Concurrency Control in Event-Driven Programs

by

Yonglun Li

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Engineering

at

The University of Wisconsin–Milwaukee

May 2023

Abstract

Concurrency Control in Event-Driven Programs

by

Yonglun Li

The University of Wisconsin–Milwaukee, 2023
Under the Supervision of Professor Tian Zhao

Functional reactive programming (FRP) is a programming paradigm that utilizes the concepts of functional programming and time-varying data types to create event-driven applications. In this paradigm, data types in which values can change over time are primitives and can be applied to functions. These values are composable and can be combined with functions to create values that react to changes in values from multiple sources. Events can be modeled as values that change in discrete time steps. Computation can be encoded as values that produce events, with combination operators, it enables us to write concurrent event-driven programs by combining the concurrent computation as events. Combined with the denotational approach of functional programming, we can write programs in a concise manner.

The style of event-driven programming has been widely adopted for developing graphical user interface applications, since they need to process events concurrently to stay responsive. This makes FRP a fitting approach for managing complex state and handling of events concurrently.

In recent years, real-time systems such as IoT (internet of things) applications have become an important field of computation. Applying FRP to real-time systems is still an active area of research. For IoT applications, they are commonly tasked to perform data capturing in real time and transmit them to other devices. They need to exchange data with other applications over the internet and respond in a timely manner. The data needs to be processed, for simple analysis or more computation intensive work such as machine learning. Designing applications that perform these tasks and remain efficient

and responsive can be challenging.

In this thesis, we demonstrate that FRP is a suitable approach for real-time applications. These applications require soft real-time requirements, where systems can tolerate tasks that fail to meet the deadline and the results of these tasks might still be useful. First, we design the concurrency abstractions needed for supporting asynchronous computation and use it as the basis for building the FRP abstraction. Our implementation is in Haskell, a functional programming language with a rich type system that allows us to model abstractions with ease. The concurrency abstraction is based on some of the ideas from the Haskell solution for asynchronous computation, which elegantly supports cancelation in a composable way.

Based on the Haskell implementation, we extend our design with operators that are more suitable for building web applications. We translate our implementation to JavaScript as it is more commonly used for web application development, and implementing the RxJS interface. RxJS is a popular JavaScript library for reactive programming in web applications. By implementing the RxJS interface, we argue that our programming model implemented in Haskell is also applicable in mainstream languages such as JavaScript.

© Copyright by Yonglun Li, 2023
All Rights Reserved

Table of Contents

1 Introduction **1**
1.1 Functional Reactive Programming (FRP) . . . 2
1.2 Variations of FRP . . . 2
1.3 Real-time IoT Applications . . . 4
1.4 Concurrent Programming with JavaScript . . . 5
1.5 Reactive Extensions for Data Streaming . . . 6
1.6 Overview . . . 7

2 Asynchronous Stream for Reactive IoT Programming **9**
2.1 Introduction . . . 9
2.1.1 Challenges of IoT Data Processing . . . 10
2.1.2 Functional Reactive Programming . . . 10
2.1.3 Proposed Solution . . . 11
2.1.4 Contributions . . . 13
2.2 Cancellable Continuation Monad . . . 14
2.3 Push-Based Reactive Stream . . . 19
2.3.1 Functor . . . 21
2.3.2 Monad . . . 23
2.3.3 Buffered Stream . . . 25
2.3.4 Real-Time Push Stream . . . 27
2.4 Pull-Based Data Stream . . . 30
2.4.1 Push Pull Conversion . . . 31
2.4.2 Event . . . 32
2.4.3 Behavior . . . 33
2.5 Discussion . . . 37
2.6 Related Work . . . 39
2.7 Summary . . . 41

3 A Concurrency Model with Cooperative Cancellation Concurrency for JavaScript **42**
3.1 Introduction . . . 42
3.2 Thread-like Concurrency . . . 43
3.3 Concurrency with Cancellation . . . 45
3.3.1 Thread ID and Cancellation . . . 45
3.3.2 Asynchronous Exception . . . 46
3.3.3 Fork and Hierarchical Cancellation . . . 47
3.3.4 Pause and Resume Threads . . . 49
3.3.5 Synchronization Mechanism . . . 51
3.4 Operational Semantics . . . 53
3.4.1 Program Transitions . . . 55
3.4.2 Transition Rules . . . 57
3.5 Additional Constructs . . . 60
3.6 Evaluation . . . 62

3.7 Related Work 65
3.8 Summary 68

4 Semantics and Debugging **69**
4.1 Introduction 69
4.2 Subscription as Dataflow Graph 71
4.3 RxJS 73
4.4 Operational Semantics for RxJS 74
4.4.1 Syntax 74
4.4.2 Runtime Values 76
4.4.3 Evaluation of Expressions 77
4.4.4 Event Propagation 80
4.4.5 Additional Operators 84
4.5 Implementation 86
4.5.1 Composite Observable 87
4.6 Testing and Debugging 90
4.6.1 Debugging with Subscription Graph 90
4.6.2 Stack Trace 90
4.6.3 Subscription State 92
4.6.4 Runtime Invariant 93
4.7 Related Work 94
4.7.1 FRP 94
4.7.2 RxJS 95
4.8 Summary 97

5 Conclusion **98**
5.1 Summary of Contributions 98
5.2 Future Work 99

List of Figures

1.1 A program that monitors the current THD and the power of an inverter. The solid lines represent asynchronous tasks and the dashed lines represent synchronous tasks. 4

2.1 A use case of the push-pull model, where the solid lines represent push computation or push/pull conversion, the dashed lines represent pull computation, and the dotted lines represent dynamic adjustment to the push-streams. 12
2.2 Illustration of the workflow of `fetchS`. 26
2.3 Illustration of the workflow of `controlS`. 29
2.4 Illustration of the workflow of computing inverter power. 33
2.5 The peak memory use (MB) of the test programs containing 2 to 128 push streams of 10KHz samples. 38

3.1 Thread cancellation through timeout. 46
3.2 The syntax of `AsyncM`, values, and terms 54
3.3 The syntax of program states 55
3.4 Structural congruence 56
3.5 Structural transitions 56
3.6 Transition rules for AsyncM values 57
3.7 Transition rules for terms, run thread, and termination 59
3.8 Transition rules for cancellation, pause, and resumption. Rule (Cancel-Stuck) has higher priority than Rule (Async). 60
3.9 A data streaming application, where V_a, V_b, V_c, V_{dc}, V_{rms} are voltages and I_a, I_b, I_c, I_{dc}, I_{rms} are currents. 62
3.10 The architecture of the data streaming application in Figure 3.9, where A, B, C, and V are threads. 62
3.11 The subscription of a RxJS `Observable` using 2 threads (arrow circles) and an emitter (middle circle). 63

4.1 The subscription graph of the type-ahead example. 71
4.2 The initial attempt to catch query errors. 72
4.3 The correct handling of query errors. 73
4.4 The syntax of λ_{rx}, where the shaded terms are runtime entities. 74
4.5 The runtime values of λ_{rx}, where x points to shared observables and r points to the subscriptions in the heap. 76
4.6 The evaluation context for expressions 77
4.7 The evaluation rules for expressions 78
4.8 The rules for subscribe operations. 79
4.9 The rules for unsubscribe operations. 81
4.10 The reduction rules for subscriptions 1 83
4.11 The reduction rules for subscriptions 2 85
4.12 The safety rules that check the number of event sources guarded by a catch observable or root subscriber. 93

// Acknowledgements

I would like to thank my supervisor Professor Tian Zhao for his support and patience throughout many years of my studies. It was his course on program analysis that led me to take an interest in studying in this area, and later his encouragement led me to the Ph.D. program. It would not have been possible to complete this thesis without his help. I am truly grateful for everything you have done for me.

I would also like to thank Professor John Boyland, Professor Christine Cheng, and the faculty members for the knowledge and experience that I have gained as their student and teaching assistant. They are an invaluable part of my life.

Finally I would like to thank my parents, family, and friends who have always been there for me.

Chapter 1

Introduction

Functional reactive programming (FRP) is a programming paradigm that utilizes a denotational approach for creating programs with time-varying values. To explain it in a simple way, time-varying values can be understood as values that change over time. They are primitive constructs in FPR and are composable in the way that the time-varying values are represented in the continuous time domain, thus they can be composed and transformed without losing precision by prematurely sampling the values. It is these properties of composability and denotational that make this an interesting approach.

FRP was first introduced in Fran [16] in 1997 for creating animations and graphical user interfaces (GUI). Over the years, it has also been applied to different fields of applications such as game development [38, 12], data processing [4], real-time systems [55], distributed systems [35, 41, 42], and are very much active in research. One of our goals in this thesis is to apply the FRP approach for creating real-time IoT applications. In addition, we also adapt our design for real-time Web applications. There are existing programming language extensions and libraries inspired by the FRP approach that is popular in Web development, such as FlapJax and RxJS, to provide a pattern for writing asynchronous programs. These libraries focus on providing an abstraction for better concurrency control because they are difficult to program correctly as the flow of the programs does not follow a linear execution. However, they deviate from the concepts of the classic FRP and introduce additional concepts that focus on discrete changes in values and the propagation of events. With side effects in asynchronous computation, it becomes difficult to understand the precise meaning of the resulting program. As such, they have often been complained about as being difficult to program and debug.

1.1 Functional Reactive Programming (FRP)

FRP is a programming paradigm introduced originally for creating animation and graphical user interfaces. The concepts of FRP are based on functional programming and reactive systems where programmers define time-varying values and combine them to create values that react to changes.

FRP introduces two types of value as primitives, behavior and event. A behavior is a function of time and Events are occurrences of sorts that happen at certain points in time.

$$\begin{aligned} Behavior\ a &= Time \to a \\ Event\ a &= [(Time, a)] \end{aligned}$$

Behaviors can be thought of as a function of time, representing a value that changes as time changes. Events represent occurrences of value that occur at some discrete time. These two types of values are the basis of FPR programs. In addition, a set of operators are provided for transforming these time-varying reactive values. Behaviors and events can be composed with operators and combined into a program that can describe complex behaviors. As an example, an animation of a ball bounding on the ground can be modeled using a behavior to represent the position of the ball, composed with operators and functions that simulate applying gravitational force onto the ball. The behavior can be further composed to create events of the ball touching the ground, reacting to the event by bounding the ball in the opposite direction. The animation sequence can be generated by sampling at any time as required for the desired frame rate.

1.2 Variations of FRP

In classic FRP, the original implementation suffers the problem of *space-time leak*. This problem occurs because a behavior could depend on past values that accumulate over time, leading the values to be kept in memory (space leak). Because Haskell features lazy evaluation strategy, it may need to evaluate the accumulated computation of

a behavior at the same time (time leak). A solution to the space-time leak is to use the Arrows abstraction in the category theory [29]. In this arrowized FRP variant, behaviors are represented as signal functions and composed using arrowized combinators like function composition where signals are not first-class values. A well-known implementation of arrowized FRP is the Yampa [12] library. Another variation of FRP attempts to address the space-time leaks problem by limiting the amount of historic data to store and enforcing strict evaluation when applying certain operators. In Krishnaswami's [26] FRP, its implementation uses a strategy to ensure there is no space-time leak by restricting reactive value that depends on future time to be only evaluated in the future and deleting all old values when the clock advances.

There are other variations of FPR that tackle different problems in FRP. Traditionally, FRP has pull-based implementation. Events that derive from behaviors must be polled periodically in order to detect the occurrence of events. Push-pull FRP [15] implements an evaluation strategy that combines the push and pull based approach that is driven by push events instead of depending on direct evaluation by polling. It also abandoned the notion of continuous time domain, where the reactive value of behaviors changes discretely. Many other derivatives also take on this semantics as it allows them to have a simpler and more efficient implementation for data-driven applications.

Other similar push-based FRP such as FrTime [11], Flapjax [34], Scala React [32], ReactiveX [50], and Elm [13] wait on event occurrences and only run when an event occurs. Though this provides timely responses to events and avoids re-computation when events do not occur, there may be *glitches* where the events propagated from the same source are not evaluated at the same time. Solutions to this problem usually involve a central planner that oversees the event dispatching and propagation. For example, in the original Elm, first-order signals form a graph where a global dispatcher takes events from their sources and push updates through the graph. In Flapjax, where a dataflow graph is used, the graph nodes are updated based on topological order. In Monadic FRP [47], a program runs in a loop where at each iteration, it collects a set of future events, uses blocking IO to wait for one of the events to occur, and then starts the next iteration with

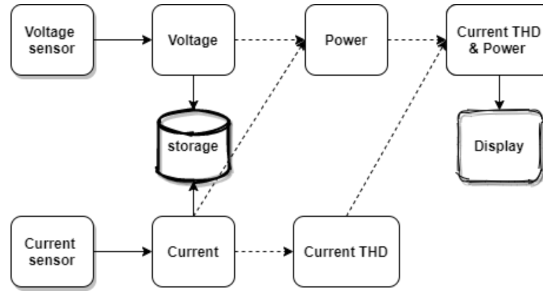

Figure 1.1: A program that monitors the current THD and the power of an inverter. The solid lines represent asynchronous tasks and the dashed lines represent synchronous tasks.

the event.

1.3 Real-time IoT Applications

Industrial IoT applications commonly perform tasks such as data capturing, storage, analysis, and display. These applications typically involve sensory devices collecting data and sending them over the network to a centralized platform where it is further processed for analytics. For example, we could have networks of sensors collecting data and calculating the Key Performance Indicators (KPI) for users to monitor the health of the systems. For a more demanding application, the data collected could be used in algorithms to make prediction on when maintenance is necessary. These algorithms may take much more time to complete compared to those in simpler applications.

Let us consider an application that monitors the health of an electric system. KPIs are calculated based on sensors reading from the voltage, current and among other measurements. Some calculations require high frequency of measured data. For example, a Total Harmonic Distortion (THD) calculation of a current signal requires discrete sampling at a rate that can reproduce the continuous signal. Depending on the input signal frequency, the amount of data collected could push the network to its limit. In some cases, the calculation may involve a moving window of historical data, whose window size can vary per signal.

For this example application, some of the key challenges are the followings. Sensor data sent over the network needs to be handled in real-time in-order without creating

excessive back-pressure. If there is a decrease in network speed, it is better to decrease the sampling rate of signals to adjust to available network bandwidth than to increase latency or lose data. Many devices could be connected to the system, so requests must be handled concurrently. In the events of issue being detected, such as a sensor going offline, we need to dynamically reconfigure the processing pipeline in the KPIs calculation to handle the failures. Because of these challenges, building IoT applications can be a complex task.

Chapter 2 will present our FRP based approach for real-time IoT applications. Since the research of FPR and many of its variations were presented in Haskell, our implementation will also be presented first in Haskell, along with their JavaScript implementation.

1.4 Concurrent Programming with JavaScript

In practice, IoT applications are often written in high-level programming languages that support concurrency for both ease of use and performance reasons. JavaScript is a popular choice because it is simple to learn and widely used in web technologies. However, JavaScript has many issues. For concurrent programming, the lack of flexible concurrency control in JavaScript makes it harder to maintain complex programs. Since JavaScript uses an event loop to drive concurrency, traditionally JavaScript programs have rely on using callback functions with side effects to program a concurrent application. The callback functions would be registered to the event loop, so that when an event occurs, the callback function is executed. Since JavaScript is single-threaded, it has no issue with race conditions that are common in a true multi-threaded program. However, data race may still exist when shared states have changed between the execution of the callback functions. For example, a long-running computation is broken into multiple callback functions, due to some IO operations. These callback functions are intended to run in sequence one after another. However, at runtime, the event loop may schedule to execute some other callbacks when it is in the middle of the sequence of callbacks, potentially changing the value in the shared state. A simple solution is to use additional variables to

keep track of the changes in between execution of callback functions. But as the program becomes more complex, it becomes harder to maintain.

Using many callbacks in the code creates another issue called the callback hell, where many callbacks are nested inside callbacks. In JavaScript, the solution is to use *promise*. The callback function, wrapped inside the promise object, resolves or rejects the Promise by calling the continuation. Multiple promise objects can be chained to run in sequence, allowing more modular code. However, there is no way to cancel a promise chain.

1.5 Reactive Extensions for Data Streaming

RxJS, or reactive extension for JavaScript, is a reactive programming library for JavaScript, which has been integrated into frameworks such as Angular and React to handle user interface and other asynchronous events. Due to the success of these frameworks, they popularized the use of RxJS in other event-driven applications. By itself, RxJS is a library based on the ideas of `Observables`, an abstraction for synchronous or asynchronous data streams. Operators are provided to manipulate values in data streams, or compose to make new Observables. However, RxJS can be difficult to program and debug due to its declarative and asynchronous nature since the declarative program logic greatly differs from the usual imperative programming style. Debugging is difficult because the program logic is also not directly reflected in the control flow logic represented by the call stack at the break point. Studies have found that users often have to resort to inserting print statements in the source code [1]. There are tools designed to give a visual representation of the RxJS program data flow in hope that the programmer can realize the mistakes they make. But the core issue is that these problems can be solved by presenting a formal model for the RxJS programs in order fully understand the expected behavior.

Since JavaScript does not have concurrency abstraction that supports cancellation, oftentimes programmers are resorted to using a flag in callback functions as a way to synchronize and cancel asynchronous computation. In Chapter 3, we will take our concur-

rency abstraction implemented in Haskell and re-implement it in JavaScript. In Chapter 4, using the concurrency abstraction in JavaScript, we will extend it by implementing the RxJS abstraction and present a formal operational semantics for RxJS programs.

1.6 Overview

This thesis is divided into three main chapters as follows:

In Chapter 2, we begin by discussing the problems in developing IoT applications that motivate us to find a better way to do concurrent programming with soft real-time capability. The rest of the chapter describes our solution based on the push-push FRP implemented in Haskell. We designed a reactive stream abstraction for real-time IoT applications. The design is based on the push and pull model to support efficient and high-throughput real-time data processing. As part of the implementation in Haskell, we designed a concurrency model which we will use as a model for transitioning to a JavaScript implementation. We call this abstraction `AsyncM` defined in section 2.2. Finally, there is a brief discussion of the implementation performance and the related works.

Chapter 3 presents a concurrency abstraction in JavaScript featuring operative cancellation. The motivation for needing this abstraction is because JavaScript does not have a concurrency model that supports cancellation. Our contribution is to provide a thread-like concurrency model for JavaScript, similar to the Promise abstraction, with proper support for synchronization and cancellation. The implementation is based on `AsyncM` that is implemented in Haskell, adding the ability to pause and resume. An operational semantics for this abstraction is also described (section 3.4). The runtime performance and the related works are discussed in the end.

Chapter 4 presents a formal model for RxJS programs. We provide an implementation of RxJS using the concurrency abstraction in Chapter 3, implementing the core set of operators in RxJS. We provide an operational semantics for implemented operators, with reduction rules modeling data flow logic as a subscription graph and reactive changes as

event propagation. Through this formal model, we describe how to debug RxJS programs like traditional debuggers, where we can set break-points and pause execution of the RxJS program and get back a stack trace that actually reflects the program flow.

The final chapter is the conclusion that summarizes the work done in this thesis and the plan for future work that could be done.

Chapter 2

Asynchronous Stream for Reactive IoT Programming

2.1 Introduction

With the proliferation of the web, increasingly more electronics are being attached to networks. Industrial electrical systems which once required offline controls and displays are now able to communicate and be controlled through the web. The increased demand for these IoT devices causes an increased demand in their reliability too. Occasionally the sole purpose of being connected to the web is to run these reliability checks. We call these "checks" *key performance indicators* (KPI). KPIs are calculations performed on signals (such as voltages or current readings from sensors) which can tell an end-user the health of the system. As a simple example, a system could perform an efficiency calculation for an internet-enabled solar panel to determine whether the solar panel is performing according to the manufacturer's efficiency rating. If the panel is not, the user would know to further investigate the device.

However, KPIs are often more complex than just a low sampling-rate point-wise calculation. Some KPIs can push networks to their limits. For example, a total harmonic distortion (THD) calculation of a current signal requires discrete sampling at a rate that can reproduce the original continuous signal. Depending on the input signal's frequency, it could force the sampling rate to be 10Khz or more. On top of that, the calculation involves a moving window, whose window size can vary per signal. A complete system could have many devices each with many signals each with their own KPI calculations. To monitor the health of such a system, software must be able to notify users of changes to the KPIs and react to changes with low latency without causing excessive back-pressure.

A monitoring system that runs computations on high frequency data from multiple devices must be concurrent, reactive, and composable. Concurrency is necessary so that

one device's computations do not block another device's, as well as to keep the latency low by processing multiple KPIs at once. Reactivity allows the system to scale dynamically in unstable environments. For example, if devices can set sampling rates on signals, during a decrease in network speed it would be advantageous to alter the sampling rate of the signals to avoid back-pressure. Composability is important to ensure program correctness is preserved when combining KPI processes on different signals.

2.1.1 Challenges of IoT Data Processing

An industrial IoT application runs in real-time with a large amount of data, performs frequent IO operations, and reacts to asynchronous events. In a workflow where we monitor the performance of a power inverter by capturing the voltage and current signals of the inverter, computing KPIs such as the power of the inverter and the THD of the current, storing the raw signals in a database, and displaying the KPIs on a dashboard. These tasks have different constraints on their speed and run with different sampling rates. For example, the signals may be captured and stored at a sampling rate of 20KHz but down-sampled to 10KHz for power and THD calculation, and the KPIs are displayed at 60Hz. To reduce latency, some tasks such as data capturing, storage, and display should be implemented asynchronously. However, there cannot be glitches for tasks such as inverter power, which must multiply the voltage and current sampled at the same time. Moreover, if the network and system load increases, some of the tasks may slow down, which can cause latency, memory leak, or data loss. To deal with this, each task can update its sampling rate to maintain its speed, which means that some tasks need to re-sample their input.

2.1.2 Functional Reactive Programming

A reactive programming model is a natural choice to process signals and allow for dynamic reactions to data. Reactive programming is a broad area of study ranging from pull-based FRP (e.g. Fran [16], Yampa [12], monadic stream functions [40, 4], FRP Now [48]), to push-based designs (e.g. Reactive Extensions [50], FrTime [11], Flapjax [34],

Scala React [32], Elm [13], Monadic FRP [47]), to hybrid model (e.g. push-pull FRP [15]). However, none of the existing models has the ideal characteristics that are specialized for IoT data that can scale sampling rates to its environment.

The core concepts of FRP [16] are behaviors and events, where a behavior is a continuous time function that can switch on events. In the existing FRP designs, the events and the behaviors have the same time domain. However, in IoT applications, the time parameter of IoT data analysis is the *data time* (i.e. when the data is sampled at the remote sensors, which may be in the arbitrary past if it is historical data), not the *system time* when the network events containing the IoT samples are received by the IoT programs.

The push-based models [34, 32, 13] use separate mechanisms such as signal graph to prevent harmful glitches by ensuring global ordering of events. While this works well for applications such as graphic interface, it is not suitable for IoT data. For example, the current and voltage data in Figure 1.1 should match their sampling time but the network events containing the data do not need to be globally ordered. Some pull-based models [48, 4] use scheduler or type-level clocks to combine asynchronous events and synchronous data at different sampling rate. However, they do not distinguish event time from the behavior time, which is necessary for adjusting sampling rate of the data in response to the changes in network and system load.

2.1.3 Proposed Solution

Our solution is based on the observation that separate stages of IoT computation can be implemented by either push or pull models, which can be connected by simple mechanisms like buffering and timeout loops (or clocks). We define push-based streams for handling system events. The push-streams can be converted to pull-based streams through buffering. We then use those pull-streams for pure computations such as KPI calculations. The pull-streams are driven by clocks to form push-streams for asynchronous computation related to storage, display, and user interface.

With this design, the push-streams do not need to maintain global ordering of the

asynchronous events since any synchronous computation that they are part of is driven by the same clock that pulls data from buffers, which ensures their ordering. The pull-streams do not interact with asynchronous IO directly since they only pull data from buffers.

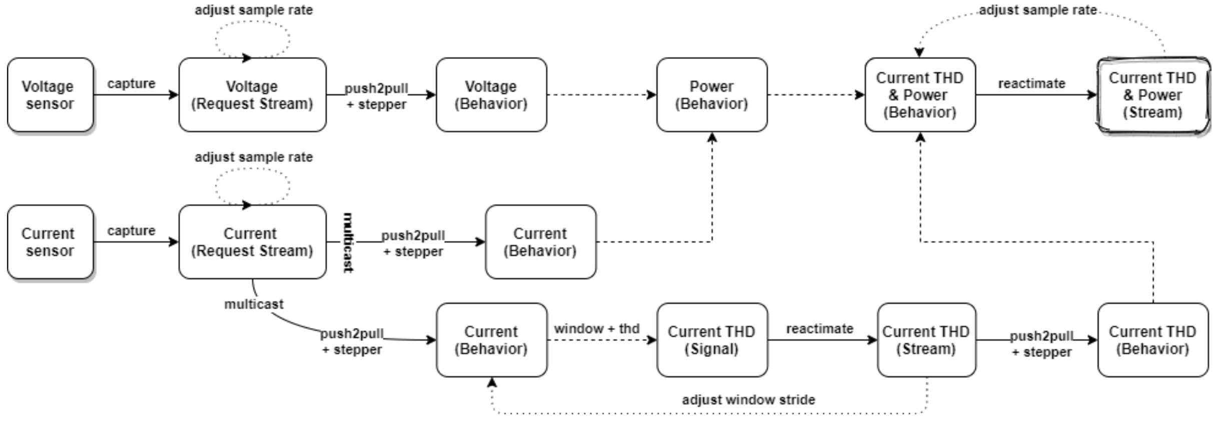

Figure 2.1: A use case of the push-pull model, where the solid lines represent push computation or push/pull conversion, the dashed lines represent pull computation, and the dotted lines represent dynamic adjustment to the push-streams.

Figure 2.1 illustrates a push-pull implementation of the workflow in Figure 1.1. The voltage (current) data is captured from the voltage (current) sensor using a push stream, which sends a stream of requests to capture the data and then emits the responses in the order in which the requests are sent. The stream sends data requests asynchronously to reduce latency and can adjust the sampling rate of the requested data to sustain its speed. The push streams are converted to pull streams (i.e. behaviors) through a buffering and a stepper function and the pull streams are used to compute power KPI. The conversion to pull-streams is to ensure that the voltage and current data can be synchronized and to allow re-sampling since the sampling rate of the push streams may change over time. The current data is also used to compute the THD of the current. The THD calculation includes Fourier transform on a sliding window of the current, which is computationally intensive and may need to increase the stride of the sliding window to maintain real-time speed. Since the current data for THD and power is pulled independently, the push stream for current is multicast into two push-streams before converting to pull streams. The power and THD of current are paired together and sent to display but since the

display is asynchronous, the power and THD behavior is converted to a push stream first with a reactimate function that samples the data around 60Hz. Similar method can be used to save the data to a storage (the details are omitted).

2.1.4 Contributions

Our design for push-stream is adapted from the *Reactive Value* abstraction proposed in push-pull FRP [15]. A reactive value is a value followed by another reactive value that occurs in the future. Reactive values are composable through a monadic interface. Joining higher-order reactive values requires racing two future reactive values. The original proposal was to use Haskell's threads. However, many IoT applications are implemented in dynamic languages that use event loops for concurrency. Also, our push streams can be shared through multicast or be converted to pull streams through buffering. This creates multiple streams with independent controls. A stream may continue to run even if its results are no longer used, which can cause memory leak.

Thus, our **first contribution** is a design of a reactive stream that represents future values using `AsyncM`, which is a form of continuation monad that implicitly carries a list of cancellation tokens. This design is lightweight and works with event loops or threads. Using `AsyncM`, we can race or cancel future values. If we cancel an `AsyncM` value, then any nested `AsyncM` values are cancelled as well, which is crucial for the dynamic switching of push-pull streams.

The **second contribution** is a hybrid push-pull design that supports high-throughput real-time data processing. The push-streams can make independent adjustment to their computation parameters to maintain real-time data speed. The pure functions on data can be implemented as synchronous computation on pull-streams.

For the rest of the chapter, we first introduce the design of a cancellable continuation monad in Section 2.2, which is the foundation for our hybrid push-pull model. In Section 2.3, we describe the monadic interface of our push-based stream and how it can be used to process IoT data in real-time. In Section 2.4, we show how pull-based signals/behaviors can be derived from push-streams, how to run signals/behaviors as push

streams, how to run computations with independently adjusted sampling rates. We discuss implementation and performance in Section 2.5 and related work in Section 2.6. The implementation is at `https://github.com/tianzhao/asyncm`.

2.2 Cancellable Continuation Monad

To implement an asynchronous and reactive model, we need an abstraction to represent asynchronous computation. The abstraction should not depend on threads so that it can be implemented in a single-threaded language like JavaScript. The abstraction should also support cancellation semantics so that a reactive stream can run independently, be shareable, and be stopped when it is no longer needed.

Our design is a continuation monad called `AsyncM`, which supports collaborative cancellation. Each AsyncM `m` runs with a progress value `p` so that if `p` is cancelled, `m` will stop at a checkpoint where it checks the cancellation status of `p`. The cancellation semantics is modular as `AsyncM` is composed with monadic bind and the progress value is threaded through the bind operator.

`AsyncM` is defined in Listing 2.1, which is a function that takes a `Progress` value and a continuation callback `k`, and passes its asynchronous result to `k`. A `Progress` value is a list of cancellation tokens, each of which has the "`MVar ()`" type, which either is empty (meaning alive) or has the unit value () (meaning cancelled).

```haskell
type AsyncM a = Progress -> (a -> IO ()) -> IO ()

type Progress = [MVar ()]
```

Listing 2.1: The definition of AsyncM

`AsyncM` is an instance of Functor, Monad, and MonadIO (where an IO action can be lifted to an `AsyncM` using `liftIO`) since it can be defined with monad transformers as follows.

```haskell
newtype AsyncM a = AsyncM {
    runAsyncM :: ReaderT Progress (ContT () IO) a
}
deriving (Functor, Applicative, Monad, MonadIO)
```

We can run an `AsyncM` for its side effect by calling it with an empty continuation.

```haskell
-- run an AsyncM for its side effect
runM :: AsyncM a -> IO ()
runM m = m [] (\_ -> return ())
```

For example, the `download` function defined below downloads an HTTP document and saves it to a file, but if the download is not completed in 2 seconds, the downloaded document is not saved.

```haskell
download :: String -> Path -> AsyncM ()
download url path = do r <- anyM (http url) (timeout 2000)
                       case r of Left str -> save path str
                                 Right _ -> return ()

timeout :: Int -> AsyncM ()
timeout n = \p k -> do async (threadDelay (n*10^3) >>= k)
                       return ()

http :: String -> AsyncM ByteString
http url = \p k -> do async (simpleHttp url >>= k)
                      return ()

save :: Path -> ByteString -> AsyncM ()
save path str = \p k -> writeFile path str >> k ()
```

The functions for racing and canceling `AsyncM`s are defined in Listing 2.2. Racing `m1` and `m2` is just to run them in parallel with a new progress value, where `m1` and `m2` should check whether the progress is still alive (with `aliveM`) and cancel it (with `cancelM`) before completion; otherwise the continuation `k` may be called twice.

The functions for `Progress` values are also in Listing 2.2. To cancel a `Progress` value, we put `()` in the head token, and to test whether a progress is alive, we check all of that value's cancellation tokens so an `AsyncM` can be cancelled in any context.

```haskell
anyM :: AsyncM a -> AsyncM b -> AsyncM (Either a b)
anyM m1 m2 = raceM (do x1 <- m1
                       commitM
                       return (Left x1))
                   (do x2 <- m2
                       commitM
                       return (Right x2))

commitM = ifAliveM >> cancelM
```

```haskell
raceM :: AsyncM a -> AsyncM a -> AsyncM a
raceM m1 m2 = \p k -> do p' <- consP p
                         m1 p' k
                         m2 p' k

ifAliveM :: AsyncM ()
ifAliveM = \p k -> do b <- isAliveP p
                      if b then k () else return ()

cancelM :: AsyncM ()
cancelM  = \p k -> do b <- cancelP p
                      if b then k () else return ()

-- extend the progress p with a new cancellation token
consP :: Progress -> IO Progress
consP p = (:p) <$> newEmptyMVar

-- test whether a progress is cancelled
isAliveP [] = return True
isAliveP (v:p) =
    do b <- isEmptyMVar v
       if b then isAliveP p else return False

-- try cancel a progress and return true if succeeds
cancelP :: Progress -> IO Bool
cancelP (v:_) = tryPutMVar v ()
```

Listing 2.2: Concurrency control with AsyncM.

As another example, `download'` attempts to download a document from a list of alternative URLs and if any download completes in 2 seconds, it is saved, and the program exits. Otherwise, the next URL in the list is tried.

```haskell
download' :: [String] -> Path -> AsyncM ()
download' [] _ = return ()
download' (url:rest) path =
    do r <- anyM (http url) (timeout 2000)
       case r of Left str -> save path str
                 Right _ -> download' rest path
```

Since `download'` contains the checkpoint `isAliveM`, we can stop the entire download process (e.g. after 5 seconds) as follows.

```haskell
download'' :: [String] -> Path -> AsyncM ()
download'' lst path =
```

```
anyM (download' lst path) (timeout 5000)
```

Note that for a finitely nested `anyM`, the cost of `ifAliveM` is not significant compared to the IO within the `anyM`. However, if a function is called recursively within an `anyM`, the size of the progress grows with each recursive call and the cost of `ifAliveM` increases correspondingly. To avoid performance issues, long-running recursive calls should occur outside of `anyM` (e.g. the push-stream's monadic join in Section 2.3.2).

Promises are similar to the continuation monads for concurrency [10, 28], which allow asynchronous callbacks be chained together without deeply nested scopes. One difference is that a `Promise` object, once constructed, starts immediately while continuation monads are started explicitly.

```
1   new Promise((resolve, reject) => {
2       // call resolve with results
3       // or call reject with error
4   )}
```

A `Promise` is instantiated with an executor function with two parameters: *resolve* and *reject*. A `Promise` runs exactly once, which results in a fulfilled state (if *resolve* is called) or a rejected state (if *reject* is called or an exception is thrown). If neither functions are called, then the `Promise` object remains in a pending state. Promises can be sequenced using *then* method and exceptions can be handled using *catch* method.

```
1   p.then(x => { /* returns a promise */ })
2    .catch(e => { /* handle error */ })
```

Our design is a cancellable concurrency monad with a thread ID. We implement it as a reader monad that wraps a function that takes a thread ID (of the type `Progress`) and returns a `Promise` object.

```
1   Progress -> Promise a
```

We define a JavaScript class `AsyncM` to represent this concurrency monad.

```
1 class AsyncM {
2       // run :: Progress -> Promise a
3       constructor (run) { this.run = run; }
4 
5       // pure :: a -> AsyncM a
6       static pure = x => new AsyncM (
```

```
7           p => Promise.resolve(x)
8         )
9         // fmap :: AsyncM a -> (a -> b) -> AsyncM b
10        fmap = f => new AsyncM (
11           p => this.run(p).then(f)
12        )
13        //bind :: AsyncM a -> (a->AsyncM b) -> AsyncM b
14        bind = f => new AsyncM (
15          p => this.run(p).then(x => f(x).run(p))
16        )
17 }
```

The static method `pure` converts a constant value to an `AsyncM` that always return that value. The `fmap` method applies a function to `this` `AsyncM` while the `bind` method composes `this` `AsyncM` with a function that returns an `AsyncM`. These methods allow `AsyncM`s to be composed so that a `Progress` value can be passed implicitly through the `AsyncM` computation. This style of composition does come with syntactic overhead since it prevents the direct use of `async` and `await` keywords for composing `Promises`.

```
1 // lift :: ((a -> ()) -> ()) -> AsyncM a
2 static lift = f => new AsyncM (p =>
3   new Promise((resolve, reject) => {
4     // cancel by throwing an exception
5                 let c = _ => reject("interrupted");
6
7                 if (!p.cancelled) { // check thread status
8                         p.addCanceller(c); // add a canceller
9
10                        // remove canceller when 'f' completes
11                        let k = x => {
12                                p.removeCanceller(c)
13                                resolve(x);
14                        }
15                        f(k) // starts an asynchronous operation
16                }
17                else c(); // cancel if the thread is dead
18  })
19 )
20
21 static timeout = n => AsyncM.lift(k =>
22                                 setTimeout(k, n))
```

An `AsyncM` that runs an asynchronous operation is constructed with the `lift` method,

whose parameter f performs an asynchronous operation such as `setTimeout`. The `lift` method also enables cancellation, which may happen when this `AsyncM` is blocked on its call to f. The `AsyncM` returns a resolved `Promise` if f completes with a result and it returns a rejected `Promise` if it is cancelled. For simplicity, the error handling of f is not described, which involves passing f a handler h so that if f raises an error e, then h removes the canceller c and calls the `reject` function with e.

2.3 Push-Based Reactive Stream

In many classic FRP implementations [16, 15], a behavior is a function from time to value, and an event source is a list of time/value pairs. A behavior can switch to new behaviors by reacting to the occurrence of events using a switch operator. Classic FRP samples the behavior values and detects event occurrences synchronously. However, IoT sensors are often distributed and synchronous sampling of sensor telemetries can cause unacceptable delay due to network latency. In addition, the overhead of synchronous sampling is unsuitable for high-frequency data such as the electrical signals that may be sampled at 10KHz or more. Thus, an asynchronous implementation is necessary for reactive IoT computations.

We adopt a push-based design for representing the stream of discrete events (similar to the `Reactive` value of push-pull FRP [15]). A value of the type "`Stream a`" contains a "`Maybe a`" value and a future stream "`AsyncM (Stream a)`".

```
-- the reactive stream
data Stream  a = Next (Maybe a) (AStream a)

-- the future reactive stream
type AStream a = AsyncM (Stream a)
```

The `Maybe` type is used for stream values because not all streams in IoT applications have sensible initial values when started. For example, if we set the initial value of a voltage stream to 0, there could be a KPI calculation which divides some signal by the voltage – producing a division by zero. In this case, `Nothing` should be the initial value, which will be skipped in KPI calculation.

```haskell
-- a stream that starts with Nothing
repeatS :: AsyncM a -> Stream a
repeatS m = Nothing `Next` repeatA m

 -- an asynchronous stream by repeating m
repeatA :: AsyncM a -> AStream a
repeatA m = do a <- m
               ifAliveM -- cancellation checkpoint
               return (Just a `Next` repeatA m)
```

Listing 2.3: Make a stream by running an `AsyncM` repeatedly.

We can make a stream with `repeatS m` (Listing 2.3) that waits for the value of m, checks the progress status, and then repeats itself. For instance, `repeatS (timeout 1000)` is a stream of 1 second intervals.

For the rest of the chapter, we assume there exists a function `getData` that captures sensor data by a sampling period in seconds and a duration in milliseconds.

```haskell
getData :: String -- name of the sensor
        -> Int    -- capturing time in milliseconds
        -> Double -- sampling period in seconds
        -> AsyncM [Double] -- resulting data batch
```

For example, `repeatS (getData "Va" 1000 0.0002)` is a stream of data batches, where each batch contains 1000 milliseconds of voltage data with a sampling period of 0.0002 seconds (or 5000 Hz). Streams like this can capture data from multiple sensors without accumulative delay.

A stream can be run with `runS` that sends the stream events to a function `k` which has side effects (e.g. saving data), where the `Nothing` events are skipped.

```haskell
runS :: Stream a       -- stream to run
     -> (a -> IO ())   -- side-effecting function
     -> AsyncM ()      -- resulting computation

runS (a `Next` ms) k =
     do ifAliveM       -- cancellation checkpoint
        liftIO (f a)   -- run 'k' with event 'a'
        s <- ms
        runS s k
  where -- no effect for the Nothing event
        f Nothing = return ()
```

```haskell
        f (Just x) = k x
```

Listing 2.4: Run a stream with a callback function.

2.3.1 Functor

`Stream` is a functor, where its `fmap` method (`<$>`) recursively applies `f` to the stream events. Since `a` is a `Maybe` value, "`f <$> a`" is `Nothing` if `a` is `Nothing` and is "`Just (f x)`" if `a` is "`Just x`".

```haskell
instance Functor Stream where
  fmap f (a `Next` ms) = (f <$> a) `Next` (fmap f <$> ms)
```

For example, the code below calculates the KPI of a sensor signal, saves it to a database, and at the same time, displays it on a user interface. The function `kpi` calculates a KPI value for every second of sensor samples, and `forkM` starts an `AsyncM` without waiting for it to complete.

```haskell
let s = kpi <$> repeatS (getData "Va" 1000 0.0002)
in do forkM (runS s writeDB)
      forkM (runS s display)

-- run m with a new progress p' and return p' immediately
forkM :: Async a -> Async Progress
forkM m = \p k -> do -- p' extends p with a new token
                     p' <- consP
                      -- discard the result of m
                     m p' (\_ -> return ())
                     k p' -- return p' right away
```

Listing 2.5: Run an `AsyncM` and return its progress value.

However, this example has a flaw since it captures sensor data and computes its KPI twice. A better version below avoids recomputing the stream `s` by broadcasting its values to an Emitter `e`, and then events from `e` are received by two separate streams for saving to a database and displaying.

```haskell
let s = kpi <$> repeatS (getData "Va" 1000 0.0002)
in do (e, _) <- broadcast s
      let s' = receive e
      forkM (runS s' writeDB)
      forkM (runS s' display)
```

The call “`broadcast s`” creates a new emitter `e`, emits each value of `s` to `e`, and returns `e`, whose values are received by the stream “`receive e`”. Since we use `runS` to broadcast the stream `s`, the `Nothing` events in `s` are skipped.

Both `broadcast` and `receive` (Listing 2.6) can be cancelled since both contain the checkpoint `isAliveM` (where the checkpoint of `broadcast` is in `runS`). Also, `broadcast` can be cancelled explicitly through the `Progress` value it returns.

```
-- broadcast the events of a stream to an emitter
broadcast :: Stream a -> AsyncM (Emitter a, Progress)
broadcast s = do e <- liftIO newEmitter
                 -- keep emitting events of s to e
                 p <- forkM (runS s (emit e))
                 -- return progress for cancellation
                 return (e, p)

receive :: Emitter a -> Stream a
receive e = Nothing `Next` h
  where h = do a <- listen e -- listen for event on e
               ifAliveM -- cancel checkpoint
               return (Just a `Next` h)
```

Listing 2.6: Broadcast a stream to an emitter to enable sharing.

The function `listen` takes an emitter `e` and returns an `AsyncM` that yields the future value of `e` by registering a callback on `e`. The call “`emit e a`” writes `a` to `e`, fires any callbacks registered on `e`, and then clears the callback list.

```
data Emitter a = Emitter (MVar a) -- the previous event
                         (MVar [a -> IO ()]) -- callbacks

newEmitter = pure Emitter <*> newEmptyMVar <*> newMVar []

-- listen for an event on an emitter
listen :: Emitter a -> AsyncM a
listen (Emitter _ kv) =
    -- add 'k' to the callback list of the emitter
    \_ k -> modifyMVar_ kv (\lst -> return (k : lst))

-- emit an event 'a' to the emitter
emit :: Emitter a -> a -> IO ()
emit (Emitter av kv) a = do
    tryTakeMVar av  -- clear previous event
```

```
    putMVar av a     -- set current event
    lst <- swapMVar kv [] -- clear callback list
    forM_ lst (\k -> k a) -- fire registered callbacks
```

Listing 2.7: Create, listen, or emit an event to an emitter.

2.3.2 Monad

`Stream` has a monad interface defined in Listing 2.8, where '`return x`' is a stream with just `x` followed by `neverM`, which is an `AsyncM` that never completes. The bind operator `>>=` is defined with a `join` function that flattens a stream of stream.

```
instance Monad Stream where
   -- neverM is an AsyncM that never completes
  return x = Just x `Next` neverM

  -- join the Stream of Stream 'fmap k s'
  s >>= k  = join (fmap k s)

join :: Stream (Stream a) -> Stream a
join (Nothing `Next` mss) = Nothing `Next` (join <$> mss)
join (Just s  `Next` mss) = switch s mss
```

Listing 2.8: Monad interface of reactive stream

The `join` function has two cases. In the first case, the inner stream is `Nothing` and we skip it, and continue to join the future outer stream. In the second case, we use the `switch` function in Listing 2.9 to run the inner stream `s` until the future outer stream `mss` emits.

```
1 switch :: Stream a -> AStream (Stream a) -> Stream a
2 switch (a `Next` ms) mss = a `Next` (h ms =<< spawnM mss)
3   where h ms mss =
4     let
5         f (Left ss) = join ss
6         f (Right (a `Next` ms')) = a `Next` h ms' mss
7     in
8         f <$> anyM mss (unscopeM ms)
```

Listing 2.9: Switch the inner stream when the future outer stream emits.

The `switch` function races the future inner stream `ms` with the future outer stream `mss` (line 8) using `anyM`. If `mss` wins the race with a new outer stream `ss`, then we abandon

ms and continue with "join ss" (line 5). If ms wins the race with an event a, then a is emitted and we continue the race of the next future inner stream ms' with mss (line 6).

The switch function calls the local function h (defined at line 3) to perform the race. If ms wins the race, the call reduces to "h ms' mss" (line 6), where ms' is the next future inner stream. Notice that mss is reused in the recursion, which is problematic for two reasons. First, mss may contain an asynchronous request (e.g. a timeout) that returns after the response is received. In this case, each run of mss will start a new request with a new completion time. Second, mss may be a composite AsyncM (e.g. another race), so restarting it wastes runtime. Therefore, when switch starts, it runs "h ms =<< spawnM mss" at line 2 to cache the result of mss using "spawnM mss", which starts mss and returns an AsyncM that waits for the result of mss.

We race ms and mss in "anyM mss (unscopeM ms)" (line 8), where unscopeM runs ms with the progress outside anyM. This is needed since anyM cancels its progress (and any AsyncM running with the progress) once the race completes but there may be a pending AsyncM started inside ms (e.g. using spawnM) that has not completed when ms wins the race.

```
1  neverM :: AsyncM a  -- never complete by not calling 'k'
2  neverM = \p k -> return ()
3
4  -- avoid cancelling 'm' inside a race
5  unscopeM :: AsyncM a -> AsyncM a
6  unscopeM m = \p k -> m (tail p) k
7
8  -- start m and return an AsyncM waiting for m's result
9  spawnM :: AsyncM a -> AsyncM (AsyncM a)
10 spawnM m = \p k -> do e <- newEmitter
11                       m p (emit e)
12                       k (wait e)
13
14 -- try reading a future event from the emitter and
15 -- register a callback if the event is not available
16 wait :: Emitter a -> AsyncM a
17 wait (Emitter av kv) = \p k ->
18               do a <- tryReadMVar av
19                  case a of Just x -> k x
```

```
20                                Nothing -> swapMVar kv [k]
```

Listing 2.10: Auxiliary functions to run `AsyncM`.

The function `spawnM` in Listing 2.10 starts an `AsyncM` with a callback that writes its result to an emitter `e` at line 11, and then returns "`wait e`" at line 12 to wait for the result. To avoid memory leaks, "`wait e`" only keeps one callback in `e` (using `swapMVar` at line 19) since `wait e` (returned from `spawnM`) can be started many times, but only the last instance is needed.

Use of Monad Interface Our monadic interface is convenient for switching. For example, we can define a stream below that displays the KPI of a sensor chosen by users, where `sensorSource` is an `AsyncM` that waits on user input to choose the sensor for KPI calculation.

```
let s = do src <- repeatS sensorSource
           repeatS (getData src 1000 0.0002)
in runS (kpi <$> s) display
```

Using `switch`, we can define functions such as `stopS` that stops a stream after `n` milliseconds.

```
stopS n s = s `switch` do timeout n
                          return (Nothing `Next` neverM)
```

2.3.3 Buffered Stream

There is usually some latency between capturing IoT sensor data and the KPI calculation. For example, it may take 2 seconds to run "`getData "Va" 1000 0.0002`" to retrieve 1 second of samples (with 1 second of network delay). If we only send a request after receiving the response from the previous request, then the samples between two successive requests will be lost. We may be able to hide this latency using a queue and two streams: the first stream sends the requests at a regular interval and pushes the future response for each request to the front of the queue. The second stream extracts the future response from the end of the queue and waits for it to resolve. If the latency is not due to the lack of network bandwidth, it may be able to be hidden this way.

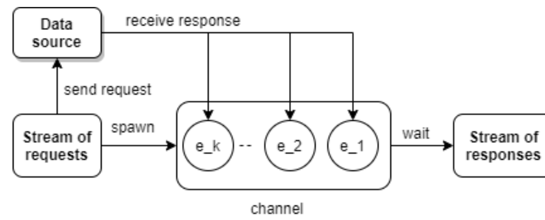

Figure 2.2: Illustration of the workflow of `fetchS`.

The example below creates a `request` stream at line 3 that sends a data request every second. Another stream `response` at line 5 is created using the `fetchS` function (Listing 2.11).

```
1 let clock = repeatS (timeout 1000)
2     -- request  :: Stream (AsyncM [Double])
3     request = getData "Va" 1000 0.0002 <$ clock
4     -- response :: Stream [Double]
5     response = fetchS request
6 in  runS (fmap kpi response) display
```

The `fetchS` function in Listing 2.11 spawns an `AsyncM` for each request in the `request` stream, and writes it to a channel (line 3). It returns a new stream (line 6) that reads each `AsyncM` from the channel and waits for it to yield a result. The workflow of `fetchS` is shown in Figure 2.2.

```
1 -- take a request stream and return a response stream
2 fetchS :: Stream (AsyncM a) -> Stream a
3 fetchS sm = Nothing `Next` do
4     c <- liftIO newChan
5     forkM $ runS (sm >>= liftS . spawnM) (writeChan c)
6     repeatA $ join $ liftIO (readChan c)
7
8 -- lift an AsyncM to a Stream
9 liftS :: AsyncM a -> Stream a
10 liftS m = Nothing `Next` (m >>= return . return)
```

Listing 2.11: Fetch data by sending requests in a stream.

In Listing 2.11, the expression "`sm >>= liftS . spawnM`" at line 5 converts the stream of requests `sm` to a stream of future responses using `spawnM`. At line 6, "`liftIO (readChan c)`" has the type `AsyncM (AsyncM [Double])`, which is flattened by `join` to `AsyncM [Double]` that reads a response from the channel and waits for it to complete.

2.3.4 Real-Time Push Stream

The `fetchS` function can help reduce the latency and increase the throughput of IoT data retrieval. However, the data retrieval throughput is also subject to limitations such as network bandwidth. For the previous example, if the amount of data being transmitted exceeds the network bandwidth, then the latency between IoT data capturing and KPI calculation will gradually increase, and the KPI calculation will no longer be in real-time. One way to prevent this problem is to monitor the speed of the response stream and if it is lower than the speed of the request stream, then we reduce the amount of data for each request (e.g. by lowering the sensor sampling rate).

To measure the speed of the response stream, we can use the `countS` function in Listing 2.12 to count the events of a stream during a time interval. The `countS` function is based on `foldS`, which folds a stream of functions for a duration, and returns the last event.

```haskell
-- count the number of events of s in n milliseconds
countS :: Int -> Stream a -> AsyncM Int
countS n s = foldS n 0 ((+1) <$ s)

-- fold s for n milliseconds with the initial value c
foldS :: Int -> a -> Stream (a -> a) -> AsyncM a
foldS n c s = do r <- lastS (accumulate c s) (timeout n)
                 return $ fromJust r

-- accumulate s with the initial value a
accumulate :: a -> Stream (a -> a) -> Stream a
accumulate a (f `Next` ms) =
    let a' = maybe a ($ a) f
    in Just a' `Next` (accumulate a' <$> ms)

-- return the last event of s when m returns
lastS :: Stream a -> AsyncM () -> AsyncM (Maybe a)
lastS s m = spawnM m >>= h s
  where h (a `Next` ms) m = do
          r <- anyM ms m
          case r of Left s -> h s m
                    Right () -> return a
```

Listing 2.12: Count the events of a stream in a period of time.

The function `foldS` calls `accumulate` (Listing 2.12) to recursively apply a stream of functions to an initial value, then uses `lastS` to stop the accumulation and returns the last value. Note that `countS` does not include `Nothing` since `accumulate` emits the same value if `f` is `Nothing`.

Using `countS`, we can implement a stream that makes runtime adjustments to sustain its speed. For example, the `getBatch` function in Listing 2.13 returns a stream of sample batches from a data source, where the sampling rate is adjusted based on the request/response speed. The function `req_fun` (line 4) takes the sampling-period `dt` and returns a request stream for 1 second of data sampled at `1/dt` Hz. The call to `controlS` at line 6 compares the number of requests sent and responses received within 10 seconds and adjusts `dt` by a ratio of 1.1 if the numbers are not equal.

```
1 getBatch :: String -> Stream (Double, [Double])
2 getBatch src =
3   let clock = repeatS (timeout 1000)
4       req_fun dt = getData src 1000 dt <$ clock
5       adjust b dt = if b then dt*1.1 else dt/1.1
6   in controlS req_fun (10^4) 0.0002 adjust
```

Listing 2.13: A data stream with adjustable sampling period.

```
1  controlS :: (t -> Stream (AsyncM a)) -- request function
2           -> Int                      -- duration (in ms)
3           -> t                        -- request parameter
4           -> (Bool -> t -> t)         -- adjust function
5           -> Stream (t, a)            -- result stream
6
7  controlS req_fun duration dt adjust = join $ h dt
8    where h dt = do
9      -- multicast creates shareable streams
10     (request, p1) <- multicast $ req_fun dt
11     (response, p2) <- multicast $ fetchS request
12
13     let mss = do -- account for initial latency
14                  timeout duration
15                  -- measure request/response speed
16                  (x, y) <- allM (countS duration response)
17                                 (countS duration request)
18                  if x == y then mss
19                  else do -- cancel multicast streams
```

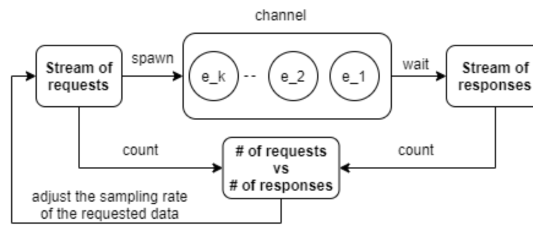

Figure 2.3: Illustration of the workflow of `controlS`.

```
20                          liftIO (cancelP p1 >> cancelP p2)
21                          -- restart adjusted stream
22                          return $ h $ adjust (x < y) dt
23     -- make a stream of response stream
24     Just ((,) dt <$> response) `Next` mss
25
26 -- run two AsyncMs and wait for both to return
27 allM :: AsyncM a -> AsyncM b -> AsyncM (a, b)
```

Listing 2.14: Make a request stream that can adjust to the speed of the response.

The function `controlS` (Listing 2.14) takes a request function `req_fun`, a duration, a request parameter `dt`, an adjustment function, and outputs a response stream that can self-adjust based on the relative request/response speed. The `controlS` function builds a stream of stream at line 23 with a `response` stream that runs until a new stream is emitted from `mss` (when the speed of request and response differs). The nested stream is joined at line 6, and the request parameter `dt` is added to the response stream at line 23. The basic workflow of `controlS` is shown in Figure 2.3.

The `controlS` function uses the `multicast` function in Listing 2.15 to make shareable streams such as `request` and `response`. However, the multicast streams will keep running even if they are not used. To avoid redundant computation, we cancel them by cancelling their `Progress` values at line 19 (Listing 2.14) when the stream switching occurs.

```
multicast :: Stream a -> Stream (Stream a, Progress)
multicast s =
    Nothing `Next` do (e, p) <- broadcast s
                      return $ return (receive e, p)
```

Listing 2.15: Make a shareable and cancellable stream.

2.4 Pull-Based Data Stream

While IoT data is often captured, stored, and displayed asynchronously, it is more convenient to analyze IoT data (e.g. calculating KPI) synchronously. For example, suppose we capture the voltage and current of an inverter in two push streams, whose data arrive in the following order.

```
Time     0    1     2     3     4     5
voltage  V1         v2          V3
current       I1          I2          I3
```

If we use the monad interface of `Stream` to compute the inverter power, we will produce incorrect output below.

```
Time     0    1     2     3     4     5
power       V1*I1 V2*I1 V2*I2 V3*I2 V3*I3
```

Since the voltage and current of the same data time may arrive at different system times, we have to match the voltage and current events by their sampling timestamps, which is costly. But, if the two streams are buffered, they can be processed synchronously by pulling their events from the respective buffers to yield the correct output.

```
Time     0    1     2     3     4     5
power       V1*I1       V2*I2       V3*I3
```

For this purpose, we define the `Signal` type in Listing 2.16, which is a recursive data structure that provides discrete data events on demand. Signal is an Applicative Functor, where the app operation `gf <*> ga` is defined as pairwise applications of the function signal `gf` to argument signal `ga`.

```
-- Pull-based stream
newtype Signal m a = Signal {runSignal :: m (a, Signal a)}

instance (Monad m) => Functor (Signal m) where
  fmap f g = Signal $ do (a, g') <- runSignal g
                         return (f a, fmap f g')

instance (Monad m) => Applicative (Signal m) where
  pure a = Signal $ return (a, pure a)

  gf <*> gx = do (f, gf') <- runSignal gf
                 (x, gx') <- runSignal gx
```

```haskell
                return (f x, gf' <*> gx')
```

Listing 2.16: The definition of pull-based `Signal` stream.

2.4.1 Push Pull Conversion

The function `push2pull` (Listing 2.17) converts a push-stream `s` to a pull-signal by sending the events of `s` to a channel buffer and returning a signal that reads from the channel. We run a signal using the `reactimate` function that returns a `Stream` that emits the signal values with a fixed delay.

```haskell
push2pull :: Stream a -> AsyncM (Signal IO a)
push2pull s =  do
     -- make an event buffer
     c <- liftIO newChan
     -- write events of 's' to the buffer
     forkM $ runS s $ writeChan c
     -- read events from the buffer
     let g = Signal $ do a <- readChan c
                         return (a, g)
     return g

-- run signal with event delay
reactimate :: Int -> Signal IO a -> Stream a
reactimate delay g = Nothing `Next` h g
  where h g = do timeout delay
                 ifAliveM
                 (a, g') <- liftIO $ runSignal g
                 return $ Just a `Next` h g'
```

Listing 2.17: Conversion between Stream and Signal.

It is more convenient to calculate KPI with `Signal`s. For example, the function `power` below computes the power of an inverter and returns the results in a stream of batches. Each event in "`power 1000 0.0002`" is a batch of 5000 samples since the sampling period is 0.0002 seconds, and each batch contains 1000 milliseconds of data.

```haskell
power :: Int -> Double -> Stream [Double]
power duration dt = do
   clock <- multicast $ repeatS (timeout duration)
   -- a stream of requests for a given source
   let req src = getData src duration dt <$ clock
```

```
-- convert a request stream to a signal
let req2signal s = liftS (push2pull (fetchS s))

voltage <- req2signal (req "Va") -- Signal IO [Double]
current <- req2signal (req "Ia") -- Signal IO [Double]

let power = pure (zipWith (*)) <*> voltage <*> current

reactimate 1 power -- run Signal as a push stream
```

Signal of data batches is not suitable for computation that needs to change sampling rate or operate on a moving window of data samples. We can create the voltage (and current) streams by calling `fetchS` on a stream of data requests that capture 1 sample per request. However, this is inefficient due to the overhead of sampling and transmission. A more realistic solution is to fetch data as a signal of batches and then convert the batch signal into sample signal.

In Section 2.3.3, we gave an example of a stream of voltage batches with varying sampling rate to match the speed of request and response. This creates a problem when we have a stream of voltages and a stream of currents with possibly different sampling rates. We need to re-sample the data before inverter power can be calculated, which requires the sampling period be included in the signal events. Moreover, KPIs such as THD (which can measure the quality of an inverter signal) take the sampling period as input. For these reasons, we need to have signals with time.

2.4.2 Event

`Event` is a signal of sampling-period and value pairs. Since the sensor signal converted from a request stream may have varying sampling rate, the sampling period should be included in the signal, which forms a sequence of delta-time and value pairs.

```
type DTime = Double -- sampling period

-- Event is a signal of sampling-period and value pairs
type Event a = Signal IO (DTime, a)
```

IoT data can be retrieved as an `Event` with a given sampling period using the function

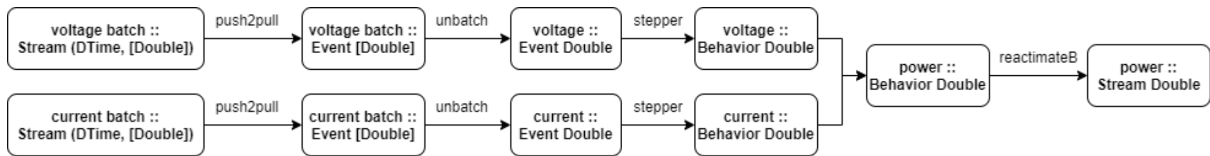

Figure 2.4: Illustration of the workflow of computing inverter power.

`fetchE` in Listing 2.18, which uses `fetchS` to retrieve a stream of the responses to the requests sent with a sampling period `dt`, and then pairs the results with `dt`.

```
-- fetch Event Signal
fetchE :: (DTime -> Stream (AsyncM a)) -- request stream
       -> DTime                        -- sampling period
       -> AsyncM (Event a)             -- event signal
fetchE req_fun dt =
    push2pull $ (,) dt <$> fetchS (req_fun dt)
```

Listing 2.18: Convert a request Stream to an Event.

An `Event` of batches can be easily converted to an `Event` of samples using the `unbatch` function in Listing 2.19, where the sampling period of a batch is repeated in the unbatched samples. When the sampling period of a batch changes, the corresponding sampling period of the unbatched samples changes as well.

```
-- flatten the Event of batches to an Event of samples
unbatch :: Event [a] -> Event a
unbatch eb = Signal $ do
     ((dt, b), eb') <- runSignal eb
     h dt b eb'
  where h _ [] eb' = runSignal $ unbatch eb'
        h dt (a:b) eb' =
          return ((dt, a), Signal $ h dt b eb')
```

Listing 2.19: Convert batch Event to sample Event.

An event of samples is not only easier for KPI calculations, but also allows re-sampling so that IoT data of varying sampling rate can be used in a computation with a constant sampling rate through re-sampling.

2.4.3 Behavior

To support re-sampling, below we define the `Behavior` type as a signal that takes a sampling-period as input. Each value of the behavior is the summary of some sample

values over a given sampling period.

```
type Behavior a = Signal (ReaderT DTime IO) a

stepper :: ([(DTime, a)] -> a) -- summary function
        -> Event a              -- input Event
        -> Behavior a           -- output Behavior

stepper sum ev = Signal $ ReaderT $ \t -> h [] t ev
   where h lst t ev = do
         ((t', a), ev') <- runSignal ev
         if (t == t')
         then return (sum ((t,a):lst), stepper sum ev')
         else if (t < t')
              then return (sum ((t,a):lst), stepper sum
                  $ Signal $ return ((t'-t, a), ev'))
              else h ((t',a):lst) (t-t') ev'
```

A `stepper` function can be defined to convert an `Event` to a `Behavior` with the help of a summary function that summarizes a sequence of time/value pairs to a value. The idea is to repeat the sample value of the Event for up-sampling (when the sampling period of the Behavior is shorter than that of the Event), and use the summary function for down-sampling (when the sampling period of the Behavior is longer). The summary function depends on the IoT data, which may be numeric values or discrete states.

Using the `Behavior` abstraction, we can calculate inverter power using two streams with variable sampling rates. For example, below is a code snippet that first obtains two self-adjusting streams by calling `getBatch` (Listing 2.13) and then converts the streams to behaviors using `stream2behavior` (Listing 2.20). The workflow of computing inverter power is shown in Figure 2.4, where `getBatch` produces voltage/current batches `Stream (DTime, [Double])`, which are streams of sampling period and data batch pairs.

```
do voltage <- stream2behavior (getBatch "Va")
   current <- stream2behavior (getBatch "Ia")
   let power = pure (*) <*> voltage <*> current
   runS (reactimateB 1 0.001 power) display
```

The `stream2behavior` function goes through the steps of converting a push-stream to an `Event` of batches, to an `Event` of samples, and to a `Behavior`.

```
avg :: [(DTime, Double)] -> Double -- weighted average
```

```
stream2behavior :: Stream (DTime, [Double])
                -> AsyncM (Behavior Double)
stream2behavior s =
                  (stepper avg . unbatch) <$> push2pull s
```

Listing 2.20: Convert a batch stream to a behavior

We run a behavior using the function `reactimateB` in Listing 2.21, which turns a behavior to a push-stream given a delay and sampling period.

```
reactimateB :: Int                -- delay between pulls
            -> DTime              -- sampling period
            -> Behavior a         -- input behavior
            -> Stream (DTime, a)  -- output stream

reactimateB delay dt b = Next Nothing (h b)
  where h b = do
      timeout delay
      ifAliveM
      (a, b') <- liftIO $ (runReaderT $ runSignal g) dt
      return $ Just (dt, a) `Next` h b'
```

Listing 2.21: Run a behavior as a stream.

KPI calculations can be expensive. For example, calculating THD for dozens of electrical signals can overwhelm some systems. To avoid this problem, we can measure the data speed by adding up the sampling periods of the stream `reactimate`d from a behavior and divide it by the system time used. If the ratio is less than 1 (or a number less than 1 considering runtime overhead), then it is operating at less than real-time speed and computation load should be reduced. The `speedS` function in Listing 2.22 is for this purpose.

```
-- measure the total amount of sample time
-- within an given interval of system time
speedS :: Int                -- duration in milliseconds
       -> Stream (DTime, a) -- stream of time/value pairs
       -> AsyncM Double      -- relative data speed

speedS n s =
    f <$> (foldS n 0.0 $ (\(dt,_) t -> t + dt) <$> s)
```

```
  where f t = t * 1000.0 / fromIntegral n
```

Listing 2.22: Measure the data speed of a Behavior.

Note that the sampling-rate of a stream `reactimated` from a behavior can be independent from the sampling-rate of the streams that the behavior depends on. For example, suppose that a stream of voltage samples is captured at 10KHz and is saved to a database. If the behavior for KPI calculations cannot run in real-time for this amount of data, then the behavior can be `reactimated` at a lower sampling rate (e.g. 5Khz). The data can still be captured at 10KHz and be saved to the database, but when the 10KHz stream is converted to the behavior, it is down-sampled by the stepper function.

```
-- convert a Behavior into an Event of data batches
window :: Int                          -- window size
       -> Int                          -- stride
       -> DTime                        -- sampling period
       -> Behavior a -> Event [a]

-- up-sample a Behavior by a factor
upsample :: Int                        -- up-sample factor
         -> Behavior a -> Behavior [a]

-- down-sample a Behavior by a factor
downsample :: Int                       -- down-sample factor
           -> ([(DTime, a)] -> a) -- summary function
           -> Behavior a -> Behavior a
```

Listing 2.23: Utility functions for `Behavior`

Additional operations such as up/down sampling can be defined to support computation that operates at different frequency. A `window` function (Listing 2.23) is especially useful for IoT data since KPIs such as THD takes batches of samples in order to calculate the harmonics of the electric signals. The `window` function can generate data batches based on a specific window size and stride (the number of samples to skip between two consecutive batches).

Assume we have a `thd` function that takes a sampling period and a list of values and returns the THD value. The `thd_stream` function below produces a stream of THD values from an inverter current behavior by calling the `window` function to make batch

input to the thd function.

```
thd :: DTime -> [Double] -> Double -- return THD value

thd_stream :: Int -> Int -> DTime -> Behavior Double
           -> Stream (DTime, Double)

thd_stream size stride dt behavior =
    reactimate 1 $ f <$> window size stride dt behavior
  where f (dt', w) = (dt', thd dt w)
```

We can use speedS function to measure the data speed of a stream and make runtime adjustment. The code below computes the THD of inverter current by first creating a stream of sliding windows of 5000 samples with the sampling period of 0.0002 seconds (line 5). The stride of the sliding window is 100, which means that one THD value is computed every $100 \times 0.0002 = 0.2$ seconds. Since larger stride means less computation, we can adjust the stride (line 10) if the data speed is outside the range of 0.9 to 1.1.

```
1  adjust :: Bool -> Int -> Int -- adjust the stride
2
3  do current <- stream2behavior (getBatch "Ia")
4     let f stride = do
5          stream <- thd_stream 5000 stride 0.0002 current
6          (s, p) <- multicast stream
7          let mss = do x <- speedS 1000 s -- measure speed
8                       if 0.9 < x && x < 1.1 then mss
9                       else do liftIO cancelP p
10                              f $ adjust (x < 0.9) stride
11         Just s `Next` mss
12    join $ f 100
```

2.5 Discussion

Our definition of Stream is similar to the monadic stream MStream in [40]. In particular, AsyncM (Stream a) can be defined as MStream AsyncM (Just a). However, the monadic interface of Stream depends on AsyncM. Also, for better efficiency, the Stream type in our implementation includes an End case. This change avoids the need to race any AsyncM with neverM even though the latter can never win a race.

```
data Stream a = Next (Maybe a) (AStream a)
```

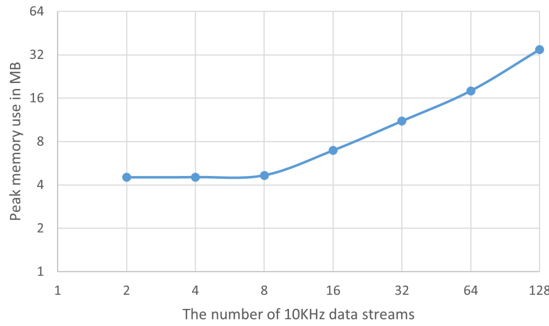

Figure 2.5: The peak memory use (MB) of the test programs containing 2 to 128 push streams of 10KHz samples.

```
               | End (Maybe a)

instance Monad Stream where
-- return x = Just x `Next` neverM
   return x = End (Just x)
```

The definition of `Signal` is structurally identical to `MStream`. We choose this name to mean sensor signal instead of the FRP signal [12], which is a time to value function.

This design intends to work with high-frequency IoT data (e.g. 5–20KHz) but our push streams are driven by clocks of much lower frequency. IoT sensor data is often transmitted in batches due to the limitation of the sensors and the transmission links. Our design is to handle the batches of high-frequency samples asynchronously (e.g. data capture, storage, display) and to process the individual samples synchronously (e.g. KPI calculation). The clocks that drive the push streams may operate in the range of 1–100Hz.

Performance To evaluate the overhead of our design, we ran tests that use several push streams to emit 10KHz of numeric data. The push streams are converted to behaviors, which are combined into one behavior using `Behavior`'s applicative interface and arithmetic operators. The final behavior is `reactimate`d to a push-stream that prints the results. The tests primarily measure the memory overhead of push/pull conversion, unbatching, and resampling. The peak memory use of the tests is shown in Figure 2.5, where the memory use is close to linear to the number of data streams.

The tests were run with one physical thread on an Intel i7 processor (4 cores). We do not have precise measurements of the runtime overhead but for 128 data streams, the

CPU usage is about 4% and for 64 data streams, it is about 2%.

2.6 Related Work

Push-Pull Our work was influenced by push-pull FRP [15], which was a modernization of Fran [16]. Push-pull FRP models a behavior as a reactive time function, where a push-event can cause a pull-based behavior to switch to another one. `Reactive` is recursively defined as a value followed by a future `Reactive` where the racing of future values is implemented with threads. Our push-stream shares the same structure as `Reactive` and its monadic interface. The difference is that we implement the future value using `AsyncM`. Racing future values with `AsyncM` is lightweight, does not involve threads so that it is suitable for dynamic languages with event loops. Moreover, a stream is also run as an `AsyncM`, which can be shared through `multicast`.

Variations of classic FRP First-class behaviors can lead to space-time leaks and wasteful re-computation. Jeltsch [24] used phantom types to tie discrete push-signals to specific start time to avoid restarting a signal after switching and used memoization to avoid duplicated computation of signals. The paper's motivation is related to stateful signals such as the one that counts network traffic. Such a signal is recomputed if used in multiple places and gets restarted after switching. Our push-stream does not prevent this type of issue through types. Instead, we can `multicast` a stream so that multiple uses will not cause re-computation and switching will not cause restart. Krishnaswami [26] used a static approach to ensure that past values cannot be accessed and Patai [39] achieved similar goals by distinguishing streams and streams of streams at the type level. FRP Now [48] provided a variation to Fran that does not cause space leaks and also supports asynchronous IO. This approach erases past values with an optimization based on mutable memory. It handles asynchronous IO in a behavior by running the IO on a new thread, which passes the results as an event to the next round of the clock that runs the behavior.

Arrowized FRP Another type of solution to the space-time leak problem is to use the Arrows abstraction [29]. Yampa is an arrowized FRP variant which composes signal functions using arrow combinators where signals are not first-class values. A drawback of the arrowized approach is that it requires inputs and outputs be threaded throughout the entire program, and imposes a point-free style of programming [12]. Scalable FRP [9] improved on Yampa by providing an imperative implementation which has most of the expressiveness of Yampa with better performance. Arrowized FRP has been generalized into a monad stream function in Ivan Perez's Dunai [40], which can model FRP signals and stateful reactive programming by stacking different monads. A later version called Rhine [4] introduced type-level clocks for processing data at different rates, where synchronous processes are run with an atomic clock on signal functions while asynchronous processes are run with schedules on resampling buffers. Rhine statically checks for correct composition involving clocks, and concurrent data is processed by threads that pass results through channels.

Dataflow Languages Before FRP, dataflow languages (e.g. Lucid [54]) and synchronous dataflow languages (e.g. Lustre [46] and SIGNAL [27]) provided an efficient and correct solution to real-time processing of signals. However, they are limited in power, as their dataflow graphs are static, and they do not support a form of first-class signals. In these models, signals use implicit time based on the ordering of events, rather than an explicit continuous time or discrete time interval. Without switching operator, adjusting sample rates to external factors is not possible with these languages. Hiphop.js [7] is a dataflow programming language that builds on the programming model of Esterel [5] and allows mixing of synchronous and asynchronous programming. Hophop.js focuses on Web orchestration with a declarative interface while our design is on real-time data processing.

Distributed Reactive Programming Distributed FRP focuses on solving issues such as glitch-freedom, scalability, and fault-tolerance that arise differently compared to non-distributed systems. QPROP [37] is a propagation algorithm designed for distributed

systems. It works by exploring the graph to find the dependency nodes before start propagation and its variant supports dynamic graph changes. XFRP [51] is a distributed FRP language based on actor model that compiles to Erlang. Though our model can interact with remote data sources and sinks, it is not a distributed design.

2.7 Summary

In this chapter, we presented a push-pull reactive programming model for IoT data processing which uses asynchronous streams for processing events and synchronous signals for processing data. Separating asynchronous streams from synchronous signals allows our model to isolate side-effecting computation which can run asynchronously such as fetching batches of data via HTTP requests from the pure synchronous computation of processing data.

Furthermore, we demonstrated how the model can be used for real-time processing of high sampling-rate signals while reacting to changes in processing speed by adjusting the sampling rate. This dynamic switching is afforded by the `AsyncM` monad, a continuation monad with implicitly threaded cancellation tokens, which is the basis for our asynchronous computations that allows for multi-threaded as well as single-threaded event loop implementations.

Chapter 3

A Concurrency Model with Cooperative Cancellation Concurrency for JavaScript

In the previous chapter, we presented the reactive abstractions for real-time IoT applications, implemented in Haskell. In this chapter, we propose a concurrency model for JavaScript with thread-like abstractions and cooperative cancellation. JavaScript uses an event-driven model, where an active computation runs until it completes or blocks for an event while concurrent computations wait for other events as callbacks. With the introduction of Promises, the control flow of callbacks can be written in a more direct style. However, the event-based model is still a source of confusion with regard to execution order, race conditions, and termination.

3.1 Introduction

JavaScript provides concurrency through its event loop, where a computation either runs or waits for an event as a listener. As JavaScript applications grow in complexity, it is common to have numerous callbacks with complex dependencies, which makes it difficult to identify concurrent computations. The introduction of Promises [14] allowed the control flow of event callbacks be written in a more direct style. A `Promise` object can encapsulate an event callback. Once constructed, the `Promise` object starts immediately and upon completion, it either resolves successfully with a result or rejects with an error value. When combined with `async` and `await` keywords, the `Promise` abstraction allows asynchronous operations be composed with synchronous operations in a sequential program.

It may seem unnecessary to provide a thread-based concurrency model for JavaScript since `Promise`s already behave like threads with non-preemptive scheduling and the meth-

ods to wait for their completion such as `race()` and `all()`. However, a `Promise` does not provide utilities for thread synchronization and for cancellation. User-defined constructs for such purposes may not have well-defined semantics, which can result in unexpected behavior. Without proper cancellation, a JavaScript program may contain abandoned computations running in the background, producing unexpected side effects. In this chapter, we propose a concurrency model implemented as a library with a formal semantics of thread synchronization and cancellation. A thread, once started, runs until it completes, is blocked on an event, is paused, or is cancelled. A paused thread can be resumed or cancelled.

In this chapter, we make the following contributions.

- We motivate the need of a JavaScript concurrency model with cancellation semantics in Section 3.2.
- We propose a library-based design[1] using the reader monad to represent thread-like abstractions in Section 3.3. The reader monad implicitly passes a thread ID throughout a computation so that it can be cancelled, paused, and resumed.
- We define a primitive like Haskell's `MVar` and show how it supports communication and cancellation in Section 3.3.5 and how other primitives such as bounded buffer can be defined in Section 3.5.
- We give an operational semantics of our concurrency model in Section 3.4.
- We discuss the usability and overhead in Section 3.6.

3.2 Thread-like Concurrency

JavaScript concurrency is based on event handling via its event loop. Event sources behave like stateful objects that dispatch events to registered listeners. The desire to have thread-like behavior resulted in libraries such as node-fibers, which is an implementation of coroutine in node-js with non-preemptive scheduling. This work is not to replicate

[1] `https://github.com/tianzhao/thread`

such capabilities, or suggest a solution of parallel computation using web workers, or leverage the idle time of event-loop for synchronous operations. Our focus is on thread-like abstraction for asynchronous computation, where each thread continues to run until it is blocked, paused, or cancelled. Our goal is help reduce concurrency errors in JavaScript by bringing back the familiar concepts of thread, synchronization primitives, and cancellation mechanisms.

Event races are common types of concurrency errors in JavaScript programs where multiple events arrive in an order or at a rate that is not expected by the programming logic, resulting in unexpected effects [44, 49, 23]. For example, event-race errors can be caused by the concurrent access to an external resource (e.g. a web service) if it does not protect against such access. These errors can be difficult to debug since they are reflected in the incorrect states at the external resource instead of at the JavaScript program. Furthermore, research indicates that even well-tested JavaScript applications often do not adequately cover event-dependent or asynchronous callbacks [19], inviting alternative methods to identify issues in such constructs.

While Promises [14] have helped reduce deeply-nested callbacks, its semantics is still complex [30, 36] and the number and breadth of issues reported on platforms like Stack Overflow indicate that users often struggle to understand its proper use. Static methods like Promise Graph [31] were proposed to track when `Promise`s are defined and activated/resolved as a step toward helping developers identify issues, it still does not indicate when pieces may execute in parallel and may cause concurrency errors.

We argue that the thread abstraction has several advantages over Promises.

- The first advantage is conceptual. To run an operation in a separate thread, one must start the thread explicitly. However, a `Promise` object is concurrent by default. If a `Promise` object is run for its side effect, one can easily forget to sequence it (e.g. using *await*) without realizing that it may run in a different order.

- Secondly, since threads have a well-defined abstraction boundary, it is easier to recognize concurrent access to shared resources so that synchronization primitives can be used to protect their access.

- Thirdly, `Promise` objects do not have methods for cancellation. While we can use the `race()` method (which waits for and returns the first value produced by a collection of `Promise` objects) to implement a task such as to timeout an asynchronous operation, the operation does not actually stop – only its results are abandoned. Though JavaScript is not preemptive, cancelling a thread at the earliest opportunity can reduce unintended side effects from the abandoned operations.

3.3 Concurrency with Cancellation

3.3.1 Thread ID and Cancellation

The thread IDs are used for cancellation. A thread ID is a `Progress` object, which forms a tree, where each tree node has a cancellation flag and a set of canceller functions.

```
class Progress {
  constructor(parent) {
    if (parent) {
      this.parent = parent;
      parent.children.push(this);
  }}
  cancelled = false
  children = []
  cancellers = [] }
```

To start a thread, we simply run an `AsyncM` with a new `Progress` object and return it.

```
class AsyncM {
    start = _ => {
        let p = new Progress()
        let f = _ => this.run(p)
        setTimeout(f, 0) // start f asynchronously
        return p
}}
```

When a lifted `AsyncM` is run with a `Progress` object p, a canceller function is added to p. If the `AsyncM` completes, the canceller is removed. If the `AsyncM` is cancelled before its completion, then the canceller runs, which causes the `AsyncM` to return a rejected `Promise`.

For example, the variable m below fetches data using an $ajax$ call, performs some

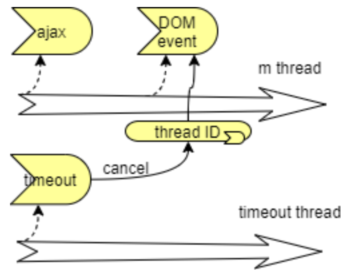

Figure 3.1: Thread cancellation through timeout.

computation, and sends the results for display. We start m with a thread ID t, which is used to cancel m if it is not completed within a second.

```
let m = AsyncM.lift(ajax) // fetch data
         .fmap(compute) // synchronous action
         .bind(display) // visualize data

let t = m.start() // starts m with thread ID t

AsyncM.timeout(1000)
       .fmap(_ => t.cancel())
       .start()   // starts a timeout thread
```

As shown in Figure 3.1, the timeout thread references the thread ID of m and may use it to cancel m when m is blocked on an event (e.g. when m calls the `ajax` or `display` function). However, like other non-preemptive designs, the timeout will not have immediate effects if it occurs while a synchronous operation like `compute` is running.

Unlike a `Promise` object, which runs immediately after its composition, the composition of an `AsyncM` object is separate from its execution, which helps identify the concurrent operations. One can delay the execution of a `Promise` by defining a function that returns a `Promise` (such as an `async` function). However, there is no syntactic difference between calling an `async` function and calling a regular function. It is easy to forget the difference between calling an `async` function with or without using `await` to wait for its completion.

3.3.2 Asynchronous Exception

When a thread ID t is cancelled, an interrupt exception is sent to the thread running with t. This design is similar to the asynchronous exception of Concurrent Haskell [33], except that our interrupt exception can only be received at some locations. In our case, the exception is received immediately if the thread is blocked, resulting in a rejected

`Promise`. Otherwise, the exception is received when the thread makes a blocking call or checks the status of the thread ID. For example, if a thread cancels another thread, the effect is immediate. However, if a thread cancels itself, there may be some delay.

The interrupt exceptions may be received by an `AsyncM` such as `AsyncM.lift(f)`. When this `AsyncM` runs, it first checks whether its `Progress` p is alive. If it is not alive, then it returns a rejected `Promise`. Otherwise, it adds a canceller c to the progress object p so that if p is cancelled, then c will be called to cause an interrupt exception.

The interrupt exception of the previous example can be handled with the *catch* method as shown below.

```
class AsyncM {
  catch = h => new AsyncM(p =>
                            this.run(p).catch(h))
}
let m = AsyncM.lift(ajax) // fetch data
          .fmap(compute) // synchronous action
          .bind(display) // visualize data
          .catch(print)  // print exception
```

3.3.3 Fork and Hierarchical Cancellation

Other than starting an independent thread, we can also fork a child thread with a `Progress` object that is a child of the current `Progress`. The parent `Progress` has a reference to the child progress so that if the parent is cancelled, so is the child. When the forked thread completes, the reference from a parent `Progress` to its child `Progress` is removed using the `unlink` method to avoid memory leak.

```
class AsyncM {
  fork = _ => new AsyncM (async p => {
      const p1 = new Progress(p)
      // start the thread asynchronously
      // unlink the reference from p to p1
      // after the thread completes
      AsyncM.timeout(0).bind(_ => this).run(p1)
            .finally(_ => p1.unlink());
      return p1;
  })
}
class Progress {
  // remove the parent to child reference
  unlink = _ => {
      let p = this.parent
      if(p) p.children = p.children
```

```
17                              .filter(c => c != this)
18    }
19    // call all cancellers recursively
20    cancel = _ => {
21      this.cancelled = true // set cancel flag
22      this.cancellers.forEach(c => c())
23      this.children.forEach(c => c.cancel())
24      this.cancellers = [] // clear cancellers
25    }
26 }
```

The cancel method of the Progress class sets the cancellation flag, calls each registered canceller to signal interrupts, and recursively cancels its children.

Using fork, we can run the m thread in the last example as a child of the timeout thread so that both threads can be cancelled by a user action such as pressing a “stop” button.

```
1  // run 'm' as a child of the timer thread
2  let t = m.fork()
3           .bind(t1 => AsyncM.timeout(1000)
4                        .fmap(_ => {
5                           t1.cancel()
6                           console.log("timeout")
7                        })
8           ).start()
9
10 // user cancels 'm' and the timer thread
11 $("#stop").one('click', _ => t.cancel())
```

The above example has 3 possible outcomes:

1. m completes,

2. m is cancelled by the timer, which prints ‘timeout’ message, and

3. user stops both m and the timer thread.

If it is inconvenient to use fork and bind, one can also run threads directly with Progress objects as shown below.

```
1 let t = new Progress()
2 let t1 = new Progress(t)
3
4 m.run(t1) // run m with t1
5
6 AsyncM.timeout(1000)
7       .fmap(_ => {
8            t1.cancel()
```

```
            console.log("timeout")
        }).run(t) // run timer with t

$("#stop").one('click', _ => t.cancel())
```

We can also use a `Progress` like a cancellation token. For example, we can start a set of threads by calling their `run` methods with a `Progress` t or the children of t so that any thread in the group can cancel the group through t.

3.3.4 Pause and Resume Threads

Our threads can be paused and resumed. This is useful in cases such as debugging and the implementation of user interfaces. For example, users can pause threads in browser console to check the states of the program or add controls to user interface to pause and resume animation threads such as real-time data charts.

```
$("#pause").on('click', _ => t.pause())
$("#resume").on('click', _ => t.resume())
```

We can add buttons to the previous example to allow users to pause and resume the timer and m threads. Unlike thread cancellation, which throws exceptions to the cancelled threads, thread suspension is based on polling. A thread returning from a blocking call checks whether it is paused and if so, it adds its continuation to the progress object, on which the `pause` method is called. This means that pausing a thread does not suspend its asynchronous calls but the handlers to the calls.

Like cancellation, thread suspension is hierarchical. If we pause a thread t, then t and its children are paused. Also, while a paused thread can be cancelled, a cancelled thread cannot be resumed. Pausing a cancelled or completed thread has no effect. For our example, if `t.pause()` is called before the timer and m complete, then m may be suspended after the `ajax` or `display` call returns while the timer thread will not advance beyond the timeout.

To reduce unintended side effects, a thread can only be resumed by the same `Progress` that the thread is paused with. That is, the threads that are paused together can only be resumed together. For example, if m is paused by the call `t.pause()`, then it can only be resumed by the call `t.resume()`. A thread can be paused or resumed by any

code with access to its thread ID. Thus, it is possible that a paused thread t can remain paused forever, if the thread that will resume t is cancelled.

```
static lift = f => new AsyncM (
  p => new Promise(
    (resolve, reject) => {
      let c = _ => reject("interrupted");

      if (!p.cancelled) {
        p.addCanceller(c);

        let k = x => {
          p.removeCanceller(c)

          // suspend the thread if it is paused
          if (! p.isPaused(_ => resolve(x)))
            resolve(x);
        }
        f(k)
      }
      else c();
    }
  )
)
```

The `lift` method above is revised to poll the pause status of a thread. It checks whether its `Progress` p is paused after the lifted function f returns and if so, it adds the `resolve` continuation to p.

```
class Progress {
  paused = false // pause status flag
  pending = []   // pending thread continuations

  pause = _ => { this.paused = true }

  isPaused = k => { // k: thread continuation
    if (this.paused) {
      this.pending.push(k)
      return true
    }
    else if (this.parent) {
      return this.parent.isPaused(k)
    }
    else {
      return false
    }
  }
  // resume paused threads
  resume = _ => {
    this.paused = false
    if (! this.cancelled)
      this.pending.forEach(k => setTimeout(k, 0))
    this.pending = []
  }
```

```
}
```

The `Progress` class is added a pause-status flag and a list of the pending thread continuations. The `pause` method simply sets the status flag to true while the `isPaused` method checks the status of this progress and its ancestors recursively and adds the continuation k is to the paused `Progress`. The `resume` method restarts paused threads by calling the pending continuations after a timeout.

3.3.5 Synchronization Mechanism

In JavaScript, an event race can be caused by the concurrent access to shared resources. To help prevent event races, we include synchronization primitives similar to Haskell's `MVar`, which can be used as locks to protect resources from concurrent access. For example, in a real-life application[2], a bug was caused by an user interface that sends concurrent requests to a remote service without support for concurrent access. If a user sends a new request before the previous request completes, then the new request will cause an internal error in the remote service.

The code below illustrates this problem, where the response to the request is sent to an user callback *cb*.

```
$("#button").on("click", _ => sendRequest(cb))
```

A simple fix is to use a flag to stop the handler from responding to the button click before a request completes.

```
let flag = true
$("#button").on("click", _ => {
    if (flag) {
        flag = false;
        sendRequest(x => {
            flag = true
            cb(x)
        })
    }
})
```

However, this fix is not ideal since some clicks would lead to a response while others do not. If we want each button click to trigger a response safely, we can use a synchronization

[2] `https://github.com/TryGhost/Ghost/issues/1834`

primitive like `MVar`.

A `MVar` object m can hold one value and is either empty or full. A thread that puts value in m blocks if m is full. A thread that takes value from m blocks if m is empty. If multiple `take` (or `put`) threads are blocked on m, only the first one on queue is allowed to proceed after m is filled (or emptied).

```
let m = new MVar()              // create a lock
$("#button").on("click",  _ =>
        m.put(0)               // obtain the lock
          .bind(_ => AsyncM.lift(sendRequest))
          .bind(x => m.take() // release lock
                     .fmap(_ => cb(x)))
          .start()
)
```

In the code above, m is used as a lock to ensure that `sendRequest` is called once at a time. If the button is clicked before the previous request completes, the new request will be blocked on m until the previous request releases the lock.

Like threads blocked on events, threads blocked on a `MVar` can also be cancelled. The `put` method shown below adds a canceller c to the `Progress` object p if it is blocked (i.e. the `MVar` is full) and the canceller is removed when the thread unblocks (i.e. `MVar` is emptied). The canceller would remove the thread from the list of threads pending on the `MVar` and throw an exception.

```
class MVar {
  isEmpty = true;
  pending = []; // pending put or take threads

  // put :: MVar a -> a -> AsyncM ()
  put = x => new AsyncM(p => new Promise(
   (resolve, reject) => {
    if (!this.isEmpty) { // block if not empty
      let k = _ => { // 'put' continuation
        p.removeCanceller(c)
        this._put(x)
        setTimeout(resolve, 0) // resume later
      }
      let c = _ => { // removes pending thread
        this.pending =
             this.pending.filter(t => t != k)
        reject("interrupted"); // raise exception
      }
      p.addCanceller(c) // enable cancellation
      this.pending.push(k)
    }
    else { // put 'x' and continue if empty
```

```
23        this._put(x)
24        resolve()
25      }
26    }))
27    // put 'x' in MVar when it is empty
28    _put = x => {
29      this.isEmpty = false
30      this.value = x
31
32      // wake up a pending 'take' thread
33      if (this.pending.length > 0)
34        this.pending.shift()()
35    }
36
37    // the 'take' method is similar
38  }
```

The `take` method (details omitted) registers a canceller (identical to that of the `put` method) with the `Progress` p if it is blocked (i.e. the `MVar` is empty). This canceller will be removed when the `take` thread unblocks.

Like other synchronization mechanisms, our `MVar` is susceptible to deadlocks. For example, a `take` thread blocked on an empty `MVar` m is in a deadlock state if it also holds locks that prevent other threads from putting data in m. However, since JavaScript is not preemptive, it is less likely to enter a deadlock state due to non-determinism than a language with preemptive scheduling.

It may be possible to simulate priority-based scheduling by assigning a priority level to each thread when it is started. `MVar` could be modified so that it wakes up the pending threads based on their priorities. A thread can then yield to other threads by blocking itself on the modified `MVar`.

3.4 Operational Semantics

In this section, we formalize our design by giving an operational semantics in the style of Concurrent Haskell. This semantics includes two sets of rules: asynchronous rules for the reduction of `AsyncM`, which encodes thread computation and is possibly blocking, and synchronous rules for other non-blocking computation.

In Figure 4.4, we define terms and values. The symbol V ranges over values such as constants, thread ID, `MVar`s, functions, and `AsyncM` values.

f	$::=$	x => M	Functions
A	$::=$		AsyncM values
	$\vert$	$\mathsf{pure}(V)$	return value
	$\vert$	$\mathsf{throw}(c)$	throw exception
	$\vert$	$\mathsf{lift}(f)$	asynchronous action
	$\vert$	$A.\mathsf{bind}(f)$	monadic bind
	$\vert$	$A.\mathsf{catch}(f)$	handle exception
	$\vert$	$A.\mathsf{fork}()$	fork a child thread
	$\vert$	$m.\mathsf{put}(V)$	put value in MVar
	$\vert$	$m.\mathsf{take}()$	take value from Mvar
V	$::=$		Value
	$\vert$	c	constant
	$\vert$	undef	undefined value
	$\vert$	t	thread ID (progress)
	$\vert$	m	MVar
	$\vert$	f	
	$\vert$	A	
M, N	$::=$		Terms
	$\vert$	V	
	$\vert$	x	variable
	$\vert$	$M.\mathsf{start}()$	start a thread
	$\vert$	$M.\mathsf{cancel}()$	cancel a thread
	$\vert$	$M.\mathsf{pause}()$	pause a thread
	$\vert$	$M.\mathsf{resume}()$	resume a thread
	$\vert$	new `MVar`	allocate `MVar`
	$\vert$	$M\ (N)$	call
	$\vert$	$M\ ?\ N_1 : N_2$	branch
	$\vert$	$\ldots$	
t	$::=$		Thread ID
	$\vert$	p	root progress
	$\vert$	$t \cdot p$	child progress
u	$::=$		Main or thread ID
	$\vert$	ϵ	ID of main
	$\vert$	t	

Figure 3.2: The syntax of `AsyncM`, values, and terms

P, Q, R	$::=$		States
	$\mid$	$(\!\|M\|\!)_u$	live thread with ID u
	$\mid$	$(\!\|M\|\!)_t^{\bullet}$	stuck thread
	$\mid$	$(\!\|M\|\!)_t^{\circ}$	ready thread
	$\mid$	$(\!\|\|\!)_u$	completed thread
	$\mid$	$\langle\rangle_m$	empty `MVar` named m
	$\mid$	$\langle V\rangle_m$	`MVar` m filled with V
	$\mid$	$[\![t \lightning \mathsf{cancel}]\!]$	t is cancelled
	$\mid$	$[\![t/t_i \lightning \mathsf{pause}]\!]$	t_i is paused by t
	$\mid$	$[\![t/t_i \lightning \mathsf{resume}]\!]$	t_i is resumed by t
	$\mid$	$\nu x.P$	restriction
	$\mid$	$P \mid Q$	parallel composition

Figure 3.3: The syntax of program states

The symbol A ranges over `AsyncM` which includes primitives such as $\mathsf{pure}(V)$ that returns value V, $\mathsf{throw}(c)$ that throws an error c, and $\mathsf{lift}(f)$ that runs asynchronous function f and waits for its results. `AsyncM` also includes combinators: $A.\mathsf{bind}(f)$ that passes the value of A to f, $A.\mathsf{catch}(f)$ that catches the error of A with f, and $A.\mathsf{fork}()$ that runs A in a child thread.

The symbols M and N range over terms, which include values, function call, branch, new `MVar`, and the term to start/cancel/pause/resume a thread. We omit other terms such as $M.\mathsf{bind}(f)$, which should be reduced to the value $A.\mathsf{bind}(f)$ before being reduced as a monad.

The symbol t ranges over thread ID, which is either a root `Progress` p or a child `Progress` $t \cdot p$ with t as the parent. For the main program, we use ϵ to denote its ID.

3.4.1 Program Transitions

We define our semantics by describing the transitions between program states. A program state (Figure 3.3) is a parallel composition of threads, `MVar`s, and the cancel/-pause/resume actions on threads.

A thread is either alive $(\!|M|\!)_u$, stuck $(\!|M|\!)_t^{\bullet}$, or ready $(\!|M|\!)_t^{\circ}$, where the subscript is the

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & \text{(Comm)} \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \text{(Assoc)} \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P & \text{(Swap)} \\
(\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q),\ x \notin fn(Q) & \text{(Extrude)} \\
\nu x.P & \equiv & \nu y.P[y/x],\ \ x \notin fn(P) & \text{(Alpha)}
\end{array}
$$

Figure 3.4: Structural congruence

$$
\mathsf{live}((\!|M|\!)_u) \quad \frac{\mathsf{live}(P)}{\mathsf{live}(\nu x.P)} \quad \frac{\mathsf{live}(P)}{\mathsf{live}(P \mid Q)} \quad \frac{\mathsf{live}(Q)}{\mathsf{live}(P \mid Q)}
$$

$$
\frac{P \xrightarrow{\alpha} Q \quad \neg\mathsf{live}(R)}{P \mid R \xrightarrow{\alpha} Q \mid R} \text{(Par)} \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \text{(Nu)}
$$

$$
\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \text{(Equiv)}
$$

Figure 3.5: Structural transitions

thread ID. A live thread is currently running, the stuck threads are waiting for events or blocked on `MVar`s, and a ready thread will run when there is no other live thread. The initial program state is the main thread $(\!|M|\!)_\epsilon$.

A program state can transition to the next state with or without a label, which is written as: $P \xrightarrow{\alpha} Q$. The label, if present, represents an asynchronous event c received by the JavaScript event loop: $P \xrightarrow{?c} Q$.

The transitions of the program states are supported by the equivalence relation $\equiv$ defined in Figure 3.4 – identical to the one in Concurrent Haskell [33, 45].

The structural transitions of program states are defined in Figure 3.5. Since JavaScript is not preemptive, there can be at most one live thread at any time. To model this behavior, we place a restriction in Rule (Par) so that the transition from P to Q can cause transition from $P \mid R$ to $Q \mid R$ only if R does not contain live threads. In other words, while a live thread is running, no other threads can become alive.

$$
\begin{array}{rcll}
\llparenthesis \mathbb{E}[\mathsf{pure}(V).\mathsf{bind}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[f(V)] \rrparenthesis_t & \text{(Bind)} \\
\llparenthesis \mathbb{E}[\mathsf{pure}(V).\mathsf{catch}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{pure}(V)] \rrparenthesis_t & \text{(Propagate-Value)} \\
\llparenthesis \mathbb{E}[\mathsf{throw}(c).\mathsf{catch}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[f(c)] \rrparenthesis_t & \text{(Catch)} \\
\llparenthesis \mathbb{E}[\mathsf{throw}(c).\mathsf{bind}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{throw}(c)] \rrparenthesis_t & \text{(Propagate-Error)} \\
\llparenthesis \mathbb{E}[A.\mathsf{fork}()] \rrparenthesis_t & \longrightarrow & \nu p.(\llparenthesis A \rrparenthesis^{\circ}_{t \cdot p} \mid \llparenthesis \mathbb{E}[\mathsf{pure}(t \cdot p)] \rrparenthesis_t), & \text{(Fork)} \\
 & & \quad p \notin fn(\mathbb{E}, A) & \\
\llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis^{\bullet}_t & \text{(Stuck-Async)} \\
\llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis^{\bullet}_t & \xrightarrow{?c} & \llparenthesis \mathbb{E}[\mathsf{pure}(c)] \rrparenthesis_t & \text{(Async)} \\
\langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis^{b}_t & \longrightarrow & \langle V \rangle_m \mid \llparenthesis \mathbb{E}[\mathsf{pure}(\mathsf{undef})] \rrparenthesis_t & \text{(Put-MVar)} \\
\langle V \rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis^{b}_t & \longrightarrow & \langle\rangle_m \mid \llparenthesis \mathbb{E}[\mathsf{pure}(V)] \rrparenthesis_t & \text{(Take-MVar)} \\
\langle V' \rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis_t & \longrightarrow & \langle V' \rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis^{\bullet}_t & \text{(Stuck-Put-MVar)} \\
\langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis_t & \longrightarrow & \langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis^{\bullet}_t & \text{(Stuck-Take-MVar)}
\end{array}
$$

Figure 3.6: Transition rules for AsyncM values

3.4.2 Transition Rules

This section explains the transition rules for `AsyncM` values (Figure 3.6), the rules for terms (Figure 3.7), and the rules for the actions on threads (Figure 3.8). The transition rules for `AsyncM` values describe the thread computation, which takes place within an evaluation context $\mathbb{E}$ defined below.

$$\mathbb{E} \quad ::= \quad [\cdot] \mid \mathbb{E}.\mathsf{bind}(f) \mid \mathbb{E}.\mathsf{catch}(f)$$

Bind and Catch Rule (Bind) describes how a value is passed to the sequenced function. Rule (Catch) describes how an error is caught by a handler. The two propagate rules describe how values and errors are propagated through catch and bind.

Fork Rule (Fork) says that a child thread is forked with an ID that is the child of the current ID. The child thread starts in the ready state (denoted by the superscript $\circ$) so that the parent thread can continue to run.

Async Rules (Stuck-Async) and (Async) describe how the expression $\mathsf{lift}(f)$ runs the asynchronous function f and waits for its result c as an event. $\mathsf{lift}(f)$ first transitions to a stuck state denoted by the superscript $\bullet$. After receiving an event, the stuck thread $(\!|\mathbb{E}[\mathsf{lift}(f)]|\!)_t^\bullet$ transitions to a state that returns the event value c.

MVar Rule (Put-MVar) says the thread $(\!|\mathbb{E}[m.\mathsf{put}(V)]|\!)_t^b$ fills an empty MVar $\langle\rangle_m$ with its value V, where the superscript b means that the thread is either alive or stuck. If m has value, then by Rule (Stuck-Put-MVar), $m.\mathsf{put}(V)$ transitions to a stuck state. Rules for $m.\mathsf{take}()$ are similar.

Terms The transitions of terms (shown in Figure 3.7) take place within the context $\mathbb{F}$ defined below.

$$
\begin{array}{rcl}
\mathbb{F} & ::= & [\cdot] \\
& | & \mathbb{F}\ (M) \\
& | & f\ (\mathbb{F}) \\
& | & \mathbb{F}\ ?\ M_1 : M_2 \\
& | & \mathbb{F}.\mathsf{start}() \\
& | & \mathbb{F}.\mathsf{fork}() \\
& | & \mathsf{pure}(\mathbb{F}) \\
& | & \mathbb{F}.\mathsf{bind}(f) \\
& | & \mathbb{F}.\mathsf{catch}(f) \\
& | & \mathbb{F}.\mathsf{put}(M) \\
& | & m.\mathsf{put}(\mathbb{F}) \\
& | & \mathbb{F}.\mathsf{take}() \\
& | & \mathbb{F}.\mathsf{cancel}() \\
& | & \mathbb{F}.\mathsf{pause}() \\
& | & \mathbb{F}.\mathsf{resume}()
\end{array}
$$

$$
\begin{array}{rcll}
(\!|\mathbb{F}[A.\mathsf{start}()]|\!)_u & \longrightarrow & \nu p.((\!|A|\!)_p^\circ \mid (\!|\mathbb{F}[p]|\!)_u), \quad p \notin fn(\mathbb{F}, A) & \text{(Start)} \\
(\!|\mathbb{F}[\mathsf{new}\ MVar]|\!)_u & \longrightarrow & \nu m.(\langle\rangle_m \mid (\!|\mathbb{F}[m]|\!)_u), \quad m \notin fn(\mathbb{F}) & \text{(New-MVar)} \\
(\!|\mathbb{F}[(x => M)(V)]|\!)_u, & \longrightarrow & (\!|\mathbb{F}[M[V/x]]|\!)_u & \text{(Call)} \\
(\!|\mathbb{F}[true\ ?\ M_1 : M_2]|\!)_u, & \longrightarrow & (\!|\mathbb{F}[M_1]|\!)_u & \text{(True)} \\
(\!|\mathbb{F}[false\ ?\ M_1 : M_2]|\!)_u, & \longrightarrow & (\!|\mathbb{F}[M_2]|\!)_u & \text{(False)} \\
(\!|A|\!)_t^\circ & \longrightarrow & (\!|A|\!)_t & \text{(Run)} \\
(\!|\mathsf{pure}(V)|\!)_t & \longrightarrow & (\!||\!)_t & \text{(Value-End)} \\
(\!|\mathsf{throw}(c)|\!)_t & \longrightarrow & (\!||\!)_t & \text{(Error-End)} \\
(\!|V|\!)_\epsilon & \longrightarrow & (\!||\!)_\epsilon & \text{(Main-End)} \\
(\!||\!)_u \mid P & \longrightarrow & P & \text{(GC)}
\end{array}
$$

Figure 3.7: Transition rules for terms, run thread, and termination

The terms include expressions like new `MVar`, function call, and branch, whose transitions are non-blocking. That is, a live thread will continue to be alive after the transition. Rule (Start) describes how $A.\mathsf{start}()$ starts a new thread with a root progress as its thread ID. The new thread is also in the ready state so that the calling thread can continue to run.

Run and Termination By Rule (Run), a thread in ready state can transition to a live thread; and by Rule (Par) in Figure 3.5, this transition is allowed if there is no other live threads. This semantics is implemented by running the thread with a 0 second timeout so that a ready thread is scheduled to run by the JavaScript event loop after other threads complete or become stuck. Figure 3.7 also includes the rules for thread termination, which states that a forked or started thread terminates if it returns a value or throws an error while the main thread terminates when it reduces to a value. Rule (GC) says that a terminated thread is removed from the program.

Thread Actions Figure 3.8 defines the transition rules for terms that cancel, pause, and resume threads. Rule (Cancel) states that the term $t.\mathsf{cancel}()$ reduces an undefined value while producing actions to cancel threads with ID t_i that satisfies the relation $\mathsf{prefix}(t, t_i)$. The actions are denoted by the set $\{[\![t_i \lightning \mathsf{cancel}]\!]\}_{\mathsf{prefix}(t,t_i)}$. The relation

$$
\begin{array}{rcll}
(\!|\mathbb{F}[t.\mathsf{cancel}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t_i \lightning \mathsf{cancel}]\!]\}_{\mathsf{prefix}(t,t_i)} & \text{(Cancel)} \\
(\!|\mathbb{F}[t.\mathsf{pause}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t/t_i \lightning \mathsf{pause}]\!]\}_{\mathsf{prefix}(t,t_i)} & \text{(Pause)} \\
(\!|\mathbb{F}[t.\mathsf{resume}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t/t_i \lightning \mathsf{resume}]\!]\}_{\mathsf{prefix}(t,t_i)} & \text{(Resume)}
\end{array}
$$

$$
\begin{array}{rcll}
[\![t \lightning \mathsf{cancel}]\!] \mid (\!|\mathbb{E}[V]|\!)_t^{\bullet} & \longrightarrow & (\!|\mathbb{E}[\mathsf{throw}(\text{“}\mathit{interrupted}\text{”})]|\!)_t & \text{(Cancel-Stuck)} \\
[\![t \lightning \mathsf{cancel}]\!] \mid (\!|\mathbb{E}[\mathsf{lift}(f)]|\!)_t & \longrightarrow & (\!|\mathbb{E}[\mathsf{throw}(\text{“}\mathit{interrupted}\text{”})]|\!)_t & \text{(Cancel-Async)} \\
[\![t/t_i \lightning \mathsf{pause}]\!] \mid (\!|\mathbb{E}[\mathsf{lift}(f)]|\!)_{t_i}^{\bullet} & \xrightarrow{?c} & (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)_{t_i}^{\bullet,t} & \text{(Pause-Stuck)} \\
[\![t/t_i \lightning \mathsf{resume}]\!] \mid (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)_{t_i}^{\bullet,t} & \longrightarrow & (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)_{t_i} & \text{(Resume-Paused)}
\end{array}
$$

Figure 3.8: Transition rules for cancellation, pause, and resumption. Rule (Cancel-Stuck) has higher priority than Rule (Async).

$\mathsf{prefix}(t, t_i)$ is defined below, which means that t either equals t_i or is an ancestor of t_i.

$$
\mathsf{prefix}(t,t) \qquad \frac{\mathsf{prefix}(t,u)}{\mathsf{prefix}(t,u \cdot p)}
$$

In other words, $t.\mathsf{cancel}()$ will cancel any threads running with t or the children of t as thread IDs. By Rule (Cancel-Stuck), the cancel action will cause a stuck thread to throw an exception. This rule takes precedence over Rule (Async) so that a stuck thread, if cancelled, will always terminate. By Rule (Cancel-Async), the cancel action will cause a live thread with an asynchronous operation to throw an exception. The rule is applicable when a thread is cancelling itself.

Rule (Pause) describes how $t.\mathsf{pause}()$ reduces to the undef value in its context and produces a set of the pause actions $\{[\![t/t_i \lightning \mathsf{pause}]\!]\}_{\mathsf{prefix}(t,t_i)}$. By Rule (Pause-Stuck), each of the pause actions can pause a stuck thread when it unblocks, which transitions to another stuck thread $(\!|\mathbb{E}[\mathsf{pure}(c)]|\!)_{t_i}^{\bullet,t}$. The superscript t indicates that this thread can only be resumed by the call $t.\mathsf{resume}()$ according to the rules (Resume) and (Resume-Paused).

3.5 Additional Constructs

Race and All We provide a `race` combinator to conduct a race among a list of threads and an `all` combinator to run a list of threads concurrently and to wait for their results. The composed threads can be cancelled altogether without any additional logic due to

our cancellation mechanism.

If we race a list of threads, the losing threads will also be cancelled. The `race` method below first allocates a child `Progress` $p1$ and then runs the component threads with $p1$. When one of the threads wins the race, the call $p1$.cancel() terminates the rest of the threads.

```
// race :: [AsyncM a] -> AsyncM a
static race = lst => new AsyncM (p => {
  let p1 = new Progress(p);
  return Promise.race(lst.map(a => a.run(p1)))
    .finally(_ => {
        p1.cancel(); // cancel losing threads
        p1.unlink(); // remove p to p1 link
    });
})

// race :: [AsyncM a] -> AsyncM [a]
static all = lst => new AsyncM (p =>
    Promise.all(lst.map(a => a.run(p)))
)
```

Channel We can use `MVar` to implement other primitives. For example, we have implemented a buffered channel using a `MVar` to hold the pending readers when the channel is empty and to hold the pending writers when the channel is full. Any threads that are blocked on a channel can be cancelled because they are blocked on its `MVar`.

Safe points All asynchronous operations defined with *lift* are potential points of cancellation. An alternative is to run these operations using `Promises` instead and to poll the cancellation status at specific check points. The *ifAlive* method below can be composed with other `AsyncMs` as a safe point for cancellation.

```
static ifAlive = new AsyncM (async p => {
  if (! p.cancelled) return;
  else throw("interrupted");
});
```

Control flow Implementing control flow with `AsyncM` is more awkward than using *async*/*await*. For simple cases such as infinite loops, we can use methods like *loop* below.

```
// e.g. a.loop() runs forever unless cancelled
class AsyncM {
```

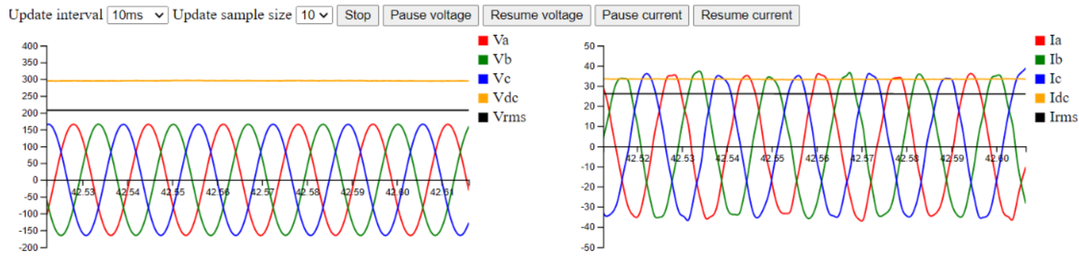

Figure 3.9: A data streaming application, where V_a, V_b, V_c, V_{dc}, V_{rms} are voltages and I_a, I_b, I_c, I_{dc}, I_{rms} are currents.

```
  loop = _ => new AsyncM (async p => {
      while (true) { await this.run(p) }
  })
}
```

For more complex control flow, it is better to compose the `AsyncM`s indirectly by first converting them to `Promise`s, which can be composed with `await`. The composed `Promise`s can then be converted back to an `AsyncM` object using an `async` function with a `Progress` parameter.

3.6 Evaluation

Our model is implemented with 400 lines of JavaScript. To evaluate its usability, we used it in two applications.

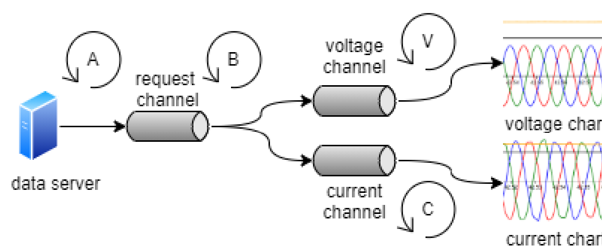

Figure 3.10: The architecture of the data streaming application in Figure 3.9, where A, B, C, and V are threads.

Data streaming The first application (Figure 3.9) streams the voltage and current signals (sampled at 1KHz) from a remote source and visualizes them in two real-time charts. The voltage and current charts can be paused/resumed separately and can be stopped simultaneously.

To hide network latency, this application uses a thread (Thread A in Figure 3.10) to fetch data in batches at a regular interval and to push each data request to the request channel. Thread B reads a data request from the request channel, waits for it to complete, and writes its result to the voltage channel and the current channel using the `AsyncM.all` method. Thread V retrieves a batch of data from the voltage channel and displays the voltage data incrementally in a chart (e.g. by updating the chart with 10 new samples every 10 ms). Thread C performs similar actions for the current data.

All 4 threads run independently and communicate through the channels, which allows the speeds of data transmission and chart rendering to match. For example, if data transmission is faster than chart rendering, one or both of the data channels will become full, causing Thread B to block and the request channel to become full, which blocks Thread A. Also, we can pause the voltage (or the current) chart by pausing Thread V (or Thread C). Note that if Thread V is paused, the voltage channel will become full if Thread B keeps writing to it, which eventually blocks Thread B from writing to the current channel even if it is not full. Thus, if a chart is paused, Thread B may be cancelled and restarted so that it only writes data to the channel of the running chart.

Our thread abstractions help reduce the complexity of this application, where the channels manage thread synchronization, hierarchical thread cancellation allows all threads be cancelled as a group, and the ability to pause and resume threads makes it easy to pause and resume chart animations.

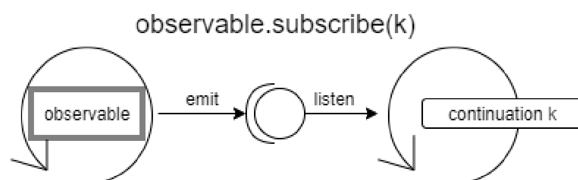

Figure 3.11: The subscription of a RxJS `Observable` using 2 threads (arrow circles) and an emitter (middle circle).

RxJS The second application[3] is a subset of RxJS interface implemented with our thread library. RxJS[4] is a popular JavaScript library for reactive programming, which rep-

[3] `https://github.com/tianzhao/rxjs`
[4] `http://reactivex.io`

resents event streams as dataflow graphs. The core abstraction of RxJS is the `Observable` interface, which emits events to its observers connected through the `Subscription` objects. Despite its popularity, RxJS is difficult to debug [1] since, unlike the functional designs of reactive programming [15, 56], RxJS is imperative and its control flow does not correspond to its dataflow, which is hard to inspect with debuggers.

In our design, each `Observable` is implemented with a function that takes an `emitter` object and returns a thread that emits events to the emitter. As shown in Figure 3.11, when an `Observable` is subscribed, two threads are started: one is the `Observable` thread that emits events to an `emitter`, while the other thread calls a continuation k with the events received from the `emitter`. When the `Subscription` is unsubscribed, both threads are cancelled through a shared thread ID. Within an `Observable`, if an inner `Observable` is subscribed, its thread is the child of the parent `Observable`'s thread. Therefore, when an `Observable` is unsubscribed, its inner `Observable`s are unsubscribed automatically due to the hierarchical cancellation of our thread model.

Our version of RxJS is easier to debug since the dataflow graph of a RxJS expression is embedded in the returned `Subscription` object, which contains an `emitter`, a thread ID, and the references to the `Subscription`s to the inner `Observable`s. Users can debug a RxJS program by navigating the `Subscription` object to examine the dataflow graph, inspecting its emitter for the past events, and checking its thread ID for the status of the subscription threads.

Runtime overhead The most significant overhead in our design is due to the `AsyncM.lift` method, which allocates continuations and adds/removes cancellers. The table below shows in milliseconds the amount of time it takes to run a trivial synchronous and asynchronous computation (0s timeout) over a number of iterations. For synchronous computation, the overhead of `lift` is significant compared to `Promise`-based implementation. However, for asynchronous computation, the overhead due to `lift` is less noticeable.

iterations	Synchronous		Asynchronous	
	`AsyncM`	`Promise`	`AsyncM`	`Promise`
100	1.12	0.16	158.38	147.61
500	5.16	0.33	753.66	721.82
1000	5.42	0.55	1479.62	1458.18
5000	11.76	2.47	7314.38	7292.97
10000	17.99	4.92	14628.99	14590.47

Table 3.1: Measured baseline runtime overhead

3.7 Related Work

Promises This work enhances JavaScript's `Promise`s [14] by providing a thread-like concurrency model with cooperative cancellation. While `Promise`s (together with `async` and `await`) allow us to write asynchronous programs in JavaScript in sequential style, the execution order of the programs is not always clear. JavaScript's event loop maintains separate queues for tasks (e.g. timeouts) and micro-tasks (e.g. `Promise`s). `Promise`s are executed in the order in which they are added to the micro-task queue. For example, the execution of two `Promise` chains can interleave even if they are all synchronous. Even if we call an `async` function without `await`ing for its result, its synchronous portion still runs first. Also, the `resolve` and `reject` functions of a `Promise` can be saved and invoked later by some other code, which can cause further interleaving of `Promise` execution. Consequently, a `Promise` chain may not have exclusive access to shared states in between asynchronous operations, which can lead to subtle race conditions. By wrapping `Promise`s inside `AsyncM`s, we ensure that the threads must be explicitly started, the synchronous part of a thread has exclusive access to the shared states, and the locks such as `MVar`s can be used to protect shared states between threads.

`Promise`s also do not have builtin methods for cancellation but hand-crafted solution can easily forget pending callbacks or `Promise`s, which leads to unintended side effects. Our proposal is intended to complement `Promise`s by providing a more consistent way to terminate unused computation.

Coroutine The exact definition for coroutines varies between languages, but they are generally understood as non-preemptive thread-like concepts [52, 25, 18]. They are non-preemptive in the way that control of execution can be suspended and transferred explicitly. A JavaScript generator is a limited form of coroutine that allows computation to switch back and forth between a generator and its caller through the `yield` mechanism. Together with `Promise`s, generators can be used to compose asynchronous computation using `for..of` or `for await..of` loops. Python's *asyncio* library is an asynchronous framework that supports non-preemptive scheduling and also cancellation [20]. It provides low-level API that allows scheduling work from a different OS thread, in which case thread-safety needs to be considered.

Concurrency monad Claessen [10] described the use of continuation monad for concurrency in Haskell. The design permits a limited form of concurrency on monadic computations without adding primitives to the language. The concurrency monad builds on the continuation monad, where computations can be evaluated concurrently by interleaving evaluation of lifted operations. By implementing it as a monadic transformer, the existing monads can be extended with concurrency operations and can be entirely defined as a library without introducing new language primitives. This idea was later adopted by Li and Zidancwic [28] in their scalable network services that provide type-safe abstractions for both events and threads. They use a continuation monad to build traces that are scheduled by the event loops.

Asynchronous Exception Asynchronous exception is introduced in Concurrent Haskell [33], which allows one thread to throw an asynchronous exception to another thread. The asynchronous exception raises a synchronous exception in the receiving thread, which can be handled or cause the thread to terminate. Since Concurrent Haskell has preemptive scheduling, the asynchronous exception can interrupt the receiving thread at any point. To protect critical regions, Concurrent Haskell includes block and unblock primitives to mask the regions that cannot be interrupted. Despite this, threads blocked on `MVar` or IO can always be interrupted to reduce the chance of deadlock.

We adopt a similar strategy by throwing interrupt exception to target threads. However, since JavaScript is not preemptive, the interrupt exceptions are only received at the locations where the threads are blocked or polling the thread status. To disable the interrupt exceptions, we can run an `AsyncM` with a new `Progress` using the *block* method below.

```
class AsyncM {
  block = _ =>
    new AsyncM(_ => this.run(new Progress())) }
```

Our operational semantics is also modeled after that of Concurrent Haskell [33], where the reduction of processes is based on the chemical abstract machine [6, 8].

Cooperative Cancellation AC [22] introduced language constructs to insert code blocks for asynchronous IO in native languages like C/C++. Each of these blocks is delimited by the `do..finish` keywords. Within a block, the keywords `async` and `cancel` can be used to start an asynchronous operation and to cancel it, respectively. The `cancel` keyword can be used with a label to indicate which `async` operation to cancel, but it must be used within the enclosing `do..finish` block. Cancelling an `async` operation will propagate cancellation into any nested `async` branches recursively. The execution of a `do..finish` block does not complete until all `async` operations within it are finished or cancelled.

.NET uses cancellation tokens for cooperative cancellation, where the concurrent tasks use their cancellation tokens to decide how to handle cancellation requests. The design distinguishes the cancellation source, which is used for requesting cancellation, from the tokens, which are used for polling cancellation status, registering cancellation callbacks, and enabling blocked tasks to wait for cancel events. A task can react to multiple cancellation tokens by linking them in a new cancellation source. *F#* [43, 53] provides an asynchronous programming model through CPS transformation on its `async` expressions. The language supports cancellation by implicitly threading cancellation tokens (derived from a cancellation source) through the program execution. Cancellation tokens are checked at IO primitives and various control flow constructs.

Our cancellation mechanism is similar to the cancellation tokens in that a thread reacts to the cancellation requests at some program points. However, we integrate cancellation into a thread model, where a thread ID is both the cancellation source and token. The hierarchical structure of threads allows a child thread to react to a cancellation request to its parent thread without explicitly linking cancellation tokens.

3.8 Summary

We needed an abstraction to represent asynchronous computation since JavaScript does not have a standard way to handle cancellation. So we designed an implementation in Haskell and leverage its powerful type system to allows us use it as a model to reimplement in other languages such as JavaScript and Python that are less type-safe. But writing directly using our JavaScript implementation, and similarly in Python, can still be cumbersome because the lack of static type checking, and can be difficult to debug problems at runtime.

We have presented a thread-based concurrency model for JavaScript that can cancel, pause, and resume threads. The thread abstraction makes it easier to reason about asynchronous programs while synchronization primitives can protect shared resources. These advantages help reduce the occurrences of race conditions. The ability to cancel threads helps prevent the side effects of unwanted computation. The ability to pause and resume threads may be used for debugging concurrency errors in a browser environment and providing a simple way to suspend computation such as animation.

This design is implemented as a JavaScript library and since each `AsyncM` wraps a `Promise` function, it is compatible with the `Promise` abstraction and can be integrated with other types of programs using `async` and `await` to implement complex logic. The overhead of thread abstractions and cancellation is not significant relative to the computation time of the asynchronous operations that they support.

Chapter 4

Semantics and Debugging

In previous chapters, we have implemented abstractions to better handle concurrent event-based programs. In this chapter, we focus on RxJS, a popular JavaScript library for writing reactive programs. We will present an implementation of the RxJS abstraction using our own concurrency abstraction as the foundation, and provide a formal model for the RxJS programs. Our motivation for this work is to provide a precise semantics for the RxJS programs in an effort to make testing and debugging easier.

4.1 Introduction

RxJS is a reactive programming library for JavaScript, which has been integrated in frameworks such as Angular and React to handle UI and other asynchronous events. Despite its success, debugging of RxJS programs is still difficult. A recent study [1] examined the challenges in debugging RxJS programs due to the disparity between the declarative interface for defining dataflow logic [2] and the imperative implementation based on the Observer Pattern [21]. As pointed out in [1], traditional debugger does not offer much help in identifying the cause of the bugs since the dataflow logic of RxJS programs is not directly reflected in the control flow logic represented by the call stacks at the break points. A RxJS program constructs and mutates dataflow graphs and pushes events through the graphs to perform pure computation and to produce side effects. The dataflow graphs are implicit constructs that are not readily accessible to users during the debugging phase. To find the cause of an error, one needs to examine the states of the dataflow graphs, which include the graph shapes and the events at each graph node. Programmers often resort to indirect approaches such as printing event traces and drawing dataflow graphs by instrumenting the source program or using debugging tools

such as rxjs-spy, rxviz, and rxjs-playground. While these methods are useful, they are informal, require manual inspection, and does not scale to larger programs.

In this chapter, we present a formal semantics for a selected subset of RxJS operators to provide a precise definition of their meaning. Using the semantics as a model, we can define some debugging related applications to help discover problems in RxJS programs. The semantics models RxJS runtime with a heap that contains the subscription graph, where each node is a subscription, and a queue that contains the external and internal events. The reduction of a RxJS program alternates between graph construction phase and reactive phase where events are propagated through the graph. Using the semantics as a model, we can define a representation of stack trace specific to the RxJS programs, a set of rules to check subscription states, and a runtime invariant of the subscription graph to ensure that the error for each event source can be uniquely identified.

Based on this semantics, we have implemented a large subset of RxJS using a thread-like abstraction using concurrency monad [57, 56], where each thread can be cancelled via its thread ID. We use this thread abstraction to implement the subscription to an observable so that unsubscribe operation is the same as thread cancellation. When an observable is subscribed, it returns a subscription that holds an emitter, a thread ID, and the subscriptions to the source and child observables if any. Through the subscription object, users can cancel the subscription or use it to navigate the dataflow graphs, check the state of each node, and examine the previous events.

In this chapter, we make the following contributions:

1. We give a motivational example in Section 4.2 to demonstrate how semantics can help identify problems in a RxJS program.

2. In Section 3.4, we present a formal model of RxJS by defining an operational semantics for a selected subset of RxJS operators. The semantics provides a simplified model for reasoning about the expected and unexpected behavior of a RxJS program.

3. In Section 4.6.1, we provide some example applications of the semantics for debug-

ging purposes, which include stack-trace suitable for reactive programs, unexpected states of observables, and checking invariant on subscription graph,

4. We describe an implementation of RxJS library that conforms to this semantics in Section 4.5.

5. The related works are discussed in Section 4.7.

4.2 Subscription as Dataflow Graph

In this section, we use the following example to motivate the need of a formal semantics for RxJS. This example is a simple type-ahead client, which sends a query request each time the user types a character in a text box. If a previous query does not complete before a new request, then old query is canceled. If a query is answered on time, then the result is displayed.

```
1 let subscription =
2   fromEvent('#type-ahead', 'keyup') // input
3   .map(e => e.target.value)     // input text
4   .switchMap(x => from(query(x)))    // query
5   .subscribe(display);     // display results
```

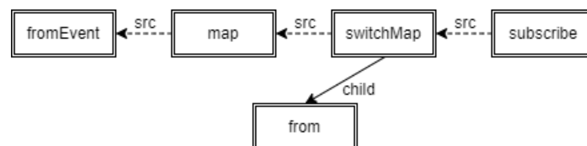

Figure 4.1: The subscription graph of the type-ahead example.

The structure of the subscription graph is shown in Figure 4.1, where the arrows point to the source or child of each subscription. The subscription graph mirrors the dataflow, from which we can also examine the past events emitted by each observable. There are three types of events emitted by an observable.

1. `next(v)`, which holds event value v;

2. `end`, which indicates that the observable completes;

3. `error`, which signals an exception and it recursively propagates to the subscribers until it is either caught by a `catchError` observable or by the root subscriber. Observable that throws an error is cancelled.

Using the subscription graph, users can find the source of errors more easily. For example, the type-ahead server returns an error for certain inputs. The user wants to catch the error by inserting a `catchError` operator so that the error does not stop the client.

```
1 let subscription =
2   fromEvent('#type-ahead', 'keyup')
3   .map(e => e.target.value)
4   .switchMap(x => from(query(x)))
5   .catchError(_ => of('error'))  // A bug
6   .subscribe(display);
```

The code above shows the initial attempt of catching error but it fails to keep the type-ahead client running – it still terminates if an error is emitted. The cause of the problem is that the `catchError` protects the `switchMap`, and when it catches an error, it must terminate the entire `switchMap` instead of the `from` observable.

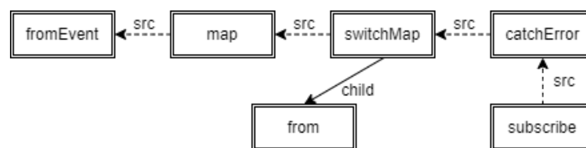

Figure 4.2: The initial attempt to catch query errors.

This relation is illustrated in Figure 4.2. Note that while it is possible to spot this problem by inspecting the source code, it is much harder to identify similar problems in larger applications. A systematic solution should allow automatic analysis of the subscription graph to detect similar problems. For example, we can require that every source observable such as `fromEvent` or `from` be uniquely protected by a `catchError` or a root subscriber. In Figure 4.2, both the `fromEvent` and `from` are guarded by the same `catchError`, which may indicate a potential problem.

We can fix the bug by placing the `catchError` within the `switchMap` operator as shown below.

```
let subscription =
  fromEvent('#type-ahead', 'keyup')
  .map(e => e.target.value)
  .switchMap(x =>
       from(query(x))
       .catchError(_ => of('error'))
   ) // catch error and emit a string event
  .subscribe(display);
```

The resulting subscription graph is shown in Figure 4.3, where `fromEvent` is protected by the root subscriber while `from` is protected by `catchError`.

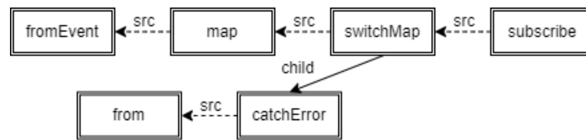

Figure 4.3: The correct handling of query errors.

While the subscription graph is static in this example, in general, a subscription graph may change at runtime. This means that the checking of the subscription graph should be done at runtime when the graph is updated. However, to avoid unnecessary overhead, the runtime checking should be limited to graph updates that may cause a violation of the invariant. To understand how to check this invariant, in the next section, we present a formal semantics to a selected set of RxJS operators.

4.3 RxJS

RxJS is a reactive programming library for JavaScript. It is a popular library used for web applications and it is integrated in frameworks such as Angular and React to handle UI and other asynchronous events. It provides numerous operators to create, combine, transform, and filter discrete events and to handle errors. These operators may be stateful and have side effects, which makes it difficult to understand the precise meaning of the resulting computation. While the interface is functional, it differs from the classical FRP that it only operates on discrete events.

Studies have found that users often have to resort to inserting print statements or using debugging tools such as rxjs-spy, rxviz, and rxjs-playground. While these methods are

$$
\begin{array}{lcl}
e \in \textsf{Expression} & ::= & x \\
& | & o \\
& | & e.\texttt{sub}(f) \\
& | & e.\texttt{unsub}() \\
& | & o.\texttt{sub}() \\
& | & x = o.\texttt{share}() \\
& | & e;\ e' \\
& | & x \Rightarrow e \\
& | & e(e') \\
o \in \textsf{Observable} & ::= & x \\
& | & \texttt{fromEvent}(elm, evt) \\
& | & \texttt{combine}(o_1, o_2) \\
& | & \texttt{concat}(o_1, o_2) \\
& | & o.\texttt{every}(f) \\
& | & o.\texttt{catchError}(f) \\
& | & o.\texttt{take}(n) \\
& | & o.\texttt{map}(f) \\
& | & o.\texttt{switchMap}(f)
\end{array}
$$

Figure 4.4: The syntax of λ_{rx}, where the shaded terms are runtime entities.

useful, they are informal, require manual inspection, and do not scale to larger programs.

4.4 Operational Semantics for RxJS

In this section, we define a formal model of RxJS computation using a selected subset of RxJS operators, which we call λ_{rx}. We will first present the syntax of RxJS operators for creating, combine, transform observables. Then we will presents runtime values and the operational semantics for evaluation and event propagation.

4.4.1 Syntax

The syntax of λ_{rx} is shown in Figure 4.4, where e ranges over expressions such as variables, observables, subscribe operations $e.\texttt{sub}(f)$, unsubscribe operations $e.\texttt{unsub}()$, create shared observables, sequences, functions, and function calls. The subscribe operation $e.\texttt{sub}(f)$ starts an observable e and calls a function f with the observable events as inputs. The unsubscribe operation stops the subscription to an observable. An al-

ternative subscribe operation $o.\texttt{sub}()$ subscribes to the observable o without calling a side-effecting function on observable events and it evaluates to a reference r to the subscription on the observable o. The assignment expression $x = o.\texttt{share}()$ is used to share the observable o through the variable x.

The variable o ranges over x (for shared observables) and observable expressions, which include one or two operators from each group of RxJS operators detailed as follows.

- $\texttt{fromEvent}(elm, evt)$ is a creation operator that creates a primitive observable that emits events of type evt from a DOM element elm.
- $\texttt{combine}(o_1, o_2)$ is a combination operator that combines the latest events from o_1 and o_2.
- $\texttt{concat}(o_1, o_2)$ is a combination operator that concatenates the event streams of o_1 and o_2.
- $o.\texttt{every}(f)$ is a conditional operator that emits true if all events from o satisfy the predicate f and emits false otherwise.
- $o.\texttt{catchError}(f)$ is an error handling operator that emits next events from o but if o emits an error, then it stops o and emits events from $f()$ instead.
- $o.\texttt{share}()$ is a multicasting operator that shares the events of o with multiple subscribers and manages the lifetime of o using reference counting. When the shared observable is first subscribed, it is transformed to $\texttt{share}(o, r)$, where r points to a subscription of o.
- $o.\texttt{take}(n)$ is a filtering operator that takes up to n events from o.
- $o.\texttt{map}(f)$ is a transformation operator that for each event value v of o, it emits $f(v)$.
- $o.\texttt{switchMap}(f)$ is also a transformation operator that for each event value v of o, it emits the events of the new inner observable $f(v)$ after it stops the existing inner observable if any.

$$
\begin{array}{lcl}
s \in \textsf{Subscription} & ::= & \texttt{fromEvent}(elm, evt) \\
 & | & \texttt{combine}(r_1, r_2, v_1, v_2, n) \\
 & | & \texttt{concat}(r_1, o_2) \\
 & | & \texttt{every}(r, f) \\
 & | & \texttt{catchError}(r, f) \\
 & | & \textsf{sharing}(x) \\
 & | & \texttt{take}(r, n) \\
 & | & \texttt{map}(r, f) \\
 & | & \texttt{switchMap}(r_1, f, r_2, b) \\
 & | & \texttt{sub}(r, f) \\
v \in \textsf{Value} & ::= & n \mid o \mid \textsf{true} \mid \textsf{false} \mid \epsilon \mid x \Rightarrow e \\
h \in \textsf{Shared observable} & ::= & o.\texttt{share}() \\
 & | & \texttt{share}(o, r) \\
H \in \textsf{Heap} & ::= & \{x \mapsto h;\ \ldots; r \mapsto s;\ \ldots\} \\
Q \in \textsf{Queue} & ::= & [(t, r),\ \ldots] \\
t \in \textsf{Event} & :: & \textsf{next}(v) \\
 & | & \textsf{end} \\
 & | & \textsf{error}
\end{array}
$$

Figure 4.5: The runtime values of λ_{rx}, where x points to shared observables and r points to the subscriptions in the heap.

4.4.2 Runtime Values

A subscribe operation $o.\texttt{sub}()$ evaluates to a subscription, which is a runtime value shown in Figure 4.5.

Each observable operator has a corresponding subscription value, which may carry additional state information of the subscription. We store the subscriptions in a heap H that maps references (denoted by r) to subscriptions. Each subscription can refer to its source subscriptions through the references, which is detailed as follows.

- $\texttt{combine}(r_1, r_2, v_1, v_2, n)$ contains r_1 and r_2, which are the subscriptions to the source observables. It also contains v_1 and v_2, which are either the last events or undefined (denoted by ϵ). n is the number of source observables that have completed.

- $\texttt{concat}(r_1, o_2)$ contains the subscription r_1 and an observable expression o_2 that starts once r_1 completes.

- $\texttt{every}(r, f)$, $\texttt{catchError}(r, f)$, and $\texttt{map}(r, f)$ contain the subscription r to their

$$
\begin{array}{rcl}
E & ::= & \cdot \\
 & | & E.\mathtt{sub}(f) \\
 & | & E.\mathtt{unsub}() \\
 & | & E; e \\
 & | & E(e) \\
 & | & v(E)
\end{array}
$$

Figure 4.6: The evaluation context for expressions

source observable.

- $\mathsf{sharing}(x)$ contains x that references a shared observable $\mathtt{share}(o, r)$, where r is a subscription to o.
- $\mathtt{take}(r, n)$ contains the subscription r to its source observable and the remaining number of events n.
- $\mathtt{switchMap}(r_1, f, r_2, b)$ contains the subscriptions to the outer and inner observables r_1, r_2, and b, which is a Boolean that indicates whether the inner or outer observable has ended.

A queue Q is used to temporarily hold events that are emitted from observables but have not yet been received by the subscribers. Q holds a list of pairs (t, r), where t is an event and r refers to the subscription that emitted the event. Each event is either a $\mathsf{next}(v)$ event with value v, an error, or an end event that signals the completion of an observable. The event queue can be used to model some variations of event scheduling in RxJS.

4.4.3 Evaluation of Expressions

The evaluation rules for expressions are shown in Figure 4.7, where each rule has the form of $H, e \rightarrow H', e'$. The expressions such as assignment and unsubscribe operation are evaluated for their side effects.

The expression $o.\mathtt{sub}(f)$ is the starting point of RxJS computation, where the observable o is subscribed so that its events are used to run f for its side effects. By Rule E_{sub1},

$$H, v \to H, v \qquad E_{Val}$$

$$\frac{r \text{ is fresh} \quad H, o.\mathtt{sub}() \to H', r'}{H, o.\mathtt{sub}(f) \to H'[r \mapsto \mathtt{sub}(r', f)], r} \qquad E_{Sub1}$$

$$\frac{r \text{ is fresh} \quad H, o.\mathtt{sub}() \rightsquigarrow H', s}{H, o.\mathtt{sub}() \to H'[r \mapsto s], r} \qquad E_{Sub2}$$

$$H, x = o.\mathtt{share}() \to H[x \mapsto o.\mathtt{share}()], \epsilon \qquad E_{Assign}$$

$$\frac{H, r.\mathtt{unsub}() \rightsquigarrow H'}{H, r.\mathtt{unsub}() \to H' \backslash \{r \mapsto H'(r)\}, \epsilon} \qquad E_{Unsub}$$

$$\frac{H, e \to H', e'}{H, E[e] \to H', E[e']} \qquad E_{Cong}$$

$$H, v; e \to H, e \qquad E_{Seq}$$

$$H, (x \Rightarrow e)(v) \to H, [v/x]e \qquad E_{Call}$$

Figure 4.7: The evaluation rules for expressions

the subscribe operation evaluates to a reference r that points to $\mathtt{sub}(r', f)$, which listens on the events of r' to run f and r' references the subscription to o. By Rule E_{sub2}, $o.\mathtt{sub}()$ reduces to a fresh reference that maps to the subscription value s evaluated from $o.\mathtt{sub}()$. The evaluation of $o.\mathtt{sub}()$ to a subscription value is defined by the rules in Figure 4.8.

Subscribe Operations In λ_{rx}, each subscribe operation reduces to a reference r that is mapped to a subscription value s in the heap. The subscribers to s receives its events through r such that the subscriptions are linked as a graph through references like r.

A subscribe operation $o.\mathtt{sub}()$ may cause additional subscriptions to the observables in o. This behavior is detailed in Figure 4.8. For example, in Rule $S_{combine}$, the subscription to a '`combine`' observable leads to the subscriptions to its source observables o_1 and o_2.

Most of the RxJS operators create *cold* observables [50], each of which has a single subscriber. The group of multicast operators such as `share` create *hot* observables that can be shared by multiple subscribers. Rule S_{Share1} says that when $H(x) = o.\mathtt{share}()$ is subscribed the first time in a call $x.\mathtt{sub}()$, o is subscribed and the heap H is updated so that x is mapped to $\mathtt{share}(o, r)$, where r points to the subscription to o. By Rule S_{Share2}, each subsequent call to $x.\mathtt{sub}()$ reduces to $\mathsf{sharing}(x)$, which is a subscription value that represents shared access to $\mathtt{share}(o, r)$.

$$\frac{o = \texttt{fromEvent}(elm, evt)}{H, o.\texttt{sub}() \rightsquigarrow H, \texttt{fromEvent}(elm, evt)} \quad S_{From}$$

$$\frac{\begin{array}{c} o = \texttt{combine}(o_1, o_2) \\ H, o_1.\texttt{sub}() \rightarrow H_1, r_1 \\ H_1, o_2.\texttt{sub}() \rightarrow H_2, r_2 \end{array}}{H, o.\texttt{sub}() \rightsquigarrow H_2, \texttt{combine}(r_1, r_2, \epsilon, \epsilon, 0)} \quad S_{Combine}$$

$$\frac{\begin{array}{c} o = H, \texttt{concat}(o_1, o_2) \\ H, o_1.\texttt{sub}() \rightarrow H_1, r_1 \end{array}}{H, o.\texttt{sub}() \rightsquigarrow H_1, \texttt{concat}(r_1, o_2)} \quad S_{Concat}$$

$$\frac{o = o'.\texttt{every}(f) \quad H, o'.\texttt{sub}() \rightarrow H', r}{H, o.\texttt{sub}() \rightsquigarrow H', \texttt{every}(r, f)} \quad S_{Every}$$

$$\frac{\begin{array}{c} o = o'.\texttt{catchError}(f) \\ H, o'.\texttt{sub}() \rightarrow H', r \end{array}}{H, o.\texttt{sub}() \rightsquigarrow H', \texttt{catchError}(r, f)} \quad S_{Catch}$$

$$\frac{\begin{array}{c} H(x) = o.\texttt{share}() \\ H, o.\texttt{sub}() \rightarrow H', r \\ H'' = H'[x \mapsto \texttt{share}(o, r)] \end{array}}{H, x.\texttt{sub}() \rightsquigarrow H'', \mathsf{sharing}(x)} \quad S_{Share1}$$

$$\frac{H(x) = \texttt{share}(o, r)}{H, x.\texttt{sub}() \rightsquigarrow H, \mathsf{sharing}(x)} \quad S_{Share2}$$

$$\frac{o = o'.\texttt{take}(n) \quad H, o'.\texttt{sub}() \rightarrow H', r}{H, o.\texttt{sub}() \rightsquigarrow H', \texttt{take}(r, n)} \quad S_{Take}$$

$$\frac{o = o'.\texttt{map}(f) \quad H, o'.\texttt{sub}() \rightarrow H', r}{H, o.\texttt{sub}() \rightsquigarrow H', \texttt{map}(r, f)} \quad S_{Map}$$

$$\frac{\begin{array}{c} o = o'.\texttt{switchMap}(f) \\ H, o'.\texttt{sub}() \rightarrow H', r \end{array}}{H, o.\texttt{sub}() \rightsquigarrow H', \texttt{switchMap}(r, f, \epsilon, \mathsf{false})} \quad S_{Switch}$$

Figure 4.8: The rules for subscribe operations.

Unsubscribe Operations While the subscribe operation constructs the subscription graph from the observables, the unsubscribe operation does the opposite by removing the subscriptions to the observables from the heap. Rule E_{Unsub} in Figure 4.7 evaluates $r.\texttt{unsub}()$ by first applying the rules for unsubscribe operation in Figure 4.9 and then removing the mapping of r from the heap. The rules in Figure 4.9 apply the E_{Unsub} rule to the references in each subscription value to unsubscribe them recursively.

The only exceptions are the rules for the shared observable, which uses reference counting to decide whether to unsubscribe its source. By Rule U_{Share1}, if a shared observable x is still used by another subscription r', then the unsubscribe operation $r.\texttt{unsub}()$ does not change the heap. Otherwise, by Rule U_{Share2}, the source observable of r will be unsubscribed.

4.4.4 Event Propagation

Once a subscription graph is constructed, the next stage of computation is event propagation, which may be interleaved with further modification to the subscription graph. The event propagation computation is described by the rules in Figure 4.10 and 4.11, where each reduction step has the form of $H, Q \to H', Q'$. Each rule (except Rule R_{From}) removes an event and subscription pair (t, r_1) from the front of the event queue Q and sends t to r_1's subscriber(s) – r, which may trigger further updates to the heap and the queue.

Most of the rules are concerned with $\mathsf{next}(v)$ event and end event. By Rule R_{Error}, the error event from a reference r is forwarded by default to the subscriber of r. This rule applies to the error propagation of all observables except the shared observable, which is handled by Rule R_{Share}

By Rule R_{Catch}, the error event of r' can be handled by the subscription $H(r) = \texttt{catchError}(r', f)$, which calls the error handler f to obtain an observable o, subscribes to o, and maps r to the subscription to o. In the end, r' is unsubscribed.

When processing events in the subscription graph, the new events may be added to the back or the front of the queue. For example, by Rule R_{From}, the event emitted

$$\frac{H(r) = \texttt{fromEvent}(elm, evt)}{H, r.\texttt{unsub}() \rightsquigarrow H} \quad U_{From}$$

$$\frac{\begin{array}{c} H(r) = \texttt{combine}(r_1, r_2, v_1, v_2, n) \\ H, r_1.\texttt{unsub}() \to H_1 \\ H_1, r_2.\texttt{unsub}() \to H_2 \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H_2} \quad U_{Combine}$$

$$\frac{\begin{array}{c} H(r) = \texttt{concat}(r_1, o_2) \\ H, r_1.\texttt{unsub}() \to H_1 \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H_1} \quad U_{Concat}$$

$$\frac{H(r) = \texttt{every}(r', f) \quad H, r'.\texttt{unsub}() \to H'}{H, r.\texttt{unsub}() \rightsquigarrow H'} \quad U_{Every}$$

$$\frac{\begin{array}{c} H(r) = \texttt{catchError}(r', f) \\ H, r'.\texttt{unsub}() \to H' \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H'} \quad U_{Catch}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{sharing}(x) \\ \exists r' \neq r.\ H(r') = \mathsf{sharing}(x) \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H} \quad U_{Share1}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{sharing}(x) \quad H(x) = \texttt{share}(o, r') \\ \nexists r'' \neq r.\ H(r'') = \mathsf{sharing}(x) \\ H, r'.\texttt{unsub}() \to H' \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H'[x \mapsto o.\texttt{share}()]} \quad U_{Share2}$$

$$\frac{H(r) = \texttt{take}(r', n) \quad H, r'.\texttt{unsub}() \to H'}{H, r.\texttt{unsub}() \rightsquigarrow H'} \quad U_{Take}$$

$$\frac{H(r) = \texttt{map}(r', f) \quad H, r'.\texttt{unsub}() \to H'}{H, r.\texttt{unsub}() \rightsquigarrow H'} \quad U_{Map}$$

$$\frac{\begin{array}{c} H(x) = \texttt{switchMap}(r_1, f, r_2, b) \\ H, r_1.\texttt{unsub}() \to H_1 \\ H_1, r_2.\texttt{unsub}() \to H_2 \end{array}}{H, r.\texttt{unsub}() \rightsquigarrow H_2} \quad U_{Switch}$$

$$\frac{H(r) = \texttt{sub}(r', f) \quad H, r'.\texttt{unsub}() \to H'}{H, r.\texttt{unsub}() \rightsquigarrow H'} \quad U_{Sub}$$

Figure 4.9: The rules for unsubscribe operations.

from the subscription to $\texttt{fromEvent}(elm, evt)$ is added to the back of the event queue. This is consistent with the event handling of JavaScript, where asynchronous events are processed in the macro task queue.

Other than external events such as those from DOM elements, there are also internal events created by RxJS operators. For example, the `combine` operator emits the latest values from its sources when one of them emits (after both have emitted). By Rule $R_{Combine2}$, the event of the `combine` operator is put in front of the queue so that it is received immediately by the subscriber of this observable. Rule $R_{Combine1}$ applies to the case when one of the source observable has not emitted any event. By Rules $R_{Combine3}$ and $R_{Combine4}$, an `end` event is emitted after all source observables have completed. The `end` event is also placed in front of the queue so that it is received immediately by the subscriber.

By Rules $R_{Concat1}$ and $R_{Concat2}$, $\texttt{concat}(o_1, o_2)$ combines the event streams of o_1 and o_2 sequentially such that it subscribes to o_1 first and after o_1 completes, it subscribes to o_2. The observable $o.\texttt{every}(f)$ emits a false value immediately if an event value from o does not satisfy the predicate function f (by Rule R_{Every1}). It emits a true value after all its events satisfy f (by Rules R_{Every2} and R_{Every3}).

By Rule R_{Share}, the observable $o.\texttt{share}()$ broadcasts the events of o to all subscribers of the shared observable. The observable $o.\texttt{take}(n)$ forwards the first n events of o to its subscriber (by Rules R_{Take1} and R_{Take2}) or until o emits an `end` event (by Rule R_{Take3}). Note that if the subscription r' of o is still running after n events have been emitted, it must be unsubscribed (Rule R_{Take2}).

By Rules R_{Map1} and R_{Map2}, the observable $o.\texttt{map}(f)$ transforms each $\texttt{next}(v)$ event from o and re-emits the result. For $o.\texttt{switchMap}(f)$, there are 5 rules: Rule $R_{Switch1}$ says that the events from the inner observable are re-emitted. Rules $R_{Switch2}$ and $R_{Switch3}$ describe how the event $\texttt{next}(v)$ from the outer observable interrupts the inner observable (if any) and starts a new subscription to the observable evaluated from $f(v)$. Rules $R_{Switch4}$ and $R_{Switch5}$ say that if both of the outer and inner observables have completed, then an `end` event is emitted.

$$\frac{H(r) = \texttt{fromEvent}(elm, evt) \quad elm \text{ emits } evt}{H, Q \to H, Q@[(\mathsf{next}(evt), r)]} \quad R_{From}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, \epsilon, n) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v, \epsilon, n)]}{H, (\mathsf{next}(v), r_i) :: Q \to H', Q} \quad R_{Combine1}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, n) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v, v_j, n)] \quad Q' = (\mathsf{next}((v, v_j)), r) :: Q}{H, (\mathsf{next}(v), r_i) :: Q \to H', Q'} \quad R_{Combine2}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, 0) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v_i, v_j, 1)]}{H, (\mathsf{end}, r_i) :: Q \to H', Q} \quad R_{Combine3}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, 1) \quad i \neq j}{H, (\mathsf{end}, r_i) :: Q \to H, (\mathsf{end}, r) :: Q} \quad R_{Combine4}$$

$$\frac{H(r) = \texttt{concat}(r_1, o_2)}{H, (\mathsf{next}(v), r_1) :: Q \to H, (\mathsf{next}(v), r) :: Q} \quad R_{Concat1}$$

$$\frac{H(r) = \texttt{concat}(r_1, o_2) \quad H, o_2.\mathsf{sub}() \rightsquigarrow H', s \quad H'' = H'[r \mapsto s]}{H, (\mathsf{end}, r_1) :: Q \to H'', Q} \quad R_{Concat2}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad H, f(v) \to H', \mathsf{false} \quad H', r_1.\texttt{unsub}() \to H'' \quad Q' = [(\mathsf{next}(\mathsf{false}), r), (\mathsf{end}, r)]@Q}{H, (\mathsf{next}(v), r_1) :: Q \to H'', Q'} \quad R_{Every1}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad H, f(v) \to H', \mathsf{true}}{H, (\mathsf{next}(v), r_1) :: Q \to H', Q} \quad R_{Every2}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad Q' = [(\mathsf{next}(\mathsf{true}), r), (\mathsf{end}, r)]@Q}{H, (\mathsf{end}, r_1) :: Q \to H, Q'} \quad R_{Every3}$$

$$\frac{H(r) = \texttt{catchError}(r_1, f) \quad H, f() \to H_1, o \quad H_1, o.\mathsf{sub}() \rightsquigarrow H_2, s \quad H_2[r \mapsto s], r_1.\texttt{unsub}() \to H'}{H, (\mathsf{error}, r_1) :: Q \to H', Q} \quad R_{Catch}$$

$$\frac{r_1 \text{ appears in } H(r)}{H, (\mathsf{error}, r_1) :: Q \to H, (\mathsf{error}, r) :: Q} \quad R_{Error}$$

Figure 4.10: The reduction rules for subscriptions 1

Rules R_{Sub1}, R_{Sub2}, and R_{Sub3} define the behavior of a top-level subscription $o.\mathsf{sub}(f)$, which calls f with v for each $\mathsf{next}(v)$ event from o until an end event is emitted. It stops the subscription to o if an error is emitted. Note that in RxJS, separate callback functions can be given to handle the end and the error events but this detail is omitted for simplicity.

Scheduler and Event Queue RxJS supports multiple types of event schedulers. By default, internal events are scheduled synchronously. For example, the events of the `combine` operator can be passed to its subscribers directly via function calls. However, a subscription can run on schedulers such as the async scheduler, which is based on the JavaScript queue for asynchronous events. The rules in Figure 4.10 and 4.11 correspond to the default scheduler where the internal events are placed in front of the queue so that they must be processed immediately. To emulate the behavior of an async scheduler, we can modify the rules so that the internal events are placed at the end of Q so that they are processed after the events on the queue are processed.

4.4.5 Additional Operators

RxJS has over 100 operators and this semantics only considered a subset selected based on functionalities, which include combination, creation, error handling, multicasting, filtering, and transformation. Other operators in these classes can be formalized in similar ways with additional complexities such as buffering.

For example, the operator `o.concatAll()` subscribes to each of the observables emitted from `o` and concatenates the resulting event streams. This operator uses a buffer to hold the new observables emitted from o while waiting for the current observable to complete. The operator `zip(o1,o2)` pairs events from `o1` and `o2` until one of them completes and buffers are needed to hold events from either observables in case that they do not emit at the same rate.

Some operators may be implemented using others. For example, `o.combineAll()` combines the events from the observables emitted from `o` in tuples, which can be imple-

$$\frac{\exists x.\ H(x) = \texttt{share}(o, r) \quad \forall i \in \{1..n\}.\ H(r_i) = \mathsf{sharing}(x) \quad Q' = [(t, r_1), \ldots, (t, r_n)]@Q}{H, (t, r) :: Q \to H, Q'} \; R_{Share}$$

$$\frac{H(r) = \texttt{take}(r', n) \quad n \geq 2 \quad H' = H[r \mapsto \texttt{take}(r', n-1)]}{H, (t, r') :: Q \to H', (t, r) :: Q} \; R_{Take1}$$

$$\frac{H(r) = \texttt{take}(r', 1) \quad H, r'.\texttt{unsub}() \to H'}{H, (t, r') :: Q \to H', [(t, r), (\mathsf{end}, r)]@Q} \; R_{Take2}$$

$$\frac{H(r) = \texttt{take}(r', n)}{H, (\mathsf{end}, r') :: Q \to H, (\mathsf{end}, r) :: Q} \; R_{Take3}$$

$$\frac{H(r) = \texttt{map}(r', f) \quad H, f(v) \to H', v'}{H, (\mathsf{next}(v), r') :: Q \to H', (\mathsf{next}(v'), r) :: Q} \; R_{Map1}$$

$$\frac{H(r) = \texttt{map}(r', f)}{H, (\mathsf{end}, r') :: Q \to H, (\mathsf{end}, r) :: Q} \; R_{Map2}$$

$$\frac{H(r) = \texttt{switchMap}(r_1, f, r_2, b)}{H, (\mathsf{next}(v), r_2) :: Q \to H, (\mathsf{next}(v), r) :: Q} \; R_{Switch1}$$

$$\frac{H(r) = \texttt{switchMap}(r_1, f, \epsilon, b) \quad H, f(v) \to H_1, o \quad H_1, o.\mathsf{sub}() \to H_2, r_2 \quad H_3 = H_2[r \mapsto \texttt{switchMap}(r_1, f, r_2, b)]}{H, (\mathsf{next}(v), r_1) :: Q \to H_3, Q} \; R_{Switch2}$$

$$\frac{H(r) = \texttt{switchMap}(r_1, f, r_2, b) \quad H, f(v) \to H_1, o \quad H_1, o.\mathsf{sub}() \to H_2, r_2' \quad H_3 = H_2[r \mapsto \texttt{switchMap}(r_1, f, r_2', \mathsf{false})] \quad H_3, r_2.\texttt{unsub}() \to H'}{H, (\mathsf{next}(v), r_1) :: Q \to H', Q} \; R_{Switch3}$$

$$\frac{H(r) = \texttt{switchMap}(r_1, f, r_2, \mathsf{true}) \quad Q' = (\mathsf{end}, r) :: Q}{H, (\mathsf{end}, r_i) :: Q \to H, Q' \quad i \in \{1, 2\}} \; R_{Switch4}$$

$$\frac{H(r) = \texttt{switchMap}(r_1, f, r_2, \mathsf{false}) \quad H' = H[r \mapsto \texttt{switchMap}(r_1, f, r_2, \mathsf{true})]}{H, (\mathsf{end}, r_i) :: Q \to H', Q \quad i \in \{1, 2\}} \; R_{Switch5}$$

$$\frac{H(r) = \mathsf{sub}(r', f) \quad f(v)}{H, (\mathsf{next}(v), r') :: Q \to H, Q} \; R_{Sub1}$$

$$\frac{H(r) = \mathsf{sub}(r', f)}{H, (\mathsf{end}, r') :: Q \to H, Q} \; R_{Sub2}$$

$$\frac{H(r) = \mathsf{sub}(r', f) \quad H, r'.\texttt{unsub}() \rightsquigarrow H'}{H, (\mathsf{error}, r') :: Q \to H', Q} \; R_{Sub3}$$

Figure 4.11: The reduction rules for subscriptions 2

mented using `combineLatest` after collecting all the observables from `o`. The operator `o.endsWith(x)` appends to the events from `o` with the value `x`, which can be implemented with `concat` and the operator `of` (that creates an observable out of values).

The semantics of some operators is difficult to characterize concisely. For example, the operator `o.debounce(selector)` controls the emission rate of `o` by racing its events with the observable returned from the `selector` function. Other filter operators such as `distinct` and `throttle` also have complex semantics related to the value and timing of the events. Even more complex are the transformation operators that group the source events using buffers or windows based on the event count, event timing, and timing observables.

Lastly, the multicasting operators use classes like `Subject` to broadcast events from a source observable to multiple subscribers. The operators such as `window` and `groupBy` can use `Subject` to implement observables that emit events selected from a source observable. The `Subject` class also has methods for direct event emission, which can be used to emit events for a observable from any part of the program as a side effect. While we can represent operators like `share`, we have not modeled this type of event emission.

4.5 Implementation

To provide a testing ground for the proposed semantics, we implemented a subset of RxJS operators using a thread-like abstractions called `AsyncM` [57], which allows asynchronous computation be implemented like a cancellable thread. This implementation includes 100 RxJS operators with about 2500 lines of code (`https://github.com/tianzhao/rxjs`).

`AsyncM` allows us to chain asynchronous computation just like JavaScript promises except that `AsyncM` can be interrupted via an associated progress object. This abstraction provides a convenient way to implement observables like `fromEvent`, which waits for external events in a loop.

Unlike the operational semantics, the implementation does not need to maintain an

explicit queue to hold external events, since `AsyncM` uses promises to handle asynchronous events. Also unlike the semantic, which uses heap variables to maintain bidirectional relation between an observable and its sources, in the implementation, an observable uses variables to access its sources but uses an emitter object to send events to its subscribers.

Subscription When an observable is subscribed, two threads are launched, which share an emitter and a progress object that are stored in the returned subscription object. This structure is shown in Figure 3.11.

The class for observable has a subscribe method that runs its argument k for each event value until either the `end` event is emitted or an `error` has percolated to the top, which cancels the subscription.

4.5.1 Composite Observable

The composition operators are derived from the methods of observable class. In this section, we explain a few of the operators.

map The call `o.map(f)` applies a synchronous function f to each event value from `o` except the `end` event. Any exception in f is caught and emitted as an `error` event. If the input event is an error, then it is re-emitted by default.

switchMap The call `o.switchMap(f)` applies f to each event of the outer observable `o`, which returns an inner observable. Each time the outer observable emits an event, the current inner observable (if exists) is unsubscribed and a new inner observable is subscribed.

The implementation of `switchMap` is similar to that of `map` in that any exception raised in f is caught and emitted as an error event. The subscription to inner observable is referenced as a child of the subscription to the `switchMap`. An error event from the outer or the inner observable is re-emitted by default.

One tricky thing in `switchMap` is to determine when the end is. In RxJS, the end event does not have a value. For example, `of(1,2,3).filter(x=>x<3)` will emit 1 and

2 and then end. Thus, we cannot determine the end of `switchMap` by observing the end of the last inner observable since we do not know whether an inner observable is the last one until the outer observable emits its end event.

The implementation of `switchMap` uses a flag to mark the end of the outer or the inner observable. When the outer observable emits `end` event, `switchMap` will emit the `end` event if the flag is set (which indicates that the current inner observable has ended). Otherwise, it will set the flag and wait for the inner observable (which we now know is the last one) to end.

concatAll A few of the RxJS operators are buffered such as `concatAll`, which concatenates a stream of observables as a single observable. Each inner observable must end before the next one starts. To prevent the loss of the outer observable events, a channel is used. A channel will block its read method if it is empty and block its write method if it is full (if a bound is set). The channel class is also implemented with `AsyncM` so that threads blocked on it can be cancelled.

The `concatAll` operator writes the inner observable emitted from the outer observable into the channel buffer and subscribes to each inner observable read from the channel. The `concatAll` observable ends when an `end` event is read from the channel.

share The RxJS observables are not shared by default so that each subscription starts a new instance of an observable. RxJS allows an observable be shared through its `share` operator, which returns a subject that starts running when it is first subscribed and emits events shared by each subsequent subscription. The subject uses reference counting to determine when it should end. That is, when the number of subscriptions drops to zero, the subject terminates.

The subject class derives from the observable class and overrides its internal subscribe method so that multiple subscriptions will listen on events from the same emitter.

catchError An observable may emit error events originated from a primitive or composite observable due to causes such as a rejected promise or an exception in a map

function. By default, if an error event is not handled, it will propagate outwards until it reaches the top-level subscriber, when it will cause all subscriptions be cancelled. An error event can be handled with the `catchError` operator, which cancels the source observable and replaces it with a new one.

The `catchError` operator subscribes to its source observable with an error handler that will stop the current subscription and replace it with the subscription to a new observable. The new observable is returned from the argument function f given the error value. However, if the call to f also fails with an exception, then `catchError` emits an error event of its own instead.

Break point As explained in [1], the difficulties with debugging RxJS program include the inability to set break points for expressions such as `take(n)` and that the stack trace at a break point does not correspond to the dataflow states of the RxJS program. Our design does provide the ability to inspect the dataflow graphs at break points through the subscription objects. However, we have not implemented a way to add break points to expressions like `take(n)` since it is only used to compose the observable and the subsequent event-handling does not go through the call to `take(n)`.

Stack trace We have implemented a strategy to capture the subscription graph and the different stack traces when an error occurs. To capture the stack trace, an error handler is used at places that would execute client code such as `fmap` and `switchMap`. The error handler will catch any error thrown from the client code and since the handler is at the topmost part of the stack running in the reactive code, we can easily separate the two types of stack by removing the portion of the stack that share with the handler's stack.

We examine the stack trace by relying on a non-standard feature of all major Web browsers and JavaScript runtime, so it is only possible if the exception value is an instance of the built-in `Error` object. To trace the state of the reactive code, we only need to capture the portion of the subscription graph prior to the error site. This is done by following through the edge to the subscription sources.

Runtime Safety Invariant The safety invariant is implemented by following the rules in Figure 4.12. Starting from a root subscriber, we examine the subscription type and make the appropriate assertion by recursively traversing to the source and child subscriptions, checking whether each event source is covered by at least one `catchError` operator. To enable rechecking of the subscription graph when a new subscription is added, each subscription that we came across during the first traversal is assigned to a location accessible to its source/child subscriptions. When a new subscription is added, the subscription assigned previously can be used as a starting point for rechecking the graph.

4.6 Testing and Debugging

4.6.1 Debugging with Subscription Graph

The semantics of λ_{rx} provides a simplified model for describing the debugging support for RxJS programs. In this section, we discuss the representation of stack trace, detecting unexpected states of observables, and checking the invariant of subscription graph.

4.6.2 Stack Trace

For sequential programs, debuggers can be used to set break-points to pause the execution so that the programmers can examine the runtime states and discover the sources of errors. The runtime state includes a call stack with a list of stack frames that contains the local variables. The call stack can be dumped as a stack trace if a program crashes due to an exception. However, for RxJS program, such a stack trace is not very informative for debugging purposes. The dataflow information of a subscription graph is obscured by the underlying implementation of RxJS. For example, the event propagation in the subscription graph may appear as direct function calls, invocation of callback listeners, or resolution of promises depending on the types of the events and the event scheduler. Thus, we need a more abstract representation of the runtime state of a RxJS program.

The stack trace of a RxJS program may include the stack frames of normal function calls and/or the dataflow paths from observables to their subscribers. There are three types of stack traces in a RxJS program.

- The first type represents the construction phase of a subscription graph, where the stack trace consists of only stack frames.
- The second type represents the reactive phase of the computation, where the trace includes only dataflow paths. Since the only event source in λ_{rx} is asynchronous, event propagation will not start until the synchronous computation has completed, which means that the call stack is empty at this point.
- The third type includes dataflow paths followed by stack frames, which may occur when an expression break-point is reached during event propagation. For example, in $o = o'.\texttt{switchMap}(f)$, when the outer observable o' emits an event, the function f runs. If a break-point within f is reached, then the stack trace will include the dataflow path that leads to o and the call stack that leads to the break-point in f.

Break point In λ_{rx}, we can set break-points for expressions by placing labels. For example, the expression e^{ℓ} has the break-point ℓ. When a computation reaches the break-point ℓ as shown below, it pauses with the current heap H and call stack E.

$$\frac{H_1, e_1 \rightarrow H_2, e_2}{H_1, E[e_1^{\ell}] \rightarrow H_2, E[e_2]}$$

We can set break-point for subscription s by attaching a label ℓ to the observable that s is reduced from.

$$H, o^{\ell}.\texttt{sub}() \rightsquigarrow H', s^{\ell}$$

If an event is received at or emitted from a subscription s with a break-point ℓ, where $H(r) = s^{\ell}$ and $H, Q \rightarrow H', Q'$, then the program pauses and allows programmers to inspect the current state H' and Q'. This type of break-points can be enhanced with filters to limit the type of events.

Dataflow Path The stack trace may include dataflow path if a break-point is reached at a subscription or at an expression that runs due to event propagation (e.g. `switchMap`). The dataflow path should allow a programmer to trace the event sources from the break-point. Since each subscription has references to its source subscriptions, we can follow these references to recover the dataflow path. However, if a subscription has multiple event sources (e.g. `combine`), then we must know the latest event source, which can be implemented with a flag. Detailed data such as the last k events and their time-stamps can be added to help with debugging.

4.6.3 Subscription State

We can debug a RxJS program by monitoring the state of a subscription. RxJS has numerous operators and the subscription to each operator have multiple states. If a subscription enters a state unexpected by the programmers, the resulting behavior may cause an error that is hard to debug.

For example, the subscription to a `combine` observable will emit an event only after both of its source observables have emitted at least one event. If one of the source observable completes before emitting any value, then the `combine` observable will not emit anything regardless how the other source observable behaves. Thus, it may be useful to raise an alert that one of the source observables to a `combine` observable only emits an `end` event. This may help identify the cause why a `combine` observable never emits.

For the subscription `concat`(r_1, o_2), we can raise an alert if r_1 completes without emitting any value. This may be incorrect since the result of the concatenation would be entirely that of o_2, which may be unexpected.

For the subscription `every`(r_1, f), if its source observable r_1 ends before firing any event, it will emit `true` and then complete instead of ending without emitting any value. This behavior may not be expected either.

For the subscription `switchMap`(r_1, f, r_2, b), if the outer observable r_1 emits before the inner observable r_2 emits any event, then the `switchMap` observable will not fire any

$$\mathsf{safe}_H(\texttt{fromEvent}(_,_),1)$$

$$\frac{\mathsf{safe}_H(r_1,n_1) \quad \mathsf{safe}_H(r_2,n_2) \quad n_1+n_2 \le n}{\mathsf{safe}_H(\texttt{combine}(r_1,r_2,_,_,_),n)}$$

$$\frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{concat}(r,_),n)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{every}(r,_),n)}$$

$$\frac{\mathsf{safe}_H(r,1)}{\mathsf{safe}_H(\texttt{catchError}(r,_),0)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\mathsf{sharing}(r),n)}$$

$$\frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{take}(r,_),n)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{map}(r,_),n)}$$

$$\frac{\mathsf{safe}_H(r_1,n_1) \quad \mathsf{safe}_H(r_2,n_2) \quad n_1+n_2 \le n}{\mathsf{safe}_H(\texttt{switchMap}(r_1,_,r_2,_),n)}$$

$$\frac{\mathsf{safe}_H(r,1)}{\mathsf{safe}_H(\texttt{sub}(r,_),0)} \qquad \frac{\mathsf{safe}_H(H(r),n)}{\mathsf{safe}_H(r,n)}$$

$$\frac{\forall r.\ H(r)=s=\texttt{sub}(_,_) \quad \mathsf{safe}_H(s,0)}{\mathsf{safe}(H)}$$

Figure 4.12: The safety rules that check the number of event sources guarded by a catch observable or root subscriber.

events from r_2. Furthermore, if r_1 always emits events at a higher rate than r_2, then the `switchMap` may not emit anything. Thus, we may want to raise an alert if an inner observable of `switchMap` is unsubscribed before it emits any event.

4.6.4 Runtime Invariant

When a RxJS program crashes, it is not always clear where the error comes from, especially when there are multiple event sources. An unhandled error from any of the event sources can cause the entire program to stop. One way to prevent this is to ensure that all potential sources of errors are protected by a catch observable so that the source of an error can be uniquely identified and the program can possibly recover from the error. To this end, we can check the subscription graph to ensure that each `catchError` (or the root subscriber) can trace to at most one event source without going through another `catchError`.

Figure 4.12 shows the safety rules to check whether in a heap H, each event source (i.e. `fromEvent`) is uniquely guarded by a `catchError` or root subscriber. The predicate $\mathsf{safe}_H(s,n)$ says that in the subscription s, there are at most n event sources that are not

guarded by a `catchError` or `sub`. Given e, if $\emptyset, e \rightarrow^* H, v$, we can check the safety of H by checking the predicate $\mathsf{safe}_H(s, 0)$ for each $s = \mathtt{sub}(x, f)$ in H, where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

Since the subscription graph in the heap changes during computation, we also need to recheck H when new subscriptions are added. Among the RxJS operators that we considered, only `concat`, `catchError`, and `switchMap` operators will dynamically add subscriptions to the heap. Note that since operators like `map` and `every` can execute arbitrary expression, it can add subscriptions to the heap as well though this is not how they are typically used. Thus, we can check the safety of a RxJS program e as follows:

- If $\emptyset, e \rightarrow^* H, v$, then $\mathsf{safe}(H)$ (as defined in Figure 4.12).
- If $H, Q \rightarrow H', Q'$ by Rule $R_{Concat2}$, R_{Catch}, or $R_{Switch3}$, then $\mathsf{safe}(H')$.

For the updated heap, we can just check the new subscriptions instead of the entire heap for better efficiency.

4.7 Related Work

4.7.1 FRP

Functional reactive programming (FRP) [16, 15] is a framework for modeling continuously changing behaviors that react to discrete events. Classic FRP is pull-based, which detects events by polling in discrete time steps with the implication that the event latency depends on the step size and the behaviors are checked for possible switching every time step. Push-based FRP such as FrTime [11], Flapjax [34], Scala React [32], ReactiveX [50], and Elm [13] provide timely responses to events and avoid re-computation when events do not occur. They wait on event occurrences and only run when an event occurs.

Many research efforts have been devoted to fix problems such as space-time leaks and event glitches using methods such as global dispatcher [34], blocking IO [47], static analysis [26], type-based restrictions [39], mutable memory [48], arrow-based abstractions [29,

12, 9], and a combination of arrow and monad [40, 4]. In practice, however, most of the research designs have not seen wide adoptions like ReactiveX and its JavaScript version RxJS. Despite its flaws, such as the potential of glitches and space leaks, users find its wide range of features appealing.

4.7.2 RxJS

The reference implementation[1] of RxJS uses Observer Pattern, where an observable passes events to its concrete observers through a generic observer interface. A composition operator acts like a bridge between a source observable and a destination observer where the composition logic is implemented as a decorator of the destination observer. Because a source observable is connected to its intermediate and final observers through references, each event is passed through specific method calls, which includes *next*, *complete*, and *error* events. To check the events between an observable and its intermediate/final observers, a user has to locate and monitor the correct method calls in many classes.

In our design, an observable emits events to its intermediate/final observers through emitters and all types of events pass through the emitter where users can examine a finite history of past events, which include event types, values, and (optional) timing information.

In RxJS, a subscription object unsubscribes by mutating object states. In our design, observables are implemented using `AsyncM` that runs like cancellable threads. In a composite observable, the subscriptions to the inner/source observables are child threads. `AsyncM` supports hierarchical cancellation so that the cancellation of a thread also cancels its child threads. This simplifies the implementation since to unsubscribe an observable, all it takes is to cancel the subscription thread.

Debugging tools for RxJS This work is motivated by providing helpful debugging information for reactive applications. Due to the complexity of RxJS, users often rely on logging to find errors, which may be helped by logging tools like rxjs-spy[2], which adds tag

[1] http://reactivex.io/rxjs
[2] https://github.com/cartant/rxjs-spy

operators to RxJS so that a trace log can be monitored, paused, and replayed through console. While logging tool reduces debugging workload, trace logs can be difficult to interpret by visual inspection when numerous events are emitted. In comparison, our formal semantics shows that the subscription graph can be inputs to test functions for automated verification and error detection. Visualization tools such as RxFiddle [3], and RxViz[3], and rxjs-playground[4] help visualize the dataflow graph and timing of event emissions through (animated) marble diagrams. These tools are useful for understanding the semantics of RxJS programs but like trace logs, the resulting diagrams have to be manually inspected.

Concurrency `AsyncM` leverages promises [14] to provide a thread-like concurrency model with cooperative cancellation. While promises do not have builtin methods for cancellation, `AsyncM` enhances the promise constructs with a more consistent way to terminate unused computation.

`AsyncM` is a form of concurrency monad, which is used by Claessen [10] for supporting a simple form of concurrency in Haskell and by Li and Zidancwic [28] in their design for scalable network services. The cancellation mechanism of `AsyncM` is similar to the asynchronous exception of Concurrent Haskell [33], which allows a thread to terminate another thread by throwing an asynchronous exception. However, since JavaScript is not preemptive, the interrupt exceptions in our design are only received at the locations where the threads are blocked or polling the thread status.

The progress object in our design is similar to the cancellation token of *F#* [43, 53] and .Net[5]. The difference is that our design integrates cancellation into a thread model, where a progress (i.e. thread ID) is both the cancellation source and token. The hierarchical structure of threads allows a child thread to react to cancellation requests to its parent threads without explicitly linking cancellation tokens.

[3]https://rxviz.com/

[4]https://github.com/hediet/rxjs-playground

[5]`https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads`

4.8 Summary

In this chapter, we defined a formal semantics for a selected set of operators in RxJS library. The semantics clarifies the representation of stack trace for RxJS programs, identifies potentially unexpected states of observables, and provides rules for checking sufficient error handling. We provided an implementation of RxJS based on the semantics, which uses an abstraction of cancellable threads to implement observables and their subscriptions. The subscription graph and the events in each graph node are available for debugging purpose.

Since JavaScript is single threaded, there are no simultaneous events. All external events occur in a sequence while internal events are processed when they are generated and before the external events. Thus, our semantics is deterministic, where the same sequence of events to a subscription will trigger the same sequence of reduction rules and result in the same reactive behavior. However, it is unclear whether RxJS is entirely deterministic but using our implementation, programmers can expect predictable outcome.

There are a number of complications in RxJS that are not considered in the semantics. For example, RxJS supports multiple forms of scheduling, which by default is synchronous for internal events. A queue scheduler may be used to process interval events in the order in which they are emitted. RxJS also supports ASAP scheduler that uses the queue for promises and the async scheduler that schedules internal events using event loop or animation frame. Our current semantics corresponds to the default scheduler though it can be easily modified to model the async scheduler. However, separate queues are needed for other type of schedulers. Also, operators like `concatAll` use buffer to hold events until they are used but this may lead to lost events if the buffer is finite or out-of-memory exception if the buffer is unbounded. Shared observables such as a subject can start, stop, or fire events as the side effects of some other observables. These features should be considered to provide accurate modeling of RxJS programs.

Chapter 5

Conclusion

5.1 Summary of Contributions

Our goal is to make it easier for programmers to develop real-time IoT and Web applications. In this thesis, we have studied different approaches of FPR solutions and present our own solution for real-time IoT applications. Our implementation is a push-pull reactive programming model to efficiently handle asynchronous data streams that can be used for real-time processing of high sampling-rate signals. We implemented the push-based reactive stream in order to minimize the latency in gathering data over the network and then use pull-based constructs in the application logics. The data collected from the push stream are stored in the buffer and resampled. This way we avoid the issue with glitches. We also demonstrated how to apply the operators to dynamically switch reactive streams in order to adjust the sampling-rate.

The presented implementation in Chapter 2 is written in Haskell. Implementing the same abstraction in JavaScript post a challenge for us because JavaScript does not have a concurrency model that supports cancellation. JavaScript uses an event-driven concurrency model with callback functions, but callback functions are not composable and it is often a source of confusion with regard to execution order, race conditions, and termination. For these reasons, we implemented a thread-like concurrency model called `AsyncM` based on the continuation monad and reader monad. Continuation monad enables handling concurrency in single-threaded settings, while the reader monad is used to thread an object for synchronizing and canceling asynchronous computation as if they were threads. The operational semantics for `AsyncM` is also presented in the same chapter.

RxJS is a reactive programming library for JavaScript, commonly used for developing Web applications. Difficult to use and debug are often cited problems for RxJS (and

other similar reactive languages). We presented a formal semantics for a selected set of operators and implemented them using `AsyncM`. Generally, it is accepted there is some sort of a data-flow graph that represent the reactive program logic. With this formal semantics, it gives us a simple model to reason about RxJS programs.

5.2 Future Work

Fault tolerance Having a strategy to handle failures is important for networking applications, especially for IoT applications since they are generally deployed to run for a long period of time. Logical errors can be encoded in value to signify error states and captured by program logic. But there are faults that are less detectable such as transmission error which causes data alteration or complete loss of data [41]. A solution to this type of fault can simply be adding redundancy [17]. Estimating the reliability of the system is another proposed solution [42]. All these indicate that this topic warrants great research in order to find out what needs to be implemented and how they impact the abstractions at the language level.

Type checking We have written our implementation in both Haskell and JavaScript. Haskell has a rich type system that always enforces the code to be type correct. However, JavaScript does not. In Chapter 4 section 4.6, we implemented an invariant check that analyzes the subscription graph at runtime, as an experiment to provide some help for users to *type check* their programs. This method of checking was very limited and should be explored in better ways to perform the check.

Bibliography

[1] Manuel Alabor and Markus Stolze. "Debugging of RxJS-Based Applications". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 15–24. ISBN: 9781450381888. DOI: 10.1145/3427763.3428313. URL: https://doi.org/10.1145/3427763.3428313.

[2] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: https://doi.org/10.1145/2501654.2501666.

[3] Herman Banken, Erik Meijer, and Georgios Gousios. "Debugging Data Flows in Reactive Programs". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 752–763. ISBN: 9781450356381. DOI: 10.1145/3180155.3180156. URL: https://doi.org/10.1145/3180155.3180156.

[4] Manuel Bärenz and Ivan Perez. "Rhine: FRP with Type-Level Clocks". In: *SIGPLAN Not.* 53.7 (Sept. 2018), pp. 145–157. ISSN: 0362-1340. DOI: 10.1145/3299711.3242757. URL: https://doi.org/10.1145/3299711.3242757.

[5] Gérard Berry. "The Constructive Semantics of Pure Esterel". In: (1999).

[6] Gérard Berry and Gérard Boudol. "The Chemical Abstract Machine". In: *Theor. Comput. Sci.* 96.1 (Apr. 1992), pp. 217–248. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90185-I. URL: https://doi.org/10.1016/0304-3975(92)90185-I.

[7] Gérard Berry and Manuel Serrano. "HipHop. js:(A) Synchronous reactive web programming." In: *PLDI*. 2020, pp. 533–545.

[8] Gérard Boudol. *Asynchrony and the Pi-Calculus*. 1992.

[9] Guerric Chupin and Henrik Nilsson. “Functional Reactive Programming, restated”. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. 2019, pp. 1–14.

[10] Koen Claessen. “A Poor Man’s Concurrency Monad”. In: *J. Funct. Program.* 9.3 (May 1999), pp. 313–323. ISSN: 0956-7968. DOI: 10.1017/S0956796899003342. URL: https://doi.org/10.1017/S0956796899003342.

[11] Gregory H Cooper and Shriram Krishnamurthi. “Embedding dynamic dataflow in a call-by-value language”. In: *European Symposium on Programming*. Springer. 2006, pp. 294–308.

[12] Antony Courtney, Henrik Nilsson, and John Peterson. “The yampa arcade”. In: *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 2003, pp. 7–18.

[13] Evan Czaplicki and Stephen Chong. “Asynchronous Functional Reactive Programming for GUIs”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, June 2013, pp. 411–422.

[14] ECMA International. *ECMA-262: ECMAScript 2015 Language Specification*. Standard. ECMA International, June 2015. URL: http://www.ecma-international.org/ecma-262/6.0/.

[15] Conal Elliott. “Push-pull functional reactive programming”. In: *Haskell Symposium*. 2009. URL: http://conal.net/papers/push-pull-frp.

[16] Conal Elliott and Paul Hudak. “Functional reactive animation”. In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 1997, pp. 263–273.

[17] Steven Engelen, Eberhard Gill, and Chris Verhoeven. “On the reliability, availability, and throughput of satellite swarms”. In: *IEEE Transactions on Aerospace and Electronic Systems* 50.2 (2014), pp. 1027–1037.

[18] Ralf S. Engelschall. *The GNU Portable Threads*. June 2006. URL: https://www.gnu.org/software/pth/.

[19] Amin Milani Fard and Ali Mesbah. "JavaScript: The (Un)Covered Parts". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 2017, pp. 230–240. DOI: 10.1109/ICST.2017.28.

[20] Python Software Foundation. *The Python Language Reference*. URL: https://docs.python.org/3/reference/expressions.html.

[21] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[22] Tim Harris et al. "AC: Composable Asynchronous IO for Native Languages". In: *SIGPLAN Not.* 46.10 (Oct. 2011), p. 903920. ISSN: 0362-1340. DOI: 10.1145/2076021.2048134. URL: https://doi.org/10.1145/2076021.2048134.

[23] Shin Hong, Yongbae Park, and Moonzoo Kim. "Detecting Concurrency Errors in Client-Side JavaScript Web Applications". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 61–70. DOI: 10.1109/ICST.2014.17.

[24] Wolfgang Jeltsch. "Signals, Not Generators!" In: *Trends in Functional Programming*. 2009.

[25] D.E. Knuth. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium*. Pearson Education, 2005. ISBN: 9780321657312. URL: https://books.google.com/books?id=imjwBQAAQBAJ.

[26] Neelakantan R. Krishnaswami. "Higher-Order Functional Reactive Programming without Spacetime Leaks". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 221–232. ISSN: 0362-1340. DOI: 10.1145/2544174.2500588. URL: https://doi.org/10.1145/2544174.2500588.

[27] Paul LeGuernic et al. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336.

[28] Peng Li and Steve Zdancewic. “Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives”. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 189–199. ISSN: 0362-1340. DOI: 10.1145/1273442.1250756. URL: https://doi.org/10.1145/1273442.1250756.

[29] Hai Liu and Paul Hudak. “Plugging a space leak with an arrow”. In: *Electronic Notes in Theoretical Computer Science* 193 (2007), pp. 29–45.

[30] Matthew C Loring, Mark Marron, and Daan Leijen. “Semantics of asynchronous JavaScript”. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. 2017, pp. 51–62.

[31] Magnus Madsen, Ondřej Lhoták, and Frank Tip. “A Model for Reasoning About JavaScript Promises”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), p. 86. ISSN: 2475-1421. DOI: 10.1145/3133910.

[32] Ingo Maier and Martin Odersky. *Deprecating the observer pattern with Scala. react.* Tech. rep. 2012.

[33] Simon Marlow et al. “Asynchronous Exceptions in Haskell”. In: *SIGPLAN Not.* 36.5 (May 2001), pp. 274–285. ISSN: 0362-1340. DOI: 10.1145/381694.378858. URL: https://doi.org/10.1145/381694.378858.

[34] Leo A Meyerovich et al. “Flapjax: a programming language for Ajax applications”. In: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 2009, pp. 1–20.

[35] Ragnar Mogk et al. “Fault-tolerant distributed reactive programming”. In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[36] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. “Detecting JavaScript Races that Matter”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 381–392. DOI: 10.1145/2786805.2786820.

[37] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. “Distributed reactive programming for reactive distributed systems”. In: *arXiv* (2019), arXiv–1902.

[38] Henrik Nilsson, Antony Courtney, and John Peterson. “Functional Reactive Programming, Continued”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell ’02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 51–64. ISBN: 1581136056. DOI: 10.1145/581690.581695. URL: https://doi.org/10.1145/581690.581695.

[39] Gergely Patai. “Efficient and Compositional Higher-Order Streams”. In: *Functional and Constraint Logic Programming*. Ed. by Julio Mariño. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 137–154. ISBN: 978-3-642-20775-4.

[40] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. “Functional reactive programming, refactored”. In: *ACM SIGPLAN Notices* 51.12 (2016), pp. 33–44.

[41] Ivan Perez, Alwyn Goodloe, and William Edmonson. “Fault-tolerant swarms”. In: *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE. 2019, pp. 47–54.

[42] Ivan Perez and Alwyn E Goodloe. “Fault-tolerant functional reactive programming (extended version)”. In: *Journal of Functional Programming* 30 (2020).

[43] Tomas Petricek and Don Syme. “The F# Computation Expression Zoo”. In: *Practical Aspects of Declarative Languages*. Ed. by Matthew Flatt and Hai-Feng Guo. Cham: Springer International Publishing, 2014, pp. 33–48. ISBN: 978-3-319-04132-2.

[44] Boris Petrov et al. “Race Detection for Web Applications”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 251–262. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254095.

[45] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. “Concurrent Haskell”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: As-

sociation for Computing Machinery, 1996, pp. 295–308. ISBN: 0897917693. DOI: 10.1145/237721.237794. URL: https://doi.org/10.1145/237721.237794.

[46] Daniel Pilaud, N Halbwachs, and JA Plaice. "LUSTRE: A declarative language for programming synchronous systems". In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY.* Vol. 178. 1987, p. 188.

[47] Atze van der Ploeg. "Monadic Functional Reactive Programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell.* Haskell '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 117–128. ISBN: 9781450323833. DOI: 10.1145/2503778.2503783. URL: https://doi.org/10.1145/2503778.2503783.

[48] Atze van der Ploeg and Koen Claessen. "Practical Principled FRP: Forget the Past, Change the Future, FRPNow!" In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 302–314. ISSN: 0362-1340. DOI: 10.1145/2858949.2784752. URL: https://doi.org/10.1145/2858949.2784752.

[49] Veselin Raychev, Martin Vechev, and Manu Sridharan. "Effective Race Detection for Event-driven Programs". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications.* OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 151–166. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509538.

[50] *Reactive Extensions.* http://reactivex.io/. Accessed: 2020-07-02.

[51] Kazuhiro Shibanai and Takuo Watanabe. "Distributed functional reactive programming on actor-based runtime". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control.* 2018, pp. 13–22.

[52] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. "Efficient Coroutines for the Java Platform". In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java.* PPPJ '10. Vienna, Austria:

Association for Computing Machinery, 2010, pp. 20–28. ISBN: 9781450302692. DOI: 10.1145/1852761.1852765. URL: https://doi.org/10.1145/1852761.1852765.

[53] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# Asynchronous Programming Model". In: *Practical Aspects of Declarative Languages*. Ed. by Ricardo Rocha and John Launchbury. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 175–189. ISBN: 978-3-642-18378-2.

[54] William W Wadge and Edward A Ashcroft. *LUCID, the dataflow programming language*. Vol. 198. Academic Press London, 1985.

[55] Zhanyong Wan, Walid Taha, and Paul Hudak. "Real-time FRP". In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 2001, pp. 146–156.

[56] Tian Zhao, Adam Berger, and Yonglun Li. "Asynchronous Monad for Reactive IoT Programming". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 25–37. ISBN: 9781450381888. DOI: 10.1145/3427763.3428314. URL: https://doi.org/10.1145/3427763.3428314.

[57] Tian Zhao and Yonglun Li. "A Concurrency Model for JavaScript with Cooperative Cancellation". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 55–67. ISBN: 9781450391115. DOI: 10.1145/3486608.3486911. URL: https://doi.org/10.1145/3486608.3486911.