two possible programs P_1 will both be equivalent to P_2. Alternatively, it might assume that output streams are in-order; but in this case, neither P_1 nor P_1' will be equivalent to P_2, because data for different taxi

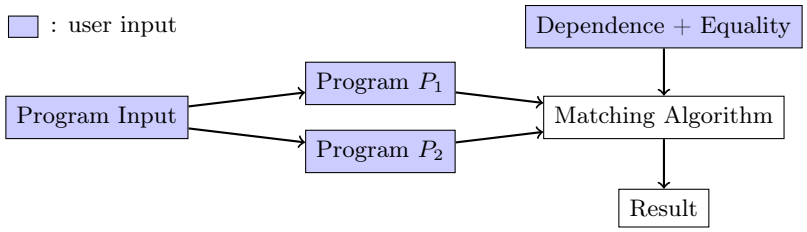


Figure 7.3: DiffStream architecture.

IDs will be out of order in the parallel solution. To solve this, the programmer additionally specifies a *dependence relation*: given two events of the output stream, it returns *true* if the order between them should be considered significant. For this example, output events are dependent if they have the same taxi ID. In general, the dependence relation can be used to describe a flexible combination of ordered, unordered, or partially ordered data.

The end-to-end testing architecture is shown in Figure 7.3. In summary, the programmer provides: (1) a program (i.e., streaming dataflow graph) P_1 which they wish to test for correctness; (2) a correct reference implementation P_2; (3) a *dependence relation* which tells the tester which events in the output stream may be out-of-order; (4) if needed, overriding the definition of equality for output stream events (for example, this can be useful if the output items may contain timestamps or metadata that is not relevant for the correctness of the computation); and (5) optionally, a custom generator of input data streams, or a custom input stream—otherwise, the default generator is used to generate random input streams. The two programs are then connected to our differential testing algorithm, which consumes the output data, monitors whether the output streams so far are equivalent, and reports a mismatch in the outputs as soon as possible.

7.5. Writing Specifications in DiffStream

In this section we describe how the programmer writes specifications in DiffStream. Let's look back at the taxi example from before. The second stage of the program computes the total distance traveled by each taxi by computing the distance between the current and the previous location, and adding that to a sum. For this computation to return correct results, location events for each taxi should arrive in order in its input—a requirement that must be checked if we want to test the first stage of the program.

```
(ev1, ev2) ->
    ev1.taxiID == ev2.taxiID
```

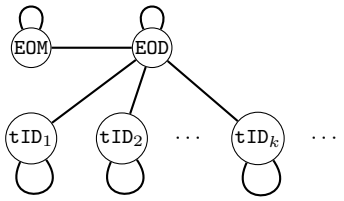
(a) Specification in DiffStream

(b) Dependence visualized as a graph

Figure 7.4: Example specification in DiffStream for the taxi example. Taxi events with the same `taxiID` are dependent.

```
(ev1, ev2) ->
    ev1.isEOD() ||
    ev2.isEOD() ||
    (ev1.isEOM() && ev2.isEOM()) ||
    (ev1.isTaxiEv() &&
     ev2.isTaxiEv() &&
     ev1.taxiID == ev2.taxiID)
```

(a) Specification in DiffStream



(b) Dependence visualized as a graph

Figure 7.5: Example specification in DiffStream for the extended taxi example. Taxi events with the same `taxiID` are dependent and all events are dependent with end-of-day (EOD) events.

A dependence relation is a symmetric binary relation on events of a stream with the following semantics. If x D y, then the order of x and y in a stream is significant and reordering them gives us two streams that are not equivalent. This could be the case if the consumer of an output stream produces different results depending on the order of x and y. Thus, the dependence relation can be thought of as encoding the pairwise ordering requirements of the downstream consumer.

It is often helpful to visualize dependence relations as unordered graphs, where nodes are equivalence classes of the dependence relation. For the taxi example, the dependence relation is visualized in Figure 7.4b, and it indicates that events with the same taxi identifier are dependent. In DiffStream, dependence relations can be specified using a Boolean function on a pair of events. These functions should be pure and should only depend on the fields of the two events. The DiffStream specification of the dependence relation from Figure 7.4b is shown in Figure 7.4a.

Now let's consider an extension of the above example where the downstream consumer computes the total distance traveled by each taxi *per day*, and also computes the average daily distance by each taxi every month. To make this possible, the output of the program under test is now extended with special EOD (*end-of-day*) and EOM (*end-of-month*) events. The ordering requirements on this output, while more subtle, can still be precisely specified using a dependence relation. For example,

```
(ev1, ev2) -> distance(ev1.loc, ev2.loc) < 1
```

Figure 7.6: Example specification in DiffStream where events are dependent if their locations are close.

```
(ev1, ev2) -> (ev1.isPunctuation() &&
               ev2.timestamp < ev1.timestamp) ||
              (ev2.isPunctuation() &&
               ev1.timestamp < ev2.timestamp)
```

Figure 7.7: Example specification in DiffStream where punctuation events, used to enforce progress, depend on other events only if the punctuation timestamp is larger.

EOD events are dependent with taxi events since all events of a specific day have to occur before the EOD event of that day for the total daily distance to be correctly computed. On the other hand, EOM events do not have to be dependent with taxi events since daily distances are computed on EOD events. Therefore, an EOM event can occur anywhere between the last EOD event of the month and the first EOD event of the next month. The DiffStream specification of the dependence relation and its visualization are both shown in Figure 7.5.

Several frequently occurring dependence relations can be specified using a combination of the predicates seen in the above examples. This includes predicates that check if an event is of a specific type (e.g. `isEOD()`, `isTaxiEv()`), and predicates that check a field (possibly denoting a key or identifier) of the two events for equality (e.g. `ev1.taxiID == ev2.taxiID`). However, it is conceivable that the dependence of two events is determined based on a complex predicate on their fields.

Another interesting dependence relation occurs in cases where output streams contain punctuation events. Punctuations are periodic events that contain a timestamp and indicate that all events up to that timestamp, i.e., all events `ev` such that `ev.timestamp < punc.timestamp`, have *most likely* already occurred. Punctuation events allow programs to make progress, completing any computation that was waiting for events with earlier timestamps. However, since events could be arbitrarily delayed, some of them could arrive after the punctuation. Consider as an example a taxi that briefly disconnects from the network and sends the events produced while disconnected after it reconnects with the network. These events are usually processed with a custom out-of-order handler, or are completely dropped. Therefore, punctuation events are dependent with events that have an earlier

Input: Equality relation $\equiv$, dependence relation D
Input: Connected stream s with $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$
Require: Relations $\equiv$ and D are compatible

```
1:  function StreamsEquivalent(s)
2:      u_1, u_2 ← empty logically ordered sets
3:      Ghost state: p_1, p_2 ← empty logically ordered sets
4:      Ghost state: f ← empty function p_1 → p_2
5:      for (x, i) in s do
6:          j ← 3 − i
7:          if x is minimal in u_i and ∃y ∈ min u_j : x ≡ y then
8:              u_j ← u_j \ {y}
9:              p_i ← p_i ∪ {x}; p_j ← p_j ∪ {y}
10:             f ← f[x ↦ y] if i = 1 else f[y ↦ x]
11:         else if ∃y ∈ u_j : x D y then
12:             return false
13:         else
14:             u_i ← u_i ∪ {x}
15:     return (u_1 = ∅ and u_2 = ∅)
```

Figure 7.8: DiffStream algorithm: checking equivalence of two streams.

timestamp, since reordering them alters the result of the computation, while they are independent of events with later timestamps. This can be specified in DiffStream as shown in Figure 7.7.

7.6. Differential Testing Algorithm

Algorithm 7.8 checks for equivalence of two streams. As described in the overview, the algorithm has two main features: (i) it can check for equivalence up to any reordering dictated by a given dependence relation, and (ii) it is online—it processes elements of the stream one at a time. In this section we present Algorithm 7.8, our algorithm for checking equivalence of two streams. We prove that our algorithm is correct, and we show that it is optimal in the amount of state it stores during execution.

7.6.1. Background

Before getting to the algorithm itself, we need to introduce some terminology. A *stream* s is a bounded or unbounded sequence of elements: $s = \langle x_1, x_2, \ldots \rangle$. We write $x \in s$ to denote that x is an element of s, we write $s[n]$ for the nth element of s, and we write $s[{:}n]$ for the bounded substream of elements up to and including the nth element.

We follow the convention that all elements of a stream (denoted with x, y, etc.) are distinct. This is so that we can unambiguously refer to the location of x in the stream s, and for example, say which of x and y occurs earlier. We use $x \equiv y$ to refer to equality of the *underlying values*, rather than the elements as positioned in the stream.

Two streams s_1, s_2 are given as input to the algorithm as a *connected stream* s, which is a stream obtained by arbitrarily interleaving the elements of s_1 and s_2. More precisely, the elements of the connected stream s are of the form (x, i) such that $i \in \{1, 2\}$ and $x \in s_i$. We can recover the original streams by using *projections* π_1 and π_2: $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$. Conversely, given a stream s, we can form connected streams using *injections* ι_1 and ι_2: $\iota_1(s)$ is obtained by mapping each $x \in s$ to $(x, 1)$, and analogously, $\iota_2(s)$ is obtained by mapping each $x \in s$ to $(x, 2)$. Thus, $\iota_1(s)$ and $\iota_2(s)$ are characterized by $\pi_1(\iota_1(s)) = \pi_2(\iota_2(s)) = s$ and $\pi_1(\iota_2(s)) = \pi_2(\iota_1(s)) = \emptyset$. The motivation for connected streams comes from the fact that the streams s_1 and s_2 are produced by the stream processing system asynchronously.

Next, we need to describe what it means for two streams to be equivalent. Our notion of equivalence relies on two relations on the elements of the streams: an *equality relation*, denoted by $\equiv$, and a *dependence relation*, denoted by D. The equality relation is provided by the user (e.g., in Java by overriding the method equals()) and is required to be an equivalence relation, that is, it should be reflexive, transitive, and symmetric. For elements x and y, we write $x \equiv y$ instead of $x = y$ for the equality relation to emphasize that it refers to equality on the underlying values, rather than equality of stream elements. The dependence relation is required to be symmetric, that is, for elements x and y, x D y implies y D x. Finally, the equality and the dependence are required to be *compatible*: if x D y and $x \equiv x'$, then x' D y. The three requirements—the equality being an equivalence relation, the dependence being a symmetric relation, and the equality and dependence being compatible—need to be ensured by the user.

Given a stream s, a dependence relation D gives rise to a *logical order* on the elements in s: for elements $x, y \in s$, x logically precedes y, denoted by $x < y$, if x precedes y in the stream and either x and y are dependent or they are transitively dependent—there are intermediate elements $x_1, \ldots, x_n \in s$ given in their order of occurrence in s such that x D x_1 D $\ldots$ D x_n D y. It can be shown that the logical order is irreflexive and transitive, that is, it is a strict partial order on the

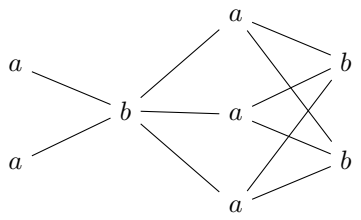


Figure 7.9: The logical order of the stream from Example 7.6.1. Vertically aligned elements are logically unordered, and for two elements that are not aligned, the left one logically precedes the right one. The two leftmost elements are minimal.

elements of the stream s. Recall that this makes sense because by convention all the elements are distinct, even though the underlying values may be equivalent according to the equality relation $\equiv$.

Example 7.6.1. Consider a stream $s = \langle a, a, b, a, a, a, b, b\rangle$. The equality relation $\equiv$ is given by $a \equiv a$ and $b \equiv b$, and the dependence relation D is given by a D b (and b D a). The logical order arising from D is shown in Figure 7.9. The logical orderings between elements include $s[1] < s[3]$, $s[3] < s[4]$, and $s[5] < s[7]$. Also $s[4] \parallel s[5]$, $s[4] \parallel s[6]$, and $s[5] \parallel s[6]$. Note that $s[1] < s[4]$ even though $s[1] \not\mathrel{\mathrm{D}} s[4]$ (both elements are a). This is because they both depend on $s[3] = b$, which is in between.

Given two streams s and s', an equality relation $\equiv$, and a dependence relation D, we say that s and s' are *equivalent* if they give rise to the same logical order. More precisely, we say they are equivalent if there exists a bijective mapping $f\colon s \to s'$, called a *matching*, that matches equal elements and preserves the logical order, that is, for every $x, y \in s$, $f(x) \equiv x$ and $f(x) < f(y)$ if and only if $x < y$. In case the streams are equivalent, we write $s \equiv_{\mathrm{D}} s'$, or simply $s \equiv s'$ if the dependence relation is clear from the context. We call two streams that are not equivalent *distinguishable*.

If the two streams s and s' are bounded, one way to think about them being equivalent is as follows: we can get from s to s' in finitely many steps by either swapping two adjacent logically unordered elements or by replacing an element with another equal element. In particular, bounded equivalent streams have the same length.

Example 7.6.2. Streams $s_1 = \langle a, c, b\rangle$ and $s_2 = \langle c, a, b\rangle$ are equivalent with respect to a dependence relation given by a D b and c D b. A (unique) matching is given by $f\colon s_1[1] \mapsto s_2[2]$, $s_1[2] \mapsto s_2[1]$,

and $s_1[3] \mapsto s_2[3]$. Note that the same streams are not equivalent with respect to a dependence relation where additionally $a \mathrel{D} c$.

When it comes to unbounded streams, it may be impossible to algorithmically decide whether they are equivalent or not. For example, consider $s_1 = \langle a, a, a, \ldots \rangle$ and $s_2 = \langle b, b, b, \ldots \rangle$ with $a \not\mathrel{D} b$. Clearly, $s_1 \not\equiv s_2$, but an algorithm processing a connected stream s with $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$ one element at a time can never reach a conclusion: perhaps eventually b's will start arriving on the first stream, and a's will start arriving on the second stream. However, there are situations when an algorithm can reach a definite decision even if the streams are unbounded. We say that a connected stream s is *finitely distinguishable* if there is a position n such that for every continuation s' of $s[{:}n]$, the projected streams $\pi_1(s[{:}n] \cdot s')$ and $\pi_2(s[{:}n] \cdot s')$ are distinguishable.

Example 7.6.3. If s is a connected stream such that $\pi_1(s) = \langle a, a, b \rangle$ and $\pi_2(s) = \langle a, b \rangle$, and $a \mathrel{D} b$, then for no continuation of s will the two projections ever be equivalent. Thus, s is finitely distinguishable.

Given a partial order p, we say that an element $x \in p$ is *minimal* if no other element is less than x. There can be multiple minimal elements in p; we denote the set of minimal elements in p by $\min p$. Given two partial orders p and q such that $p \subseteq q$, we say that p is a *prefix* of q if for every element $x \in p$, p also contains all the elements that are less than x in q.

7.6.2. Algorithm Description

We now give a general specification of an online equivalence-checking algorithm. The algorithm's inputs are an equality relation $\equiv$, a dependence relation D, and a connected stream s with the projections $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$. We require the equality and the dependence relations to be compatible. The algorithm provides a procedure StreamsEquivalent that returns return **true** or **false** to report whether or not $s_1 \equiv s_2$. The function is allowed to iterate over s exactly once.

The algorithm is correct if it has the following behavior:

(I) StreamsEquivalent returns **true** if and only if s is bounded and s_1, s_2 are equivalent.

(II) StreamsEquivalent returns **false** if and only if either s is finitely distinguishable or it is bounded and the streams s_1, s_2 are distinguishable. Additionally, if s is finitely distinguishable, it returns **false** after processing $s[n]$ for the first position n such that $s[:n]$ is finitely distinguishable.

Algorithm 7.8 achieves the required behavior in the following way. Intuitively, it tries to construct a matching to demonstrate equivalence of s_1 and s_2. In doing so, as part of its state it maintains two logically ordered sets u_1 and u_2, both initially empty. Their role is to keep track of the unmatched elements from s_1 and s_2, respectively. In addition to u_1 and u_2, which constitute the physical state, the algorithm maintains the so-called "ghost state," written in gray in Algorithm 7.8. The ghost state need not exist in any real implementation of the algorithm; its sole purpose is to aid in proving correctness. As part of the ghost state, the algorithm maintains two additional logically ordered sets p_1 and p_2, whose role is to keep track of the successfully matched prefixes of s_1 and s_2. In the ghost state, the algorithm also explicitly keeps track of the matching $f\colon p_1 \to p_2$.

When processing a new element x from s_i (lines 5–14 in Algorithm 7.8), there are three distinguished cases:

1. The element x is minimal in u_i and there is a corresponding unmatched minimal element $y \in u_j$ such that $x \equiv y$ (line 7). In this case we remove y from u_j. In the ghost state, we add x to p_i and y to p_j, and we extend the matching f to map x to y or y to x, depending on whether $x \in s_1$ or $y \in s_1$ (lines 9–10).

2. The element x depends on some unmatched element $y \in u_j$ (line 11). If this is the case, then we have detected finite distinguishability and the function returns **false** (line 12).

3. If neither of the previous cases holds (line 13), then for every $y \in u_j$, x and y are unequal and independent. We add x to u_i as an unmatched element (line 14).

If the whole connected stream has been processed and the function StreamsEquivalent did not return **false** in line 12, it returns **true** in line 15 if and only if both sets of unmatched elements are empty.

Example 7.6.4. Let us demonstrate the execution of Algorithm 7.8 on streams s_1 and s_2 from Example 7.6.2. Suppose the connected stream given as input is

$$s = \langle (a,1), (c,2), (c,1), (b,1), (a,2), (b,2) \rangle .$$

At the time of processing the element $s[3] = (c,1)$, the first two elements have already been processed, and both of them are unmatched: $u_1 = \{a\}$ and $u_2 = \{c\}$. The algorithm detects that the new element c is minimal in u_1 and it can be matched with the element $c \in u_2$, so it updates the matching f with $f(s_1[2]) = s_2[1]$ and removes c from u_2. Next, it processes $s[4] = (b,1)$: the element b is not minimal in u_1 as it depends on $a \in u_1$. It is also not dependent on any element in u_2, as u_2 is empty. Therefore, it is added to u_1, which now contains the ordering $a < b$. Finally, the two elements $s[5] = (a,2)$ and $s[6] = (b,2)$ arrive precisely in the right order to be matched with the elements in u_1, and the algorithm concludes that the streams are equivalent.

If the dependence relation contained the additional dependence $a \mathrel{D} c$, the processing would have stopped at the element $s[2] = (c,2)$, since the element c from s_2 would have been dependent on an unmatched element $a \in u_1$. And indeed, the connected stream $s[:2] = \langle (a,1), (c,2) \rangle$ would have been finitely distinguishable.

7.6.3. Correctness

In order to show that the algorithm is correct, we will show that the loop in StreamsEquivalent (lines 5–14) maintains the following invariants.

(I0) For every $x \in u_1$ and $y \in u_2$, x and y are unequal and independent: $\forall x \in u_1, \forall y \in u_2 : x \neq y \wedge x \not\mathrel{D} y$.

(I1) p_1 is a prefix of s_1: $\forall x \in p_1, \forall y \in s_1 : y < x \Rightarrow y \in p_1$.

(I2) p_2 is a prefix of s_2: $\forall x \in p_2, \forall y \in s_2 : y < x \Rightarrow y \in p_2$.

(I3) $f : p_1 \to p_2$ is a maximal matching, that is, it is a matching and no extension $f' : p_1' \to p_2'$ to proper supersets $p_1' \supset p_1$ and $p_2' \supset p_2$ is a matching. We also say that p_1 and p_2 are maximally matched prefixes.

Lemma 7.6.5. The loop in StreamsEquivalent (lines 5–14) maintains invariants (I0)–(I3).

Proof. Of the four invariants, the one that is least straightforward is (I3), so let us show that it holds. In particular, let us show that after the update in lines 9–10 of Algorithm 7.8, the function f remains a matching between p_1 and p_2. Clearly it is still a bijection and it maps elements in p_1 to equal elements in p_2. In order to show that it still preserves the logical order, we only need to show that for every $x' \in p_1$, $x' < x$ if and only if $f(x') < f(x) = y$. Let us show this claim in one direction (the other one is analogous). Assume $x' < x$. By definition, there exists $n \geq 1$ and elements $x_0, \ldots, x_n \in p_1$ such that $x' = x_0 \mathrel{\mathrm{D}} x_1 \mathrel{\mathrm{D}} \ldots \mathrel{\mathrm{D}} x_n = x$. By the compatibility of $\equiv$ and D, we also have $f(x') = f(x_0) \mathrel{\mathrm{D}} f(x_1) \mathrel{\mathrm{D}} \ldots \mathrel{\mathrm{D}} f(x_n) = y$. By the invariant (I2), y does not logically precede $f(x_{n-1})$, so it must be $f(x_{n-1}) < y$, and finally by transitivity $f(x') < y$. As for the maximality of f, since (I3) holds at the start of the loop, any extension to f must involve the element x that is being processed. Thus, if f can be extended, it is extended by the ghost statements in lines 9–10. □

Lemma 7.6.6. In a bounded connected stream s, all maximally matched prefixes of $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$ are equivalent.

Proof. Let $f \colon p_1 \to p_2$ and $g \colon q_1 \to q_2$ be two maximal matchings, where p_1 and q_1 are prefixes of s_1, and p_2 and q_2 are prefixes of s_2. Let $u_1 = s_1 \setminus p_1, u_2 = s_2 \setminus p_2$, and $v_1 = s_1 \setminus q_1, v_2 = s_2 \setminus q_2$ be the corresponding unmatched elements. To show the claim, it suffices to show that $p_1 \equiv q_1$.

Clearly the identity function $\mathrm{id} \colon p_1 \cap q_1 \to p_1 \cap q_1$ is a matching. Let $h \colon p_1' \to q_1'$ be a maximal matching between p_1 and q_1 that extends id; thus, p_1' and q_1' are prefixes such that $p_1 \cap q_1 \subseteq p_1' \subseteq p_1$ and $p_1 \cap q_1 \subseteq q_1' \subseteq q_1$. To show $p_1 \equiv q_1$, it suffices to show that $p_1' = p_1$ and $q_1' = q_1$.

First we note that for the matchings f, g, h, an analog of the invariant (I0) holds. In particular, the sets $p_1 \setminus p_1'$ and $q_1 \setminus q_1'$, as well as v_1 and v_2 are pairwise independent and unequal. Moreover, let us set $p_2' = f(p_1')$ and $q_2' = g(q_1')$. The sets $p_2 \setminus p_2'$ and $q_2 \setminus q_2'$ are also pairwise independent and unequal. Let us show the claim for $p_1 \setminus p_1'$ and $q_1 \setminus q_1'$. Suppose first that $x \in p_1 \setminus p_1'$ and $y \in q_1 \setminus q_1'$ are elements such that $x \mathrel{\mathrm{D}} y$. Since x and y are both in s_1, we either have $x < y$ or $y < x$. If $x < y$, then we have $x \in q_1$ since q_1 is a prefix, and consequently $x \in p_1 \cap q_1 \subseteq p_1'$, which is a contradiction. Likewise, $y < x$ leads to $y \in q_1'$, which is also a contradiction. Suppose now that $x \in p_1 \setminus p_1'$ and $y \in q_1 \setminus q_1'$ are elements such that $x \equiv y$. They cannot both be minimal, for we would be able to extend h with $h(x) = y$. Thus, one of x and y has a logical predecessor; say $x' \in p_1 \setminus p_1'$ is such that

$x' < x$. Without loss of generality, x' D x. Since $\equiv$ and D are compatible, this leads to x' D y, which we have just shown to be impossible.

We are now ready to prove $p'_1 = p_1$ ($q'_1 = q_1$ is analogous). For the sake of contradiction, suppose that $p'_1 \neq p_1$, that is, there exists an element $x \in p_1 \setminus p'_1$. Note that $x \notin q_1$, that is, $x \in v_1$. We send x to s_2 via f; we have $f(x) \in p_2 \setminus p'_2$. The element $f(x)$ cannot be in v_2 since $x \in v_1$ and $x \equiv f(x)$. The element $f(x)$ also cannot be in $q_2 \setminus q'_2$, since it is in $p_2 \setminus p'_2$ and $f(x) \equiv f(x)$. Hence, $f(x) \in q'_2$. Now we pull $f(x)$ back to s_1 via g and h. Set $x_1 = h^{-1}(g^{-1}(f(x)))$; we have $x_1 \in p'_1$. Since $x \notin p'_1$, x_1 and x are distinct elements such that $x_1 \equiv x$. We can continue iterating the described process. Suppose we have defined distinct elements $x_1, \ldots, x_n \in p'_1$ for some $n \geq 1$ such that $x \equiv x_1 \equiv \ldots \equiv x_n$. We send x_n to s_2 via f and note that neither $f(x_n) \notin v_2$ (otherwise we would have $x \in v_1$, $f(x_n) \in v_2$, and $x \equiv f(x_n)$) nor $f(x_n) \in q_2 \setminus q'_2$ (otherwise we would have $f(x) \in p_2 \setminus p'_2$, $f(x_n) \in q_2 \setminus q'_2$, and $f(x) \equiv f(x_n)$). Therefore, $f(x_n) \in q'_2$ and we can bring it back to s_1 via g and h to get a well-defined element $x_{n+1} = h^{-1}(g^{-1}(f(x_n))) \in p'_1$. Suppose $x_{n+1} = x_k$ for some k with $1 \leq k \leq n$. By removing k layers of application of $h^{-1} \circ g^{-1} \circ f$, we conclude that $x_{n+1-k} = x$, which cannot be since $x_{n+1-k} \in p'_1$ and $x \notin p'_1$. Therefore, x_{n+1} is distinct from all previously defined elements.

By defining the described process, we have shown that there are infinitely many distinct elements in p'_1, which cannot be since p'_1 is a finite set. Hence, $x \in p_1 \setminus p'_1$ cannot exist in the first place, and we have established that $p_1 = p'_1$. Analogously, we have $q_1 = q'_1$, and since $p'_1 \equiv q'_1$, we also have $p_1 \equiv q_1$. Finally, using $p_1 \equiv q_1$ as a link, we establish that $p_2 \equiv p_1 \equiv q_1 \equiv q_2$, that is, all the maximally matched prefixes in s are equivalent. □

Lemma 7.6.7. If StreamsEquivalent returns **false** in line 12 while processing $s[n]$ for some $n \geq 1$, then the connected stream $s[:n]$ is finitely distinguishable.

Proof. Let $s[n] = (x, 1)$ and assume that ProcessElement returns **false** in line 12 when processing $s[n]$. Since the condition in line 11 was satisfied, there exists $y \in u_2$ such that x D y. Without loss of generality, let y be a minimal such element in u_2.

The challenge here is that even though f can never be extended to match either x or y, it is plausible that $s[:n]$ can nevertheless be extended to s' with a completely different matching that somehow accommodates both elements. To show that such a scenario is impossible, suppose s' is an extension

of $s[:n]$ such that $\pi_1(s') = s'_1 \supseteq s_1$, $\pi_2(s') = s'_2 \supseteq s_2$, and suppose $g\colon s'_2 \to s'_1$ is a matching (it helps to view g in the direction opposite of f). Since x D y and $g(y) \equiv y$, we have x D $g(y)$, so x and $g(y)$ are logically ordered in s'_1. There are three possibilities: $g(y) = x$ (the elements are identical), $g(y) < x$, or $x < g(y)$.

The first possibility can easily be discarded. From $g(y) = x$ it follows that $x \equiv y$. The elements x and y cannot both be minimal elements unmatched by f, otherwise x would have been matched in lines 7–10. Therefore, either there is $x' \in u_1$ such that $x' < x$ and x' D x, or there is $y' \in u_2$ such that $y' < y$ and y' D y. In the former case we have x' D y, contradicting the invariant (I0), and in the latter case we have x D y' and $y' < y$, contradicting the choice of y as a minimal unmatched element such that x D y.

The second possibility, that $g(y) < x$, can be discarded as follows. Set $y_0 = y$. The element $g(y_0)$ cannot be in u_1 as that would break the invariant (I0); therefore it has to be in p_1. We can send it to p_2 via f; let $y_1 = f(g(y_0))$. Since $y_0 \notin p_2$, y_0 and y_1 are distinct elements such that $y_0 \equiv y_1$. We iterate the process: suppose we have defined the elements $y_1, \ldots, y_n \in p_2$ for some $n \geq 1$ such that all of them are equal to y_0, and all of them together with y_0 are distinct. Since $g(y_n)$ D x, $g(y_n)$ and x are logically ordered. It cannot be $g(y_n) \geq x$, as that would imply $g(y_n) > g(y_0)$ and consequently $y_n > y_0$. But then, since $y_n \in p_2$ and p_2 is a prefix, we would have $y_0 \in p_2$, which is a contradiction. Thus, $g(y_n) < x$ and consequently $g(y_n) \in p_1$ due to the invariant (I0). Hence, $y_{n+1} = f(g(y_n))$ is a well-defined element such that $y_{n+1} \in p_2$. Clearly y_{n+1} is distinct from y_0, and $y_{n+1} \equiv y_0$. Suppose $y_{n+1} = y_k$ for some k with $1 \leq k \leq n$. By "pealing off" k layers of applications of f and g, we would get $y_{n+1-k} = y_0$, which is a contradiction. Therefore, the new element is distinct from every previously defined element. We have thus defined infinitely many distinct elements in the finite set p_2, which is a contradiction. Hence, $g(y) \not< x$.

The third possibility, that $x < g(y)$, is discarded by a similar argument. The idea is again to start from $x_0 = x$ and define an infinite sequence of distinct elements $x_1, x_2, \ldots$ in p_1, all of which are equal to and distinct from x_0. However, the argument that shows the sequence is well-defined is slightly different. Similarly as before, given x_n for $n \geq 1$ we start by arguing that $g^{-1}(x_n) \in p_2$. We first establish that $g^{-1}(x_n) < y$, as otherwise $g^{-1}(x_n) \geq y > g^{-1}(x_0)$ would imply $x_n > x_0$ and consequently $x_0 \in p_1$. Next, if $g^{-1}(x_n) \in u_2$, then we argue as in the first possibility: either x and $g^{-1}(x_n)$ can be matched, or there is $x' < x$ in u_1 such that x' D $g^{-1}(x_n)$, contradicting the invariant

(I0), or there is $y' < g^{-1}(x_n) < y$ in u_2 such that $x \mathrel{D} y'$, contradicting the minimality of y. Hence, $g^{-1}(x_n) \in p_2$ and $x_{n+1} = f^{-1}(g^{-1}(x_n))$ is well-defined. Showing that it is distinct from previously defined elements is done using the same argument as before. Thus, we again reach a contradiction, and $x \not< g(y)$.

By discarding all three possibilities, we conclude that the extension of $s[:n]$ to a connected stream s' such that $s'_1 \equiv s'_2$ does not exist. Hence, $s[:n]$ is finitely distinguishable. □

Theorem 7.6.8. Algorithm 7.8 is correct.

Proof. We first show that Algorithm 7.8 satisfies the correctness condition (I). If StreamsEquivalent returns **true** in line 15, then the whole connected stream s has been processed. Hence, s is bounded. Moreover, in this case both u_1 and u_2 are empty, implying $p_1 = s_1$ and $p_2 = s_2$. Therefore, $s_1 \equiv s_2$ follows from the invariant (I3). Conversely, if s is bounded and s_1, s_2 are equivalent, by Lemma 7.6.7, StreamsEquivalent does not return **false** on line 12, and the loop in lines 5–14 finishes. By the invariant (I3) and Lemma 7.6.6, f must fully match s_1 and s_2. Hence, u_1 and u_2 are empty and StreamsEquivalent returns **true**.

Next, we show the correctness condition (II). If StreamsEquivalent returns **false**, it either returns **false** in line 12 and s is finitely distinguishable by Lemma 7.6.7, or it returns **false** in line 15. In the latter case, s is bounded and one of u_1 and u_2 is not empty. Hence, s_1 and s_2 are distinguishable by the invariant (I3) and Lemma 7.6.6. Conversely, if s is bounded and s_1, s_2 are distinguishable, by (I) the function StreamsEquivalent does not return **true**, so it returns **false**.

It remains to show that if s is finitely distinguishable, then StreamsEquivalent returns **false** in line 12 for the first position n such that $s[:n]$ is finitely distinguishable. Clearly StreamsEquivalent does not return **false** on line 12 when processing $s[k]$ for $k < n$, since in that case by Lemma 7.6.7 already $s[:k]$ would be finitely distinguishable. Let $s[n] = (x, 1)$ and let u_1 and u_2 be the logically ordered sets of unmatched elements at the start of the loop when processing $s[n]$. If x can be matched with an element $y \in \min u_2$, then we extend $s[:n]$ with $\iota_1(u_2 \setminus \{y\})$ followed by $\iota_2(u_1)$, with the elements of $u_i, i \in \{1, 2\}$ given in the order in which they appear in $s[:n-1]$. Likewise, if $x \not D\!\!\!/ y$ for every $y \in u_2$, then we extend $s[:n]$ with $\iota_1(u_2)$ followed by $\iota_2(u_1 \cup \{x\})$. In either case, from the invariants (I0)–(I3) it follows that Algorithm 7.8 would decide that the two streams in the extension

are equivalent, and since the algorithm is correct for bounded equivalent streams, the streams would indeed be equivalent. Therefore, in both cases the connected stream $s[{:}n]$ would not be finitely distinguishable. Since $s[{:}n]$ is finitely distinguishable, the only remaining option is that there exists $y \in u_2$ such that $x \mathrel{\mathrm{D}} y$, and hence StreamsEquivalent returns **false** in line 12 when processing $s[n]$. □

7.6.4. Optimality

When it comes to stateful stream processing programs, space usage is an important topic. If a stateful stream processing program inadvertently stores too much of the stream's history, since the stream is potentially unbounded, the program's space usage may grow unboundedly as well. In case of Algorithm 7.8, its space usage may indeed grow unboundedly. Since Algorithm 7.8 stores sets of unmatched elements, it can be forced to keep the complete history of the connected stream it takes as input. For example, this would happen on a connected stream s such that $\pi_1(s) = \langle a, a, \ldots \rangle$, $\pi_2(s) = \langle b, b, \ldots \rangle$, with $a \not\equiv b$ and $a \not\mathrel{\mathrm{D}} b$. However, it turns out that for a correct equivalence-matching algorithm there is no way around this: a correct equivalence-checking algorithm must store a certain amount of unmatched elements in one way or another. In each step, Algorithm 7.8 stores minimal sets of unmatched elements (the complements of maximally matched prefixes), and as we show in this subsection, in this sense it is optimal.

Recall that by Lemma 7.6.6, in a bounded connected stream s with $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$, all maximally matched prefixes of s_1 and s_2 are equivalent. It follows that their complements—minimal sets of unmatched elements—are equivalent as well. More precisely, let (p_1, p_2) and (q_1, q_2) be two pairs of maximally matched prefixes such that $p_i, q_i \subseteq s_i$ for $i \in \{1, 2\}$, and let $u_i = s_i \setminus p_i$ and $v_i = s_i \setminus q_i$ for $i \in \{1, 2\}$. Then $u_1 \equiv v_1$ and $u_2 \equiv v_2$. This allows us to define up to equivalence a function $u(s) = (u_1, u_2)$, where u_1 and u_2 are any minimal sets of unmatched elements in s_1 and s_2. We write $(u_1, u_2) \equiv (v_1, v_2)$ to mean $u_1 \equiv v_1$ and $u_2 \equiv v_2$.

If a bounded connected stream s with $u(s) = (u_1, u_2)$ is not finitely distinguishable, then an analog of the invariant (I0) holds for u_1 and u_2: for every $x \in u_1$ and $y \in u_2$, $x \not\equiv y$ and $x \not\mathrel{\mathrm{D}} y$.

Theorem 7.6.9. Algorithm 7.8 is optimal. More precisely, let A be any other *correct* algorithm for the equivalence-checking problem. If s and s' are two bounded connected streams that are not

finitely distinguishable, and if Algorithm 7.8 reaches a different state after processing s and s', then A reaches a different state after processing s and s'.

Proof. The state stored by Algorithm 7.8 on processing a string s is $u(s)$. Suppose a correct equivalence-checking algorithm reaches the same state after processing s and s'. Let $u(s) = (u_1, u_2)$ and $u(s') = (u'_1, u'_2)$. We extend both s and s' with $\iota_1(u_2)$ followed by $\iota_2(u_1)$. It is not difficult to see that the two connected streams in the extension of s are equivalent, and the streams in the extension of s' are not equivalent. However, since the algorithm is deterministic, it would come to the same decision for both extensions, which contradicts its correctness. □

7.7. Practical Bounds on Space Usage

While the space used by Algorithm 7.8 can grow with the input stream in the worst case, Theorem 7.6.9 shows that it is impossible to write an algorithm which uses a smaller amount of space. In addition to this result, it is possible to give concrete bounds on the space usage in certain cases. Here we discuss some patterns that we have encountered, including the examples we implemented in Section 7.8, and how bounds on the space usage can be derived for these patterns.

The simplest pattern is differential testing of sequential outputs: both streams are completely ordered, i.e., any two events are dependent. In this case, any differential testing algorithm must at least keep track of the difference of the two streams seen so far (assuming their prefixes are equal). For example, suppose both streams are sequences of integers, and one stream has seen m integers and the other has seen n integers, where $m < n$. Then if the first m integers are equal, any matching algorithm must keep track of the remaining $(n - m)$ integers. Thus, the space usage of the algorithm is bounded by the maximum *drift* between the two streams, defined as the difference in the number of events produced. In practice, such drift is typically bounded since inputs arrive at the same rate for both the implementations, and most systems try to ensure that no operator in the stream processing dataflow graph lags behind the others by accumulating a large queue of unprocessed inputs. This dependence relation pattern (and the resulting bound) occurs in the case studies of Sections 7.8.2 and 7.8.3.

A second common pattern is *key-based parallelism*, where events with the same key are dependent, but events with different keys are independent. In such a case, considering the drift between the

two streams is not enough. For example, suppose there are only two keys, a and b, and one stream produces n as, but the other stream produces n bs. Then although the two streams are producing the same number of items, because stream 1 never produces an a and stream 2 never produces a b, any algorithm for correct matching must keep at least the as and the bs so far until they are matched on the other stream. To address this, we can obtain a bound on the space by considering the drift *per key*. In general, "keys" can be generalized as dependent subsets of events, and a bound can be obtained by taking the maximum drift on any dependent subset, together with the number of independent keys in the input. This dependence relation pattern (and the resulting bound) occur in the case study of Section 7.8.1. Related to key-based parallelism, *fully independent parallelism* (where all input events are independent) occurs in the case studies of Sections 7.8.3 and 7.8.4.

Finally, we have encountered cases where the input includes special synchronization events, such as punctuation marks and end-of-day markers, as described in Section 7.5. These events are dependent on all other events. If both input programs to the differential testing algorithm produce these events at regular intervals, then the space usage becomes bounded. In particular, suppose that the *frequency* of such events is at least one in every k events, and the *drift* restricted only to such events is d. Then the space usage of our algorithm is at most $k \times d$.

To ensure that bounded drift holds between the two streams in practice, one approach would be to leverage *back-pressure* of the underlying system [80, 170, 75]. In particular, back-pressure may prevent drift from growing in cases where one implementation is significantly faster than the other, since the system would slow down the fast implementation to prevent unbounded buffering.

7.8. Evaluation

We implemented the matcher algorithm in DiffStream, a differential testing library written in Java. The matcher can be used to test programs in any distributed stream processing system given an output interface. For our case studies we chose Flink as the target platform because it is one of the most widely used distributed stream processing frameworks [215]. We integrated DiffStream with JUnit-QuickCheck [152] to support generation of streams of random input values.

Our first case study (Section 7.8.1) is used to qualitatively measure the developer effort needed to test an application with non-trivial ordering dependency in its output. In the second case study

(Section 7.8.2) we demonstrate that getting performance benefits from parallelization in Flink might require an elaborate implementation and we illustrate how our tool can be used to streamline that process. In the third case study (Section 7.8.3), we show that our framework is successful in finding real bugs while largely avoiding false positives by adapting a set of non-deterministic MapReduce programs from the literature [277]. The final case study (Section 7.8.4) investigates the performance overhead when using DiffStream for online monitoring of long-running applications.

7.8.1. Taxi Distance

This case study illustrates the process that one has to follow in order to test their implementation using our tool. Recall the taxi distance example in Section 7.3. This example shows two seemingly equivalent implementations of the same query that produce different results in the presence of parallelism. Here is an instantiation of that example in Flink; the first implementation preserves the order of events for each key, while the second one does not:

```
inStream.keyBy("taxiID").project("taxiID", "position");
```

```
inStream.project("taxiID", "position").keyBy("taxiID");
```

Note that both implementations preserve the order when executed sequentially. Since such subtle differences are difficult to spot manually, we would like to be able to test a parallel implementation against a sequential one, before deploying it. A slightly simplified example of a test that can exercise this bug using our framework is shown in Figure 7.10. First, the Flink execution environment is initialized and the dataflow graph is setup. Then, a random input stream is generated and fed to both implementations, that are finally compared using the matcher for output equivalence.

Notice that the final argument of the matcher is a lambda expression representing the dependence relation that is expected from the consumer of the output. This specific instantiation represents the dependence that was shown in Figure 7.4a—i.e., that two items are dependent (and thus must be ordered) if they have the same `taxiID`. If the user did not want to test differences in the ordering of the output, they can use `(ev1, ev2) -> false` as the dependence relation.

In order to compare the effort required to write a test with and without using our framework, we manually implemented a test that exercises this bug. The manually implemented matcher spans two

```
public void testKeyBy() throws Exception {
    StreamExecutionEnvironment env = ...;

    DataStream input = generateInput(env);

    StreamEquivalenceMatcher matcher =
        StreamEquivalenceMatcher.createMatcher(
            sequentialImpl(input), parallelImpl(input),
            (ev1, ev2) -> ev1.taxiID == ev2.taxiID);

    env.execute();
    matcher.assertStreamsAreEquivalent();
}
```

Figure 7.10: DiffStream example code.

Java classes, totaling around 100 LoC (in contrast to the 13 LoC of the test in our framework shown in Figure 7.10). This does not include input generation, for which we used JUnit QuickCheck. The manually implemented matcher keeps two hashmaps—one for each implementation—that map keys to lists, in order to encode the dependence of events of the same key. It appends each output item to the list associated with its key. After the two implementations stop executing, it checks that the two hashmaps represent equivalent output. Note that this manual matcher is not online, in the sense that the two implementations have to stop producing outputs for it to make a decision. We also implemented an online version of it by extending it with 30 more lines of code.

The important point is that the dependence relation abstraction enables the design of a reusable testing framework that can be used for testing applications with different ordering requirements on their outputs. In contrast, the main drawback of the manual matcher is that it is tied to a specific ordering requirement on the output; whenever a user wants to write a test that requires a different output dependence relation, they would have to implement a new matcher that maintains the output in a data structure suitable for the specific dependence. This can quickly become an overhead if the user wants to write tens or hundreds of tests for different parts of their application.

In summary, we have shown that using our tool to write a test for a stream processing application is significantly easier than writing custom tests (∼10 LoC vs. ∼100 LoC). In addition, our tool offers additional flexibility, as it can be used to test any two implementations just by changing the dependence relation given to it. This flexibility reduces the effort needed to implement tests for

an application. It also exposes ordering requirements, forcing the developer to think about them explicitly.

7.8.2. Topic Count

The main goal of our second case study is to show that achieving parallelism in distributed stream processing programs can be very difficult and require a drastically more complicated solution than the sequential code. In particular, we consider the example introduced in Section 7.3, that involves counting topics associated with words in a long document and outputting the most frequent topic as the overall topic of the document. The documents are streamed word by word, with end-of-file markers delineating words in different documents. In the sequential solution (Figure 7.2, top), we process each word by querying the topic for that word, then updating the total count for that topic; when we get end-of-file, we emit the counts. We feed this output to a second operator *count max*, which finds the maximum over all topics of the count.

At first, one may think that going from a sequential to a parallel program is simply a matter of setting the Flink's parallelism parameter to more than 1. Unfortunately, this is not the case. Consider the first operator in the sequential dataflow shown in Figure 7.2 (top). The problem with parallelizing this operator is that an end-of-file marker for a particular document would only be processed by a single sub-operator; thus the other sub-operators would not be able to properly delineate words from this document and the next one. Another way of stating the problem is to say that even though the words themselves are independent, they are dependent on the end-of-file markers, and thus the obvious parallelization is not possible. A differential test that compares the sequential dataflow to the same dataflow with parallelism set to more than 1 quickly discovers that the two versions are indeed not equivalent.

Instead, a correct parallel solution (Figure 7.2, bottom) works as follows. We first attach logical timestamps to each word in the input, corresponding to the number of the document that we are currently processing. We then replace end-of-file markers, which act as explicit punctuation in the stream, with punctuated watermarks—a mechanism in Flink that informs the dataflow operators about the passage of logical time. Unlike the explicit end-of-file markers that cannot be shared by multiple sub-operators, the watermarks are seamlessly propagated by the system. In effect, they allow us to break the explicit dependence between words and end-of-file markers in the input stream,

Solution	Lines of Code	Speedup due to parallelism (par.) par. 2	par. 4	par. 6	par. 8
Sequential	68	–	–	–	–
Correct Parallel	133	2.01	4.33	5.0	4.99

Figure 7.11: Results of the second case study on difficulty of writing parallel code.

allowing the later stages of the dataflow to be parallelized. We parallelize with *key-by* (keys can be assigned arbitrarily to words), and query the database for each word. The next operator is a tumbling window which uses the logical timestamps from earlier to form the window of events *for a single document*, still parallelized. Finally, we sum up the values in each window by topic, and in the last stage *count max* we find the maximum over all topics of the count.

In our search for a correct parallel solution, we consulted with Flink users on the Flink mailing list. Several iterations of feedback were needed to find a correct parallel implementation, and this shows that it is not obvious. Having the differential testing framework helped guide the search by quickly dismissing wrong implementations. The final solution is the dataflow shown in Figure 7.2 (bottom), consisting of 6 dataflow operators and twice as many lines of code as the sequential solution.

It is not necessarily true that a solution that seems parallel achieves a speed-up in practice. We therefore finally need to measure the performance of the parallel solution to show that it indeed takes advantage of parallelism and scales performance with the level of parallelism. We evaluated our parallel solution on an input stream of 5 documents, each consisting of 500,000 words randomly selected from a list of 10,000 most common English words. Each word had previously been randomly assigned one of 20 topics, and the association had been stored in a standalone Redis key-value store. The purpose of having the Flink program query the Redis store was to simulate a series of non-trivial operations that would benefit by being parallelized. We executed this experiment on a server with an Intel Xeon Gold 6154 processor and 384 GB of memory. The results of the evaluation are shown in Figure 7.11. By increasing parallelism from 1 to 6, the execution time decreases from 80 s to 16 s (on our setup, the benefits taper off after 6).

In summary, although there is a clear performance benefit in having a parallel solution (Figure 7.11), the correct solution is difficult to find, as evidenced qualitatively by our search and discussions on the Flink mailing list, and quantitatively because it has about twice as many lines of code as the

sequential solution. Applying differential testing to the parallel solution ensures that bugs were not introduced in the process.

7.8.3. Real-World MapReduce Programs

To determine whether our testing framework can successfully find bugs in real-world programs, we surveyed the literature for empirical studies which have collected and categorized bugs in stream- and batch-processing programs. We excluded works that focus on job failures and performance issues [239, 167, 180, 289], as they do not provide examples of semantic bugs where the output might be incorrect. In contrast, Xiao et. al. [277] study nondeterminism due to parallelism in MapReduce jobs in production workflows, identifying both real bugs as well as several nondeterministic code patterns which are bug-free. This empirical study provides a good starting point to evaluate whether our framework can successfully identify the bugs, while not falsely flagging the bug-free examples.

By examining 507 custom (i.e., user-written) reduce functions, the study identifies 5 common reducer patterns (spanning 258 of the custom reducers) which are *non-commutative* on general input data, meaning that there is potential for nondeterminism in the output due to parallelism. An example reducer pattern reported by the study is shown in Figure 7.13 (left). While the original custom reducers are not publicly available, the 5 code patterns are nevertheless minimal test cases which exhibit the same behavior, and a large majority of non-commutative reducers (88%) were found to fall into these categories. However, as the study notes, there are two reasons why code written using these patterns might not be erroneous. First, with certain input assumptions, the nondeterminism may disappear (this is possible in 4 out of 5 patterns). Second, nondeterminism may simply be acceptable for the particular application (this is most conceivable in 3 out of 5 patterns). We therefore evaluate our testing framework for each of the five patterns, considering three possibilities for the application-specific requirements: determinism (nondeterminism is not acceptable), determinism under certain input assumptions, and no determinism required. We want to answer the following questions, corresponding to these three possibilities:

Q1. If determinism is required on all inputs, can we write a test which successfully detects the nondeterminism and reports a bug?

Q2. If determinism is required but only under certain input assumptions, can we write a test using those input assumptions to avoid a false positive?

Code pattern	Determinism	Application-Specific Requirements	
		Determinism under input assumptions	None (nondeterminism acceptable)
`SingleItem`	✓	✓	n/a
`IndexValuePair`	✓	✓	n/a
`MaxRow`	✓	✓	✗
`FirstN`	✓	✓	✗
`StrConcat`	✓	n/a	✓

Figure 7.12: Results of the MapReduce case study. A ✓ indicates successfully identifying the bug in the first column, and successfully avoiding a false positive in the second and third columns, for each of the 5 reducers implemented.

Q3. If determinism is not required at all, can we write a test which avoids a false positive?

For each question, we implement the reducer adapted to the streaming setting, and run DiffStream to compare a sequential and parallel version. The results are summarized in Figure 7.12, where columns 1, 2, and 3 correspond to the above three questions, respectively.

For example, in the `IndexValuePair` pattern in Figure 7.13 (left), the user wrote a reducer to aggregate input items with two input fields, `x` and `y`, by accumulating them in a map where the value of field `x` is set to the value of field `y`. The reducer is nondeterministic in general because there may be multiple updates to the same field: multiple items with the same `x` and a different `y` (which may be out-of-order due to parallelism in the map stage of MapReduce). However, if the input satisfies a *functional dependency* where `y` is a function of `x`, the pattern becomes deterministic. Without knowing the user's intention, it may be that determinism is required, and the functional dependency is not satisfied, in which case this code is a bug (Figure 7.12, first column); or it may be that determinism is required, and the functional dependency is satisfied, in which case this code is not a bug (Figure 7.12, second column). Though unlikely for this particular pattern, the study authors also noted for some patterns that nondeterminism in the output may be acceptable, despite the functional dependency not being satisfied (Figure 7.12, third column).

We implemented each of the 5 reducer patterns in Flink. We start by translating each reducer directly to an aggregator (`AggregateFunction` in Flink); for the `IndexValuePair` reducer, this yields the code in Figure 7.13 (right). To adapt the reducer to the streaming setting, a tumbling window is applied to the input stream, and the reducer is applied to get a result for each window. We used the same input stream item type (`Item`) for all examples, which contains fields `x` and `y`. Before constructing the tumbling window, we used an identity map operator to shuffle the data for each key,

```
Dictionary<int, int> dict
    = new ...;
foreach (Row row in input) {
    int x = row["x"].Integer;
    int y = row["y"].Integer;
    dict[x] = y;
    // ...
}
```

```
public class IndexValuePairReducer ... {
    public Map<Integer, Integer>
            createAccumulator() {
        return new HashMap<>();
    }
    ...
    public Map<Integer, Integer> add(
        Item in, Map<Integer, Integer> outmap
    ) {
        outmap.put(in.x, in.y);
        return outmap;
    }
    ...
}
```

Figure 7.13: Example MapReduce code: the `IndexValuePair` reducer pattern reported by the study from production MapReduce jobs [277] (left), and our implementation of the pattern in Flink (right).

so that the order is nondeterministic (thus potentially exposing bugs due to parallelism). We then compared the parallel version of this pipeline with the sequential one using our matcher to look for a bug (difference in the outputs).

To evaluate Q1, we generated arbitrary input data and fed it to the sequential and parallel versions. With enough input data (3000 input data items is sufficient), using a small number of keys and possible input data items, our tester consistently detects the incorrectly parallelized program for all 5 patterns. Concretely, of the 5 confirmed bugs found in production code by the previous study (these do not correspond to the 5 patterns), 4 of the 5 are of this nature, so our test cases likely would have identified these 4 bugs.

To evaluate Q2, we use the same setup but with a custom generator for the input data items. For all patterns except one (called `StrConcat`), the output is deterministic if certain assumptions are made on the input; the custom generator enforces these assumptions. For example, for the fields `x` and `y` which are used in the `IndexValuePair` example of Figure 7.13, we enforce the requirement that `y` is a function of `x`. We show that in 4 out of 4 patterns, the output successfully passes our tester, i.e. we avoid a false positive in these 4 scenarios.

Finally, to evaluate Q3, we look at three patterns where it is conceivable that nondeterminism in the output is acceptable. For the first two of these, we are unable to write a test which avoids a false positive. The `MaxRow` pattern involves finding the value of one field such that another field is maximized; it is nondeterministic because there may be multiple values which achieve the maximum.

In such a case, differential testing results in a false positive because the two programs may return different values, even though they are both correct. Similarly, the `FirstN` pattern is a reducer which discards all but the first N elements that are seen; on inputs with more than N elements, differential testing results in a false positive for a bug. However, we can avoid the false positive in the last `StrConcat` pattern. This reducer consumes a sequence of input items and concatenates them into a single string separated by a special character (say, `@`). It is nondeterministic because the concatenation is non-commutative, but it is likely that the application requirements consider this acceptable. For this pattern, we implement a custom-defined equality on output data items to define when strings separated by `@` are equal; using this, the pattern is able to pass our tester. Additionally, we implement a second version of `StrConcat` which is more suited to the streaming setting: instead of collecting all items in a single `@`-separated string, we output the items as a data stream. In this case, our tester successfully reports a bug when nondeterminism is undesirable; but when the dependence relation is used to indicate that nondeterminism is acceptable, the test passes.

Summary. When nondeterminism is undesirable, we have written tests to successfully identify it (if it exists) for 5 out of 5 reducer patterns. Of the reducer patterns where nondeterminism might not be present due to input data assumptions, we show how an input data generator can be used to cause the programs to pass our tester for 4 out of 4 patterns. Finally, in the reducer patterns where it is conceivable that nondeterminism in the output might be acceptable, we show how using the dependence relation *or* custom equality with our tester can successfully make the test pass for 1 of the 3 patterns (`StrConcat`). In total, out of the scenarios where the reducer might conceivably not be buggy (the 4 and 3 patterns just mentioned, respectively), we avoid a false positive in 5 out of 7.

7.8.4. Performance for Online Monitoring

In Sections 7.6.4 and 7.7 we discussed theoretical lower and upper bounds on DiffStream's space usage. We showed that DiffStream is optimal, but also that depending on the application and dependence relation, space usage in the worst case could grow unboundedly. In this subsection we evaluate the performance of the matcher in practice for monitoring a realistic streaming application. We aim to demonstrate DiffStream's applicability in online monitoring scenarios, e.g. when testing an application under production load for bugs, or for multi-version execution—a method commonly used to safely update production software [259, 154, 191].

We broadly aim to answer the question: what is the overhead of the matcher? Relevant metrics are both its memory usage (reflecting unmatched items) and its impact on the application performance (throughput and latency). We evaluate the following three questions:

Q1. What is the effect of the equivalence matcher on the maximal throughput of the application?

Q2. What is the memory footprint of the matcher? How does the memory usage vary over time?

Q3. What is the latency of the matcher, i.e. how much time does it take for the matcher to process a single event?

For the application, we chose the Yahoo Streaming Benchmark [78], a standard performance benchmark for stream processing systems. The Yahoo Streaming Benchmark implements a simple advertisement processing application that receives a stream of advertisement events in the JSON format. The events are parsed, filtered for the ad view events, joined with the ad campaign data from an external database, aggregated over time windows, and stored into the database. The application integrates a Kafka queue, a Flink program, and a Redis database, which makes it representative of streaming applications which interact with external services. In our evaluation, we run a sequential and a parallel version of the advertisement program, with the parallelism parameter set to 2. The matcher compares the streams of the two programs after the join with the ad campaign data, but before the aggregation. The aggregation simply counts the number of ad view events per campaign id and per time window, so it does not depend on the order of events. Thus, we use an empty dependence relation (all events are independent).

To answer Q1, we modified the ad event producer to steadily increase the input rate over time. We then executed two experiments: one with the matcher and one without it. (We simulated the absence of the matcher with a dummy matcher that ignores every event.) Both experiments started with an input rate of 40,000 events/s and the acceleration of 10 events/s^2 and ran for 2,500 seconds, allowing the input rate to increase to 65,000 events/s. The expected ideal throughput of the matcher is 2/3 of the input rate due to (i) the duplication for the two versions of the program, and (ii) the filtering step, which filters approximately 2/3 of the events and leaves 1/3 that finally reach the matcher. The measured throughput is shown in Figure 7.14a. Initially, in both experiments the measured throughput matches the ideal throughput. After 1,737 seconds, the experiment with the matcher reaches the throughput of 38,072 events/s, after which it rapidly drops and starts fluctuating. In

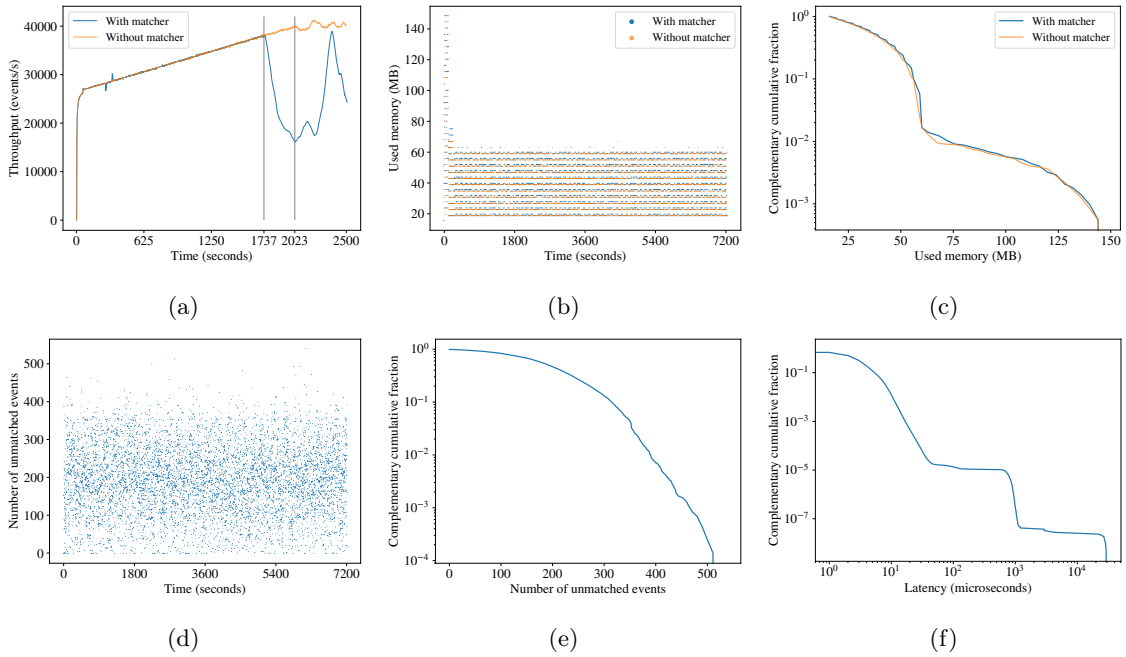


Figure 7.14: Results of the fourth case study: performance measurements of monitoring an application with DiffStream on the Yahoo streaming benchmark over a span of 2 hours, compared to the same application without the DiffStream matcher.

the experiment without the matcher, the throughput continues to rise with the input rate until it reaches 40,082 events/s after 2,023 seconds, after which it also starts fluctuating. Thus, using the matcher results in approximately 5% decrease in the maximal throughput.

To answer Q2 and Q3, we implemented a rudimentary memory profiler that outputs the total heap memory used by the Java Virtual Machine running the Flink program in 1-second intervals. In addition, we measured the latencies of the matcher by measuring the time it takes to process each event; we stored the latencies in a file during execution. We executed two experiments, with and without the matcher, with a constant input rate of 45,000 events/s and the duration of 7,200 seconds (2 hours). The input rate of 45,000 events/s translated into the throughput of 30,000 events/s, which was stable during the execution.

Figure 7.14b shows a scatter plot of the memory samples taken during execution. In particular, the total memory used by the program stays bounded throughout the duration of the experiment. Figure 7.14c visualizes the same information in a different way: for an amount of used memory x on the x-axis, the corresponding value on the y-axis is the fraction of memory samples that exceed

x. The two figures together show that there is virtually no difference in the memory usage in the two experiments. We get a more fine-grained view if we look at the number of unmatched events that the matcher stores during execution. Figure 7.14d shows a scatter plot of samples of unmatched events taken in 1-second intervals during execution. In particular, in almost all samples, the number of unmatched events is below 500. This is even more clearly illustrated in Figure 7.14e, which shows the fraction of samples exceeding the given number of unmatched events. Thus, to answer Q2, the memory footprint of the matcher is bounded and negligible relative to the memory used otherwise and the throughput of 30,000 events/s.

Finally, to answer Q3, Figure 7.14f shows the fraction of recorded latencies that exceed a given number of microseconds. The mean latency is 2 microseconds. While the maximal latency is 30 milliseconds, this is rare: 98.77% of latencies are at most 10 microseconds, and $> 99.99\%$ of latencies are at most 100 microseconds.

Summary. The results show that the overhead of running DiffStream in practice is low. First, the matcher results in a modest 5% reduction in maximal throughput. Second, the memory usage is stable over time and reflects the theoretical optimality for applications where drift between the two streams is bounded: the number of unmatched elements is $< 1.67\%$ compared to the number events that it processes. Finally, the latency of the matcher is at most 100 microseconds in $> 99.99\%$ of cases, which competitively meets streaming performance standards.

7.9. Discussion

As presented, the input to DiffStream consists of the two programs as well as a *dependence relation* which is used to describe which output events must be produced in order, and an optional custom equality which is used to compare output events. It is also possible to view DiffStream as being a runtime assertion checker: it checks assertions of the form

```
assert s1 == s2
```

where `s1` and `s2` are streams of a given stream type S, and == is stream equivalence $s_1 \equiv s_2 \bmod S$. Moreover, it does this as efficiently as possible in a streaming complexity sense.

This is how I like to view the library today; an online stream diff calculator. Given a generator and two versions of the program under test, asserting stream equality on their outputs is the usual use case for DiffStream. However, there are other use cases for an equality test, such as validating that a result stream computed by two different nodes is the same.

Keeping in mind the stream-diff-calculator view, here are some possibilities for extension that I currently think would be promising:

- First, the library could be extended not just to check equality (yes or no answer), but to truly compute a diff between the two streams. The core algorithmic problem would be more challenging.

- Second, the algorithm could be revised to avoid the worst-case behavior described in Section 7.7 if we relax a simple requirement: if we require that differences between the two streams are detected *with high probability*, rather than necessarily. In particular, one could make use of hash functions for this purpose. Consider the failure case where the two input streams are bags. Rather than store the full diff of the input streams, could one just store a sum of the hash values of each bag? It is mathematically rare for $\sum_{s \in S_1} \texttt{hash}(s) = \sum_{s \in S_2} \texttt{hash}(s)$ unless $S_1 = S_2$.

An honest limitation of DiffStream is that it focuses on bugs due to parallelism. In fact, this is only a small class of all bugs; and even smaller if we exclude bugs that could be caught through unit testing without requiring nontrivial dependence relations. In future work, it would be prudent to target other classes of bugs, including those due to node or network faults. Additionally, we would like to generalize the definition of correct behavior, which currently assumes that the output should be determined up to allowed reordering and equality of data items. There are cases where this is too strong, for instance when operations are approximate or randomized.

Another limitation of DiffStream is that it does not address the problem of input data generation, but instead uses an off-the-shelf generator (JUnit-QuickCheck). There is an interesting problem of generating input streams of a type S, taking into account the type of S, which we don't really consider in this work. To clarify, DiffStream does check for type safety and determinism, but only on a specific example trace at a time. That is, on an input stream and an equivalent input stream, it

can check whether the outputs are equivalent. But, as with all runtime testing techniques, this only allows testing finitely many input traces. This is the reason for the Partial entries in Figure 4.4.

DiffStream is open source and is available on GitHub[2].

[2] https://github.com/fniksic/diffstream

CHAPTER 8 : Performance Bounds

It is well known that even for simple calculations it is impossible to give an a priori *upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic.*

—Rabin and Scott in "Finite Automata and Their Decision Problems," 1959 [228]

In this section we describe data transducers, an intermediate representation for modeling stream processing operators as finite state transducers over data words [3, 4, 1]. Data transducers support succinct constructions, making them compositional. We also describe the QRE-Past monitoring language, which can be used for monitoring stream processing applications.

The key takeaways of this section are related to performance: data transducers as an IR allow *formal guarantees* on performance. The key theorems are Theorem 8.3.1, which states streaming evaluation of a data transducer takes linear space and time (independent of the input stream size); and Theorem 8.3.1 which shows that a QRE-Past query can be compiled to a data transducer of quadratic size. Putting these two results together we get formal bounds on performance for QRE-Past queries.

Unlike the rest of the thesis, this section does not consider distribution. Providing performance upper bounds is difficult even in the sequential case, so it is a reasonable starting point. Because it is sequential, the model is also deterministic.

Section 8.3 introduces the model of data transducers with illustrative examples. In Section 8.5 we consider a number of semantic operations with corresponding succinct constructions on DTs, and we define and study the key property of restartability necessary for some of them. In Section 8.6, we define the query language QRE-Past, and show how constructions on DTs immediately yield modular compilation into a streaming evaluation algorithm. We also show how QRE-Past is useful in specifying a cardiac arrhythmia detection algorithm. Section 8.7 discusses the expressiveness and succinctness of DTs compared to cost register automata, to finite automata, and to general streaming computations. We conclude in Section 8.8.

8.1. Motivation

Applications ranging from network traffic engineering to runtime monitoring of autonomous control systems require computation over data streams in an efficient and incremental manner. Declarative programming is a particularly appealing approach to specify the desired logic in such applications as it can provide natural and high-level constructs for processing streaming data with guaranteed bounds on computational resources used by the compiled implementation. This has motivated the development of a number of declarative query languages. For example, in runtime verification, a monitor observes a sequence of events produced by a system, and issues an alert when a violation of a safety property is detected, where the safety property is described in a temporal logic with past-time operators such as *always-in-the-past* and *since* [188, 143]. In quantitative monitoring, a monitor associates a numerical value with an input stream of data values, where the desired computation is described using *quantitative regular expressions* (QREs) that combine regular patterns with numerical aggregation operations such as min, max, sum, and average [25, 187, 284]. In each such case, the declarative specification is automatically compiled into a monitor that adheres to the streaming model of computation [204]: memory and per-item processing time is polynomial in the size of the specification of the query and, roughly speaking, does not grow with the length of the input stream.

In existing query languages over streaming data, while a programmer can specify the desired computation in a modular fashion using constructs of the query language, the compiler generates monolithic code for a given query. What is lacking though is an intermediate representation for streaming computations that supports composition operations with succinct constructions so that high-level queries can be compiled modularly. The motivation for such a model is two-fold. From a practical viewpoint, it can facilitate the design of new query languages. For instance, suppose a user wants to specify a monitoring property using past-time temporal logic, where the atomic predicates involve comparing quantitative summaries defined using QREs. Such a specification contains combinators from two different languages (QREs and past-temporal logic), and we could try to design a compiler from scratch for streaming evaluation of the more expressive, integrated language. However, if we have a *modular* compilation algorithm for the combinators of the two component languages, we get a compiler for the integrated language for free. From a theoretical viewpoint, designing such a representation is a technical challenge since it needs to support both combining values from parallel threads of computation (i.e. parallel composition) and unambiguous

regular parsing. In particular, although QREs can be compiled into quantitative automata known as *cost register automata* [24], since this compilation has provably exponential lower bound, it is not employed by current QRE evaluation algorithms, and in fact, no existing formalism can support modular compilation of QREs.

8.2. Contributions

The main contribution of this paper is the model of *Data Transducers* (DT) as this desired modular intermediate representation for streaming computations. A data transducer processes a data stream—a sequence of tagged data values—and produces a numerical (or Boolean) value using a fixed set of data variables that are updated using a constant number of operations as it processes each tagged data value. A DT can be viewed as a quantitative generalization of (unambiguous) NFAs. Whereas an NFA configuration consists of a finite set of states, each of which is either inactive or active, a DT configuration consists of a finite set of data variables, each of which can be inactive (*undefined*), active with a value (*defined*), or in a special "conflict" mode (*conflicted*). A DT configuration thus consists of succinctly represented finite control integrated with data values. As a DT computes by consuming tagged data values, it updates its variables using a specified allowed set of operations. The values of defined variables can be combined using operations to form new values, but there is also the possibility of a "collision". This is analogous to how two tokens of active NFA states can be merged into one token during evaluation when they are placed on the same state. Since the merging of data values is not in general a meaningful operation, a collision of values results in a variable being set to conflict. Since multiple transitions can write to the same data variable while processing a single tagged data value, and the updated value of a variable can depend on the updated values of the others, the semantics is defined using fixed points. We show how this semantics can be implemented by an efficient streaming algorithm for evaluation that executes a linear (in the size of DT) number of data operations while processing each tagged data value.

The language of a DT, i.e. the set of stream histories for which its output is defined, is a regular language over the tags of the input stream. In fact, DTs capture a robust class of functions with an elegant logical characterization: MSO-definable string-to-DAG transformations with a special "no backward edges" requirement. This class, which we call *streamable regular transductions*, has been studied in [105, 85] [1], and the closure properties of this class, as opposed to some specific constructs

supported by query languages in the existing literature, guide the choice of operations over DTs for which we seek succinct constructions.

In particular, we show that DTs are closed under quantitative concatenation, quantitative iteration, union, and parallel composition operations, and that the corresponding constructions are succinct. We also consider the *prefix-sum* operation that combines the outputs on all prefixes using a specified aggregator; this also has a simple and succinct construction on DTs. Temporal operators such as "always in the past", "sometime in the past", and "since" can be implemented using prefix-sum. The design choices in the precise formal definition of the model turn out to be critical in these constructions. A key restriction on DTs, which we call *restartability*, that is required for constructions related to unambiguous parsing is identified. This restriction says that it is possible to "restart" the automaton during a computation by placing new data values at its initial states. Then, although we only need to store a single automaton configuration in memory, the output is the same as if multiple copies of the automaton were computing independently on multiple stream suffixes as long as only one of these copies ultimately contributes to the final output. This ability is necessary for efficient unambiguous parsing: several parsing possibilities are explored simultaneously, but the required space is constant.

To illustrate the benefits of modular compilation, we define a new query language, called QRE-Past, that combines the features of past-time temporal logic and QREs. We specify a cardiac arrhythmia detection algorithm [15, 286] in QRE-Past to illustrate how the combination of features leads to a natural high-level specification. The theory of DTs immediately leads to a streaming evaluation algorithm for QRE-Past, since every construct in QRE-Past maps to a corresponding construction on component DTs without causing blow-up. In fact, there is nothing sacred about QRE-Past: the designer of a high-level query language over streaming data for a specific domain can introduce new combinators, in addition to the ones in this paper, as long as there are corresponding succinct constructions on the low-level model of DTs.

Finally, while there are existing models with identical expressiveness, DTs are exponentially more succinct (for instance, compared to unambiguous cost register automata). To gain a better understanding of the expressiveness and succinctness of DTs, consider a (generic) streaming algorithm that maintains a fixed number of Boolean and data variables, and processes each tagged data value by updating these variables by executing a loop-free code. While such algorithms capture *all* streaming

computations, the class of all streaming computations is not suitable for modular specifications. For instance, consider the quantitative concatenation operation: given transductions f and g, and a binary data operation op, $h = split(f, g, op)$ splits the inputs stream w uniquely into two parts $w = w_1 w_2$ and returns $h(w) = op(f(w_1), g(w_2))$. While DTs are closed under this operation, the class of all streaming algorithms is not. We can enforce regularity of a generic streaming algorithm by requiring, for instance, that the updates to the Boolean variables are not influenced by the values of the data variables. We show that streaming algorithms with these restrictions can be translated to DTs without any blow-up, thus establishing that DTs are the most succinct (up to a constant factor) representation of streamable regular transductions. The structure of a DT—as variables ranging over undefined/defined/conflict values and update code as a set of transitions of a particular form, as opposed to traditional loop-free update code—not only enforces regularity, but is also what allows us to define succinct constructions on the representation.

8.3. Data Transducers

8.3.1. Preliminaries

To model data streams we use *data words*. Let $\mathbb{D}$ be a (possibly infinite) set of *data values*, such as the set of integers or real numbers, and let Σ be a finite set of *tags*. Then a *data word* is a sequence of tagged data values $\mathbf{w} \in (\Sigma \times \mathbb{D})^*$. We write $\mathbf{w} \downarrow \Sigma$ to denote the projection of $\mathbf{w}$ to a string in Σ^*. We use bold $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ to denote data words. We reserve non-bold u, v, w for plain strings of tags in Σ^*. We write d, d_i for elements of $\mathbb{D}$. We use σ to denote an arbitrary tag in Σ, and in the examples we write particular tags in typewriter font, e.g. $\mathtt{a}, \mathtt{b}$.

A *signature* is a tuple $(\mathbb{D}, \mathsf{Op})$, where $\mathbb{D}$ is a set of data values and Op is a set of *allowed operations*. Each operation has an *arity* $k \geq 0$ and is a function from $\mathbb{D}^k$ to $\mathbb{D}$. We use Op_k to denote the k-ary operations. For instance, if $\mathbb{D}$ is all 64-bit integers, we might support 64-bit arithmetic, as well as integer division and equality tests. Alternatively we might have $\mathbb{D} = \mathbb{N}$ with the operations $+$ (arity 2), min (arity 2), and 0 (arity 0). In general, we may have arbitrary user-defined operations on $\mathbb{D}$. Given a signature $(\mathbb{D}, \mathsf{Op})$, and a collection of variables Z, the set of *terms* $\mathsf{Tm}[Z]$ consists of all syntactically correct expressions with free variables in Z, using operations Op. So $\min(x, 0) + \min(y, 0)$ and $x + x$ are terms over the signature $(\mathbb{N}, \{+, \min, 0\})$ with $Z = \{x, y\}$.

We define two special values in addition to the values in $\mathbb{D}$: $\bot$ denotes *undefined* and $\top$ denotes *conflict*. We let $\overline{\mathbb{D}} := \mathbb{D} \cup \{\bot, \top\}$ be the set of *extended data values*, and refer to elements of $\mathbb{D}$ as *defined*. We lift Op to operations on $\overline{\mathbb{D}}$ by thinking of $\bot$ as the empty multiset, elements of $\mathbb{D}$ as singleton multisets, and $\top$ as any multiset of two or more data values. The specific behavior of $op \in \mathsf{Op}$ on values in $\overline{\mathbb{D}}$ is illustrated in the table below for the case $op \in \mathsf{Op}_2$. We also define a *union* operation $\sqcup : \overline{\mathbb{D}} \times \overline{\mathbb{D}} \to \overline{\mathbb{D}}$: if either of its arguments is undefined it returns the other one, and in all other cases it returns conflict. This represents multiset union. Note that $d_1 \sqcup d_2 = \top$ even if $d_1 = d_2$. This is essential: it guarantees that for all operations on extended data values, whether the result is undefined, defined, or conflict can be determined from knowing only whether the inputs are undefined, defined, or conflict. For instance, we rely on this guarantee for the theorems in Section 8.3.5 and for the translation from QRE-Past in Section 8.6.2. It's not needed for most of the constructions in Section 8.5.

$\sqcup$	$\bot$	d_2	$\top$
$\bot$	$\bot$	d_2	$\top$
d_1	d_1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$

op	$\bot$	d_2	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
d_1	$\bot$	$op(d_1, d_2)$	$\top$
$\top$	$\bot$	$\top$	$\top$

$\overline{\mathbb{D}}$ is a *complete lattice*, partially ordered under the relation $\leq$ which is defined by $\bot \leq d \leq \top$ for all $d \in \mathbb{D}$, and distinct elements $d, d' \in \mathbb{D}$ are incomparable. For a finite set X, we write the set of functions $X \to \overline{\mathbb{D}}$ as $\overline{\mathbb{D}}^X$; its elements are untagged *data vectors*, denoted $\mathbf{x}$, $\mathbf{y}$. The partial order extends coordinate-wise to an ordering $\mathbf{x} \leq \mathbf{y}$ on data vectors $\mathbf{x}, \mathbf{y} \in \overline{\mathbb{D}}^X$. All operations in Op are *monotone increasing* w.r.t. this partial order. Union ($\sqcup$) is commutative and associative, with identity $\bot$ and absorbing element $\top$, and *all* k-ary operations distribute over it.

8.3.2. Syntax

Let $(\mathbb{D}, \mathsf{Op})$ be a fixed signature. A *data transducer (DT)* is a 5-tuple

$$\mathcal{A} = (Q, \Sigma, \Delta, I, F),$$

where:

- Q is a finite set of *state variables* (*states* for short) and Σ is a finite set of *tags*. We write Q' for a copy of the variables in Q: for $q \in Q$, $q' \in Q'$ denotes the copy. When the states of the DT are updated, q' will be the new, updated value of q.

- Δ is a finite set of *transitions*, where each transition is a tuple (σ, X, q', t).

  - $\sigma \in \Sigma \cup \{\mathtt{i}\}$, where $\mathtt{i} \notin \Sigma$, and if $\sigma = \mathtt{i}$ this is a special *initial transition*.

  - $X \subseteq Q \cup Q'$ is a set of *source variables* and $q' \in Q'$ is the *target variable*.

  - $t \in \mathsf{Tm}[X \cup \{\mathsf{cur}\}]$ gives a new value of the target variable given values of the source variables and given the value of "cur", which represents the current data value in the input data word. Assume that $\mathsf{cur} \notin X$. We allow X to include some variables not used in t. For initial transitions, we additionally require that $X \subseteq Q'$ and that cur does not appear in t.

- $I \subseteq Q$ is a set of *initial states* and $F \subseteq Q$ is a set of *final states*.

The *number of states* of $\mathcal{A}$ is $|Q|$. The *size* of $\mathcal{A}$ is the the number of states plus the total length of all transitions (σ, X, q', t), which includes the length of description of all the terms t.

8.3.3. Semantics

The input to a DT has two components. First, an *initial vector* $\mathbf{x} \in \overline{\mathbb{D}}^I$, which assigns an extended data value to each initial state. Second, an *input data word* $\mathbf{w} \in (\Sigma \times D)^*$, which is a sequence of tagged data values to be processed by the transducer. On input $(\mathbf{x}, \mathbf{w})$, the DT's final *output vector* is an extended data value at each of its final states. Thus, the semantics of $\mathcal{A}$ will be

$$[\![\mathcal{A}]\!] : \overline{\mathbb{D}}^I \times (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}^F.$$

A *configuration* is a vector $\mathbf{c} \in \overline{\mathbb{D}}^Q$. For every $\sigma \in \Sigma$, the set of transitions (σ, X, q', t) collectively define a function $\Delta_\sigma : \overline{\mathbb{D}}^Q \times \mathbb{D} \to \overline{\mathbb{D}}^Q$: given the current configuration and the current data value from the input data word, Δ_σ produces the next configuration. We define $\Delta_\sigma(\mathbf{c}, d)(q) := \mathbf{c}'(q')$, where $\mathbf{c}' \in \overline{\mathbb{D}}^{Q \cup Q' \cup \{\mathsf{cur}\}}$ is the *least vector* satisfying $\mathbf{c}'(\mathsf{cur}) = d$; for all $q \in Q$, $\mathbf{c}'(q) = \mathbf{c}(q)$; and

$$\text{for all } q' \in Q', \quad \mathbf{c}'(q') = \bigsqcup_{(\sigma, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X), \tag{8.1}$$

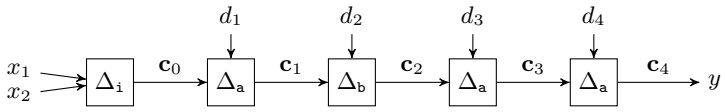


Figure 8.1: Example evaluation of a data transducer $\mathcal{A}$ with two initial states and one final state on initial vector $(\mathbf{x}_1, \mathbf{x}_2)$ and an input data word $\mathbf{w}$ consisting of four characters (tagged data values): $(\mathtt{a}, d_1)$, $(\mathtt{b}, d_2)$, $(\mathtt{a}, d_3)$, $(\mathtt{a}, d_4)$, to produce output y. Here $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$, and $\mathbf{c}_4$ are configurations; $d_i \in \mathbb{D}$; and $x_1, x_2, y \in \overline{\mathbb{D}}$. Each Δ_σ is a set of transitions, collectively describing the next configuration in terms of the previous one.

where we define $[\![t]\!](\mathbf{c}'|_X)$ to be $\bot$ if there exists $x \in X$ such that $\mathbf{c}'(x) = \bot$; otherwise, $\top$ if there exists $x \in X$ such that $\mathbf{c}'(x) = \top$; otherwise, if all variables in X are defined, then $[\![t]\!](\mathbf{c}'|_X)$ is the value of the expression t with variables assigned the values in $\mathbf{c}'$. So, $[\![t]\!](\mathbf{c}'|_X)$ produces $\bot$ or $\top$ if some variable in X is $\bot$ or $\top$. The above union is over all transitions with label σ and target variable q'. Since $\overline{\mathbb{D}}$ is a complete lattice, this least fixed point exists by the Knaster-Tarski theorem.

The case of initial transitions ($\Delta_{\mathtt{i}}$) is slightly different. The purpose of initial transitions is to compute an initial configuration $\mathbf{c}_0 \in \overline{\mathbb{D}}^Q$, given the initial vector $\mathbf{x} \in \overline{\mathbb{D}}^I$. There is no previous configuration, and no current data value, which is why we required $X \subseteq Q'$ for initial transitions and cur was not allowed. We define the function $\Delta_{\mathtt{i}} : \overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^Q$ with the same fixed point computation from Equation (8.1), except that the initial states are additionally assigned values given by the vector $\mathbf{x}$. Define that $\mathbf{x}(q) = \bot$ if $q \notin I$. Then define $\Delta_{\mathtt{i}}(\mathbf{x}) = \mathbf{c}'$, where $\mathbf{c}'$ is the *least vector* satisfying, for all $q \in Q$, $\mathbf{c}'(q') = \mathbf{x}(q) \sqcup \bigsqcup_{(\mathtt{i}, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X)$.

Now $\mathcal{A}$ is evaluated on input $(\mathbf{x}, \mathbf{w}) \in \overline{\mathbb{D}}^I \times (\Sigma \times \mathbb{D})^*$ by starting from the initial configuration and applying the update functions in sequence as illustrated in Figure 8.1. Finally, the output $\mathbf{y} \in \overline{\mathbb{D}}^F$ is given by $\mathbf{y} = \mathbf{c}|_F$, the projection of $\mathbf{c}$ to the final states.

8.3.4. Streaming Evaluation Algorithm

Evaluation complexity of a data transducer depends on the underlying operations, so we give a conditional result where the complexity is stated in terms of the number of data registers and number of operations on those data registers.

Theorem 8.3.1. Evaluation of a data transducer $\mathcal{A}$, with number of states n and size m on input $(\mathbf{x}, \mathbf{w})$, requires $O(n)$ data registers to store the state, and $O(m)$ operations and additional data

$\mathbf{c} \leftarrow \Delta_{\mathtt{i}}(\mathbf{x})$; produce output $\mathbf{y} = \mathbf{c}|_F$
for each character (σ, d) in $\mathbf{w}$ **do**
  for each state $q \in Q$ **do** $val(q) \leftarrow \mathbf{c}(q)$; $val(q') \leftarrow \bot$
  for each transition $\tau \in \Delta_\sigma$ **do** $val(\tau) \leftarrow \bot$; $num\_undef(\tau) \leftarrow |X|$
  $worklist \leftarrow Q' \cup \Delta_\sigma$
  while *worklist* is nonempty, **get** *item* from *worklist* and **do**
    if *item* is a transition $\tau = (\sigma, X, q', t) \in \Delta_\sigma$: **then**
      $val(\tau) \leftarrow [\![t]\!](val|_X)$
      if $val(q') \neq \top$ **then** add q' to *worklist*
    else if *item* is a state $q' \in Q'$ **then**
      if $val(q') = \bot$ **then**
        for each $\tau \in \Delta_\sigma$ with source variable q' **do** $num\_undef(\tau) \leftarrow num\_undef(\tau) - 1$
      $val(q') \leftarrow \bigsqcup_{\tau=(\sigma,X,q',t)} val(\tau)$
      for each $\tau \in \Delta_\sigma$ with target variable q' **do**
        if $val(\tau) \in \mathbb{D}$ or ($val(\tau) = \bot$ and $num\_undef(\tau) = 0$) **then** add τ to *worklist*
  for each $q \in Q$ **do** $\mathbf{c}(q) \leftarrow val(q')$
  produce output $\mathbf{y} = \mathbf{c}|_F$

Figure 8.2: Data transducer evaluation algorithm (Theorem 8.3.1). On input $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ over $(\mathbb{D}, \mathsf{Op})$, an initial vector $\mathbf{x} \in \overline{\mathbb{D}}^I$, and a data stream $\mathbf{w} \in (\Sigma \times \mathbb{D})^*$, produces the output vector $\mathbf{y} \in \overline{\mathbb{D}}^F$ on each prefix of $\mathbf{w}$.

registers to process each element in $\Sigma \times \mathbb{D}$, independent of $\mathbf{w}$. The evaluation algorithm is given in Figure 8.2.

8.3.5. Regularity

Data transducers define *regular transductions* on data words (see Section 8.7.1). Here, we show regularity in a simpler sense: whether an output value is defined (or undefined, or conflict) depends only on whether the input values are undefined, defined, or conflict, together with some regular property of the string of tags. For data vectors $\mathbf{x}_1, \mathbf{x}_2 \in \overline{\mathbb{D}}^X$, we say that $\mathbf{x}_1$ and $\mathbf{x}_2$ are *equivalent*, and write $\mathbf{x}_1 \equiv \mathbf{x}_2$, if for all $x \in X$, $\mathbf{x}_1(x)$ and $\mathbf{x}_2(x)$ are both undefined, both defined, or both conflict.

Theorem 8.3.2. Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over $(\mathbb{D}, \mathsf{Op})$. Then: (i) For all initial vectors $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{D}^I$, and for all input words $\mathbf{w}_1, \mathbf{w}_2$, if $\mathbf{x}_1 \equiv \mathbf{x}_2$ and $\mathbf{w}_1 \downarrow \Sigma = \mathbf{w}_2 \downarrow \Sigma$, then $[\![\mathcal{A}]\!](\mathbf{x}_1, \mathbf{w}_1) \equiv [\![\mathcal{A}]\!](\mathbf{x}_2, \mathbf{w}_2)$. (ii) For every equivalence class of initial vectors $\mathbf{x}$ and equivalence class of output vectors $\mathbf{y}$, the set of strings $\mathbf{w} \downarrow \Sigma$ such that $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \equiv \mathbf{y}$ is regular.

Proof. In evaluating a DT we may collapse all values in $\mathbb{D}$ to a single value $\star$, so each state takes values in $\{\bot, \star, \top\}$. This gives a projection from $\mathcal{A}$ to a DT $\mathcal{P}$ over the *unit signature* $(\mathbb{U}, \mathsf{UOp})$, where $\mathbb{U} = \{\star\}$ is a set with just one element, and UOp consists of, for each k, the unique map $o_k : \mathbb{U}^k \to \mathbb{U}$. The projection homomorphically preserves the semantics. Then, (i) follows because the

computation of $\mathcal{P}$ is exactly the same on $\mathbf{x}_1, \mathbf{w}_1$ and $\mathbf{x}_2, \mathbf{w}_2$, and (ii) follows because $\mathcal{P}$ has finitely many possible configurations. □

We can thus define the *language* of $\mathcal{A}$ to be $L(\mathcal{A}) = \{\mathbf{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \in \mathbb{D}^F \text{ for some } \mathbf{x} \in \mathbb{D}^I\}$, so $L(\mathcal{A}) \subseteq \Sigma^*$. This is the set of tag strings $\mathbf{w} \downarrow \Sigma$ such that, if the initial vector of values is all defined, after reading in $\mathbf{w}$ all final states are defined. We similarly define the set of strings on which a DT is *defined or conflict*, on input of the same form: the *extended language* $\overline{L}(\mathcal{A})$ is $\{\mathbf{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \in (\mathbb{D} \cup \{\top\})^F \text{ for some } \mathbf{x} \in (\mathbb{D} \cup \{\top\})^I\}$. An immediate corollary of Theorem 8.3.2 is that (i) $L(\mathcal{A})$ is regular, (ii) $\overline{L}(\mathcal{A})$ is regular, and (iii) $L(\mathcal{A}) \subseteq \overline{L}(\mathcal{A})$. Finally, say that DTs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equivalent* if for all $\mathbf{x}_1 \equiv \mathbf{x}_2$ and for all $\mathbf{w}$, $[\![\mathcal{A}_1]\!](\mathbf{x}_1, \mathbf{w}) \equiv [\![\mathcal{A}_2]\!](\mathbf{x}_2, \mathbf{w})$.

Theorem 8.3.3. On input DTs $\mathcal{A}_1$, $\mathcal{A}_2$, deciding if $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent is PSPACE-complete.

Proof. We first decide if the two are *not* equivalent in NPSPACE. It suffices to project $\mathcal{A}_1$ and $\mathcal{A}_2$ to DTs over the unit signature, $\mathcal{P}_1$ and $\mathcal{P}_2$, as in the previous proof, and decide if $\mathcal{P}_1 \not\equiv \mathcal{P}_2$. Let n be the number of states between $\mathcal{P}_1$ and $\mathcal{P}_2$, and let m be their combined size. The number of configurations for $\mathcal{P}_1$ and $\mathcal{P}_2$ together is 3^n. Therefore, if there is a counterexample, it is some string over Σ of length at most 3^n. Guessing the counterexample one character at a time requires linear in n space to record the count and $O(m)$ space to update $\mathcal{P}_1$ and $\mathcal{P}_2$ (by Theorem 8.3.1).

To show it is PSPACE-hard, it suffices to exhibit a translation from NFAs to DTs which reduces language equality of NFAs to equivalence of DTs. Specifically, we create $\mathcal{A}$ with one final state which is undefined on strings for which the NFA is undefined, and $\top$ on strings for which the NFA is defined. The translation works by directly copying the states and transitions of the NFA, except we add *two* additional transitions from accepting states of the NFA to the new final state of $\mathcal{A}$. □

8.4. Examples

We do not envision that DTs would be directly programmed by users, due to the conceptual difficulty of tracking undefined, defined, and conflicted values. Rather, DTs would be a low-level, back-end model for streaming and monitoring. The purpose of this section is mainly to illustrate, informally and through examples, the basic features and execution semantics of the model.

We present only *acyclic* DTs in this section, and we take $I = \varnothing$: all initialization is done with initial transitions $\Delta_{\mathtt{i}}$. Additionally, we use the abbreviation $q' := t$ to denote a transition (σ, X, q', t), where X is exactly the set of variables present in the term t (in contexts where σ is clear). In general, X may include other variables unused in t, and the semantics of the transition does depend on the unused variables as well (see §8.3.3, "Why do variables in X unused in t affect the semantics?").

Pattern matching. DTs are based on the idea of merging *data registers* and *finite control* into the single set of "state variables" Q. Suppose we wish to monitor a stream of `a`-events, `b`-events, and `#`-events, where each `a`- or `b`-event is the price at which an item was bought, and `#` indicates the end of a day. We thus have $\mathbb{D} = \mathbb{Q}$ and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{\#}\}$. For the operations Op, we allow $+, -, \cdot, \max, \min$, division / (this must return a default value on division by 0), and integer constants. Suppose we want to output the average price of a *sliding window* containing the last three `a` prices, which resets at the end of the day. This is essentially a *pattern match* over the input tags to locate the last three, which are then averaged. $\mathcal{A}_1$ in Figure 8.3 is based on this idea. The transitions listed under $\mathsf{transitions}(\sigma)$ are those labeled with σ; we use $\|$ to emphasize that the transitions are not ordered.

The machine $\mathcal{A}_1$ uses state variables `sum1`, `sum2`, and `sum3` to keep track of the sum of the last 1, 2, and 3 `a` prices (in the current day). Each variable matches a certain pattern of tags in the input stream, namely, strings with at least 1, 2, and 3 `a`'s so far. In addition to pattern-matching, the variables are updated to keep track of the sum. For example, the transition $\mathtt{sum2}' := \mathtt{sum1} + \mathsf{cur}$ indicates that *if* `sum1` was defined before then `sum2` should now be defined and equal to the sum plus the current data value. The transition $\mathtt{avg}' := \mathtt{sum3}' \,/\, 3$ indicates that *if* `sum3` is *now* defined (note the $\mathtt{sum3}'$), then `avg` should be set to the average of the last three prices.

Multiple transitions with a single target. The machine $\mathcal{A}_1$ has a simplifying syntactic property that for every $\sigma \in \Sigma$ and for every state q', there is only one transition $q' := t$. In other words, there is only one *rule* stating how to assign q' a value. In general, there may be multiple rules, and the resulting value of q' will be the union ($\sqcup$) over all transitions. For instance, suppose we have the same input stream over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{\#}\}$, and we want to output the average price of an `a`-item at the end of each day. However, if there are no `a`-items on a given day, we instead want to output the average from the previous day. A machine implementation of this is provided by $\mathcal{A}_2$ in Figure 8.4.

In $\mathcal{A}_2$, `sum` and `count` store the sum of `a`-items and number of `a`-items on each day, respectively, and are defined only if there has been at least one `a`. On the other hand, `prev_avg` stores the previous

$Q = \{\texttt{sum1}, \texttt{sum2}, \texttt{sum3}, \texttt{avg}\},\ I = \varnothing,\ F = \{\texttt{avg}\}$

$$
\begin{aligned}
\texttt{transitions(i)} &= \varnothing \\
\texttt{transitions(a)} &= \ \| \ \texttt{sum1}' := \texttt{cur} \\
&\quad \| \ \texttt{sum2}' := \texttt{sum1} + \texttt{cur} \\
&\quad \| \ \texttt{sum3}' := \texttt{sum2} + \texttt{cur} \\
&\quad \| \ \texttt{avg}' := \texttt{sum3}' \,/\, 3 \\
\texttt{transitions(b)} &= \ \| \ \texttt{sum1}' := \texttt{sum1} \\
&\quad \| \ \texttt{sum2}' := \texttt{sum2} \\
&\quad \| \ \texttt{sum3}' := \texttt{sum3} \\
\texttt{transitions(\#)} &= \varnothing
\end{aligned}
$$

Example evaluation on input

$$\mathbf{w} = (\texttt{a}, 6)(\texttt{a}, 5)(\texttt{a}, 7)(\texttt{b}, 2)(\texttt{a}, 8)(\#, 0)(\texttt{b}, 2)(\texttt{a}, 7).$$

w (input)	sum1	sum2	sum3	avg (output)
	$\bot$	$\bot$	$\bot$	$\bot$
$(\texttt{a}, 6)$	6	$\bot$	$\bot$	$\bot$
$(\texttt{a}, 5)$	5	11	$\bot$	$\bot$
$(\texttt{a}, 7)$	7	12	18	6.000
$(\texttt{b}, 2)$	7	12	18	$\bot$
$(\texttt{a}, 8)$	8	15	20	6.667
$(\#, 0)$	$\bot$	$\bot$	$\bot$	$\bot$
$(\texttt{b}, 2)$	$\bot$	$\bot$	$\bot$	$\bot$
$(\texttt{a}, 7)$	7	$\bot$	$\bot$	$\bot$

Figure 8.3: Data transducer $\mathcal{A}_1$ monitoring a stream of purchase events for two types of items, tagged `a` and `b`, and `#` to represent the end of each day. Throughout the day we output the average price in a sliding window of the last three `a`-items. The language of strings on which $\mathcal{A}_1$ produces output is $(\texttt{a} \cup \texttt{b} \cup \texttt{\#})^*\texttt{ab}^*\texttt{ab}^*\texttt{a}$.

average, but it is defined only if there has *not* been any `a` yet. (We also initialize this to 0 arbitrarily on the very first day.) The state `avg` stores the output, and is only defined after a `#` event. The logic of this computation involves two places where we need to have multiple transitions targeting a state. First, on receiving an `a`, we set `sum` to be equal to the previous sum plus the current value, but we also set it to be equal to $0 \cdot \texttt{prev\_avg} + \texttt{cur}$. This works because exactly one of these two values will be defined, and the other will be $\bot$: either we have seen an `a` already, in which case we can update the sum, or we haven't seen one yet, in which case `prev_avg` is still defined. Second, the overall output `avg` has two possible values, either `sum`/`count` or `prev_avg`, and again, exactly one of these two values will be defined, and the other will be $\bot$. Thus, we have designed $\mathcal{A}_2$ so that each union operation ($\sqcup$) never produces a conflict ($\top$).

Combining output from parallel threads of computation. Our final example attempts to illustrate the feature which gives DTs their succinctness (see §8.7): the ability to update multiple computations independently and then combine their results. Suppose we want to compute, at the end of each day, the difference between the maximum price of `a` and the maximum price of `b`, if there was at least one `a` and at least one `b`. The DT $\mathcal{A}_3$ in Figure 8.5 implements this computation. The state `a_init` of $\mathcal{A}_3$ stores 0 and is only defined if we haven't seen an `a` yet; similarly for `b_init`.

$Q = \{\texttt{sum}, \texttt{count}, \texttt{avg}, \texttt{prev\_avg}\},\ I = \varnothing,\ F = \{\texttt{avg}\}$

$\texttt{transitions(i)} = \parallel \texttt{prev\_avg}' := 0$

$\texttt{transitions(a)} = \parallel \texttt{sum}' := \texttt{prev\_avg} \cdot 0 + \texttt{cur}$

$\parallel \texttt{sum}' := \texttt{sum} + \texttt{cur}$

$\parallel \texttt{count}' := \texttt{prev\_avg} \cdot 0 + 1$

$\parallel \texttt{count}' := \texttt{count} + 1$

$\texttt{transitions(b)} = \parallel \texttt{sum}' := \texttt{sum}$

$\parallel \texttt{count}' := \texttt{count}$

$\parallel \texttt{prev\_avg}' := \texttt{prev\_avg}$

$\texttt{transitions(\#)} = \parallel \texttt{avg}' := \texttt{sum} \,/\, \texttt{count}$

$\parallel \texttt{avg}' := \texttt{prev\_avg}$

$\parallel \texttt{prev\_avg}' := \texttt{avg}'$

Example evaluation on input

$(\texttt{b}, 2)(\texttt{a}, 6)(\texttt{b}, 2)(\texttt{a}, 8)(\texttt{a}, 7)(\texttt{\#}, 0)(\texttt{b}, 2)(\texttt{\#}, 0)(\texttt{a}, 7)(\texttt{a}, 6).$

w (input)	**sum**	**count**	**avg** (output)	**prev_avg**
	$\bot$	$\bot$	$\bot$	0
$(\texttt{b}, 2)$	$\bot$	$\bot$	$\bot$	0
$(\texttt{a}, 6)$	6	1	$\bot$	$\bot$
$(\texttt{b}, 2)$	6	1	$\bot$	$\bot$
$(\texttt{a}, 8)$	14	2	$\bot$	$\bot$
$(\texttt{a}, 7)$	21	3	$\bot$	$\bot$
$(\texttt{\#}, 0)$	$\bot$	$\bot$	7.0	7.0
$(\texttt{b}, 2)$	$\bot$	$\bot$	$\bot$	7.0
$(\texttt{\#}, 0)$	$\bot$	$\bot$	7.0	7.0
$(\texttt{a}, 7)$	7	1	$\bot$	$\bot$
$(\texttt{a}, 6)$	13	2	$\bot$	$\bot$

Figure 8.4: Data transducer $\mathcal{A}_2$ monitoring the stream to produce, at the end of each day, either the average price of an a-item (if there was at least one a) or the previous average (if there was no a). When there are multiple transitions $q' := t_1$ and $q' := t_2$, the semantics is such that we assign $q' := t_1 \sqcup t_2$.

8.5. Constructions

Our primary interest in the DT model is to support a variety of succinct *composition operations* which are not simultaneously supported by any existing model. In particular, such composition operations can enable a quantitative monitoring language like QRE-Past in Section 8.6: language constructs can be implemented by the compiler as constructions on DTs, rather like how (traditional) regular expressions are compiled to nondeterministic finite automata.

For example, suppose we have DTs implementing two functions $f, g : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, and we would like to implement the function $f + g$, which applies f and g to the input stream and adds the results. To do so, we copy the states of the transducers for f and g, and we initialize and update the states in parallel (they do not interfere). Then, we provide a new final state, and a single new transition which says that the new final state should be assigned the value of the final state of f plus the value of the final state of g. This works for every operation, and not just $+$: the combination of k computations by applying a k-ary operation $op \in \mathsf{Op}_k$ can be implemented by a corresponding k-ary construct on the k underlying DTs. Moreover, the size of the DT will only be the sum of the sizes of the k DTs, plus a constant. In contrast, even this simple operation $f + g$ is not succinctly implementable using the most natural existing alternative to DTs, Cost Register Automata (see Section 8.7).

$Q = \{\texttt{a\_init}, \texttt{a\_max}, \texttt{b\_init}, \texttt{b\_max}, \texttt{ab\_diff}\}$
$I = \varnothing,\ F = \{\texttt{ab\_diff}\}$

$\mathsf{transitions}(\texttt{i}) = \| \ \texttt{a\_init}' := 0$
$\| \ \texttt{b\_init}' := 0$

$\mathsf{transitions}(\texttt{a}) = \| \ \texttt{a\_max}' := \texttt{a\_init} + \mathsf{cur}$
$\| \ \texttt{a\_max}' := \max(\texttt{a\_max}, \mathsf{cur})$
$\| \ \texttt{b\_max}' := \texttt{b\_max}$
$\| \ \texttt{b\_init}' := \texttt{b\_init}$

$\mathsf{transitions}(\texttt{b}) = \| \ \texttt{b\_max}' := \texttt{b\_init} + \mathsf{cur}$
$\| \ \texttt{b\_max}' := \max(\texttt{b\_max}, \mathsf{cur})$
$\| \ \texttt{a\_max}' := \texttt{a\_max}$
$\| \ \texttt{a\_init}' := \texttt{a\_init}$

$\mathsf{transitions}(\texttt{\#}) = \| \ \texttt{ab\_diff}' := \texttt{a\_max} - \texttt{b\_max}$
$\| \ \texttt{a\_init}' := 0$
$\| \ \texttt{b\_init}' := 0$

Example evaluation on input

$(\texttt{b}, 2)(\texttt{a}, 6)(\texttt{b}, 3)(\texttt{b}, 1)(\texttt{a}, 8)(\texttt{\#}, 0)(\texttt{b}, 2)(\texttt{\#}, 0)(\texttt{a}, 7)(\texttt{b}, 1).$

w (input)	a_init	a_max	b_init	b_max	ab_diff (output)
	0	$\perp$	0	$\perp$	$\perp$
$(\texttt{b}, 2)$	0	$\perp$	$\perp$	2	$\perp$
$(\texttt{a}, 6)$	$\perp$	6	$\perp$	2	$\perp$
$(\texttt{b}, 3)$	$\perp$	6	$\perp$	3	$\perp$
$(\texttt{b}, 1)$	$\perp$	6	$\perp$	3	$\perp$
$(\texttt{a}, 8)$	$\perp$	8	$\perp$	3	$\perp$
$(\texttt{\#}, 0)$	0	$\perp$	0	$\perp$	5
$(\texttt{b}, 2)$	0	$\perp$	$\perp$	2	$\perp$
$(\texttt{\#}, 0)$	0	$\perp$	0	$\perp$	$\perp$
$(\texttt{a}, 7)$	$\perp$	7	0	$\perp$	$\perp$
$(\texttt{b}, 1)$	$\perp$	7	$\perp$	1	$\perp$

Figure 8.5: Data transducer $\mathcal{A}_3$ monitoring the stream to produce, at the end of each day, the difference between the maximum price of an a-item and the maximum price of a b-item.

This construction for $f + g$ requires no assumptions about the DTs implementing f and g. However, not all operations are this straightforward. Consider the following quantitative generalization of concatenation. Given $f : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, $g : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, and $op \in \mathsf{Op}_2$, we wish to implement $\texttt{split}(f, g, op)$: on input $\mathbf{w}$, split the input stream into two parts, $\mathbf{w} = \mathbf{u} \cdot \mathbf{v}$, such that $f(\mathbf{u}) \neq \perp$ and $g(\mathbf{v}) \neq \perp$ (respectively, f matches $\mathbf{u}$ and g matches $\mathbf{v}$), and return $op(f(\mathbf{u}), g(\mathbf{v}))$. Assume that the decomposition of $\mathbf{w}$ into $\mathbf{u}$ and $\mathbf{v}$ such that $f(\mathbf{u}) \neq \perp$ and $g(\mathbf{v}) \neq \perp$ is unique. In order to naively implement this operation, on an input string $\mathbf{w}$, we must not only keep track of the current configuration of f on $\mathbf{w}$, but for *every* split $\mathbf{w} = \mathbf{uv}$ where f matches $\mathbf{u}$, we must keep track of the current configuration of g on $\mathbf{v}$. If there are many possible prefixes $\mathbf{u}$ of $\mathbf{w}$ such that $f(\mathbf{u}) \neq \perp$, we may have to keep arbitrarily many configurations of g. This naive approach is therefore impossible using only the finite space that a DT allows, if we treat f and g only as black boxes.

What we need to avoid this is an additional structural condition on g. Rather than keeping multiple copies of g, we would like to keep only a single configuration in memory: whenever the current prefix matches f, *restart* g with new data values on its initial states (keeping any current data values as well). To motivate this idea, consider the analogous concatenation construction for two NFAs: every time the first NFA accepts, we are able to "restart" the second NFA by adding a token to its start state (we don't need an entirely new NFA every time). This property for DTs is called *restartability*.

Restartable DTs are an equally expressive subclass consisting of those DTs for which restarting computation on the same transducer does not cause interference in the output.

The set of strings that a DT "matches" is captured by its *extended language*, defined in Section 8.3.5. Correspondingly, we assume that whenever a DT is restarted, the new initial vector is either all $\bot$, or all *not* $\bot$ (in $\mathbb{D} \cup \{\top\}$). If the *output* of a DT also satisfies this property (on every input it is either all $\bot$, or all *not* $\bot$), then we say that the DT is *output-synchronized*. This property is required in the concatenation and iteration constructions, but it is not as crucial to the discussion as restartability.

We begin in Section 8.5.1 by giving general constructions that do not rely on restartability. We highlight the implemented semantics, the extended language, and the size of the constructed DT in terms of its constituent DTs. Then in Section 8.5.2, we define restartability and use it to give succinct constructions for unambiguous parsing operations, namely *concatenation* and *iteration*. Moreover, we show that (under certain conditions) our operations *preserve* restartability, thus enabling modular composition using the restartable DTs. We also show that checking restartability is hard (PSPACE-complete), and we mention converting a non-restartable DT to a restartable one, but with exponential blowup.

8.5.1. General Constructions

Notation It is convenient to introduce shorthand (ε, X, q', t) for the union of $|\Sigma| + 1$ transitions: (σ, X, q', t) for every $\sigma \in \Sigma \cup \{\mathtt{i}\}$. Because this includes an initial transition, this requires that $X \subseteq Q'$ and that cur does not appear in t. We call such a collection of transitions an *epsilon transition* because, like epsilon transitions from classical automata, the transition may produce a value at its target state on the empty data word and on every input character.

For readability, we abbreviate the type of a DT $\mathcal{A} : \overline{\mathbb{D}}^I \times (\Sigma \times D)^* \to \overline{\mathbb{D}}^F$ as $\mathcal{A} : I \twoheadrightarrow F$. This can be thought of as a function from input variables I of type $\overline{\mathbb{D}}$ to output variables F of type $\overline{\mathbb{D}}$, which also consumes some data word in $(\Sigma \times D)^*$ as a side effect. For sets of variables (or states) X_1, X_2, when we write $X_1 \cup X_2$ we assume that the union is disjoint, unless otherwise stated.

We also define a *data function* to be a plain function $\overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^F$ which is given by a collection of one or more terms $t : \mathsf{Tm}[I]$ for each $f \in F$ (the output value of f is then the union of the values of all

terms). If $G \subseteq F \times \mathsf{Tm}[I]$, then we write $G : I \Rightarrow F$ to abbreviate the semantics $[\![G]\!] : \overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^F$. The *size* of G is the total length of description of all of the terms t it contains.

Parallel composition. Suppose we are given two DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, and assume that the sets of initial states are the same up to some implicit bijections $\pi_1 : I \to I_1$, $\pi_2 : I \to I_2$, for a set I with $|I| = |I_1| = |I_2|$. (It is always possible to benignly extend both DTs with extra initial states so that they match, so this assumption is not restrictive.) We wish to define a DT which feeds the input $(\mathbf{x}, \mathbf{w})$ into both DTs in parallel. To do so, we define $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ to be the tuple $(Q, \Sigma, \Delta, I, F)$, where $Q = Q_1 \cup Q_2 \cup I$, $F = F_1 \cup F_2$, and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, i', \pi_1(i)', i') : i \in I\big\} \cup \big\{(\varepsilon, i', \pi_2(i)', i') : i \in I\big\}.$$

Here, the transitions we added (those in Δ but not in Δ_1 or Δ_2) *copy* values from I into both I_1 and I_2. This is only relevant on initialization $\Delta_{\mathtt{i}}$, since after that states I will not be defined, but we used an epsilon transition instead of just an `i` transition to preserve restartability, which will be discussed in Section 8.5.2. Since we added no other transitions, the least fixed point Equation (8.1) defining the next (or initial) configuration decomposes into the least fixed point on states Q_1, and on states Q_2. It follows that the semantics satisfies $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{u}) = ([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{u}), [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{u}))$. Here, $(\mathbf{y}_1, \mathbf{y}_2)$ denotes the vector $\mathbf{y} \in \overline{\mathbb{D}}^F$ that is $\mathbf{y}_1$ on F_1 and $\mathbf{y}_2$ on F_2. Parallel composition is commutative and associative. The utility of parallel composition is that it allows us to combine the outputs $\mathbf{y}_1$ and $\mathbf{y}_2$ later on. This is accomplished by *concatenation* with another DT which combines the outputs (Section 8.5.2).

Parallel composition. If $\mathcal{A}_1 : I \twoheadrightarrow F_1$ and $\mathcal{A}_2 : I \twoheadrightarrow F_2$, then $\mathcal{A}_1 \parallel \mathcal{A}_2 : I \twoheadrightarrow F_1 \cup F_2$ satisfies

$$[\![\mathcal{A}_1 \parallel \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = ([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}), [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{w})),$$

such that $\mathsf{size}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|I|)$. It therefore matches the set of tag strings $\overline{L}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cap \overline{L}(\mathcal{A}_2)$.

Union. Suppose we are given DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, and assume that the sets of initial and final states are the same up to some bijections: $\pi_1 : I \to I_1$,

$\pi_2 : I \to I_2$, $\rho_1 : F \to F_1$, $\rho_2 : F \to F_2$, for sets I and F with $|I| = |I_1| = |I_2|$ and $|F| = |F_1| = |F_2|$. We wish to define a DT which feeds the input $(\mathbf{x}, \mathbf{w})$ into both DTs in parallel and returns the union $(\sqcup)$ of the two results. We define $\mathcal{A} = \mathcal{A}_1 \sqcup \mathcal{A}_2 = (Q, \Sigma, \Delta, I, F)$ by $Q = Q_1 \cup Q_2 \cup I \cup F$ and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, i', \pi_1(i)', i') : i \in I\big\} \qquad \cup \big\{(\varepsilon, i', \pi_2(i)', i') : i \in I\big\}$$
$$\cup \big\{(\varepsilon, \rho_1(f)', f', \rho_1(f)') : f \in F\big\} \cup \big\{(\varepsilon, \rho_2(f)', f', \rho_2(f)') : f \in F\big\}.$$

Similar to the parallel composition construction, the additional transitions here ensure that we copy values from I into I_1 and I_2, and copy values from F_1 and F_2 into F, whenever these values are defined. In particular, on initialization the initial vector $\mathbf{x}$ will be copied into I_1 and I_2, and on every data word the output values $\mathbf{y}_1$ and $\mathbf{y}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ will be copied into the *same* set of final states, so that they have to be joined by $\sqcup$. In particular, if both $\mathbf{y}_1$ and $\mathbf{y}_2$ are defined, the output will be $\top$. We see therefore that the semantics is such that $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{u}) = [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{u}) \sqcup [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{u})$. Like parallel composition, union is commutative and associative.

Union. If $\mathcal{A}_1 : I \twoheadrightarrow F$ and $\mathcal{A}_2 : I \twoheadrightarrow F$, then $\mathcal{A}_1 \sqcup \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \sqcup \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}) \sqcup [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}),$$

s.t. $\mathsf{size}(\mathcal{A}_1 \sqcup \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|I| + |F|)$. It matches $\overline{L}(\mathcal{A}_1 \sqcup \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cup \overline{L}(\mathcal{A}_2)$.

Prefix summation. Now we consider a more complex operation. Suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and a data word $\mathbf{w}$, such that the output on the empty data word is $\mathbf{y}_1^{(0)}$, the output after receiving one character of the data word is $\mathbf{y}_1^{(1)}$, and in general the output after k characters is $\mathbf{y}_1^{(k)}$. The problem is to return the *sum* of these outputs: we want a DT that returns $\mathbf{y}^{(i)} = \mathbf{y}_1^{(0)} + \cdots + \mathbf{y}_1^{(i)}$ after receiving i characters. This is called the *prefix sum* because $\mathbf{y}_1^{(k)}$ is the value of $\mathcal{A}$ on the kth prefix of the data word. In general, instead of $+$, we can take an arbitrary operation which folds the outputs of $\mathcal{A}_1$ on each prefix. We suppose that this operation is given by a data function G which, for some set F, is a function $\overline{\mathbb{D}}^{F \cup F_1} \to \overline{\mathbb{D}}^{F}$. It takes the previous "sum" $\mathbf{y}^{(i-1)} \in \overline{\mathbb{D}}^{F}$, combines it with the new output of $\mathcal{A}_1$, $\mathbf{y}_1^{(i)} \in \overline{\mathbb{D}}^{F_1}$, and produces the next "sum" $\mathbf{y}^{(i)} \in \overline{\mathbb{D}}^{F}$. So, we'll have $G(\mathbf{y}^{(i-1)}, \mathbf{y}_1^{(i)}) = \mathbf{y}^{(i)}$. We want a DT that, on input initial values for I_1 and initial values $\mathbf{y}^{(-1)}$

for F, will return $\mathbf{y}^{(i)}$. Formally, we convert G to a DT $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, with bijections $\pi : (F \cup F_1) \to I_2$, $\rho : F \to F_2$, which only contains epsilon-transitions: for each term t in G with variables $P \subseteq (F \cup F_1)$ giving a value of $f \in F$, we create an epsilon transition $(\varepsilon, \pi(P)', \rho(f)', t)$. Then we define the prefix sum $\oplus_G \mathcal{A}_1 = (Q, \Sigma, \Delta, (I_1 \cup F), F_2)$, where $Q = Q_1 \cup Q_2 \cup F$ and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, f_1', \pi(f_1)', f_1') : f_1 \in F_1\big\}$$
$$\cup \big\{(\varepsilon, f', \pi(f)', f') : f \in F\big\} \quad \cup \big\{(\sigma, \rho(f), \pi(f)', \rho(f)) : f \in F, \sigma \in \Sigma\big\}.$$

First on the empty data word, the outputs F_2' of $\mathcal{A}_1$ and the initial vector in F' are copied into I_2, and $\mathcal{A}_2$ produces the correct output $\mathbf{y}^{(0)} = [\![G]\!](\mathbf{y}^{(-1)}, \mathbf{y}_1^{(0)})$. Now, when we read in a character in $\Sigma \times \mathbb{D}$, the final states F_2' flow back into inputs to $\mathcal{A}_2$, and the new output of $\mathcal{A}_1$ also flows in. Because the machine $\mathcal{A}_2$ was constructed to be just a set of epsilon-transitions from I_2 to F_2, it does not save any internal state, but just computes the output in terms of the input again. So the next output will be $[\![G]\!](\mathbf{y}^{(0)}, \mathbf{y}_1^{(1)})$, and then $[\![G]\!](\mathbf{y}^{(1)}, \mathbf{y}_1^{(2)})$, and so forth.

Prefix sum. If $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $G : F \cup Z \Rightarrow F$, then $\oplus_G \mathcal{A}_1 : I \cup F \twoheadrightarrow F$ implements the semantics

$$[\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \varepsilon) = [\![G]\!](\mathbf{y}, [\![\mathcal{A}_1]\!](\mathbf{x}, \varepsilon))$$
$$[\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \mathbf{w}(\sigma, d)) = [\![G]\!]([\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \mathbf{w}), [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}(\sigma, d)))$$

such that $\mathsf{size}(\oplus_G \mathcal{A}_1) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(G) + O(|Z| + |F|)$.

Conditioning on undefined and conflict values. A DT that is constructed using the other operations—particularly union, and concatenation and iteration from Section 8.5.2—may produce undefined ($\bot$) or conflict ($\top$) on certain inputs. In such a case, we may want to perform a computation which *conditions* on whether the output is undefined, defined or conflict: for instance, we may want to produce 1 if there is a conflict, or we may want to replace all $\bot$ and $\top$ outputs with concrete data values. (In particular, in Section 8.6, we will want to replace $\bot$ and $\top$ with Boolean values.) We give a construction for this purpose. To simplify the problem, suppose that we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and we want to construct a DT $\mathcal{A}_\bot$ with no initial states, the same set of final states, and the following behavior: for all $\mathbf{x} \in \mathbb{D}^{I_1}$ (*not* $\overline{\mathbb{D}}^{I_1}$), all $\mathbf{u} \in (\Sigma \times \mathbb{D})^*$, and all $f_1 \in F_1$,

if $[\![\mathcal{A}_1]\!](\mathbf{x},\mathbf{u})(f_1) = \bot$ then $[\![\mathcal{A}_\bot]\!](\mathbf{u})(f_1) \in \mathbb{D}$, and otherwise, $[\![\mathcal{A}_\bot]\!](\mathbf{u})(f_1) = \bot$. Here, since $I = \varnothing$, the first argument is omitted. We similarly want to define $\mathcal{A}_\mathbb{D}$ which is in $\mathbb{D}$ if $\mathcal{A}_1$ is in $\mathbb{D}$, and $\bot$ otherwise, and $\mathcal{A}_\top$ which is in $\mathbb{D}$ if $\mathcal{A}_1$ is $\top$, and $\bot$ otherwise. So that $\mathbb{D}$ is not empty, we assume that there is some constant operation in Op_0, say $d_\star$ (so $d_\star \in \mathbb{D}$).

The idea of the construction is that we replace Q_1 with $Q_1 \times \{\bot, \star, \top\}$. For each state $q \in Q_1$, at all times, exactly one of $(q,\bot)$, $(q,\star)$, and $(q,\top)$ will be $d_\star$ and the other two will be $\bot$. Which state is $d_\star$ should correspond to whether q was undefined, defined, or conflict. (This is adapted from the classic trick of dealing with negation by replacing all values with pairs of either (true, false) or (false, true).) However, in order for this to work without blowup our DT needs to be *acyclic*. Therefore we begin with a preliminary stage of converting the DT to acyclic. Observe that in the semantics of Section 8.3.3, iterating the assignment (8.1) $2n$ times would be sufficient to reach the fixed point, where n is the number of states of the DT. So we create $2n$ copies of the states of the DT, with one set of transitions from each copy to the next. In this preliminary stage the size of the transducer may be *squared*, i.e. there is quadratic blowup. Now assuming $\mathcal{A}$ is acyclic, for each variable $q' \in Q_1'$, whether q' is undefined, defined, or conflict is a Boolean function of all the source states of transitions that target q'; this function can be built as a Boolean circuit by adding intermediate states and intermediate transitions, in number at most the total size of the transitions targeting q'. $\mathcal{A}_\bot$, $\mathcal{A}_\mathbb{D}$, and $\mathcal{A}_\top$ differ only in which states are final—$F_1 \times \{\bot\}$, $F_1 \times \{\star\}$, and $F_1 \times \{\top\}$, respectively.

Support. Let $d_\star \in \mathbb{D}$. If $\mathcal{A}_1 : I \twoheadrightarrow F$, then $[\mathcal{A}_1 = \bot] : \varnothing \twoheadrightarrow F$, $[\mathcal{A}_1 \in \mathbb{D}] : \varnothing \twoheadrightarrow F$, and $[\mathcal{A}_1 = \top] : \varnothing \twoheadrightarrow F$. These constructions implement the following semantics. For all $f \in F$:

$$[\![[\mathcal{A}_1 = \bot]]\!](\mathbf{w})(f) = d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x},\mathbf{w})(f) = \bot \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}$$

$$[\![[\mathcal{A}_1 \in \mathbb{D}]]\!](\mathbf{w})(f) = d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x},\mathbf{w})(f) \in \mathbb{D} \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}$$

$$[\![[\mathcal{A}_1 = \top]]\!](\mathbf{w})(f) = d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x},\mathbf{w})(f) = \top \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}$$

such that $\mathsf{size}([\mathcal{A}_1 = \bot]) = O(\mathsf{size}(\mathcal{A}_1)^2)$ and likewise for the other two. Alternatively, if $\mathcal{A}_1$ is acyclic, the size will only be $O(\mathsf{size}(\mathcal{A}_1))$.

8.5.2. Unambiguous Parsing and Restartability

We now want to capture the idea of restartability—that multiple threads of computation may be replaced by updates to a single configuration—with a formal definition. Recall the example in the introduction of $\texttt{split}(f, g, op)$. During the execution of f on input $\mathbf{w}$, whenever the current prefix $\mathbf{u}$ of $\mathbf{w}$ matches, i.e. $f(\mathbf{u}) \neq \perp$, we could (naively and inefficiently) implement $\texttt{split}(f, g, op)$ by keeping a separate configuration (thread) of g from that point forward. For example, suppose that $\mathbf{w} = (\texttt{a}, d_1)(\texttt{b}, d_2)(\texttt{a}, d_3)(\texttt{a}, d_4)$, and that the output of f is defined after receiving each a-item, and undefined otherwise. Then f is defined on input $(\texttt{a}, d_1)$, on $(\texttt{a}, d_1)(\texttt{b}, d_2)(\texttt{a}, d_3)$, and on $(\texttt{a}, d_1)(\texttt{b}, d_2)(\texttt{a}, d_3)(\texttt{a}, d_4)$. Corresponding to these three inputs, we would have three threads of g: $\mathbf{c}_1$ on input $(\texttt{b}, d_2)(\texttt{a}, d_3)(\texttt{a}, d_4)$, $\mathbf{c}_2$ on input $(\texttt{a}, d_4)$, and $\mathbf{c}_3$ on input ε. Suppose that each configuration $\mathbf{c}_i$ includes an final state with the value of $\mathbf{y}_i = op(f(\mathbf{u}), g(\mathbf{v}))$. The value of $\texttt{split}(f, g, op)$ could then be computed as the *union* of the outputs from all these threads: $\texttt{split}(f, g, op)(\mathbf{w}) = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \mathbf{y}_3$. We apply the union here because we expect the split $\mathbf{w} = \mathbf{u} \cdot \mathbf{v}$, where $\mathbf{u} \in \overline{L}(f)$ and $\mathbf{v} \in \overline{L}(g)$, to be unique. Thus all but at most one of $\mathbf{y}_i$ will be $\perp$, and the union gives us the unique answer (if any).

A DT will be called restartable if a *single configuration* $\mathbf{c}$ can *simulate* the behavior of these several configurations $\mathbf{c}_1, \mathbf{c}_2$, and $\mathbf{c}_3$. This is a relation between configurations of g and an arbitrarily long sequence of configurations of g (we could have used a multiset instead of a sequence). The relation $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3]$ is intended to capture that $\mathbf{c}$ is observationally indistinguishable from the sequence $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$. For starters, we require that the output is the same: if $\mathbf{y}$ is the output of $\mathbf{c}$, then $\mathbf{y} = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \mathbf{y}_3$. But we also require that the simulation is preserved when we update the sequence of configurations of g, by reading in a new input character and/or starting a new thread. The definition allows the simulation to be undefined on configurations that are never reachable in an actual execution—it need not be true that *every* sequence $[\mathbf{c}_1, \ldots, \mathbf{c}_k]$ is simulated by some $\mathbf{c}$, but it should be true that every sequence that can be reached by a series of updates is simulated.

With this intuition, the simulation relation on configurations of g should satisfy the following properties (see the definition below). Property (i) addresses the base case before any input characters are received (i.e. initialization `i`). Suppose that on initialization, the machine for g is started with $k \geq 0$ threads, given by initial vectors $\mathbf{x}_1, \ldots, \mathbf{x}_k$. (In our example, these threads would arise as the output of f on initialization.) Then the configuration in a single copy of g on input $\mathbf{x}_1 \sqcup \cdots \sqcup \mathbf{x}_k$ should simulate the behavior of k separate copies of g. Property (ii) requires that the simulation

then be preserved as input characters are read in. Suppose that $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$, and we now read in a character (σ, d) to g. Simultaneously, we start zero or more new threads represented by the vector $\mathbf{x}$ (e.g., $\mathbf{x}$ is the new output produced by f on input (σ, d)). Then if we update and re-initialize the initial states of $\mathbf{c}$ with $\mathbf{x}$, that configuration should simulate updating each $\mathbf{c}_i$ separately, *and* adding one or more new threads represented by $\mathbf{x}$. Finally, property (iii) says that our simulation is sound: for every configuration which simulates a sequence of configurations, the output of the one configuration is equal to the union of the sequence of outputs.

For property (ii) in particular, we need to define what it means to update a configuration $\mathbf{c}$ and simultaneously restart new threads by placing values $\mathbf{x}$ on the initial states I'. (Such an update function is only needed for the simulating configuration, not the sequence of simulated configurations.) For each $\sigma \in \Sigma$ and for every $\mathbf{x} \in \overline{\mathbb{D}}^I$ we define a generalized evaluation function $\Delta_{\sigma,\mathbf{x}} : \overline{\mathbb{D}}^Q \times \mathbb{D} \to \overline{\mathbb{D}}^Q$. This represents executing Δ_σ and then starting *zero or more* new threads, by initializing the new initial states with $\mathbf{x}$. We modify the least fixed point definition of $\mathbf{c}'$ in Equation 8.1) to include the new initialization on states I': $\mathbf{c}'$ is the least vector satisfying

$$\mathbf{c}'(q') = \mathbf{x}(q) \sqcup \bigsqcup_{(\sigma, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X),$$

where $\mathbf{x}(q) = \bot$ if $q \notin I$. This resembles the way we already incorporated $\mathbf{x}$ into the definition of $\Delta_{\mathtt{i}}$. We restrict the vector $\mathbf{x}$ in each restart to be in the space $\mathcal{X} = \{\bot\}^I \cup (\mathbb{D} \cup \{\top\})^I$, which is closed under $\sqcup$. Let $\vec{\bot}$ be the vector with every entry equal to $\bot$.

Definition 8.5.1 (Restartability)**.** Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over signature $(\mathbb{D}, \mathsf{Op})$; let $C = \overline{\mathbb{D}}^Q$ be the set of configurations of $\mathcal{A}$, and $[C]$ the set of *finite lists* of configurations of $\mathcal{A}$. Let $\mathcal{X} = \{\bot\}^I \cup (\mathbb{D} \cup \{\top\})^I$ be the set of possible initializations for a restarted thread. $\mathcal{A}$ is *restartable* if there exists a binary relation $\sim \subseteq C \times [C]$ (called a "simulation") with the following properties:

i. **(Base case)** For all $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathcal{X}$, $\Delta_{\mathtt{i}} \left(\bigsqcup_{i=1}^k \mathbf{x}_i \right) \sim [\Delta_{\mathtt{i}}(\mathbf{x}_1), \ldots, \Delta_{\mathtt{i}}(\mathbf{x}_k)]$. (If $k = 0$, we get $\Delta_{\mathtt{i}}(\vec{\bot}) \sim []$, where $[] \in [C]$ denotes the empty list.)

ii. **(Update with restarts)** For all $(\sigma, d) \in (\Sigma \times \mathbb{D})$, for all $x \in \mathcal{X}$, and for all $\mathbf{c}$, $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$, $\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l$, if $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k]$ and $\Delta_{\mathtt{i}}(\mathbf{x}) \sim [\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l]$ then

$$\Delta_{\sigma,\mathbf{x}}(\mathbf{c}, d) \sim [\Delta_\sigma(\mathbf{c}_1, d), \ldots, \Delta_\sigma(\mathbf{c}_k, d), \hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l].$$

iii. **(Implies same output)** If $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k]$, and the output vectors for these configurations (extended data values at the final states) are $\mathbf{y}, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k$, respectively, then we have $\mathbf{y} = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \cdots \sqcup \mathbf{y}_k$.

A simple example (and counterexample) are in order. First, consider the following DT $\mathcal{A}$ with two states: $Q = \{i, f\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $I = \{i\}$, $F = \{f\}$, and one transition on input $\mathtt{a}$, $f' := i + \mathsf{cur}$. The DT on input $(x, (\mathtt{a}, d))$ returns $x + d$, and on every other input is undefined. Then $\mathcal{A}$ is restartable. We can represent configurations as ordered pairs (x, y), where $x \in \overline{\mathbb{D}}$ is the value of i and $y \in \overline{\mathbb{D}}$ is the value of f. We define that $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$ whenever $\mathbf{c} = \bigsqcup_{i=1}^k \mathbf{c}_i$. Then (i), (ii), and (iii) hold. For example, the base case says that $x = \bigsqcup_{i=1}^k x_k$, then $(x, \bot) \sim [(x_1, \bot), \ldots, (x_k, \bot)]$, which is true by definition. The intuition is that, in this simple case, we can say that a configuration of $\mathcal{A}$ simulates a set of configurations (threads) if the configuration is the union of all those threads. The semantics just takes (x, y) to (z, x) on updating and restarting with z, so it preserves this relation.

For a counterexample, consider a DT $\mathcal{A}$ which sums the value of a single initial state and the last $\mathtt{a}$: take $Q = \{i, f\}$, $I = \{i\}$, $F = \{f\}$, and the following transitions on input $\mathtt{a}$: $i' := i$, $f' := i' + \mathsf{cur}$. We may represent configurations as (x, y), for the values at i, f, respectively. To see this is not restartable, consider starting $\mathcal{A}$ with a single input $x_1 \in \mathbb{D}$, then reading in $(\mathtt{a}, d)$ and starting a second input $x_2 \in \mathbb{D}$ (i.e. applying $\Delta_{\mathtt{a},x_2}$). Starting with x_1 results in the configuration $(x_1, \bot)$; then reading in $(\mathtt{a}, d)$ and starting with x_2 results in $(\top, \top)$. However, if $\mathcal{A}$ were restartable, then by property (ii), we could instead read in $(\mathtt{a}, d)$ and add the second input x_2 separately: we thus would have $(\top, \top) \sim [(x_1, x_1 + d), (x_2, \bot)]$. The problem is that this violates (iii): the output of $\mathcal{A}$ is $\top$, which is not the same as $(x_1 + d) \sqcup \bot = x_1 + d$.

What is relevant for properties (i), (ii), and (iii) is actually only the configurations, input, and output *up to equivalence*, i.e., where we replace $\overline{\mathbb{D}}$ with $\{\bot, \star, \top\}$. There are only finitely many configurations up to equivalence. This is why restartability is decidable (see Theorem 8.5.3).

Concatenation Suppose we have two DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, where F_1 and I_2 are the same up to bijection (say, $\pi : F_1 \to I_2$). Now we want to compute the following parsing operation: on input $(\mathbf{x}, \mathbf{w})$, consider all splits of $\mathbf{w}$ into two strings, $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2$. Apply $\mathcal{A}_1$ to $(\mathbf{x}, \mathbf{w}_1)$ to get a result $\mathbf{y}_1$, and apply $\mathcal{A}_2$ to $(\mathbf{y}_1, \mathbf{w}_2)$ to get $\mathbf{y}_2$. Return the union ($\sqcup$) over all such splits of $\mathbf{y}_2$. In particular, assuming there is only one way to split $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2$ such that $\mathbf{y}_2$ does not end up being undefined, this operation splits the input string uniquely into two parts such that $\mathcal{A}_1$ matches $\mathbf{w}_1$ and $\mathcal{A}_2$ matches $\mathbf{w}_2$, and then applies $\mathcal{A}_1$ and $\mathcal{A}_2$ in sequence.

We implement this by taking $\mathcal{A} = \mathcal{A}_1 \cdot \mathcal{A}_2 = (Q, \Sigma, \Delta, I, F)$ with $Q = Q_1 \cup Q_2$, $I = I_1$, $F = F_2$, and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, \{f_1'\}, \pi(f_1)', f_1') : f_1 \in F_1\big\}.$$

The idea is very simple; every output of $\mathcal{A}_1$ (i.e. a value produced at a state in F_1) should be copied into the corresponding initial state of $\mathcal{A}_2$. This happens on initialization, and on every update. However, the semantics is not so simple, because every time we read in a character, $\mathcal{A}_2$'s initial states I_2 are being re-initialized with new values (the values from F_1).

This "re-initialization" is exactly captured by our generalized update function $\Delta_{\sigma,\mathbf{x}}$ from earlier. Let us represent configurations of $\mathcal{A}$ by $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_i$ is the component restricted to Q_i, i.e. the induced configuration of $\mathcal{A}_i$. Now consider an input $(\mathbf{x}, \mathbf{w})$ to $\mathcal{A}$. We see that for the ith configuration of $\mathcal{A}$ $(\mathbf{c}_1^{(i)}, \mathbf{c}_2^{(i)})$, $\mathbf{c}_1^{(i)}$ is the same as the ith configuration of $\mathcal{A}_1$ on input $(\mathbf{x}, \mathbf{w})$. Moreover, if $\mathbf{y}_1^{(i)}$ is the ith output of $\mathcal{A}_1$, this is used to reinitialize $\mathcal{A}_2$; so we see that $\mathbf{c}_2^{(i)} = \Delta_{\sigma,\mathbf{y}_1^{(i)}}(\mathbf{c}_2^{(i-1)}, d)$ (where this is the update function of $\mathcal{A}_2$). The output $\mathbf{y}_2^{(i)} = \mathbf{c}_2^{(i)}|_F$ of $\mathcal{A}_2$ is the output of $\mathcal{A}$.

Assume that $\mathcal{A}_1$ is *output-synchronized*: this means that each $\mathbf{y}_1^{(i)} \in \mathcal{X}$, i.e., all values are $\perp$ or all values are in $\mathbb{D} \cup \{\top\}$. And assume that $\mathcal{A}_2$ is *restartable*. Then the simulation relation allows us to, at every step, replace $\mathbf{c}_2$ by a list of configurations where each configuration is $\mathcal{A}_2$ on a different suffix of $\mathbf{w}$. In particular, we recursively replace $\Delta_{\sigma,\mathbf{y}_1^{(i)}}(\mathbf{c}_2^{(i-1)}, d)$ with the list of configurations for $\Delta_\sigma(\mathbf{c}_2^{(i-1)}, d)$ and a single new thread $\Delta_{\mathtt{i}}(\mathbf{y}_1^{(i)})$. Because $\mathbf{y}_1^{(i)} \in \mathcal{X}$, this is guaranteed by property (ii) of restartability. Property (iii) then implies the semantics given in the following summary.

Concatenation. Let $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $\mathcal{A}_2 : Z \twoheadrightarrow F$, such that $\mathcal{A}_1$ is output-synchronized and $\mathcal{A}_2$ is restartable. Then $\mathcal{A}_1 \cdot \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \cdot \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = \bigsqcup_{\mathbf{w}=\mathbf{w}_1\mathbf{w}_2} [\![\mathcal{A}_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2).$$

such that $\mathsf{size}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|Z|)$. It matches $\overline{L}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cdot \overline{L}(\mathcal{A}_2)$.

Concatenation with data functions. A special case of concatenation can be described which does *not* require restartability, and which we use in Section 8.6. Suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and we want to concatenate with a data function $G_2 : F_1 \Rightarrow F_2$: on input $(\mathbf{x}, \mathbf{w})$, return $[\![G_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}))$. This can be implemented by converting G_2 into a DT $\mathcal{A}_2$ on states $F_1 \cup F_2$ (as in the prefix sum construction), and then simply constructing $\mathcal{A}_1 \cdot \mathcal{A}_2$. Even if $\mathcal{A}_2$ is not restartable, we can see directly that on every input, the final states F_2 are equal to G_2 applied to the output of $\mathcal{A}_1$. Similarly, if $G_1 : I_1 \Rightarrow I_2$ and $\mathcal{A}_2 : (Q_2, \Sigma, \Delta_2, I_2, F_2)$, then we may convert G_1 into a DT $\mathcal{A}_1$ on states $I_1 \cup I_2$. Then the construction $\mathcal{A}_1 \cdot \mathcal{A}_2$, on every input $(\mathbf{x}, \mathbf{w})$, returns $[\![\mathcal{A}_2]\!]([\![G_1]\!](\mathbf{x}), \mathbf{w})$. We overload the concatenation notation and write these constructions as $\mathcal{A}_1 \cdot G_2$ and $G_1 \cdot \mathcal{A}_2$. For these constructions, as with prefix sum, we do not write out the extended language of matched strings explicitly.

Concatenation with data functions. If $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $G_2 : Z \Rightarrow F$, then $\mathcal{A}_1 \cdot G_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \cdot G_2]\!](\mathbf{x}, \mathbf{w}) = [\![G_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w})),$$

such that $\mathsf{size}(\mathcal{A}_1 \cdot G_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(G_2) + O(|Z|)$. Likewise, if $G_1 : I \Rightarrow Z$ and $\mathcal{A}_2 : Z \twoheadrightarrow F$, then $G_1 \cdot \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![G_1 \cdot \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = [\![\mathcal{A}_2]\!]([\![G_1]\!](\mathbf{x}), \mathbf{w}),$$

such that $\mathsf{size}(G_1 \cdot \mathcal{A}_2) = \mathsf{size}(G_1) + \mathsf{size}(\mathcal{A}_2) + O(|Z|)$.

Iteration Now suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, where I_1 and F_1 are the same up to some bijection. On input $(\mathbf{x}, \mathbf{w})$, we want to split $\mathbf{w}$ into $\mathbf{w}_1\mathbf{w}_2\mathbf{w}_3\ldots$, then apply $[\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}_1)$ to get $\mathbf{y}_1$, $[\![\mathcal{A}_1]\!](\mathbf{y}_1, \mathbf{w}_2)$ to get $\mathbf{y}_2$, and so on. Then, the answer is the union over all possible ways to write $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2\ldots\mathbf{w}_k$ of $\mathbf{y}_k$. Let I be a set the same size as I_1, F_1 with bijections $\pi : I \to I_1$, $\rho : F \to F_1$. Then we implement this by taking $\mathcal{A} = (\mathcal{A}_1)^* = (Q, \Sigma, \Delta, I, I)$ with $Q = Q_1 \cup I$ and

$$\Delta = \Delta_1 \cup \big\{(\varepsilon, \{i'\}, \pi(i)', i') : i \in I\big\} \cup \big\{(\varepsilon, \{\rho(i)'\}, i', \rho(i)') : i \in I\big\}.$$

The idea is again very simple; we have a set of states I that is both initial and final; we always copy the values of these states into the input of $\mathcal{A}_1$ and copy the final states of $\mathcal{A}_1$ back into I. But the semantics is again more complicated. Here (unlike all other constructions), we do not necessarily preserve acyclicity. When we copy F_2 into I and back into I_2, this may then propagate back into F_2 again. Essentially, if $\mathcal{A}_1$ produces output on the empty data word, then $(\mathcal{A}_1)^*$ will always be $\top$, as this will create a cycle with least fixed point $\top$.

We assume that $\mathcal{A}_1$ is both output-synchronized and restartable. We can write configurations of $\mathcal{A}$ as $(\mathbf{c}, \mathbf{y})$, where $\mathbf{c}$ is a configuration of $\mathcal{A}_1$. On an input word $\mathbf{w} = (\sigma_1, d_1), \ldots, (\sigma_k, d_k)$, let the sequence of configurations be $(\mathbf{c}_0, \mathbf{y}_0)$, $(\mathbf{c}_1, \mathbf{y}_1)$, …, $(\mathbf{c}_k, \mathbf{y}_k)$, so the output of $\mathcal{A}$ is $\mathbf{y}_k$. Then the least-fixed-point semantics of Equation (8.1) implies that, for $i = 1, \ldots, k$, $\mathbf{y}_i$ is the least vector satisfying $\mathbf{y}_i = (\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i))\,|_{F_1}$. Similarly, for $i = 0$, $\mathbf{y}_0$ is the least vector satisfying $\mathbf{y}_0 = (\Delta_{\mathtt{i}}(\mathbf{y}_0))\,|_{F_1}$. Now we want to show by induction that $\mathbf{c}_i$ simulates the list, over all possible splits of $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2\cdots\mathbf{w}_k$, of the configuration of $\mathcal{A}_1$ obtained by sequentially applying $\mathcal{A}_1$ k times. The proof of the inductive step is to take the property $\mathbf{y}_i = (\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i))\,|_{F_1}$ and decompose the configuration $\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i)$ using the simulation relation, and see that it simulates the list of all splits $\mathbf{w} = \mathbf{w}_1 \cdots \mathbf{w}_k$ where $\mathbf{w}_k$ has size at least 1, plus the additional initialized thread $\Delta_{\mathtt{i}}(\mathbf{y}_i)$.

Iteration. Let $\mathcal{A} : I \twoheadrightarrow I$ be output-synchronized and restartable. Then $\mathcal{A}^* : I \twoheadrightarrow I$ satisfies

$$[\![\mathcal{A}^*]\!](\mathbf{x}, \mathbf{w}) = \bigsqcup_{\mathbf{w}=\mathbf{w}_1\mathbf{w}_2\cdots\mathbf{w}_k} [\![\mathcal{A}]\!](\ldots[\![\mathcal{A}]\!]([\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2)\ldots, \mathbf{w}_k),$$

s.t. $\mathsf{size}(\mathcal{A}^*) = \mathsf{size}(\mathcal{A}) + O(|I|)$. It matches $\overline{L}(\mathcal{A}^*) = \overline{L}(\mathcal{A})^*$.

Properties of restartability. All operations except "support" preserve restartability. The "output-synchronized" property is also preserved by union, concatenation, and iteration, but is not guaranteed with parallel composition: $\mathcal{A}_1 \parallel \mathcal{A}_2$ is output-synchronized only if $\overline{L}(\mathcal{A}_1) = \overline{L}(\mathcal{A}_2)$.

Theorem 8.5.2. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are restartable, then so are $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \sqcup \mathcal{A}_2$. If $\mathcal{A}_1$ is additionally output-synchronized, then $\mathcal{A}_1 \cdot \mathcal{A}_2$ and ${\mathcal{A}_1}^*$ are restartable. If $\mathcal{A}_1$ is restartable and output-synchronized and additionally $\overline{L}(\mathcal{A}_1) = \Sigma^*$, and if G is a data function where each output value is given by a single term over the input values, then $\oplus_G \mathcal{A}_1$ is restartable.

Proof. For $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \sqcup \mathcal{A}_2$, we represent configurations of the machine has pairs $(\mathbf{c}_1, \mathbf{c}_2)$, and we define $(\mathbf{c}_1, \mathbf{c}_2) \sim [(\mathbf{c}_{1,1}, \mathbf{c}_{2,1}), \ldots, (\mathbf{c}_{1,k}, \mathbf{c}_{2,k})]$ if *both* $\mathbf{c}_1 \sim [\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,k}]$ and $\mathbf{c}_2 \sim [\mathbf{c}_{2,1}, \ldots, \mathbf{c}_{2,k}]$. For prefix sum, the restartability holds for somewhat trivial reasons: if we restart with only $\vec{\perp}$, the output is $\perp$: if we restart with only one non-$\vec{\perp}$ thread, the output is the prefix-sum, and if we restart with two or more non-$\vec{\perp}$ threads, the output is $\top$ everywhere. For concatenation, we have configurations that are pairs $(\mathbf{c}_1, \mathbf{c}_2)$ of a configuration in $\mathcal{A}_1$ and one in $\mathcal{A}_2$. We define $(\mathbf{c}_1, \mathbf{c}_2) \sim [(\mathbf{c}_{1,1}, \mathbf{c}_{2,1}), \ldots, (\mathbf{c}_{1,k}, \mathbf{c}_{2,k})]$ if $\mathbf{c}_1 \sim [\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,k}]$ and *there exists* sequences $l_{2,1}$, $l_{2,2}$, …, $l_{2,k}$, such that $\mathbf{c}_{2,i}$ simulates $l_{2,i}$ and $\mathbf{c}_2$ simulates the entire sequence of sequences, $l_{2,1} \circ l_{2,2} \circ \cdots \circ l_{2,k}$. The idea is that a configuration in $\mathcal{A} = \mathcal{A}_1 \cdot \mathcal{A}_2$ simulates a list of configurations where each configuration consists of only a single thread in $\mathcal{A}_1$, but may have many threads in $\mathcal{A}_2$ (since one thread in $\mathcal{A}_1$ may cause $\mathcal{A}_2$ to be restarted several times). However, we still need that *there exists* some further simulation of the configuration in $\mathcal{A}_2$ into a set of individual threads, such that the overall configuration of $\mathcal{A}_2$ in $\mathcal{A}$ simulates all of these individual threads. For iteration $\mathcal{A} = {\mathcal{A}_1}^*$, we have to do this recursively. The simulation on $\mathcal{A}$ includes $\mathcal{A}_1$ but extends it to the least relation such that whenever $\mathbf{c}_i \sim [\mathbf{c}_{i,1}, \ldots, \mathbf{c}_{i,k}]$ for each i, if $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$ then $\mathbf{c} \sim [\mathbf{c}_{i,j}]_{i,j}$. □

Theorem 8.5.3. Given a DT $\mathcal{A}$ as input, checking if $\mathcal{A}$ is restartable is PSPACE-complete.

Proof. Construct $\mathcal{P}$ as in the proof of Theorem 8.3.2, a DT over $(\mathbf{u}, \mathsf{UOp})$ where $\mathbf{u} = \{\star\}$. Use c_i and p_i to denote configurations of $\mathcal{A}$ and $\mathcal{P}$, respectively.

We first prove a lemma: that $\mathcal{A}$ is restartable iff $\mathcal{P}$ is restartable. The forward direction is immediate: define the relation $p \sim [p_1, p_2, \ldots, p_k]$ if there exists $c \sim [c_1, c_2, \ldots, c_k]$ such that p_i is the projection of c_i to $\mathbf{u}$; then facts (i), (ii), and (iii) are homomorphically preserved. The backward direction is

nontrivial. We need to define the simulation relation between configurations and lists of configurations. We define the *reachable* relation $R \subseteq C \times [C]$ to be the minimal relation that is implied by properties (i) and (ii), i.e. the set of pairs $(c, [c_1, c_2, \ldots, c_k])$ reachable from initialization followed by some sequence of updates-with-restarts $\Delta_{\sigma,\mathbf{x}}$. We will show that R is a simulation by showing that (iii) holds of all reachable pairs. The key observation—which holds even if $\mathcal{A}$ is not restartable—is that for every reachable pair $(c, [c_1, c_2, \ldots, c_k])$, $c \geq c_i$ for all i (where $\geq$ is the coordinate-wise partial ordering on data vectors defined in Section 8.3). This is proven inductively. Using this we claim that R satisfies (iii). Let $(c, [c_1, c_2, \ldots, c_k])$ be reachable. Fix $f \in F$. Since $\mathcal{P}$ is restartable, we know that $c(f)$ and $c_1(f) \sqcup \cdots \sqcup c_k(f)$ are either both undefined, both defined, or both conflict. Thus the only way they can be unequal (violating (iii)) is if they are both in $\mathbb{D}$, and distinct. If they are both in $\mathbb{D}$, then $c_i(f) = \bot$ for all i except one, say $c_j(f) = d'$. But from the key observation above, $c(f) \geq c_j(f)$, and since $c(f), c_j(f) \in \mathbb{D}$, we have equality $c(f) \geq c_j(f)$.

We give a coNPSPACE algorithm to check restartability of a DT $\mathcal{A}$. By the above lemma, it is enough to work with $\mathcal{P}$ instead. So we need to check if there exists a reachable pair $(p, [p_1, \ldots, p_k])$, where p and p_i are configurations of $\mathcal{P}$, such that $F(p) = F(p_1) \sqcup F(p_2) \sqcup \cdots \sqcup F(p_k)$. But choose k to be minimal; then we do not need to keep track of $p_1, \ldots, p_{k-1}$, but can instead collapse these into a single configuration p'. Specifically, before the kth restart, suppose we are at $(p', [p'_1, p'_2, \ldots, p'_{k-1}])$; then rather than keeping p'_1 through p'_{k-1}, we know the output will always be the same as taking p', so we keep track only of p'. Using this trick, the space required to store $(p, [p_1, \ldots, p_k])$ is constant: three configurations of $\mathcal{P}$. Overall, we guess a sequence of moves to get to $(p', [p'_1, \ldots, p'_{k-1}]$, then guess a sequence of moves to get to p from there, and guess a place to stop and try checking if $p(f) = p_1(f) \sqcup p_2(f) \sqcup \cdots \sqcup p_k(f)$ for all $f \in F$. The total space is bounded and some thread accepts if and only if there is a counterexample, meaning the machine is not restartable.

PSPACE-hardness can be shown by a reduction from the problem of universality for NFAs. We carefully exploit that if NFAs N_1 and N_2 are translated to DTs which always output $\bot$ or $\top$, and G is a single binary operation, the DT construction $(N_1 \parallel N_2) \cdot G$ is restartable iff there do not exist strings u, v such that $u \in L(N_1)$, $u \notin L(N_2)$, $uv \notin L(N_1)$, $uv \in L(N_2)$, or vice versa. □

Converting to restartable. It is shown in Theorem 8.7.1 that a DT of size m can be converted to a deterministic CRA of size $\exp(m)$; and that a deterministic CRA of size m can be converted into a *restartable* DT of size $O(m)$. This gives a procedure to convert DT to restartable DT, unfortunately

with exponential blowup. Fortunately, Theorem 8.5.2 guarantees that such exponential blowup does not arise in the compilation of the QRE-Past language of Section 8.6.

8.6. The QRE-Past Monitoring Language

In this section we present the QRE-Past query language for quantitative runtime monitoring (Quantitative Regular Expressions with Past-time temporal operators). Each query compiles to a streaming algorithm, given as a DT, whose evaluation has precise complexity guarantees in the size of the query. Specifically, the complexity is a quadratic number of registers and quadratic number of operations to process each element, in the size of the query, independent of the input stream. Our language employs several constructs from the StreamQRE language [187]. To this core set of combinators we add the `prefix-sum` operation, `fill` and `fill-with` operations, and also past-time temporal logic operators which allow querying temporal safety properties: for example, "is the average of the last five measurements always more than two standard deviations above the average over the last two days?" We have picked constructs which we believe to be intuitive to program and useful in the application domains we have studied, but we do not intend them to be exhaustive; there are many other combinators which could be defined, added to the language, and implemented using the back-end support provided by the constructions of Section 8.5.

By compiling to the DT machine model, we show that the compiled code has the same precise complexity guarantee of the code produced by the StreamQRE engine of [187], including the additional temporal operators. Since compiled StreamQRE code was shown to have better throughput than popular existing streaming engines (RxJava, Esper, and Flink) when deployed on a single machine, this is good evidence that QRE-Past would see similar success with more flexible language constructs.

8.6.1. Syntax of QRE-Past

Expressions in the language are divided into three types: *quantitative queries* of two types, either base-level (α) or top-level (β), and *temporal queries* (φ). Base-level quantitative queries specify functions from data words to quantities (extended data values $\overline{\mathbb{D}}$), and are compiled to *restartable DTs* with a single initial state and single final state, of quadratic size. These queries are based on StreamQRE and the original Quantitative Regular Expressions of [25]. Top-level quantitative queries also specify functions from data words to quantities, but the compiled DT may not be

$$
\begin{array}{lll}
\alpha := \mid \texttt{atom}(\sigma, t) & \{\sigma\} & \sigma \in \Sigma, t \in \mathsf{Tm}[\mathsf{cur}] \\
\quad \mid \texttt{eps}(t) & \{\varepsilon\} & t \in \mathsf{Tm}[\varnothing] \\
\quad \mid \texttt{or}(\alpha_1, \alpha_2) & \overline{L}(\alpha_1) \cup \overline{L}(\alpha_2) & \\
\quad \mid \texttt{split}(\alpha_1, \alpha_2, op) & \overline{L}(\alpha_1) \cdot \overline{L}(\alpha_2) & op \in \mathsf{Op}_2 \\
\quad \mid \texttt{iter}(\alpha_1, init, op) & (\overline{L}(\alpha_1))^* & init \in \mathbb{D}, op \in \mathsf{Op}_2 \\
\quad \mid \texttt{combine}(\alpha_1, \ldots, \alpha_k, op) & \overline{L}(\alpha_1) \cap \cdots \cap \overline{L}(\alpha_k) & op \in \mathsf{Op}_k;\ \text{well-typed if } \overline{L}(\alpha_1) = \cdots = \overline{L}(\alpha_k) \\
\quad \mid \texttt{prefix-sum}(\alpha_1, init, op) & \Sigma^* & init \in \mathbb{D}, op \in \mathsf{Op}_2;\ \text{well-typed if } L(\alpha_1) = \Sigma^* \\
\beta := \mid \alpha_1 & \overline{L}(\alpha_1) & \\
\quad \mid \texttt{fill}(\alpha_1) & L(\alpha_1) \cdot \Sigma^* & \\
\quad \mid \texttt{fill-with}(\alpha_1, \alpha_2) & L(\alpha_1) \cup \overline{L}(\alpha_2) &
\end{array}
$$

$$
\begin{array}{lll}
\varphi := \mid \beta_1 \; comp \; \beta_2 & \Sigma^* & comp \in \{\leq, \geq, =\};\ \text{well-typed if } L(\beta_1) = L(\beta_2) = \Sigma^* \\
\quad \mid \varphi_1 \; bop \; \varphi_2 \quad \mid \neg\varphi_1 & \Sigma^* & bop \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
\quad \mid \odot\varphi_1 \quad \mid \boxdot\varphi_1 \quad \mid \Diamond\varphi_1 & \Sigma^* & \\
\quad \mid \varphi_1 \; \mathcal{S}_w \; \varphi_2 \quad \mid \varphi_1 \; \mathcal{S}_s \; \varphi_2 & \Sigma^* &
\end{array}
$$

Figure 8.6: Summary of the QRE-Past language: syntax for quantitative queries α, β and temporal queries φ. The second column gives the rate of the query as a regular expression.

restartable. Temporal queries specify functions from data words to Booleans, may be constructed from quantitative queries, and are compiled to DTs which output Booleans. Temporal queries are based on the operators of past-time temporal logic [188] and informed by successful existing work on monitoring of safety properties [143], which adapts to our setting via constructions on DTs.

We model Booleans as elements in $\mathbb{D}$. Thus, we assume that $0, 1 \in \mathbb{D}$, and that $\leq, \geq, = \in \mathsf{Op}_2$: these are comparison operations on data values returning 0 or 1. We also assume that we have Boolean operators $\neg \in \mathsf{Op}_1$ and $\wedge, \vee, \rightarrow, \leftrightarrow \in \mathsf{Op}_2$, which treat 0 as false and every $d \neq 0$ as true.

Each query has an associated regular *rate* $\overline{L}(\alpha)$, given by a regular expression on Σ defined recursively with the query. The rate expresses the set of strings on which the compiled DT is defined *or* conflict. For temporal queries φ, the rate is Σ^*. We also may refer to the *language* $L(\alpha) \subseteq \overline{L}(\alpha)$, which is the set of strings on which the compiled DT is defined. There are a few *typing restrictions*, mainly constraints on the rates of the queries. Because each rate is given by a regular expression, the typing restrictions are *type-checkable* in polynomial time. The typing restrictions arise in order to guarantee restartability so that the constructions of Section 8.5 apply.

8.6.2. Semantics and Compilation Algorithm

We describe each construction's semantics, and how it is directly implemented as a DT. For technical reasons, for each quantitative query (*not* for temporal queries) α or β we produce *two* DTs. The first is $\mathcal{A}_\alpha : X \twoheadrightarrow Y$, where $|X| = |Y| = 1$. The semantics will be such that $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w})$ is the value of query α on input $\mathbf{w}$, if x is defined. So x is not really used, except to allow the machine to be restartable (at least one initial state is needed for restarts). The second is $\mathcal{I}_\alpha : X \twoheadrightarrow Y$, where $|X| = |Y| = 1$, which has the following *identity* semantics: $[\![\mathcal{I}_\alpha]\!](x, \mathbf{w}) = x$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) \in \mathbb{D}$, $\top$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) = \top$, and $\bot$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) = \bot$. In particular, $\mathcal{I}_\alpha$ is *equivalent* to $\mathcal{A}_\alpha$ (definition in Section 8.3.5). We use this second machine $\mathcal{I}_\alpha$ to *save* values for using later. For example, to implement $\texttt{split}(f, g, op)$ we concatenate the machine for f with a machine which both saves the output of f *and* starts g; then when g is finished we combine the saved output of f with the output of g via op. We will guarantee in the translation that $\mathcal{I}_\alpha$ has size only linear in the query, but $\mathcal{A}_\alpha$ has worst-case quadratic size.

Atomic expressions: `atom`, `eps`. The atomic expressions are the building blocks of all queries. For $t \in \mathsf{Tm}[\mathsf{cur}]$, the query $\texttt{atom}(\sigma, t)$ matches a data word containing a single character (σ, d), and returns t evaluated with $\mathsf{cur} = d$. Similarly, the query $\texttt{eps}(t)$ matches the empty data word and returns the evaluation of t. Both of these are implementable using a DT with two states, $Q = \{q_i, q_f\}$, with $I = \{q_i\}$ and $F = \{q_f\}$. $\mathcal{A}_{\texttt{atom}(\sigma,t)}$ uses one transition from $\{q_i\}$ to q'_f with term t, and $\mathcal{A}_{\texttt{eps}(t)}$ uses an epsilon transition from $\{q'_i\}$ to q'_f with term t. These machines are restartable by a similar argument as the example immediately following Definition 8.5.1 (alternatively, if they aren't, just convert to an equivalent restartable DT as in Section 8.5.2, last paragraph). The definition of $\mathcal{I}_{\texttt{atom}(\sigma,t)}$ is the same as $\mathcal{A}_{\texttt{atom}(\sigma,t)}$ except that the term t in the transition is replaced by q_i; and likewise for $\mathcal{I}_{\texttt{eps}(t)}$.

Regular operators: `or`, `split`, `iter`. These regular operators are like traditional union, concatenation, and iteration (respectively), except that if the parsing of the string (data word) is not unique, the result will be $\top$. The union operation $\texttt{or}(\alpha_1, \alpha_2)$ should match every data word that matches either α_1 or α_2; if it matches only one, its value is that query, but if it matches both, its value is $\top$. In particular, conflict values "propagate upwards" because even if only one of α_1, α_2 matches, if the value is $\top$ then the result is $\top$. This is exactly the semantics of the DT construction $\mathcal{A}_{\alpha_1} \sqcup \mathcal{A}_{\alpha_2}$.

It is restartable because $\mathcal{A}_{\alpha_1}$ and $\mathcal{A}_{\alpha_2}$ are restartable, by Theorem 8.5.2. Similarly, we can take $\mathcal{I}_{\texttt{or}(\alpha_1,\alpha_2)} = \mathcal{I}_{\alpha_1} \sqcup \mathcal{I}_{\alpha_2}$. Both of these constructions add only a constant to the size.

The operation $\texttt{split}(\alpha_1, \alpha_2, op)$ splits a data word $\mathbf{w}$ into two parts, $\mathbf{w}_1 \cdot \mathbf{w}_2$, such that $\mathbf{w}_1$ matches α_1 and $\mathbf{w}_2$ matches α_2. If there are multiple splits, the result is $\top$; otherwise, the result is $op(\alpha_1(\mathbf{w}_1), \alpha_2(\mathbf{w}_2))$. Here, we have to do some work to save the value of $\alpha_1(\mathbf{w}_1)$ in the DT construction. We implement $\texttt{split}$ as $\mathcal{A}_{\texttt{split}(\alpha_1,\alpha_2,op)} := (\mathcal{A}_{\alpha_1} \cdot (\mathcal{I}_{\alpha_2} \parallel \mathcal{A}_{\alpha_2})) \cdot G_{op}$, where G_{op} is a data function with two inputs y_1, y_2 which returns one output $op(y_1, y_2)$, where y_1 is the final state of $\mathcal{I}_{\alpha_2}$ and y_2 is the final state of $\mathcal{A}_{\alpha_2}$. Let's parse what this is saying. We split the string $\mathbf{w}$ into two parts $\mathbf{w}_1 \cdot \mathbf{w}_2$ such that $\mathbf{w}_i \in \overline{L}(\alpha_i)$, and apply α_1 to the first part; for the second part, we have a transducer which takes the output of α_1 as input and produces both that value as y_1, as well as the new output of α_2 as y_2. Then both of these are passed to G_{op} which returns $op(y_1, y_2)$. To define $\mathcal{I}_{\texttt{split}(\alpha_1,\alpha_2,op)}$ is easier: we take $\mathcal{I}_{\alpha_1} \cdot \mathcal{I}_{\alpha_2}$.

The operation $\texttt{iter}(\alpha_1, init, op)$ splits $\mathbf{w}$ into $\mathbf{w}_1 \cdots \mathbf{w}_k$ such that $\mathbf{w}_i \in \overline{L}(\alpha_1)$ and then *folds* op over the list of outputs of α_1, starting from $init$, to get a result: for instance if $k = 3$, the result is $op(op(op(init, \alpha_1(\mathbf{w}_1)), \alpha_1(\mathbf{w}_2)), \alpha_1(\mathbf{w}_3))$. If the parsing is not unique, the result is $\top$. We implement this as $\mathcal{A}_{\texttt{iter}(\alpha_1,init,op)} := G_{init} \cdot ((\mathcal{I}_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot G_{op})^*$, where G_{init} is a data function which outputs the initial value $init$. The idea here is that $(\mathcal{I}_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot G_{op}$ takes an input, both saves it and performs a new computation $\mathcal{A}_{\alpha_1}$, and then produces op of the old value and the new value. When this is iterated, we get the desired fold operation. For $\mathcal{I}_{\texttt{iter}(\alpha_1,init,op)}$ we can simply take $(\mathcal{I}_{\alpha_1})^*$.

We claim that these constructions preserve restartability. For concatenation, we need that the $\parallel$ is output-synchronized: we need that $\mathcal{A}_{\alpha_2}$ and $\mathcal{I}_{\alpha_2}$ have the same rate. This is true by construction: $\mathcal{I}$ is equivalent to $\mathcal{A}$ and only differs in that it is the identity function from input to output. So the three DTs concatenated are all output-synchronized. Restartability is preserved because the data function G_{op} is converted to a restartable DT in the concatenation construction. The size of the concatenation construction is bounded by a quadratic polynomial because we have added additional size equal to the size of $\mathcal{I}_{\alpha_2}$, which is bounded by a linear polynomial. For iteration, $\parallel$ is similarly only applied to equivalent DTs, and G_{op} is converted to a restartable DT in concatenation. As with $\texttt{split}$, the size of our construction includes the size of $\mathcal{A}_{\alpha_1}$ but adds a linear size due to inclusion of $\mathcal{I}_{\alpha_1}$, so we preserve a quadratic bound on size. The constructions $\mathcal{I}_{\alpha_1} \cdot \mathcal{I}_{\alpha_2}$ and $(\mathcal{I}_{\alpha_1})^*$ preserve a linear bound and are restartable because $\mathcal{I}_{\alpha_1}$ and $\mathcal{I}_{\alpha_2}$ are restartable.

Parallel combination: `combine`. This is the first operation in our language which requires a typing restriction. For $\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)$, the computation is simple: apply every α_i to the input stream to get a result, then *combine* all these results via operation *op*. The implementation as a DT is $\mathcal{A}_{\texttt{combine}(\alpha_1,\ldots,\alpha_k,op)} := (\mathcal{A}_{\alpha_1} \parallel \cdots \parallel \mathcal{A}_{\alpha_k}) \cdot G_{op}$, where G_{op} applies *op* to the k final states of the $\parallel$. For $\mathcal{I}_{\texttt{combine}(\alpha_1,\ldots,\alpha_k,op)}$, we do the same thing but replace *op* by the term y_1 (i.e. we use $G_{y_1} : \{y_1, \ldots, y_k\} \Rightarrow \{y\}$ where y is the final output variable). The construction for `combine` is well-defined even if the typing restriction is not satisfied, but does not preserve restartability in that case. We use the non-restartable version in some other constructions. If the typing restriction *is* satisfied, then this exactly states that the left part of the concatenation is output-synchronized, and given that the right data function is converted to a restartable DT, restartability is preserved. The size of both $\mathcal{A}_{\texttt{combine}(\alpha_1,\ldots,\alpha_k,op)}$ and $\mathcal{I}_{\texttt{combine}(\alpha_1,\ldots,\alpha_k,op)}$ are linear in the sizes of the constituent DTs, so these constructions preserve the quadratic and linear bound on size, respectively.

Prefix sum: `prefix-sum`. The prefix sum $\texttt{prefix-sum}(\alpha_1, \mathit{init}, op)$ is defined only if α_1 is defined (not conflict) on all input. Its value should be $op(\mathit{init}, \alpha_1(\varepsilon))$ on the empty string, and then fold *op* over the outputs of α_1 after that. This is implemented directly using the prefix-sum constructor.

$$\mathcal{A}_{\texttt{prefix-sum}(\alpha_1,\mathit{init},op)} := G_{\mathit{init},\mathit{init}} \cdot \left(\oplus_{G_{op}} \mathcal{A}_{\alpha_1}\right).$$

Here, $G_{\mathit{init},\mathit{init}}$ is a data function to return two copies of *init*. We need two copies because $\mathcal{A}_{\alpha_1}$ has one initial state, which needs an initial value (anything in $\mathbb{D}$ would work just as well).

Fill operations: `fill`, `fill-with`. These operations are ways to *fill in* the values which are $\bot$ and $\top$ with other values. This will not preserve restartability, so it is only allowed in top-level queries; however, it is useful to do this in order to get a query defined on all input data words, so that comparison β_1 *comp* β_2 can be applied. The query $\texttt{fill}(\alpha_1)$ always returns the *last* defined value returned by α_1. For instance, if the sequence of outputs of α_1 is $\bot, \top, 3, \top, 4, 5, \bot$, the outputs of $\texttt{fill}(\alpha_1)$ should be $\bot, \bot, 3, 3, 4, 5, 5$. The query $\texttt{fill-with}(\alpha_1, \alpha_2)$, instead of outputting the last defined value returned by α_2, just outputs the value returned by α_2 if α_1 is not defined. So, if α_2 is the constant always returning 0, the sequence of outputs of $\texttt{fill-with}(\alpha_1, \alpha_2)$ should be $0, 0, 3, 0, 4, 5, 0$.

To accomplish these constructions, we first obtain two DTs $\mathcal{A}_+$ and $\mathcal{A}_-$ which are defined when α_1 is defined and when α_1 is not defined, respectively: $\mathcal{A}_+ = [\mathcal{I}_{\alpha_1} \in \mathbb{D}]$ and $\mathcal{A}_- = [\mathcal{I}_{\alpha_1} = \bot] \sqcup [\mathcal{I}_{\alpha_1} = \top]$. Here, $[= \bot]$ and $[= \top]$ have quadratic blowup, but because we use $\mathcal{I}$ in the argument to those constructions instead of $\mathcal{A}$, $\mathcal{A}_+$ and $\mathcal{A}_-$ only have quadratic size. Now, let $\mathrm{fst}, \mathrm{snd} : \mathbb{D}^2 \to \mathbb{D}$ be the first and second projection operations. Then we implement the fill operations as:

$$\texttt{fill-with}(\alpha_1, \alpha_2) := \texttt{combine}(\mathcal{A}_{\alpha_1}, \mathcal{A}_+, \mathrm{fst}) \sqcup \texttt{combine}(\mathcal{A}_{\alpha_2}, \mathcal{A}_-, \mathrm{fst})$$

$$\texttt{fill}(\alpha_1) := \oplus_G \left(\texttt{combine}(\mathcal{A}_{\alpha_1}, \mathcal{A}_+, \mathrm{fst}) \parallel \mathcal{A}_-\right),$$

where G is a data function which expresses how to update the fill result based on the previous fill result, and whether $\mathcal{A}_{\alpha_1}$ is defined or not: if defined, we should take the new defined value, and otherwise, we should take the old fill result.

Comparison: $\leq, \geq, =$. The semantics of $\beta_1 \mathit{comp} \beta_2$ is just to apply *comp*: for example if *comp* is $<$, and if y_1 and y_2 are the outputs of β_1 and β_2 (which are always defined), then $\beta_1 < \beta_2$ should output $y_1 < y_2$ (which is 0 or 1). Therefore, this construction can be implemented as $\texttt{combine}(\beta_1, \beta_2, \mathit{comp})$. We do not need to worry about restartability for temporal queries, and we also don't define $\mathcal{I}$.

Boolean operators: $\wedge, \vee, \to, \leftrightarrow, \neg$. Similarly, the Boolean operators are implemented by applying the corresponding operation. For example, $\varphi_1 \vee \varphi_2$ is implemented as $\texttt{combine}(\varphi_1, \varphi_2, \vee)$.

Past-temporal operators: $\odot, \boxdot, \Diamond, \mathcal{S}_w, \mathcal{S}_s$. These have the usual semantics on finite traces: for example $\odot(\varphi_1)$ says that φ_1 was true at the previous item, and is false initially, and $\Diamond(\varphi_1)$ says that φ_1 was true at some point in the trace up to this point (including at the present time). The implementation of $\odot$ uses concatenation while the others all use prefix sum. Define $\mathcal{A}_{\odot(\varphi_1)} := \mathcal{A}_{\varphi_1} \cdot \mathcal{I}_\Sigma$, where we define $\mathcal{I}_\Sigma$ to be a DT which matches any data word of length 1, and has the identity semantics (returns the initial value as output). This concatenation is defined because $\mathcal{I}_\Sigma$ is restartable; it has the correct semantics because $\odot$ means to look at the prefix of the input except the last character. For the `prefix-sum` temporal operators, we illustrate only the example of $\Diamond(\varphi_1)$; the other cases are similar. Define a data function G which computes the truth value of $\Diamond(\varphi_1)$ on input $\mathbf{w}(\sigma, d)$ given its truth value on $\mathbf{w}$ and given the truth value of φ_1 on input $\mathbf{w}(\sigma, d)$ (so, G is just disjunction). Define $\mathcal{A}_{\Diamond(\varphi_1)} := G_{0,0} \cdot \oplus_G \mathcal{A}_{\varphi_1}$, where $G_{0,0}$ is a data function outputting two copies of 0 (false) to initialize the computation.

Complexity of QRE-Past evaluation. Our implementations give us the following theorem. In particular, combining with Theorem 8.3.1, the evaluation of any query on an input data stream requires quadratically many registers and quadratically many operations per element, independent of the length of the stream.

Theorem 8.6.1. For every well-typed base-level quantitative query α, the compilation described above via the constructions of Section 8.5 produces a restartable DT $\mathcal{A}_\alpha$ of quadratic size in the length of the query. For every well-typed top-level quantitative query β or temporal query φ, the compilation produces a DT of quadratic size which implements the semantics.

8.7. Succinctness

8.7.1. Comparison with Cost Register Automata

Cost register automata (CRAs) were introduced in [24] as a machine-based characterization of the class of *regular transductions*, which is a notion of regularity that relies on the theory of MSO-definable string-to-tree transductions. One advantage of CRAs over other approaches is that they suggest an obvious algorithm for computing the output in a streaming manner. A CRA has a finite-state control that is updated based only on the tag values of the input data word, and a finite set of write-only registers that are updated at each step using the given operations. The original CRA model is a deterministic machine, whose registers can hold data values as well as functions represented by terms with parameters. Each register update is required to be *copyless*, that is, a register can appear at most once in the right-hand-side expressions of the updates.

In [1], the class of *Streamable Regular* (SR) transductions is introduced, which has two equivalent characterizations: in terms of MSO-definable string-to-dag (directed acyclic graph) transductions without backward edges, and in terms of *possibly copyful* CRAs. Since the focus is on streamability, and terms can grow linearly with the size of the input stream, the registers are restricted to hold only values, not terms. This CRA model is expressively equivalent to DTs.

Theorem 8.7.1. The class of transductions computed by data transducers is equal to the class SR.

Proof sketch. It suffices to show semantics-preserving translations from (unambiguously nondeterministic, copyful) CRAs to DTs and vice versa. Suppose $\mathcal{A}$ is an unambiguous CRA with states

Q and registers X. We construct a DT $\mathcal{B}$ with states $Q \times X$. In the other direction, suppose $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ is a DT. We construct a deterministic CRA $\mathcal{B}$ with states $\{\bot, \star, \top\}^Q$ and variables Q. A configuration of $\mathcal{B}$ consists of a state in $\{\bot, \star, \top\}^Q$ and an assignment $\mathbb{D}^Q$, and therefore uniquely specifies a configuration of $\mathcal{A}$. For each state in $\mathcal{B}$ and each σ, the transition to the next state can be determined from the set of transitions Δ_σ in $\mathcal{A}$. □

However, DTs—even restartable DTs—are exponentially more succinct than (unambiguously nondeterministic, copyful) CRAs. The succinct modular constructions on DTs are not possible on CRAs. For example, the parallel composition of CRAs requires a product construction, whereas the parallel composition of DTs employs a disjoint union construction ($\|$). This is why multiple parallel compositions of CRAs can cause an exponential blowup of the state space, but the corresponding construction on DTs causes only a linear increase in size.

Theorem 8.7.2. For some $(\mathbb{D}, \mathsf{Op})$, (restartable) DTs can be exponentially more succinct than CRAs.

Proof sketch. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, $\mathbb{D} = \mathbb{N}$, and $\mathsf{Op} = \{+\}$ (addition). Suppose that $\mathcal{A}_i$ for $i = 1, \ldots, k$ is a DT that outputs the sum of all values if the input contains σ_i, and 0 otherwise. Notice that $\mathcal{A}_i$ can be implemented with two state variables. Now, $\mathcal{A}$ is the restartable DT with $O(k)$ states that adds the results of $\mathcal{A}_1, \ldots, \mathcal{A}_k$. A CRA that implements the same function as $\mathcal{A}$ needs finite control that remembers which tags have appeared so far. This implies that the CRA needs exponentially many states, and this is true even if unambiguous nondeterminism is allowed. □

8.7.2. Comparison with Finite-State Automata

Another perspective on succinctness is to compare DTs with finite automata for expressing regular languages. To simplify this, consider DTs over a singleton data set $\mathbb{D} = \{\star\}$, with no initial states and one final state. Each such DT $\mathcal{A}$ computes a regular language $\overline{L}(\mathcal{A})$. If we further restrict to *acyclic* DTs, they are exactly as succinct as *reversed alternating finite automata* (r-AFA). In particular, this implies that acyclic DTs (and hence DTs) are exponentially more succinct than DFAs and NFAs.

An r-AFA [69, 235] consists of $(Q, \Sigma, \delta, I, F)$ where the transition function δ assigns to each state in Q a *Boolean combination* of the previous values of Q. For example, we could assign $\delta(q_3) = q_1 \wedge (q_2 \vee \neg q_3)$.

An r-AFA is equivalent to an AFA where the input string is read in the opposite order. The translation from DT to r-AFA copies the states, and on each update, sets each state to be equal to the disjunction of the transitions into it, where each transition is the conjunction of the source variables. Thus, the total size of δ is bounded by the size of the DT. For the other direction, we first remove negation in the standard way; then, conjunction becomes *op* and disjunction becomes $\sqcup$ (multiple transitions with a single target) in the DT.

It is known [69, 109] that L is recognized by a r-AFA with n states if and only if it is recognized by a DFA with 2^n states. This gives an exponential gap in state complexity between acyclic DTs and finite automata, both DFAs and NFAs. To see the gap for NFAs, consider a DFA with 2^n states which has no equivalent NFA with a fewer number of states. Acyclic DTs are a special case, so DTs are exponentially more succinct than both DFAs (uniformly) and NFAs (in the worst case).

8.7.3. Comparison with General Stream-Processing Programs

Finally, we consider a general model of computation for efficient streaming algorithms. The algorithm's maintained state consists of a fixed number of Boolean variables (in $\{0, 1\}$) and data variables (in $\mathbb{D}$), where the Boolean variables support all Boolean operations, but the data variables can only be accessed or modified using operations in Op. The behavior of the algorithm is given by an *initialization* function, an *update* function, a distinguished *output* data variable and a Boolean output flag (which is set to indicate output is present). The initialization and update functions are specified using a *loop-free* imperative language with the following constructs: assignments to Boolean or data variables, sequential composition, and conditionals. This model captures all efficient (bounded space and per-element processing time) streaming computations over a set of allowed data operations Op. We write $\textsc{Stream}(\mathsf{Op})$ to denote the class of such efficient streaming algorithms. The problem with the class $\textsc{Stream}(\mathsf{Op})$ is that it is not suitable for modular specifications. As the following theorem shows, it is not closed under the `split` combinator.

Theorem 8.7.3. Let $\Sigma = \{a, b\}$, $\mathbb{D} = \mathbb{N}$, and let Op be the family of operations that includes unary increment, unary decrement, the constant 0, and the binary equality predicate. Define the

transductions $f, g : (\Sigma \times \mathbb{D})^* \to \mathbb{D} \cup \{\bot\}$ as follows:

$$L(f) = \{w \in \Sigma^* \ : \ |w|_a = 2 \cdot |w|_b\} \qquad L(g) = \{w \in \Sigma^* \ : \ |w|_a = |w|_b\}$$

$$f(\mathbf{w}) = \begin{cases} 1, & \text{if } \mathbf{w} \downarrow \Sigma \in L(f) \\ \bot, & \text{otherwise} \end{cases} \qquad g(\mathbf{w}) = \begin{cases} 1, & \text{if } \mathbf{w} \downarrow \Sigma \in L(g) \\ \bot, & \text{otherwise} \end{cases}$$

where $|w|_a$ is notation for the number of a's that appear in w. Both f and g are streamable functions (i.e. are computable in $\textsc{Stream}(\mathsf{Op})$), but $h = \texttt{split}(f, g, (x, y) \mapsto 1)$ is not.

Proof. Both f and g can be implemented efficiently by maintaining two counters for the number of a's and the number of b's seen so far. On the other hand, any streaming algorithm that computes h requires a linear number of bits (in the size of the stream seen so far). Specifically, consider the behavior of such a streaming algorithm on inputs of the form $a(aab|aba)^n ab$. On these 2^n distinct inputs, each of length $3n + 3$, the streaming algorithm would have to reach 2^n different internal states, because the inputs are pairwise distinguished by reading in a further string of the form b^k. Thus on inputs of size $O(n)$ the streaming algorithm requires at least n bits to store the state. Any streaming algorithm in $\textsc{Stream}(\mathsf{Op})$, however, employs a finite number of integer registers whose size (in bits) can grow only logarithmically. □

Theorem 8.7.3 suggests that some restriction on the domains of transductions is necessary in order to maintain closure under modular constructions. We therefore enforce *regularity* of a generic streaming algorithm by requiring that the values of the Boolean variables depend solely on the input tags. That is, they do not depend on the input data values or the values of the data variables. Under this restriction, a streaming algorithm can be encoded as a DT of roughly the same size.

Theorem 8.7.4. A streaming algorithm of $\textsc{Stream}(\mathsf{Op})$ that satisfies the regularity restriction can be implemented by a DT over Op. This construction can be performed in linear time and space.

Proof sketch. Consider an arbitrary streaming algorithm of $\textsc{Stream}(\mathsf{Op})$ that satisfies the regularity restriction. Each data variable is encoded as a DT state that is always defined. Each Boolean variable b is encoded using two DT states x_b and $x_{\bar{b}}$ as follows: if $b = 0$ then $x_b = \bot$ and $x_{\bar{b}} = d_\star$, and if $b = 1$ then $x_b = d_\star$ and $x_{\bar{b}} = \bot$, where $d_\star$ is some fixed element of $\mathbb{D}$. □

8.8. Discussion

The original paper did not report on an implementation, but we implemented data transducers later as a library in Rust and it is available on GitHub[1]. One direction for future work is to apply this library for either streaming operator performance bounds, query optimization, or both.

The end goal is something like this: part of the library allows writing a query using a distributed variant of quantitative regular expressions. The query is then compiled to Timely with an input and output synchronization schema derived from the query as part of the dataflow representation above.

While this allows for ordering guarantees, one question is how to then derive performance guarantees for the operator as data transducer performance is only *up to* the cost of base operations on the base data types. One possible approach is a combination of empirical and analytical information: using the state machine representation we can calculate an analytical space and time bound for processing items, and then using empirical testing we can derive running time bounds for the operators in the graph. Another approach would be to define an abstract notion of latency and throughput in number of steps only (without the empirical component), and use this to derive logical verification conditions related to number of state updates required to process a single input item for each operator (latency), and number of state updates that can be processed in parallel (throughput).

One notable operation missing from our constructions in Section 8.5 is that of *sequential composition*, in which we pass the sequence of outputs of one machine as the input stream to another. This operation is crucial to many applications, has been included in some previous presentations of QREs [25, 187], and should be considered in future implementations of this work. We omitted it here because it introduces notational complexity: in order to define sequential composition, both the input and output streams need to be tagged (not just the input stream), and (at least) the final states of a DT need to be associated with output tags. An alternative, which better matches the implementation of QREs [187] would be to adopt ideas from the theory of symbolic automata and transducers [100, 91]. In this case, the input and output alphabets would not be tagged, but would both be simple sequences of data items. An important but difficult question would be to design a model that, despite the expressiveness of symbolic transitions combined with quantitative computation, remains closed under sequential composition.

[1] https://github.com/cdstanford/data-transducers

CHAPTER 9 : Outlook

In all affairs it's a healthy thing now and then to hang a question mark on the things you have long taken for granted.

—Attributed to Bertrand Russell [271]

9.1. Vision

The work in this thesis proposes many abstractions for working with streams. These share some common themes of streamability, type safety, and determinism. However, in many other ways the projects have been quite different. A long term vision I have is to integrate them together in a library for streaming that offers all of the benefits of the various abstractions: compositional programming over streams, with safety-preserving automatic distribution under the hood, testing capabilities, and formal performance bounds on operators.

9.2. Short-Term Ideas

We are working an implementation of the type system in Chapter 4 on top of Timely Dataflow [194, 203] in Rust [256]. Timely is a promising choice because it offers a semantically sound low-level dataflow representation, and we aim to leverage Rust's type system for compile-time guarantees, while generating external verification conditions to prove user programs correct. The verification conditions should be formal and interpretable by an appropriate tool such as an SMT solver.

As discussed in Section 4.7.1, another short-term direction is to generalize the type system in Chapter 4. The type system can be studied and generalized in several interesting ways, including Singleton stream types, as well as incorporating other structured stream combinators such as concatenation of stream types. More generally, it could include types that can encode patterns over a stream, and not just sets of events and dependencies between them. These types should also support ways to define producers and consumers for streams in an incremental way, possibly based on derivatives of regular expressions [54, 30]. Our existing work on symbolic derivatives [12] could be a promising starting

point; using symbolic operations is advantageous because patterns over data streams would typically be symbolic. Derivatives have also been defined for quantitative regular expressions [26].

Another interesting direction is how to incorporate edge computing metrics into the stream processing framework to achieve network-aware (or geo-distributed) streaming; some ideas are laid out in [10].

9.3. Long-Term Ideas

More generally, there are many opportunities for future work on distributed and data processing systems in the online setting.

Verified dataflow programming. Today's online dataflow programming frameworks make it too easy for software engineers to write incorrect code. To prevent this, I am interested in designing a library for *verified* dataflow. This effort would build on my work in correctness and performance guarantees for online applications, but would require extending to other families of guarantees (including fault tolerance) and to check the guarantees statically through an SMT or proof assistant back-end. The desired library has to (1) formally specify the required semantics (incorporating partial order requirements, faults, and performance bounds) and (2) expose a usable API with minimal proof burden on the end user. Similar to existing dataflow APIs, a program would be built by chaining together operators in sequence and in parallel (or even connected in cycles), but each of these operators should be annotated with formal requirements on its behavior, and these requirements should compose. One challenge is how to ensure that the formal requirements are met by the operator logic for user-defined, stateful operators: one approach would be to generate external *verification conditions* based on the formal requirements which verify the user's code is satisfactory.

Privacy and security. Researchers are only beginning to explore the question of what privacy and security mean for data-intensive online applications. In cases where most data is aggregated, compressed, or thrown away after an initial pass, what are the appropriate definitions of privacy, including differential privacy, and how can they be ensured? In particular, the differential privacy of streaming algorithms with constant or near-constant space usage should be investigated and quantified. This research would have a direct impact on real-world privacy guarantees if implemented in today's software. In the security of online applications, assumptions need to be articulated on input data (e.g., well-formed input), data rates, and node failures in order to provide security guarantees. I am also interested in the design and implementation of lightweight *specification languages* for privacy

and security in online and cloud applications. All of these questions should be investigated from the spectrum from theory to practice: developing new theories and formal abstractions that are missing in this space, and demonstrating their implementation and utility through practical tools.

Query languages. Decades ago, systems and databases researchers envisioned a world where users implement application logic with simple queries over online data (e.g., using SQL and its variants). The increasing complexity of application logic has moved us steadily away from that goal; in practice, most online applications require very specialized development by distributed systems experts, and even programs written in higher-level frameworks often involve custom logic in the form of stateful functions. Today, we need higher-level abstractions which serve the same purpose but are *intuitive*, have *safe semantics* and are not arbitrary user-defined code. Allowing users to describe queries in *English* would be an even more ambitious goal interdisciplinary with natural language processing. I am currently investigating the design of query languages built on top of stream processing systems to make online applications such as distributed health monitoring systems simply a matter of writing a query and passing it to an online system.

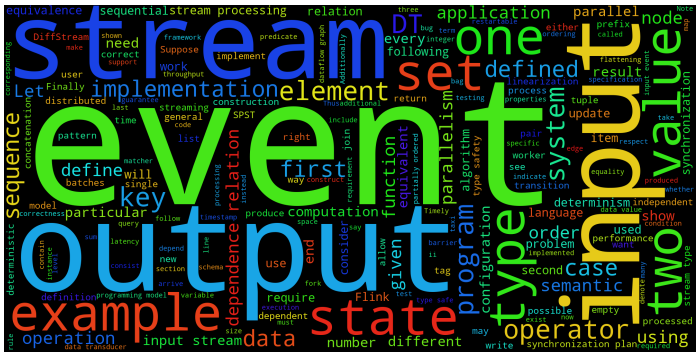
APPENDIX

Well-chosen, non-frivolous epigraphs can enhance a thesis.

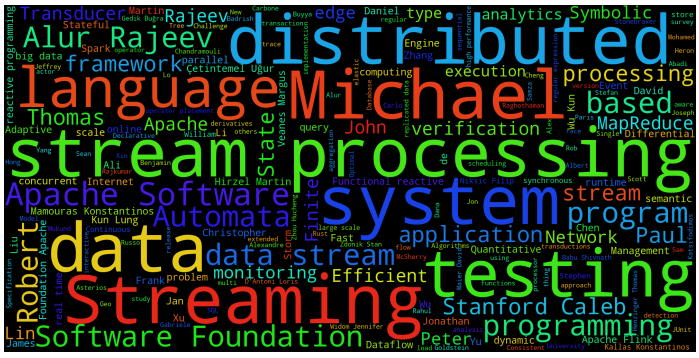
—Dave Clarke [249]

A.1. Word Clouds

Here is a word cloud generated from all the text in this thesis. I used the `wordcloud_cli` Python tool[1] on all `.tex` sources and `sed` to crudely filter out LaTeX-related words. In particular, I filtered out comments (anything after `%`), commands (words starting with \), and selected arguments (contents inside `\begin`, `\end`, `\label`, `\ref`, `\cite`, etc.).



And here is a word cloud for the references (bibliography) file. This only includes text in the `title=` and `author=` fields.



[1] https://github.com/amueller/word_cloud

A.2. Epigraph Outtakes

Elegance is not a dispensable luxury but a quality that decides between success and failure.

—Edsger Dijkstra

Definitions belong to the definers, not the defined.

—Toni Morrison

A good concept is one that is closed (1) under arbitrary composition (2) under recursion.

—Gilles Kahn, 1974 [164]

The restriction of finiteness appears to give a better approximation to the idea of a physical machine. Of course, such machines cannot do as much as Turing machines, but the advantage of being able to compute an arbitrary general recursive function is questionable, since very few of these functions come up in practical applications.

—Rabin and Scott

Almost every problem that you come across is befuddled with all kinds of extraneous data of one sort or another; and if you can bring this problem down into the main issues, you can see more clearly what you're trying to do.

—Claude Shannon

A.3. Special Mention

PRIMARY REFERENCES

[1] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science (TCS)*, 807, 2020.

[2] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. Synchronization schemas. *Invited contribution, Principles of Database Systems*, 2021.

[3] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Automata-based stream processing. In *44th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2017.

[4] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

[5] Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen. Interfaces for stream processing systems. In *Principles of Modeling: Festschrift Symposium in honor of Edward A Lee*. Springer, 2018.

[6] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. DiffStream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[7] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization (extended version). arXiv preprint https://arxiv.org/abs/2104.04512, 2021.

[8] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2022.

[9] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[10] Caleb Stanford. Geo-distributed stream processing. *University of Pennsylvania Doctoral Written Preliminary Exam (WPE-II), Technical Report*, 2020.

[11] Caleb Stanford, Konstantinos Kallas, and Rajeev Alur. Correctness in stream processing: Challenges and opportunities. In *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[12] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

BIBLIOGRAPHY

[13] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[14] Daniel J Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2), 2003.

[15] Houssam Abbas, Rajeev Alur, Konstantinos Mamouras, Rahul Mangharam, and Alena Rodionova. Real-time decision policies with predictable performance. *Proceedings of the IEEE*, 106(9), 2018.

[16] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[17] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. TSpoon: Transactions on a stream processor. *Journal of Parallel and Distributed Computing*, 140, 2020.

[18] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *30th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2004.

[19] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. Stateful functions as a service in action. *Proceedings of the VLDB Endowment*, 12(12), 2019.

[20] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[21] Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. The extensibility framework in Microsoft StreamInsight. In *27th IEEE International Conference on Data Engineering (ICDE)*, 2011.

[22] Shaull Almagor, Udi Boker, and Orna Kupferman. What's decidable about weighted automata? In *9th International Symposium on Automated Technology for Verification and Analysis*, 2011.

[23] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In *39th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2012.

[24] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2013.

[25] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *European Symposium on Programming (ESOP)*, 2016.

[26] Rajeev Alur, Konstantinos Mamouras, and Dogan Ulus. Derivatives of quantitative regular expressions. In *Models, algorithms, logics and tools*. Springer, 2017.

[27] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 8, 2010.

[28] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[29] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2013.

[30] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science (TCS)*, 155(2), 1996.

[31] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[32] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*. Springer, 2003.

[33] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006.

[34] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *International Conference on Management of Data (SIGMOD)*, 2018.

[35] Joe Armstrong. The development of Erlang. In *Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 196–203, 1997.

[36] Kevin Ashton. That 'Internet of Things' thing. *RFID journal*, 22(7), 2009.

[37] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2002.

[38] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4), 2013.

[39] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *Third Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[40] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2004.

[41] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all-an efficient and syntactically idiomatic approach to

management of streams and tables. In *International Conference on Management of Data (SIGMOD)*, 2019.

[42] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *International Conference on Management of Data (SIGMOD)*, 2018.

[43] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), 2003.

[44] Philip A Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. Indexing in an actor-oriented database. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[45] Philip A Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z Tomsic, Shivaram Venkataraman, and Wentao Wu. Serverless event-stream processing over virtual actors. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[46] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[47] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science (TCS)*, 411(4), 2010.

[48] Stephen Blackheath. *Functional reactive programming*. Simon and Schuster, 2016.

[49] Roderick Bloem and Joost Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *Journal of Computer and System Sciences*, 61(1), 2000.

[50] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Transactions on Computational Logic (TOCL)*, 12(4), 2011.

[51] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26(2-4), 2004.

[52] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio FR Geyer, and Luiz Gustavo L Fernandes. DSPBench: a suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8, 2020.

[53] Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. *Theoretical Computer Science (TCS)*, 631, 2016.

[54] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4), 1964.

[55] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[56] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-Up: a complete and automatic linearizability checker. In *31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[57] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[58] William H Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1), 1975.

[59] Manuel Bärenz and Ivan Perez. Rhine: FRP with type-level clocks. In *11th ACM SIGPLAN International Symposium on Haskell*, 2018.

[60] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12), 2017.

[61] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond analytics: The evolution of stream processing systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.

[62] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[63] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *10th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2016.

[64] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9), 2018.

[65] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, 87, 2018.

[66] Gustavo Carneiro. NS-3: Network simulator 3. In *UTM Lab Meeting April*, volume 20, 2010.

[67] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. *ACM Sigplan Notices*, 45(6):363–375, 2010.

[68] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-Paxos for distributed consensus. In *International Symposium on Formal Methods*. Springer, 2016.

[69] Ashok K Chandra, Dexter C Kozen, and Larry J Stockmeyer. Alternation. *Journal of the ACM*, 28(1), 1981.

[70] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[71] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[72] Krishnendu Chatterjee, Laurent Doyen, and Thomas A Henzinger. Quantitative languages. *ACM Transactions on Computational Logic (TOCL)*, 11(4), 2010.

[73] Krishnendu Chatterjee, Thomas A Henzinger, and Jan Otop. Nested weighted automata. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2015.

[74] Krishnendu Chatterjee, Thomas A Henzinger, and Jan Otop. Quantitative monitor automata. In *23rd International Symposium on Static Analysis*, 2016.

[75] Xin Chen, Ymir Vigfusson, Douglas M Blough, Fang Zheng, Kun-Lung Wu, and Liting Hu. GOVERNOR: Smoother stream processing through smarter backpressure. In *IEEE International Conference on Autonomic Computing*, 2017.

[76] Yu-Fang Chen, Lei Song, and Zhilin Wu. The commutativity problem of the MapReduce framework: A transducer-based approach. In *International Conference on Computer Aided Verification (CAV)*. Springer, 2016.

[77] Bin Cheng, Apostolos Papageorgiou, Flavio Cirillo, and Ernoe Kovacs. Geelytics: Geo-distributed edge analytics for large scale IoT systems based on dynamic topology. In *IEEE Second World Forum on Internet of Things*, 2015.

[78] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016.

[79] Cisco. Cisco Global Cloud Index: Forecast and methodology, 2016–2021. White paper. Updated Febuary 1, 2018. (Retrieved from https://virtualization.network/Resources/Whitepapers/0b75cf2e-0c53-4891-918e-b542a5d364c5_white-paper-c11-738085.pdf, July 2022.).

[80] Rebecca L Collins and Luca P Carloni. Flexible filters: load balancing through backpressure for stream programs. In *7th ACM International Conference on Embedded Software*, 2009.

[81] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[82] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Third ACM Symposium on Cloud Computing*, 2012.

[83] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European symposium on programming (ESOP)*. Springer, 2006.

[84] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3), 2013.

[85] Bruno Courcelle. Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science (TCS)*, 126(1), 1994.

[86] Antony Courtney. Frappé: Functional reactive programming in java. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2001.

[87] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. New ideas track: testing MapReduce-style programs. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC-FSE)*, 2011.

[88] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning*. IEEE, 2005.

[89] Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *25th International Conference on Computer Aided Verification (CAV)*, 2013.

[90] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[91] Loris D'Antoni and Margus Veanes. Automata modulo theories. *Communications of the ACM*, 64(5), 2021.

[92] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8), 2016.

[93] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103, 2018.

[94] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[95] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)*, 10(3), 2009.

[96] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1), 2017.

[97] Volker Diekert and Grzegorz Rozenberg. *The Book of Traces*. World Scientific, 1995.

[98] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10), 1972.

[99] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer, 2009.

[100] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification (CAV)*, 2017.

[101] Raphael Eidenbenz and Thomas Locher. Task allocation for distributed stream processing. In *35th Annual IEEE International Conference on Computer Communications*, 2016.

[102] Conal Elliott. Modeling interactive 3d and multimedia animation with an embedded language. In *DSL*, 1997.

[103] Conal Elliott and Paul Hudak. Functional reactive animation. In *Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1997.

[104] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2), 2001.

[105] Joost Engelfriet and Sebastian Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154(1), 1999.

[106] Cristina Valentina Espinosa, Enrique Martin-Martin, Adrián Riesco, and Juan Rodríguez-Hortalá. FlinkCheck: property-based testing for Apache Flink. *IEEE Access*, 7:150369–150382, 2019.

[107] Robert B Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2007.

[108] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. arXiv preprint https://arxiv.org/abs/1711.03829, 2017. (Accessed July 2022.).

[109] Abdelaziz Fellah, Helmut Jürgensen, and Sheng Yu. Constructions for alternating finite automata. *International Journal of Computer Mathematics*, 35(1-4), 1990.

[110] Thomas Ferrère, Thomas A Henzinger, and N Ege Saraç. A theory of register monitors. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2018.

[111] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting statistics over runtime executions. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.

[112] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *Interactions*, 19(3), 2012.

[113] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011.

[114] Apache Software Foundation. Streaming columns – Apache Derby 10.0. https://db.apache.org/derby/, 2013. (Accessed July 2022.).

[115] Apache Software Foundation. Apache Flink. https://flink.apache.org/, 2019. (Accessed July 2022.).

[116] Apache Software Foundation. Apache Heron (originally Twitter Heron). https://heron.incubator.apache.org/, 2019. (Accessed July 2022.).

[117] Apache Software Foundation. Apache Samza. https://samza.apache.org/, 2019. (Accessed July 2022.).

[118] Apache Software Foundation. Apache Spark Streaming. https://spark.apache.org/streaming/, 2019. (Accessed July 2022.).

[119] Apache Software Foundation. Apache Storm. https://storm.apache.org/, 2019. (Accessed July 2022.).

[120] Apache Software Foundation. Concepts – Apache Storm 2.2.0. https://storm.apache.org/releases/2.2.0/Concepts.html, 2019. (Accessed July 2022.).

[121] Apache Software Foundation. Announcing the release of Apache Samza 1.5.1. https://samza.apache.org/blog/2020-08-28-announcing-the-release-of-apache-samza--1.5.1, 2020. (Accessed July 2022.).

[122] Apache Software Foundation. FLIP-8: Rescalable non-partitioned state – Apache Flink. https://cwiki.apache.org/confluence/display/FLINK/FLIP-8%3A+Rescalable+Non-Partitioned+State, 2020. (Accessed July 2022.).

[123] Apache Software Foundation. KTable state stores and improved semantics – Apache Kafka. https://cwiki.apache.org/confluence/display/KAFKA/KIP-114%3A+KTable+state+stores+and+improved+semantics, 2020. (Accessed July 2022.).

[124] Apache Software Foundation. Side inputs for local stores – Apache Samza. https://cwiki.apache.org/confluence/display/SAMZA/SEP-27%3A+Side+Inputs+for+Local+Stores, 2020. (Accessed July 2022.).

[125] Apache Software Foundation. Stateful functions 2.0 – an event-driven database on Apache Flink. https://flink.apache.org/news/2020/04/07/release-statefun-2.0.0.html, 2020. (Accessed July 2022.).

[126] Apache Software Foundation. 0.20.4-incubating release (Apache Heron blog). https://heron.apache.org/blog/2021/05/27/0.20.4-incubating-release, 2021. (Accessed July 2022.).

[127] Apache Software Foundation. Apache Beam. https://beam.apache.org/, 2021. (Accessed July 2022.).

[128] Apache Software Foundation. Announcing the release of Apache Flink 1.15. https://flink.apache.org/news/2022/05/05/1.15-announcement.html, 2022. (Accessed July 2022.).

[129] Apache Software Foundation. Streaming – Apache Calcite. https://calcite.apache.org/docs/stream.html, 2022. (Accessed July 2022.).

[130] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[131] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[132] Nishant Garg. *Apache Kafka*. Packt Publishing Birmingham, UK, 2013.

[133] Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. SPADE: The System S declarative stream processing engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[134] Phillip B Gibbons and Ephraim Korach. The complexity of sequential consistency. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, 1992.

[135] Phillip B Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4), 1997.

[136] 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)*, 2018.

[137] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. LNCS 1032. Springer, 1996.

[138] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[139] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE)*. IEEE, 2007.

[140] Lin Gu, Deze Zeng, Song Guo, Yong Xiang, and Jiankun Hu. A general communication cost optimization framework for big data stream processing in geo-distributed data centers. *IEEE Transactions on Computers*, 65(1), 2015.

[141] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. BigDebug: Debugging primitives for interactive big data processing in Spark. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016.

[142] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

[143] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.

[144] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *25th Symposium on Operating Systems Principles (SOSP)*. ACM, 2015.

[145] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[146] Seth Hettich and Stephen D Bay. KDDCUP 1999 dataset, UCI KDD Archive. https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999. (Accessed July 2022.).

[147] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4), 2013.

[148] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), 2014.

[149] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[150] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Elastic stream processing for the internet of things. In *IEEE 9th International Conference on Cloud Computing*, 2016.

[151] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. SnailTrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[152] Paul Holser. junit-quickcheck: Property-based testing, JUnit-style. https://github.com/pholser/junit-quickcheck, 2013. Commit e0f51f9. (Accessed July 2022.).

[153] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile Fog: A programming model for large-scale applications on the internet of things. In *Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, 2013.

[154] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 2013.

[155] Farzin Houshmand, Javad Saberlatibari, and Mohsen Lesani. Hamband: RDMA replicated data types. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[156] Fabian Hueske. A practical guide to broadcast state in Apache Flink. https://flink.apache.org/2019/06/26/broadcast-state.html, 2019. (Accessed July 2022.).

[157] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Victor Bahl, and Matthai Philipose. VideoEdge: Processing camera streams using hierarchical clusters. In *ACM/IEEE Symposium on Edge Computing*. IEEE, 2018.

[158] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[159] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 grand challenge. In *8th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2014.

[160] Zbigniew Jerzak and Holger Ziekow. DEBS 2014 grand challenge: Smart homes. https://debs.org/grand-challenges/2014/, 2014. (Accessed July 2022.).

[161] Jakob Joachim. *Methodology for Debugging Flink Applications*. Bachelor's thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.

[162] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in Gigascope. In *31st International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2005.

[163] Albert Jonathan, Abhishek Chandra, and Jon Weissman. WASP: Wide-area adaptive stream processing. In *21st International Middleware Conference*, 2020.

[164] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74, 1974.

[165] Gowtham Kaki, Prasanth Prahladan, and Nicholas V Lewchenko. RunTime-assisted convergence in replicated data types. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[166] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science (TCS)*, 134(2), 1994.

[167] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production MapReduce cluster. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010.

[168] Danish Khan, Kshiteej Mahajan, Rahul Godha, and Yuvraj Patel. Empirical study of stragglers in Spark SQL and Spark Streaming. Technical report, University of Wisconsin-Madison, 2015. (Retrieved from https://pages.cs.wisc.edu/~dkhan/sparkstraglers.pdf, July 2022.).

[169] Alexander Kolb et al. Flinkspector: Framework for Apache Flink unit tests. https://github.com/ottogroup/flink-spector, 2019. (Accessed July 2022.).

[170] 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 *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[171] Lindsey Kuper and Ryan R Newton. LVars: lattice-based data structures for deterministic parallelism. In *Second ACM SIGPLAN Workshop on Functional High-Performance Computing*, 2013.

[172] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[173] Geetika T Lakshmanan, Ying Li, and Rob Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6), 2008.

[174] Doug Lea. A Java fork/join framework. In *ACM Conference on Java Grande*, 2000.

[175] Edward A Lee. The problem with threads. *Computer*, 39(5), 2006.

[176] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[177] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5), 2009.

[178] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019.

[179] 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)*, 2012.

[180] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 2013.

[181] Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. Automating distributed partial aggregation. In *ACM Symposium on Cloud Computing*, 2014.

[182] Xunyun Liu and Rajkumar Buyya. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Computing Surveys (CSUR)*, 53(3), 2020.

[183] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos MB Duarte. A performance comparison of open-source stream processing platforms. In *IEEE Global Communications Conference*, 2016.

[184] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[185] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[186] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.

[187] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[188] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[189] João Eugenio Marynowski, Michel Albonico, Eduardo Cunha de Almeida, and Gerson Sunyé. Testing MapReduce-based systems. arXiv preprint https://arxiv.org/abs/1209.6580, 2013. (Accessed July 2022.).

[190] Nathan Marz. Storm (initial release). https://github.com/nathanmarz/storm, 2011. Commit 9d91adb. (Accessed July 2022).

[191] Matthew Maurer and David Brumley. TACHYON: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security)*, 2012.

[192] Antoni Mazurkiewicz. Trace theory. In *Advanced course on Petri nets*. Springer, 1986.

[193] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[194] Frank McSherry et al. Timely Dataflow (Rust implementation). https://github.com/TimelyDataflow/timely-dataflow/, 2014. (Accessed July 2022.).

[195] Frank McSherry et al. Trait Broadcast (Timely Dataflow official documentation). https://docs.rs/timely/latest/timely/dataflow/operators/broadcast/trait.Broadcast.html, 2021. (Accessed July 2022.).

[196] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[197] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Uğur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. S-Store: streaming meets transaction processing. *Proceedings of the VLDB Endowment*, 8(13), 2015.

[198] Mae Milano and Andrew C Myers. MixT: A language for mixing consistency in geodistributed transactions. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[199] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C Myers. A tour of Gallifrey, a language for geodistributed programming. In *Third Summit on Advances in Programming Languages*, 2019.

[200] Mae Milano, Joshua Turcotti, and Andrew C Myers. A flexible type system for fearless concurrency. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[201] Jesús Morán, Claudio de la Riva, and Javier Tuya. Testing MapReduce programs: A systematic mapping study. *Journal of Software: Evolution and Process*, 31(3), 2019.

[202] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3), 2011.

[203] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[204] Shanmugavelayutham Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[205] Christopher Mutschler, Christoffer Löffler, Nicolas Witt, Thorsten Edelhäußer, and Michael Philippsen. Predictive load management in smart grid environments. In *8th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2014.

[206] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Springer Science & Business Media, 2006.

[207] Robert HB Netzer and Barton P Miller. On the complexity of event ordering for shared-memory parallel program executions. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[208] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (later merged into TOPLAS)*, 1(1), 1992.

[209] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3), 2004.

[210] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, 2002.

[211] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12), 2017.

[212] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

[213] Christopher Olston and Benjamin Reed. Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.

[214] Matthew Eric Otey, Amol Ghoting, and Srinivasan Parthasarathy. Fast distributed outlier detection in mixed-attribute data sets. *Data Mining and Knowledge Discovery*, 12(2-3), 2006.

[215] Stack Overflow. Questions tagged with [apache-flink]. https://stackoverflow.com/questions/tagged/apache-flink, 2022. (Accessed June 7, 2022.).

[216] Stack Overflow. Tags. https://stackoverflow.com/tags, 2022. (Accessed June 7, 2022.).

[217] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[218] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[219] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011.

[220] Barbara Partee. Compositionality. *Varieties of Formal Semantics*, 3, 1984.

[221] Barbara Partee, Alice ter Meulen, and Robert E Wall. *Mathematical methods in linguistics*, volume 30. Dordrecht: Kluwer Academic Publishers, 1990.

[222] Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale. SamzaSQL: Scalable fast data management with streaming SQL. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016.

[223] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *International Conference on Computer Aided Verification (CAV)*. Springer, 1994.

[224] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In *9th ACM SIGPLAN International Symposium on Haskell*, 2016.

[225] Ivan Perez and Alwyn E Goodloe. Fault-tolerant functional reactive programming (extended version). *Journal of Functional Programming*, 30, 2020.

[226] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE)*. IEEE, 2006.

[227] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[228] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2), 1959.

[229] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[230] Alex Raizman, Asvin Ananthanarayan, Anton Kirilov, Badrish Chandramouli, and Mohamed H Ali. An extensible test framework for the Microsoft StreamInsight query processor. In *DBTest*, 2010.

[231] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[232] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *IEEE First International Conference on Fog and Edge Computing*, 2017.

[233] Stamatia Rizou, Frank Durr, and Kurt Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *19th International Conference on Computer Communications and Networks*. IEEE, 2010.

[234] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. SpanEdge: Towards unifying stream processing over central and near-the-edge data centers. In *IEEE/ACM Symposium on Edge Computing*, 2016.

[235] Kai Salomaa, Xiuming Wu, and Sheng Yu. Efficient implementation of regular languages using reversed alternating finite automata. *Theoretical Computer Science (TCS)*, 231(1), 2000.

[236] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1), 2017.

[237] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[238] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2), 2013.

[239] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4), 2009.

[240] Marcel Paul Schützenberger. On the definition of a family of automata. *Information and control*, 4(2), 1961.

[241] Koushik Sen. Race directed random testing of concurrent programs. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[242] Amazon Web Servies. Amazon Kinesis: Easily collect, process, and analyze video and data streams in real time. https://aws.amazon.com/kinesis/, 2022. (Accessed July 2022.).

[243] Vivek Shah and Marcos Antonio Vaz Salles. Reactors: A case for predictable, virtualized actor database systems. In *International Conference on Management of Data (SIGMOD)*, 2018.

[244] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 2011.

[245] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), 2016.

[246] Kazuhiro Shibanai and Takuo Watanabe. Distributed functional reactive programming on actor-based runtime. In *8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2018.

[247] Krishnamoorthy C Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[248] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*. Springer, 2010.

[249] Dave Clarke (Academia StackExchange). What are your thoughts on epigraphs in theses? https://academia.stackexchange.com/a/12566/7368, September 10, 2013. (Accessed July 2022.).

[250] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7), 1997.

[251] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4), 2005.

[252] Vertica Systems. Event-based windows – Vertica 9.3.x documentation. https://www.vertica.com/docs/9.3.x/HTML/Content/Authoring/AnalyzingData/SQLAnalytics/Event-basedWindows.htm, 2022. (Accessed July 2022.).

[253] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2007.

[254] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *29th International Conference on Very Large Data Bases (VLDB)*. Elsevier, 2003.

[255] The JUnit Team. JUnit 5 (testing framework). https://junit.org/junit5/, 2019. (Accessed July 2022.).

[256] The Rust Team. Rust programming language. https://www.rust-lang.org/, 2020. (Accessed July 2022.).

[257] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113, 2005. Fourth Workshop on Runtime Verification.

[258] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, 2002.

[259] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009.

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

[261] Nikos Tziritas, Thanasis Loukopoulos, Samee U Khan, Cheng-Zhong Xu, and Albert Y Zomaya. On improving constrained single and group operator placement using evictions in big data environments. *IEEE Transactions on Services Computing*, 9(5), 2016.

[262] Brown University, Brandeis University, and MIT. The Aurora project. https://cs.brown.edu/research/aurora/, 2002. (Accessed July 2022.).

[263] Brown University, Brandeis University, and MIT. The Borealis project. https://cs.brown.edu/research/borealis/public/, 2008. (Accessed July 2022.).

[264] Margus Veanes and Nikolaj Bjørner. Symbolic automata: The toolkit. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2012.

[265] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. Symbolic finite state transducers: Algorithms and applications. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[266] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[267] Alexandre Vianna, Waldemar Ferreira, and Kiev Gama. An exploratory study of how specialists deal with testing in data stream processing applications. arXiv preprint https://arxiv.org/abs/1909.11069, 2019. (Accessed July 2022.).

[268] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6), 2016.

[269] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[270] Ashish Vulimiri, Carlo Curino, P Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[271] DeWitt Wallace and Lila Bell Acheson Wallace. *The Reader's Digest*, volume 37. Reader's Digest Association, 1940.

[272] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[273] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying

distributed systems. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[274] Jeannette M Wing and Chun Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2), 1993.

[275] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2008.

[276] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

[277] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014.

[278] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *IEEE 34th International Conference on Distributed Computing Systems*, 2014.

[279] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[280] Zhihong Xu, Martin Hirzel, and Gregg Rothermel. Semantic characterization of MapReduce workloads. In *IEEE International Symposium on Workload Characterization*, 2013.

[281] Zhihong Xu, Martin Hirzel, Gregg Rothermel, and Kun-Lung Wu. Testing properties of dataflow program operators. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2013.

[282] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[283] Qian Ye and Minyan Lu. SPOT: Testing stream processing programs with symbolic execution and stream synthesizing. *Applied Sciences*, 11(17):8057, 2021.

[284] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with NetQRE. *ACM SIGCOMM Conference on Data Communication*, 2017.

[285] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[286] Jan Zdarek and Carsten W Israel. Detection and discrimination of tachycardia in ICDs manufactured by St. Jude Medical. *Herzschrittmachertherapie + Elektrophysiologie*, 27(3), 2016.

[287] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. AWStream: Adaptive wide-area streaming analytics. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[288] Yunjian Zhao, Zhi Liu, Yidi Wu, Guanxian Jiang, James Cheng, Kunlong Liu, and Xiao Yan. Timestamped state sharing for stream analytics. *IEEE Transactions on Parallel and Distributed Systems*, 32(11), 2021.

[289] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015.