ORDERED TYPES FOR STREAM PROCESSING

Joseph Cutler

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

2026

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

Graduate Group Chairperson
Anindya De, Associate Professor of Computer and Information Science

Dissertation Committee
Stephanie C. Weirich, ENIAC President's Distinguished Professor of Computer and Information Science
Stephan A. Zdancewic, Schlein Family President's Distinguished Professor and Associate Chair of Computer and Information Science
Val Tannen, Professor of Computer and Information Science
Ningning Xie, Assistant Professor of Computer Science, University of Toronto

ORDERED TYPES FOR STREAM PROCESSING

Mom, Dad, and Nathaniel.

ACKNOWLEDGEMENT

While this document bears my name and signifies the completion of five years of PhD-ing, its contents are the combined work of many people over many more years. There is *no way* I could have completed this degree without the love, friendship, and support of the countless friends, family, and mentors who have supported me along the way.

It's been the privilege of a lifetime to be advised by Benjamin Pierce. Benjamin's thoughtful guidance has pushed me to grow in ways I didn't think I could, and flex research muscles I didn't know I had. Benjamin has taught me to never stop asking "why", a lesson which I won't soon forget.

PLClub has been an incredible home for me for the past five years: I'm grateful to every single PLClubber with whom I've overlapped for helping to make the last five years such a positive experience. Particular thanks go to Harry Goldstein, Cassia Torczon, Jessica Shi and Yiyun Liu: thank you for being such excellent friends and colleagues. You will all always have a spot on whatever couch I possess, in thanks for my many hours of hogging the one in your office. Thia Richey and Daniel Sainati: I've been extremely lucky to get to collaborate with you both for the past year, and I'm extremely proud of what you've accomplished — I hope you are too. I can't wait to see what you both do next. Stephanie Weirich, Steve Zdancewic, and Mike Hicks: your generosity with your time and knowledge is unparalleled, and your continued stewardship of the PLClub culture is what makes it such a special place. Val Tannen and Ningning Xie also deserve serious thanks for agreeing to serve on my dissertation committee and advise the construction of this document. Thanks also to my friends elsewhere at Penn CIS: Eli Margolin, Liam Dugan, Alyssa Hwang, and many others have made me proud to be a part of this community.

I also owe the faculty and students of my alma mater, Wesleyan University, an enormous debt. I will be forever grateful to Dan Licata, Norman Danner, Joomy Korkut, Alex Kavvos, and many others for sparking and nurturing my love of this wonderful field.

During my internships away from Penn, I've had an excellent suite of mentors at each company I've had the chance to visit. Thank you to Mike Hicks, Emina Torlak, Chris Casinghino, Daniel Weber, and Vinod Grover: you've all played important parts in helping me develop my research taste and build software skills.

I feel lucky that, of all the fields of computer science I could have ended up in, I found my way to pro-

gramming languages research. The PL community is full of fantastic people; the best corner of computer science I could possibly have found. In particular, José Manuel Calderón Trilla, Aaron Eline, the members of `(Nat -> Nat) -> (Nat -> Nat) -> (Nat -> Nat)`, and the folks I used to chat (argue) with on PL Twitter during the golden years have made me proud to call PL home.

Getting to spend my mid-20s in Philadelphia has been an unexpectedly excellent side-effect of coming to Penn, and I feel grateful to the citizens of this incredible town for letting me call it home for five years. In particular, the members of Philadelphia Runner Track Club and the rotating cast of characters who work at and frequent Café Musette have been excellent friends and company along the way.

Finally, my family. Mom, Dad, and Nathaniel, your love and guidance have continually buoyed me through every phase of the past five years, from the easy and fun to the incredibly challenging. You've constantly supported me as my trajectory has changed, helping me press forward all the while. This PhD is as much yours as it is mine. Je t'aime.

ABSTRACT

ORDERED TYPES FOR STREAM PROCESSING

Joseph Cutler

Benjamin C. Pierce

Stream processing is ubiquitous in modern computing, from distributed data analytics to digital signal processing and AI inference. Stream programs must process data incrementally in order of arrival, while also operating in bounded memory, since inputs are unbounded. Despite its ubiquity, programming models for stream processing remain relatively impoverished compared to general-purpose programming.

Many common stream programming languages and libraries restrict programmers to a fixed set of stream-processing combinators like `map`, `filter`, or `fold`. When a task does not cleanly decompose into a composition of combinators, programmers must drop all the way down to writing streaming state machines manually. Meanwhile, functional programmers enjoy the same combinators for lists, but have a much more pleasant alternative for when combinators don't do the trick: pattern matching and general recursion.

In this dissertation, we build the best of both worlds with the help of ordered types, a lesser-used kind of substructural type system. With ordered types, programmers must use variables in order. By aligning the variable usage ordering with the streaming data arrival ordering, we can give streaming semantics to general recursive functional list programs.

We present two language designs exploring this idea. The first, Delta, uses a bunched ordered type system to guarantee incremental (but not bounded-space) processing. Delta programs execute as state machines that consume input prefixes and produce output prefixes on demand. The type system supports novel stream types including concatenation (a sequential composition of streams) and parallel streams. We investigate the metatheory of Delta and prove that programs are deterministic regardless of the arrival order of parallel data.

The second language, Yoink, uses a different ordered type system to give streaming semantics, this time guaranteeing bounded space usage. Yoink accomplishes this by adopting a pull-based semantics, where programs explicitly request input rather than reacting to pushed data. Last, we build a compiler for Yoink that produces fused imperative programs.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iv

ABSTRACT vi

LIST OF TABLES xi

LIST OF ILLUSTRATIONS xii

Chapter 1 : Introduction 1

1.1 Delta (Chapter 3) 4

1.2 Foundations of Delta in λ^{ST} (Chapter 4) 5

1.3 Pull Streams and λ^{y} (Chapter 5) 7

1.4 Compiling λ^{y} and Yoink (Chapter 6) 8

Chapter 2 : Background and Related Work 10

2.1 What is a Stream? 10

2.1.1 Dataflow Graphs 11

2.1.2 Push, Pull, and State Machines 13

2.2 Stream Programming 17

2.2.1 Unix Streams 18

2.2.2 Functional Stream Programing 18

2.2.3 Stream Programming in Distributed Systems 23

2.2.4 Stream Programming in Databases 26

2.2.5 Stream Programming in Embedded Systems and Hardware Accelerators 27

2.3 Fusion and Streams 30

2.3.1 Fusion with Metaprogramming 31

2.4 Substructural and Ordered Type Systems 31

2.4.1 The Substructural Zoo 33

2.4.2 Bunched Type Systems . . . 33
2.4.3 Ordered Type Systems . . . 34

Chapter 3 : Delta: Functional Programming with Streaming Semantics . . . 35
3.1 Delta Tutorial and Examples . . . 38
3.1.1 Delta's State Machine Semantics . . . 39
3.1.2 The Ordredness Checker . . . 40
3.1.3 Rich Stream Types . . . 41
3.1.4 Writing Combinators in Delta . . . 42
3.1.5 Using State . . . 45
3.1.6 Windowing and Punctuation . . . 47
3.1.7 Parallel Type . . . 50
3.1.8 Partitioning and Routing . . . 50
3.1.9 Determinisitic Merge . . . 51
3.1.10 The Brightness Levels Example . . . 52
3.1.11 FM Radio Example . . . 53
3.2 Delta Implementation Details . . . 55

Chapter 4 : λ^{ST}, the Formal Foundation of Delta . . . 56
4.0.1 Kernel Typing Rules . . . 58
4.0.2 Prefixes and Semantics . . . 61
4.0.3 The Homomorphism Property and Determinism . . . 67
4.1 Full λ^{ST} . . . 70
4.1.1 Sums . . . 70
4.1.2 Star . . . 71
4.1.3 Let-Binding . . . 73
4.1.4 Recursion . . . 74
4.1.5 Stateful Transformers . . . 75

Chapter 5 : λ^{y}, A Functional Calculus with Pull Semantics . . . 79

5.1 Failure of Constant Space in Push . . . 80
5.2 State and Imperative Pull Streams . . . 82
5.3 The Typing Constraints of Pull Streams Functions . . . 84
5.4 The λ^y Type System . . . 85
5.4.1 Poset Definitions . . . 86
5.4.2 Type System . . . 87
5.5 Events . . . 94
5.6 Pull Streams . . . 96
5.7 Semantics . . . 97
5.7.1 Building Pull Graphs . . . 98
5.7.2 Semantics of Pull Graphs . . . 102
5.7.3 Putting it All Together . . . 110

Chapter 6 : Compiling λ^y to Fused Imperative Programs . . . 112
6.1 The Compiler . . . 113
6.1.1 Compiling the Step Function . . . 114
6.1.2 Compiling with Continuations . . . 115
6.1.3 The Rules . . . 116
6.1.4 Compiling Recursive Resets . . . 121
6.1.5 Compiling the Initial State . . . 121
6.2 Yoink . . . 121
6.2.1 Implementation of Yoink . . . 124
6.2.2 Examples of Yoink Programs . . . 125

Chapter 7 : Future Work . . . 132
7.1 Proofs about λ^y . . . 132
7.2 Further Implementation of Yoink . . . 133
7.3 Combinator Safety . . . 133

Chapter A : λ^{ST} Definitions . . . 135

A.1 Basics . . . 135
A.2 Derivatives . . . 138
A.3 Environments . . . 140
A.4 Concatenation . . . 143
A.5 Historical Contexts . . . 146
A.6 Context Subtyping . . . 147
A.7 Type System . . . 148
A.7.0.1 Inertness . . . 148
A.8 Sink Terms . . . 151
A.9 Semantics . . . 153

BIBLIOGRAPHY . . . 160

LIST OF TABLES

LIST OF ILLUSTRATIONS

FIGURE 1.1 Common Functional Streaming Combinators 2

FIGURE 2.1 The Structural Rules 32

FIGURE 4.1 Kernel λ^{ST} syntax and typing rules 58
FIGURE 4.2 Prefixes for Types 62
FIGURE 4.3 Environments for Contexts 63
FIGURE 4.4 Incremental semantics of Kernel λ^{ST} 65
FIGURE 4.5 Derivatives 67
FIGURE 4.6 Rules for Let-Bindings 74
FIGURE 4.7 Historical Program Typing Rule 76

FIGURE 5.1 Typing Rules of λ^{y} 89
FIGURE 5.2 Recursion Control Rules (Part 1) 92
FIGURE 5.3 Recursion Control Rules (Part 2) 93
FIGURE 5.4 Translation from λ^{y} Terms to Pull Graphs 100
FIGURE 5.5 Graph Semantics Rules (Part 1) 103
FIGURE 5.6 Graph Semantics Rules (Part 2) 104
FIGURE 5.7 Graph Semantics Rules (Part 3) 105
FIGURE 5.8 Subgraph Reset Operation 106
FIGURE 5.9 Top-Level Pull Graph Step Rules 111

FIGURE 6.1 Target Language Grammar 113
FIGURE 6.2 Compiling to Accumulator 114
FIGURE 6.3 Compiling to Socket 114
FIGURE 6.4 Compiling to Accumulator, With Continuations 116
FIGURE 6.5 Step Function Compilation Rules (Part 1) 117
FIGURE 6.6 Step Function Compilation Rules (Part 2) 118
FIGURE 6.7 Recursive Reset Compilation Rules 122
FIGURE 6.8 Initial State Compilation Rules 123
FIGURE 6.9 Identity Function in Yoink 126
FIGURE 6.10 First projection in Yoink 126
FIGURE 6.11 Second projection in Yoink 127
FIGURE 6.12 Map in Yoink 127
FIGURE 6.13 Map with body in Yoink 128
FIGURE 6.14 Concat Map in Yoink(Part 1) 129
FIGURE 6.15 Concat Map in Yoink(Part 2) 130
FIGURE 6.16 Concat Map in Yoink(Part 3) 131

FIGURE A.1 λ^{ST} Full Typing Rules (Part 1) 155
FIGURE A.2 λ^{ST} Full Typing Rules (Part 2) 156
FIGURE A.3 λ^{ST} Semantics (Part 1) 157
FIGURE A.4 λ^{ST} Semantics (Part 2) 158
FIGURE A.5 λ^{ST} Recursive Argument Semantics 159

Chapter 1

Introduction

Some of the earliest programmable computers—such as the ENIAC [144], The Mark I [44], and the Z3 [CITE]—were developed to tackle large batch processing tasks. Rather than assign human computers to the task of computing nuclear critical masses and ballistic trajectories, engineers in the 1940s took some early steps towards our world of electronic programmable computers. Lesser-known among these early computers is the MIT Whirlwind [1]. Developed in the late 1940s by researchers at MIT and funded by the Office of Naval Research, the Whirlwind was designed to power a new flight simulator for training bomber crews. Unlike the ENIAC and its batch-processing brethren, the Whirlwind has the honor of being the first *real-time* computer. Rather than performing a single large calculation based on fixed inputs, the flight simulation application demanded *incremental* processing: results generated live as new data arose from pilot inputs. While the Whirlwind was never actually used for flight simulation, the design evolved into SAGE, a US Air Force system for incrementally processing data from radar sites to detect a Soviet attack [1].

This scenario—where the inputs to a computation are not static but must be continually processed as they arrive— is referred to in the modern computing parlance as *stream programming*, or streaming. Streaming consists of essentially two propositions, to wit:

1. *Incrementality*: In scenarios where input data arrives continually over time, there is no sense in waiting until "the end" to produce results. Indeed, there may not be an end! As such, stream processing programs must be incremental, producing outputs when possible as new inputs arrive. In the case of the Whirlwind, this required the ability to take in sensor data — initially conceived as pilot input, and later radar data — and produce visual output on a CRT screen.

2. *Memory Usage*: When a system is expected to process an unbounded sequence of inputs continuously, there is also no sense in holding on to all of it. No matter how big your computer's memory is, attempting to collect and hold onto the full sequence of inputs will breach its storage limit in the fullness of time. Indeed, holding onto any non-constant (in input length) amount of memory

will eventually overflow. Because of this, stream processing programs must use memory resources sparingly, holding onto only a small amount of data as they process inputs.

Three quarters of a century later, stream programming is ubiquitous. Essentially every large-scale program that interacts with the outside world in some way has a streaming component, and many entire domains of computing have a fundamentally streaming flavor. Digital signal processing [117, 188], distributed systems [112, 126, 130, 169], hardware programming [137], data analytics [171], and AI inference [4, 7, 172] all depend heavily on stream programming. But despite the ubiquity of stream processing, the programming models available for building streaming programs in practice remain relatively impoverished, compared to the litany of mature language paradigms available for general-purpose tasks. Programmers who build stream processing programs are often relegated to either manually building state machines or using a limited embedded DSL or library.

In many cases, stream programming libraries are simply a collection of *functional streaming combinators*. These libraries expose an opaque type like `Stream(a)`. To construct, transform, and consume streams, they provide combinators: functions (usually higher-order) that accept and return values of `Stream` type. While each library has its own API and design, some common combinators are: `map`, `filter`, and `fold`, among others (Figure 1.1).

$$
\begin{aligned}
\texttt{map} &: (a \to b) \to \texttt{Stream}(a) \to \texttt{Stream}(b) \\
\texttt{filter} &: (a \to \texttt{Bool}) \to \texttt{Stream}(a) \to \texttt{Stream}(a) \\
\texttt{fold} &: (b \to a \to b) \to b \to \texttt{Stream}(a) \to b \\
\texttt{scan} &: (b \to a \to b) \to b \to \texttt{Stream}(a) \to \texttt{Stream}(b) \\
\texttt{concatMap} &: (a \to \texttt{Stream}(b)) \to \texttt{Stream}(a) \to \texttt{Stream}(b) \\
\texttt{zip} &: \texttt{Stream}(a) \to \texttt{Stream}(b) \to \texttt{Stream}(a \times b)
\end{aligned}
$$

Figure 1.1: Common Functional Streaming Combinators

These combinators for manipulating streams do not actually originate in the stream processing literature. Rather, they come from functional programming [129]. In LISP-family languages like Scheme or Racket and ML-family languages like OCaml or Haskell, linked-list versions of these combinators (where $\texttt{List}(a)$ replaces $\texttt{Stream}(a)$) are provided as part of standard libraries [74, 140]. In these languages, linked

lists are the workhorse datatype, commonly used as a go-to data structure for simple tasks involving all sorts of iteration. One reason that programming with linked lists is so popular in functional languages is that it benefits from a combination of two features common in functional languages: pattern matching and general recursion. List functions pattern match on input, make recursive calls, and typically recurse on the tail of the list (or some list derived from it). This idiom is so ubiquitous that it is commonly taught to beginner functional programmers very early in their journeys. Some patterns of recursion are so common that they deserve to be built into the language: these form the common core of list combinators.

However, when the function a programmer is writing does not cleanly break down as a composition of combinators, they can "break out" and write an arbitrary recursive function using pattern matching. Indeed, while combinators are helpful, writing list programs exclusively using them is not usually desirable. Recent work in PL/HCI has shown that functional programmers are more productive when given the full expressive power of functional list code, compared to when they are restricted to just using combinators [123].

Functional streaming combinator libraries thus play a cruel trick on programmers. By providing *only* the common recursive patterns in the form of combinators—but not allowing for any way to break out and write more general recursive programs with pattern matching—they tie programmers' hands behind their backs. In Chapter 3, we'll see some examples of streaming programs that are significantly simpler to write in direct recursive style. The restriction of streaming libraries to combinators is not artificial or for lack of trying. The incrementality and space-usage guarantees of streaming are not simple to provide, and attempting to give arbitrary recursive list programs a streaming semantics seems extremely challenging. Indeed, efforts to run functional programs—both (a) incrementally over partial data (as it arrives or changes), and (b) in bounded space—have a long and complex history, some of which we touch on in Chapter 2.

In light of all this, the primary contribution of this dissertation is a small magic trick. By using an oft-overlooked kind of substructural type system—ordered types—we can build functional languages with streaming semantics, where programmers still have access to both pattern matching and recursion. Ordered types [6, 114, 148], a sibling of the much more famous linear types [76, 186], require that the variables in a program are used *in order*. By associating this syntactic notion of orderedness with the intrinsically

ordered nature of data streams, we can give streaming behavior to well-typed programs.

In this dissertation, we explore two language designs that use ordered types to provide streaming semantics to programs with functional list syntax[1]. The first language design, dubbed Delta, guarantees programmers incremental data processing without having to leave the confines of familiar functional list programming syntax. Despite our best efforts, it turns out that Delta, while guaranteeing incrementality, does not actually guarantee bounded space usage. The second language design, Yoink, returns to the drawing board, attempting to redesign the language to guarantee bounded space usage. In our effort to do so, we end up completely changing the underlying semantics of the language, and changing the type system to a different (but still ordered) one to reflect the updated constraints.

We begin in Chapter 2 with a broad survey of stream processing across computing, examining the programming models and languages available to programmers for streaming domains ranging from functional programming to distributed systems and embedded hardware. We also recall some relevant foundational concepts in both streaming and PL, including Kahn Process Networks [100], push versus pull stream semantics [47], and substructural type systems.

1.1 Delta (Chapter 3)

In Chapter 3, we describe the language design of Delta, the first of our two language designs. This chapter is a tutorial-style walk through the language design of Delta that serves as a programming-first introduction to many of the type-theoretic concepts we rely on for the rest of the document. We focus on how the type system enables an incremental streaming semantics, what programs are disallowed and why. Most importantly, we give a vision of a world where stream programmers can break free of combinator-only programming.

The design and framing of Delta was developed in concert with my collaborators Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. Some of the content of Chapter 3 is derived from code examples found in the body and appendix of our paper *Stream Types* [48], which was published at the 45th ACM SIGPLAN Conference on Programming Language

[1] To maintain consistency with the published co-authored papers included in this dissertation, I will use "we" to refer to myself, and sometimes the pair of myself and you (the reader). In the rest of this chapter, I make clear which of the following chapters are joint work, and which are mine alone.

Design and Implementation (PLDI 2024). The implementation of Delta can be found at https://github.com/alpha-convert/delta.

Delta programs are syntactically very similar to functional programs in your favorite ML-family language: they support `nil`, `cons`, pattern matching, and general recursion. However, the variables in Delta programs do not range over in-memory data, but rather *streams*, the elements of which will arrive at some point in the future. Functions in Delta thus run incrementally as state machines. When an event from an input stream arrives, the program processes it, produces some output in response, and takes a step to a resultant program, ready to accept future inputs.

This semantics is enabled by Delta's ordered type system. Variables in the typing context are listed in order of their data's arrival, and must be used in that order. This way, when some prefix of the input arrives, there is a corresponding prefix of the program that has all of its inputs ready. The semantics can then run that prefix of the program, produce its output, and "delete it", taking a step to a state where we can run the rest of the program when its inputs arrive.

The proof-theoretic structure of the ordered type system also gives rise to novel *types of streams*, which can statically enforce temporal patterns in the underlying data. For instance, Delta supports a *concatenation type*: given stream types `s` and `t`, we can form the type `s . t`. Streams of this type look like a stream of type `s` followed by a stream of type `t`, with a "punctuation mark" in the middle. Moreover, many (most) stream programs transform multiple input streams that can arrive in parallel. To handle this, Delta supports a different kind of product type, the *parallel streams type*: Given streams of type `s` and `t`, we can form the stream type `s || t`. The values of this type are streams of type `s` *interleaved* with streams of type `t`. The elements of the two substreams may arrive in any order relative to each other. We will see that these types are not merely a curiosity; they enable novel typed programming patterns not available to users of traditional streaming libraries.

1.2 Foundations of Delta in λ^{ST} (Chapter 4)

Chapter 4 presents λ^{ST}, the type theory that serves as the formal foundation of Delta. In this chapter, we dive deep into the type system, operational semantics, and metatheory that support the language design of Delta from the previous chapter. We begin by presenting a subset of calculus (Kernel λ^{ST}) that suffices to

illustrate the key mathematical ideas of using ordered types to guarantee streaming execution. Afterwards, we move on to present the rest of λ^{ST}, which models the various programming-critical features that Delta supports like sum and star types, recursion, and buffering values into memory. The text of this chapter is borrowed verbatim from Sections 3 and 4 of the *Stream Types* paper [48]. I was the primary driver of this project, leading the technical development, proof work, and implementation.

The λ^{ST} type system is based on a novel design. The central challenge is handling both the ordered concatenation type and the unordered parallel type in a single type system. This is accomplished using a *bunched* type system, similar to that of the bunched implication underlying separation logic [141]. Contexts form trees with two kinds of nodes: either a semicolon (;) context connective, or a comma (,) context connective. These two context connectives are subject to different structural rules. Semicolon nodes are ordered and affine — weakening, no exchange, no contraction— while comma nodes are affine — weakening and exchange, no contraction. As such, semicolon induces the concatenation type, while comma induces the parallel type.

The operational semantics of λ^{ST} is similarly technically involved. Instead of using homogeneous sequences of events as the values, λ^{ST}'s semantics operates on *structured stream prefixes*. These are typed values whose shapes are determined by their (stream) types. The operational semantics of λ^{ST} then renders a term as a streaming state machine, accepting prefixes that are sent to it, and producing prefixes as output in response. When a prefix arrives at a λ^{ST} term, it produces an output prefix, and takes a step to a new resultant term which is ready to accept the rest of the input and produce the rest of the output. This resultant is well-typed in a new context and output type, which are given by a derivative (in the sense of Brzozowski [35]) of the original type with respect to the consumed input and produced output.

This chapter also includes three main technical results, all of which are mechanized in Rocq[2]. The first is a soundness theorem, formalizing the preservation argument described informally above. The next is the homomorphism theorem, which states that running a transformer incrementally on prefixes of input and combining the outputs produces the same result as running it on the combined input all at once. Last, the homomorphism theorem has a key corollary: a determinism theorem, which proves that λ^{ST} programs (and hence Delta programs) produce identical results regardless of the arrival order of parallel data.

[2]The development can be found at https://github.com/alpha-convert/lambda-st-proofs

1.3 Pull Streams and λ^{Y} (Chapter 5)

In Chapter 5, we confront an unfortunate fact about Delta and λ^{ST}: while they guarantee incremental processing, they do not guarantee bounded space usage! While the goal of Delta was to build a language for stream processing, it fails on one of the two core principles we set out at the very beginning of this document. We analyze the situation and discover that Delta's lack of bounded-space computation is intrinsic to its *push*-based semantics. In Delta, data arrives when producers send it, not when consumers are ready to receive it. This mismatch can force programs to buffer arbitrary amounts of data if they are not able to keep up with data as it arrives.

In this chapter, we present λ^{Y}: another functional core calculus with stream types, but this time with *pull*-based semantics. With pull streams, the program controls when elements arrive by explicitly requesting them. This way, λ^{Y} terms never need to buffer data they're not yet ready to process, and can run in bounded space. The semantics of λ^{Y} are carefully designed to make this bounded-space property *obvious*.

Instead of directly interpreting λ^{Y} terms, the semantics first translates them into *pull graphs*—directed graphs whose nodes represent tiny primitives, each with a statically-sized piece of mutable state. These graphs then serve as the states in a state-machine operational semantics. Crucially, no unbounded auxiliary data is ever materialized: the entire state of the program is stored in the mutable data in the nodes of the graph. This design makes it plain that programs use bounded space: the only state in the system is the graph itself, which is fully determined by the source program.

One wrinkle of this semantics is that the pull streams of λ^{Y} are fundamentally imperative. When you request the next element of a stream, the element is produced by mutably updating some internal state. Requesting the next element again yields a different element, as the state has advanced. Multiple streams may share underlying state—for instance, when a stream of type $s \cdot t$ is destructed to get its two halves of type s and t, both refer to the same mutable state of the original stream of type $s \cdot t$: the projections are just "views" into the underlying stream. The ordered type system of λ^{Y} guarantees that shared state is used in the correct order. To ensure that the user does not accidentally clobber shared state, type system uses orderedness to track state dependencies and ensure that variables are not used out of order.

Naturally, we must pay a price in some form for the added guarantee of bounded space usage. To

pay this debt, we choose to fork over some expressivness: λ^{Y} is restricted to a smaller subset of recursive programs than λ^{ST}, allowing only "semantically tail-recursive" programs: those that do not require maintaining a call stack, which would need unbounded memory. This restriction is slightly different from traditional tail recursion: common programs like `map`, `concat`, and `concatMap` are directly expressible in λ^{Y} despite not being tail recursive.

The focus of this chapter is essentially a feasibility study: before building a compiler that targets constant-space execution, we first ask whether constant-space execution is achievable at all for a non-combinator calculus of stream programs. While the necessary definitions are presented and the soundness theorem is stated to provide intuition for how λ^{Y} achieves bounded space usage, the primary contribution is the conceptual development that demonstrates the viability of this approach and lays the groundwork for the compiler developed in the next chapter.

The research presented in Chapter 5 is solely my own, and the contents of this chapter have not yet been published.

1.4 Compiling λ^{Y} and Yoink (Chapter 6)

Having established the pull-based semantics of λ^{Y} in the previous chapter, Chapter 6 tackles a more practical question: can we actually execute these programs efficiently? Indeed, an operational semantics that "runs" in constant space is a good proof of concept, but not sufficient. What's really needed is a *compiler* that translates pull graphs into bounded-state imperative programs.

The approach I take is to *specialize* the operational semantics from Chapter 5 to each specific program—a technique often called the Futamura projection [72]. The compiler operates on the pull graph representation used by the λ^{Y} semantics, and targets a minimal imperative language with assignment, conditionals, and loops. The result is a fused imperative program: no intermediate streams are materialized, producing tight loops that only touch static memory and require no allocation. Like previous work on metaprogramming-based stream program compilation [105, 106, 108], this technique produces imperative code from a language of pull streams. However, because λ^{Y} is not restricted to combinators, we can compile and fuse much more general recursive programs written in functional style. Indeed, the class of programs we can compile includes some that are not handled by previous systems—notably, `concatMap` written in

the usual recursive way compiles to fused code [105, 108].

This chapter, like the last, is focused on working towards an implementation, and so it does not include any theoretical development. In particular, I have not proven a compiler correctness theorem connecting the semantics from Chapter 5 and the compiler.

The final chapter concludes with a discussion of Yoink, a small Python eDSL that implements the λ^{y} type system, operational semantics, and compiler. Yoink includes a number of more practical extensions to the λ^{y} calculus, including (bounded) buffering of values and higher-order functions by way of macros. Along with being a great platform for proving out the ideas of λ^{y} and exploring extensions, we discover that Yoink's type system lets it support a more expressive set of imperative pull streams programs than comparable pull stream libraries.

Chapter 2

Background and Related Work

As one of the oldest and broadest concepts in computing, "Stream Processing" is exceptionally hard to pin down into a single pithy definition or even a small set of concepts. Nevertheless, in this chapter we will attempt to define (for the purposes of this document) streams and stream processing, trace their theoretical development, and give an overview of how stream processing is practiced across computing. We will not attempt a definitive or comprehensive history of streams or stream processing—for that, we refer the reader to the lovely extended abstract by Biboudis et al. [27]. After our tour of streaming, we also give background on some other PL preliminaries, including substructural and ordered type systems.

2.1 What is a Stream?

A straightforward answer to the question posed by the title of this section, dating back to the early days of functional programming [37], is that a stream is an unbounded sequence of items, produced by one part of a system (or the external world) and consumed by another. We call these two programs the *producer*, and the *consumer*.

The unboundedness in the above definition has important implications for choices about formalizing streams. Indeed, the theory of streams is usually tied up with notions of infinity, and streams are often modeled as infinite or coinductive objects [164]. For the purposes of this document, streams are *finite but unbounded*: a consumer of a stream knows that it will end eventually, but does not know a priori how long it will be. From the consumer's context, it must always be ready to accept more inputs, which from the local perspective is indistinguishable from being prepared to accept inputs forever.

Streams cannot exist in a vacuum: they are necessarily communication links, between (at least) two programs. We refer to any program that sits at either end of a stream as a *stream program* or *streaming program*. Below, we identify the two key principles that all stream programs must abide by: incrementality, and careful resource usage.

Incrementality The first principle is an extensional concept which we will refer to as incrementality. A program acts incrementally if it can produce partial outputs in response to partial inputs, and will continue to produce subsequent output when subsequent input arrives.[3]

Incrementality is a spectrum: some stream programs are more incremental than others. For example, consider an `addOne` streaming program that accepts a stream of integers. When an integer arrives, `addOne` adds one to it, and sends it out on its output stream. Now, consider a program `addOneInPairs` that also adds one to each element, but sends out elements in pairs, waiting for two elements to arrive before adding one to both and then emitting them. If there is an outstanding singleton element when the stream ends, `addOneInPairs` emits its successor and then ends its output. These programs define identical transformations over an entire history of a stream, but they are different streaming functions: `addOne` is *more incremental* than `addOneInPairs`. Moreover, we consider streaming functions to be incremental even if they only produce output after their input ends: the `sum` program that sums a stream of integers and emits the result once the input has ended is still incremental.

Resource Usage The second principle is an intensional one: resource usage. If streams themselves are potentially unbounded but program memory is bounded, a program that transforms streams has no choice but to conserve resources: attempting to save the full history of inputs is a fool's errand. The particular way that resource constraints apply to stream processing varies by context, and is often determined culturally by the particular field of computing. In the embedded systems setting, programs might only be considered streaming if they use only bounded static memory, while in distributed systems or streaming databases, a stream program might be fine so long as it doesn't use "too much" memory in most cases.

2.1.1 Dataflow Graphs

Stream programs are connected together into *dataflow graphs*. The nodes in a dataflow graph are stream processing programs, and the directed edges between them are streams, carrying data from the output of one program to the input of another. Conceptually, every node maintains an unbounded FIFO queue at each input to ensure no values are dropped during processing. In general, dataflow graphs can be cyclic. In

[3]This incrementality is a refinement of incrementality in the sense of incremental computing/self-adjusting computation [8], where outputs change based on changes in input. In streaming, inputs and outputs only grow.

some models of dataflow the graph is unchanging during execution, while in other models the graph may change. The former is referred to as static (or statically scheduled) dataflow, while the latter is referred to as dynamic dataflow. The modern concept of the dataflow graph originates from computer architecture literature [18, 53, 184], where they were used for parallel processor design. However, the concept of state machines communicating over channels goes back to at least the development of Petri Nets [145].

One common use of dataflow graphs is as the semantics of *dataflow languages*: declarative programming languages whose semantic model is dataflow graphs. Some examples of this are the Lustre, Signal, and Esterel languages [26, 39, 119]. These languages are primarily used in the development of embedded and safety-critical systems, so we postpone their discussion until Section 2.2.5. Languages with semantics based ondataflow graphs also appear in machine learning: frameworks like TensorFlow, PyTorch, and JAX are essentially dataflow languages [4, 70, 142].

Kahn Process Networks and Determinism

Kahn Process Networks (KPNs) [100] are an early and influential work on streaming. While originally designed to model concurrent programs, KPNs effectively describe a dataflow streaming graph. Edges in a KPN are infinite streams, while nodes with inputs $X_1, \ldots, X_n$ and outputs $Y_1, \ldots, Y_m$ are ω-continuous [5] functions $X_1 \times \cdots \times X_n \to Y_1 \times \cdots \times Y_m$.

The graph begins in an initial state, with values pre-set in the nodes' FIFOs. Then, evaluation begins by repeatedly nondeterministically choosing a node and "firing" it, taking the values out of its FIFOs and running them through the nodes' functions. These domain-theoretic condition that each node be ω-continuous guarantees that, in the limit, this process converges to the least fixedpoint of a set of domain equations.

While this is an important theoretical result, the KPN paper's main enduring insight is a simple operational one: to ensure that a streaming program does in fact define an ω-continuous function out of the product of its inputs, it suffices to guarantee that every time the program attempts to read from one of its input channels, it must block until an event occurs on that channel. In other words, programs cannot be allowed to "peek" to check for potential inputs. This guarantees that the program defines a function that takes all of its inputs "independently," and can be modeled as a map out of a product, instead of a map out

of the possible interleavings of its inputs.

The graph-wide deterministic execution property of KPNs is extremely desirable. Not only are individual nodes deterministic functions, but (in the limit) the evaluation of the graph is uniquely determined by the initial conditions. However, the so-called "Kahn Condition," that stream programs are not allowed to peek, is too strong to be practical. It turns out that a weaker semantic condition—called "eager evaluation" by Laddad et al [113], "homomorphism" by our work and Hou et al. [90], and "factorization independence" by Mamouras [125]—is sufficient to guarantee that streaming programs are definable as maps out of a product of their inputs, and hence deterministic.

2.1.2 Push, Pull, and State Machines

Streams can be characterized by whether they are *push* or *pull*. These perspectives differ in which end—the sender or the receiver—controls the flow of data through the stream [60].

With push streams, the producer controls when stream items are sent. This represents the conventional conception of streams, requiring uni-directional communication. A consumer of a push stream must either keep pace with the incoming data or buffer elements as they arrive to avoid losing information. Push streams naturally arise as external or networked data sources where information just arrives when it arrives. Conversely, pull streams give control to the consumer, who can request the next element when it is ready. This approach is more efficient for the consumer but may be less efficient for the producer, which must wait for requests before sending data. Pull streams can also be thought of as a form of imperative coroutine [116].

Pull and push streams also differ theoretically. A push stream is something that a consumer must be ready to accept elements of, and so they are defined by their folds: a pull stream is:

```
Stream a = forall b.  b -> (b -> a -> b) -> b
```

Meanwhile, a pull stream is something that a consumer can request the next element from, and thus is defined by its unfold [47]:

```
Stream a = exists s.  (s, s -> Maybe (s,a))
```

In the object-oriented world, push streams are referred to as Observers, while pull streams are Iterators [73].

Somewhere between push and pull is streams with backpressure, which are fundamentally push streams—sending data as they see fit—until a receiver starts to struggle to keep up, wherein it can request that the upstream producer please slow down.

Functional Push Streams and Combinators

From the perspective of a consumer, a push stream is something to be *handled*. From a stream of `a`s, one must be ready to accept an `a` at any time. This intuition is made precise by folds: given a push stream, one can commit to a way of folding over its elements to produce a value. Formally, we define push streams *as* their folds:

```
PushStream a = forall b.  (b -> a -> b) -> b -> b
```

A `PushStream a` is, for any choice of type `b`, a plan for incrementally turning a sequence of `a`s into a `b`. With this definition in hand, we can explore a few combinator implementations using push streams[4].

```
map :: (a -> b) -> PushStream a -> PushStream b
map f s = \step init -> s (\b a -> step b (f a)) init

filter :: (a -> Bool) -> PushStream a -> PushStream a
filter f s = \step init -> s (\b a -> if f a then step b a else b) init

sum :: PushStream Int -> Int
sum s = s (\a b -> a + n) 0
```

The `map` combinator transforms elements by wrapping the step function to apply `f` to each incoming element before passing it along. Similarly, `filter` first checks the predicate, and only "passes along" values that have succeeded. The `sum` shows an instance of consuming the push stream directly, using addition as the step function and zero as the initial accumulator. Note that all three combinators say nothing about *when* values are accepted or produced: they stand ready to accept values, and produce a whole `b` value per step.

```
zip :: PushStream a -> PushStream b -> PushStream (a,b)
```

[4]In this document, we will occasionally use pseudo-Haskell as pseudocode.

```
zip s1 s2 = ??
```

However, the push streams are limited in their expressivness: they cannot express combinators that consume multiple streams simultaneously, such as `zip`, shown above[5]. The problem is (conceptually) that the two arguments produce elements independently, and so there is no way to "synchronize" them. The only way to do so would be to convert both streams into lists, zipping them together in memory, and streaming them out.

Functional Pull Streams

Pull streams [47] provide exactly the sort of control lacking above. A pull stream produces its elements only when asked, producing the first on the first request, the second on the next, and so on. As such, pull streams are essentially state machines, keeping track of and updating some state that determines which value they're expected to produce next. This concept is formalized with the following pair of type definitions:

```
Step s a = Done | Skip s | Yield s a

PullStream a = exists s.  (s,s -> Step s a)
```

A `PullStream a` encapsulates some state of type `s`, and a function `s -> Step s a`, which turns the current state into a decision to (a) be `Done` with the stream, (b) simply produce a new state without producing a value (`Skip`), or (c) `Yield` a value of type `a`, stepping the state along the way.

```
map :: (a -> b) -> PullStream a -> PullStream b
map g (x0,f) = (x0,f')
  where
    f' x = case f x of
          Done -> Done
          Skip x' -> Skip x'
          Yield x' a -> Yield x' (g a)

filter :: (a -> Bool) -> PullStream a -> PullStream a
```

[5]Exercise to the reader: convince yourself that you cannot fill in this hole satisfactorally

```
filter p (x0,f) = (x0,f')
  where
    f' x = case f x of
           Done -> Done
           Skip x' -> Skip x'
           Yield x' a -> if p a then Yield x' a else Skip x' a

sum :: PullStream Int -> Int
sum (x0,f) = go x0 0
  where
    go x acc = case f x of
               Done -> acc
               Skip x' -> go x' acc
               Yield x' a -> go x' (acc + a)

zip :: PullStream a -> PullStream b -> PullStream (a,b)
zip (s1,f1) (s2,f2) = ((s1, s2, Nothing), step)
  where
    step (x1, x2, Nothing) =
      case f1 x1 of
        Done -> Done
        Skip x1' -> Skip (x1', x2, Nothing)
        Yield x1' a -> Skip (x1', x2, Just a)
    step (x1, x2, Just a) =
      case f2 x2 of
        Done -> Done
        Skip x2' -> Skip (x1, x2', Just a)
        Yield x2' b -> Yield (x1, x2', Nothing) (a, b)
```

The `map` and `filter` combinators are structurally similar to their push stream counterparts, wrapping the step function to transform or conditionally skip elements. The `sum` combinator consumes the stream by pulling all of the values in a loop and accumulating them. Crucially, unlike push streams, pull streams can express `zip`. The `zip` combinator uses a `Maybe a` in its state to hold onto a single element from the first stream while waiting for an element from the second stream. When both streams have yielded values, it emits the pair and clears the buffer. This works because pull streams give the consumer control—`zip`

can decide when to pull from each stream, synchronizing them explicitly.

It is possible to convert between the two stream representations, though each direction makes different tradeoffs. To turn a push stream into a pull stream, we must buffer the values as they arrive into a list, and then serve them on demand. Conversely, to turn a pull stream into a push stream, we simply pull all values out as fast as possible and feed them to the push stream's step function. These transformations are witnessed by the following pair of functions, which implement the above ideas:

```
pushToPull :: PushStream a -> PullStream a
pushToPull s = (xs,f)
  where
    xs = s (\ys y -> y : ys) []
    f [] = Done
    f (y:ys) = Yield ys y

pullToPush :: PullStream a -> PushStream a
pullToPush (x₀,f) g y₀ = go x₀ y₀
  where
    go x y =
      case f x of
      Done -> y
      Skip x' -> go x' y
      Yield a x' -> go x' (g y a)
```

Pull streams are likely familiar to imperative programmers, though likely under a different name. If we make the step function actually imperative—allowing it to perform side effects and maintain internal mutable state—we arrive at the familiar Gang of Four Design Pattern known as *iterators*, found in languages like Java, Rust, and Python [73]. Indeed, iterators in imperative languages are essentially imperative pull streams, and the combinator-based APIs found in modern languages (such as Rust's `Iterator` trait or Python's `itertools`) are essentially functional pull stream libraries.

2.2 Stream Programming

In which contexts do programmers encounter stream programming, and how do they do it? In this section, we survey some of the many domains of computing in which programmers write stream processing code,

and discuss the programming models available to them in each. While this dissertation is focused primarily on functional-style streaming libraries, we survey here the broad range of other ways that programmers can write stream programs across domains.

2.2.1 Unix Streams

One of the oldest and most prominent instances of stream programming is in the Unix shell [158]. In Unix, everything is a file, and some files—those that are actually IO, or for whatever reason do not have random access with `lseek(2)`—are streams. There are several such streams built into Unix, including the three "standard" streams (`stdin`, `stdout`, and `stderr`) for every process, as well as some special constant streams like `/dev/zero` (the all-zero stream), `/dev/null` (a sink), and `/dev/random` (a stream of random bytes). Moreover, many other types of files are actually streams, like pipes [93] or BSD sockets [183].

Because of this, Unix shell scripting is essentially a very simple stream processing language. The shell includes a small set of primitives like redirects and `tee` for orchestrating and marshalling streams between executables. The actual stream transformations are handled by a rich ecosystem of tools like `grep`, `sed`, and `awk`, as well as any executables that calls out to the POSIX API [92].

2.2.2 Functional Stream Programing

In functional programming, streams are an abstraction of fundamental importance, used both to write code that interacts with the outside world and also just as a flexible way to structure iteration. Since these two points cover such a huge surface area of important programming tasks—from networking and file IO [45, 166] and parsing to looping over collections [97]—any sufficiently large functional program includes some amount of stream processing in one form or another. Using streams in FP is not a new idea; in fact, it is one of the oldest! Streams were identified as a core abstraction in some of the earliest work on functional programming, including work by Landin [116] and McCarthy [129].

Because the use cases of streams are so varied, most functional languages do not include a single type for streams in their standard library, instead opting to farm stream support out to libraries. Streaming libraries exist in basically every functional language with any industry usage, including Haskell, Scala, OCaml, F#, and more. Many (but certainly not all) of these functional streaming libraries are also built on

top of a concurrency or async programming framework, allowing programmers to define compositions of streaming operations and then execute them concurrently as dataflow graphs within a process.

Generically speaking, functional streaming libraries expose a type like `Stream a`, where `a` is a base type in the language. To construct, transform, and consume streams, they provide *combinators*: functions (often higher order) that accept and return values of `Stream` type. While each library has its own API and design, some common combinators are `map`, `filter`, `foldl`, `scanl`, and `concatMap`. These functions are directly inspired by the related operations on lists. Over the entire history of a stream, these operations do the same thing as their list function counterpart. However, the stream versions run incrementally (and if possible in bounded space), producing new outputs in response to new inputs.

The other important family of operations widely supported by functional streaming libraries are windowing (also known as "chunking") operations. Windowing operations take a stream and turn it into a stream of "windows" of contiguous elements, based on some criterion. For example, a so-called sliding window groups elements into overlapping fixed-size chunks. For each new element, the window "slides" forward by dropping the oldest element and adding the newest. Meanwhile, "tumbling" windows group elements into disjoint chunks, collecting a fixed number of elements before emitting them all together as a window. Windowing operations similarly run incrementally, emitting windows incrementally as they are ready.

In some sense, the above operations are a "shared API" between nearly all stream processing libraries in functional languages. While implementation details differ, functional streaming APIs basically all include these operations.

In the following few sections, we review some major streaming libraries in functional languages.

Scala

Scala has several mature functional streaming libraries. Two of the most prominent competitors are ZIO Streams and FS2. ZIO streams has push-based semantics with backpressure support. It also emphasizes concurrent execution of streaming pipelines, using the structured concurrency primitives of the broader ZIO ecosystem [195]. FS2 is a pull-based streaming library built on top of the Typelevel ecosystem of libraries [182]. Like ZIO, FS2 also supports concurrent execution, instead using Typelevel's cats-effect

library.

OCaml

The OCaml standard library includes a lightweight stream type `'a Seq.t`, the values of which are essentially lazy lists. This API is very minimal, eschewing support for windowing or concurrency for a simpler implementation and lack of dependencies. For a more full-featured streaming library, OCaml programmers often look to `iter`, a full-featured push-based streaming library, or the `Lwt_stream.t` type from the `lwt` concurrency library [149].

Haskell

Haskell is a somewhat different story. Because Haskell is a nonstrict language, values of the default list type are actually streams. Moreover, some functions on lists in Haskell also have streaming semantics. It is folklore in the Haskell community that a "sufficiently lazy" list program can be run incrementally, over a list defined by an external process using a clever trick with lazy IO [178].

This is quite cool, but not a practical way to build streaming programs. For one, the "sufficient laziness" condition is syntactically brittle, and requires an expert Haskell programmer to carefully ensure that all functions involved are lazy in the just the right way. Secondly, lazy IO in Haskell comes with significant drawbacks [104]. The core issue is that lazy IO introduces nondeterminism in when file handles and other resources are acquired and released. This unpredictable resource management can lead to resource leaks, premature closing of files that are still needed, or keeping files open long after they're no longer required.

Because of this, Haskell programmers almost never use the built-in streaming capabilities of their language, instead reaching for streaming libraries like Pipes [78], Iteratees [104], FoldL [79], Conduit [167], Streamly [175], and others to ensure their programs (a) have a streaming semantics, and (b) correctly manage IO resources.

These libraries differ mostly on their treatment of effects, which is an endless source of debate in the Haskell community. Because Haskell is pure, streaming libraries in Haskell usually include types for *effectful streams*, which can alternate between producing pure values and running effectful computations.

Some Haskell streaming libraries also carve out other niches for exploration besides effects. For exam-

ple, Pipes is based on a sophisicated categorical model of streaming [78], Streamly aims for the highest-possible performance using GHC rewrite rules [175], and Conduit emphasizes deterministic resource handling [167].

Functional Reactive Programming

FRP [59, 61, 138, 187] is not strictly a form of stream processing, but it is sufficiently important and related that it bears mentioning here. While traditional stream processing focuses on discrete sequences of values, FRP — introduced by Elliot and Hudak in their seminal paper [61] — is centered around programming with time-varying values called signals (or behaviors). These signals can be conceptualized (and are often implemented) as functions of type `Time -> a` describing the values at each point in time.

Unlike stream processing, bounded resource usage was not originally a central driving force behind FRP development, and early FRP implementations suffered from widespread "space leaks" and other performance issues. However as FRP has become more widely adopted, modern FRP implementations and theories have evolved to address these challenges [22, 109]. Concurrently, the FRP community has also investigated parallel and concurrent extensions to the FRP model [22, 23].

While the core type of FRP has remained time-varying signals, more sophisticated types for FRP have been investigated. Work by Jeffrey [99] permits the type of a signal to vary over time, using dependent types inspired by Linear Temporal Logic [147]. Concurrently with our work, Bahr and Møgelberg proposed both (1) a modal type system for asynchronous FRP [21, 63], and (2) a pull-based imperative FRP language [20]. Lastly, Paykin and Krishnaswami [143] developed a modal type system which expresses low-level state machines.

While FRP was spawned from the Haskell community, it has found a following more broadly, especially with the RX family of libraries [151–153]. Applications-wise, FRP is often used in domains similar to stream processing, in applications like GUIs, games, and robots [46].

Session Types

Session Types [89] describe complex sequential protocols between communicating processes in a process calculus as they evolve through time. Process calculi are more general than stream processing, and can

define much more complex bidirectional patterns of communication than dataflow models of streaming. Another difference between session types and streams is that the session type of a process describes the *protocol* for its communications with other processes—i.e., the sequence of sends and receives on different channels—while the types of streams program describes only the data that it communicates. Indeed, a program producing a value of type `Stream a` may send all of its values at once extremely quickly in response to no input, or it may wait for all of its input to arrive before producing any output.

Incremental Computing

Incremental computing—initiated by Acar [8]—is a paradigm where program outputs automatically "adjust" in response to changes in their inputs. This is much more general than streaming: the idea is that you write a program from arbitrary inputs to outputs, and a compiler or library "incrementalizes" it for you. Sometimes this is conceptualized as accepting a stream of *deltas* to inputs and producing a stream of deltas to outputs. One key way this differs from streaming (and our notion of incrementality) is that it is non-monotone. In general, the data structures in incremental computing can change in arbitrary ways, which makes implementing this efficiently and building incremental data structures quite challenging. In streaming (and our conception of incrementality for the purpose of this document), the inputs and outputs only grow. Examples of incremental computing systems include Delta ML [9], Adaptive in Haskell [24], and Jane Street's Incremental library [98]. This concept also shows up in the databases literature as incremental view maintenance [84], which we discuss later.

Emerging Models

Concurrently with the work of this dissertation, researchers continue to investigate other completely novel programming models for stream processing. Recent work by Rioux and Zdancewic [156] investigates a datalog-inspired lambda calculus λ_{V} where a program increases in definedness as it evaluates, essentially defining a stream of its own growing value. Meanwhile, Mell et al. [133] work towards a world where Python scripts (in this case used glue together calls to a large language model) can execute asyncrhonously as a dataflow graph ordered by dependency.

2.2.3 Stream Programming in Distributed Systems

In situations where high-throughput or high-volume streams must be processed, programmers often build distributed systems to do stream processing. Distributed streaming takes the dataflow graph model of streaming to its logical conclusion, seeking to dramatically improve the throughput and performance of large streams of data by running logical graphs of streaming operations on distributed clusters of networked machines [68].

The application domain of distributed stream processing is somewhat more specialized than generic stream programming. A very common use for distributed streaming is data analytics [32, 139, 171], where either the volumes of data involved are too large to store (and hence batch processing is infeasible), or because results need to be live and presented in real-time. Distributed streaming also forms the backbone of AI infrastructure, both for training and inference [11, 70].

This basic conceptual model is straightforward, but the introduction of networking and multiple nodes leads to many challenges, mostly of the system design variety. Indeed, a majority of the academic and industrial advances in this area have been around how to successfully build and operate stream processing clusters. Some issues in this domain are state management across distributed nodes [67, 111, 122] checkpointing and recovery mechanisms for the inevitable node failures, as well as exactly-once and at least once delivery guarantees for streams themselves [68].

The default programming model for distributed stream processing derives almost entirely from the functional streaming combinators model of Section 2.2.2. Of course, the reality of a distributed implementation must leak into programs somewhat. In this section, we will discuss the particular ways that programming distributed stream processing systems differs from traditional FP streaming.

Low-Level State Machines and Distributed Dataflow

Initially, distributed streaming programs were not build with high-level abstractions and functional libraries, but instead built by manually describing the physical structure of dataflow graphs, and manually writing state machines to run on the nodes.

A very early and influential instance of this was Apache Storm [68], a Java-based library for defin-

ing distributed dataflow graphs. In Storm, dataflow graphs are known as "topologies," borrowing from networking terminology. Storm topologies are directed acyclic graphs, where data flows from sources, through some number of transformations, to sinks, where they exit the system. Nodes in a Storm topology are defined as observers [73]: objects with an event-handling function, which is called when an element arrives on one of the node's input streams. Despite its relatively low-level programming model, Storm has been extremely influential, laying the foundation for distributed streaming with fault-tolerance and delivery guarantees.

Since Storm, alternate distributed dataflow programming models have been proposed. The most prominent among them is Timely, which imposes more structure on dataflow graphs to ensure certain properties about event timing across distributed workers. In Timely, built-in punctuation lets nodes make decisions based on when upstream components have completed their computation, even when the dataflow graphs are cyclic [132]. Differential Dataflow [131], an incremental computation framework built atop Timely, powers the Materialize streaming database [130].

Some other academic work attempts to build very general compile targets for other stream processing libraries, for example Brooklet [168] and the DON Calculus [55].

Keyed Parallelism and Partitioning

Most modern distributed stream programming libraries mimic the combinator API of functional streaming libraries. Examples of these include Twitter's Summingbird [32], Apache Flink [66], Apache Beam [69], Apache Spark [192], and Kafka Streams [13]. The basic idea of these libraries is to let programmers declaratively specify dataflow graphs in the form of a more regular functional stream program, and then automatically compile and distribute them to a physically-parallel programming running on a streaming framework like Storm. These particular libraries and their ilk do differ in (a) how they choose to turn a streaming computation into a dataflow graph, and (b) the different distributed systems guarantees they give to programmers, but all expose extremely similar FP-style streaming APIs with the usual FP combinators of Section 2.2.2.

The main thing added to distributed streaming libraries that isn't present in FP streaming libraries is data-level parallelism control. Since the programmer isn't manually describing the dataflow topology, the

library needs to know where it can safely introduce parallelism and distribute computation across multiple nodes.

Safety is important here. Partitioning streaming computations onto distributed nodes does not in general preserve the semantics of the source program and can introduce undesirable nondeterminism [88, 126, 165] due to network latency and the reordering of events that can happen when splitting and joining substreams.

The usual abstraction for this is "keyed streams". In addition to a usual stream type `Stream a`, distributed streaming libraries include a type like `KeyStream k a`. Values of this type are k-many logically independent streams of `a`s, which the underlying system can decide to process in parallel by sending some of the sub-streams them to different nodes.

Programmers expose this parallelism opportunity with some kind of `groupBy` operation, which takes a key-selection function `a -> k`, and turns a `Stream a` into a `KeyStream k a`. Depending on the use case, the key type and key-selection function can be lots of different things. A common choice k = `Int`, with the partitioning function being some sort of hash of a data field mod `n`, or perhaps just projecting out some kind of ID. Then, the library APIs also expose the usual streaming combinators to operate on `KeyStream`s, with per-key substreams being processed logically independently, and potentially physically in parallel.

Then, libraries provide operations to join `KeyStream`s back into `Stream`s. In cases where the programmer cares about determinism, this is usually done with an AC-reduce operation [52]. The associative (A) property means that it doesn't matter the order that the sub-streams are reduced toether, while the commutative (C) property ensures that reorderings due to network latency cannot affect the result. In contexts where determinism isn't required, `KeyStream`s can be turned into `Stream`s by simply unionining the sub-streams together and forgetting the key.

This parallel treatment of streams eschews the traditional view of streams as sequences of data. In light of this, recent work has updated the foundations of streaming to incorporate the parallel perspective, reframing streams as *partially ordered multisets* (or pomsets) [12, 12, 101–103, 126]. Inspired by work in concurrency theory [57, 128]. Data items in a pomset may be completely ordered (a sequence), completely unordered (a bag), or somewhere in between. Some recent works have also proposed pomset-based and structured monoid-based types for streams [12, 125, 126].

Time

One way that the distributed implementation of streaming programs can leak into the normal programming model is the treatment of time. Some operations in a streaming language might make decisions based on time; the most common instance of this is time-based windowing, where windows are created based on the arrival time of events. Of course, in a distributed setting, "time" is a finicky concept [115]: one must contend with different nodes processing data at different rates and also with network-induced delays.

There are two regular ways of handling time in a streaming system, and so there are often two versions of any function that uses time in a stream processing system. The first is processing time, where the current system's clock is used to determine the current time. This has the benefit of being montonic and easy to determine, but it has the downside of inducing nondeterminism in many cases: network latency can change the semantics of a program that depends on processing time. The second is event time, where time is a property of individual stream events, usually determined by a field on the event with a timestamp. This has the benefit of allowing for deterministic processing, but the drawback that timestamps might not monotonically increase since events can sometimes be reordered within a stream (maybe due to parallelism).

The work of mitigating the drawbacks of these two choices is mostly a systems-level problem, but programmers writing distributed stream programs must be aware of the two options, and choose the correct one for their use case.

2.2.4 Stream Programming in Databases

Another domain where stream processing is used is databases. As opposed to running queries over inert data—usually referred to as "batch" queries—one might want to run database queries over streaming sources. The underlying operational model of streaming databases is very similar to stream processing for data analytics or distributed streaming, with query plans for stream queries often being distributed. The primary difference is the programming model, which naturally takes a more database flavor, usually derived from traditional query-processing languages like SQL.

In the database literature, instead of viewing streams as sequences of events, streams are often viewed

as entire relations (sets of tuples) that change over time. A time-varying relation can be viewed as either a function from timestamps to finite relations or an infinite set of timestamped values. Streaming database programming is thus query programming over a time-indexed family of databases instead of a single database. Research on streaming database programming then focuses on (1) developing query languages that also expose stream-specific operations, and (2) efficiently computing incremental changes to queries in response to changes in the underlying database.

Along the first line of work, streaming databases have mostly settled (unsurprisingly) on stream-capable variants of SQL. This was initiated by early streaming query languages like CQL [14, 16, 96], and has remained popular with subsequent streaming databse systems, including Aurora [2] and Borealis [3], TelegraphCQ [41] and CACQ [124], and STREAM [15]. Usually, these languages are standard SQL, with stream-to-relation operations (such as windowing) to turn a time-varying relation into an instantaneous relation, and relation-to-stream operations to turn an instantaneous relation into a time-varying one.

The second line of work focuses on "incremental view maintenance": efficiently computing query results as the underlying data changes [83, 84, 194]. Rather than recomputing an entire query result from scratch when the database changes, incremental view maintenance techniques compute only the changes to the query result. View maintenance is actually used in non-streaming applications databases as well (since all databases change over time), but it is especially important in streaming. Recently, DBSP [36] and Differential Dataflow [131] have been proposed as particularly exciting solutions to incremental view maintenance, with the latter powering the Materialize streaming database [130].

2.2.5 Stream Programming in Embedded Systems and Hardware Accelerators

Stream processing also appears in the contexts of embedded systems and hardware, where the resource constraints component of stream processing takes on a significantly more serious role. In these domains, bounded memory usage for streaming programs is not merely desirable but often a hard requirement. For instance, many embedded systems require all memory to be statically allocated [161]. Similarly, hardware designs have strict silicon area or lookup table (LUT) constraints. Stream programs that run on embedded or hardware systems also often operate in real time, with strict timing requirements.

Stream programming in these domains also takes place on a variety of specialized hardware platforms,

from microcontrollers [159, 193], to chips specialized for digital signal processing [118, 188], to FPGAs, to hardware accelerators like GPUs or Machine Learning chips [7]. These unsual hardware environments require specialized stream processing languages with compile-time guarantees for resource usages and execution timings [91].

The uses for embedded or hardware streaming are also quite different from functional or distributed stream processing. Some selected examples are digital signal processing, IoT devices [179], control systems [10], machine learning acceleration [7], and network device programming [29].

Embedded Systems

Embedded systems programming is its own world, with plenty of non-streaming tasks and abstractions. Indeed, many embedded systems are programmed directly in C or assembly. However, some embedded systems tasks — such as interfacing with and reading from sensors or peripherals — do look like stream processing. Of the high-level programming models available to embedded systems programmers, dataflow languages are the most prominent option that has streaming semantics. As discussed in the earlier section, dataflow languages have as their semantics dataflow graphs. Notably, most of these languages *do not* adopt a functional streaming syntax. They usually look more like imperative array-processing code or simply declarative mathematical operations, usually with some amount of timing control involved to describe when operations should happen. Some examples of dataflow languages are Lustre [39], Esterel [26], Signal [119], Zelus [31], and Scade [43]. These languages are commonly used to build safety-critical systems such as aircraft or automotive control systems [77, 85]. Because of these applications are so safety-critical, some dataflow languages even offer built-in formal verification support [30, 86].

Relatedly, some embedded applications simply seek to monitor streams of sensor data, and alert when complex conditions are matched. For these tasks, programmers can reach for stream runtime verification languages, which provide high-level declarative or specification-based interfaces for writing monitors. Examples include LOLA [50], HLola [40], RTLola [64], Striver [81], HStriver [80].

Hardware Accelerators and FPGAs

In recent years, the end of Moore's Law [120] has lead to some compute-intensive applications being offloaded to domain-specific hardware accelerators and even custom hardware designs on FPGAs. Many of these tasks — such as machine learning, networking, and digital signal processing — have a stream processing component. To this end, languages for programming hardware accelerators and designing custom hardware expose stream programming-like interfaces. Accelerator and HDL programming has many of the same constraints as embedded systems programming: resources are limited and must be used carefully, timeframes are short, and deadlines cannot be missed.

Hardware accelerators, when not programmed directly in some sort of assembly, are often programmed with dataflow languages similar to those used in embedded systems. Halide [150] is the prototypical example in this space, featuring a dataflow sublanguage to define mathematical operations, and a scheduling sublanguage to map work onto hardware resources. Spatial and Aetherling [58] are related languages for programming accelerators. Aetherling actualy adopts an API closer the standard functional streaming API, though it's primarily designed for digital signal processing applications.

Hardware designs for FPGAs operate at an even lower level, describing both the computation to be performed and the hardware that will execute it. Standard hardware description languages (HDLs) like Verilog and VHDL include abstractions (Always and Process blocks, respectively) for defining state machines, that can be conceptualized as defining stream processing programs. Because streaming at the lowest levels requires thinking about byte-exchange level details such as handshaking and packet/frame headers, HDL programmers use libraries ("IP", in hardware parlance) such as AXI-Stream and Avalon Streaming [17, 94] to abstract over the nitty details of establishing and maintaining stream connections on a layer-1 interface.

Newer high-level HDLs are also emerging. Filament [137] and Dahlia [136] are declarative dataflow languages with sophisticated type systems for enforcing safe pipelining and resource usage, respectively. Meanwhile, Clash [19] and Lava [28] represent HDLs embedded in Haskell — Lava adopts a more dataflow-like approach, while Clash is actually a proper functional programming-style steram combinator DSL that compiles to hardware.

2.3 Fusion and Streams

Fusion, also known as deforestation [185] is an important class of interprocedural compiler optimizations. The general idea to transform a composition of functions $g(f(x))$ into a single function h that does the same thing, while either (a) using less space by not materializing the intermediate result of $f(x)$, or (b) operating more efficiently in some way. The canonical example of this in functional programing is map fusion, transforming `map g (map f xs)` into `map (g . f) xs`. Assuming f and g are pure, this has the benefits of (a) not materializing the intermediate list, and (b) only taking one pass over the input.

Effective and efficient fusion is an ongoing research project the functional programming research community. One foundational (though uncommonly-used) approach is supercompilation [181] which attempts essentially full program specialization. Another more recent approach uses a sophisicated type system to expose and exploit fusion opportunities [42]. However, most approaches to fusion in functional language compilers simplify the problem by attempting to fuse programs only by specializing compositions of common built-in combinators. Most famous among such approaches is a pair of essentially dual techniques, short-cut fusion [75] fusion and stream fusion [47]. The former transforms functions into a composition of `unfold`s and `fold`s to attempt to fuse them together, essentially turning list programs into *push* stream programs. This approach is employed inside GHC to fuse list functions. The latter transforms list functions into corresponding *pull* stream functions, which fuse "automatically".

Beyond these approaches, many modern fusion passes employ ad-hoc systems based on rewriting. In these systems, compiler authors carefully develop lists of equations that can be applied to transform less efficient programs into more efficient fused ones: for example `map g (map f xs) = map (g . f) xs`. Then, the system searches over a user program, finding possible matches for the LHS, and transforming them into the RHS. Recently, the more principled approach of equality saturation [174] has become a popular way of systematicaly exploring the space of rewriting results [135, 189].

These rewriting and pattern matching approaches are especially popular in high-performance computing compilers, where fusion is critical to avoid materializing large matrices. Array languages like Futhark [87] and machine learning frameworks like Jax [70], Tensorflow [4], and PyTorch [142] all use pattern-matching and rewriting fuson to optimize sequences of matrix operations into optimized ker-

nels [49, 170].

Fusion and streaming would seem to be inexerably tied together. Indeed, many of the above-mentioned techniques for fusing functional code work by transforming (arbitrarily sized) intermediate data structures into streams of (constant-sized) values, and processing them incrementally instead of in batch. Conversely, fusion is critical for streaming, since intermediate allocations have the potential to nulify the resource usage guarantees that many streaming libraries attempt to provide. As such, many fusion techniques are directly applicable to the problem of fusing streaming programs, and often transfer directly.

2.3.1 Fusion with Metaprogramming

Most important to this document is metaprogramming-based or staging-based fusion. Staging-based fusion takes a different approach from rewriting: rather than attempting to fuse together some programs after the fact, they instead focus on interpreting code as *metaprograms* which directly generate fused code. This approach is exemplified by metaprogramming community's emphasis on "DSLs without regret" [33, 160], where library combinators are designed so that their compositions automatically code-generate efficient code that does not allocate any intermediates. A particularly influential example, based on the stream fusion technique of Coutts et al. [47], is *Stream Fusion to Completeness* [105]. This paper uses typed metaprogramming to transform compositions of pull stream combinators into fused imperative streaming programs. We draw heavily on these techniques in Chapter 6 for our own compilation strategy for λ^{γ}.

2.4 Substructural and Ordered Type Systems

Most of the area of type system design deals with adding complex typing rules that add expressivness to type systems in order to rule out bad behavior. Substructural type systems take a somewhat different tack, instead *removing* rules to rule out bad behavior!

Substructural type systems are so named for the three structural rules of type systems (Figure 2.1). Unlike most of the rules in a type system, which govern the behavior of different type constructors, the structural rules are purely focused on manipulating the typing context. Read bottom-to-top, the rules say

$$\frac{\Gamma_1 \vdash e : A}{\Gamma_1, \Gamma_2 \vdash e : A}\ \textsc{Weakening} \qquad \frac{\Gamma, \Gamma \vdash e : A}{\Gamma \vdash e : A}\ \textsc{Contraction} \qquad \frac{\Gamma_2, \Gamma_1 \vdash e : A}{\Gamma_1, \Gamma_2 \vdash e : A}\ \textsc{Exchange}$$

Figure 2.1: The Structural Rules

the following:

- The Weakening rule says that a subcontext of variables can be thrown away. This means that you don't always have to use all of your variables, and any of them can be "dropped".
- The Contraction rule says that a subcontext of variables can be duplicated. This means that you can use variables as many times as you wish.
- The Exchange rule says that you can always permute the order of subcontexts. This means that variables may be listed in any order.

Taken together, these three rules say that the context is essentially a bag of variables. This is how most simple (as opposed to dependent) type systems behave. Indeed, in many papers on type systems, this context behavior is assumed, and so these rules are simply implicit in the presentation.

A type system is *substructural* if it forgoes one or more of these rules, restricting the way that variables can be used. Substructural type systems are derived from susbtructural *logic*, the study of the logical systems that emerge when you play the same trick (removing structural rules) on a logical system of inference [154].

Working in a substructural system that uses only a subset of these rules changes not only how variables can be used, but also how the product type $A \times B$ behaves in the type system. For example, in a system without Contraction, there is no term witnessing $A \to A \times A$, while in a system without Weakening, the product does not have projections $A \times B \to A$ and $A \times B \to B$. As we will see in Chapters 4 and 5, we can use this connection to our advantage in the reverse direction. If we know what equations we want to hold of the product type, we can choose the subset of structural rules to match.

2.4.1 The Substructural Zoo

The most famous susbtructural type system is linear types, based on Girard's linear logic [76]. In linear types, you drop Weakening and Contraction, but keep Exchange. With this combination of rules, every variable must be used exactly once. Wadler introduced linear types to programming languages—already well-understood in the logic world— with the dictum "Linear Types can Change the World!" [186]. With this, he was referring to the fact that linear types enable in-place updates without aliasing concerns, since the type system guarantees exclusive access to linear values[6].

Moreover, linear types can be used to model resources that must be "consumed" exactly once, like file handles, network connections, or memory allocations. By statically enforcing that these are consumed exactly once, the type system can prevent resource leaks and use-after-free errors [25, 62, 177].

Closely related to linear types are *affine* types, which drop Contraction but keeps both Weakening and Exchange. This means that variables may be used *at most* once—they can be dropped, but not duplicated. Affine types enforce many of the same safety properties as linear types, i.e. preventing use-after-free errors, weakening means that resources may be leaked.

2.4.2 Bunched Type Systems

A close cousin of substructural types are *bunched* types. Owing to the logic of bunched implication (BI) [141] that underlies separation logic [95, 155], bunched type systems generalize the structure of contexts from lists to trees. While normal contexts are generated by the grammar $\Gamma := \cdot \mid x : s \mid \Gamma, \Gamma$, bunched contexts add another context former to get $\Gamma := \cdot \mid x : s \mid \Gamma, \Gamma \mid \Gamma; \Gamma$. This generates a logic where there are naturally two product types: one that corresponds to the comma context former (i.e. the usual product), and a new one that corresponds to the semicolon context former (a new product).

In the original formulation of BI, the semicolon connective is fully structural (has all three structural rules), while the comma connective is linear. This, in turn, generates two products, one fully structural and the other linear. However, it need not be this way: one could choose different structural rules for either or both of the context formers, and get different products out.

[6]Though much of the early literature on linear types conflates the two, linearity and uniquness are not the same. This is discussed in detail by Marshall et al. [127]

2.4.3 Ordered Type Systems

Type systems with (at least one of the context conectives having) Weakening, but no Contraction or Exchange are of primary interest to this dissertation. In such a type system, you must use each variable at most once, *in the order* they appear in the context (we'll choose left-to-right). Compared to the other substructural type systems described above, ordered types are seldom studied in the programming languages literature. Indeed, the most famous use of ordered "types" is in proof-theoretic form with the Lambek calculus [114]. However, some research in programming languages has explored ordered types for language design and verificaiton.

Concurrent with the Stream Types project, work by Saffrich et al. [163] uses ordered types as an alternate encoding of typestate [173], ensuring that resources like file handles are first opened and then closed. The type system they propose is remarkably similar to the one we develop in Chapter 3, using a bunched context with only one of the context formers ordered. Older work has also used ordered types for modeling concurrency [56] and for logic programming [148]. The most closely related work to ours is undoubtedly that of Kodama et al. [107], which employs ordered types for a sort of stream processing! Their system uses an ordered type system to translate XML-processing code—which pattern-matches against and extracts fields from static XML documents—into stream-processing code that incrementally transforms documents represented as streams of tags. This is real a conceptual parallel to our work.

Chapter 3

Delta: Functional Programming with Streaming Semantics

> The expressive power of a programming language derives from its strictures, and not from its affordances.
>
> — Robert Harper

As evidenced by the enormous body of related work presented in the previous chapter, stream processing is extremely widely practiced across computing. However, there are few programming models for actually building stream processing programs. To a first approximation (aside from approaches specific to databases), either (a) one builds streaming state machines manually, defining stateful functions to accept and process events, or (b) one uses a functional-style stream processing language or library, and builds their program out of pre-defined high-level streaming primitives like `map` or `filter`.

Both of these approaches have downsides. Thinking in state machines is too low of an abstraction level for most tasks, where programmers simply want to define transformation on sequences and have it run incrementally. This issue is fixed by instead using a streaming library, which allows programmers to operate at a much higher level of abstraction. While this works for many tasks, the common streaming primitives are occasionally *too high* of a level of abstraction programmers must take the idea for a sequence program they have in their head, and cram it down into the set of combinators. In the limit, this often looks like writing everything in terms of `fold`, which is (in essence) just like writing state machines. To illustrate this phenomenon, where streaming libraries require programmers to contort themselves to express simple programs, we consider the following simple example:

A Nasty Example Suppose we had a stream of integers S, representing (say) the brightness of some light sensor as a number between 0 and 100, reported once per second. Our goal is to transform this to a stream of numbers that are averages of elements of S during runs when it went above some fixed threshold

k. For example, for $k = 50$, an instance of this transformation might look like the following.

$$11, 30, 53, 56, 53, 30, 10, 60, 10, \ldots \longrightarrow 54, 60, \ldots$$

Importantly, this function runs incrementally, i.e. the `54` is produced in the output as soon as the "run" ends, and the input drops below 50 (when the 30 arrives).

How would we write this program in a functional stream combinator library of the form discussed in Chapter 2? At first glance, this may seem like a straightforward use of `filter`: we filter out elements below k, and then compute averages. But filtering the stream "forgets" where one run ends and the next begins, preventing us from computing averages. Another idea is to use a `window`, using some kind of average as our windowed operation. Unfortunately, this cannot be accomplished with windowed operations that use standard window-creation strategies[7], since the runs are of an unknown and dynamically-determined length.

Unfortunately, the best option is to `fold` over the stream holding onto a state containing an option which, if a run is active, contains (a) the length of the current run and (b) the total. Each time a run ends, we output the average of the run. In Apache Flink [38], this takes the form of the code below.

```
xs.flatMapWithState((x : Int, st : Option[Int,Int]) =>
  st match {
    case None => if x > k then ([],Some(1,x)) else ([],None)
    case Some(num, tot) =>
      if x > k then
        ([],Some(num + 1, tot + x))
      else
        ([(x + tot) / (num + 1)], None)
  }
)
```

While beauty is in the eye of the beholder, I personally find this program objectionable on aesthetic grounds, and I feel unhappy to have written it. More importantly, this monolithic fold solution is poor software engineering practice. It mixes concerns, lacks clarity, and is not composed of reusable parts: The

[7] This window-based approach can work if the language supports advanced data-dependent windowed operations, as has been proposed by some prior work [82]. Indeed, this light-levels example was developed from a use-case in this paper.

logic of collecting runs above k is intermixed with emitting an average once the run ends. This kind of contortion and mixing of concerns is common to code written in the combinator style.

Indeed, I am far from the first to notice that writing sequence (or streaming) code in purely terms of list combinators is bad for reading and writing alike. Recent research in PL/HCI has shown that functional programmers are significantly more productive when given the full expressive power of functional list code, compared to when they are restricted to combinators [123]. All of this begs the question...

Is There Another Way? Stepping back, if you have random access to the stream (i.e. it is an in-memory list instead), writing this operation as an idiomatic functional list program is straightforward:

```
runs k [] = []
runs k (x:xs) = if x <= k then runs k xs
                else let (run,rest) = span (> k) yx in
                     (x:run):(runs rest)

averageSingle run =
  let n = length run in
  let x = sum run in
  x / n

averageRuns k xs = map averageSingle (runs k xs)
```

The solution `averageRuns` first calls `runs` to compute a list of lists, one sublist for each run. Then, it maps `averageSingle` over that list of runs, which computes the length and sum of each run, and then divides them. The `runs` function is the most intricate. It walks the input list, looking for the first element above `k`. Once it does, it calls `span (> k)` on `xs` to compute `run` — the largest prefix of `xs` whose elements are all above `k` — and `rest` — the remainder of the list after `run`. Finally, it returns the first run `x:run`, cons'd onto the recursive call `runs k rest` to construct the sequence of runs from the remainder of the input.

This solution is interesting for a few reasons. First, it requires nested lists. In a language with only lists (or streams) of base types, this solution would not be implementable. Next, it uses combinators like `map`, `length`, `sum`, and `span`, but it is not restricted to them. In particular, `runs` makes critical use of pattern matching, as well as `nil` and `cons`. The combinators are also applied at non-base types: `map` is used to map over a list of lists, not a list of base types. Last, the `runs` function makes use of nonstructural general recursion by recursing on the `rest` of the list.

Of course, none of this would seem to have anything to do with stream processing. Enter: Delta. Delta is a prototype programming language with the syntax of standard functional list programming — providing the constructors nil and cons, pattern matching with `case`, and general recursion—but the semantics of a stream processing language. In other words, we can write normal list programs, and have them run incrementally. Of course, not all list programs have well-behaved incremental interpretation: how would one stream the `reverse` program? The insight of Delta is that using an *ordered* and *affine* substructural type system is a sufficient condition. Orderedness guarantees that variables are used in the same order that their values will arrive, while affinity ensures that values need not be stored in memory between uses. We explore all this and more in the next section, where we take a deep dive into Delta's syntax, types, and semantics. The implementation of Delta can be found at https://github.com/alpha-convert/delta.

3.1 Delta Tutorial and Examples

Our first example of a Delta program—which adds one to each element in a stream of `Int`s— is shown below. is shown below:

```
fun addone(xs : Int*) : Int* =
  case xs of
  | nil => nil
  | x::ys => (x + 1) :: addone(ys)
```

Delta programs are defined as a sequence of top-level functions using OCaml-style syntax. The type of streams of `Int`s in Delta is written `Int*`, inspired by regular expressions. Syntactically, values of this type are constructed with `nil` and `cons`, and destructed with `case`. Semantically, when the first integer n_0 of the incoming stream `xs` arrives at `addone`, the number n_0 gets bound to `x`, gets incremented, and then is *immediately* forwarded along into the output stream. When the next element n_1 arrives, this process repeats, with n_1 bound to the head of `ys`. When, then, does the `nil` case of the match run? Streams of type `Int*` aren't just a stream of ints, they also include an "end of stream" marker. If at any point the `addone` function receives an end of stream marker on `xs`, it will take the `nil` branch of the `match`. Since the `nil` branch also returns `nil`, it will send the "end of stream" marker into its output.

Below is a slightly more complicated example, which takes a stream of ints and produces the stream

of odd-indexed elements of the input, starting from zero.

```
fun odds(xs : Int*) : Int* =
  case xs of
  | nil => nil
  | _::ys => case ys of
             | nil => nil
             | z :: zs => z :: odds(zs)
```

When the first element of `xs` arrives, it is dropped, since the head is bound to a wildcard pattern `_`. When the next element arrives, it is bound to `z`, which is immediately forwarded along into the output. The program then makes a recursive call on the rest of the stream `zs`, which has the effect of jumping back to the start.

3.1.1 Delta's State Machine Semantics

Note that the semantics of this Delta program implicitly define a state machine with three states: (1) waiting on the first branch, (2) waiting on the second branch, (3) done. However, the programmer does not have to explicitly define the states or event handlers of this machine. In fact, all Delta programs are state machines, which gives them their incremental semantics. A good mental model of a delta program's state machine semantics follows; we make it formal with an operational semantics in Chapter 4.

When an event arrives on the input stream, we now have a concrete value for a prefix of the input variable to the function. I.e. in the `addone` example, when the first `Int` v arrives, the data required to run (1) the outer case analysis, and (2) the first argument of the cons (`::`) are completely known. As such, we can simply run this prefix, emit any results onto the output stream, and then step to a new state including only the *remainder* of the program. In the `addone` case, this remainder term is just the recursive call `addone(ys)`. Then, when we receive more input, we repeat the same process on the remainder.

Slightly more precisely: for a Delta stream transformer defined by some code $\mathtt{e}_0$ with input stream xs, its state machine interpreation is given by a machine with initial state $\mathtt{e}_0$—the code for the term itself. The state transition function is given by the following procedure. When some event arrives v arrives on xs:

1. Get the current state e.

2. Compute e', the largest closed prefix of the term e that is concrete, given the input event v

3. Evaluate e', and send its results (if any).

4. Compute e'', the remainder of e after e', and set this as the new state.

This style of formal semantics where terms define state machines is unusual but not actually novel, having been pioneered by the Esterel language [26].

3.1.2 The Ordredness Checker

Delta's state machine semantics imposes an implicit requirement on the structure of programs that are not met by all list programs. In particular, it is important that "e', the largest closed prefix of the term e that is concrete" exists! This puts a burden on the typechecker, which must then statically enforce that the list program the user has written can be implemented as a streaming program. It turns out that a substructural *ordered* type discipline is the right constraint to impose here.

To illustrate, consider the following program, which swaps adjacent pairs of elements in a stream:

```
fun swap2(xs : Int*) : Int* =
  case xs of
  | nil => nil
  | y::ys => case ys of
             | nil => nil
             | z :: zs => z :: y :: swap2(zs)
```

This program cannot be executed by Delta's state machine semantics: when the first element of `xs` arrives (bound to y), its *first* usage (on the right-hand side of the `::`) comes *after* the first element `z` of the tail `ys = z::zs`. The prefix of the program that uses `y` is "blocked" behind a use of `z`.

To prevent this, Delta's type system includes an orderedness check: variables must be used in the same order—in the sense of being to the left/right of `::` or other sequential operations— as the data that gets bound to them arrives. The head of a list arrives before its tail, and also before any of the elements of the tail, and so on. As we will see, destructing other types also induces orderings on the bound variables.

Delta also imposes an *affineness* restriction, which means that variables can only be used once. This restriction exists to simplify the semantics of the language, and is not a practical programming impediment.

Variables can explicitly be used multiple times using the `wait` construct, which we discuss in Section 3.1.5.

3.1.3 Rich Stream Types

`Int*` is not the only type in Delta. In fact, Delta's types are extremely rich, and can define much more interesting shapes of streams. One example of a stream type is just the type `Int`, sans star. A stream of type `Int` is just a "singleton" stream, on which there is exactly one value of type `Int`. To receive a stream of type `Int` is to expect to receive exactly one `Int`. To produce a stream of type `Int` is to be obligated to produce exactly one `Int`. Relatedly, the unit stream type `1` is a singleton stream containing a single value of the unit type. Streams of type `Int` and `1` are introduced in Delta with numerical literals and `()`, respectively. `Int`s are eliminated with arithmetic or comparison.

There is also an empty stream type called epsilon, written `Eps`. There is exactly one stream of this type, and it contains no events. To produce a stream of type `Eps`, one simply does nothing, and when accepting an input of type `Eps`, one expects to receive nothing. Note that this is not a bottom or void type — it is always possible to produce a stream of type `Eps` with the term `eps` (which sends nothing), and you cannot do anything special (such as derive absurdity) if given a variable of type `Eps`.

For any stream type `s`, we can form the type `s*`. A stream of this type is zero or more streams of type `s`, one after another. The values of type `s*` also includes *punctuation* elements [180] after each sub-stream of type `s`, to tell when one ends and the next begins. In this way, a stream of type (for example) `Int**` is distinguishable from one of type `Int*`. Additionally, there is a final "done" punctuation mark after the last `s`. A stream of type `s*` will always contain finitely many sub-streams of type `s`. However, consumers of streams of type `s*` do not get to know how many `s`s they will receive, and so from their perspective the stream is unbounded. Dually, a producer of a stream of type `s*` does not at any point have to commit to how many `s`s it will send in the future; if it so choses, it can continue to send more as long as it wishes.

Next is the concatenation type. Given stream types `s` and `t`, we can form the type `s.t`. Streams of this type look like a stream of type `s` followed by a stream of type `t`, with a punctuation mark separating them to eliminate any possible confusion. Streams of type `s.t` are introduced with `(e1;e2)` — which first runs `e1` and then runs `e2` to produce their outputs in order — and eliminated with `let (x;y) = e in e'` — where data will first be bound to `x` when it arrives, and then bound to `y` when the first component

completes. These terms are both subject to orderedness and affineness restrictions. Because the `e1` in `(e1;e2)` will run first, its variables must have their data arrive first. Similarly, the `x` bound in `let (x;y) = e in e'` must be used before the `y`.

Last, we have sum types. For types `s` and `t`, there is a sum type `s + t`. A stream of this type is either a stream of type `s` or a stream of type `t`, preceeded by a boolean that says which of the two is about to take place. Sums are introduced with `inl(e)` and `inr(e)` — which first sends the corresponding boolean and then runs `e` — and eliminated with a case analysis just like that for star streams.

While extremely elegant, these types are not merely a theoretical curiosity. Delta's rich language of stream types are actually a large part of what gives it its expressive power, and is a major difference between Delta and existing streaming languages[8]. For an example of how Delta programs can be written compositionally thanks to the expressive stream types, see Section 3.1.10

Singletons and Head

In most streaming languages, all streams can have unbounded length. But in many practical situations, a stream will only be expected to contain a single element; a constraint that cannot be expressed with homogeneous streams. Using stream types, we can write stream transformers that are statically known to only produce a single output. For example, the "head" function is trivially expressible in the same manner as head on lists, as shown on the right. If the stream is empty, we return `inr(eps)`. If there is a head, we return it and discard the tail.

```
fun head [s] (xs : s*) : s + Eps =
  case xs of
    nil => inr(eps)
  | y :: _ => inl(y)
```

3.1.4 Writing Combinators in Delta

Of course, the standard list functions like `map`, `filter`, and `fold` are, on the whole, extremely beneficial. They lift the level of abstraction of stream programming up to an acceptable level (much higher than mere

[8]The expressive power of these types (as opposed to "streaming semantics with list syntax") was the original framing of this whole line of work, as presented in the PLDI'24 Stream Types paper.

state machines), and make it feasible to do stream programming "in the large". It would be a shame to have a stream programming language without them. Luckily, all of the common streaming functions are *implementable* from the primitives Delta provides. In fact, most of them are written essentially identically to their list-programming equivalents.

Functions as Macros in Delta

Since most of these streaming combinators are higher order functions, we must discuss how Delta handles higher-order functions. Support for higher order functions in Delta is limited, but they can be implemented using *macros*. A function written as `fun g<f : Int -> Int>(x : Int*) : Int* = e` is a macro which takes another function `f : Int -> Int` as a parameter. Calls to `g` in other functions then look like `g<f'>`, where `f'` is either (a) another function defined at top level, or (b) a call to yet another macro. If the macro `g` is recursive, its recursive calls do not receive a macro argument—all recursive usages of a macro get passed the initial macro parameter `f`. This discipline ensures that the macro usage does not depend on runtime data, and so higher-order functions can be fully resolved to λ^{ST} terms statically. Neither of these features—standard top-level functions and higher-order macros—require the use of first-class function types, which Delta does not currently support. Defining true higher-order functions would allow for streams of *functions*, such as `(s -> t)*`.

Functions in Delta can also be (prenex-) polymorphic [134]. Polymorphic functions definitions are annotated with an list of their type arguments, like `fun f[s,t](x : s*) : t* = e`. When such a function is called, the type arguments must be passed explicitly like `f[Int,Bool]`.

Map

Given a transformer from `s` to `t`, we can lift it to a transformer from `s*` to `t*` with a `map` operation. The code for this function is essentially identical to the familiar functional program, but it runs incrementally, applying `f` to each element in turn as it arrives.

```
fun map [s,t] <f : s -> t> (xs : s*) : t* =
  case xs of
    nil => nil
```

```
  | y :: ys => f(y) :: map(ys)
```

Note that the type of `map` is also more general than the standard `map` function from streaming libraries, which has type `(a → b) → (Stream a → Stream b)`. The types s and t here can be arbitrary Delta types, not just base types. For example, picking `s, t = Int*` and instantiating `f` with `addone`, we get `map[Int*,Int*]<addone>`, which is a streaming transformer of type `Int** -> Int**`.

FilterMap

```
fun filterMap[s,t]<f : s -> t + Eps> (xs : s*) : t*=
  case xs of
    nil => nil
  | y :: ys => case f y of
               | inl(t) => t :: filterMap(ys)
               | inr(_) => filterMap(ys)
```

Similarly, given a "predicate" function f from s to $t + \varepsilon$ (the streaming version of t `option`), we can transform an incoming stream of `s*` to include just the transformed elements which pass the filter.

Let-Binding in Delta

One might think to rewrite `filterMap` like this, let-binding the call to `filterMap(ys)`, and hoisting the binding out of the `case`.

```
fun filterMap[s,t]<f : s -> t + Eps> (xs : s*) : t*=
  case xs of
    nil => nil
  | y :: ys =>
    let zs = filterMap(ys) in
      case f y of
      | inl(t) => t :: zs
      | inr(_) => zs
```

Is this an order-checker violation in Delta? After all, the let-binding uses `ys` and it appears syntactically before the use of `y`. In fact, this program typechecks in Delta, and executes identically to the previous

version. This program is not an order-checker violation because the bound variable `zs` is used in the two branches of the `case` *after* `y` is used, so the usage of `ys` happens temporally after the usage of `ys`. More formally, the RHS of a let-binding does not run until the data it depends on data arrives. One should think about `let` in Delta approximately as they think about `let` in Haskell: the binding is evaluated as needed. For formal details about how let-binding interacts with orderedness, see Chatper 4.

3.1.5 Using State

All of the programs so far have been stateless, as enforced by Delta's ordered and affine type system. But sometimes you really do need to use state! Delta includes a construct called *waiting* to explicitly pull data out of an incoming stream and save it in memory. If `x` is a variable, we can write `wait x do e end` to buffer the entire stream bound to `x` into state. Once `x` is complete, this continues the streaming computation `e`, which then has random access to `x` and may use it as many times as it wishes, in any order. If `e` is a streaming expression, we can also write `wait e as x do e' end` as sugar for `let x = e in wait x do e' end`. Of course, `wait` must be used with care. If the type of `x` includes substreams of type `s*`, this operation may require potentially unbounded memory[9].

We refer to variables that have been `wait`'d on as "historical variables", as they contain data that arrived in the past. We can refer to historical variables using historical programs, written `{M}`. Here, `M` is a program in the "historical sublanguage", essentially a by-value functional language that can express arbitrary non-streaming combinations, and has access to historical variables. Once you've saved data into memory, you can do anything with it that you like, without being subject to Delta's restrictions.

To demonstrate the use of `wait` and historical programs, we show how they can be used to write the `stutter2` and `swap2` disallowed by the orderedness checker.

```
fun stutter2[s](xs : s*) : (s . s)* =
  case xs of
  | nil => nil
  | y::ys => (wait y do ({y};{y}) end) :: stutter2(ys)

fun swap2[s](xs : s*) : s* =
  case xs of
```

[9] We return to this concern in Chapter 5, when we discuss Yoink and λ^{y}

```
  | nil => nil
  | y::ys =>
    wait y do
      case ys of
      | nil => nil
      | z::zs => z::{y}::swap2(zs)
    end
```

Folds

Delta can express both running folds (or scans), which output a stream of all their intermediate states, and functional folds, which output only the final state.

```
fun fold [s,t] <f : {t}(s) -> t>{acc : t}(xs : s*) : t =
  case xs of
    nil => {acc}
  | y :: ys => wait f{acc}(y) as acc' do
                 fold{acc'}(ys)
               end
```

The (functional) `fold` transducer maintains an in-memory accumulator of type t; this gets updated by a streaming step function `f : {t}(s) -> t` that takes the state t and the new element s and produces a t. The whole fold takes a stream `xs` of type `s*` and an initial accumulator value $acc : t$, and it eventually produces the final state t. Folds that return only the final state cannot be given this rich type in traditional stream processing languages (for the same reason as the `head` and `tail` functions). As for `map`, the code for `fold` is very similar to the traditional functional program: the only distinction is the inclusion of `waits` to marshal data into memory.

We can also define a running fold, which outputs its partial results as it goes.

```
fun runningFold[s,t]<f : {t}(s) -> t>{acc : t} (xs : s*) : t* =
  case xs of
    nil => nil
  | y :: ys => wait f{acc}(y) as acc' do
                 {acc'} :: runningFold{acc'}(ys)
               end
```

As discussed in Chapter 2, fold is essentially complete for (single-input, push-based) stream programs. In this sense, the definition of `fold` in Delta is a proof that it is as expressive as any other streaming or library based on push streams.

Filter

We can use `mapMaybe` to build a traditional predicate-based `filter` by lifting a predicate `f` of type `{s} → Bool` to a streaming function `s -> s + Eps` with `liftP`. This program simply waits for its argument to arrive, then applies the predicate to the in-memory `s`.

```
fun liftP[s]<f : {s}(Eps) -> Bool>(x : s) : s + Eps =
  wait x, f{x}(eps) as b do
      if {b} then inl({x}) else inr(eps)
  end

fun filter<f : {s}(Eps) -> Bool>(xs : s*) : s* =
  mapMaybe[s,s]<liftPred<f>>(xs)
```

3.1.6 Windowing and Punctuation

Windowing is another core concept in stream processing systems, where aggregation operations like moving averages or sums are defined over "windows"—groupings of consecutive events, gathered together into a set. In Delta, these transformers are just `map`s over a stream whose elements are windows. Given a per-window aggregation transformer `f` from an individual window `s*` to a result type `t`, plus a "windowing strategy" `win` which takes a stream `r*` and turns it into a stream of windows `s**`, we can write the windowed operation as `map<f>(win(xs))`. Delta can express a variety of windowing strategies, including sliding and tumbling size-based window operators, as well as punctuation-based windowing, where windows are delimited by punctuation marks inserted into the stream.

Many kinds of windows have been considered in the literature. The most common windows are event-based — windows defined by the number of elements they'll contain — and time-based — windows that contain all the events from a fixed length of time. Windows can also be tumbling — the next window starts after the previous ends — or sliding — every event could begin a new window.

In Delta, windowed operators are just `map`s over a stream whose elements are windows. Given a per-window stream transformer `f` which takes windows `s*` to a result type `t`, and a "windowing strategy" `win` which takes a stream `r*` and turns it into a stream of windows `(s*)*`, we can write a windowed operation of type `r* -> t*` as follows: `xs : r* |- map(f)(win(xs)) : t*`.

For example, if we wanted to compute a size-3 sliding sum of a stream of `Int`s, we would use a windower `win` which takes `Int*` to `(Int*)*` where the inner streams are the windows, and `f` from `Int*` to `Int` is the sum operation.

Every per-window function commonly used in stream processing practice operates on entire windows at once, which is accomplished in Delta by `wait`-ing on the whole window, and then aggregating it with an embedded historical program. For this reason, we focus primarily on the window construction aspect.

Fixed-Size Tumbling Windows

The k-size tumbling windower creates windows of size k, where each new window starts immediately after the last window ended. For instance when $k = 2$, a stream $1, 2, 4, 7, 3, 8, \ldots$ turns into a stream $\langle 1, 2\rangle, \langle 4, 7\rangle, \langle 3, 8\rangle, \ldots$. The code for a fixed-size tumbling window is exactly the functional code for computing k-strides of a list, by grouping together the first k elements, and recursing down the rest of the stream.

```
fun firstN[s]{n : Int}(xs : s*) : s* . s* =
  case xs of
    nil => (nil;nil)
   | y::ys => if {n > 0} then
                let (predN;rest) = firstN{n-1}(ys) in
                (y::predN;rest)
              else (nil;y::ys)

fun tumble[s]{k : Int}(xs : s*) : s** =
  let (first;rest) = firstN[s]{k}(xs) in first :: tumble{k}(rest)
```

k-size window transformers can actually have the even stronger output type $(\mathtt{s}^k)*$, where $\mathtt{s}^k$ is the k-fold concatenation of `s`. If the window function being used requires that the windows all have exactly size k (like taking pairwise differences for $k = 2$), this type can be used instead. The following program

implements size-2 windows with this stronger type by casing two-deep into the stream at a time, and pairing up elements into concatenation pairs.

```
fun parsepairs[s](xs : s*) : (s . s)* =
  case xs of
    nil => nil
  | y :: ys => case ys of
                 nil => nil
               | z :: zs => (y;z) :: parsepairs(zs)
```

Fixed-Size Sliding Windows

A k-sized sliding windower produces a new window for each new element, including both the new element and the $k-1$ previous ones. The code for this windower keeps the current window under construction in memory. When each new stream element arrives, we emit the current window. For the first k elements, we only add to the window. After k, we start evicting from the window.

```
fun slidingWindower(acc : s*; xs : s*) : s** =
  case xs of
    nil => acc :: nil
  | y::ys => wait y do
               let next = {if |acc| < k then y :: acc else y :: (init acc)} in
               next :: slidingWindower(next;xs)
             end
```

Punctuation-Based Windows

Time-based windows are commonly implemented by way of *punctuation*: unit elements inserted into a stream to authoritatively mark that a period of time has ended. This is required because in the presence of network delays, it's impossible to know if a time period is over (and so a window can be emitted) or if there are more elements in the period to arrive. A punctuated stream has type `(Eps + s)*`, where the punctuation events mark the end of each time period.

The following code computes a windowed stream `s**` from a punctuated stream `(Eps+s)*` by emitting windows which are the (potentially empty) runs of `s` between punctuation marks.

```
fun tilFirstPunc[s](xs : (Eps + s)*) : s* . (Eps + s)* =
  case xs of
    nil => (nil;nil)
  | y::ys => case y of
               inl _ => (nil;ys)
             | inr s => let (cur;rest) = tilFirstPunc(ys) in
                        (s::cur;rest)

fun puncWindow[s](xs : (Eps + s)*) : s** =
  let (run;rest) = tilFirstPunc[s](xs) in
  run :: puncWindow(rest)
```

(Lack Of) Time Based Windows

At present, Delta has no notion of time, and so it has no time-based windows! As will be discussed in the next section, Delta is actually a completely deterministic language. Time-based windows are inherently nondeterministic, as they make processing decisions based on arrival time.

3.1.7 Parallel Type

Delta actually has one last type, the parallel type. Given streams of type `s` and `t`, we can form the stream type `s||t` (pronounced "`s` par `t`"). A stream of this type is a stream of type `s` interleaved with a stream of type `t`, origin-tagged to avoid confusion. This type models *parallel streams*, of the form described in Chapter 2. Producing a stream of type `s||t`, requires independently producing a stream of type `s` and a stream of type `t`, while consuming a stream of type `s||t` requires independently consuming streams of these types.

3.1.8 Partitioning and Routing

A crucial streaming idiom is partitioning, where a single stream of data is split into two or more parallel streams, which are then routed to different downstream nodes in the dataflow graph. The purpose of partitioning is to expose parallelism: the different downstream operators can be run separately, potentially on different physical machines. Depending on the situation, a programmer may choose to use different

partitioning strategies. In Delta, some common partitioning strategies are directly implementable.

Round Robin Partitioning

A round-robin partitioner distributes an incoming stream of type `s*` *fairly* into a parallel pair of streams `s* || s*`. It does this by sending the first element to the left branch, the second to the right, the third to the left, and so on. In Delta, we write this by maintaining a boolean accumulator, and negating after each item. If the boolean is true, we send the element left, if it's false, we send it right.

```
fun roundRobin[s]{b : Bool}(xs : s*) : s* || s* =
  case xs of
    nil => (nil,nil)
  | y::ys =>
      let (zs,ws) = roundRobin{!b}(ys) in
      if b then (y::zs,ws) else (zs,y::ws)
```

Decision-Based Partitioning

A decision-based partitioner routes stream elements based on the result of a predicate.

```
fun decPartition[s,t,r]<f : (s) -> t + r>(xs : s*) : t* || r* =
  case xs of
    nil => (nil,nil)
  | y::ys =>
          let (ts,rs) = decPartition(ys) in
          case f(y) of
               inl t => (t::ts,rs)
             | inr r => (ts,r::rs)
```

3.1.9 Determinisitic Merge

Parallel streams of star type can be *synchronized*, pairing off one element from one stream with one element of another. Given a stream of type `s* || t*`, we can produce a stream of type `(s || t)*`. This type's similarity to the standard functional program `zip` is more than just surface level: the program below has essentially the same code.

```
fun sync[s,t](xs : s*, ys : t*) : (s || t)* =
case xs of
  nil => nil
| x'::xs' => case ys of
               nil => nil
             | y'::ys' => wait x',y' do
                            {(x',y')} :: sync(xs',ys')
                          end
```

Semantically, this program `waits` until a full element from each of the parallel input streams has arrived, sends them both out, and then continues with zipping the two tails. This is necessarily blocking: the output type guarantees that exactly one s and t will be produced before the next pair begins, and so we must wait for both to arrive before sending the other out. The upshot is that because this program is well typed in Delta, it is necessarily deterministic. Moreover, for parallel streams of windows, synchronization enables database-style *streaming joins*. Given parallel streams `(s*)*` and `(t*)*`, we can synchronize to get `(s* || t*)*`, and then apply a join operation to each parallel pair of windows.

3.1.10 The Brightness Levels Example

The core type of the brightness-levels example from the beginning of the chapter can be encoded as the type `(Int . Int*)*`: a stream of nonempty streams of `Int`s, representing "runs" of brightness levels greater than some threshold. The thresholding operation `thresh` takes a stream of `Int`s and produces runs of elements above the threshold. Whenever the incoming stream goes above the threshold `t`, we collect all of the subsequent elements into a run, emit it, and recurse down the rest of the stream. This uses an operation `spanGt : {Int} (Int*) -> Int . Int*` that returns the initial "span" of elements above `t`, followed by the rest of the stream. It's important to note that `thresh` does not wait for a complete run to produce output: as soon as the first element above `t` arrives, it is forwarded along, as are all subsequent elements until the stream drops below `t`. In contrast with homogeneously typed streaming languages, Delta's type safety guarantees that `thresh` outputs a stream that adheres to the protocol, and ensures downstream transformers do not have to replicate this parsing logic.

We can use this parsed stream of runs to compute per-run averages, by `mapping` an `averageSingle`

operation—taking `Int . Int*` to `Int`—over the stream of runs. This operation is defined by computing the sum and length of a run in parallel, waiting for the results, then computing the average. If it consumed a homogeneous stream type like `(Start + Int + End)*`, this average-each-run operation would need to be written in a low-level, more stateful manner, remembering the current run of `Int`s until an `End` event arrives, averaging, and handling the divide-by-zero error which could in principle occur if no `Int`s arrived between a `Start` and an `End`. The complete program, first calling `thresh`, and then mapping `averageSingle` over the stream of runs, is `averageAbove`.

```
fun thresh{t : Int}(xs : Int*) : (Int . Int*)* =
  case xs of
    nil => nil
  | y :: ys => wait y do
                 if {y > t} then
                   let (run;rest) = spanGt{t}(ys) in
                   ({y};run) :: thresh{t}(rest)
                 else
                   thresh{t}(ys)
               end

fun averageSingle (run : Int . Int*) : Int =
  let (x;xs) = run in
  let (sm,len) = (sum(xs), length(xs)) in
  wait x,sm,len do
    {(x + sm) / (1 + len)}
  end

fun averageAbove{t : Int}(xs : Int*) : Int* =
  map<averageSingle>(thresh{t}(xs))
```

3.1.11 FM Radio Example

Strymonas [105, 106] is a stream processing library in OCaml, focused on low latency code. Strymonas is notable for using multi-stage programming to guarantee complete stream fusion of all operations down to a single imperative loop. The flagship application example used by the Strymonas project is an AM softare-defined radio [106], which is a hundred or so lines of Strymonas OCaml code. To give an example

of how Delta can make the sort of combinator-heavy code nicer to read, write, and maintain, we present the process of writing an indicative function here.

One important routine in the AM radio module is upsampling. Upsampling takes a stream, and replicates each sample in the stream some number $k = \frac{\texttt{rate\_to}}{\texttt{rate\_from}}$ times. The way that this must be written in Strymonas is quite strange: they `flat_map` a function over the stream that takes each sample `s`, builds the list from 0 to $k - 1$, and then maps `const s` over it, to replicate the value `s` k times.

```
let upsample (rate_from:int) (rate_to:int) : 'a stream -> 'a stream =
  let k = rate_to / rate_from in
  assert (k > 1 && k * rate_from = rate_to );
  flat_map (fun s -> map (Fun.const s) (from_to C.(int 0) (C.int (k - 1))))
```

Importantly, `k` is completely known at compile time.

Because of this, we can give this program the much nicer type $\texttt{s} \longrightarrow \texttt{s} \cdot \ldots \cdot \texttt{s}$, with `k` copies. For `k = 2`, we write this program as

```
fun dup2[s](x : s) : s . s =
  wait x do
    ({x};{x})
  end

fun upsample2[s](xs : s*): (s . s)* = map<dup2> xs
```

If we wanted to give this the weaker type `s* -> (s*)*` (potentially because k was large), this is also straightforward:

```
fun dup[s]{k : Int}(x : s) : s* =
  wait x do
    { replicate k x }
  end

fun upsample[s]{k : Int}(xs : s*) : s** =
  map<dup{k}>(xs)
```

Of course, to arrive at the original type of the strymonas example, we can simply postcompose this function with `concat`, written as usual.

3.2 Delta Implementation Details

The implementation of Delta can be found at https://github.com/alpha-convert/delta. The core of this implementation is a direct-style interpreter for λ^{ST} core calculus presented in Chapter 4. The rest of the implementation is simply a set of desugarings, typechecking passes, macro expansions, and elaborations that transform the human-writeable surface syntax down into the core calculus. We briefly describe this sequence of transformations here.

The implementation first lowers the surface syntax to an "elaborated syntax" via a transformation which eliminates shadowing, resolves function calls, and transforms the syntax into the sequent calculus representation by introducing intermediate variables for subexpressions. Elaborated terms are then typechecked. Typechecking expands macros and produces *monomorphizers* of λ^{ST} terms: functions from closed types (to plug in for type variables) to monomorphic λ^{ST} terms. Terms of base type can then be evaluated with a definitional interpreter that implements the λ^{ST} semantics. The typechecker uses an orderedness-checking algorithm for typechecking our variant of ordered & bunched terms. While we have tested the typechecker with many terms, we have not proved that the algorithm is sound and complete with respect to the declarative type system. The interpreter, on the other hand, is very straightforward: it is a direct translation of the λ^{ST} semantics into code.

Chapter 4

λ^{ST}, the Formal Foundation of Delta

In this chapter, we present λ^{ST}, the ordered type theory that underlies Delta. We begin by defining the most important constructors of stream types and the corresponding features of the term language; these form the "kernel" of the λ^{ST} calculus. The rest of the types and terms of Full λ^{ST} will layered on bit by bit in Section 4.1. The complete definitions are somewhat lengthy and hence presented in Appendix A. Full proofs can be found in the associated Rocq mechanization, found at .

The *concatenation* constructor $\cdot$ describes streams that *vary* over time: if s and t are stream types, then $s \cdot t$ describes a stream on which all the elements of s arrive first, followed by the elements of t. A producer of a stream of type $s \cdot t$ must first produce a stream of type s and then a stream of type t, while a consumer can assume that the incoming data will first consist of data of type s and then of type t. The transition point between the s and t parts is handled automatically by λ^{ST}'s semantics: the underlying data of a stream of type $s \cdot t$ includes a *punctuation marker* [180] indicating the cross-over. One consequence of this is that, unlike Kleene Star for regular languages, streams of type $s^\star$ are distinguishable from streams of type $s^\star \cdot s^\star$ because a transformer accepting the latter can see when its input crosses from the first $s^\star$ to the second.

On the other hand, the *parallel* stream type $s \| t$ describes a stream with two parallel substreams of types s and t. Semantically, the s and t components are produced and consumed independently: a transformer that produces $s \| t$ may send out an entire s first and then a t, or an entire t and then the s, or any interleaving of the two. Conversely, a transformer that accepts $s \| t$ must handle all these possibilities uniformly, by processing the s and t parts independently. To enable this, each element in the parallel stream is tagged to indicate which substream it belongs to. This means that streams of type $s \| t$ are isomorphic, but not identical, to streams of type $t \| s$, and similarly $\texttt{Int}^\star \| \texttt{Int}^\star$ is not the same as $\texttt{Int}^\star$.

Parallel types can be combined with concatenation types in interesting ways. For example, a stream of type $(s \| t) \cdot r$ consists of a stream of interleaved items from s and t, followed (once all the s data and t data has arrived) by a stream of type r. By contrast, a stream of type $(s \cdot t) \| (s' \cdot t')$ has two interleaved components, one a stream described by s followed by a stream described by t and the other an s' followed

by a t'. The fact that the parallel type is on the outside means that the change-over points from s to t and s' to t' are completely independent.

The base type 1 describes a stream containing just one data item, itself a unit value. The other base type is ε, the type of the empty stream containing no data; it is the unit for both the $\cdot$ and $\|$ constructors—i.e., $s \cdot \varepsilon$, $\varepsilon \cdot s$, $\varepsilon \| s$ and $s \| \varepsilon$ are all isomorphic to s, in the sense that there are λ^{ST} transformers that convert between them.

In summary, the Kernel λ^{ST} stream types are given by the grammar on the top left in Figure 4.1. (So far, these types can only describe streams of fixed, finite size. In Section 4.1.2 we will enrich the kernel type system with unbounded streams via the Kleene star type $s^\star$.)

What about terms? Recall that our goal is to develop a language of core terms e, typed by stream types, where well-typed terms $x : s \vdash e : t$ are interpreted as stream transformers accepting a stream described by s and producing one described by t. The term e runs by accepting some inputs as described by s, producing some outputs as described by t, and then stepping to a new term e', with an updated type t', that is ready to accept the rest of the input and produce the rest of the output. This process happens reactively: output is only produced when an input arrives. The formal semantics of λ^{ST} is described in Section 4.0.2.

To represent stream transformers with multiple parallel and sequential inputs, we draw upon insights from proof theory. Both the types $s \cdot t$ and $s \| t$ are *product types*, in the sense that a stream of either of these types contains both the data of a stream of type s and a stream of type t—although the temporal structure differs between the two. A standard observation from proof theory is that, in situations where a logic or type theory includes two products with different structural properties, the corresponding typing judgment requires a context with two different *context formers*.[10]

The first context former, written with a comma ($\Gamma\,,\,\Delta$), describes inputs to a transformer arriving in parallel, one component structured according to Γ and the other according to Δ. The second context former, written with a semicolon ($\Gamma\,;\,\Delta$) describes inputs that will first arrive from the environment according to Γ, then according to Δ.

These interpretations are enforced by restricting the ways that these contexts can be manipulated using

[10]Such *bunched* contexts were first introduced in the Logic of Bunched Implication [141], the basis of modern separation logic [155]. Our bunched contexts differ from those of BI by the choice of structural rules: our substructural type former is affine ordered, while the BI one is linear.

$$s, t, r := 1 \mid \varepsilon \mid s \cdot t \mid s \| t \qquad \Gamma ::= \cdot \mid \Gamma, \Gamma \mid \Gamma; \Gamma \mid x : s$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1, e_2) : s \| t}\ \text{T-Par-R} \qquad \frac{\Gamma(x : s, y : t) \vdash e : r}{\Gamma(z : s \| t) \vdash \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e : r}\ \text{T-Par-L}$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : t}{\Gamma\,;\Delta \vdash (e_1; e_2) : s \cdot t}\ \text{T-Cat-R} \qquad \frac{\Gamma(x : s; y : t) \vdash e : r}{\Gamma(z : s \cdot t) \vdash \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e : r}\ \text{T-Cat-L}$$

$$\frac{}{\Gamma \vdash \texttt{eps} : \varepsilon}\ \text{T-Eps-R} \qquad \frac{}{\Gamma \vdash (\,) : 1}\ \text{T-One-R} \qquad \frac{}{\Gamma(x : s) \vdash x : s}\ \text{T-Var}$$

$$\frac{\Gamma \leq \Gamma' \qquad \Gamma' \vdash e : s}{\Gamma \vdash e : s}\ \text{T-SubCtx}$$

Figure 4.1: Kernel λ^{ST} syntax and typing rules

structural rules. Comma contexts can be manipulated in all the ways standard contexts can: their bindings can be reordered (from Γ, Δ to Δ, Γ) duplicated, and dropped. Semicolon contexts, on the other hand, are ordered and affine: a context $\Gamma\,;\Delta$ cannot be freely rewritten to a context $\Delta\,;\Gamma$, and a context Γ cannot be duplicated into $\Gamma\,;\Gamma$. These restrictions enforce the interpretation of $\Gamma\,;\Delta$ as data arriving according to Γ and then Δ: to exchange them would be to allow a consumer to assume that the data is sent in the opposite order, and to duplicate would be to assume that the data input will be replayed.

Thus, part of our type system is *substructural*: the semicolon context former is ordered (no exchange) and affine (no contraction), while the comma context former is fully structural. Both context formers are associative, with the empty context serving as a unit for each. (The full list of structural rules can be found in the extended Stream Types paper [48]) Formally, stream contexts are drawn from the grammar at the top right of Figure 4.1.

4.0.1 Kernel Typing Rules

The typing rules for Kernel λ^{ST} are collected in Figure 4.1. The typing judgment, written $\Gamma \vdash e : s$, says that e is a stream transformer from a collection of streams structured like Γ to a single stream structured

like s. [11]

The most straightforward typing rule is the right rule for parallel (T-Par-R). It says that, from a context Γ, we can produce a stream of type $s \| t$ by producing s and t independently from Γ, using transformers e_1 and e_2. We write the combined transformer as a "parallel pair" (e_1, e_2). Semantically, it operates by copying the inputs arriving on Γ, passing the copies to e_1 and e_2, and merging the tagged outputs into a parallel stream. Similarly, the T-Cat-R rule is used to produce a stream of type $s \cdot t$. It uses a similar pairing syntax—if term e_1 has type s and e_2 has type t, then the "sequential pair" $(e_1; e_2)$ has type $s \cdot t$—but the context in the conclusion differs. Since e_1 needs to run before e_2, the part of the input stream that e_1 depends on must arrive before the part that e_2 depends on. Semantically, this term will operate by accepting data from the Γ part of the context and running e_1; once e_1 has produced its output it will switch to running e_2, consuming data from Δ.

These right rules describe how to *produce* a stream of parallel or concatenation type. The corresponding left rules describe how to *use* a variable of one of these types appearing somewhere in the context. Syntactically, the terms take the form of let-bindings that deconstruct variables of type $s \cdot t$ (or $s \| t$) as pairs of variables of type s and t, connected by ; (or ,). We use the standard BI notation $\Gamma(-)$ for a context with a hole and write $\Gamma(\Delta)$ when this hole has been filled with the context Δ. In particular, $\Gamma(x : s)$ is a context with a distinguished variable x.

The T-Par-L rule says that if z is a variable of type $s \| t$ somewhere in the context, we can replace its binding with with a pair of bindings for variables x and y of types s and t and use these in a continuation term e of final type r. When typing e, the variables x and y appear in the same position as the original variable z, separated by a comma—i.e., x and y are assumed to arrive in parallel. Similarly, the rule T-Cat-L says that if a variable z of type $s \cdot t$ appears somewhere in the context, it can be let-bound to a pair of variables x and y of types s and t that are again used in the continuation e. This time, though, x and y are separated by a semicolon—i.e., the substream bound to x will arrive and be processed first, followed by the substream bound to y.

T-Eps-R and T-One-R are the right rules for the two base types, witnessed by the terms `eps` and `()`.

[11] To simplify aspects of the semantics, the typing rules are presented in sequent-calculus style, rather than the more familiar natural-deduction style. The main difference is that sequent calculi have left and right rules—describing how to eliminate a connective when it appears in the context or in the result type—in place of natural deduction's introduction and elimination rules.

Semantically, `eps` does nothing: it accepts inputs on Γ and produces no output. On the other hand, `()` emits a unit value as soon as it receives its first input and never emits anything else.

The variable rule (T-Var) says that if $x : s$ is a variable somewhere in the context, then we can simply send it along the output stream. Semantically, it works by dropping everything in the context except for the s-typed data for x, which it forwards along.

The rule T-SubCtx bundles together all of the structural rules as a subtyping relation on contexts. For example, the weakening rule for semicolon contexts is written $\Gamma ; \Delta \leq \Gamma$ and the comma exchange rule is $\Gamma , \Delta \leq \Delta , \Gamma$.

Examples and Non-Examples

To show the typing rules in action, here are two small examples of transformers written in Kernel λ^{ST}, as well as three examples of programs that are *rejected* by the type system. The first example is a simple "parallel-swap" transformer, which accepts a stream z of type $s \| t$ and outputs a stream of type $t \| s$, swapping the parallel substreams:

$$z : s \| t \vdash \texttt{let } (x, y) \texttt{ = } z \texttt{ in } (y, x) : t \| s$$

It works by splitting the variable $z : s \| t$ into variables $x : s$ and $y : t$ and yielding a parallel pair with the order reversed.

$$\dfrac{\dfrac{\lightning}{x : s \vdash y : t}\ \text{T-Var} \quad \dfrac{\lightning}{y : t \vdash x : s}\ \text{T-Var}}{x : s ; y : t \vdash (y; x) : t \cdot s}\ \text{T-Cat-R}}{z : s \cdot t \vdash \texttt{let}_s\ (x; y) = z \texttt{ in } (y; x) : t \cdot s}\ \text{T-Cat-L}$$

The most important *non*-example is the lack of a corresponding "cat-swap" term, which would accept a stream z of type $s \cdot t$, and produce a stream of type $t \cdot s$. This program is undesirable because it is not

implementable without a space leak. Implementing it requires the entire stream of type s to be saved in memory to emit it after the stream of type t.[12] The natural term for this program would be $\texttt{let}_s\ (x; y) = z\ \texttt{in}\ (y; x)$, but this does not typecheck. Applying the syntax-directed rules gets us to a point where we must show that y has type t in a context with only x and that x has type s in a context with only y. This is because the T-Cat-R splits the context, but the variables are listed in the opposite order from what we'd need. The lack of a structural rule to let us permute the x and y in the context means that there is nothing we can do here, and a typechecker will reject this program.

The second example is a "broadcast" transformer, which takes a variable $x : s$ and outputs a stream of type $s \| s$, duplicating the variable and sending it out to two parallel outputs: $x : s \vdash (x, x) : s \| s$.

The second non-example is the "replay" transformer, which would (if it existed) take a variable $x : s$ and produce a stream $s \cdot s$ that repeats the input stream twice. This is the concat-equivalent of the broadcast transformer, and it is undesirable for the same reason as the cat-swap program: it would require saving the entire incoming stream of type s in order to replay it. This time, the failure of the natural term $(x; x)$ to typecheck comes down to the lack of a contraction rule for semicolon contexts: we are not permitted to turn a context $x : s$ into a context $x : s\,; x : s$.

The last non-example is a "tie-breaking" transformer, which would take a stream $z : \texttt{Int} \| \texttt{Int}$ of two ints in parallel and produce a stream of type `Int` by forwarding along the `Int` that arrives first. This program, like others that require inspecting the interleaving of substreams in a stream of type $s \| t$, is not expressible. In Section 4.0.3, we'll prove that a well-typed program cannot implement this behavior.

4.0.2 Prefixes and Semantics

We next define the semantics of Kernel λ^{ST}. The natural notion of "values" in this semantics is finite prefixes of streams, and the meaning of a well-typed term $\Gamma \vdash e : s$ is a function that accepts an environment mapping variables in Γ to prefixes of streams and produces a prefix of a stream of type s. Because the streams that λ^{ST} programs operate over are more structured than traditional homogeneous streams—including cross-over punctuation in streams of type $s \cdot t$ and disambiguating tags in streams of type $s \| t$—the prefixes are also more structured. That is, a prefix in λ^{ST} is not a simple sequence of data items, but a structured

[12] A program with this behavior is actually implementable in λ^{ST}, but only with a special additional construct—see Section 4.1.5—ensuring that leaky programs like this one cannot be written accidentally.

$$\frac{}{\texttt{epsEmp} : \texttt{prefix}\,(\varepsilon)} \qquad \frac{}{\texttt{oneEmp} : \texttt{prefix}\,(1)} \qquad \frac{}{\texttt{oneFull} : \texttt{prefix}\,(1)}$$

$$\frac{p : \texttt{prefix}\,(s) \quad p' : \texttt{prefix}\,(t)}{\texttt{parPair}(p, p') : \texttt{prefix}\,(s \| t)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{catFst}(p) : \texttt{prefix}\,(s \cdot t)} \qquad \frac{p' : \texttt{prefix}\,(t) \quad p : \texttt{prefix}\,(s) \quad p\ \texttt{maximal}}{\texttt{catBoth}(p, p') : \texttt{prefix}\,(s \cdot t)}$$

Figure 4.2: Prefixes for Types

value whose possible shapes are determined by its type [125].

There are two prefixes of a stream of type 1: the empty prefix, written `oneEmp`, and the prefix containing the single element (), written `oneFull`. Similarly, the unique stream of type ε has a single prefix, the empty prefix, which we write `epsEmp`.

What about $s \| t$? A parallel stream of type $s \| t$ is conceptually a pair of independent streams of type s and t, so a prefix of a parallel stream should be a pair $\texttt{parPair}(p_1, p_2)$, where p_1 is a prefix of a stream of type s, and p_2 is a prefix of a stream of type t. Crucially, this definition encodes no information about any interleaving of p_1 and p_2: the prefix $\texttt{parPair}(p_1, p_2)$ equally represents a situation where all of p_1 arrived first and then all of p_2, one where p_2 arrived before p_1, and many others where the elements of p_1 and p_2 arrived in some interleaved order. In a nutshell, this definition is what guarantees deterministic processing. By representing all possible interleavings using the same prefix value, we ensure that a transformer that operates on these values cannot possibly depend on ordering information that isn't present in the type.

Finally, let's consider the prefixes of streams of type $s \cdot t$. One case is a prefix that only includes data from s because it cuts off before reaching the point where the $s \cdot t$ stream stops carrying elements of s and starts on t. We write such a prefix as $\texttt{catFst}(p)$, with p a prefix of type s. The other case is where the prefix does include the crossover point—i.e., it consists of a "completed" prefix of s plus a prefix of t. We write this as $\texttt{catBoth}(p, p')$, with p a prefix of s and p' a prefix of t. The requirement that p be completed is formalized by the judgment $p\ \texttt{maximal}$, which ensures that the prefix p describes an entire completed stream (see the extended Stream Types paper [48] for details). We formalize all these possibilities as a judgment $p : \texttt{prefix}\,(s)$, shown in Figure 4.2.

Every type s has a distinguished *empty prefix*, written $\texttt{emp}_s$ and defined by straightforward recursion on s (see the extended Stream Types paper [48]). We then lift the idea of prefixes from types to contexts,

$$\frac{}{\eta \,:\, \mathtt{env}\,(\cdot)} \qquad \frac{\eta(x) \mapsto p \qquad p \,:\, \mathtt{prefix}\,(s)}{\eta \,:\, \mathtt{env}\,(x : s)} \qquad \frac{\eta \,:\, \mathtt{env}\,(\Gamma) \qquad \eta \,:\, \mathtt{env}\,(\Delta)}{\eta \,:\, \mathtt{env}\,(\Gamma\,,\Delta)}$$

$$\frac{\eta \,:\, \mathtt{env}\,(\Gamma) \qquad \eta \,:\, \mathtt{env}\,(\Delta) \qquad (\eta\ \mathtt{maximalOn}\ \Gamma) \vee (\eta\ \mathtt{emptyOn}\ \Delta)}{\eta \,:\, \mathtt{env}\,(\Gamma\,;\Delta)}$$

Figure 4.3: Environments for Contexts

defining an *environment* η for a context Γ to be a mapping from the variables $x : s$ in Γ to prefixes of the corresponding types s; we write this with a judgment $\eta \,:\, \mathtt{env}\,(\Gamma)$ (Figure 4.3). Besides ensuring that η has well-typed bindings for all variables, the judgment ensures that the prefixes respect the order structure of the context. In particular, an environment η for a semicolon context $\Gamma\,;\Delta$ must assign prefixes *in order*: the prefixes for Γ, the earlier part of the context, must all be complete before any of the prefixes for Δ can begin. In other words, either η assigns maximal prefixes to every variable in Γ—which we write $\eta\ \mathtt{maximalOn}\ \Gamma$—or η assigns empty prefixes to every variable in Δ—which we write $\eta\ \mathtt{emptyOn}\ \Delta$.

Semantics

We next describe how well-typed λ^{ST} terms behave with an operational semantics. Given a well-typed term $\Gamma \vdash e : s$ and an input environment $\eta \,:\, \mathtt{env}\,(\Gamma)$, this semantics describes how to run e with η to produce an output prefix p. It also describes how to produce a "resultant" term e' that is ready to continue the computation once further data arrives on the input stream data. Formally, the semantics is given by a judgment $\eta \Rightarrow e \downarrow e' \Rightarrow p$, pronounced "running the core term e on the input environment η yields the output prefix p and steps to e'." The rules for this judgment are gathered in Figure 4.4 and described below; the full set of rules for all of λ^{ST} can be found in Appendix A.

The following theorem establishes the soundness of the Kernel λ^{ST} semantics, formalizing the intuitive description given above: If we run a well-typed core term e on an environment η of the context type, it will return a prefix p with the result type s and step to a term e' that is well typed in context "the rest of Γ" after η and has type "the rest of s" after p. The "rest" of a type (or context) after a prefix (or environment) is, intuitively, its *derivative* with respect to the prefix (or environment), in the sense of standard Brzozowski derivatives of regular expressions [35]—we make this formal in Section 4.0.2. Most critically, the types of the variables in e and e' are different: if x has type s in e, then x has type $\delta_{\eta(x)}s$ in e', having already

consumed $\eta(x)$.

Theorem 4.0.1 (Soundness of the Kernel λ^{ST} Semantics)**.** *Suppose:* $\Gamma \vdash e : s$ *and* $\eta : \mathtt{env}(\Gamma)$. *Then there are* p *and* e' *such that* $\eta \Rightarrow e \downarrow e' \Rightarrow p$, *with* $p : \mathtt{prefix}(s)$ *and* $\delta_\eta(\Gamma) \vdash e' : \delta_p(s)$

See the mechanization for the proof of soundness for Full λ^{ST}.

In light of the soundness theorem, the operational semantics can be thought of as defining a reactive state machine. Well-typed terms $\Gamma \vdash e : s$ are the states, while the semantic judgment defines the transition function: when new inputs η arrive, the machine produces an output prefix p and steps to a new state $\delta_\eta(\Gamma) \vdash e' : \delta_p(s)$. This form of semantics—a state machine with terms themselves as states, typed by derivatives—was pioneered by the Esterel programming language [26].

Semantics of the Right Rules

The right rules for parallel and concatenation are the simplest to understand. For S-Par-R, we accept an environment η and use it to run the component terms e_1 and e_2, independently producing outputs p_1 and p_2 and stepping to new terms e_1' and e_2'. The pair term (e_1, e_2) then steps to (e_1', e_2') and produces the output $\mathtt{parPair}(p_1, p_2)$.

There are two rules, S-Cat-R-1 and S-Cat-R-2, for running the concatenation pair $(e_1; e_2) : s \cdot t$. In either case, we begin by running e_1 with the environment η, producing a prefix p and term e_1'. If p is not maximal, we stop there: more of the input is needed for the first component to produce the rest of s, so it is not yet time to start running e_2 to produce t. This case is handled by S-Cat-R-1, where the resulting term is $(e_1'; e_2)$ and the output prefix is $\mathtt{catFst}(p)$. On the other hand, if p is maximal, then we also run e_2, which steps to e_2' and produces a prefix p' using rule S-Cat-R-2; the entire term then outputs $\mathtt{catBoth}(p, p')$ and steps to e_2'. Note that the pair is eliminated in the process: we step from $(e_1; e_2)$ to just e_2'. This is because we are done producing the s part of the $s \cdot t$, and so a subsequent step of evaluation only has to run e_2' to produce the rest of the t.

Semantics of Variables

The variable semantics S-Var is a simple lookup. We find the prefix bound to the variable x in the environment, return it, and then step to x itself.

$$\frac{\eta(x) \mapsto p}{\eta \Rightarrow x \downarrow x \Rightarrow p}\ \text{S-Var} \qquad \frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1 \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1, e_2) \downarrow (e_1', e_2') \Rightarrow \texttt{parPair}(p_1, p_2)}\ \text{S-Par-R}$$

$$\frac{\eta(z) \mapsto \texttt{parPair}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e \downarrow \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e' \Rightarrow p'}\ \text{S-Par-L}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \neg\,(p\ \texttt{maximal})}{\eta \Rightarrow (e_1; e_2) \downarrow (e_1'; e_2) \Rightarrow \texttt{catFst}(p)}\ \text{S-Cat-R-1}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad p\ \texttt{maximal} \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow (e_1; e_2) \downarrow e_2' \Rightarrow \texttt{catBoth}(p, p')}\ \text{S-Cat-R-2}$$

$$\frac{\eta(z) \mapsto \texttt{catFst}(p) \qquad \eta[x \mapsto p, y \mapsto \texttt{emp}_t] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e \downarrow \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e' \Rightarrow p'}\ \text{S-Cat-L-1}$$

$$\frac{\eta(z) \mapsto \texttt{catBoth}(p, p') \qquad \eta[x \mapsto p, y \mapsto p'] \Rightarrow e \downarrow e' \Rightarrow p''}{\eta \Rightarrow \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e \downarrow \texttt{let}\ x = \texttt{sink}_p\ \texttt{in}\ e'[z/y] \Rightarrow p''}\ \text{S-Cat-L-2}$$

$$\frac{}{\eta \Rightarrow \texttt{eps} \downarrow \texttt{eps} \Rightarrow \texttt{epsEmp}}\ \text{S-Eps-R}$$

$$\frac{}{\eta \Rightarrow \texttt{()} \downarrow \texttt{eps} \Rightarrow \texttt{oneFull}}\ \text{S-One-R}$$

Figure 4.4: Incremental semantics of Kernel λ^{ST}

Semantics of Left Rules

The left rules for concatenation and parallel are similar, both accepting an environment η with a binding for $z : s \otimes t$ where $\otimes$ is one of the two products, binding variables x and y of types s and t to the two components of the product, and using the updated environment to run the continuation term. In the case of the left rule for parallel (S-Par-L), looking up z of type $s \| t$ will always yield a prefix $\texttt{parPair}(p_1, p_2)$. The rule binds p_1 to x and p_2 to y and runs the continuation term, stepping to e' and producing the output prefix p. Then the whole term steps to $\texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e'$ and produces p.

The left rule for concatenation has two cases, depending on what kind of prefix comes back from the lookup for z. If the lookup yields is $\texttt{catFst}(p)$, then the rule S-Cat-L-1 applies. Since no data for y has arrived, we bind y to $\texttt{emp}_t$, the empty prefix of type t, and run the continuation.[13] If the result comes back as $\texttt{catBoth}(p, p')$, then the rule S-Cat-L-2 applies, so we run the continuation with x and y bound to p and p'.

Both rules output the prefix obtained from running the continuation, but they step to different resulting terms. If $\eta(z) = \texttt{catFst}(p)$, then the resulting term must be another use of Cat-L: the variable z still expects to get some more of the first component of the concatenation, and then the second component. If $\eta(z) = \texttt{catBoth}(p, p')$ on the other hand, the z stream has crossed over to the second part. In this case, we close over the (now not-needed) x variable in e' and connect z to the y input of e' by substituting y for z.

Derivatives

When $p : \texttt{prefix}\ (s)$, we write $\delta_p\ (s)$ for the derivative [35] of s by p—the type of streams that result after a prefix of type p has been "chopped off" the beginning of a stream of type s. Because this operation is partial—$\delta_p\ (s)$ is only defined when $p : \texttt{prefix}\ (s)$—we formally define this as a a 3-place relation, written as $\delta_p\ (s) \sim s'$ and pronounced as "the derivative of s with respect to p is s'" (see Figure 4.5).

The derivative of the type 1 with respect to the empty prefix `oneEmp` is 1 (the rest of the stream is the entire stream), and its derivative with respect to the full prefix `oneFull` is ε (there is no more stream left after the unit element has arrived). For parallel, the derivative is taken component-wise. The interesting cases

[13] This need to compute $\texttt{emp}_t$ from t at runtime to bind to $y \mapsto \texttt{emp}_t$ is the reason that the term for `T-Cat-L`, $\texttt{let}_t\ (x; y) = z\ \texttt{in}\ e$, includes a t in the syntax. In Section 4.1, the case analysis expressions for star types and sum types will have similar annotations for the same reason.

$$\frac{}{\delta_{\texttt{epsEmp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\texttt{oneEmp}}\,(1) \sim 1} \qquad \frac{}{\delta_{\texttt{oneFull}}\,(1) \sim \varepsilon} \qquad \frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s' \cdot t}$$

$$\frac{\delta_{p'}\,(t) \sim t'}{\delta_{\texttt{catBoth}(p,'p)}\,(s \cdot t) \sim t'} \qquad \frac{\delta_p\,(s) \sim s' \qquad \delta_{p'}\,(t) \sim t'}{\delta_{\texttt{parPair}(p,p')}\,(s \| t) \sim s' \| t'}$$

Figure 4.5: Derivatives

are those for the concatenation type. If the prefix has the form $\texttt{catFst}(p)$, the derivative $\delta_{\texttt{catFst}(p)}\,(s \cdot t)$ is $\left(\delta_p\,(s)\right) \cdot t$, i.e., some of the s has gone by but not all, and once it does we still expect t to come after it. On the other hand, if the prefix has the form $\texttt{catBoth}(p, p')$, the derivative $\delta_{\texttt{catBoth}(p,p')}\,(s \cdot t)$ is just $\delta_{p'}\,(t)$, i.e., the s component is complete, and the rest of the stream is just the part of t after p'.

This definition is lifted to contexts and environments pointwise: if $x : s$ is a variable in Γ, the derivative of $\delta_\eta\,(\Gamma)$ has $x : \delta_{\eta(x)}\,(s)$ in the same location.

4.0.3 The Homomorphism Property and Determinism

The semantics is designed to run a stream transformer on "input chunks" of any size, from individual input events one at a time all the way up to the entire stream at once. The cost of this flexibility is that it raises the question of *coherence*—i.e., whether we are guaranteed to arrive at the *same* final output depending on how we carve up a transformer's input into a series of prefixes. Fortunately, this is indeed guaranteed. Coherence is a corollary of our main technical result: a *homomorphism theorem* that says running a term e on an environment η and then running the resulting term e' on an environment η' of appropriate type produces the same end result as running e on the combined environment [14]

Theorem 4.0.2 (Homomorphism Theorem)**.** *Suppose*

1. $\Gamma \vdash e : s$
2. $\eta \;:\; env\,(\Gamma)$
3. $\eta' \;:\; env\left(\delta_\eta\,(\Gamma)\right)$

[14]This theorem is similar to the "factorization independence" property required of stream transducers in work by Mamouras [125], or the "eager evaluation" of Laddaj [113].

4. $p : \mathit{prefix}(s)$,

5. $p' : \mathit{prefix}(\delta_p(s))$

6. $\eta \Rightarrow e \downarrow e' \Rightarrow p$

7. $\eta' \Rightarrow e' \downarrow e'' \Rightarrow p'$

Then, if $\eta \cdot \eta' \Rightarrow e \downarrow e''' \Rightarrow p''$, *we have* $p'' = p \cdot p'$, *and* $e''' = e''$

The proof of this theorem for Full λ^{ST} can be found in the corresponding Rocq development [CITE].

The operation $p \cdot p'$ here is *prefix concatenation*, which takes a prefix p of type s and a prefix p' of type $\delta_p(s)$ and produces the prefix of type s that is first p and then p'. Formally, this is defined as a 3-place partial inductive relation $p \cdot p' \sim p''$, which is defined when p and p' have types s and $\delta_p(s)$, respectively. The operation $\eta \cdot \eta' \sim \eta''$ does the same for environments. See Appendix A for details.

The theorem's name derives from a reframing of the result: when we view prefixes and environments as a monoid under concatenation, the homomorphism property says that core terms e behave like a sort of stateful monoid homomorphism: $e(\eta \cdot \eta') = e(\eta) \cdot e'(\eta')$, where e' is the term that e steps to after processing η.

The homomorphism theorem not only justifies running the semantics on prefixes of any size; it also implies deterministic processing of parallel streams. Intuitively, determinism states that the results of a stream transformer do not depend on the particular order in which parallel data arrives. We formalize this through the following scenario. Suppose $\Gamma, \Gamma' \vdash e : s$ is a term with two parallel contexts serving as its input, and suppose that η is an environment for Γ, Γ'. Write $\eta_1 = \eta|_\Gamma$ and $\eta_2 = \eta|_{\Gamma'}$, for the restrictions of η to the variables in Γ and Γ', respectively. There are two different ways of running e on this data. One is to first run e on $\eta_1 \cup \mathtt{emp}_{\Gamma'}$ (which has η_1 bindings for Γ and then the empty prefix for everything in Γ') and then run the resulting term on $\eta_2 \cup \mathtt{emp}_\Gamma$ (with an empty prefixes for Γ). The other does the opposite, first running e on $\eta_2 \cup \mathtt{emp}_\Gamma$ and then running the resulting term on $\eta_1 \cup \mathtt{emp}_{\Gamma'}$. Determinism says that these strategies produce equal results.

Theorem 4.0.3 (Determinism)**.** *Suppose*

1. $\Gamma, \Gamma' \vdash e : s$

2. η : $prefix(\Gamma, \Gamma')$

3. $\eta|_\Gamma \cup emp_{\Gamma'} \Rightarrow e \downarrow e_1 \Rightarrow p_1$ *and* $\eta|_{\Gamma'} \cup emp_\Gamma \Rightarrow e_1 \downarrow e_2 \Rightarrow p_2$

4. $\eta|_{\Gamma'} \cup emp_\Gamma \Rightarrow e \downarrow e_1' \Rightarrow p_1'$

5. $\eta|_\Gamma \cup emp_{\Gamma'} \Rightarrow e_1' \downarrow e_2' \Rightarrow p_2'$

Then $e_2 = e_2'$ *and* $p_1 \cdot p_2 = p_1' \cdot p_2'$.

Proof. By two uses of Theorem 4.0.2, we have:

$$\Gamma, \Gamma' \cdot \texttt{parPair}(p_1, \texttt{emp}_{\Gamma'})\texttt{parPair}(\texttt{emp}_\Gamma, p_2) \Rightarrow e \downarrow e_2 \Rightarrow s \cdot p_1' p_2'$$

$$\Gamma, \Gamma' \cdot \texttt{parPair}(\texttt{emp}_\Gamma, p_2)\texttt{parPair}(p_1, \texttt{emp}_{\Gamma'}) \Rightarrow e \downarrow e_2' \Rightarrow s \cdot p_1'' p_2''$$

But the definition of prefix concatenation for parallel tells us that both of these inputs are equal to $\texttt{parPair}(p_1, p_2)$:

$$\begin{aligned} prefixConcat\texttt{parPair}(p_1, \texttt{emp}_{\Gamma'})\texttt{parPair}(\texttt{emp}_\Gamma, p_2) &= \texttt{parPair}(p_1, p_2) \\ &= \texttt{parPair}(\texttt{emp}_\Gamma, p_2) \cdot \texttt{parPair}(p_1, \texttt{emp}_{\Gamma'}) \end{aligned}$$

By the fact that the inputs uniquely determine the outputs of the incremental semantics, this means that the resulting terms e_2 and e_2' are equal, and the outputs $s \cdot p_1' p_2'$ and $s \cdot p_1'' p_2''$ are also equal, as required. □

Two crucial observations make the proof work. First, prefixes are morally canonical representatives of equivalence classes of sequences of stream elements, up to the possible reorderings defined by their type [169]. The homomorphism theorem then guarantees that these normal forms are processed compositionally, and so are independent of the actual temporal ordering of parallel data—it suffices to compute on the combined normal forms from the two steps

Since the publication of the *Stream Types* paper, this perspective on determinsim has been formalized by Laddad et al. [113], who proved a very general form of the theorem above. Any stream processing program that (1) "behaves homomorphically" (i.e. the chunk size of inputs does not matter), and (2) structures parallel inputs so that concatenation along distinct channels commutes, must be deterministic. Follow-on

work by Hou et al. [90] inspects the converse, showing that a broad class of deterministic stream programs can be implemented as stateful monoid homomorphisms.

4.1 Full λ^{ST}

We now sketch the remaining types and terms of λ^{ST} that are not part of Kernel λ^{ST}.

4.1.1 Sums

Sum types in λ^{ST}, written $s+t$, are *tagged* unions: a stream of type $s+t$ is either a stream of type s or a stream of type t, and a consumer can tell which. Streams of type s are not the same as streams of type $s + s$, and streams of type $s + t$ are isomorphic to, but not identical to, streams of type $t + s$. Operationally, a producer of a sum stream sends a tag bit before sending the rest of the stream, to tell downstream consumers which side to expect. Conversely, a consumer of $s + t$ first reads the bit to learn which it is getting next.

A prefix of $s + t$ can be a prefix of one of s or one of t, written $\texttt{sumInl}(p)$ or $\texttt{sumInr}(p)$, or it can be $\texttt{sumEmp}$, the empty prefix of type $s + t$, which does not even include the initial tag bit. The derivatives with respect to these prefixes are defined as follows: (a) the empty prefix takes nothing off the type ($\delta_{\texttt{sumEmp}}\,(s + t) = s + t$) and (b) the two injections reduce to taking the derivative of the corresponding branch of the sum ($\delta_{\texttt{sumInl}(p)}\,(s + t) = \delta_p\,(s)$ and $\delta_{\texttt{sumInr}(p)}\,(s + t) = \delta_p\,(t)$).

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \texttt{inl}\,(e) : s + t}\ \text{T-Sum-R-1} \qquad \frac{\Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\Gamma(z : s + t) \vdash \texttt{case}_r(z, x.e_1, y.e_2) : r}\ \text{T-Sum-L-Surf}$$

The typing rules for sums are the normal injections on the right (T-Sum-R-1 and a symmetric rule T-Sum-R-2) and a case analysis rule on the left (T-Sum-L-Surf). The right rules operate by prepending their respective tags and then running the embedded terms. The left rule does case analysis: if the incoming stream z comes from the left of the sum, it is processed with e_1; if from the right, e_2. To run a sum case term, the semantics must dispatch on the tag that says if the stream z being destructed is a left or a right. But the prefix z might not include a tag, if only data from the surrounding context has arrived. In this

case, z will map to `sumEmp`, and we have no way of determining which branch to run. The solution is to run neither! Instead, we hold on to the environment, saving *all* incoming data to the program until the tag arrives. Once we get a prefix that includes the tag, we continue by running the corresponding branch with the accumulated inputs. Note that this buffering is necessarily a blocking operation[15].

$$\frac{\eta \,:\, \mathtt{env}\,(\Gamma(z : s + t)) \qquad \Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\delta_\eta\,(\Gamma(z : s + t)) \vdash \mathtt{case}_r\,(\eta; z, x.e_1, y.e_2) : r}\ \text{T-Sum-L}$$

All this requires a slightly generalized typing rule (T-Sum-L) that includes a *buffer environment* $\eta \,:\, \mathtt{env}\,(\Gamma(z : s + t))$ of the context type in the term. This buffer holds all of the input data we've seen so far. As prefixes arrive, we append to this buffer until we get the tag. Accordingly, the context in this rule is $\delta_\eta\,(\Gamma(z : s + t))$: the term is typed in the context consisting of everything after the part of the stream that has so far been buffered.

Fortunately, the only typing rule that a λ^{ST} programmer needs to concern themselves with is T-Sum-L-Surf. While writing the program, and before it runs, the buffer is empty ($\eta = \mathtt{emp}_{\Gamma(z:s+t)}$). In this case, the $\delta_\eta\,(\Gamma(z : s + t)) = \Gamma(z : s + t)$, and so the generalized rule T-Sum-L simplifies to the "surface" rule, T-Sum-L-Surf. Full details on the semantics of case analysis can be found in Appendix A.9.

4.1.2 Star

Full λ^{ST} also includes a type constructor for unbounded streams, written $s^\star$ because it is inspired by the Kleene star from the theory of regular languages. (We do not need to distinguish between unbounded finite streams and "truly infinite" ones, because our operational semantics is based on prefixes: we're always only operating on "the first part" of the input stream, and it doesn't matter whether the part we haven't seen yet is finite or infinite.) The type $s^\star$ describes a stream that consists of zero or more sub-streams of type s, in sequence.

[15] Depending on the rest of the context, it could also require unbounded memory! Fortunately, we believe we can detect this, and flag it as a warning to the user: running a case on $z : s + t$ in a context $\Gamma(z : s + t)$ could require buffering all variables to the left of z or in parallel with z in the context. Unbounded memory is required if and only if any of those variables have star type. We hope to demonstrate this formally in future work.

In ordinary regular languages, $r^\star$ is equal to $\varepsilon + r \cdot r^\star$. In the language of stream types, this equation says that a stream of type $r^\star$ is either empty (ε) or a stream of type r followed by another stream of type $r^\star$—i.e., $sr\star$ can be understood as the least fixpoint of the stream type operator $x \mapsto \varepsilon + r \cdot x$. The definitions of prefixes and typing rules for star all follow from this perspective.

In particular, $\texttt{prefix}(s^\star) = \texttt{prefix}(\varepsilon + s \cdot s^\star)$. The empty prefix of type $s^\star$, written `starEmp`, is effectively the empty prefix of the sum that makes up $s^\star$. The second form of prefix—the "done" prefix of type $s^\star$—is written `starDone`. It corresponds to the left injection of the sum, and receiving it means that the stream has ended. Note that, despite containing no s data, this prefix is not *empty*: it conveys the information that the stream is complete. The final two cases correspond to the right injection of the sum, i.e., a prefix of type $s \cdot s^\star$. This is either $\texttt{starFirst}(p)$, with p a prefix of s, or $\texttt{starRest}(p, p')$, with p a maximal prefix of type s and p' another prefix of $s^\star$.

For derivatives, the empty prefix leaves the type as-is ($\delta_{\texttt{starEmp}}\left(s^\star\right) = s^\star$). Because no data will arrive after the done prefix, the derivative of $s^\star$ with respect to `starDone` is ε. In the case for $\texttt{starFirst}(p)$, after some of an s has been received, the remainder of $s^\star$ looks like the remainder of the first s followed by some more $s^\star$, so the derivative is defined as $\delta_{\texttt{starFirst}(p)}\left(s^\star\right) = \left(\delta_p\left(s\right)\right) \cdot s^\star$. Finally, $\delta_{\texttt{starRest}(p,p')}\left(s^\star\right) = \delta_{p'}\left(s^\star\right)$.

The typing rules for star are again motivated by the analogy with lists. There are right rules for `nil` and `cons` and a case analysis principle for the left rule. The "nil" rule T-Star-R-1 corresponds to the left injection into the sum $s^\star = \varepsilon + s \cdot s^\star$: from any context, we can produce $s^\star$ by simply ending the stream. The "cons" rule T-Star-R-2 is the right injection: from a context $\Gamma;\Delta$, we can produce an $s^\star$ by producing one s from Γ and the remaining $s^\star$ from Δ.

$$\frac{}{\Gamma \vdash \texttt{nil} : s^\star}\ \text{T-Star-R-1}$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : s^\star}{\Gamma;\Delta \vdash e_1 :: e_2 : s^\star}\ \text{T-Star-R-2}$$

Operationally, this should run the same way as the T-Cat-R rule: by first running e_1, and if an entire s is produced, continuing by running e_2 to produce some prefix of the tail.

The T-Star-L rule is a case analysis principle for streams of star type: either such a stream is empty, or else it comprises one s followed by an $s^\star$. The fact that the head s will come first and the tail $s^\star$ later tells us that the variables $x : s$ and $xs : s^\star$ should be separated by a semicolon in the context. Like T-Sum-L, this rule includes a buffer, collecting input environments until the prefix bound to z is enough to make the decision for which branch of the case to run.

$$\frac{\Gamma(\cdot) \vdash e_1 : r \qquad \Gamma(x : s; xs : s^\star) \vdash e_2 : r \qquad \eta \,:\, \mathsf{env}\left(\Gamma(z : s^\star)\right)}{\delta_\eta\left(\Gamma(z : s^\star)\right) \vdash \mathtt{case}_{s,r}\,(p; z, e_1, x.xs.e_2) : r}\ \text{T-Star-L}$$

The semantics of the right rules are straightforward: the rules for T-Star-R-1 are like those for T-Eps-R, while the rules for T-Star-R-2 are like those for T-Cat-R. The semantics of T-Star-L is just like T-Sum-L, buffering input prefixes until either (a) we get $z \mapsto \mathtt{starDone}$, at which point we run e_1, or (b) we get $z \mapsto \mathtt{starFirst}(p)$ or $z \mapsto \mathtt{starRest}(p, p')$, in which case we run e_2. For full details on the semantics of star, see Appendix A.9.

4.1.3 Let-Binding

Full λ^{ST} also allows for more general let-binding. Given a transformer e whose output is used in the input of another term e', we can compose them to form a single term $\mathtt{let}\ x = e\ \mathtt{in}\ e'$ that operates as the sequential composition of e followed by e'. The rules for this construct are in Figure 4.6 [16]. Note that this sequencing is not the same kind of sequencing as in a concat-pair $(e; e')$. The latter produces data that follows the sequential pattern $s \cdot t$, while the former is sequential composition of code. When a let binding is run, both terms are evaluated, and the output of the first is passed to the input of the second. An important point to note is that this semantics is non-blocking: even if e produces the empty prefix, we still run e', potentially producing output.

The semantic rule S-Let for let-binding (in Figure 4.6) is a straightforward encoding of this behavior.

[16] Traditionally in sequent calculi, this rule, known as "Cut," is introduced only to be immediately shown to be admissible. We expect the cut rule in Kernel λ^{ST} will indeed be admissible. Indeed, Frumin [71] has proven that BI plus an arbitrary set of structural rules admits cut but we have not proven it for Kernel λ^{ST}. Because the point of cut elimination is to enable effective proof search, whereas we are most interested in the calculus from a programming perspective, we will not dwell on this point.

$$\frac{\Delta \vdash e : s \qquad \Gamma(x : s) \vdash e' : t \qquad e \ \texttt{inert}}{\Gamma(\Delta) \vdash \texttt{let}\, x = e\, \texttt{in}\, e' : t}\ \textsc{T-Let}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \eta[x \mapsto p] \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \downarrow \texttt{let}\, x = e_1'\, \texttt{in}\, e_2' \Rightarrow p'}\ \textsc{S-Let}$$

Figure 4.6: Rules for Let-Bindings

Given the input environment η, we run the term e, bind the resulting prefix p to x, and run the continuation e', returning its output. The resultant term is another let-binding between the resultant terms of e and e'[17].

The typing rule T-Let says that if e has type s in context Δ and e' has type t in a context $\Gamma(x : s)$ with a variable of type s, we can form the let-binding term $\texttt{let}\, x = e\, \texttt{in}\, e'$, which has type t in context $\Gamma(\Delta)$. The soundness of the semantics rule S-Let depends on a subtle requirement: e must not produce nonempty output until e' is ready to accept it. This is enforced by the third premise of the T-Let rule, which states that e must be *inert*: it only produces nonempty output when given nonempty input. This restriction rules out let-bindings such as $\texttt{let}\, x = ()\, \texttt{in}\, e'$, since the semantics of `()` always produces nonempty output (namely `oneFull`), even when given an environment mapping every variable to an empty prefix[18]. In actuality, inertness is not a purely syntactic condition on terms, but depends also on typing information. To this end, inertness is tracked like an effect through the type system: see Appendix A.7.0.1 for details.

4.1.4 Recursion

To write interesting transformers over $s^\star$ streams, we provide a way to define transformers recursively. Adding a traditional general recursion operator $\texttt{fix}(x.e)$ does not work in our context, as arrow types are required to define functions this way. We instead add explicit term-level recursion and recursive call operators. The program $\texttt{fix}\ (e_{\texttt{args}})\ .\ (e)$ defines a recursive transformer with body e and initial arguments $e_{\texttt{args}}$. Recursive calls are made inside the body e with a term $\texttt{rec}\ (e_{\texttt{args}})$, which calls the function being defined with arguments $e_{\texttt{args}}$. This back-reference works in the same way that uses of the variable x in

[17] If you're reading this, hi! You've decided to read *very* deep into my dissertation, a choice that demonstrates somewhat questionable taste in reading material. For your dedication, you get this easter egg.

[18] Because such let-bindings are essentially trivial, we expect that they can be eliminated, and hope to investigate this in Future Work

the body of a traditional fix point $\texttt{fix}(x.e)$ refer to the term $\texttt{fix}(x.e)$ itself. This function-free approach is approach is inspired by the concept of *cyclic proofs* [34, 54, 65] from proof theory, where derivations may refer back to themselves. Alternatively, one can think of this construction as defining our terms and proof trees as infinite *coinductive* trees; then the term-level `fix` operator defines terms as *co*fixpoints.

In brief, to typecheck a fixpoint term, we simply type its body e, assuming that all instances of the `rec` in e have the same type as the fixpoint itself. Then, to run a fixpoint term $\texttt{fix}\,\left(e_{\texttt{args}}\right).\,(e)$, the rule unfolds the recursion one step by substituting the body e for instances of `rec` in itself, then runs the resulting term, binding all of the arguments to their variables. Full details of the typing rules and semantics of fixpoints can be found in Appendices A.7 and A.9.

Naturally, this semantics can lead to non-termination, as $\texttt{fix}(\texttt{rec})$ unfolds to itself.[19] To bound the depth of evaluation, we *step index* both semantic judgments by adding a fuel parameter that decreases when we unfold a `fix`. The semantic judgment then looks like $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$: when we run e on η, it steps to e' producing p and unfolding at most n uses of `fix` along the way.

The inclusion of a step index now means that there are well-typed terms about which the λ^{ST} semantics say nothing at all. In particular, an "infinite generator" term $\cdot \vdash_{\texttt{NoRec}} \texttt{fix}(()::\texttt{rec}) : 1^\star$, which runs forever and should produce an infinite sequence of unit values, has no meaning in λ^{ST}. Semanticists may find this behavior odd, but it mimics the incremental semantics of present-day stream processing systems, which wait for a step of computation to terminate before sending out any of its results.

4.1.5 Stateful Transformers

In the λ^{ST} typing judgment $\Gamma \vdash e : s$, the variables in Γ range over *future values* that have yet to arrive at the transformer e. The ordered nature of semicolon contexts means that variables further to the right in Γ correspond to data that will arrive further in the future. This imposes a strong restriction on programming: if earlier values in the stream are used at all, they must be used *before* later values; once a value in the stream has "gone by," there is no way to refer to it again. By using variables from the Γ context, a term e can refer to values that will arrive in the future; but it has no way of referring to values that have arrived in the

[19] Cyclic proof systems usually ensure soundness by imposing a guardedness condition [34] which requires certain rules be applied before a back-edge can be inserted in the derivation tree. Because we are not primarily concerned with λ^{ST} as a logic at the moment, we leave a guardedness condition to future work.

$$\frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash \langle M : s \rangle : s} \text{ T-HistPgm}$$

Figure 4.7: Historical Program Typing Rule

past. This limitation is by design: from a programming perspective, referring to variables from the past requires *memory*, which is a resource to be carefully managed in streaming contexts. Of course, while some important streaming functions (e.g., map and filter) can get by without state, but many others (e.g., "running sums") require it. In this section, we add support for stateful stream transformers.

To maintain state from the past, we extend the typing judgment of λ^{ST} to include a second context, Ω, called the *historical context*, which gives types to variables bound to values stored in memory. We write $\Omega \mid \Gamma \vdash e : s$ to mean "e has type s in context Γ and historical context Ω".

What types do variables in the historical context have? Once a complete stream of type $\left(\texttt{Int}^\star \| \texttt{Int}^\star\right) \cdot \texttt{Int}^\star$ has been received and is stored in memory, we may as well regard the data as a value of the standard type $(\texttt{list(Int)} \times \texttt{list(Int)}) \times \texttt{list(Int)}$ from the simply typed lambda-calculus (STLC). In other words, parts of streams that *will* arrive in the future have stream types, parts of streams that *have* arrived in the past can be given standard STLC types. The "flattening" operation $\langle s \rangle$ transforms stream types into STLC types. The interesting cases of its definition are $\langle s \cdot t \rangle = \langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$ and $\langle s^\star \rangle = \texttt{list}\ (\langle s \rangle)$.

The historical context is a fully structural: $\Omega ::= \cdot \mid \Omega, x : A$, where the types A are drawn from some set of conventional lambda-calculus types including at least products, sums, a unit, and a list type. Operationally, the historical context behaves like a standard context in a functional programming language: at the top level, terms to be run must be typed in an empty historical context; at runtime, historical variables get their values by substitution.

Rather than giving a specific set of ad-hoc rules for manipulating values from the historical context, we parameterize the λ^{ST} calculus over an arbitrary language with terms M, typing judgment $\Omega \vdash M : A$, and big-step semantics $M \downarrow v$. We call any such fixed choice of language the *history language*. Programs from the history language can be embedded in λ^{ST} programs using the T-HistPgm rule in Figure 4.7, which says that a historical program M of type $\Omega \vdash M : \langle s \rangle$ with access the historical context can be used in place of a λ^{ST} term of type s. Operationally, as soon as any prefix of the input arrives, we run the historical program

to completion and yield the result as its stream output (after converting it into a value of type s).

How does information get added to the historical context? Intuitively, a variable in Γ (a stream that will arrive in the future) can be moved to Ω, where streams that have arrived in the past are saved, by waiting for the future to become the past! Formally, we define an operation called "wait," which allows the programmer to specify part of the incoming context and block this subcomputation until that part of the input stream has arrived in full. Once it has, we can bind it to the variables in the historical context and continue by running e.

$$\frac{\Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash e : s}{\Omega \mid \Gamma(x : s) \vdash \texttt{wait}_s(x)\ (e) : s}\ \text{T-Wait-Surf}$$

The T-Wait-Surf rule encodes the typing content of this behavior. It allows us to specify a variable x of the input, flatten its type, and then move it to the historical context, so that the continuation e can refer to it in historical terms. Semantically, this works by buffering in environments until a maximal prefix for x has arrived. Once we have a full prefix for x, we substitute it into e and continue running the resulting term.[20] This buffering is implemented the same way as in the left rules for plus and star, by generalizing the typing rule T-Wait-Surf to a rule T-Wait (found in Appendix A.7) which includes an explicit prefix buffer. As with plus and star, the generalized rule simplifies to the surface rule when the buffer is empty. The generalized rule and the semantics of both the wait and historical program constructs can be found in Appendix A.7. The remaining typing rules in λ^{ST} change only by adding an Ω to the typing judgment everywhere.

Updated Soundness Theorems

Adding recursion and the historical context requires us to update to the soundness theorem from that of Kernel λ^{ST} to Full λ^{ST}. If a well-typed term has (a) closed historical context, and (b) no unbound recursive calls, takes a step on a well-typed input *using some amount of gas*, then the output and resulting term are

[20] The semantics of the T-Wait-Surf rule is reminiscent of the "blocking reads" of Kahn Process Networks, where every read from a parallel stream blocks all other reads to ensure determinism. Here, we choose a variable and block the rest of the program until it is complete and in memory.

also well typed.

Theorem 4.1.1 (Soundness of the λ^{ST} Semantics)**.** *If* $\cdot \mid \Gamma \vdash e : s$, *and* $\eta \ : \ \mathit{env}\,(\Gamma)$, *and* $\eta \Rightarrow e \downarrow^{n} e' \Rightarrow p$, *then* $p \ : \ \mathtt{prefix}\,(s)$ *and* $\cdot \mid \delta_{\eta}\,(\Gamma) \vdash e' : \delta_{p}\,(s)$

A similarly updated statement of the homomorphism theorem can be found in Appendix A.9, and full proofs can be found in the mechanization.

Chapter 5

λ^{Y}, A Functional Calculus with Pull Semantics

In Chapter 1, we set out a definition of stream programming as providing incremental processing of data, with efficient space usage. Then, in Chapters 3 and 4, we defined and formalized Delta, which provides incremental processing in functional programming form, mediated by a bunched & ordered type system. Alas—by our own definition, this would seem to be only half the battle. In this chapter, we attempt to fight the rest of the battle by building a functional language with incremental processing *and* efficient use of memory.

Unfortunately, this goal is significantly loftier than one might anticipate. The previous theoretical development is not merely missing a theorem, and we cannot simply bolt space-usage guarantees onto Delta without restricting the language significantly. It turns out that Delta's semantics evade a theorem bounding the memory of programs for fundamental reasons, and essentially any program with multiple parallel inputs may use unbounded space. Indeed, a quick scan of the operational semantics of Chapter 4 will turn up instances of potentially unbounded run-time state everywhere, from the environments stored on the syntax of case analysis terms, to the unbounded blowup of the terms themselves under unfolding in recursive functions. The culprit here is the *push*-based perspective with which the λ^{ST} semantics operates.

The focus of this section is a change of perspective. Here, we build a new core calculus called λ^{Y} (the superscript in λ^{Y} for Yoink, the name of the prototype language that implements it). Equipped with a *pull*-based semantics, λ^{Y} guarantees that programs run in bounded space. The consequences of moving to a pull-based semantics are manifold. One of the major changes is that it requires us to fundamentally revisit our type system. Indeed, the bunched & ordered type system of λ^{ST} was designed to maintain the invariants of the λ^{ST} semantics, which is a push-based system. It turns out that the static typing required for pull streams is somewhat different.

The type system of λ^{Y} is ordered, but not bunched. Formally, its contexts are partially-ordered sets (*posets*) of variables and their types. Moreover, the order on variables does not correspond to arrival order. In a world where variables range over pull streams rather than push streams, elements do not "arrive"

without warning: they are only produced when requested. As such, the arrival ordering is not determined by the environment, and is instead induced by the program. Instead, the order on variables enforces the correct use of the aliased mutable state encapsulated by the pull streams bound to variables! Intuitively, $x \leq y$ in the context if using y before x would cause x's state to get clobbered.

Another major difference between λ^{ST} and λ^{y} is that in a pull setting, the concept of a "parallel input" collapses somewhat. As such, λ^{y} *does not have* a parallel type, and "parallel" inputs are simply those unordered with respect to each other in the context's partial order.

In terms of expressiveness compared to λ^{ST}, the language design that λ^{y} implies is in certain ways more expressive and in certain ways less. Some programs that are untypeable in λ^{ST} with push streams are perfectly implementable with the pull streams of λ^{y}. On the other hand, the language's recursion is more restricted. Only programs that are "semantically tail recursive" are permitted. Implementing fully general recursive programs requires maintaining a call stack, which need not be bounded in size. Nevertheless, the set of recursive programs that are typable in λ^{y} is larger than those that are tail-recursive in the traditional sense. For instance, common combinators such as `map`, `concat`, and `concatMap` are all examples of typable programs despite not being traditionally tail-recursive [21]. In practice, this restriction boils down to a restriction that one cannot scrutinize or let-bind the result of a recursive call.

An important caveat: some fundamental theorems for λ^{y} remain unproven. Most notably, we lack a semantic type soundness theorem establishing that the type system guarantees the required invariants of the semantics. In Section 5.7.3 we set up the necessary definitions and state this theorem to give the reader a mental model of the invariants in play, but we do not prove it. A formal proof remains future work, and the focus of this chapter is the necessary conceptual development for what comes next. In the following chapter, we will put these concepts to work: first by building a compiler from λ^{y} to imperative code with statically-bounded space usage, and then implementing it all in a new language called Yoink.

5.1 Failure of Constant Space in Push

Let us take a moment to see in detail why Delta (and λ^{ST}) does not guarantee bounded space usage. The most illustrative example of this limitation is the `sync` program, shown below. This function takes a pair

[21] The definition is a little bit closer to tail-mod-cons [157].

of streams `xs` and `ys` and pairs off their elements two-by-two, producing a stream of parallel pairs.

```
fun sync[s,t](xs : s*, ys : t*) : (s || t)* =
case xs of
  nil => nil
| x::xs' => case ys of
              nil => nil
            | y::ys' => wait x,y do
                          {(x,y)} :: sync(xs',ys')
                        end
```

The Delta semantics execute this program sequentially as written. The outer `case` forces the semantics to first wait for the first element of `xs` to arrive (if any) before it can determine which branch to take. Only after observing the cons cell from `xs` does the program proceed to wait for the first element of `ys`. This sequential waiting reflects the program's control flow: the nested cases impose an ordering on when the program is ready to receive data from each stream.

However, the two input streams `xs` and `ys` are independent push streams that arrive in parallel. The Delta runtime environment makes no guarantees about the order in which elements from these streams will arrive. Any interleaving is possible, and the "ideal" arrival order where elements of `xs` and `ys` alternate is by no means guaranteed. Consider what happens when many elements of `ys` arrive before the first element of `xs`: the program is blocked waiting for `xs`, but elements of `ys` keep arriving. These elements cannot simply be discarded—the program will need them later—so they must be buffered.

This buffering behavior is manifest directly in the operational semantics of λ^{ST}. Recall from Chapter 4 that the semantics rules for sum and star case must inspect each arriving event to determine whether it matches the tag at the head of the stream being scrutinized. If the event belongs to a different parallel stream, the semantics cannot simply ignore it. Instead, the event is placed into an in-syntax buffer, and the program continues waiting for the expected tag. This buffer can grow without bound: if one stream produces elements much faster than another, or if the streams happen to arrive in an adversarial order, the amount of buffered data is limited only by the total size of the input.

One might wonder whether this problem could be avoided by restricting the type system further. For instance, we could disallow using any parallel streams in the body of a case expression. But this restriction

is far too severe: it would make `sync` itself untypeable, and the nested case analysis in `sync` is an important pattern.

The core of the problem is that Delta's streams are push-based: data arrives when the producer decides to send it, not when it is convenient for the consumer to receive it. Indeed, the consumer has no control over the arrival order, and must therefore be prepared to buffer parallel data that arrives out of order with respect to its control flow. Intuitively, if we were to instead switch to *pull*-based streams, this problem would be alleviated. With pull streams, the `sync` program could simply pull the first element from `xs`, then pull the first element from `ys`, emit the pair, and repeat. Because the program controls when elements arrive—by explicitly requesting them from an upstream producer—it can ensure that data arrives exactly when needed. No "surprise" elements need to be buffered, because nothing ever arrives before the program is ready for it.

Of course, to make a semantics like this work, it does not suffice to simply replace the input streams in Delta with pull streams. Since variables can be bound to the results of arbitrary computations via let-binding, every term in the language needs to be "pull-able." This requirement inverts our perspective on the entire language. Instead of interpreting terms as push state machines that produce outputs when inputs happen to arrive, they must be interpreted as pull state machines that produce outputs upon request. The downstream consumer, not the upstream producer, drives execution.

In the remainder of this chapter, we develop exactly such a pull-based semantics, and a type system that renders it safe. First, however, we investigate this new state of affairs to understand which of the assumptions from λ^{ST} carry over to the pull setting and which must change. As we shall see, the shift from push to pull does not merely solve the space usage problem—it fundamentally changes the constraints that the type system must enforce.

5.2 State and Imperative Pull Streams

Recall from Chapter 2 that pull streams come in two flavors: functional and imperative. We established that switching from push to pull streams would alleviate the space usage problems of Delta, but we have not yet committed to which kind of pull stream we will use. In this section, we discuss why we choose an imperative treatment of pull streams to achieve bounded space in λ^{y}.

Pull streams are attractive because they bound the amount of data in flight at any given time. This property is necessary for bounded-space stream programming, but it is not sufficient. Switching our semantics to pull eliminates the need to buffer unprocessed inputs, but it does not bound the state used *during* processing. Indeed, a pull stream program must still maintain state: the states of the input streams, any intermediate or intentionally buffered stream values, and the control state of the program itself. If this state can grow without bound, we have not actually solved the problem.

An advantage of giving a semantics in terms of pull streams is that this processing state is at least made explicit. Recall from Chapter 2 that a functional pull stream has the form $\exists s.\, (s, s \to \mathrm{Step}\, s\, a)$: an existentially quantified state type s, an initial state, and a step function that produces either the next element along with a new state, just a new state, or signals termination. The state s directly captures all the information the stream needs to produce its remaining elements. By bounding the size of s, we can bound the space usage of the whole program.

With this in mind, we observe that the *functional* approach to pull streams is a non-starter for guaranteeing bounded space. The problem is two-fold. First, if we give a functional pull stream semantics to Delta terms, each term's state must contain a copy of the states of all the streams it depends on. This poses a problem for elimination forms like $\texttt{let}\ (x; y) = z\ \texttt{in}\ e$. Both x and y must be able to refer to the state of z: pulling on x advances z through its first half, while pulling on y advances z through its second half. But x and y are distinct stream values, both of which need access to z's state. With functional streams, each would require its own copy, and these copies would diverge as soon as either stream is pulled. Second, the step function of a functional pull stream produces new states dynamically. Each call to the step function returns a fresh state value, which must be stored somewhere. In a pure functional setting, this means allocating new memory for each step, significantly complicating the space usage analysis.

These considerations lead us to the conclusion that *imperative* pull streams are the right choice for λ^{y}. With imperative streams, stream programs (and thus the interpretation of terms) hold references to their input streams' states, rather than copies, and calling a step function mutates state. This design directly supports the aliasing we need: in $\texttt{let}\ (x; y) = z\ \texttt{in}\ e$, both x and y hold references to z's state, and pulling on either one mutates that shared state.

5.3 The Typing Constraints of Pull Streams Functions

What stays the same, and what changes, when we move from push to pull?

Consider first the concatenation introduction form $(e; e')$. In the pull setting, when a downstream consumer requests the next element from this term, we pull from e and produce whatever it yields. Once e is exhausted, subsequent requests pull from e' instead. Unlike the corresponding construct in λ^{ST}, this semantics does not require that the variables e depends on and those e' depends on arrive in order. In particular, the term $(x; y)$ is allowed even if x and y are parallel input streams. Because *we* control when to request elements—first exhausting x, then moving on to y—there is no conflict.

The situation is even more interesting for concatenation elimination $\texttt{let}\ (x; y) = z\ \texttt{in}\ e$. How are we allowed to use x and y in the body e? The key observation is that x and y both refer to slices of the same underlying stream z: x refers to the s-typed prefix of z, up to the punctuation mark that separates the two halves, while y refers to the t-typed suffix that follows. Because they both refer to z, they share references to a single mutable state (the state of z). and so in some sense share a single mutable state. As such, pulling on either x or y advances the state of the underlying stream z.

In many cases, this is shared mutable state is unproblematic. For example, the term $\texttt{let}\ (x; y) = z\ \texttt{in}\ x$ simply forwards the first part of z and ignores the rest. Meanwhile, the term $\texttt{let}\ (x; y) = z\ \texttt{in}\ y$ pulls through the entire x portion of z, discarding those elements, and then forwards along the y portion—also fine. Note that this also differs from the push semantics of λ^{ST}, where elements of both x and y would arrive regardless of whether the program uses them; here, only the elements we explicitly request are produced.

But what about $\texttt{let}\ (x; y) = z\ \texttt{in}\ (y; x)$? This is a case where aliased mutable state rears its head. Repeatedly pulling from this this term attempts to first produce all of y, then all of x (suitably punctuated). According to the semantics we have described, producing the first element of y requires pulling through the underlying stream z until we reach the punctuation mark—that is, until the entire x portion has been consumed and discarded. At that point, the state of z has advanced past x, and there is no way to recover the x elements. The subsequent attempt to pull from x would find nothing there. We think of pulling on y as *clobbering* the state that x depends on: getting any element of y requires consuming z past the point where x's data lives. Conversely, pulling on x does not affect y: once x is exhausted, the stream z sits ready

to produce data for y.

The lesson here is that variables with shared underlying state must be used in an order consistent with that state. The term $\texttt{let}\ (x; y) = z\ \texttt{in}\ e$ introduces x and y with a shared dependency on the state of z, and this sharing imposes an ordering constraint: if e uses both x and y, it must use x before y. Using y first would clobber x's state, making x unusable.

The type system for λ^{y} thus must track the state dependencies between variables and ensure they are used in a consistent order. Unsurprisingly, we can formalize this with ordered types! An ordering $x \leq y$ between variables means that x and y may depend on shared state, and pulling on y before finishing with x may preclude any further use of x. The "may" here is important: it allows for order weakening, where we conservatively treat variables as ordered even when they happen to have independent state.

It is worth pausing to note just how different this is from λ^{ST}. In λ^{ST}, the concatenation term $(e; e')$ requires that the inputs for e arrive before those for e'—this is an arrival-time constraint imposed by the push semantics. In λ^{y}, the constraint is instead about state consistency: the inputs for e' must not depend on state that would be clobbered by pulling on them before e's inputs. Variables that are unordered in the input may be freely sequenced by the program, since the program controls when to pull, but variables that share underlying state must be used in the order dictated by that sharing.

5.4 The λ^{y} Type System

We now formalize the intuitions from the previous section into the λ^{y} type system. In λ^{ST}, the ordering structure on variables is encoded implicitly through the two connectives that make up its bunched context: the comma separates parts of the context that arrive in parallel, while the semicolon indicates parts that arrive in sequence. This approach works well for push streams, where the parallel inputs *must* be processed independently, and so having an explicit comma context former is important to ensure that programs respect this independence. For the pull streams in λ^{y}, the situation is more flexible. Because parallel inputs can be used in any order, we do not need to track any explicit *lack* of ordering in the context: all that matters is which variable orderings *do* exist, to ensure we do not use them out of order. Neither bunched contexts nor fully ordered contexts capture this asymmetry.

Instead, the orderings on variables are tracked in a partially ordered set (poset), separately from the

context that encodes their types. Below, we review some preliminary definitions related to partial orders.

5.4.1 Poset Definitions

Definition 5.4.1 (Poset)**.** *A* poset P *is a finite partially ordered set over variables. We write* $|P|$ *for the underlying set of variables in* P*, and* $x \leq_P y$ *to indicate that* x *precedes* y *in* P*.*

Definition 5.4.2 (Poset Inclusion)**.** *We write* $P \lesssim Q$ *when* $|P| = |Q|$ *and for all* $x, y \in |P|$*, if* $x \leq_P y$ *then* $x \leq_Q y$*. That is,* P *has the same variables as* Q *but potentially fewer ordering constraints.*

Definition 5.4.3 (Edge Substitution)**.** *Given a poset* P *with* $z \in |P|$*, we define* $P[(x \leq y)/z]$ *as the poset where:*

- *The underlying set is* $(|P| \setminus \{z\}) \cup \{x, y\}$
- *The ordering is the transitive closure of:*
  - $x \leq y$
  - *For all* $z' \in |P|$*: if* $z' \leq_P z$ *then* $z' \leq x$
  - *For all* $z' \in |P|$*: if* $z \leq_P z'$ *then* $y \leq z'$
  - *All orderings from* P *not involving* z

The fresh variables x *and* y *jointly take the place of* z*, with* x *inheriting* z*'s lower bounds and* y *inheriting* z*'s upper bounds.*

Definition 5.4.4 (Poset Substitution)**.** *Given posets* P *and* Q *with* $z \in |P|$*, we define* $P[z/Q]$ *as the poset where:*

- *The underlying set is* $(|P| \setminus \{z\}) \cup |Q|$
- *The ordering is the transitive closure of:*
  - *All orderings internal to* Q
  - *For all* $z' \in |P|$ *and* $q \in |Q|$*: if* $z' \leq_P z$ *then* $z' \leq q$
  - *For all* $z' \in |P|$ *and* $q \in |Q|$*: if* $z \leq_P z'$ *then* $q \leq z'$

– *All orderings from P not involving z*

This is similar to edge substitution: the poset Q is "spliced in" where z was, inheriting all of z's external relationships.

Definition 5.4.5 (Poset Concatenation)**.** *Given posets P_1 and P_2 with disjoint underlying sets, we define $P_1 \cdot P_2$ as the poset where:*

- *The underlying set is $|P_1| \cup |P_2|$*
- *The ordering is: $x \leq_{P_1 \cdot P_2} y$ iff $x \leq_{P_1} y$, or $x \leq_{P_2} y$, or $x \in |P_1|$ and $y \in |P_2|$*

That is, all elements of P_1 precede all elements of P_2.

5.4.2 Type System

λ^y tracks variable orderings separately in the typing judgment with a partial order. Formally, our typing contexts come in two halves, written $\Gamma \mid P$. The first component Γ is a standard typing context—a list of variables and their types—with no ordering structure. The second component P is a partial order over the variables of Γ, recording the ordering dependencies between variables. Intuitively, variables ordered $x \leq y$ in P are those that might[22] share state in a way that requires x to be pulled before y to avoid clobbering. Variables that are not ordered with respect to each other in the ordering are guaranteed to not have any state dependencies, and hence all interleavings of of uses are safe.

The top level typing judgment looks like:

$$\Gamma \mid P \vdash_\Sigma e : s$$

The $\Gamma \mid P$ is a two-part typing context as described above, e is the term being typed, and s is its type. The Σ on the turnstile is a recursion signature. Like in λ^{ST}, the λ^y type system tracks the type of the recursive function that the being-typed term is a subterm of, if any. Formally, a recursion signature is either empty, or a "function type":

$$\Sigma ::= \emptyset \mid (\Gamma \mid P \to s)$$

[22] "Might", because of weakening — we allow for unneeded orderings to be added to the partial order.

.

We present the rules of the λ^y type system in Figure 5.1.

Variables The variable rule Tp-Var is straightforward. It says that if $x : s$ is a variable in the context—listed both in the typing context with the correct type, and also in the order P—then we can produce it as our output. Semantically, this term responds to requests to output by pulling on the stream bound to x, and forwarding the results along.

Cat The rule Tp-Cat-R builds a stream of type $s \cdot t$ from $e_1 : s$ and $e_2 : t$, allowing e_1 access to variables as dictated by $\Gamma_1 \mid P_1$, and e_2 as $\Gamma_2 \mid P_2$. The critical premise is $P \lesssim P_1 \cdot P_2$ (Definitions 5.4.2 and 5.4.5), which says that the input variable ordering P must be *at most as ordered* as $P_1 \cdot P_2$. That is, we cannot have any orderings $x \leq_P y$ where $x \in |P_2|$ and $y \in |P_1|$. Such an ordering would say that x and y share state in a way that requires x to be pulled first, but the semantics of $(e_1; e_2)$ pulls e_1 (which in turn pulls on y) before e_2 (which pulls on x). Orderings in the other direction—the direction consistent with the P_1 then P_2—are fine, as they ensure the shared state is advanced in the correct order. This premise also allows for *fewer* orderings than the "complete" ordering from P_1 to P_2, and so e_1 and e_2 may refer to variables that are unordered with respect to each other. This last point is a key difference from λ^{ST}, where variable referring to parallel data cannot appear on different sides of a concatenation intro term.

The rule Tp-Cat-L is the elimination form. Binding-wise, it does what you would expect: given a variable z of type $s \cdot t$, it destructs it into two variables $x : s$ and $y : t$. The important thing is the way it manipulates the ordering. The body e is typed with the ordering $P[(x \leq y)/z]$ (Definition 5.4.3), which adds an ordering $x \leq y$ and ensures that x and y play the same role as z in relation to other variables. Adding $x \leq y$ is what ensures (in concert with Tp-Cat-R's ordering constraint) that we cannot type $\texttt{let}\ (x; y) = z\ \texttt{in}\ (y; x)$. Meanwhile, adding x and y in the same "place" as z in the ordering ensures that state-sharing constraints propagate correctly. For example, if z shares state with some other variable z' in a way that requires $z \leq z'$, we cannot write $(z'; x)$, as this would implicitly clobber x through its dependence on z.

Sums The introduction rules Tp-Inl and Tp-Inr are standard: they inject a stream of type s or t into the sum type $s + t$ without changing the typing context. The elimination rule Tp-SumCase does case analysis on the first element of the stream z, which determines whether the rest of the stream has type s or t. The

$$\frac{x \in |P|}{\Gamma, x : s \mid P \vdash_\Sigma x : s}\ \textsc{Tp-Var} \qquad \frac{\Gamma_1 \mid P_1 \vdash_\Sigma e_1 : s \qquad \Gamma_2 \mid P_2 \vdash_\Sigma e_2 : t \qquad P \lesssim P_1 \cdot P_2}{\Gamma_1, \Gamma_2 \mid P \vdash_\Sigma (e_1; e_2) : s \cdot t}\ \textsc{Tp-Cat-R}$$

$$\frac{\Gamma, x : s, y : t \mid P[(x \leq y)/z] \vdash_\Sigma e : r}{\Gamma, z : s \cdot t \mid P \vdash_\Sigma \mathtt{let}\ (x; y) = z\ \mathtt{in}\ e : r}\ \textsc{Tp-Cat-L} \qquad \frac{\Gamma \mid P' \vdash_\Sigma e : s \qquad \Gamma, x : s \mid P[x/P'] \vdash_\Sigma e' : t}{\Gamma \mid P \vdash_\Sigma \mathtt{let}\, x = e\, \mathtt{in}\, e' : t}\ \textsc{Tp-Let}$$

$$\frac{\Gamma \mid P \vdash_\Sigma e : s}{\Gamma \mid P \vdash_\Sigma \mathtt{inl}\,(e) : s + t}\ \textsc{Tp-Inl} \qquad \frac{\Gamma \mid P \vdash_\Sigma e : t}{\Gamma \mid P \vdash_\Sigma \mathtt{inr}\,(e) : s + t}\ \textsc{Tp-Inr}$$

$$\frac{\Gamma, x : s \mid P[x/z] \vdash_\Sigma e_1 : r \qquad \Gamma, y : t \mid P[y/z] \vdash_\Sigma e_2 : r}{\Gamma, x : s + t \mid P \vdash_\Sigma \mathtt{case}\,(z, x.e_1, y.e_2) : r}\ \textsc{Tp-SumCase} \qquad \frac{}{\Gamma \mid \emptyset \vdash_\Sigma \mathtt{nil} : s^\star}\ \textsc{Tp-Nil}$$

$$\frac{\Gamma_1 \mid P_1 \vdash_\Sigma e_1 : s \qquad \Gamma_2 \mid P_2 \vdash_\Sigma e_2 : s^\star \qquad P \lesssim P_1 \cdot P_2}{\Gamma_1, \Gamma_2 \mid P \vdash_\Sigma e_1 :: e_2 : s^\star}\ \textsc{Tp-Cons}$$

$$\frac{\Gamma \mid P \setminus z \vdash_\Sigma e_1 : r \qquad \Gamma, x : s, y : s^\star \mid P[(x \leq y)/z] \vdash_\Sigma e_2 : r}{\Gamma, z : s^\star \mid P \vdash_\Sigma \mathtt{case}\,(z, e_1, x.y.e_2) : r}\ \textsc{Tp-StarCase} \qquad \frac{}{\Gamma \mid P \vdash_\Sigma \mathtt{eps} : \varepsilon}\ \textsc{Tp-Eps}$$

$$\frac{}{\Gamma \mid P \vdash_\Sigma n : \mathtt{Int}}\ \textsc{Tp-Int} \qquad \frac{\Gamma \mid P \vdash_{\Gamma | P \to s} e : s \qquad \mathtt{semTailRec}\,(e)}{\Gamma \mid P \vdash_\Sigma \mathtt{fix}\,(e) : s}\ \textsc{Tp-Fix} \qquad \frac{P \lesssim P'}{\Gamma \mid P \vdash_{\Gamma | P' \to s} \mathtt{rec} : s}\ \textsc{Tp-Rec}$$

$$\frac{\Gamma \mid P' \vdash_\Sigma e : s \qquad P \lesssim P'}{\Gamma \mid P \vdash_\Sigma e : s}\ \textsc{Tp-Weaken-Order} \qquad \frac{\Gamma \mid P \setminus \{x\} \vdash_\Sigma e : t}{\Gamma, x : s \mid P \vdash_\Sigma e : t}\ \textsc{Tp-Weaken-Var}$$

$$\frac{\Gamma, y : t, x : s, \Gamma' \mid P \vdash_\Sigma e : r}{\Gamma, x : s, y : t, \Gamma' \mid P \vdash_\Sigma e : r}\ \textsc{Tp-Exchange}$$

Figure 5.1: Typing Rules of λ^{y}

bound variables x and y in the two branches are substituted (Definition 5.4.4) into the partial order in place of z. This is because the streams bound to x and y *are* the stream bound to z (less the first punctuation), and so they must play the same role in the ordering.

Epsilon and Singleton The rules Tp-Eps and Tp-Int are straightforward: they produce streams of type ε and `Int` respectively, and do not interact with the ordering at all. The `eps` term produces an empty stream that immediately terminates when you pull on it, while an integer literal n yields a singleton stream that produces only n on the first pull, and terminates after.

Star The rules for star combine aspects of the rules for sums and concatenation. Tp-Nil produces the nil stream of type $s^\star$, with typing identical to the rule for `eps`. The rule Tp-Cons is similar to Tp-Cat-R because the semantics are essentially identical: we run e_1 `::` e_2 by first pulling from e_1, then from e_2 once the first is complete. As with concatenation, the premise $P \lesssim P_1 \cdot P_2$ ensures that variables in e_2 do not come before those in e_1 in the ordering.

The elimination rule Tp-StarCase combines Tp-SumCase and Tp-Cat-L. Like sum elimination, it does case analysis on the first element of the stream z, branching on whether the stream is empty or a cons. Like Tp-Cat-L, the cons branch binds two variables x and y, imposing the ordering constraint $P[(x \leq y)/z]$ on the continuation. This is for an analogous reason to concatenation elimination: x is the head of z and y is the tail, and so they both share share the underlying state of z. Meanwhile, the nil branch of the case takes no arguments, and is typed in a context without z.

Recursion Recursion in λ^y follows a similar structure to λ^{ST}: we have judgment-level recursion with `fix` $(-)$ and `rec`, which together allow us to define recursive functions. However, recursion in λ^y is more restricted than in λ^{ST}. In λ^{ST}, the semantics substitute the entire term for `rec` when evaluation reaches the `fix` $(-)$, in effect building up an arbitrarily large term whose evaluation context serves as a call stack. We cannot do this in λ^y, as these arbitrarily large call stacks would need to be materialized as unbounded state in the pull stream's state machine. Instead, we restrict ourselves to tail recursion, which requires only constant stack space. Essentially, recursion in λ^y operates like a "jump back" from the `rec` to the `fix` $(-)$. This is implemented by resetting the relevant bits of the pull stream's state machine back to its initial state at the point of the fix. We discuss this in more detail in Section 5.7.

This jump-based recursion means that some patterns of recursion that were meaningful in λ^{ST} are

disallowed in λ^{y}. For example, any scrutinizing of a recursive call (by way of concatenation or star elimination) is disallowed, since we will never "come back" from the recursive call to perform the scrutiny. Similarly, using a recursive call in the e_1 position of a $(e_1; e_2)$ is disallowed, as we would never return to pull from the e_2 portion.

In effect, this restriction gives us a strong form of tail recursion. Indeed, recursive calls do not have to syntactically be in tail position in the traditional functional programming sense—i.e., `map` is definable with the standard code where you write $f\ x$ `::` `rec`—but they do have to be in tail position with respect to the pull-based semantics. We explore this further in Section 5.7.

This tail position constraint is enforced in the typing judgment with three mutually-dependent syntactic judgments (Figures 5.2 and 5.3):

- `nonRec` (e) holds when e contains no recursive calls to the current `fix` $(-)$ at all.
- `tailVar` (x, e) holds when x is only used in tail positions within e—that is, x is never scrutinized by a case analysis and never appears in the first position of a $(e_1; e_2)$ or e_1 `::` e_2.
- `semTailRec` (e) holds when all recursive calls in e appear in valid tail positions. This judgment uses `nonRec` (e) and `tailVar` (x, e) as auxiliaries.

The term typing rules for recursion are Tp-Fix and Tp-Rec. A recursive function is defined with `fix` (e), which records the current context and result type in the recursion signature, and then types the body e under this signature. Importantly, Tp-Fix also enforces the `semTailRec` (e) condition on the body to ensure that all recursive calls are in valid positions. Inside the body of a `fix` $(-)$, recursive calls are made with `rec`, typed by Tp-Rec.

Structural Rules There are three structural rules. First, Tp-Weaken-Var is standard variable weakening: we can drop variables from the context. Next, Tp-Weaken-Order allows weakening by imposing additional orderings. If $P \lesssim P'$, then to type e with orderings P, it suffices to type it with a stricter ordering P'. Last, Tp-Exchange is exchange: we can permute variables in the context Γ. We include this rule to emphasize that the Γ portion of the context carries no ordering information—all ordering is controlled by P.

Rules for `nonRec` (e):

$$\frac{}{\texttt{nonRec}\ (x)}\ \text{NR-Var} \qquad \frac{}{\texttt{nonRec}\ (\texttt{eps})}\ \text{NR-Eps} \qquad \frac{}{\texttt{nonRec}\ (n)}\ \text{NR-Int} \qquad \frac{}{\texttt{nonRec}\ (\texttt{nil})}\ \text{NR-Nil} \qquad \frac{}{\texttt{nonRec}\ (\texttt{fix}\ (e))}\ \text{NR-Fix}$$

$$\frac{\texttt{nonRec}\ (e)}{\texttt{nonRec}\ (\texttt{inl}\ (e))}\ \text{NR-Inl} \qquad \frac{\texttt{nonRec}\ (e)}{\texttt{nonRec}\ (\texttt{inr}\ (e))}\ \text{NR-Inr} \qquad \frac{\texttt{nonRec}\ (e_1) \qquad \texttt{nonRec}\ (e_2)}{\texttt{nonRec}\ ((e_1; e_2))}\ \text{NR-Cat-R}$$

$$\frac{\texttt{nonRec}\ (e)}{\texttt{nonRec}\ (\texttt{let}\ (x, y) = z\ \texttt{in}\ e)}\ \text{NR-Cat-L}$$

$$\frac{\texttt{nonRec}\ (e) \qquad \texttt{nonRec}\ (e')}{\texttt{nonRec}\ (\texttt{let}\ x = e\ \texttt{in}\ e')}\ \text{NR-Let} \qquad \frac{\texttt{nonRec}\ (e_1) \qquad \texttt{nonRec}\ (e_2)}{\texttt{nonRec}\ (\texttt{case}\ (z, x.e_1, y.e_2))}\ \text{NR-SumCase}$$

$$\frac{\texttt{nonRec}\ (e_1) \qquad \texttt{nonRec}\ (e_2)}{\texttt{nonRec}\ (e_1 :: e_2)}\ \text{NR-Cons} \qquad \frac{\texttt{nonRec}\ (e_1) \qquad \texttt{nonRec}\ (e_2)}{\texttt{nonRec}\ (\texttt{case}\ (z, e_1, x.y.e_2))}\ \text{NR-StarCase}$$

Rules for `tailVar` (x, e):

$$\frac{}{\texttt{tailVar}\ (x, y)}\ \text{TV-Var} \qquad \frac{}{\texttt{tailVar}\ (x, \texttt{eps})}\ \text{TV-Eps} \qquad \frac{}{\texttt{tailVar}\ (x, n)}\ \text{TV-Int} \qquad \frac{}{\texttt{tailVar}\ (x, \texttt{nil})}\ \text{TV-Nil}$$

$$\frac{\texttt{tailVar}\ (x, e)}{\texttt{tailVar}\ (x, \texttt{fix}\ (e))}\ \text{TV-Fix} \qquad \frac{}{\texttt{tailVar}\ (x, \texttt{rec})}\ \text{TV-Rec}$$

$$\frac{\texttt{tailVar}\ (x, e)}{\texttt{tailVar}\ (x, \texttt{inl}\ (e))}\ \text{TV-Inl} \qquad \frac{\texttt{tailVar}\ (x, e)}{\texttt{tailVar}\ (x, \texttt{inr}\ (e))}\ \text{TV-Inr} \qquad \frac{x \notin \texttt{fv}\ (e_1) \qquad \texttt{tailVar}\ (x, e_2)}{\texttt{tailVar}\ (x, (e_1; e_2))}\ \text{TV-Cat-R}$$

$$\frac{x \neq z \qquad \texttt{tailVar}\ (x, e)}{\texttt{tailVar}\ (x, \texttt{let}\ (y_1, y_2) = z\ \texttt{in}\ e)}\ \text{TV-Cat-L}$$

$$\frac{\texttt{tailVar}\ (x, e) \qquad \texttt{tailVar}\ (x, e')}{\texttt{tailVar}\ (x, \texttt{let}\ y = e\ \texttt{in}\ e')}\ \text{TV-Let} \qquad \frac{x \neq z \qquad \texttt{tailVar}\ (x, e_1) \qquad \texttt{tailVar}\ (x, e_2)}{\texttt{tailVar}\ (x, \texttt{case}\ (z, y_1.e_1, y_2.e_2))}\ \text{TV-SumCase}$$

$$\frac{x \notin \texttt{fv}\ (e_1) \qquad \texttt{tailVar}\ (x, e_2)}{\texttt{tailVar}\ (x, e_1 :: e_2)}\ \text{TV-Cons} \qquad \frac{x \neq z \qquad \texttt{tailVar}\ (x, e_1) \qquad \texttt{tailVar}\ (x, e_2)}{\texttt{tailVar}\ (x, \texttt{case}\ (z, e_1, y_1.y_2.e_2))}\ \text{TV-StarCase}$$

Figure 5.2: Recursion Control Rules (Part 1)

Rules for `semTailRec` (e):

$$\frac{}{\texttt{semTailRec}\,(x)}\ \text{TR-Var} \qquad \frac{}{\texttt{semTailRec}\,(\texttt{eps})}\ \text{TR-Eps} \qquad \frac{}{\texttt{semTailRec}\,(n)}\ \text{TR-Int} \qquad \frac{}{\texttt{semTailRec}\,(\texttt{nil})}\ \text{TR-Nil}$$

$$\frac{}{\texttt{semTailRec}\,(\texttt{rec})}\ \text{TR-Rec} \qquad \frac{\texttt{semTailRec}\,(e)}{\texttt{semTailRec}\,(\texttt{fix}\,(e))}\ \text{TR-Fix}$$

$$\frac{\texttt{semTailRec}\,(e)}{\texttt{semTailRec}\,(\texttt{inl}\,(e))}\ \text{TR-Inl} \qquad \frac{\texttt{semTailRec}\,(e)}{\texttt{semTailRec}\,(\texttt{inr}\,(e))}\ \text{TR-Inr} \qquad \frac{\texttt{nonRec}\,(e_1) \qquad \texttt{semTailRec}\,(e_2)}{\texttt{semTailRec}\,((e_1; e_2))}\ \text{TR-Cat-R}$$

$$\frac{\texttt{semTailRec}\,(e)}{\texttt{semTailRec}\,(\texttt{let}\,(x, y) = z\,\texttt{in}\,e)}\ \text{TR-Cat-L}$$

$$\frac{(\texttt{nonRec}\,(e) \vee \texttt{tailVar}\,(x, e')) \qquad \texttt{semTailRec}\,(e')}{\texttt{semTailRec}\,(\texttt{let}\,x = e\,\texttt{in}\,e')}\ \text{TR-Let}$$

$$\frac{\texttt{semTailRec}\,(e_1) \qquad \texttt{semTailRec}\,(e_2)}{\texttt{semTailRec}\,(\texttt{case}\,(z, x.e_1, y.e_2))}\ \text{TR-SumCase} \qquad \frac{\texttt{nonRec}\,(e_1) \qquad \texttt{semTailRec}\,(e_2)}{\texttt{semTailRec}\,(e_1 :: e_2)}\ \text{TR-Cons}$$

$$\frac{\texttt{semTailRec}\,(e_1) \qquad \texttt{semTailRec}\,(e_2)}{\texttt{semTailRec}\,(\texttt{case}\,(z, e_1, x.y.e_2))}\ \text{TR-StarCase}$$

Figure 5.3: Recursion Control Rules (Part 2)

5.5 Events

What are the *values* of $\lambda^{\mathcal{Y}}$? In λ^{ST}, the values are prefixes: finite chunks of a stream that have arrived since the last step. This makes sense in a push setting, where data arrives and must be processed as it comes. In a pull setting like $\lambda^{\mathcal{Y}}$, however, we want to produce the smallest possible response to a request—in essence, prefixes of "size one."

To this end, we define *events*. An event of stream type s is the head of a stream of that type: a possible first element that could be produced when you pull. The grammar of events is:

$$x ::= \texttt{baseev}\,(n) \mid \texttt{PlusA} \mid \texttt{PlusB} \mid \texttt{CatPunc} \mid \texttt{CatEv}\,(x)$$

Like prefixes in λ^{ST}, events have a typing relation $x : \texttt{event}\,(s)$ (Definition 5.5.1) and a derivative operation. For an event x of type s, the derivative $\delta_x\,(s) \sim s'$ gives the type s' of the stream that remains after pulling x. This lets us define well-typed event sequences (Definition 5.5.3): a sequence is well-typed for s when the first event has type s, and the rest of the sequence is well-typed for its derivative.

Definition 5.5.1 (Event Typing Relation)**.** *We define a binary relation* $x : \texttt{event}\,(s)$ *as follows:*

$$\frac{}{\texttt{baseev}\,(n) : \texttt{event}\,(\texttt{Int})}$$

$$\frac{}{\texttt{PlusA} : \texttt{event}\,(s + t)} \qquad \frac{}{\texttt{PlusB} : \texttt{event}\,(s + t)} \qquad \frac{}{\texttt{CatPunc} : \texttt{event}\,(\varepsilon \cdot t)}$$

$$\frac{x : \texttt{event}\,(s)}{\texttt{CatEv}\,(x) : \texttt{event}\,(s \cdot t)} \qquad \frac{}{\texttt{PlusA} : \texttt{event}\,(s^\star)} \qquad \frac{}{\texttt{PlusB} : \texttt{event}\,(s^\star)}$$

Note that $s + t$ and $s^\star$ share the same punctuation events. Intuitively, this is because $s^\star$ can be unrolled as $\varepsilon + (s \cdot s^\star)$: we reuse the events of the latter for the former.

Definition 5.5.2 (Event Derivative Relation)**.** *We define a ternary relation* $\delta_x\,(s) \sim s'$.

$$\frac{}{\delta_{baseev(n)}\,(\mathit{Int}) \sim \varepsilon} \qquad \frac{}{\delta_{PlusA}\,(s+t) \sim s}$$

$$\frac{}{\delta_{PlusB}\,(s+t) \sim t} \qquad \frac{}{\delta_{CatPunc}\,(\varepsilon \cdot t) \sim t} \qquad \frac{\delta_x\,(s) \sim s'}{\delta_{CatEv(x)}\,(s \cdot t) \sim s' \cdot t} \qquad \frac{}{\delta_{PlusA}\,(s^\star) \sim \varepsilon}$$

$$\frac{}{\delta_{PlusB}\,(s^\star) \sim s \cdot s^\star}$$

Definition 5.5.3 (Event Sequence)**.**

$$\frac{}{[\,] : \mathit{events}\,(s)}\ \textit{ES-Nil} \qquad \frac{x : \mathit{event}\,(s) \qquad \delta_x\,(s) \sim s' \qquad xs : \mathit{events}\,(s')}{x :: xs : \mathit{events}\,(s)}\ \textit{ES-Cons}$$

Unfolding these definitions, we get the following event sequence structures for each stream type:

- **`Int`:** A stream of type `Int` is a one-element sequence $[\texttt{baseev}\,(n)]$.
- **Sum Type $s + t$:** A stream of type $s + t$ is either $\texttt{PlusA}::xs$, where $xs : \texttt{events}\,(s)$, or $\texttt{PlusB}::ys$, where $ys : \texttt{events}\,(t)$.
- **Cat Type $s \cdot t$:** A stream of type $s \cdot t$ is a sequence

$$\texttt{CatEv}\,(x_1), \ldots, \texttt{CatEv}\,(x_n), \texttt{CatPunc}, y_1, \ldots, y_n$$

  where $\overline{x_i} : \texttt{events}\,(s)$ and $\overline{y_i} : \texttt{events}\,(t)$.
- **Star Type $s^\star$:** A stream of type $s^\star$ is either $[\texttt{PlusA}]$, or $\texttt{PlusB}::zs$, where $zs : \texttt{events}\,(s \cdot s^\star)$.

5.6 Pull Streams

In the formalism of this chapter, a pull stream is a pair consisting of a state of type X, and a function $X \to \texttt{Step}$. A *step* is either `done`, `skip`, or `yield`. The step $\texttt{done}\,(x)$ signals that the stream has no more events to produce, and the final stream state is x. The step $\texttt{skip}\,(x)$ means the state has updated (new state x), but no event has been produced. The step $\texttt{yield}\,(e, x)$ means an event e has been produced, and the state of the stream has been updated to x. We note that because this is an on-paper mathematization of pull streams, it is intrinsically functional—we must thread the state through explicitly. The "imperativeness" of our semantics will come from the fact that there is one global state that different parts of the program may modify, and we use it linearly. In Chapter 6, we will compile λ^{γ} to programs that actually update their state imperatively.

Definition 5.6.1 (Pull Stream)**.** *A pull stream is a pair $\langle x_0 : X, f : X \to \texttt{Step}\rangle$, for some type X. `Step` is the set defined by the following grammar*

$$st \quad ::= \quad \texttt{done}\,(x) \mid \texttt{skip}\,(x) \mid \texttt{yield}\,(e, x)$$

A pull stream has type s if it produces a sequence of events of type s, as in Definition 5.5.3. Because pulling on the stream may in principle only ever produce `skip`s, we step-index this definition and write $\langle x, f\rangle \vDash^n s$ to mean that the stream $\langle x, f\rangle$ behaves like one of type s for n steps.

Definition 5.6.2 (Well Typed Pull Stream)**.** *We define what it means for a pull stream $\langle x, f\rangle$ has type s for*

n steps, written $\langle x, f\rangle \vDash^n s$ *by the following judgment:*

$$\frac{}{\langle x, f\rangle \vDash^0 s}$$

$$\frac{fx = \mathit{done}\,(\_)}{\langle x, f\rangle \vDash^n \varepsilon}$$

$$\frac{fx = \mathit{skip}\,(x') \qquad \langle x', f\rangle \vDash^n s}{\langle x, f\rangle \vDash^{n+1} s}$$

$$\frac{fx = \mathit{yield}\,(e, x') \qquad \delta_e\,(s) \sim s' \qquad \langle x', f\rangle \vDash^n s'}{\langle x, f\rangle \vDash^{n+1} s}$$

5.7 Semantics

We now give a semantics for λ^y in terms of pull streams. This semantics proceeds in two parts, corresponding to the two components of a pull stream $\langle x, f\rangle$. First, we translate λ^y terms into stream states—the x component of the pull stream. These states are *pull graphs*: directed graphs over a special set of nodes. Nodes in the graph correspond loosely to subterms of a λ^y term, though the graph structure makes the sharing manifest: programs that share state translate to subgraphs that share nodes. Second, we define a step function `step`—the f component of the pull stream—that takes a graph and reference to one of the nodes, and produces an updated graph. Importantly, the edge structure of the graph never changes; only the mutable state stored at each node is updated.

This construction is designed to maintain the following invariant. If we take an open term $x : s \mid \{x\} \vdash e : t$ and translate it to a graph G with start node n, then for all k, if we plug in a stream that behaves like type s for k steps (i.e., $\langle x_0, f\rangle \vDash^k s$) to obtain graph G', then the resulting stream behaves like type t for k steps: $\langle (G', n), \mathtt{step}\rangle \vDash^k t$. We give a more formal version of this theorem later (Theorem 5.7.1), though we note that this theorem remains unproven—we hope to establish it in future work.

5.7.1 Building Pull Graphs

Definition 5.7.1 (Pull Graph)**.** *A* pull graph *is a pair* (G, n_{start}) *where:*

- G *is a finite set of bindings* $n \Rightarrow \mathtt{N}$*, where* n *is a* node identifier *(node-id) and* $\mathtt{N}$ *is a* pull node
- n_{start} *is a distinguished* start node *identifying the root of the computation.*

We write $G[n] \Rightarrow \mathtt{N}$ to mean that $n \Rightarrow \mathtt{N}$ is in G, and $G[n \Rightarrow \mathtt{N}']$ to denote the graph obtained by updating the binding for n to $\mathtt{N}'$.

Definition 5.7.2 (Pull Nodes)**.** *Pull nodes are pieces of syntax drawn from the following grammar:*

$$
\begin{aligned}
N \;::=\;& \mathtt{CatRNode}\,(b, n, n) \mid \mathtt{CatCoordNode}\,(b, n) \mid \mathtt{CatProjNode}_1\,(n) \mid \mathtt{CatProjNode}_2\,(n) \\
\mid\;& \mathtt{InLNode}\,(b, n) \mid \mathtt{InRNode}\,(b, n) \mid \mathtt{CaseNode}\,(k, n, n, n) \\
\mid\;& \mathtt{EpsNode} \mid \mathtt{IntNode}\,(b, k) \\
\mid\;& \mathtt{FixNode}\,(n) \mid \mathtt{RecNode}\,(n) \\
\mid\;& \mathtt{SourceNode}\,(\langle x, f \rangle)
\end{aligned}
$$

Each node type has some number of fields of various types. The simplest field type is a node-id field. For example, the node $n \Rightarrow \mathtt{CatProjNode}_1\,(n')$ has a node-id field storing n'. These fields specify the edge structure of the graph: the node n has an outgoing edge to n'. Conceptually, pulling on a node n to request some output might produce further requests to nodes along its outgoing edges. Nodes may also have data fields, storing booleans b or integers k. These fields are mutable state that tells the step function where we are in execution, and what should happen next. Importantly, only these fields—never the node-id fields—are changed at run time, and so the structure of the graph remains constant. Source nodes—written $\mathtt{SourceNode}\,(\langle x, f \rangle)$—are special, as they represent the inputs of a stream graph, and maintain an arbitrary pull stream $\langle x, f \rangle$. During execution, only the state x will change.

The translation of λ^y terms into pull graphs is defined by the judgment $N, R \vdash [\![e]\!] \rightsquigarrow (G, n)$. Intuitively, this judgment states that the stream state for term e is represented by the pull graph (G, n). The context N is a map from free term variables to node-ids to resolve variable references during translation. The R in the judgment is either a node-id r — a reference to the definition site of the innermost enclosing fix-

point of the current term — or a "$\cdot$" — indicating that we are not currently building the state of a recursive function. At the top level, the map N is initially populated with bindings $x_i \mapsto n_i$ for all free variables in the term e to be transalted. After we have completed translating e to a graph (G, n), we add source nodes $\bigcup\{n_i \Rightarrow \texttt{SourceNode}\,(\langle x_{0i}, f_i\rangle)\}$ corresponding to each input stream. The rules of the judgment, shown in Figure 5.4, define the translation by walking λ^{γ} terms. At each subterm, we allocate one or more fresh node-ids, make recursive calls to build subgraphs for the subterms, and then combine these subgraphs with new nodes for the top-level term itself.

Var The variable rule SG-Var is trivial: to translate a variable reference, we simply look up the node identifier that the variable is bound to in the environment N. This adds no new nodes to the graph, as the node-id already points to an existing node, either constructed by another rule, or passed in as a source node.

Cat Right In rule SG-Cat-R, we recursively build subgraphs (G_1, n_1) and (G_2, n_2) for the subexpressions e_1 and e_2, and then create a new node $n \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2)$ to manage the concatenation. This node has three fields: two node-id fields n_1 and n_2 pointing to the two component subgraphs, and a boolean field that tracks if processing has crossed over from n_1 to n_2. Initially, we set this state to `False`.

Cat Left The translation for terms $\texttt{let}\ (x; y) = z\ \texttt{in}\ e$ is the crux of the translation. In SG-Cat-L, we create three new nodes. First, we allocate a *coordinator* node $n_c \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_z)$. This coordinator node keeps a reference n_z to the stream being scrutinized, and a boolean that records whether or not the punctuation mark `CatPunc` has been pulled from n_z yet. We then create two projection nodes $n_x \Rightarrow \texttt{CatProjNode}_1\,(n_c)$ and $n_y \Rightarrow \texttt{CatProjNode}_2\,(n_c)$, both of which point to the coordinator. Last, we construct the subgraph for the continuation term e, binding variables x and y to nodes n_x and n_y, respectively. This construction is crucial because it enables state sharing between x and y: both get bound to nodes that point back to *the same* coordinator, and so they can both modify the shared boolean state.

Sum Left and Right, Constants, Epsilon The sum constructors SG-InL and SG-InR are straightforward: we recursively construct the subgraph for the inner expression e, then create a new node, either $\texttt{InLNode}\,(\texttt{False}, n')$ or $\texttt{InRNode}\,(\texttt{False}, n')$. In both node types, the boolean field indicates if we have yet sent out the initial punctuation mark (`PlusA` or `PlusB`) indicating what type the rest of the stream will

$$\frac{N(x) = n}{N, R \vdash [\![x]\!] \rightsquigarrow (\emptyset, n)}\ \text{SG-Var} \qquad \frac{N, R \vdash [\![e_1]\!] \rightsquigarrow (G_1, n_1) \qquad N, R \vdash [\![e_2]\!] \rightsquigarrow (G_2, n_2) \qquad n\ \texttt{fresh}}{N, R \vdash [\![(e_1; e_2)]\!] \rightsquigarrow (G_1 \cup G_2 \cup \{n \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2)\}, n)}\ \text{SG-Cat-R}$$

$$\frac{N(z) = n_z \qquad N[x \mapsto n_x, y \mapsto n_y], R \vdash [\![e]\!] \rightsquigarrow (G, n) \qquad n_c, n_x, n_y\ \texttt{fresh}}{N, R \vdash [\![\texttt{let}\ (x; y) = z\ \texttt{in}\ e]\!] \rightsquigarrow \left(G \cup \left\{n_c \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_z), n_x \Rightarrow \texttt{CatProjNode}_1\,(n_c), n_y \Rightarrow \texttt{CatProjNode}_2\right.\right.}$$

$$\frac{N, R \vdash [\![e_1]\!] \rightsquigarrow (G_1, n_1) \qquad N[x \mapsto n_1], R \vdash [\![e_2]\!] \rightsquigarrow (G_2, n_2)}{N, R \vdash [\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] \rightsquigarrow (G_1 \cup G_2, n_2)}\ \text{SG-Let}$$

$$\frac{N, R \vdash [\![e]\!] \rightsquigarrow (G, n') \qquad n\ \texttt{fresh}}{N, R \vdash [\![\texttt{inl}(e)]\!] \rightsquigarrow (G \cup \{n \Rightarrow \texttt{InLNode}\,(\texttt{False}, n')\}, n)}\ \text{SG-InL}$$

$$\frac{N, R \vdash [\![e]\!] \rightsquigarrow (G, n') \qquad n\ \texttt{fresh}}{N, R \vdash [\![\texttt{inr}(e)]\!] \rightsquigarrow (G \cup \{n \Rightarrow \texttt{InRNode}\,(\texttt{False}, n')\}, n)}\ \text{SG-InR}$$

$$\frac{\begin{array}{c}N(z) = n_z \\ N[x \mapsto n_z], R \vdash [\![e_1]\!] \rightsquigarrow (G_1, n_1) \qquad N[y \mapsto n_z], R \vdash [\![e_2]\!] \rightsquigarrow (G_2, n_2) \qquad n\ \texttt{fresh}\end{array}}{N, R \vdash [\![\texttt{case}\,(z, x.e_1, y.e_2)]\!] \rightsquigarrow (G_1 \cup G_2 \cup \{n \Rightarrow \texttt{CaseNode}\,(0, n_z, n_1, n_2)\}, n)}\ \text{SG-SumCase}$$

$$\frac{n, n_e\ \texttt{fresh}}{N, R \vdash [\![\texttt{nil}]\!] \rightsquigarrow (\{n \Rightarrow \texttt{InLNode}\,(\texttt{False}, n_e), n_e \Rightarrow \texttt{EpsNode}\}, n)}\ \text{SG-Nil}$$

$$\frac{N, R \vdash [\![e_1]\!] \rightsquigarrow (G_1, n_1) \qquad N, R \vdash [\![e_2]\!] \rightsquigarrow (G_2, n_2) \qquad n, n'\ \texttt{fresh}}{N, R \vdash [\![e_1 :: e_2]\!] \rightsquigarrow (G_1 \cup G_2 \cup \{n \Rightarrow \texttt{InRNode}\,(\texttt{False}, n'), n' \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2)\}, n)}\ \text{SG-Cons}$$

$$\frac{\begin{array}{c}N(z) = n_z \qquad N, R \vdash [\![e_1]\!] \rightsquigarrow (G_1, n_1) \\ N[y \mapsto n_x, ys \mapsto n_y], R \vdash [\![e_2]\!] \rightsquigarrow (G_2, n_2) \qquad n, n_x, n_y, n_c\ \texttt{fresh}\end{array}}{N, R \vdash [\![\texttt{case}\,(z, e_1, y.ys.e_2)]\!] \rightsquigarrow \left(G_1 \cup G_2 \cup \{n \Rightarrow \texttt{CaseNode}\,(0, n_z, n_1, n_2), n_x \Rightarrow \texttt{CatProjNode}_1\,(n_c), n_y \Rightarrow \texttt{CatProjNode}\right.}$$

$$\frac{n\ \texttt{fresh}}{N, R \vdash [\![\texttt{eps}]\!] \rightsquigarrow (\{n \Rightarrow \texttt{EpsNode}\}, n)}\ \text{SG-Eps} \qquad \frac{n\ \texttt{fresh}}{N, R \vdash [\![k]\!] \rightsquigarrow (\{n \Rightarrow \texttt{IntNode}\,(\texttt{False}, k)\}, n)}\ \text{SG-Int}$$

$$\frac{N, r \vdash [\![e]\!] \rightsquigarrow (G, n) \qquad r\ \texttt{fresh}}{N, R \vdash [\![\texttt{fix}\,(e)]\!] \rightsquigarrow (G \cup \{r \Rightarrow \texttt{FixNode}\,(n)\}, r)}\ \text{SG-Fix} \qquad \frac{n\ \texttt{fresh}}{N, r \vdash [\![\texttt{rec}]\!] \rightsquigarrow (\{n \Rightarrow \texttt{RecNode}\,(r)\}, n)}\ \text{SG-Rec}$$

Figure 5.4: Translation from λ^y Terms to Pull Graphs

have. Meanwhile, the SG-Int rule creates a `IntNode` (`False`, k) from the constant k, with the initial state `False` indicating that we have not yet emitted the result. Last, the SG-Eps creates a node `EpsNode`, which has no state and no outgoing edges.

Sum Case The rule SG-SumCase translates a sum case expression by building subgraphs for both branches. We look up the node-id n_z corresponding to the scrutinee, and recursively construct subgraphs (G_1, n_1) and (G_2, n_2) for the two branches e_1 and e_2. In both branches, we bind the pattern variables x and y to the node-id n_z which represents the scrutinee. Last, we create a case node $n \Rightarrow$ `CaseNode` $(0, n_z, n_1, n_2)$ to handle the dispatch. This node holds the node-ids n_z, n_1 and n_2 of the scrutinee and two branches, and an integer state to represent if we have yet to receive the tag on n_z (the initial state, 0), or if we are pulling from n_1 or n_2 (1, and 2, respectively).

We note that this setup may seem strange from a typing perspective. After all, n_z is a reference to a subgraph that that ought to produce a stream of type $s + t$, while x and y should be bound to subgraphs of type s and t, respectively. This conundrum is resolved by the way the semantics actually operates: in Section 5.7.2, we will see that one of n_1 or n_2 will only be pulled on *after* the tag event `PlusA` or `PlusB` is pulled from n_z, and so at that point, n_z will be "of type" s or t, respectively. This further goes to show that the graph representation captures a fundamentally stateful computation where the "type" of a node evolves as events are pulled through it.

Nil, Cons, and Star Case Rather than introducing dedicated node types for the star operations, we exploit the isorecursive representation of $s^\star$ as $\varepsilon + s \cdot s^\star$. This encoding allows us to reuse the sum and concatenation machinery we have already defined. Specifically, the nil constructor `nil` can be represented in the same way as `inl (eps)`, first the left injection, then the term for ε. Similarly, the cons constructor e_1 `::` e_2 corresponds to `inr` $((e_1; e_2))$: first the right injection of the sum, then a cat-pair. The rules SG-Nil and SG-Cons encode this approach: they construct pull graphs using the existing `InLNode` (_, _), `InRNode` (_, _), and `CatRNode` (_, _, _) nodes to represent the nil and cons.

We employ the same trick for the star elimination form. The star-case expression `case` $(z, e_1, y.ys.e_2)$ is represented as the sum-case `case` $(z, \_.e_1, x.$`let` $(x; y) = ys$ `in` $e_2)$, where the second branch performs a cat-split to bind the head and tail of the pair to variables y and ys, respectively. The rule SG-StarCase implements this by constructing a graph that uses a `CaseNode` (_, _, _, _) node to dispatch on the sum at

the head of the $s^\star$, combined with the coordinator mechanism to handle the cat-split in the cons branch.

Recursion The rules SG-Fix and SG-Rec build the graph nodes for recursive program sections. The graph construction judgment threads through a node identifier r representing the most recent enclosing `fix` () term. Then, in SG-Fix, when we encounter a `fix` (e), we allocate a fresh node identifier r and build the graph (G, n) for the body e under the assumption that r identifies the current recursive context. We then produce a graph including $\{r \Rightarrow \texttt{FixNode}\ (n)\}$, which ties the knot by pointing r at the root node n of e. Correspondingly, when we encounter a recursive call `rec` in SG-Rec, we emit a `RecNode` (r) that points back to the enclosing recursive function.

5.7.2 Semantics of Pull Graphs

We now define the step function `step` $(G, n) = st$, which gives the operational semantics of pull graphs. The outputs st are the steps of a pull stream function, which can be `done` (G'), `skip` (G'), or `yield` (e, G'), where e is an event and G' is the graph with updated node states. We refer to the operation performed by `step`(G, n) as "pulling on" the node n in the graph. Pulling on a node n usually involves pulling on nodes that it maintains edges to, so stepping (G, n) will often require making recursive calls to pull from (G, n'), essentially shifting our focus to the node n'. Formally, we define this function by way of inference rules in Figures 5.5, 5.6, and 5.7. We describe the different rules and their operation below.

Epsilon and Integers The `EpsNode` node is the simplest to define the semantics of: it simply emits `done` (_) upon the first request, since a stream of type ε contains no events. This is encoded by G-Eps, which simply produces `done` (G) when stepping a node $n \Rightarrow$ `EpsNode`.

The constant integer node $n \Rightarrow$ `IntNode` (b, k) corresponds to the integer constant term k. The first time it's pulled from, it should produce the value k, and then it should produce `done` on the next. We implement this by initializing the boolean field b to `False`. If we pull and find the boolean to be in this initial state (G-Int-False), we `yield` the event `baseev` (k) and set the flag to `True` by updating the node n to `IntNode` $(\texttt{True}, k)$ in the result. Next time, we'll find the boolean set to `True` (G-Int-True), and hence produce `done`. This correctly produces a stream of type `Int`: a single event `baseev` (k).

$$\frac{G[n] \Rightarrow \texttt{EpsNode}}{\texttt{step}\,(G,n) = \texttt{done}\,(G)}\ \text{G-Eps} \qquad \frac{G[n] \Rightarrow \texttt{IntNode}\,(\texttt{False}, k)}{\texttt{step}\,(G,n) = \texttt{yield}\,(\texttt{baseev}\,(k), G[n \Rightarrow \texttt{IntNode}\,(\texttt{True}, k)])}\ \text{G-Int-False}$$

$$\frac{G[n] \Rightarrow \texttt{IntNode}\,(\texttt{True}, k)}{\texttt{step}\,(G,n) = \texttt{done}\,(G)}\ \text{G-Int-True}$$

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2) \qquad \texttt{step}\,(G, n_1) = \texttt{yield}\,(x, G')}{\texttt{step}\,(G,n) = \texttt{yield}\,(\texttt{CatEv}\,(x), G')}\ \text{G-Cat-R-False-Yield}$$

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2) \qquad \texttt{step}\,(G, n_1) = \texttt{skip}\,(G')}{\texttt{step}\,(G,n) = \texttt{skip}\,(G')}\ \text{G-Cat-R-False-Skip}$$

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2) \qquad \texttt{step}\,(G, n_1) = \texttt{done}\,(G')}{\texttt{step}\,(G,n) = \texttt{yield}\,(\texttt{CatPunc}, G'[n \Rightarrow \texttt{CatRNode}\,(\texttt{True}, n_1, n_2)])}\ \text{G-Cat-R-False-Done}$$

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{True}, n_1, n_2) \qquad \texttt{step}\,(G, n_2) = st}{\texttt{step}\,(G,n) = st}\ \text{G-Cat-R-True}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(\texttt{False}, n')}{\texttt{step}\,(G,n) = \texttt{yield}\,(\texttt{PlusA}, G[n \Rightarrow \texttt{InLNode}\,(\texttt{True}, n')])}\ \text{G-InL-False}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(\texttt{True}, n') \qquad \texttt{step}\,(G, n') = st}{\texttt{step}\,(G,n) = st}\ \text{G-InL-True}$$

$$\frac{G[n] \Rightarrow \texttt{InRNode}\,(\texttt{False}, n')}{\texttt{step}\,(G,n) = \texttt{yield}\,(\texttt{PlusB}, G[n \Rightarrow \texttt{InRNode}\,(\texttt{True}, n')])}\ \text{G-InR-False}$$

$$\frac{G[n] \Rightarrow \texttt{InRNode}\,(\texttt{True}, n') \qquad \texttt{step}\,(G, n') = st}{\texttt{step}\,(G,n) = st}\ \text{G-InR-True}$$

Figure 5.5: Graph Semantics Rules (Part 1)

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(0, n', n_1, n_2) \qquad \texttt{step}\,(G, n') = \texttt{skip}\,(G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}\ \text{G-Case-Zero-Skip}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(0, n', n_1, n_2) \qquad \texttt{step}\,(G, n') = \texttt{yield}\,(\texttt{PlusA}, G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G'[n \Rightarrow \texttt{CaseNode}\,(1, n', n_1, n_2)])}\ \text{G-Case-Zero-Yield-A}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(0, n', n_1, n_2) \qquad \texttt{step}\,(G, n') = \texttt{yield}\,(\texttt{PlusB}, G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G'[n \Rightarrow \texttt{CaseNode}\,(2, n', n_1, n_2)])}\ \text{G-Case-Zero-Yield-B}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(1, n', n_1, n_2) \qquad \texttt{step}\,(G, n_1) = st}{\texttt{step}\,(G, n) = st}\ \text{G-Case-One}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(2, n', n_1, n_2) \qquad \texttt{step}\,(G, n_2) = st}{\texttt{step}\,(G, n) = st}\ \text{G-Case-Two}$$

Figure 5.6: Graph Semantics Rules (Part 2)

Sum Injection The sum injection nodes $\texttt{InLNode}\,(b, n')$ and $\texttt{InRNode}\,(b, n')$ must produce sequences that look like streams of type $s+t$. Recall that such streams begin with either a `PlusA` or `PlusB` punctuation (for left or right injection), followed by a stream of the corresponding type. The child node n' is responsible for producing this stream, so the injection node is only responsible for first emitting the appropriate punctuation mark before delegating to n'. This two-stage machine is implemented using the boolean flag b on the node, similar to the rules G-Int-False and G-Int-True. The flag is initialized to `False`. When we pull from an injection node while the flag is `False` (rules G-InL-False and G-InR-False), the node `yields` the corresponding punctuation mark and updates the node's flag to `True` in the resulting graph. Subsequent pulls from the node will find the flag to be `True` (rules G-InL-True and G-InR-True). In these cases, we recursively pull from n' and produce whatever result the child returns.

Cat Right The cat-pair node $\texttt{CatRNode}\,(b, n_1, n_2)$ is tasked with producing a stream that behaves like one of type $s \cdot t$, given access to child nodes n_1 and n_2 that produce streams of types s and t, respectively. Recall from Section 5.5 that such a stream has the form:

$$\texttt{CatEv}\,(x_1), \ldots, \texttt{CatEv}\,(x_n), \texttt{CatPunc}, y_1, \ldots, y_m$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{skip}\,(G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}\ \text{G-CatProj1-Skip}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{yield}\,(\texttt{CatEv}\,(x), G')}{\texttt{step}\,(G, n) = \texttt{yield}\,(x, G')}\ \text{G-CatProj1-Yield-CatEv}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{yield}\,(\texttt{CatPunc}, G')}{\texttt{step}\,(G, n) = \texttt{done}\,(G'[n' \Rightarrow \texttt{CatCoordNode}\,(\texttt{True}, n_s)])}\ \text{G-CatProj1-Yield-CatPunc}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{skip}\,(G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}\ \text{G-CatProj2-False-Skip}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{yield}\,(\texttt{CatEv}\,(\_), G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}\ \text{G-CatProj2-False-CatEv}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n_s) \qquad \texttt{step}\,(G, n_s) = \texttt{yield}\,(\texttt{CatPunc}, G')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G'[n' \Rightarrow \texttt{CatCoordNode}\,(\texttt{True}, n_s)])}\ \text{G-CatProj2-False-CatPunc}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{True}, n_s) \qquad \texttt{step}\,(G, n_s) = st}{\texttt{step}\,(G, n) = st}\ \text{G-CatProj2-True}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle x, f\rangle) \qquad fx = \texttt{done}\,(x')}{\texttt{step}\,(G, n) = \texttt{done}\,(G[n \Rightarrow \texttt{SourceNode}\,(\langle x', f\rangle)])}\ \text{G-Source-Done}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle x, f\rangle) \qquad fx = \texttt{skip}\,(x')}{\texttt{step}\,(G, n) = \texttt{skip}\,(G[n \Rightarrow \texttt{SourceNode}\,(\langle x', f\rangle)])}\ \text{G-Source-Skip}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle x, f\rangle) \qquad fx = \texttt{yield}\,(e, x')}{\texttt{step}\,(G, n) = \texttt{yield}\,(e, G[n \Rightarrow \texttt{SourceNode}\,(\langle x', f\rangle)])}\ \text{G-Source-Yield}$$

$$\frac{G[n] \Rightarrow \texttt{FixNode}\,(n') \qquad \texttt{step}\,(G, n') = st}{\texttt{step}\,(G, n) = st}\ \text{G-Fix} \qquad \frac{G[n] \Rightarrow \texttt{RecNode}\,(n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}\ \text{G-Rec}$$

Figure 5.7: Graph Semantics Rules (Part 3)

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(b, n_1, n_2) \qquad \texttt{reset}\,(G, n_1) = G' \qquad \texttt{reset}\,(G', n_2) = G''}{\texttt{reset}\,(G, n) = G''[n \Rightarrow \texttt{CatRNode}\,(\texttt{False}, n_1, n_2)]}\ \text{R-Cat-R}$$

$$\frac{G[n] \Rightarrow \texttt{CatCoordNode}\,(b, n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{reset}\,(G, n) = G'[n \Rightarrow \texttt{CatCoordNode}\,(\texttt{False}, n')]}\ \text{R-Coord}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{reset}\,(G, n) = G'}\ \text{R-CatProj1}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{reset}\,(G, n) = G'}\ \text{R-CatProj2}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(b, n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{reset}\,(G, n) = G'[n \Rightarrow \texttt{InLNode}\,(\texttt{False}, n')]}\ \text{R-InL} \quad \frac{G[n] \Rightarrow \texttt{InRNode}\,(b, n') \qquad \texttt{reset}\,(G, n') = G'}{\texttt{reset}\,(G, n) = G'[n \Rightarrow \texttt{InRNode}\,(\texttt{False}, n')]}\ \text{R-InR}$$

$$\frac{\begin{array}{c} G[n] \Rightarrow \texttt{CaseNode}\,(k, n', n_1, n_2) \\ \texttt{reset}\,(G, n') = G' \qquad \texttt{reset}\,(G', n_1) = G'' \qquad \texttt{reset}\,(G'', n_2) = G''' \end{array}}{\texttt{reset}\,(G, n) = G'''[n \Rightarrow \texttt{CaseNode}\,(0, n', n_1, n_2)]}\ \text{R-Case} \quad \frac{G[n] \Rightarrow \texttt{EpsNode}}{\texttt{reset}\,(G, n) = G}\ \text{R-Eps}$$

$$\frac{G[n] \Rightarrow \texttt{IntNode}\,(b, k)}{\texttt{reset}\,(G, n) = G[n \Rightarrow \texttt{IntNode}\,(\texttt{False}, k)]}\ \text{R-Int} \quad \frac{G[n] \Rightarrow \texttt{FixNode}\,(n')}{\texttt{reset}\,(G, n) = G}\ \text{R-Fix} \quad \frac{G[n] \Rightarrow \texttt{RecNode}\,(n')}{\texttt{reset}\,(G, n) = G}\ \text{R-Rec}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle x, f \rangle)}{\texttt{reset}\,(G, n) = G}\ \text{R-Source}$$

Figure 5.8: Subgraph Reset Operation

That is, events from the first stream wrapped in `CatEv` (_), followed by a punctuation mark, followed by events from the second stream. The node n must therefore first pull from n_1, wrapping the events it `yields` in `CatEv` (_) and forwarding them along. When n_1 produces `done`, the node n must then produce the punctuation mark `CatPunc` and then simply forward events from n_2 from then on. This too is implemented with a boolean flag b, initialized to `False`. When we pull from n while the flag is `False`, we recursively pull from n_1 and handle the three cases separately:

- If n_1 `yields` an event x (G-Cat-R-False-Yield), we wrap it as `CatEv` (x) and `yield` it.
- If n_1 `skips` (G-Cat-R-False-Skip), the cat-pair node also `skips`.
- If n_1 is `done` (G-Cat-R-False-Done), we `yield` the punctuation mark `CatPunc` and update the flag to `True`.

When the flag is `True` (G-Cat-R-True), the node n simply delegates to n_2, returning whatever result it produces.

Case A case node `CaseNode` (k, n', n_1, n_2) must produce values from either n_1 or n_2 based on the tag from n', which produces a stream of type $s+t$. This is implemented as a three-state machine: we begin by pulling from n' until the tag arrives, after which we switch to one of two other states that forward elements from n_1 or n_2 respectively. The state is tracked by the integer value k on the node, which is always either 0 (waiting for the tag), 1 (pulling from n_1), or 2 (pulling from n_2). When in state 0, we pull from the source stream n' and handle three cases:

- If n' `skips` (G-Case-Zero-Skip), the case node also `skips`.
- If n' `yields` the punctuation mark `PlusA` (G-Case-Zero-Yield-A), we update the state to 1 and `skip`.
- If n' `yields` the punctuation mark `PlusB` (G-Case-Zero-Yield-B), we update the state to 2 and `skip`.

Once in state 1 or 2, the rules are symmetric: we simply pull from n_1 (G-Case-One) or n_2 (G-Case-Two) respectively and forward whatever result they produce.

Concatenation Projection The rules for the concatenation projection nodes are the most important and subtle in our semantics. Recall from Section 5.7.1 that the node types $\texttt{CatProjNode}_1\ (n')$ and $\texttt{CatProjNode}_2\ (n')$ arise from the translation of the cat-elimination form $\texttt{let}\ (x; y)\ =\ z\ \texttt{in}\ e$. During the translation of e, lookups for x resolve to a $\texttt{CatProjNode}_1\ (n')$ node and lookups for y resolve to a $\texttt{CatProjNode}_2\ (n')$ node.

In each pair of $\texttt{CatProjNode}_1\ (n')$ and $\texttt{CatProjNode}_2\ (n')$ nodes, the n' is a reference to the coordinator node $\texttt{CatCoordNode}\ (b, n_s)$ that holds shared mutable state b and a reference n_s to the stream of type $s \cdot t$ being projected from. The shared state is updated to maintain the invariant that b is `True` if and only if the punctuation `CatPunc` has been pulled from the stream n_s.

First Projection. The node $\texttt{CatProjNode}_1\ (n')$ must project out the s component from a stream of type $s \cdot t$. Since such a stream has the form $\texttt{CatEv}\ (x_1), \ldots, \texttt{CatEv}\ (x_n), \texttt{CatPunc}, \ldots$, the job of this node is to pull from the source stream n_s, peel off the $\texttt{CatEv}\ (\_)$ wrappers, and `yield` the underlying values until reaching the punctuation. When we pull from a $\texttt{CatProjNode}_1\ (n')$, the coordinator's boolean state must `False`, since we have not yet seen the punctuation. Then, we pull from n_s (the coordinator's source) and handle three cases:

- If n_s `skips` (G-CatProj1-Skip), the projection also `skips`.
- If n_s `yields` $\texttt{CatEv}\ (x)$ (G-CatProj1-Yield-CatEv), the projection `yields` x, stripping off the wrapper.
- If n_s `yields CatPunc` (G-CatProj1-Yield-CatPunc), we update the coordinator's state to `True` (marking that the punctuation has been seen) and the projection returns `done`.

Second Projection. The node $\texttt{CatProjNode}_2\ (n')$ must project out the t component from a stream of type $s \cdot t$. In effect, we must advance to the punctuation mark—either by pulling and discarding events to reach it, or by noticing that a previous pull from the corresponding $\texttt{CatProjNode}_1\ (n')$ already reached it—and then `yield` the remaining events. The rules for accomplishing this also rely on the coordinator's shared state. In the simple case where we have already pulled on the corresponding $\texttt{CatProjNode}_1\ (n')$ node and the coordinator's flag is `True`, rule G-CatProj2-True applies: we simply pull from n_s and return whatever result it produces.

Otherwise, if the coordinator's flag is still `False`, we must pull from n_s to find the punctuation mark. Depending on what we get, different rules apply, but in all cases we `skip` (nothing needs to be produced yet):

- If n_s `skips` (G-CatProj2-False-Skip), we `skip`.
- If n_s `yields CatEv` (x) (G-CatProj2-False-CatEv), we `skip` (discarding the event).
- If n_s `yields CatPunc` (G-CatProj2-False-CatPunc), we update the coordinator's state to `True` and `skip`.

Once the coordinator's state becomes `True`, subsequent pulls use G-CatProj2-True to forward events from n_s.

The dance that these two projection nodes engage in through their interaction with shared state is the key complexity that the type system is designed to control. Indeed, what would happen if we first pulled from $\texttt{CatProjNode}_2\ (n')$ and then from $\texttt{CatProjNode}_1\ (n')$? We would advance through to the punctuation, set the coordinator's flag to `True`, and forward the rest of the stream. But then, attempting to pull from $\texttt{CatProjNode}_1\ (n')$ would fail: those rules expect the coordinator's flag to be `False`, since the first projection must consume events *before* the punctuation marker. The orderedness checking type system is designed to ensure that this never happens. If we ever pull from a $\texttt{CatProjNode}_1\ (n')$ node, we believe the type system guarantees[23] that we do so before pulling from the corresponding $\texttt{CatProjNode}_2\ (n')$ node.

Sources Pulling from a source node (G-Source-Done, G-Source-Skip, and G-Source-Yield), simply invokes the step function f on the current state x and interprets the three possible outcomes in the obvious way:

- If $f\ x = \texttt{done}\ (x')$ (G-Source-Done), the node returns `done` and updates its state to x'.
- If $f\ x = \texttt{skip}\ (x')$ (G-Source-Skip), the node returns `skip` and updates its state to x'.
- If $f\ x = \texttt{yield}\ (e, x')$ (G-Source-Yield), the node `yields` the event e and updates its state to x'.

[23] But have not proved! Future work, one hopes.

Recursion Recall from Section 5.7.1 that the graph construction ensures that a subgraph corresponding to a recursive operation is enclosed by a node $r \Rightarrow \texttt{FixNode}\,(n')$, where n' points to the root node of the operation. Then, recursive calls in the body of the operation are translated to $\texttt{RecNode}\,(r)$s, with r pointing back to the enclosing recursion node. This lets us implement the recursion logic. When we pull from a $\texttt{RecNode}\,(r)$, we reset the control state of all nodes reachable from r, i.e. all those within the body of the operation we're currently recursively calling. Then, the next time we pull from r we have effectively restarted the recursive function from its initial state, "jumping" back to the top of the function. This makes it clear why all recursive calls must be tail calls: the next top-level pull simply re-starts the entire operation, and we do not return to the recursive call site.

The resetting logic—defined by the subgraph reset judgment in Figure 5.8—traverses the subgraph rooted at the recursive function body, setting all flags to their initial states. It stops at (a) nested recursive functions $\texttt{FixNode}\,(n')$ (R-Fix), which maintain their own state, and (b) source nodes $\texttt{SourceNode}\,(\langle x, f\rangle)$ (R-Source). Resetting a subgraph also only modifies mutable state fields within nodes—it never changes or enlarges the structure of the graph itself, which must remain static over the course of execution.

The Rule G-Rec is where the actual recursion logic happens. This is implemented in the rule G-Rec, which calls the auxiliary judgment $\texttt{reset}\,(G, n) = G'$ (Figure 5.8), which walks the subgraph rooted at n and resets all mutable fields to their initial values. After resetting the state, G-Rec `skips`, allowing execution to continue.

Nothing particularly interesting happens in the rule G-Fix: pulling from a $\texttt{FixNode}\,(n')$ simply delegates to n', the root of the recursive function body. This node merely exists to serve as a dominator for the subgraph implementing the recursive operation.

5.7.3 Putting it All Together

At the top level, we define the semantics of a pull graph (G, n) as a single unified pull stream by repeatedly pulling from the source node n, as shown in Figure 5.9.

Theorem 5.7.1 (Semantic Soundness (Unproven))**.** *Suppose:*

1. $y_1 : s_1, \ldots y_\ell : s_\ell \mid P_{y_1,\ldots y_\ell} \vdash_\emptyset e : s$

$$\frac{\texttt{step}\,(G, n) = \texttt{done}\,(G')}{\texttt{step}(G, n) = \texttt{done}\,((G', n))}\ \textsc{Step-Done}$$

$$\frac{\texttt{step}\,(G, n) = \texttt{skip}\,(G')}{\texttt{step}(G, n) = \texttt{skip}\,((G', n))}\ \textsc{Step-Skip}$$

$$\frac{\texttt{step}\,(G, n) = \texttt{yield}\,(x, G')}{\texttt{step}(G, n) = \texttt{yield}\,(x, (G', n))}\ \textsc{Step-Yield}$$

Figure 5.9: Top-Level Pull Graph Step Rules

2. $N(y_i) = n_i$

3. $N, \cdot \vdash [\![e]\!] \rightsquigarrow (G, n)$

Then, for all k, if $\langle x_{0i}, f_i\rangle \vDash^k s_i$, then $\langle (G', n)\,, \texttt{step}\rangle \vDash^k s$, with $G' = G \cup \{n_i \Rightarrow \texttt{SourceNode}\,(\langle x_{0i}, f_i\rangle)\}$

This theorem captures the intended soundness property of the operational semantics. Consider an open term e that is well-typed with type s in a context with variables $y_i : s_i$ and a partial order P where all y_i are unrelated. Suppose we have pull streams for the input variables $\langle x_{0i}, f_i\rangle$ that semantically behave like streams of type s_i (i.e., $\langle x_{0i}, f_i\rangle \vDash^k s_i$). When we set up an initial node map with $N(y_i) = n_i$, the compiler produces $N, \cdot \vdash [\![e]\!] \rightsquigarrow (G, n)$. The theorem states that if we then run the full graph $G' = G \cup \{n_i \Rightarrow \texttt{SourceNode}\,(\langle x_{0i}, f_i\rangle)\}$, it behaves like a stream of type s: $\langle (G', n)\,, \texttt{step}\rangle \vDash^k s$.

This theorem is unproven, though we have validated it with random testing of the Yoink interpreter that implements it. See Chapter 7 for discussion of how one might prove this in the future.

Chapter 6

Compiling λ^{y} to Fused Imperative Programs

In the previous chapter, we gave a semantics for the pull graphs derived from λ^{y} terms. These operational semantics bring us much closer to our goal of running functional stream programs in bounded space: the semantics clearly do not materialize any unbounded auxiliary state. Indeed, the machine's state is entirely captured in a graph containing only nodes with bounded state. You could implement these semantics directly as an interpreter, if you wished.

However, this is not entirely satisfying. If you *were* to build such an interpreter, it would still potentially use unbounded space as it walked the pull graph. What we really want is a *compiler* that takes pull graphs and turns them into bounded-state imperative programs. That is the goal of the first half of this chapter.

Once we have a practical implementation strategy for a compiler, we can actually build one! In the second half of this chapter (Section 6.2), we present Yoink, our implementation of λ^{y} that compiles high-level functional-style stream processing functions to fused imperative code. Yoink is both more expressive than existing functional streaming eDSL (it allows pattern matching and recursion), and can also compile (and hence fuse) some functions (like `concatMap`) that are not handled by previous work.

The path we take to compile a Yoink program is a well-understood one: we simply *specialize* the semantics to that program. This is often referred to as the "Futamura Projection" [72]. While it's mostly a theoretical concept and not usually employed as a practical tool for building languages (though there are some exceptions [191]), using the Futamura Projection is feasible here because the operational semantics only recurses to a depth that is bounded by the size of the pull graph. Evaluating a single step might have to pull from every node in the pull graph, but no more than that: we can simply unfold the entire computation.

As discussed in Chapter 2, using metaprogramming or partial evaluation as a compilation technique for stream programs is not a new idea [105, 106]. Like previous work that uses metaprogramming to compile stream programs, our compilation has the added bonus of *fusing* the programs.

Like the content of the last chapter, we have not proved any equivalence between this compiler and the

```
P ::= skip
    | x := e
    | P; P
    | if e then { P } else { P }
    | while e { P }
e ::= x | k | True | False
    | BaseEv(k) | CatPunc | CatEv(e)
    | PlusA | PlusB | DONE | None
```

Figure 6.1: Target Language Grammar

semantics, though we are confident that one holds. Doing so would be fun and interesting future work!

6.1 The Compiler

The compiler for λ^{γ} uses the pull graph representation from Chapter 5 as its source language. The target language is a minimal imperative language (in the style of IMP [190]) with values for events and state machine flags; its syntax appears in Figure 6.1.

The first step of the compilation process is to take the pull graph (G, n) to be compiled, and replace each node's state with a fresh variable from the target language.

From there, compilation proceeds in two steps. First, the judgment `compileInit` (G, n) `= I` walks the graph from n and generates a statement `I` that initializes the state variables. Second, a compilation judgment `compileStep`(G, n){`dst`} `= P` (the real judgment is slightly different, and is discussed in Section 6.1.1) generates a statement `P` that implements the step function. When executed, the generated statement `P` deposits the result of the step in the destination variable `dst`: if the step yields an event e, that event gets written to `dst`; if the step skips, `dst` is set to `None`; if the step is done, `dst` is set to a distinguished `DONE` value. Intuitively, the specification for this judgment is that if we start in some state, running the generated code `P` performs the same computation as one step of the step function from the semantics in Section 5.7.2. Importantly, this statement `P` (1) *mutably* updates the state of the program instead of threading state around like the operational semantics, and (2) does not materialize any intermediate values in

$$\frac{\texttt{compileStep}(G, n)\{\texttt{dst}\} = \texttt{P} \qquad \texttt{compileInit}\,(G, n) = \texttt{I}}{\texttt{compile}\,(G, n) = \left\{\begin{array}{l}\texttt{I;}\\ \texttt{dst := None}\\ \texttt{res := []}\\ \texttt{while True \{}\\ \quad \texttt{P}\\ \quad \texttt{if dst == DONE \{ break \}}\\ \quad \texttt{if dst != None \{ res.append(dst) \}}\\ \texttt{\}}\end{array}\right\}}$$

Figure 6.2: Compiling to Accumulator

$$\frac{\texttt{compileStep}(G, n)\{\texttt{dst}\} = \texttt{P} \qquad \texttt{compileInit}\,(G, n) = \texttt{I}}{\texttt{compile}\,(G, n) = \left\{\begin{array}{l}\texttt{I;}\\ \texttt{conn := socket.create(...)}\\ \texttt{dst := None}\\ \texttt{while True \{}\\ \quad \texttt{P}\\ \quad \texttt{if dst == DONE \{ break \}}\\ \quad \texttt{if dst != None \{ conn.write(dst) \}}\\ \texttt{\}}\\ \texttt{conn.close()}\end{array}\right\}}$$

Figure 6.3: Compiling to Socket

memory other than its state variables.

This approach enables different compilation strategies depending on how we consume the stream. Figure 6.2 shows compilation to a program that accumulates stream elements into a list. The rule compiles (G, n) to a step statement `P` and initialization statement `I`, then wraps them in a loop that repeatedly runs `P` and inspects the destination variable `dst`. If `dst` contains `DONE`, we break; if `None`, we continue; otherwise, we append the event to the result list. Figure 6.3 shows an alternative that streams events over a network socket instead. In the implementation of Yoink, we compile to the least common denominator: iterators. This generalizes the choice of target, and lets clients compile once but run programs in many ways.

6.1.1 Compiling the Step Function

The core of the compiler walks the pull graph and generates imperative code implementing the step function. Each compilation rule corresponds directly to one or more operational semantics rules from Sec-

tion 5.7.2, generating code that performs the same computation in an imperative style rather than as a state transition.

6.1.2 Compiling with Continuations

This naive approach to code generation—where we pass a destination variable `dst` to each recursive compilation call, with the specification that the generated code should write its result to `dst`—leads to overly verbose generated code. Many compilation rules would recursively compile a subterm and then branch on the result, producing code like `dst := None; if dst == None then ...`, where the compiler writes to `dst` and immediately reads it back.

We can do better by using a continuation-passing style, following a technique due to Kiselyov et al. [105]. Instead of passing a destination variable, we pass a *continuation*—a function $k : \{\texttt{DONE}\} + \{\texttt{None}\} + E \to \texttt{stmt}$, where E is the set of events and `stmt` is the set of target-language statements. Anywhere we would have generated code `dst := x`, we instead generate $k(x)$.

The final step of this technique exploits the fact that a function $k : A + B \to C$ is equivalent to a pair of functions $A \to C$ and $B \to C$, and a function out of a singleton $\{\star\} \to C$ is merely an element of C. The output of a step function is always one of three things: (1) `DONE`, when the stream is finished, (2) `None`, when the stream skips without producing an event, or (3) an event e. So, instead of passing a single continuation, we can instead compile with three separate arguments: a statement `D` to run when the stream is done, a statement `S` to run when the stream skips, and a continuation $y : E \to \texttt{stmt}$ to which we throw[24] events when we want to yield them. This particular technique is an instance of a more general code-generation strategy sometimes known cryptically as "the trick" [51], which exploits CPS conversion to specialize code.

Now that we have the continuation-based step-compilation judgment in place, we can revisit the top-level compilation strategy. We can implement the compile-to-list strategy by compiling (G, n) with $D =$ `break`, $S =$ `skip`, and $y(e) =$ `res.append`(e). This is shown end-to-end in Figure 6.4.

[24] We frequently use the term "throw" in the sense of passing a value to a continuation, not in the exception-handling sense.

$$\frac{\begin{array}{c} y(e) = \texttt{res.append}(e) \\ \texttt{compileStep}\,(G, n)\,\{\texttt{break} \mid \texttt{skip} \mid y\} = \texttt{P} \qquad \texttt{compileInit}\,(G, n) = \texttt{I} \end{array}}{\texttt{compile}\,(G, n) = \left\{\begin{array}{l} \texttt{I;} \\ \texttt{res := []} \\ \texttt{while True \{ P \}} \end{array}\right\}}\ \textsc{Compile}$$

Figure 6.4: Compiling to Accumulator, With Continuations

6.1.3 The Rules

The step-function compilation judgment has the form:

$$\texttt{compileStep}\,(G, n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}$$

This judgment means: given a graph G and a node n, along with the three continuations described above (done statement `D`, skip statement `S`, and yield continuation y), we generate imperative code `P` that implements one step of the pull semantics for node n. The rules for this judgment are shown in Figures 6.5 and 6.6.

Eps and Int The rules CS-Eps and CS-Int are straightforward. With the first, we note that pulling from `eps` immediately yields `done` $(-)$ in the operational semantics (Tp-Eps), so compiling `eps` simply produces the "done" statement `D`.

The rule CS-Int implements the two-phase state machine of producing an int then stopping. The generated code first looks up the state boolean `b`. If it's false, we haven't yet produced this int, so we throw `baseev` (k) to the yield continuation y after setting `b` to true.

Cat-R The CS-Cat-R rule generates code implementing the sequential behavior of a `CatRNode` $(\texttt{b}, n_1, n_2)$. Like CS-Int, it generates a conditional that branches on the boolean flag `b`, which in this case represents the state of if we've drained n_1 and are now pulling from n_2. The first branch of this conditional (we're now pulling from n_2) runs the code `P2`, which is the result of directly compiling n_2. If the boolean is false (we're in the initial state of pulling from n_1), the code `P1` runs, which is the result of compiling n_1. The key wrinkle is that when we compile n_1, we modify the continuations to pass to the recursive call. First, we

$$\frac{G[n] \Rightarrow \texttt{EpsNode}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{D}}\ \text{CS-Eps}$$

$$\frac{G[n] \Rightarrow \texttt{IntNode}\,(\texttt{b}, k)}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{if b then D else \{b := True; } y(\texttt{BaseEv}(k))\texttt{\}}}\ \text{CS-Int}$$

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{b}, n_1, n_2) \qquad y'(e) = y(\texttt{CatEv}\,(e)) \qquad \texttt{compileStep}\,(G,n_1)\,\{\texttt{b := True} \mid \texttt{S} \mid y'\} = \texttt{P1} \qquad \texttt{compileStep}\,(G,n_2)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P2}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{if b then P2 else P1}}\ \text{CS-Cat-R}$$

$$\frac{\begin{array}{c} G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{b}, n_s) \\ y'(e) = \texttt{if } e \texttt{ == CatPunc \{b := True; S\} else \{}y(e.\texttt{inner})\texttt{\}} \\ \texttt{compileStep}\,(G,n_s)\,\{\texttt{D} \mid \texttt{S} \mid y'\} = \texttt{P} \end{array}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}\ \text{CS-CatProj1}$$

$$\frac{\begin{array}{c} G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \qquad G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{b}, n_s) \\ y'(e) = \texttt{if b then } y(e) \texttt{ else \{if } e \texttt{ == CatPunc then \{b := True;S\} else S\}} \\ \texttt{compileStep}\,(G,n_s)\,\{\texttt{D} \mid \texttt{S} \mid y'\} = \texttt{P} \end{array}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}\ \text{CS-CatProj2}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(\texttt{b}, n') \qquad \texttt{compileStep}\,(G,n')\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{if b then P else \{b := True; } y(\texttt{PlusA})\texttt{\}}}\ \text{CS-InL}$$

$$\frac{G[n] \Rightarrow \texttt{InRNode}\,(\texttt{b}, n') \qquad \texttt{compileStep}\,(G,n')\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{if b then P else \{b := True; } y(\texttt{PlusB})\texttt{\}}}\ \text{CS-InR}$$

$$\frac{\begin{array}{c} G[n] \Rightarrow \texttt{CaseNode}\,(\texttt{k}, n', n_1, n_2) \\ y'(e) = \texttt{\{if } e \texttt{ == PlusA then k := 1 else k := 2\};S} \\ \texttt{compileStep}\,(G,n')\,\{\texttt{D} \mid \texttt{S} \mid y'\} = \texttt{P} \\ \texttt{compileStep}\,(G,n_1)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P1} \qquad \texttt{compileStep}\,(G,n_2)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P2} \end{array}}{\texttt{compileStep}\,(G,n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{if k == 0 then P else \{if k == 1 then P1 else P2\}}}\ \text{CS-Case}$$

Figure 6.5: Step Function Compilation Rules (Part 1)

$$\frac{G[n] \Rightarrow \texttt{FixNode}\,(n') \qquad \texttt{compileStep}\,(G, n')\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}{\texttt{compileStep}\,(G, n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}\ \text{CS-Fix}$$

$$\frac{G[n] \Rightarrow \texttt{RecNode}\,(n') \qquad G[n'] \Rightarrow \texttt{FixNode}\,(r) \qquad \texttt{compileReset}\,(G, r) = \texttt{P}}{\texttt{compileStep}\,(G, n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \texttt{P}}\ \text{CS-Rec}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle \texttt{f} \rangle)}{\texttt{compileStep}\,(G, n)\,\{\texttt{D} \mid \texttt{S} \mid y\} = \left\{\begin{array}{l}\texttt{a := f();}\\ \texttt{if a == None then S else \{}\\ \quad \texttt{if a == DONE then D else } y(\texttt{a})\\ \texttt{\}}\end{array}\right\}}\ \text{CS-Source}$$

Figure 6.6: Step Function Compilation Rules (Part 2)

modify the yield continuation:

$$y'(e) = y(\texttt{CatEv}\,(e))$$

This way, whatever events n_1 wants to emit, we modify its code generation to instead wrap them with the `CatEv` $(-)$ tag. This implements the behavior of the semantics rule G-Cat-R-False-Yield. The other continuation that gets modified is the done continuation, where we pass the variable update `b := True`. This implements the behavior of the semantics rule G-Cat-R-False-Done: n_1 being done is our signal to switch to running n_2, so we set the flag. On the next pull, we'll take the first branch of the generated conditional to run the code that pulls from n_2.

Cat-Proj The rules CS-CatProj1 and CS-CatProj2 produce code that implements the semantics of projecting the first and second substreams from a stream of type $s \cdot t$. We recall the semantics of these nodes from Section 5.7.2.

Nodes $n_1 \Rightarrow \texttt{CatProjNode}_1\,(n')$ and $n_2 \Rightarrow \texttt{CatProjNode}_2\,(n')$ come in pairs, both referencing a shared coordinator node $G[n'] \Rightarrow \texttt{CatCoordNode}\,(\texttt{b}, n_s)$ that holds (a) a reference to the stream to be destructed n_s, and (b) a flag variable `b` that is true if and only if the punctuation mark `CatPunc` has been pulled from n_s. Pulling from n_1 pulls n_s and forwards unwrapped elements until the punctuation mark appears, and then sets the flag. Pulling from n_2 first checks if the flag has been set: if it has not, it discards the elements of n_s up to the punctuation mark. Then, pulling from n_2 simply forwards elements from n_s.

The compiler generates code that implements exactly this behavior. The rule CS-CatProj1 gener-

ates code for a $\texttt{CatProjNode}_1\ (n')$ by emitting the code generated by compiling n_s with an updated yield continuation:

$$y'(e) = \texttt{if } e \texttt{ == CatPunc \{b := True; S\} else \{}y(e\texttt{.inner})\texttt{\}}$$

Each time the scrutinee stream n_s produces an element, we first scrutinize it. If it's not a `CatPunc`, then it must be of the form $\texttt{CatEv}\ (e')$. We extract the e' with $e\texttt{.inner}$, and yield it. If it is a `CatPunc`, we set the flag to true, and run the skip continuation.

Meanwhile, the rule CS-CatProj2 generates code for a $\texttt{CatProjNode}_2\ (n')$ by emitting code generated by compiling n_s with a *different* yield continuation:

$$y'(e) = \texttt{if b then } y(e) \texttt{ else \{if } e \texttt{ == CatPunc then \{b := True;S\} else S\}}$$

If `b` is set, we are in the "forward along" phase of pulling from $\texttt{CatProjNode}_2\ (-)$, so we throw e to the original yield continuation y. Otherwise, we set `b` if the element we just pulled is the `CatPunc`, run the skip continuation either way.

Note that, like the corresponding semantics rules from Section 5.7.2, the compilation scheme encoded by these rules fundementally depends on the guarantees provided by the λ^y type system. Attemping to first run code generated by CS-CatProj2 and then run code generated by CS-CatProj1—the opposite order from what is requred by the orderedness checker— will produce nonsensical results.

InL and InR The rules CS-InL and CS-InR straightforwardly generate code implementing the first and second projection. They both generate conditionals that first check their corresponding flags `b`, which stores the state of wether or not the requisite `PlusA` or `PlusB` punctuation mark has been produced. If the flag is not set, we set it, and throw the punctuation mark to the continuation y. If the flag has been set, we simply run the program `P` compiled from the source stream.

Case The rule CS-Case generates code implementing the three-state state machine for case analysis on streams of type $s + t$. The generated code branches on the state `k` to run the appropriate code for each state. In state `0`, the code runs `P`, which is the result of compiling the scrutinee stream n' with a modified

yield continuation:

$$y'(e) = \texttt{\{if } e \texttt{ == PlusA then k := 1 else k := 2\};S}$$

This continuation inspects each event from n' to check if it is the `PlusA` or `PlusB` punctuation mark, and updates the state variable `k` accordingly before running the skip continuation. Once the state has been set, subsequent pulls will take the appropriate branch, running either `P1` (the code compiled from n_1) if `k` is 1, or `P2` (the code compiled from n_2) if `k` is 2.

Fix and Rec The rules CS-Fix and CS-Rec generate code for recursive operations. Recall from Section 5.7.2 that the graph construction ensures a recursive operation is enclosed by a `FixNode` (n') node, where n' points to the root of the operation's body, and recursive calls within the body are translated to `RecNode` (r) nodes pointing back to the enclosing recursion node. In the operational semantics, the rule G-Fix simply delegates to the body, while G-Rec implements the "jump" by calling the auxiliary judgment `reset` $(G, r) = G'$ to reset all mutable state fields in the subgraph rooted at r back to their initial values.

The compiler implements this behavior directly. The rule CS-Fix generates code by simply compiling the body n' with the same continuations—a `FixNode` (n') is entirely transparent during code generation, serving only as a marker for the extent of a recursive subgraph. The rule CS-Rec generates code that performs the reset operation. It looks up the enclosing `FixNode` (r) and emits the code `P` generated by the auxiliary judgment `compileReset` (G, r) = `P`, which walks the subgraph rooted at r and generates statements to reset all mutable state variables to their initial values (stopping at nested `FixNode` $(-)$ nodes and source nodes, which maintain their own state). After executing this reset code, the next pull from the `FixNode` (r) will effectively restart the recursive function from its initial state, implementing the tail-recursive "jump" semantics.

Source Recall from Section 5.7.2 that in the operational semantics, source nodes `SourceNode` $(\langle x, \texttt{f}\rangle)$ have two components: a functional state x and a step function `f`, with the state threaded through the semantics rules G-Source-Done, G-Source-Skip, and G-Source-Yield. However, as a precondition of the compiler, we assume that all source nodes have been replaced by imperative pull streams `SourceNode` $(\langle \texttt{f}\rangle)$ where the step function `f` encapsulates and mutates its own state internally, rather than threading it func-

tionally[25]. The rule CS-Source generates code that simply invokes this imperative step function. The generated code calls `f()` and stores the result in a temporary variable `a`, then branches on the result: if `a` is `None`, the source has skipped, so we run the skip continuation `S`; if `a` is `DONE`, we run the done continuation `D`; otherwise, `a` must be an event, so we throw it to the yield continuation y. Since source nodes encapsulate their own state, the compiler does not need to allocate or manage any additional state variables for them.

6.1.4 Compiling Recursive Resets

The rule CS-Rec for recursive calls uses the auxiliary judgment `compileReset` (G, n) = `P` to generate code that resets a recursive block. The rules for this judgment are found in Figure 6.7. This judgment and its rules mirror those of the judgment `reset` $(G, n) = G'$ from Section 5.7.2, but generates code that mutates the states to their initial settings[26].

6.1.5 Compiling the Initial State

The judgment `compileInit` (G, n) = `I` generates code `I` that initializes the state variables for the the subgraph rooted at n. The rules for this judgment (Figure 6.8) are nearly identical to those for `compileReset` (G, n) = `P`. The key difference is that initialization must initialize *everything*, including nested recursive functions, while the reset judgment stops at nested `FixNode` $(-)$ boundaries. This is reflected in rule CI-Fix, which recursively initializes the body of a fix node, whereas CR-Fix stops at nested recursions.

6.2 Yoink

To further prove out the ideas of this chapter, I've implemented Yoink[27] a small Python eDSL based on λ^y. Yoink implements the type system, semantics, and compiler of λ^y, allowing programmers to write stream processing code in functional style embedded in Python, and then compile it to Python iterators.

[25] Indeed, the compiler is only correct if this replacement has been done *correctly*, and functional source streams have been replaced by equivalent imperative ones.

[26] Eagle-eyed readers may note that because these graphs are DAGs (with join-points at coordinator nodes), uses of CR-CatProj1 and CR-CatProj2 may generate duplicate reset code. This is semantically fine, but a bit wasteful. The implementation of Yoink ensures that it only traverses each node at most once.

[27] https://github.com/alpha-convert/Yoink

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\texttt{b}, n_1, n_2) \quad \texttt{compileReset}\,(G, n_1) = \texttt{P1} \quad \texttt{compileReset}\,(G, n_2) = \texttt{P2}}{\texttt{compileReset}\,(G, n) = \texttt{b := False; P1; P2}}\ \text{CR-Cat-R}$$

$$\frac{G[n] \Rightarrow \texttt{CatCoordNode}\,(\texttt{b}, n') \quad \texttt{compileReset}\,(G, n') = \texttt{P}}{\texttt{compileReset}\,(G, n) = \texttt{b := False; P}}\ \text{CR-Coord}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \quad \texttt{compileReset}\,(G, n') = \texttt{P}}{\texttt{compileReset}\,(G, n) = \texttt{P}}\ \text{CR-CatProj1}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \quad \texttt{compileReset}\,(G, n') = \texttt{P}}{\texttt{compileReset}\,(G, n) = \texttt{P}}\ \text{CR-CatProj2}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(\texttt{b}, n') \quad \texttt{compileReset}\,(G, n') = \texttt{P}}{\texttt{compileReset}\,(G, n) = \texttt{b := False; P}}\ \text{CR-InL}$$

$$\frac{G[n] \Rightarrow \texttt{InRNode}\,(\texttt{b}, n') \quad \texttt{compileReset}\,(G, n') = \texttt{P}}{\texttt{compileReset}\,(G, n) = \texttt{b := False; P}}\ \text{CR-InR}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(\texttt{k}, n', n_1, n_2) \quad \texttt{compileReset}\,(G, n') = \texttt{P0} \quad \texttt{compileReset}\,(G, n_1) = \texttt{P1} \quad \texttt{compileReset}\,(G, n_2) = \texttt{P2}}{\texttt{compileReset}\,(G, n) = \texttt{k := 0; P0; P1; P2}}\ \text{CR-Case}$$

$$\frac{G[n] \Rightarrow \texttt{EpsNode}}{\texttt{compileReset}\,(G, n) = \texttt{skip}}\ \text{CR-Eps} \qquad \frac{G[n] \Rightarrow \texttt{IntNode}\,(\texttt{b}, k)}{\texttt{compileReset}\,(G, n) = \texttt{b := False}}\ \text{CR-Int}$$

$$\frac{G[n] \Rightarrow \texttt{FixNode}\,(n')}{\texttt{compileReset}\,(G, n) = \texttt{skip}}\ \text{CR-Fix} \qquad \frac{G[n] \Rightarrow \texttt{RecNode}\,(n')}{\texttt{compileReset}\,(G, n) = \texttt{skip}}\ \text{CR-Rec}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle \texttt{f} \rangle)}{\texttt{compileReset}\,(G, n) = \texttt{skip}}\ \text{CR-Source}$$

Figure 6.7: Recursive Reset Compilation Rules

$$\frac{G[n] \Rightarrow \texttt{CatRNode}\,(\mathsf{b}, n_1, n_2) \quad \texttt{compileInit}\,(G, n_1) = \texttt{I1} \quad \texttt{compileInit}\,(G, n_2) = \texttt{I2}}{\texttt{compileInit}\,(G, n) = \mathsf{b}\ \texttt{:= False; I1; I2}}\ \text{CI-Cat-R}$$

$$\frac{G[n] \Rightarrow \texttt{CatCoordNode}\,(\mathsf{b}, n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \mathsf{b}\ \texttt{:= False; I}}\ \text{CI-Coord}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_1\,(n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \texttt{I}}\ \text{CI-CatProj1}$$

$$\frac{G[n] \Rightarrow \texttt{CatProjNode}_2\,(n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \texttt{I}}\ \text{CI-CatProj2}$$

$$\frac{G[n] \Rightarrow \texttt{InLNode}\,(\mathsf{b}, n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \mathsf{b}\ \texttt{:= False; I}}\ \text{CI-InL}$$

$$\frac{G[n] \Rightarrow \texttt{InRNode}\,(\mathsf{b}, n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \mathsf{b}\ \texttt{:= False; I}}\ \text{CI-InR}$$

$$\frac{G[n] \Rightarrow \texttt{CaseNode}\,(\mathsf{k}, n', n_1, n_2) \quad \texttt{compileInit}\,(G, n') = \texttt{I0} \quad \texttt{compileInit}\,(G, n_1) = \texttt{I1} \quad \texttt{compileInit}\,(G, n_2) = \texttt{I2}}{\texttt{compileInit}\,(G, n) = \mathsf{k}\ \texttt{:= 0; I0; I1; I2}}\ \text{CI-Case}$$

$$\frac{G[n] \Rightarrow \texttt{EpsNode}}{\texttt{compileInit}\,(G, n) = \texttt{skip}}\ \text{CI-Eps} \qquad \frac{G[n] \Rightarrow \texttt{IntNode}\,(\mathsf{b}, k)}{\texttt{compileInit}\,(G, n) = \mathsf{b}\ \texttt{:= False}}\ \text{CI-Int}$$

$$\frac{G[n] \Rightarrow \texttt{FixNode}\,(n') \quad \texttt{compileInit}\,(G, n') = \texttt{I}}{\texttt{compileInit}\,(G, n) = \texttt{I}}\ \text{CI-Fix} \qquad \frac{G[n] \Rightarrow \texttt{RecNode}\,(n')}{\texttt{compileInit}\,(G, n) = \texttt{skip}}\ \text{CI-Rec}$$

$$\frac{G[n] \Rightarrow \texttt{SourceNode}\,(\langle \mathsf{f} \rangle)}{\texttt{compileInit}\,(G, n) = \texttt{skip}}\ \text{CI-Source}$$

Figure 6.8: Initial State Compilation Rules

6.2.1 Implementation of Yoink

Yoink is implemented as a small Python library that uses a tracing decorator for compilation, adopting an approach similar to that of Jax or Triton [70, 176]. The surface syntax is essentially a very lightweight shim over writing λ^{Y} terms directly: more work would be required to make it feel ergonomic and natural to program in. Functions in Yoink are written as Python methods annotated with a `@Yoink.jit` decorator, with the signature `def f(yoink, ...): ...`. The first argument, `yoink`, is a special object that serves as the gateway to λ^{Y}'s term constructors. The rest of the arguments are stream arguments to the function, implicitly in parallel. Inside the body of such a function, you build λ^{Y} terms by calling methods on this `yoink` object.

Each λ^{Y} term former has a corresponding method on the `yoink` object. For example, `yoink.inl(x)` and `yoink.cons(y, z)` correspond to the `inl` and `cons` constructors, respectively. Elimination forms work similarly: `yoink.catl(z)` implements the cat-elimination form and returns a tuple of the two stream components, allowing you to write `(x, y) = yoink.catl(z)` in the natural Python style. Case analysis forms like plus-elimination are written by passing Python functions that implement the branch bodies—for instance, `yoink.case(x, lambda y: _, lambda z: _)`. Beyond the core λ^{Y} features described in Chapter 5, the language includes a space-bounded version of the `wait` operation from Delta, which allows you to buffer and compute on values so long as they do not require unbounded space to store. The language also supports higher-order functions as in Delta.

When you define a function decorated with `@Yoink.jit`, the decorator executes the function body with symbolic values for the inputs, tracing the evaluation to build a λ^{Y} term. Along the way, the term is typechecked with an algorithmic form of the ordered type system from Chapter 5. By collecting a partially ordered set of variables ordered by their usages, we can ensure that no disallowed usages occur. Additionally, type annotations can be given to top-level function arguments; these are then checked (essentially bidirectionally [146]) at intro and elimination forms.

After typechecking completes successfully, the result is a pull graph representation of the program. At this point, users have two options for execution. First, you can directly run the resulting pull graph using the interpreter, which directly implements the operational semantics from Section 5.7.2. Alternatively,

you can compile the pull graph to a Python iterator using an implementation of the compiler described in this chapter. The compiler implementation uses Python's `ast` module to generate a class with an iterator implementation that executes the step function, producing efficient imperative code that can be used anywhere Python iterators are expected.

To ensure correctness, I have done extensive differential testing using the Hypothesis property-based testing framework. For each test program, we generate sequences of input events and ensure that the compiler and interpreter agree on the output events. In lieu of a proof, this is reasonably strong empirical vidence that the compiler is semantics-preserving.

6.2.2 Examples of Yoink Programs

Below we show some examples of Yoink programs of increasing complexity. These examples show both the source Python code, as well as the iterator classes that they compile to. Both the source code and compiled could use some improvement in future work. The functional style is not as clean as one might like in the source Python, owing mostly to Python's disallowing of multi-line lambdas. The compiled code works, but is extremely verbose due to the full fusion and some code duplication. We discuss both of these facts in Chapter 7. The genderated code in all examples has been cleaned up from the direct compiler output: identifiers have been de-mangled, and some lines have been collapsed to save space.

Figure 6.9 shows the identity function on a stream of type `Int**`. Note that the code generated by the identity function at any type is the same: we simply pull on the input until it is exhausted. Figures 6.10 and 6.11 show the first and second projection operations on a stream of concatenation type. These demonstrate the shared state of the coordinator, rendered as `seen_punc` in each. Figure 6.12 shows the recursive definition of the map function, as well as the compiled output of the identity map operation. Figure 6.13 demonstrates a more interesting use of the map function, adding one to every element in a stream of `Int`s. This also makes use of the `wait` operation present in Yoink that is absent from its formalization in λ^{y}. Uses of `wait` compile to bounded-sized buffers that contain the events on the waited stream. Once it's complete, the values can be modified and then re-serialized with `emit`. Finally, Figure 6.14 shows a fully-fused use of `concat_map`.

Source:

```
ints = TyStar(Singleton(int))
@Yoink.jit
def id(yoink, s : ints):
  return s
```

Compiled:

```
class CompiledIter:
  def __init__(self, *input_iterators):
      self.inputs = list(input_iterators)
      pass

  def __next__(self):
      try:
          tmp_0 = next(self.inputs[0])
          result = tmp_0
      except StopIteration:
          result = DONE
      if result is DONE:
          raise StopIteration()
      return result
```

Figure 6.9: Identity Function in Yoink

Source:

```
@Yoink.jit
def proj1(yoink,
    s : TyCat(Singleton(int),
              Singleton(int))):
    (a,b) = yoink.catl(s)
    return a
```

Compiled:

```
class CompiledIter:
  def __init__(self, *input_iterators):
      self.inputs = list(input_iterators)
      self.seen_punc = False
      self.input_exhausted = False

  def __next__(self):
      if self.input_exhausted:
          result = DONE
      elif self.seen_punc:
          result = DONE
      else:
          try:
              tmp_0 = next(self.inputs[0])
              if isinstance(tmp_0, CatEvA):
                  result = tmp_0.value
              elif isinstance(tmp_0, CatPunc):
                  self.seen_punc = True
                  result = DONE
              else:
                  result = None
          except StopIteration:
              self.input_exhausted = True
              result = DONE
      if result is DONE:
          raise StopIteration()
      return result
```

Figure 6.10: First projection in Yoink

Source:

```
@Yoink.jit
def proj2(yoink,
    s : TyCat(Singleton(int),
              Singleton(int))):
    (a,b) = yoink.catl(s)
    return b
```

Compiled:

```
class CompiledIter:
  def __init__(self, *input_iterators):
      self.inputs = list(input_iterators)
      self.seen_punc = False
      self.input_exhausted = False

  def __next__(self):
      if self.input_exhausted:
          result = DONE
      else:
          try:
              tmp_0 = next(self.inputs[0])
              if not self.seen_punc:
                  if isinstance(tmp_0, CatEvA):
                      result = None
                  elif isinstance(tmp_0, CatPunc):
                      self.seen_punc = True
                      result = None
                  else:
                      result = None
              else:
                  result = tmp_0
          except StopIteration:
              self.input_exhausted = True
              result = DONE
      if result is DONE:
          raise StopIteration()
      return result
```

Figure 6.11: Second projection in Yoink

Source:

```
def map(yoink,x,map_fn,
        result_type):
  def build_body(rec):
    def cons_case(hd,tl):
      out = map_fn(hd)
      return yoink.cons(out,rec)
    return yoink.starcase(x,
      lambda _: yoink.nil(),
      cons_case)
  return yoink.fix(build_body,
                   result_type)

ints = TyStar(Singleton(int))

@Yoink.jit
def map_id(yoink, s : ints):
  return map(yoink,s,
    lambda x: x,ints)
```

Compiled:

```
class CompiledIter:
  def __init__(self, *input_iterators):
      self.inputs = list(input_iterators)
      self.nil_tag_emitted = False;  self.catr_state = 0
      self.seen_punc = False;        self.input_exhausted = False
      self.case_tag_read = False;    self.active_branch = -1
      self.cons_tag_emitted = False

  def __next__(self):
      if not self.case_tag_read:
          try:
              tmp_0 = next(self.inputs[0]);  self.case_tag_read = True
              if isinstance(tmp_0, PlusPuncA):   self.active_branch = 0
              elif isinstance(tmp_0, PlusPuncB): self.active_branch = 1
              else: raise RuntimeError(f'Expected PlusPuncA or PlusPuncB')
              result = None
          except StopIteration: result = DONE
      elif self.active_branch == 0:
          if not self.nil_tag_emitted: self.nil_tag_emitted = True; result = PlusPuncA()
          else: result = DONE
      elif not self.cons_tag_emitted: self.cons_tag_emitted = True; result = PlusPuncB()
      elif self.catr_state == 0:
          if self.input_exhausted:   self.catr_state = 1; result = CatPunc()
          elif self.seen_punc:       self.catr_state = 1; result = CatPunc()
          else:
              try:
                  tmp_1 = next(self.inputs[0])
                  if isinstance(tmp_1, CatEvA):     result = CatEvA(tmp_1.value)
                  elif isinstance(tmp_1, CatPunc):
                      self.seen_punc = True; self.catr_state = 1; result = CatPunc()
                  else: result = None
              except StopIteration:
                  self.input_exhausted = True; self.catr_state = 1; result = CatPunc()
      else:  # reset state for next iteration
          self.nil_tag_emitted = False; self.catr_state = 0; self.seen_punc = False
          self.input_exhausted = False; self.case_tag_read = False
          self.active_branch = -1; self.cons_tag_emitted = False; result = None
      if result is DONE: raise StopIteration()
      return result
```

Figure 6.12: Map in Yoink

Source:

```
@Yoink.jit
def map_add_one(yoink,
                s : ints):
  def add_one(x):
    y = yoink.wait(x)
    return yoink.emit(y+1)
  return map(yoink,s,
             add_one,ints)
```

Compiled:

```
class CompiledIter:
  def __init__(self, *input_iterators):
      self.inputs = list(input_iterators)
      self.nil_tag_emitted = False;  self.case_tag_read = False
      self.active_branch = -1;       self.seen_punc = False
      self.input_exhausted = False;  self.emit_phase = 1
      self.emit_buffer = None;       self.emit_index = 0
      self.catr_state = 0;           self.first_exhausted = False
      self.wait_buffer = [None];     self.wait_buffer_idx = 0
      self.cons_tag_emitted = False; self.const_buf = [None]
      self.binop_buf = [None]

def __next__(self):
      if not self.case_tag_read:
          try:
              tmp_0 = next(self.inputs[0]);  self.case_tag_read = True
              if isinstance(tmp_0, PlusPuncA):   self.active_branch = 0
              elif isinstance(tmp_0, PlusPuncB): self.active_branch = 1
              else: raise RuntimeError(f'Expected PlusPuncA or PlusPuncB')
              result = None
          except StopIteration: result = DONE
      elif self.active_branch == 0:
          if not self.nil_tag_emitted: self.nil_tag_emitted = True; result = PlusPuncA()
          else: result = DONE
      elif not self.cons_tag_emitted: self.cons_tag_emitted = True; result = PlusPuncB()
      elif self.catr_state == 0:
          if not self.first_exhausted:
              if self.input_exhausted:   self.first_exhausted = True; result = None
              elif self.seen_punc:       self.first_exhausted = True; result = None
              else:
                  try:
                      tmp_1 = next(self.inputs[0])
                      if isinstance(tmp_1, CatEvA):
                          self.wait_buffer[self.wait_buffer_idx] = tmp_1.value; result = None
                      elif isinstance(tmp_1, CatPunc):
                          self.seen_punc = True; self.first_exhausted = True; result = None
                      else: result = None
                  except StopIteration:
                      self.input_exhausted = True; self.first_exhausted = True; result = None
          elif self.emit_phase == 1:
              self.const_buf[0] = BaseEvent(1)
              self.binop_buf[0] = BaseEvent(self.wait_buffer[0].value + self.const_buf[0].value)
              self.emit_index = 0; self.emit_phase = 2; result = None
          elif self.emit_index < len(self.binop_buf):
              result = CatEvA(self.binop_buf[self.emit_index]); self.emit_index += 1
          else: self.catr_state = 1; result = CatPunc()
      else:  # reset state for next iteration
          self.nil_tag_emitted = False; self.case_tag_read = False; self.active_branch = -1
          self.seen_punc = False; self.input_exhausted = False; self.emit_phase = 1
          self.emit_buffer = None; self.emit_index = 0; self.catr_state = 0
          self.first_exhausted = False; self.wait_buffer = [None]
          self.wait_buffer_idx = 0; self.cons_tag_emitted = False; result = None
      if result is DONE: raise StopIteration()
      return result
```

Figure 6.13: Map with body in Yoink

Source:

```
intss = TyStar(TyStar(
          Singleton(int)))

@Yoink.jit
def concatmap_adds(yoink,
                   s : intss):
    def add_one(x):
        y = yoink.wait(x)
        return yoink.emit(y+1)
    def map_add_one(xs):
        return yoink.map(
            xs, add_one)
    return yoink.concat_map(
        s, map_add_one)
```

Compiled:

```
class FlattenedIterator:
    def __init__(self, *input_iterators):
        self.inputs = list(input_iterators)
        self.case1_tag_read = False; self.case1_branch = -1
        self.coord1_seen_punc = False; self.coord1_exhausted = False
        self.coord2_seen_punc = False; self.coord2_exhausted = False
        self.inj1_emitted = False; self.catr1_state = 0
        self.sink1_exhausted = False; self.inj2_emitted = False
        self.case2_tag_read = False; self.case2_branch = -1
        self.inj3_emitted = False; self.wait_buf = [None]; self.wait_idx = 0
        self.catr2_state = 0; self.case3_tag_read = False; self.case3_branch = -1
        self.inj4_emitted = False; self.emit_phase = 1
        self.emit_buf = None; self.emit_idx = 0
        self.sink2_exhausted = False; self.sink3_exhausted = False
        self.coord3_seen_punc = False; self.coord3_exhausted = False
        self.const_buf = [None]; self.binop_buf = [None]

    def __next__(self):
        if not self.case1_tag_read:
            try:
                t = next(self.inputs[0]); self.case1_tag_read = True
                if isinstance(t, PlusPuncA): self.case1_branch = 0
                elif isinstance(t, PlusPuncB): self.case1_branch = 1
                else: raise RuntimeError(f'Expected tag, got {t}')
                result = None
            except StopIteration: result = DONE
        elif self.case1_branch == 0:
            if not self.inj1_emitted: self.inj1_emitted = True; result = PlusPuncA()
            else: result = DONE
        elif not self.case3_tag_read:
            if not self.case2_tag_read:
                if self.coord1_exhausted or self.coord1_seen_punc: result = DONE
                else:
                    try:
                        t = next(self.inputs[0])
                        if isinstance(t, CatEvA):
                            self.case2_tag_read = True
                            if isinstance(t.value, PlusPuncA): self.case2_branch = 0
                            elif isinstance(t.value, PlusPuncB): self.case2_branch = 1
                            else: raise RuntimeError(f'Expected tag')
                            result = None
                        elif isinstance(t, CatPunc): self.coord1_seen_punc = True; result = DONE
                        else: result = None
                    except StopIteration: self.coord1_exhausted = True; result = DONE
            elif self.case2_branch == 0:
                if not self.inj3_emitted:
                    self.inj3_emitted = True; self.case3_tag_read = True; self.case3_branch = 0; result = None
                else: result = DONE
            elif not self.inj2_emitted:
                self.inj2_emitted = True; self.case3_tag_read = True; self.case3_branch = 1; result = None
            # continued in Figure~\ref{fig:yoink-concat-map-adds-2}...
```

Figure 6.14: Concat Map in Yoink(Part 1)

Compiled (continued):

```
        # ...
        elif self.catr1_state == 0:
            if not self.sink2_exhausted:
                if self.coord3_exhausted or self.coord3_seen_punc: self.sink2_exhausted = True; result = None
                elif self.coord1_exhausted or self.coord1_seen_punc:
                    self.coord3_exhausted = True; self.sink2_exhausted = True; result = None
                else:
                    try:
                        t = next(self.inputs[0])
                        if isinstance(t, CatEvA):
                            if isinstance(t.value, CatEvA): self.wait_buf[self.wait_idx] = t.value.value; result = None
                            elif isinstance(t.value, CatPunc):
                                self.coord3_seen_punc = True; self.sink2_exhausted = True; result = None
                            else: result = None
                        elif isinstance(t, CatPunc):
                            self.coord1_seen_punc = True; self.coord3_exhausted = True
                            self.sink2_exhausted = True; result = None
                        else: result = None
                    except StopIteration:
                        self.coord1_exhausted = True; self.coord3_exhausted = True
                        self.sink2_exhausted = True; result = None
            elif self.emit_phase == 1:
                self.const_buf[0] = BaseEvent(1)
                self.binop_buf[0] = BaseEvent(self.wait_buf[0].value + self.const_buf[0].value)
                self.emit_idx = 0; self.emit_phase = 2; result = None
            elif self.emit_idx < len(self.binop_buf):
                ev = CatEvA(self.binop_buf[self.emit_idx]); self.case3_tag_read = True
                if isinstance(ev, PlusPuncA): self.case3_branch = 0
                elif isinstance(ev, PlusPuncB): self.case3_branch = 1
                else: raise RuntimeError(f'Expected tag')
                result = None; self.emit_idx += 1
            else: self.catr1_state = 1; self.case3_tag_read = True; self.case3_branch = 1; result = None
        elif not self.sink3_exhausted:
            if self.coord3_exhausted or self.coord3_seen_punc: self.sink3_exhausted = True; result = None
            elif self.coord1_exhausted or self.coord1_seen_punc:
                self.coord3_exhausted = True; self.sink3_exhausted = True; result = None
            else:
                try:
                    t = next(self.inputs[0])
                    if isinstance(t, CatEvA):
                        if isinstance(t.value, CatEvA): result = None
                        elif isinstance(t.value, CatPunc):
                            self.coord3_seen_punc = True; self.sink3_exhausted = True; result = None
                        else: result = None
                    elif isinstance(t, CatPunc):
                        self.coord1_seen_punc = True; self.coord3_exhausted = True
                        self.sink3_exhausted = True; result = None
                    else: result = None
                except StopIteration:
                    self.coord1_exhausted = True; self.coord3_exhausted = True
                    self.sink3_exhausted = True; result = None
        else:  # reset inner loop
            self.wait_buf = [None]; self.wait_idx = 0; self.inj2_emitted = False
            self.emit_phase = 1; self.emit_buf = None; self.emit_idx = 0
            self.case2_tag_read = False; self.case2_branch = -1
            self.sink2_exhausted = False; self.catr1_state = 0
            self.sink3_exhausted = False; self.inj3_emitted = False
            self.coord3_seen_punc = False; self.coord3_exhausted = False; result = None
    elif self.case3_branch == 0:
        if not self.sink1_exhausted:
            if self.coord1_exhausted or self.coord1_seen_punc: self.sink1_exhausted = True; result = None
            else:
                try:
                    t = next(self.inputs[0])
                    if isinstance(t, CatEvA): result = None
                    elif isinstance(t, CatPunc):
                        self.coord1_seen_punc = True; self.sink1_exhausted = True; result = None
                    else: result = None
                except StopIteration: self.coord1_exhausted = True; self.sink1_exhausted = True; result = None
        # continued...
```

Figure 6.15: Concat Map in Yoink(Part 2)

Compiled (continued):

```
        # ...
        else:  # reset outer state
            self.case1_tag_read = False; self.case1_branch = -1
            self.coord1_seen_punc = False; self.coord1_exhausted = False
            self.coord2_seen_punc = False; self.coord2_exhausted = False
            self.inj1_emitted = False; self.catr1_state = 0
            self.sink1_exhausted = False; self.inj2_emitted = False
            self.case2_tag_read = False; self.case2_branch = -1
            self.inj3_emitted = False; self.wait_buf = [None]; self.wait_idx = 0
            self.catr2_state = 0; self.case3_tag_read = False; self.case3_branch = -1
            self.inj4_emitted = False; self.emit_phase = 1; self.emit_buf = None; self.emit_idx = 0
            self.sink2_exhausted = False; self.sink3_exhausted = False
            self.coord3_seen_punc = False; self.coord3_exhausted = False; result = None
    elif not self.inj4_emitted: self.inj4_emitted = True; result = PlusPuncB()
    elif self.catr2_state == 0:
        if self.coord2_exhausted or self.coord2_seen_punc: self.catr2_state = 1; result = CatPunc()
        elif not self.case2_tag_read:
            if self.coord1_exhausted or self.coord1_seen_punc:
                self.coord2_exhausted = True; self.catr2_state = 1; result = CatPunc()
            else:
                try:
                    t = next(self.inputs[0])
                    if isinstance(t, CatEvA):
                        self.case2_tag_read = True
                        if isinstance(t.value, PlusPuncA): self.case2_branch = 0
                        elif isinstance(t.value, PlusPuncB): self.case2_branch = 1
                        else: raise RuntimeError(f'Expected tag')
                        result = None
                    elif isinstance(t, CatPunc):
                        self.coord1_seen_punc = True; self.coord2_exhausted = True
                        self.catr2_state = 1; result = CatPunc()
                    else: result = None
                except StopIteration:
                    self.coord1_exhausted = True; self.coord2_exhausted = True
                    self.catr2_state = 1; result = CatPunc()
        elif self.case2_branch == 0:
            if not self.inj3_emitted: self.inj3_emitted = True; result = None
            else: self.coord2_exhausted = True; self.catr2_state = 1; result = CatPunc()
        elif not self.inj2_emitted: self.inj2_emitted = True; result = None
        elif self.catr1_state == 0:
            if not self.sink2_exhausted:
                if self.coord3_exhausted or self.coord3_seen_punc: self.sink2_exhausted = True; result = None
                elif self.coord1_exhausted or self.coord1_seen_punc:
                    self.coord3_exhausted = True; self.sink2_exhausted = True; result = None
                else:
                    try:
                        t = next(self.inputs[0])
                        if isinstance(t, CatEvA):
                            if isinstance(t.value, CatEvA): self.wait_buf[self.wait_idx] = t.value.value; result = None
                            elif isinstance(t.value, CatPunc):
                                self.coord3_seen_punc = True; self.sink2_exhausted = True; result = None
                            else: result = None
                        elif isinstance(t, CatPunc):
                            self.coord1_seen_punc = True; self.coord3_exhausted = True
                            self.sink2_exhausted = True; result = None
                        else: result = None
                    except StopIteration:
                        self.coord1_exhausted = True; self.coord3_exhausted = True
                        self.sink2_exhausted = True; result = None
            elif self.emit_phase == 1:
                self.const_buf[0] = BaseEvent(1)
                self.binop_buf[0] = BaseEvent(self.wait_buf[0].value + self.const_buf[0].value)
                self.emit_idx = 0; self.emit_phase = 2; result = None
            elif self.emit_idx < len(self.binop_buf):
                ev = CatEvA(self.binop_buf[self.emit_idx])
                if isinstance(ev, CatEvA): result = CatEvA(ev.value)
                elif isinstance(ev, CatPunc):
                    self.coord2_seen_punc = True; self.catr2_state = 1; result = CatPunc()
                else: result = None
                self.emit_idx += 1
            else:
                self.catr1_state = 1
                if isinstance(CatPunc(), CatPunc):
                    self.coord2_seen_punc = True; self.catr2_state = 1; result = CatPunc()
                else: result = None
        elif not self.sink3_exhausted:
            if self.coord3_exhausted or self.coord3_seen_punc: self.sink3_exhausted = True; result = None
            elif self.coord1_exhausted or self.coord1_seen_punc:
                self.coord3_exhausted = True; self.sink3_exhausted = True; result = None
            else:
                try:
                    t = next(self.inputs[0])
                    if isinstance(t, CatEvA):
                        if isinstance(t.value, CatEvA): result = None
                        elif isinstance(t.value, CatPunc):
                            self.coord3_seen_punc = True; self.sink3_exhausted = True; result = None
                        else: result = None
                    elif isinstance(t, CatPunc):
                        self.coord1_seen_punc = True; self.coord3_exhausted = True
                        self.sink3_exhausted = True; result = None
                    else: result = None
                except StopIteration:
                    self.coord1_exhausted = True; self.coord3_exhausted = True
                    self.sink3_exhausted = True; result = None
        else:  # reset inner loop
            self.wait_buf = [None]; self.wait_idx = 0; self.inj2_emitted = False
            self.emit_phase = 1; self.emit_buf = None; self.emit_idx = 0
            self.case2_tag_read = False; self.case2_branch = -1
            self.sink2_exhausted = False; self.catr1_state = 0
            self.sink3_exhausted = False; self.inj3_emitted = False
            self.coord3_seen_punc = False; self.coord3_exhausted = False; result = None
    else:  # reset catr2
        self.catr2_state = 0; self.case3_tag_read = False; self.case3_branch = -1
        self.inj4_emitted = False; self.coord2_seen_punc = False; self.coord2_exhausted = False; result = None
    if result is DONE: raise StopIteration()
    return result
```

Figure 6.16: Concat Map in Yoink(Part 3)

Chapter 7

Future Work

Thus concludes our journey into ordered types for stream processing. But the fun need not end here! This document leaves several threads unpulled; several streams of thought unfollowed. We enumerate a few below.

7.1 Proofs about λ^{Y}

The work of Chapters 5 and 6 leave two proof obligations outstanding. These proofs would both be (a) fun, and (b) required work to complete the story of λ^{Y} to a level where it could be submitted for publication.

The first is the soundness theorem for λ^{Y} outlined in Chapter 5. This proof is a little thorny: despite the step functions being nonrecursive and terminating, pull streams are fundamentally coinductive objects, and soundness is about the multi-step behavior of streams. To avoid dealing with this coinductive structure directly in this document, we phrased the theorem in terms of safety for an arbitrary number of steps. Despite this simplification, completing the proof would probably still require employing proof techniques used in coinduction or the study of labeled transition systems [164].

The second outstanding proof is the compiler correctness theorem for the compiler in Chapter 6. Setting up this theorem is relatively straightforward: we want the semantics to serve as a Hoare logic specification for the compiled code. That is, if we start in a valid state and run the code for the generated step function, we should end up in the state produced by the step function (and potentially yield the same value). Most of this proof is likely compositional over the structure of the pull graph and generated code. For most of the cases, the only "interesting" bit is setting up a binary relation between states of the step function and variables in the generated code. However, the truly challenging aspect is probably the invariants about how the aliased state is mutated through concatenation projection. Structurally, we could probably use a lot of off-the-shelf tools to accomplish this proof, since it's fundamentally a compiler correctness proof (an extremely well-studied area [110, 121]). One interesting option is to use ITrees, both for modeling the coinductive structure of the pull graph semantics of λ^{Y}, *and* for modeling the imperative

language used as the target. Another option would be to use a program logic like Iris.

7.2 Further Implementation of Yoink

At present, the Yoink implementation is in a relatively preliminary state, and other research directions would likely emerge from continuing to build it out. One natural next step would be to integrate with existing streaming libraries in Python, allowing Yoink programs to interoperate with established streaming ecosystems. Additionally, writing more example programs and attempting to port examples from Delta would help validate the design, identify remaining extensions that would be required to make the it practical.

Another issue with Yoink is code size: the Python code generated is often quite large (as demonstrated in Chapter 6). Some of this is an inevitable property of "full" fusion, wher inlining of definitions must blow up code size somewhat. However, some of this could be fixed with a smarter compilation strategy. In particular, compiling the case expression eliminators for plus and star introduces significant overhead since we must duplicate the code for the scrutinee: once in the bit of the code that searches for the tag, and then in both branhches.

Finally, exploring other compilation targets could expand Yoink's applicability: for instance, targeting Rust iterators would be straightforward— it essentially just requires emitting code in a different syntax— and might enable compilation to embedded devices.

7.3 Combinator Safety

Much of the content of this dissertation document would seem relatively anti-combinator. Indeed, we have just concluded a long quest to free stream programmers from the confines of their combinator-only programming style. But combinators are not *always* bad! There's a reason that the set of list combinators were adopted into streaming programming in the first place: the programming patterns they encapsulate are in fact ubiquitous, and having to always rewrite the recursion every time would be tedious.

It turns out that Yoink has an interesting design side effect, enirely unrelated to its original goals: its makes it possible to safely use a larger set of streaming combinators than other languages of imperative

pull streams. As an example consider the composition of combinators, where s is a stream and n is an integer:

```
concat(drop(n,s),take(n,s))
```

What does this do? If these combinators are implemented imperatively, this program has very strange behavior: if $n = 3$ and s produces $\langle 1, 2, 3, 4, 5\rangle$, the result is just $\langle 4, 5\rangle$. If these combinators (and s) were implemented with *functional* pull streams, the result would be the (potentialy expected) $\langle 4, 5, 1, 2, 3\rangle$. With imperative pull streams, the two components of the `concat` alias and mutably update each other's state (the state of `s`), and clobber each other. Different imperative streaming libraries tackle this problem differently. Some carefully restrict their set of combinators to disallow combinations that would have behaviors different from their functional equivalents [105, 108]. Others like Rust's iterators use existing type system features to prevent aliased mutable state [162]. Last, some (like Python and Java's iterators) leave this up to programmers.

Yoink prevents this bug entirely by tye type of `concat`, which requires the first argument to come before the second. It also prevents some bugs similar to the ones Rust's linear iterators prevents, by requiring that combinators like `zip` accept parallel arguments. It would be very interesting to investigate this in future work. The ordered type system of λ^y is relatively lightweight and simple to implement, and could concievably be bolted onto existing libraries to safely extend their set of combinators.

APPENDIX A

λ^{ST} Definitions

Mechanized versions of all definitions and proofs about λ^{ST} can be founed at https://github.com/alpha-convert/lambda-st-proofs.

A.1 Basics

Stream types are defined by the following grammar. The base types included are the unit type 1 which types streams that contain exactly one unit element, the type of the empty stream ε, and the type of streams consisting of a single integer, `Int`. Larger types include the concatenation type $s \cdot t$, the sum type $s + t$, the parallel stream type $s \| t$, and the star type $s^\star$.

$$s, t, r := \quad 1 \mid \varepsilon \mid \texttt{Int} \mid s \cdot t \mid s + t \mid s \| t \mid s^\star$$

Contexts in the stream types calculus system have a bunched structure. The context former $\Gamma \, , \Delta$ corresponds to the parallel type, while the context former $\Gamma \, ; \Delta$ corresponds to the concatenation type. The two context formers share a unit, written as "$\cdot$".

$$\Gamma ::= \quad \cdot \mid \Gamma \, , \Gamma \mid \Gamma \, ; \Gamma \mid x : s$$

A stream type is *null* if it includes no data. Null types are parallel combinations of εs.

Definition A.1.1 (Nullable)**.** *We define a judgment* s `null` *as follows:*

$$\frac{}{\varepsilon \ \texttt{null}} \qquad\qquad \frac{s \ \texttt{null} \qquad t \ \texttt{null}}{s \| t \ \texttt{null}}$$

We extend to contexts pointwise.

$$\frac{}{\cdot\ \mathtt{null}} \qquad \frac{s\ \mathtt{null}}{x : s\ \mathtt{null}} \qquad \frac{\Gamma\ \mathtt{null} \qquad \Gamma'\ \mathtt{null}}{\Gamma, \Gamma'\ \mathtt{null}} \qquad \frac{\Gamma\ \mathtt{null} \qquad \Gamma'\ \mathtt{null}}{\Gamma; \Gamma'\ \mathtt{null}}$$

Prefixes are also like in Chapter 4, with a definition p `maximal` for "complete" prefixes, and a typing relation p : `prefix` (s).

Definition A.1.2 (Prefixes)**.** *The grammar of prefixes is given by:*

$$\begin{aligned} p ::= \ & \mathtt{oneEmp} \mid \mathtt{oneFull} \mid \mathtt{epsEmp} \mid \mathtt{parPair}(p, p') \\ & \mid \mathtt{catFst}(p) \mid \mathtt{catBoth}(p, p') \\ & \mid \mathtt{sumEmp} \mid \mathtt{sumInl}(p) \mid \mathtt{sumInr}(p) \\ & \mid \mathtt{starEmp} \mid \mathtt{starDone} \\ & \mid \mathtt{starFirst}(p) \mid \mathtt{starRest}(p, p') \end{aligned}$$

Definition A.1.3 (Maximal Prefix)**.**

$$\frac{}{\mathtt{epsEmp\ maximal}} \qquad \frac{}{\mathtt{oneFull\ maximal}} \qquad \frac{p_1\ \mathtt{maximal} \qquad p_2\ \mathtt{maximal}}{\mathtt{parPair}(p_1, p_2)\ \mathtt{maximal}}$$

$$\frac{p_1\ \mathtt{maximal} \qquad p_2\ \mathtt{maximal}}{\mathtt{catBoth}(p_1, p_2)\ \mathtt{maximal}} \qquad \frac{p\ \mathtt{maximal}}{\mathtt{sumInl}(p)\ \mathtt{maximal}}$$

$$\frac{p\ \mathtt{maximal}}{\mathtt{sumInr}(p)\ \mathtt{maximal}} \qquad \frac{}{\mathtt{starDone\ maximal}} \qquad \frac{p\ \mathtt{maximal} \qquad p'\ \mathtt{maximal}}{\mathtt{starRest}(p, p')\ \mathtt{maximal}}$$

Definition A.1.4 (Well-Typed Prefixes)**.**

$$\frac{}{\mathtt{epsEmp} : \mathtt{prefix}(\varepsilon)} \qquad \frac{}{\mathtt{oneEmp} : \mathtt{prefix}(1)} \qquad \frac{}{\mathtt{oneFull} : \mathtt{prefix}(1)}$$

$$\frac{p_1 : \mathtt{prefix}(s) \qquad p_2 : \mathtt{prefix}(t)}{\mathtt{parPair}(p_1, p_2) : \mathtt{prefix}(s \| t)} \qquad \frac{p : \mathtt{prefix}(s)}{\mathtt{catFst}(p) : \mathtt{prefix}(s \cdot t)}$$

$$\frac{p_1 : \mathtt{prefix}(s) \qquad p_1\ \mathtt{maximal} \qquad p_2 : \mathtt{prefix}(t)}{\mathtt{catBoth}(p_1, p_2) : \mathtt{prefix}(s \cdot t)} \qquad \frac{}{\mathtt{sumEmp} : \mathtt{prefix}(s + t)}$$

$$\frac{p : \mathtt{prefix}(s)}{\mathtt{sumInl}(p) : \mathtt{prefix}(s + t)} \qquad \frac{p : \mathtt{prefix}(t)}{\mathtt{sumInr}(p) : \mathtt{prefix}(s + t)} \qquad \frac{}{\mathtt{starEmp} : \mathtt{prefix}(s^\star)}$$

$$\frac{}{\mathtt{starDone} : \mathtt{prefix}(s^\star)} \qquad \frac{p : \mathtt{prefix}(s)}{\mathtt{starFirst}(p) : \mathtt{prefix}(s^\star)}$$

$$\frac{p : \mathtt{prefix}(s) \qquad p\ \mathtt{maximal} \qquad p' : \mathtt{prefix}(s^\star)}{\mathtt{starRest}(p, p') : \mathtt{prefix}(s^\star)}$$

For each type s, we define the "empty" prefix $\mathtt{emp}_s$ inductively on the structure of s.

Definition A.1.5 (Empty Prefix)**.** *The empty prefix is defined as follows:*

(ε) $\mathtt{emp}_\varepsilon = \mathtt{epsEmp}$

(1) $\mathtt{emp}_1 = \mathtt{oneEmp}$

$(s \| t)$ $\mathtt{emp}_{s\|t} = \mathtt{parPair}(\mathtt{emp}_s, \mathtt{emp}_t)$

$(s + t)$ $\mathtt{emp}_{s+t} = \mathtt{sumEmp}$

$(s \cdot t)$ $\mathtt{emp}_{s\cdot t} = \mathtt{catFst}(\mathtt{emp}_s)$

$(s^\star)$ $\mathtt{emp}_{s^\star} = \mathtt{starEmp}$

We lift this to contexts in the natural way, with $\mathit{emp}_{\cdot} = \mathit{epsEmp}$, *and* $\mathit{emp}_{\Gamma,\Delta} = \mathit{catFst}(\mathit{emp}_{\Gamma})$, *and* $\mathit{emp}_{\Gamma,\Delta} = \mathit{parPair}(\mathit{emp}_{\Gamma}, \mathit{emp}_{\Delta})$.

Theorem A.1.1 (Empty Prefix is Well-Typed)**.** $\mathit{emp}_s : \mathit{prefix}(s)$

Definition A.1.6 (Prefix is Empty)**.**

$$\frac{}{\mathit{epsEmp}\ \mathit{empty}} \qquad \frac{}{\mathit{oneEmp}\ \mathit{empty}} \qquad \frac{p_1\ \mathit{empty} \qquad p_2\ \mathit{empty}}{\mathit{parPair}(p_1, p_2)\ \mathit{empty}}$$

$$\frac{p\ \mathit{empty}}{\mathit{catFst}(p)\ \mathit{empty}} \qquad \frac{}{\mathit{sumEmp}\ \mathit{empty}} \qquad \frac{}{\mathit{starEmp}\ \mathit{empty}}$$

Theorem A.1.2 (Empty Prefix Is Empty)**.** $\mathit{emp}_s\ \mathit{empty}$

Proof. Induction on s. □

Theorem A.1.3 (Empty And Maximal Means Nullable)**.** *If* $p : \mathit{prefix}(s)$, *and simultaneously* $p\ \mathit{empty}$ *and* $p\ \mathit{maximal}$, *then* $s\ \mathit{null}$.

Proof. By induction on $p : \mathtt{prefix}(s)$ □

A.2 Derivatives

We define a 3-place relation $\delta_p(s) \sim s'$ between a prefix and two types.

Definition A.2.1 (Derivatives)**.**

$$\frac{}{\delta_{\texttt{epsEmp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\texttt{oneEmp}}\,(1) \sim 1} \qquad \frac{}{\delta_{\texttt{oneFull}}\,(1) \sim \varepsilon} \qquad \frac{\delta_{p_1}\,(s) \sim s' \qquad \delta_{p_2}\,(t) \sim t'}{\delta_{\texttt{parPair}(p_1,p_2)}\,(s \| t) \sim s' \| t'}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s' \cdot t} \qquad \frac{\delta_{p_2}\,(t) \sim t'}{\delta_{\texttt{catBoth}(p_1,p_2)}\,(s \cdot t) \sim t'} \qquad \frac{}{\delta_{\texttt{sumEmp}}\,(s + t) \sim s + t}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{sumInl}(p)}\,(s + t) \sim s'} \qquad \frac{\delta_p\,(t) \sim t'}{\delta_{\texttt{sumInl}(p)}\,(s + t) \sim t'} \qquad \frac{}{\delta_{\texttt{starEmp}}\,(s^\star) \sim s^\star} \qquad \frac{}{\delta_{\texttt{starDone}}\,(s^\star) \sim \varepsilon}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{starFirst}(p)}\,(s^\star) \sim s' \cdot s^\star} \qquad \frac{\delta_{p'}\,(s^\star) \sim s'}{\delta_{\texttt{starRest}(p,p')}\,(s^\star) \sim s'}$$

Definition A.2.2 (Context Derivatives)**.**

$$\frac{}{\delta_\eta\,(\cdot) \sim \cdot} \qquad \frac{\eta(x) \mapsto p \qquad \delta_p\,(s) \sim s'}{\delta_\eta\,(x : s) \sim x : s'} \qquad \frac{\delta_\eta\,(\Gamma) \sim \Gamma' \qquad \delta_\eta\,(\Delta) \sim \Delta'}{\delta_\eta\,(\Gamma\,, \Delta) \sim \Gamma'\,, \Delta'}$$

$$\frac{\delta_\eta\,(\Gamma) \sim \Gamma' \qquad \delta_\eta\,(\Delta) \sim \Delta'}{\delta_\eta\,(\Gamma\,; \Delta) \sim \Gamma'\,; \Delta'}$$

Derivatives are functions defined when the prefix input is well-typed.

Theorem A.2.1 (Derivative Function)**.** *For any p and s, there is at most one s' such that $\delta_p\,(s) \sim s'$. If $p : \texttt{prefix}\,(s)$, then such an s' exists.*

Proof. Induction on the derivation of $\delta_p\,(s) \sim s'$ for uniqueness, and $p : \texttt{prefix}\,(s)$ for existence. □

When it's guaranteed to exist, we write this s' simply as $\delta_p\,(s)$. The empty prefix is the identity for the derivative operator.

Theorem A.2.2 (Empty Prefix Derivative). $\delta_{emp_s}(s) = s$.

Theorem A.2.3 (Empty Context Derivative). *If* $\eta\ \mathtt{emptyOn}\ \Gamma$ *and* $\eta : \mathtt{env}(\Gamma)$ *then* $\delta_\eta(\Gamma) = \Gamma$.

Theorem A.2.4 (Context Derivatives Function). *There is at most one* Γ' *such that* $\delta_\eta(\Gamma) \sim \Gamma'$, *and the* Γ' *exists when* $\eta : \mathtt{env}(\Gamma)$.

Proof. Uniqueness by induction on the derivation of $\delta_\eta(\Gamma) \sim \Gamma'$, existence by induction on the derivation of $\eta : \mathsf{env}(\Gamma)$. □

Theorem A.2.5 (Maximal Derivative iff Nullable). *If* $\delta_p(s) \sim s'$ *then* $p\ \mathtt{maximal}$ *if and only if* $s'\ \mathtt{null}$

Theorem A.2.6 (Only Prefix of a Null Type is Empty). *If* $p : \mathtt{prefix}(s)$ *and* $s\ \mathtt{null}$, *then* $p = emp_s$

A.3 Environments

Definition A.3.1 (Environments and Typing). *An environment is a partial map* $\eta : \mathtt{Var} \to \mathtt{Prefix}$. *We write* $\eta : \mathtt{env}(\Gamma)$ *to mean that* η *is a well-typed environment for* Γ.

$$\frac{}{\eta : \mathtt{env}(\cdot)} \qquad \frac{\eta(x) \mapsto p \qquad p : \mathtt{prefix}(s)}{\eta : \mathtt{env}(x : s)} \qquad \frac{\eta : \mathtt{env}(\Gamma) \qquad \eta : \mathtt{env}(\Delta)}{\eta : \mathtt{env}(\Gamma, \Delta)}$$

$$\frac{\eta : \mathtt{env}(\Gamma) \qquad \eta : \mathtt{env}(\Delta) \qquad \eta\ \mathtt{emptyOn}\ \Delta \vee \eta\ \mathtt{maximalOn}\ \Gamma}{\eta : \mathtt{env}(\Gamma ; \Delta)}$$

Definition A.3.2 (All Maximal, All Empty, Agreement). *For a set* S, *we say* $\eta\ \mathtt{emptyOn}\ S$ *if for all* $x \in S$, *there is some* p *such that* $\eta(x) \mapsto p$, *and* $p\ \mathtt{empty}$. *We say* $\eta\ \mathtt{maximalOn}\ S$ *if for all* $x \in S$, *there is some* p *such that* $\eta(x) \mapsto p$, *and* $p\ \mathtt{maximal}$. *We write* $\eta\ \mathtt{emptyOn}\ \Gamma$ *and* $\eta\ \mathtt{maximalOn}\ \Gamma$ *to mean* $\eta\ \mathtt{emptyOn}\ Dom(\Gamma)$ *and* $\eta\ \mathtt{maximalOn}\ Dom(\Gamma)$, *respectively. We also write* $\eta\ \mathtt{emptyOn}\ e$ *and* $\eta\ \mathtt{maximalOn}\ e$ *to mean* $\eta\ \mathtt{emptyOn}\ fv(e)$ *and* $\eta\ \mathtt{maximalOn}\ fv(e)$, *respectively.*

Finally, we say that η *and* η' *agree on* Δ *and* Δ', *written* $\mathtt{agree}(\eta, \eta', \Delta, \Delta')$ *if* $\eta\ \mathtt{maximalOn}\ \Delta \implies \eta'\ \mathtt{maximalOn}\ \Delta'$, *and* $\eta\ \mathtt{emptyOn}\ \Delta \implies \eta'\ \mathtt{maximalOn}\ \Delta'$

An environment is also an environment for every subcontext.

Theorem A.3.1 (Environment Subcontext Lookup)**.** *If* η : $\mathsf{env}\,(\Gamma(\Delta))$*, then* η : $\mathsf{env}\,(\Delta)$

Proof. Induction on $\Gamma(-)$. □

Moreover, replacing a the environment $\eta|_\Delta$ for a subcontext Δ with another environment η' for another context Δ' yields a well-typed context, so long as η and η' agree on Δ and Δ'. If η was maximal (on Δ) then η' must also be (on Δ'), and if η was empty (on Δ), then η' must also be empty (on Δ').

Theorem A.3.2 (Environment Subcontext Bind)**.** *If* η : $\mathsf{env}\,(\Gamma(\Delta))$ *and* η' : $\mathsf{env}\,(\Delta')$ *such that* $\mathtt{agree}\,(\eta, \eta', \Delta, \Delta')$ *then* $\eta \cdot \eta'$: $\mathsf{env}\,(\Gamma(\Delta'))$

Proof. Induction on the structure of $\Gamma(-)$, inverting everything in sight. □

Theorem A.3.3 (Environment Par Bind)**.** *If* η : $\mathsf{env}\,(\Gamma(z : s \| t))$ *and* $\eta(z) \mapsto \mathtt{parPair}(p_1, p_2)$ *then* $\eta[x \mapsto p_1, y \mapsto p_2]$: $\mathsf{env}\,(\Gamma(x : s\,, y : t))$

Proof. By Theorem A.3.2. □

Theorem A.3.4 (Environment Cat Bind 1)**.** *If* η : $\mathsf{env}\,(\Gamma(z : s \cdot t))$ *and* $\eta(z) \mapsto \mathtt{catFst}(p)$ *then* $\eta[x \mapsto p, y \mapsto \mathsf{emp}_t]$: $\mathsf{env}\,(\Gamma(x : s\,; y : t))$

Proof. By Theorem A.3.2. □

Theorem A.3.5 (Environment Cat Bind 2)**.** *If* η : $\mathsf{env}\,(\Gamma(z : s \cdot t))$ *and* $\eta(z) \mapsto \mathtt{catBoth}(p_1, p_2)$ *then* $\eta[x \mapsto p_1, y \mapsto p_2]$: $\mathsf{env}\,(\Gamma(x : s\,; y : t))$

Proof. By Theorem A.3.2. □

Lastly, the structure of the above subcontext replacement operation is compatible with derivatives. Taking the derivative of $\Gamma(\Delta)$ by η yields $\Gamma'(\delta_\eta\,(\Delta))$ for some $\Gamma'(-)$, and for *any* other filler Δ_0 and environment η_0 : $\mathsf{env}\,(\Delta_0)$, the outer derivative bit of the derivative remains unchanged: $\delta_{\eta \cup \eta_0}\,(\Gamma(\Delta_0))$ is $\Gamma'(\delta_{\eta_0}\,(\Delta_0))$

Theorem A.3.6 (Environment Subcontext Bind Derivative)**.** *If* $\delta_\eta\,(\Gamma(\Delta)) \sim \Gamma_0$ *then there is some* $\Gamma'(-)$ *such that for all* Δ' *and* Δ'' *and* η'*, if* $\delta_{\eta'}\,(\Delta') \sim \Delta''$ *and* $\mathtt{agree}\,(\eta, \eta', \Delta, \Delta')$ *then* $\delta_{\eta \cup \eta'}\,(\Gamma(\Delta')) \sim \Gamma'(\Delta'')$

Proof. Induction on $\Gamma(-)$. □

Theorem A.3.7 (Environment Par Derivative)**.** *If* $\delta_\eta\ (\Gamma(z : s \| t)) \sim \Gamma'(z : s' \| t')$ *and* $\eta(z) \mapsto \texttt{parPair}(p_1, p_2)$ *then* $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\ (\Gamma(x : s, y : t)) \sim \Gamma'(x : s', y : t')$

Proof. By Theorem A.3.6. □

Theorem A.3.8 (Environment Cat Derivative 1)**.** *If* $\delta_\eta\ (\Gamma(z : s \cdot t)) \sim \Gamma'(z : s' \cdot t)$ *and* $\eta(z) \mapsto \texttt{catFst}(p)$ *then* $\delta_{\eta[x \mapsto p, y \mapsto \texttt{emp}_t]}\ (\Gamma(x : s ; y : t)) \sim \Gamma'(x : s' ; y : t)$

Proof. By Theorem A.3.6. □

Theorem A.3.9 (Environment Cat Derivative 2)**.** *If* $\delta_\eta\ (\Gamma(z : s \cdot t)) \sim \Gamma'(z : t')$ *and* $\eta(z) \mapsto \texttt{catBoth}(p_1, p_2)$ *then* $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\ (\Gamma(x : s ; y : t)) \sim \Gamma'(x : s' ; y : t')$

Proof. By Theorem A.3.6. □

Theorem A.3.10 (Environment Lookup)**.** *For any* η*, there is at most one* p *so that* $\eta(x) \mapsto p$*. When* $\eta : \texttt{env}\,(\Gamma(x : s))$*, this* p *exists, and* $p : \texttt{prefix}\,(s)$*.*

Proof. The "at most one" p is immediate from the fact that η is a deterministic partial function. If $\eta : \texttt{env}\,(\Gamma(x : s))$ then $\eta : \texttt{env}\,(x : s)$ by Theorem A.3.1. By inversion, there is some $p : \texttt{prefix}\,(s)$ such that $\eta(x) \mapsto p$. □

Theorem A.3.11 (Environment Lookup Derivative)**.** *Suppose:*

1. $\eta(x) = p$
2. $\eta : \texttt{env}\,(\Gamma(x : s))$
3. $\delta_p\ (s) \sim s'$
4. $\delta_\eta\ (\Gamma(x : s)) \sim \Gamma_0$

Then there is some $\Gamma'(-)$ *such that* $\Gamma_0 = \Gamma'(x : s')$*.*

Proof. Immediate by Theorem A.3.6 □

A.4 Concatenation

More generally, we often want to concatenate a prefix p of s with a prefix p' of $\delta_p(s)$. This is defined with another 3-place, type-indexed relation.

Definition A.4.1 (Prefix Concatenation)**.** *We define a relation* $p \cdot p' \sim p''$.

$$\frac{}{\mathit{epsEmp} \cdot \mathit{epsEmp} \sim \mathit{epsEmp}}$$

$$\frac{p : \mathit{prefix}(1)}{\mathit{oneEmp} \cdot p \sim p} \qquad \frac{}{\mathit{oneFull} \cdot \mathit{epsEmp} \sim \mathit{oneFull}}$$

$$\frac{p_1 \cdot p_1' \sim p_1'' \qquad p_2 \cdot p_2' \sim p_2''}{\mathit{parPair}(p_1, p_2) \cdot \mathit{parPair}(p_1', p_2') \sim \mathit{parPair}(p_1'', p_2'')}$$

$$\frac{p \cdot p' \sim p''}{\mathit{catFst}(p) \cdot \mathit{catFst}(p') \sim \mathit{catFst}(p'')} \qquad \frac{p \cdot p' \sim p''}{\mathit{catFst}(p) \cdot \mathit{catBoth}(p', p''') \sim \mathit{catBoth}(p'', p''')}$$

$$\frac{p' \cdot p'' \sim p'''}{\mathit{catBoth}(p, p') \cdot p'' \sim \mathit{catBoth}(p, p''')}$$

$$\frac{}{\mathit{sumEmp} \cdot p \sim p} \qquad \frac{p' \cdot p'' \sim p''}{\mathit{sumInl}(p) \cdot p' \sim \mathit{sumInl}(p'')} \qquad \frac{p \cdot p' \sim p''}{\mathit{sumInr}(p) \cdot p' \sim \mathit{sumInr}(p'')}$$

$$\frac{}{\mathit{starEmp} \cdot p \sim p} \qquad \frac{}{\mathit{starDone} \cdot \mathit{epsEmp} \sim \mathit{starDone}}$$

$$\frac{p \cdot p' \sim p''}{\mathit{starFirst}(p) \cdot \mathit{catFst}(p') \sim \mathit{starFirst}(p'')}$$

$$\frac{p \cdot p' \sim p''}{\mathit{starFirst}(p) \cdot \mathit{catBoth}(p', p''') \sim \mathit{starRest}(p'', p''')}$$

$$\frac{p' \cdot p'' \sim p'''}{\mathit{starRest}(p, p') \cdot p'' \sim \mathit{starRest}(p, p''')}$$

This relation is a function when the inputs are well-typed. Because of this, when $p : \texttt{prefix}(s)$ and $p' : \texttt{prefix}\left(\delta_p(s)\right)$, we write $p \cdot p'$ for the unique p'' that the following theorem guarantees.

Theorem A.4.1 (Prefix Concatenation Function)**.** *For all p, p' and s, there is at most one p'' such that $p \cdot p' \sim p''$. If $p : \texttt{prefix}(s)$ and $p' : \texttt{prefix}\left(\delta_p(s)\right)$, then such a p'' exists, and satisfies:*

1. $p'' : \texttt{prefix}(s)$
2. $\delta_{p''}(s) = \delta_{p'}\left(\delta_p(s)\right)$

Proof. Existence, (1), and (2) follow by induction on the derivation of $p : \texttt{prefix}(s)$. Uniqueness is immediate by the fact that the relation is a function. □

Theorem A.4.2 (Prefix Concatenation Empty)**.** *If $p : \texttt{prefix}(s)$, then $\texttt{emp}_s \cdot p \sim p$ and $p \cdot \texttt{emp}_{\delta_p(s)} \sim p$*

Proof. Induction on the derivation of $p : \texttt{prefix}(s)$. □

Theorem A.4.3 (Maximal Prefix Concatenation)**.** *Suppose $p \cdot p' \sim p''$. If p'' is maximal, then p' is maximal. If p or p' is maximal, then p'' is maximal. Moreover, if p is maximal, then $p'' = p$.*

Proof. Induction on the derivation of $p \cdot p' \sim p''$. □

Theorem A.4.4 (Prefix Concatenation Associativity)**.** $p \cdot (p' \cdot p'') = (p \cdot p') \cdot p''$*, when defined.*

Proof. Induction on derivations of concatenation. □

Definition A.4.2 (Environment Concatenation)**.** *We write $\eta \cdot \eta' \sim \eta''$ to mean that η'' is the function defined on the largest subset S of $Dom(\eta) \cap Dom(\eta')$ such that for all $x \in S$, the prefix concatenation $\eta(x) \cdot \eta'(x) \sim p$ exists, and $\eta''(x) = p$, for all $x \in S$.*

Theorem A.4.5 (Environment Concatenation Function)**.** *For any η and η', there is at most one η'' such that $\eta \cdot \eta' \sim \eta''$, and such an η'' exists when $\eta : \texttt{env}(\Gamma)$ and $\delta_\eta(\Gamma) \sim \Gamma'$ and $\eta' : \texttt{prefix}(\Gamma')$.*

Proof. Uniqueness by the "greatest" property, existence by Theorem A.4.1. □

Theorem A.4.6 (Environment Concatenation Correctness)**.** *If $\eta : \texttt{env}(\Gamma)$ and $\delta_\eta(\Gamma) \sim \Gamma'$ and $\eta' : \texttt{env}(\Gamma')$, and $\eta \cdot \eta' \sim \eta''$, then $\eta'' : \texttt{env}(\Gamma)$, and if $\delta_{\eta'}(\Gamma') \sim \Gamma''$ then $\delta_{\eta''}(\Gamma) \sim \Gamma''$.*

Theorem A.4.7 (Environment Concatenation Empty)**.** *If* η : $\mathit{env}\,(\Gamma)$ *and* $\eta \cdot \eta' \sim \eta''$*, then:*

- *If* η $\mathtt{emptyOn}\,S$ *and then* $\eta''|_S = \eta'|_S$
- *If* η' $\mathtt{emptyOn}\,S$*, then* $\eta''|_S = \eta|_S$

Proof. Induction on the derivation of η : $\mathsf{env}\,(\Gamma)$, using Theorem A.4.2. □

Theorem A.4.8 (Maximal Environment Concatenation)**.** *If* $\eta \cdot \eta' \sim \eta''$*, then* η $\mathtt{maximalOn}\,S$ *or* η' $\mathtt{maximalOn}\,S$ *if and only if* η'' $\mathtt{maximalOn}\,S$*.*

Proof. Immediate corollary of Theorem A.4.3 □

Theorem A.4.9 (Prefix Concatenation Associativity)**.** $\eta \cdot (\eta' \cdot \eta'') = (\eta \cdot \eta') \cdot \eta''$*, when defined.*

Proof. Corollary of Theorem A.4.4 □

A.5 Historical Contexts

Definition A.5.1 (Historical Context)**.** *Contexts* $\Omega := \cdot \mid \Omega, x : A$ *are fully structural contexts, where the* A *are STLC types.*

A stream type is "flattened" into an STLC type by turning concatenations and parallels into products, and stars into lists.

Definition A.5.2 (Type and Context Flatten)**.** *For* s *a stream type, we define its flattening into an STLC type, denoted* $\langle s \rangle$*, inductively:*

- $\langle 1 \rangle = 1$
- $\langle \varepsilon \rangle = 1$
- $\langle s \cdot t \rangle = \langle s \rangle \times \langle t \rangle$
- $\langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$
- $\langle s + t \rangle = \langle s \rangle + \langle t \rangle$

- $\langle s^\star \rangle = \texttt{list}\,(\langle s \rangle)$

For Γ a bunched context, we define its flattening to a standard context, $\langle \Gamma \rangle$ inductively:

- $\langle \cdot \rangle = \cdot$
- $\langle x : s \rangle = x : \langle s \rangle$
- $\langle \Gamma ; \Gamma' \rangle = \langle \Gamma \rangle, \langle \Gamma' \rangle$
- $\langle \Gamma, \Gamma' \rangle = \langle \Gamma \rangle, \langle \Gamma' \rangle$

For an STLC value $v : \langle s \rangle$, we write $\texttt{toPrefix}_s\,(v)$ for the maximal prefix of type s that it corresponds to. Dually, for a maximal prefix $p : \texttt{prefix}\,(s)$, we write $\langle p \rangle$ for the STLC value of type $\langle s \rangle$ it corresponds to.

Definition A.5.3 (Historical Programs and Substitutions)**.** *Fix a language of terms M, with type system $\Omega \vdash M : A$. Write its semantics as $M \downarrow v$. We assume that this relation is a decidable partial function, in the sense that M evaluates to at most one v, and it is decidable whether or not such a v exists. We write substitutions $\theta : \Omega' \longrightarrow \Omega$. Substitutions have a contravariant action on terms, written $M[\theta]$: if $\Omega \vdash M : A$, then $\Omega' \vdash M[\theta] : A$. We lift this substitution action to λ^{ST} terms compositionally, substituting into all historical terms. We write a list of such terms as $\overline{M}$, and lift the typing relation and semantics to lists of terms, written $\Omega \vdash \overline{M} : \overline{A}$ and $\overline{M} \downarrow \theta$.*

A.6 Context Subtyping

The following is a full listing of subtyping rules.

Definition A.6.1 (Subtyping)**.**

$$\textsc{Sub-Cong}\ \frac{\Delta\ <:\ \Delta'}{\Gamma(\Delta)\ <:\ \Gamma(\Delta')} \qquad \textsc{Sub-Refl}\ \frac{}{\Gamma\ <:\ \Gamma} \qquad \textsc{Sub-Comma-Exc}\ \frac{}{\Gamma,\Delta\ <:\ \Delta,\Gamma}$$

$$\textsc{Sub-Sng-Wkn}\ \frac{}{x : s\ <:\ \cdot} \qquad \textsc{Sub-Comma-Wkn}\ \frac{}{\Gamma,\Delta\ <:\ \Gamma} \qquad \textsc{Sub-Semic-Wkn-1}\ \frac{}{\Gamma;\Delta\ <:\ \Gamma} \qquad \textsc{Sub-Semic-Wkn-2}\ \frac{}{\Gamma;\Delta\ <:\ \Delta}$$

$$\textsc{Sub-Comma-Unit}\ \frac{}{\Gamma\ <:\ \Gamma,\cdot} \qquad \textsc{Sub-Semic-Unit-1}\ \frac{}{\Gamma\ <:\ \Gamma;\cdot} \qquad \textsc{Sub-Semic-Unit-2}\ \frac{}{\Gamma\ <:\ \cdot;\Gamma}$$

Environment typing is preserved by subtyping, and derivatives preserve subtyping relations between contexts.

Theorem A.6.1 (Subtyping Preserves Environments)**.** *If* $\eta\ :\ \mathit{env}\,(\Gamma)$ *and* $\Gamma\ <:\ \Delta$ *then* $\eta\ :\ \mathit{env}\,(\Delta)$

Proof. By induction on $\Gamma\ <:\ \Delta$, and inversion on $\eta\ :\ \mathsf{env}\,(\Gamma)$. □

Theorem A.6.2 (Derivatives Preserve Subtyping)**.** *Suppose* $\Gamma\ <:\ \Delta$ *and* $\delta_\eta\,(\Gamma) \sim \Gamma'$ *and* $\delta_\eta\,(\Delta) \sim \Delta'$*. Then,* $\Gamma'\ <:\ \Delta'$*.*

Proof. By cases on $\Gamma\ <:\ \Delta$, inverting the derivations of $\delta_\eta\,(\Gamma)\ \sim\ \Gamma'$ and $\delta_\eta\,(\Delta)\ \sim\ \Delta'$, and using the determinism of the derivative relation. □

A.7 Type System

A.7.0.1 Inertness

Most terms, like variables or case expressions, require some non-empty amount of input to arrive for them to produce a non-empty output. However, this is not true of all terms: constants like `()` and `nil`, (some) sequential terms like $(();e)$ and $\mathtt{eps} :: e$, and sum terms $\mathtt{inl}(e)$ produce nonempty output even when

given an entirely empty input prefix. Terms like these contain "information" that they are always ready to produce, even if there is no input to drive them forward. We call terms that are always ready to produce output *jumpy*, and terms that are not *inert*.

As described in Section 4.1.3, the type system requires that let-bound terms are always inert to guarantee soundness of the semantics. In particular, inertness is what guarantees that the "agreement" (Definition A.3.2) requirement in Theorem A.3.2 to hold in the soundness case for T-Let. For arbitrary terms, the maximality component of agreement always holds, but the emptiness component of agreement requires inertness.

To enforce that the bodies of let-bindings are inert, we track a syntactic over-approximation of inertness with the type system, essentially as an *effect*. This is accomplished by giving every typing judgment an *inertness annotation*, $i ::= \texttt{I} \mid \texttt{J}$, and we ensure that if e is typed with annotation `Inert`, then e produces empty output when given an empty input. This invariant is proved as an additional consequence to the soundness theorem.

We note that the choice to include inertness in the type system itself, as opposed to a predicate on (typed) terms, is essentially an arbitrary one: we choose the former to minimize the number of assumptions running around in our proofs.

For the most part, the inertness analysis is straightforward. Constants like `()` and `nil`, and injections like `inl` (e), `inr` (e), and e_1 `::` e_2 (secretly the right injection into $\varepsilon + s \cdot s^\star$) all have annotation `J`. Non-buffering elimination forms have the same inertness as their bodies, and variables and `eps` are inert. The most important ones are in the rules T-Cat-R and T-Plus-L (and the similar ones in T-Star-L and T-Plus-L).

The inertness requirement for T-Cat-R says that if the resulting term $(e_1; e_2)$ is to be typed as inert, e_1 must be inert, and the type of e_1 must not be `null`. Otherwise, $(e_1; e_2)$ could produce a maximal .

The rule for T-Plus-L says that it is inert when the buffer environment does not yet include a decision for which way to go ($\eta(z) = \texttt{sumEmp}$). Note that in practice, this is always satisfied. At the beginning of execution, η maps all variables to empty prefixes, and as soon as $\eta(z)$ gets either $\texttt{sumInl}(p)$ or $\texttt{sumInr}(p)$, we step to the corresponding branch. In fact, the result of *every* step is inert: otherwise we would've output a larger prefix in that step!

Definition A.7.1 (Typing Rules)**.** *Figure A.1 and Figure A.2 present the full typing rules.*

Definition A.7.2 (Recursion Signature)**.** *A recursion signature* Σ *is either empty (signaling that typechecking is not in the body of a recursive function), or the signature* $\Omega \mid \Gamma \longrightarrow s \,@\, i$ *of a sequent which defines the recursive function we are currently checking the body of.* $\Sigma ::= \emptyset \mid (\Omega \mid \Gamma \longrightarrow s \,@\, i)$

These typing rules are mutually defined with another typing judgment $\Omega \mid \Gamma \vdash_\Sigma A : \Gamma' \,@\, i$, meaning that A is a well-typed set of arguments (hence A) for a recursive call to a function accepting inputs Γ'. Here, A is an *tree* of terms, with either comma or semicolon nodes. This judgment ensures that $e_{\Gamma'}$ has well-typed bindings for every variable $x : s$ in Γ', and that the variables that $e_{\Gamma'}$ uses are used in accordance with Γ, its context.

Definition A.7.3 (Recursive Argument Typing)**.**

$$A ::= \cdot \mid e \mid (A, A') \mid (A; A') \mid (\cdot; A)$$

$$\frac{}{\Omega \mid \Gamma \vdash_\Sigma \cdot : \cdot \,@\, i}\ \textit{T-Args-Emp} \qquad \frac{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma e : (x : s) \,@\, i}\ \textit{T-Args-Sng}$$

$$\frac{\Omega \mid \Gamma \vdash_\Sigma A : \Delta \,@\, i_1 \qquad \Omega \mid \Gamma' \vdash_\Sigma A' : \Delta' \,@\, i_2}{\Omega \mid \Gamma\,;\Gamma' \vdash_\Sigma (A; A') : \Delta\,;\Delta' \,@\, i_3}\ \textit{T-Args-Semic-1}$$

$$\frac{\Omega \mid \Gamma' \vdash_\Sigma A : \Delta' \,@\, i \qquad \Delta\ \mathit{null}}{\Omega \mid \Gamma\,;\Gamma' \vdash_\Sigma (\cdot; A) : \Delta\,;\Delta' \,@\, i}\ \textit{T-Args-Semic-2}$$

$$\frac{\Omega \mid \Gamma \vdash_\Sigma A : \Delta \,@\, i \qquad \Omega \mid \Gamma \vdash_\Sigma A' : \Delta' \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma (A, A') : \Delta\,,\Delta' \,@\, i}\ \textit{T-Args-Comma}$$

Buffering Rules

The left rules for star and sums, as well as Wait, include a *buffer* in the term: a prefix of the input context, where we store inputs until we have received enough to run the term. For example, the Wait rule has this buffer η, which we gather until it includes a maximal prefix of $x : s$.

T-Wait

$$\frac{\eta \ : \ \texttt{env}\ (\Gamma(x : s)) \qquad \delta_\eta\ (\Gamma(x : s)) \sim \Gamma' \qquad \Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s \mathbin{@} ii' = \texttt{I} \implies \neg\ (\eta(z)\ \texttt{maximal}) \wedge \neg\ (s\ \texttt{null})}{\Omega \mid \Gamma' \vdash_\Sigma \texttt{wait}_{\eta,t}(x)\ (e) : t \mathbin{@} i'}$$

The buffer is included in the syntax of the term. Additionally, the context in the conclusion is $\delta_p\ (\Gamma(\Delta))$. If we've buffered η of the input, the term is expecting the rest of the context. Users of the calculus need not worry about this detail: when writing programs and when the program starts running, the buffer is empty: $\eta = \texttt{emp}_{\Gamma(\Delta)}$, and since $\delta_\eta\ (\Gamma(\Delta)) = \Gamma(\Delta)$, this returns Wait to the expected rule presented in the body of the paper. The other rules that include buffers are Plus-L and Star-L.

A.8 Sink Terms

Once we have produced an entire maximal prefix $p \ : \ \texttt{prefix}\ (s)$, a program e of type s needs to transition to a program emitting nothing: we compute this term from p with $\texttt{sink}_p$.

Definition A.8.1 (Sink Terms)**.** *We define a term $\texttt{sink}_p$ by induction on p.*

- $\mathit{sink}_{\mathit{epsEmp}} = \mathit{eps}$
- $\mathit{sink}_{\mathit{oneEmp}} = \mathit{eps}$
- $\mathit{sink}_{\mathit{oneFull}} = \mathit{eps}$
- $\mathit{sink}_{\mathit{parPair}(p_1,p_2)} = \left(\mathit{sink}_{p_1}, \mathit{sink}_{p_2}\right)$
- $\mathit{sink}_{\mathit{catFst}(p)} = \mathit{sink}_p$
- $\mathit{sink}_{\mathit{catBoth}(p_1,p_2)} = \mathit{sink}_{p_2}$
- $\mathit{sink}_{\mathit{sumEmp}} = \mathit{eps}$
- $\mathit{sink}_{\mathit{sumInl}(p)} = \mathit{sink}_p$

- $\texttt{sink}_{\texttt{sumInr}(p)} = \texttt{sink}_p$
- $\texttt{sink}_{\texttt{starEmp}} = \texttt{eps}$
- $\texttt{sink}_{\texttt{starDone}} = \texttt{eps}$
- $\texttt{sink}_{\texttt{starFirst}(p)} = \texttt{sink}_p$
- $\texttt{sink}_{\texttt{starRest}(p,p')} = \texttt{sink}_{p'}$

Note that (because it's easier to have this be a function rather than a relation) sink terms are defined for *all* prefixes rather than just the maximal ones.

Sink terms are closed, and have the type we expect for a stream transformer that has just emitted an maximal p of type s.

Theorem A.8.1 (Sink Terms Typing)**.** *If p $\texttt{maximal}$ and $p : \texttt{prefix}(s)$ and $\delta_p(s) \sim s'$, then $\cdot \mid \Gamma \vdash_\emptyset \texttt{sink}_p : s' \mathbin{@} \texttt{I}$*

The relevant concatenation property of sink terms is that they only depend on the the shape of the type s *after* the prefix has been emitted, so adding more to the beginning does not change anything.

Theorem A.8.2 (Sink Term Concatenation)**.** *If $p \cdot p' \sim p''$, then $\texttt{sink}_{p'} = \texttt{sink}_{p''}$.*

Proof. By induction on $p \cdot p' \sim p''$. □

Theorem A.8.3 (Fixpoint Substitution)**.** *For e, e' terms, we define $e\,[e'/\texttt{rec}]$ compositionally over the structure of e, with the only two interesting cases being:*

$$\left(\texttt{rec}\left\{\overline{M}\right\}(A)\right)[e'/\texttt{rec}] = \texttt{fix}\left\{\overline{M}\right\}(A\,[e'/\texttt{rec}]).(e')$$

and

$$\left(\texttt{fix}\left\{\overline{M}\right\}(A).(e)\right)[e'/\texttt{rec}] = \texttt{fix}\left\{\overline{M}\right\}(A\,[e'/\texttt{rec}]).(e)$$

We define this mutually with a substitution for arguments A, with $A\,[e'/\texttt{rec}]$ defined compositionally over the structure of A.

Then if $\Omega \mid \Gamma \vdash_{\Omega|\Gamma \to s \mathbin{@} i'} e' : s \mathbin{@} i'$, we have:

1. *If* $\Omega' \mid \Delta \vdash_{\Omega|\Gamma \to s\,@\,i'} e : t\,@\,i$ *then* $\Omega' \mid \Delta \vdash_{\cdot} e\,[e'/\mathtt{rec}] : t\,@\,i$

2. *If* $\Omega' \mid \Delta \vdash_{\Omega|\Gamma \to s\,@\,i'} A : \Gamma'\,@\,i$, *then* $\Omega' \mid \Delta \vdash_{\cdot} A\,[e'/\mathtt{rec}] : \Gamma'\,@\,i$

Proof. (1) and (2) are proved by a routine simultaneous induction on typing derivations. □

A.9 Semantics

Definition A.9.1 (Semantics)**.** *We define relations* $p \Rightarrow e \downarrow^n e' \Rightarrow p'$ *(as shown in Figures A.3 and A.4), and* $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ *(as shown in Figure A.5).*

Recursive Argument Semantics

The arguments semantics $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ accepts an environment η and runs it through A to produce an environment $\eta' : \mathtt{env}(\Gamma)$. This relation is essentially the same as evaluating a large nested tree T-Cat-R T-Par-R terms, structured like the context Γ. The only difference is that, because context derivatives do not remove the left component of a semicolon context (the Γ in $\Gamma\,;\Delta$) after a maximal prefix has arrived, we have a special term former $(\cdot; A')$ for cat-pair terms $(A; A')$ that have crossed over. The context is required in the semantics so we can compute the empty environment in S-Args-Semic-1-1 and S-Args-Semic-2.

Semantics of Buffering

The semantics for Plus-L andStar-L and Wait buffer in their inputs until enough of the input has arrived to run the term, where the particular value of "enough" depends on the rule in question.

To illustrate, consider the rules for Wait (S-Wait-1 and S-Wait-2 in Figure A.4). In both cases, we take the incoming environment η, and concatenate it onto the buffer η', to get the combined prefix η''. We then dispatch on whether η'' is enough input to run the continuation e. In this case, "enough" means that η'' contains a maximal prefix p of $x : s$. If it does (P-Wait-2), we run the continuation, substituting the maximal prefix in for the (historical) occurrences of x. If it does not, we simply save η'' as the new buffer in the resulting `wait` term, and return the empty prefix in P-Wait-1.

The semantics for Plus-L and Star-L are similar: in all cases, we add the incoming prefix to the buffer, and then project from the buffer. If not enough data has arrived, we return the empty prefix and step to the same term but with an updated buffer.

T-Eps-R

$$\frac{}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{eps} : \varepsilon \,@\, i}$$

T-One-R

$$\frac{}{\Omega \mid \Gamma \vdash_\Sigma () : 1 \,@\, \mathtt{J}}$$

T-Var

$$\frac{}{\Omega \mid \Gamma(x : s) \vdash_\Sigma x : s \,@\, i}$$

T-Sub

$$\frac{\Omega \mid \Delta \vdash_\Sigma e : s \,@\, i \qquad \Gamma <: \Delta}{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}$$

T-Par-R

$$\frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \,@\, i \qquad \Omega \mid \Gamma \vdash_\Sigma e_2 : t \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma (e_1, e_2) : s \| t \,@\, i}$$

T-Cat-R

$$\frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \,@\, i_1 \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : t \,@\, i_2 \qquad i_3 = \mathtt{I} \implies i_1 = \mathtt{I} \wedge \neg\, (s\ \mathtt{null})}{\Omega \mid \Gamma ; \Delta \vdash_\Sigma (e_1; e_2) : s \cdot t \,@\, i_3}$$

T-Par-L

$$\frac{\Omega \mid \Gamma(x : s, y : t) \vdash_\Sigma e : r \,@\, i}{\Omega \mid \Gamma(z : s \| t) \vdash_\Sigma \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e : r \,@\, i}$$

T-Cat-L

$$\frac{\Omega \mid \Gamma(x : s ; y : t) \vdash_\Sigma e : r \,@\, i}{\Omega \mid \Gamma(z : s \cdot t) \vdash_\Sigma \mathtt{let}_t\ (x; y) = z\ \mathtt{in}\ e : r \,@\, i}$$

T-Plus-R-1

$$\frac{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inl}(e) : s + t \,@\, \mathtt{J}}$$

T-Plus-R-2

$$\frac{\Omega \mid \Gamma \vdash_\Sigma e : t \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inr}(e) : s + t \,@\, i}$$

T-Plus-L

$$\frac{\begin{array}{c}\eta\ :\ \mathsf{env}\,(\Gamma(z : s + t)) \qquad \delta_\eta\,(\Gamma(z : s + t)) \sim \Gamma' \\ \Omega \mid \Gamma(x : s) \vdash_\Sigma e_1 : r \,@\, i_1 \qquad \Omega \mid \Gamma(y : t) \vdash_\Sigma e_2 : r \,@\, i_2 \qquad i = \mathtt{I} \implies \eta(z) = \mathtt{sumEmp}\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \mathtt{case}_r\,(\eta; z, x.e_1, y.e_2) : r \,@\, i}$$

Figure A.1: λ^{ST} Full Typing Rules (Part 1)

T-Star-R-1

$$\frac{}{\Omega \mid \Gamma \vdash_\Sigma \texttt{nil} : s^\star \mathbin{@} \texttt{J}}$$

T-Star-R-2

$$\frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \mathbin{@} i_1 \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : s^\star \mathbin{@} i_2}{\Omega \mid \Gamma ; \Delta \vdash_\Sigma e_1 :: e_2 : s^\star \mathbin{@} \texttt{J}}$$

T-Star-L

$$\frac{\begin{array}{c}\eta \,:\, \texttt{env}\left(\Gamma(z : s^\star)\right) \qquad \delta_\eta\left(\Gamma(z : s^\star)\right) \sim \Gamma' \\ \Omega \mid \Gamma(\cdot) \vdash_\Sigma e_1 : r \mathbin{@} i_1 \qquad \Omega \mid \Gamma(x : s; xs : s^\star) \vdash_\Sigma e_2 : r \mathbin{@} i_2 \qquad i = \texttt{I} \implies \eta(z) = \texttt{starEmp}\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \texttt{case}_{s,r}\left(\eta; z, e_1, x.xs.e_2\right) : r \mathbin{@} i}$$

T-HistPgm

$$\frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash_\Sigma \langle M : s \rangle : s \mathbin{@} \texttt{J}}$$

T-Wait

$$\frac{\begin{array}{c}\eta \,:\, \texttt{env}\left(\Gamma(x : s)\right) \\ \delta_\eta\left(\Gamma(x : s)\right) \sim \Gamma' \qquad \Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : t \mathbin{@} i \qquad i' = \texttt{I} \implies \neg\left(\eta(z)\ \texttt{maximal}\right) \wedge \neg\left(s\ \texttt{null}\right)\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \texttt{wait}_{\eta,t}(x)\ (e) : t \mathbin{@} i'}$$

T-Let

$$\frac{\Omega \mid \Delta \vdash_\Sigma e_1 : s \mathbin{@} \texttt{I} \qquad \Omega \mid \Gamma(x : s) \vdash_\Sigma e_2 : t \mathbin{@} i}{\Omega \mid \Gamma(\Delta) \vdash_\Sigma \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : t \mathbin{@} i}$$

T-Rec

$$\frac{\Omega' \mid \Gamma' \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} A : \Gamma \mathbin{@} i \qquad \Omega' \vdash \overline{M} : \Omega}{\Omega' \mid \Gamma' \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} \texttt{rec}\left\{\overline{M}\right\}(A) : s \mathbin{@} i}$$

T-Fix

$$\frac{\Omega \mid \Gamma \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} e : s \mathbin{@} i \qquad \Omega' \vdash \overline{M} : \Omega \qquad \Omega' \mid \Gamma' \vdash_\Sigma A : \Gamma \mathbin{@} i}{\Omega' \mid \Gamma' \vdash_\Sigma \texttt{fix}\left\{\overline{M}\right\}(A)\,.\,(e) : s \mathbin{@} i}$$

T-ArgsLet

$$\frac{\Omega \mid \Gamma' \vdash_\Sigma A : \Gamma \mathbin{@} i \qquad \Omega \mid \Gamma \vdash_\Sigma e : s \mathbin{@} i}{\Omega \mid \Gamma' \vdash_\Sigma \texttt{let}\, \Gamma = A\, \texttt{in}\, e : s \mathbin{@} i}$$

Figure A.2: λ^{ST} Full Typing Rules (Part 2)

$$\text{S-Eps-R}\ \frac{}{\eta \Rightarrow \texttt{eps} \downarrow^n \texttt{eps} \Rightarrow \texttt{epsEmp}} \qquad \text{S-One-R}\ \frac{}{\eta \Rightarrow () \downarrow^n \texttt{eps} \Rightarrow \texttt{oneFull}} \qquad \text{S-Var}\ \frac{\eta(x) \mapsto p}{\eta \Rightarrow x \downarrow^n x \Rightarrow p}$$

$$\text{S-Par-R}\ \frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1, e_2) \downarrow^{n_1+n_2} (e_1', e_2') \Rightarrow \texttt{parPair}(p_1, p_2)} \qquad \text{S-Cat-R-1}\ \frac{\eta \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p \qquad \neg\, (p\ \texttt{maximal})}{\eta \Rightarrow (e_1; e_2) \downarrow^n (e_1'; e_2) \Rightarrow \texttt{catFst}(p)}$$

$$\text{S-Cat-R-2}\ \frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad p_1\ \texttt{maximal} \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1; e_2) \downarrow^{n_1+n_2} e_2' \Rightarrow \texttt{catBoth}(p_1, p_2)}$$

$$\text{S-Par-L}\ \frac{\eta(z) \mapsto \texttt{parPair}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow^n e' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e \downarrow^n \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e \Rightarrow p}$$

$$\text{S-Cat-L-1}\ \frac{\eta(z) \mapsto \texttt{catFst}(p) \qquad \eta[x \mapsto p, y \mapsto \texttt{emp}_t] \Rightarrow e \downarrow^n e' \Rightarrow p'}{\eta \Rightarrow \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e \downarrow^n \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e' \Rightarrow p'}$$

$$\text{S-Cat-L-2}\ \frac{\eta(z) \mapsto \texttt{catBoth}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow^n e' \Rightarrow p}{\eta \Rightarrow \texttt{let}_t\ (x; y) = z\ \texttt{in}\ e \downarrow^n \texttt{let}\ x = \texttt{sink}_{p_1}\ \texttt{in}\ e'[z/y] \Rightarrow p} \qquad \text{S-Plus-R-1}\ \frac{\eta \Rightarrow e \downarrow^n e' \Rightarrow p}{\eta \Rightarrow \texttt{inl}(e) \downarrow^n e' \Rightarrow \texttt{inl}(p)}$$

$$\text{S-Plus-R-2}\ \frac{\eta \Rightarrow e \downarrow^n e' \Rightarrow p}{\eta \Rightarrow \texttt{inr}(e) \downarrow^n e' \Rightarrow \texttt{inr}(p)}$$

$$\text{S-Plus-L-1}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{sumEmp}}{\eta \Rightarrow \texttt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^n \texttt{case}_r\ (\eta''; z, x.e_1, y.e_2) \Rightarrow \texttt{emp}_r}$$

$$\text{S-Plus-L-2}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{sumInl}(p) \qquad \eta''[x \mapsto p] \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p'}{\eta \Rightarrow \texttt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^n e_1'[z/x] \Rightarrow p'}$$

$$\text{S-Plus-L-3}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{sumInr}(p) \qquad \eta''[y \mapsto p] \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^n e_2'[z/y] \Rightarrow p'}$$

Figure A.3: λ^{ST} Semantics (Part 1)

S-Star-R-1

$$\frac{}{\eta \Rightarrow \texttt{nil} \downarrow^{n} \texttt{eps} \Rightarrow \texttt{starDone}}$$

S-Star-R-2-1

$$\frac{\eta \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p \qquad \neg\,(p\ \texttt{maximal})}{\eta \Rightarrow e_1 :: e_2 \downarrow^{n} (e_1'; e_2) \Rightarrow \texttt{starFirst}(p)}$$

S-Star-R-2-2

$$\frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad p_1\ \texttt{maximal} \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow e_1 :: e_2 \downarrow^{n_1+n_2} e_2' \Rightarrow \texttt{starRest}(p_1, p_2)}$$

S-Star-L-1

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{starEmp}}{\eta \Rightarrow \texttt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \texttt{case}_{s,r}\,(\eta''; z, e_1, x.xs.e_2) \Rightarrow \texttt{emp}_r}$$

S-Star-L-2

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{starDone} \qquad \eta'' \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p}{\eta \Rightarrow \texttt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} e_1' \Rightarrow p}$$

S-Star-L-3

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{starFirst}(p) \qquad \eta''[x \mapsto p, y \mapsto \texttt{emp}_{s^\star}] \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \texttt{let}_{s^\star}\ (x; y) = z\ \texttt{in}\ e_2' \Rightarrow p'}$$

S-Star-L-4

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \texttt{starRest}(p, p') \qquad \eta''[x \mapsto p, y \mapsto p'] \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p''}{\eta \Rightarrow \texttt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \texttt{let}\ x = \texttt{sink}_p\ \texttt{in}\ e_2'[z/xs] \Rightarrow p''}$$

S-Let

$$\frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p \qquad \eta[x \mapsto p] \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \downarrow^{n_1+n_2} \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' \Rightarrow p'}$$

S-HistPgm

$$\frac{M \downarrow v \qquad p = \texttt{toPrefix}_s\,(v)}{\eta \Rightarrow \langle M : s \rangle \downarrow^{n} \texttt{sink}_p \Rightarrow p}$$

S-Wait-1

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(x) = p \qquad \neg\,(p\ \texttt{maximal})}{\eta \Rightarrow \texttt{wait}_{\eta',t}(x)\,(e) \downarrow^{n} \texttt{wait}_{\eta'',t}(x)\,(e) \Rightarrow \texttt{emp}_t}$$

S-Wait-2

$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(x) = p \qquad p\ \texttt{maximal} \qquad \eta'' \Rightarrow e[\langle p \rangle / x] \downarrow^{n} e' \Rightarrow p'}{\eta \Rightarrow \texttt{wait}_{\eta',t}(x)\,(e) \downarrow^{n} e' \Rightarrow p'}$$

S-Fix

$$\frac{\overline{M} \downarrow \theta \qquad \eta \Rightarrow \texttt{let}\ \Gamma = A\ \texttt{in}\ e\ [e/\texttt{rec}]\ [\theta] \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow \texttt{fix}\,\{\overline{M}\}\,(A)\,.\,(e) \downarrow^{n+1} e' \Rightarrow p}$$

S-ArgsLet

$$\frac{\eta \Rightarrow A@\Gamma \downarrow^{n_1} A' \Rightarrow \eta' \qquad \eta' \Rightarrow e \downarrow^{n_2} e' \Rightarrow p \qquad \delta_{\eta'}\,(\Gamma) \sim \Gamma'}{\eta \Rightarrow \texttt{let}\ \Gamma = A\ \texttt{in}\ e \downarrow^{n_1+n_2} \texttt{let}\ \Gamma' = A'\ \texttt{in}\ e' \Rightarrow p}$$

Figure A.4: λ^{ST} Semantics (Part 2)

Figure A.5: λ^{ST} Recursive Argument Semantics

$$\frac{}{\eta \Rightarrow \cdot @ \cdot \downarrow^{n} \cdot \Rightarrow \{\}}\ \textsc{S-Args-Emp} \qquad \frac{\eta \Rightarrow e \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow e @ (x : s) \downarrow^{n} e' \Rightarrow \{x \mapsto p\}}\ \textsc{S-Args-Sng}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \eta \Rightarrow A_2 @ \Gamma' \downarrow^{n_2} A_2' \Rightarrow \eta_2}{\eta \Rightarrow (A_1, A_2)\, @ \Gamma, \Gamma' \downarrow^{n_1+n_2} (A_1', A_2') \Rightarrow \eta_1 \cup \eta_2}\ \textsc{S-Args-Comma}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \neg\, (\eta_1\, \texttt{maximalOn}\, \Gamma)}{\eta \Rightarrow (A_1; A_2)\, @ \Gamma; \Gamma' \downarrow^{n_1} (A_1'; A_2) \Rightarrow \eta_1 \cup \texttt{emp}_{\Gamma'}}\ \textsc{S-Args-Semic-1-1}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \eta_1\, \texttt{maximalOn}\, \Gamma \qquad \eta \Rightarrow A_2 @ \Gamma' \downarrow^{n_2} A_2' \Rightarrow \eta_2}{\eta \Rightarrow (A_1; A_2)\, @ \Gamma; \Gamma' \downarrow^{n_1+n_2} (\cdot;)\, A_2' \Rightarrow \eta_1 \cup \eta_2}\ \textsc{S-Args-Semic-1-2}$$

$$\frac{\eta \Rightarrow A @ \Gamma' \downarrow^{n} A' \Rightarrow \eta'}{\eta \Rightarrow (\cdot; A)\, @ \Gamma; \Gamma' \downarrow^{n} (\cdot; A') \Rightarrow \texttt{emp}_{\Gamma} \cup \eta'}\ \textsc{S-Args-Semic-2}$$

BIBLIOGRAPHY

[1] The whirlwind I computer. In *Managing Requirements Knowledge, International Workshop on*, page 70, Philadelphia, PA, 1951. doi: 10.1109/AFIPS.1951.20. URL https://doi.ieeecomputersociety.org/10.1109/AFIPS.1951.20.

[2] 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. doi: 10.1007/s00778-003-0095-z.

[3] 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.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. ISBN 9781931971331.

[5] Samson Abramsky and Achim Jung. *Domain theory*, page 1–168. Oxford University Press, Inc., USA, 1995. ISBN 019853762X.

[6] V. Michele Abrusci. Non-commutative intuitionistic linear logic. *Mathematical Logic Quarterly*, 36(4):297–318, November 2006. doi: 10.1002/malq.19900360405. URL https://doi.org/10.1002/malq.19900360405.

[7] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, Kris Kang, Sahil Parmar, Andrew Ling, Andrew Bitar, Ibrahim Ahmed, and Jonathan Ross. The groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–69. IEEE Computer Society, 2022.

[8] Umut Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Melon University, 2005.

[9] Umut A. Acar and Ruy Ley-Wild. *Self-adjusting Computation with Delta ML*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04652-0. doi: 10.1007/978-3-642-04652-0_1. URL https://doi.org/10.1007/978-3-642-04652-0_1.

[10] Marian Andrzej Adamski, Andrei Karatkevich, and Marek Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005.

[11] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, Adam Paszke, Norman A. Rink, Michael Schaarschmidt, Timur Sitdikov, Agnieszka Swietlik, Dimitrios Vytiniotis, and Joel Wee. Partir: Com-

posing spmd partitioning strategies for machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 794–810, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706981. doi: 10.1145/3669940.3707284. URL https://doi.org/10.1145/3669940.3707284.

[12] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. Synchronization schemas. 2021.

[13] Apache Software Foundation. *Kafka Streams Documentation*. Apache Software Foundation, 2024. URL https://kafka.apache.org/documentation/streams/. Accessed on May 7, 2025.

[14] 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.

[15] 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.

[16] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006. doi: 10.1007/s00778-004-0147-z.

[17] ARM Limited. *AMBA AXI-Stream Protocol Specification*. ARM Limited, Cambridge, UK, 2023. https://developer.arm.com/documentation/ihi0051/latest/.

[18] Arvind and R.S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990. doi: 10.1109/12.48862.

[19] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Clash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, 2010.

[20] Patrick Bahr. Simple modal types for functional reactive programming. Preprint, December 2025, December 2025.

[21] Patrick Bahr and Rasmus Ejlers Møgelberg. Asynchronous modal frp, 2023.

[22] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341713. URL https://doi.org/10.1145/3341713.

[23] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Diamonds are not forever: Liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434283. URL https://doi.org/10.1145/3434283.

[24] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. Adaptation-based programming in haskell. *Electronic Proceedings in Theoretical Computer Science*, 66:1–23, September 2011. ISSN 2075-2180. doi: 10.4204/eptcs.66.1. URL http://dx.doi.org/10.4204/EPTCS.66.1.

[25] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

[26] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. ISSN 0167-6423. doi: https://doi.org/10.1016/0167-6423(92)90005-V. URL https://www.sciencedirect.com/science/article/pii/016764239290005V.

[27] Aggelos Biboudis, Jeremy Gibbons, and Oleg Kiselyov. All things flow: Unfolding the history of streams (extended abstract), 2021. URL http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/streams-hapoc2021.pdf.

[28] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, page 174–184, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130244. doi: 10.1145/289423.289440. URL https://doi.org/10.1145/289423.289440.

[29] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[30] Amar Bouali. Xeve, an esterel verification environment. In *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*, pages 500–504. Springer, 1998.

[31] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with odes. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 113–118, 2013.

[32] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13): 1441–1451, 2014.

[33] Ajay Brahmakshatriya and Saman Amarasinghe. Buildit: A type-based multi-stage programming framework for code generation in c++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–51, 2021. doi: 10.1109/CGO51591.2021.9370333.

[34] James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Computer Science, pages 78–92, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31822-4. doi:

10.1007/11554554_8.

[35] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4), 1964.

[36] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. Dbsp: Incremental computation on streams and its applications to databases. *SIGMOD Rec.*, 53(1):87–95, May 2024. ISSN 0163-5808. doi: 10.1145/3665252.3665271. URL https://doi.org/10.1145/3665252.3665271.

[37] William H Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1), 1975.

[38] 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.

[39] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912152. doi: 10.1145/41625.41641. URL https://doi.org/10.1145/41625.41641.

[40] Martín Ceresa, Felipe Gorostiaga, and César Sáchez. Declarative stream runtime verification (hlola). In *Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20)*, volume 12470 of *LNCS*, pages 25–43. Springer, 2020. ISBN 978-3-030-64436-9. doi: 10.1007/978-3-030-64437-6_2. URL https://link.springer.com/chapter/10.1007/978-3-030-64437-6_2.

[41] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD*, pages 668–668, 2003.

[42] Yijia Chen and Lionel Parreaux. The long way to deforestation: A type inference and elaboration technique for removing intermediate data structures. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi: 10.1145/3674634. URL https://doi.org/10.1145/3674634.

[43] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11, 2017. doi: 10.1109/TASE.2017.8285623.

[44] Collection of Historical Scientific Instruments. Harvard IBM mark I – about. https://chsi.harvard.edu/harvard-ibm-mark-1-about, 2025. Harvard University.

[45] Gregory Collins and Doug Beardsley. The snap framework: A web toolkit for haskell. *IEEE Internet Computing*, 15(1):84–87, January 2011. ISSN 1089-7801. doi: 10.1109/MIC.2011.21. URL https://doi.org/10.1109/MIC.2011.21.

[46] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137583. doi: 10.1145/871895.871897. URL https://doi.org/10.1145/871895.871897.

[47] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 315–326, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291199. URL https://doi.org/10.1145/1291151.1291199.

[48] Joseph W. Cutler, Christopher Watson, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. Stream types. 2023.

[49] Joseph W. Cutler, Alex Collins, Bin Fan, Mahesh Ravishankar, and Vinod Grover. Pattern matching in ai compilers and its formalization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 63–76, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708934. URL https://doi.org/10.1145/3696443.3708934.

[50] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005. doi: 10.1109/TIME.2005.26.

[51] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, November 1996. ISSN 0164-0925. doi: 10.1145/236114.236119. URL https://doi.org/10.1145/236114.236119.

[52] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[53] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, December 1974. ISSN 0163-5964. doi: 10.1145/641675.642111. URL https://doi.org/10.1145/641675.642111.

[54] Farzaneh Derakhshan. *Session-Typed Recursive Processes and Circular Proofs*. PhD thesis, Caregie Mellon University, May 2021. URL https://www.andrew.cmu.edu/user/fderakhs/publications/Dissertation_Farzaneh.pdf.

[55] Philip Dexter, Yu David Liu, and Kenneth Chiu. The essence of online data processing. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):899–928, 2022.

[56] Henry DeYoung. *Session-Typed Ordered Logical Specifications*. PhD thesis, Carnegie Mellon University, 2020.

[57] Volker Diekert and Grzegorz Rozenberg. *The Book of Traces*. World Scientific, 1995. doi: 10.1142/2563.

[58] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–422, 2020.

[59] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *DSL*, 1997.

[60] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009. URL http://conal.net/papers/push-pull-frp.

[61] Conal Elliott and Paul Hudak. Functional reactive animation. In *Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1997.

[62] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Not.*, 37(5):13–24, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512532. URL https://doi.org/10.1145/543552.512532.

[63] Lasse Faurby Klausen, Philip Kristian Møller Flyvholm, and Patrick Bahr. Push-pull modal functional reactive programming. Symposium on Trends in Functional Programming 2026, to appear, January 2026.

[64] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 421–431. Springer International Publishing, 2019. ISBN 978-3-030-25540-4.

[65] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 248–262, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-60-6. doi: 10.4230/LIPIcs.CSL.2013.248. URL http://drops.dagstuhl.de/opus/volltexte/2013/4201. ISSN: 1868-8969.

[66] Apache Software Foundation. Apache Flink. https://flink.apache.org/, 2019. (Accessed July 2022.).

[67] Apache Software Foundation. Apache Samza. https://samza.apache.org/, 2019. (Accessed July 2022.).

[68] Apache Software Foundation. Apache Storm. https://storm.apache.org/, 2019. (Accessed July 2022.).

[69] Apache Software Foundation. Apache Beam. https://beam.apache.org/, 2021. (Accessed July 2022.).

[70] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via

high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[71] Dan Frumin. Semantic cut elimination for the logic of bunched implications, formalized in coq. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 291–306, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503690. URL https://doi.org/10.1145/3497775.3503690.

[72] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994. ISBN 0-201-63361-2.

[74] GHC Team. *Data.List*. The University of Glasgow, 2024. Haskell base library, version 4.22.0.0.

[75] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 223–232, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 089791595X. doi: 10.1145/165180.165214. URL https://doi.org/10.1145/165180.165214.

[76] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(87)90045-4. URL https://www.sciencedirect.com/science/article/pii/0304397587900454.

[77] Rainer Gmehlich. Specification and validation of embedded systems using lustre and argos. case study: The automatic headlight leveling system. *Design Automation for Embedded Systems*, 6:151–175, 2001.

[78] Gabriella Gonzalez. Pipes. https://hackage.haskell.org/package/pipes, 2022.

[79] Gabriella Gonzalez. Foldl. https://hackage.haskell.org/package/foldl, 2024.

[80] Felipe Gorostiaga and César Sánchez. Hstriver: A very functional extensible tool for the runtime verification of real-time event streams. In *Proc. of the 24th Int'l Symp. on Formal Methods (FM'21)*, volume 13047 of *LNCS*, pages 563–580. Springer, 2021. doi: 10.1007/978-3-030-90870-6_30. URL https://doi.org/10.1007/978-3-030-90870-6_30.

[81] Felipe Gorostiaga and César Sánchez. Stream runtime verification of real-time event streams with the striver language. *International Journal on Software Tools for Technology Transfer*, 23: 157–183, 2021. doi: 10.1007/s10009-021-00605-3. URL https://link.springer.com/article/10.1007/s10009-021-00605-3.

[82] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. Frames: data-driven windows. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, page 13–24, New York, NY, USA, Jun 2016. Association for Com-

puting Machinery. ISBN 978-1-4503-4021-2. doi: 10.1145/2933267.2933304. URL https://doi.org/10.1145/2933267.2933304.

[83] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, 05 1999. ISBN 9780262287500. doi: 10.7551/mitpress/4472.003.0016. URL https://doi.org/10.7551/mitpress/4472.003.0016.

[84] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 157–166, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915925. doi: 10.1145/170035.170066. URL https://doi.org/10.1145/170035.170066.

[85] N. Halbwachs. A synchronous language at work: the story of lustre. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, pages 3–11, 2005. doi: 10.1109/MEMCOD.2005.1487884.

[86] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Advances in Computing Science—ASIAN'99: 5th Asian Computing Science Conference Phuket, Thailand, December 10–12, 1999 Proceedings 5*, pages 1–12. Springer, 1999.

[87] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062354. URL https://doi.org/10.1145/3140587.3062354.

[88] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), 2014.

[89] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008.

[90] Tyler Hou, Michael Arntzenius, and Max Willsey. Stream programs are monoid homomorphisms with state, 2025. URL https://arxiv.org/abs/2507.10799.

[91] John Hui and Stephen A. Edwards. The sparse synchronous model on real hardware. *ACM Trans. Embed. Comput. Syst.*, 23(5), August 2024. ISSN 1539-9087. doi: 10.1145/3572920. URL https://doi.org/10.1145/3572920.

[92] *Portable Operating System Interface (POSIX).* IEEE, 2017.

[93] *pipe — create an interprocess channel.* IEEE and The Open Group, 2018. POSIX.1-2017 (IEEE Std 1003.1-2017).

[94] Intel Corporation. *Avalon® Streaming Interfaces*. Intel Corporation, 12 2018. URL https://www.intel.com/content/www/us/en/docs/programmable/683364/18-1/streaming-interfaces.html. Intel® Quartus® Prime Standard Edition User Guide: Platform Designer.

[95] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 14–26, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133367. doi: 10.1145/360204.375719. URL https://doi.org/10.1145/360204.375719.

[96] 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.

[97] Marcin Jakubowski. parquet4s: Read and write Parquet in Scala, 2018. URL https://github.com/mjakubowski84/parquet4s. Accessed: 2026-01-12.

[98] Jane Street. Incremental: A library for incremental computations, 2015. URL https://github.com/janestreet/incremental.

[99] Alan Jeffrey. Ltl types frp: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, page 49–60, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311250. doi: 10.1145/2103776.2103783. URL https://doi.org/10.1145/2103776.2103783.

[100] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74, 1974.

[101] 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.

[102] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2022.

[103] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. On series-parallel pomset languages: Rationality, context-freeness and automata. *Journal of Logical and Algebraic Methods in Programming*, 103:130–153, 2019. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2018.12.001. URL https://www.sciencedirect.com/science/article/pii/S2352220817302298.

[104] Oleg Kiselyov. Iteratees. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, pages 166–181, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29822-6.

[105] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to com-

pleteness. *SIGPLAN Not.*, 52(1):285–299, January 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009880. URL https://doi.org/10.1145/3093333.3009880.

[106] Oleg Kiselyov, Tomoaki Kobayashi, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. strymonas-ocaml: A code-generation-based library for fast, bulk, single-thread in-memory stream processing, 2023. URL https://github.com/strymonas/strymonas-ocaml.

[107] Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. *Journal of Functional Programming*, 18 (3):333–371, 2008. doi: 10.1017/S0956796807006570.

[108] András Kovács. Closure-free functional programming in a two-level type theory. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi: 10.1145/3674648. URL https://doi.org/10.1145/3674648.

[109] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, 48(9):221–232, sep 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500588. URL https://doi.org/10.1145/2544174.2500588.

[110] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014. doi: 10.1145/2535838.2535841. URL https://cakeml.org/popl14.pdf.

[111] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1):574–585, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453920. URL https://doi.org/10.14778/1453856.1453920.

[112] Shadaj Laddad, Alvin Cheung, and Joseph M. Hellerstein. Suki: Choreographed distributed dataflow in rust, 2024. URL https://arxiv.org/abs/2406.14733.

[113] Shadaj Laddad, Alvin Cheung, Joseph M. Hellerstein, and Mae Milano. Flo: A semantic foundation for progressive stream processing. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704845. URL https://doi.org/10.1145/3704845.

[114] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65 (3):154–170, 1958. doi: 10.1080/00029890.1958.11989160. URL https://doi.org/10.1080/00029890.1958.11989160.

[115] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563.

[116] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, February 1965. ISSN 0001-0782. doi: 10.1145/363744.363749. URL https://doi.org/10.1145/363744.363749.

[117] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 2009.

[118] Ming-Huan Lee, Kuo-Kai Shyu, Po-Lei Lee, Chien-Ming Huang, and Yun-Jen Chiu. Hardware implementation of emd using dsp and fpga for online signal processing. *IEEE Transactions on industrial electronics*, 58(6):2473–2481, 2010.

[119] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[120] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[121] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[122] Xunyun Liu, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein, and Rajkumar Buyya. E-storm: Replication-based state management in distributed stream processing systems. In *2017 46th international conference on parallel processing (ICPP)*, pages 571–580. IEEE, 2017.

[123] Justin Lubin, Jeremy Ferguson, Kevin Ye, Jacob Yim, and Sarah E. Chasins. Equivalence by canonicalization for synthesis-backed refactoring. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656453. URL https://doi.org/10.1145/3656453.

[124] 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. doi: 10.1145/564691.564698.

[125] Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In Peter Müller, editor, *Programming Languages and Systems*, pages 394–427, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

[126] 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.

[127] Danielle Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 346–375, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99336-8.

[128] Antoni Mazurkiewicz. Trace theory. In *Advanced course on Petri nets*. Springer, 1986.

[129] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL

https://doi.org/10.1145/367177.367199.

[130] Frank McSherry. Materialize: a platform for building scalable event based systems. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 3, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393089. doi: 10.1145/3524860.3544408. URL https://doi.org/10.1145/3524860.3544408.

[131] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[132] Frank McSherry et al. Timely Dataflow (Rust implementation). https://github.com/TimelyDataflow/timely-dataflow/, 2014. (Accessed July 2022.).

[133] Stephen Mell, Konstantinos Kallas, Steve Zdancewic, and Osbert Bastani. Opportunistically parallel lambda calculus. or, lambda: The ultimate llm scripting language, 2025. URL https://arxiv.org/abs/2405.11361.

[134] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(78)90014-4. URL https://www.sciencedirect.com/science/article/pii/0022000078900144.

[135] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485496. URL https://doi.org/10.1145/3485496.

[136] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 393–407, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385974. URL https://doi.org/10.1145/3385412.3385974.

[137] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. Modular hardware design with timeline types. *Proceedings of the ACM on Programming Languages*, 7(PLDI):343–367, 2023.

[138] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, 2002.

[139] 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.

[140] OCaml Team. *Stdlib.List*. Inria, 2025. OCaml standard library, version 5.3.

[141] Peter W O'Hearn and David J Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5 (2):215–244, 1999.

[142] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[143] Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. The essence of event-driven programming. 2016.

[144] Penn Engineering. Celebrating penn engineering history: ENIAC. https://www.seas.upenn.edu/about/history-heritage/eniac/, 2025. School of Engineering and Applied Science, University of Pennsylvania.

[145] Carl Adam Petri. Communication with automata. 1966.

[146] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000. doi: 10.1145/345099.345100.

[147] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[148] Jeff Pokalow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[149] Ocsigen Project and Lwt Contributors. Lwt: Ocaml promises and concurrent i/o, 2024. URL https://github.com/ocsigen/lwt. A library for concurrent programming in OCaml providing typed, composable promises.

[150] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[151] ReactiveX Contributors. Rxjava: Reactive extensions for the jvm, 2025. URL https://github.com/ReactiveX/RxJava. A Java VM implementation of Reactive Extensions.

[152] ReactiveX Contributors. Rxjs: Reactive extensions for javascript, 2025. URL https://rxjs.dev/. A reactive programming library for JavaScript.

[153] ReactiveX Contributors and Microsoft. Reactivex: Reactive extensions, 2025. URL https://reactivex.io/. An API for asynchronous programming with observable streams available for multiple programming languages.

[154] Greg Restall. *An Introduction to Substructural Logics*. Routledge, London, England, December 1999.

[155] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[156] Nick Rioux and Steve Zdancewic. Functional meaning for parallel streaming, 2025. URL https://arxiv.org/abs/2504.02975.

[157] Tore Risch. REMREC - a program for automatic recursion removal. Technical report, Uppsala University, Department of Information Technology, 1973. URL https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A46310. URN: urn:nbn:se:uu:diva-18538.

[158] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361061. URL https://doi.org/10.1145/361011.361061.

[159] Scott Rixner. *Stream processor architecture*, volume 644. Springer Science & Business Media, 2001.

[160] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, page 127–136, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301541. doi: 10.1145/1868294.1868314. URL https://doi.org/10.1145/1868294.1868314.

[161] Rust Project. Rust for embedded development, 2025. URL https://www.rust-lang.org/what/embedded. Accessed on May 7, 2025.

[162] Rust Project Developers. *std::iter::Iterator - Rust*. Rust Project, 2025. URL https://doc.rust-lang.org/std/iter/trait.Iterator.html. Rust Documentation.

[163] Hannes Saffrich, Yuki Nishida, and Peter Thiemann. Law and order for typestate with borrowing. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689763. URL https://doi.org/10.1145/3689763.

[164] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[165] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2), 2013.

[166] Michael Snoyman. Yesod web framework, 2012. URL https://www.yesodweb.com.

[167] Michael Snoyman. Conduit. https://hackage.haskell.org/package/conduit, 2023.

[168] 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.

[169] Caleb Stanford. *Safe Programming over Distributed Streams*. PhD thesis, University of Pennsylvania,

2022.

[170] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix multiplication beyond auto-tuning: rewrite-based gpu code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344821. doi: 10.1145/2968455.2968521. URL https://doi.org/10.1145/2968455.2968521.

[171] Marco Stolpe. The internet of things: Opportunities and challenges for distributed data analysis. *Acm Sigkdd Explorations Newsletter*, 18(1):15–34, 2016.

[172] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024. URL https://arxiv.org/abs/2403.01876.

[173] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986. doi: 10.1109/TSE.1986.6312929.

[174] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, Volume 7, Issue 1, March 2011. ISSN 1860-5974. doi: 10.2168/lmcs-7(1:10)2011. URL http://dx.doi.org/10.2168/LMCS-7(1:10)2011.

[175] Composewell Technologies. Streamly. https://hackage.haskell.org/package/streamly-core, 2023.

[176] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19)*, pages 1–10, Phoenix, AZ, USA, 2019. ACM. doi: 10.1145/3315508.3329973. URL https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf.

[177] Jesse A. Tov and Riccardo Pucella. Practical affine types. *SIGPLAN Not.*, 46(1):447–458, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926436. URL https://doi.org/10.1145/1925844.1926436.

[178] Jose Manuel Calderon Trilla. personal communication, 2024.

[179] Doan Quang Tu, ASM Kayes, Wenny Rahayu, and Kinh Nguyen. Iot streaming data integration from multiple sources. *Computing*, 102(10):2299–2329, 2020.

[180] 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.

[181] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, June 1986. ISSN 0164-0925. doi: 10.1145/5956.5957. URL https://doi.org/10.1145/5956.5957.

[182] Typelevel and FS2 Contributors. Fs2: Functional streams for scala, 2024. URL https://github.com/typelevel/fs2. A purely functional, effectful, and polymorphic stream processing library for Scala.

[183] *socket(2) — BSD System Calls Manual*. University of California, Berkeley, 1983. 4.2BSD.

[184] Arthur Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18:365–396, 12 1986. doi: 10.1145/27633.28055.

[185] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(90)90147-A. URL https://www.sciencedirect.com/science/article/pii/030439759090147A.

[186] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. North-Holland, Amsterdam, 1990.

[187] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[188] Thad B Welch, Michael G Morrow, and Cameron HG Wright. Using dsp hardware to control your world. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages V–1041. IEEE, 2004.

[189] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145/3434304.

[190] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993. ISBN 0-262-23169-7.

[191] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509581. URL https://doi.org/10.1145/2509578.2509581.

[192] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *24th Symposium on Operating Systems Principles (SOSP)*. ACM, 2013. doi: 10.1145/2517349.2522737.

[193] Hong-Sheng Zheng, Yu-Yuan Liu, Chen-Fong Hsu, and Tsung Tai Yeh. Streamnet: memory-efficient streaming tiny deep learning inference on the microcontroller. *Advances in Neural Information Processing Systems*, 36:37160–37172, 2023.

[194] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. Lazy maintenance of materialized views.

In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 231–242. VLDB Endowment, 2007. ISBN 9781595936493.

[195] ZIO Contributors. Zio: A type-safe, composable library for async and concurrent programming in scala, 2024. URL https://github.com/zio/zio.