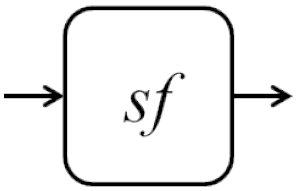


YALE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming

PhD Dissertation

Author:
Daniel WINOGRAD-CORT
dwc@cs.yale.edu

Adviser:
Paul HUDAK
paul.hudak@yale.edu

June 11, 2015

Abstract

Functional reactive programming facilitates programming with time-varying data that can be perceived as streams flowing through time. Thus, one can think of FRP as an inversion of flow control from the structure of the program to the structure of the data itself. In a typical (say, imperative) program, the structure of the program governs how the program will behave over time; as time moves forward, the program sequentially executes its statements, and at any line of code, one can make a clear distinction between code that has already been run (the past) and code that has yet to be run (the future). However, in FRP, the program acts as a signal function, and, as such, we are allowed to assume that the program executes *continuously* on its time-varying inputs—essentially, it behaves as if it is running infinitely fast and infinitely often. We consider this to be the core principle of the design and call it the *fundamental abstraction of FRP*.

Design and Performance

This work is specifically rooted in *Arrowized* FRP, where these signal functions remain static as they process the dynamic signals they act upon. However, in practice, it is often valuable to be able to dynamically alter the way that a signal function behaves over time. Typically, this is achieved with "switching" or other monadic features, but this significantly reduces the usefulness of the arrows.

We develop an extension to arrows to allow "predictably dynamic" behavior along with a notion of *settability*, which together recover the desired dynamic power. We further demonstrate that optimizations designed specifically for arrowized FRP and which do not apply to monadic FRP, such as those for Causal Commutative Arrows, are applicable to the system. Thus, it can be powerfully optimized.

Effectful FRP

In its purest form, functional reactive programming permits no side effects (e.g. mutation, state, interaction with the physical world), and as such, all effects must be performed outside of the FRP scope. In practice, this means that FRP programs must route input data streams to where they are internally used and likewise route output streams back out to the edge of the FRP context. I call this the FRP *I/O bottleneck*. This design inhibits modularity and also creates a security vulnerability whereby parent signal functions have complete access to their children's inputs and outputs. Allowing signal functions themselves to perform effects would alleviate this problem, but it can interfere with the fundamental abstraction.

We present the notion of *resource types* to address this issue and allow the fundamental abstraction to hold in the presence of effects. Resource types are phantom type parameters that are added to the type signatures of signal functions that indicate what effects those signal functions are performing and leverage the type-checker to prevent resource usage that would break the abstraction. We show type judgments and operational semantics for a resource-typed model as well as an implementation of the system in Haskell.

Asynchronous FRP

FRP typically relies on a notion of *synchrony*, or the idea that all streams of data are synchronized across time. In fact, this synchrony is a key component of maintaining the fundamental abstraction as it ensures that two disparate portions of the program will receive the same deterministically associated (synchronous) input values and that their separate results will coordinate in the same output values. However, in many applications, this synchrony is too strong.

We discuss a notion of treating time not as a global constant that governs the entire program uniformly, but rather as *relative to a given process*. In one process, time will appear to progress at one rate, but in another, time can proceed differently. Although we forfeit the global impact of the fundamental abstraction, this allows us to retain its effects on a per-process scale. That is, we can assume each process processes its inputs continuously despite the whole network having different notions of time.

To allow communication between these asynchronous processes, we introduce *wormholes*, which act as specialized connections that apply a sort of *time dilation* to information passing through them. We additionally show that they can be used to subsume other common FRP operations such as looping and causality.

Application

We apply the concepts of all of these ideas into a functional reactive library for graphical user interfaces called UISF. Thus, this work concludes with an overview and examples of practically using our version of FRP.

Acknowledgments

I am very grateful to my late adviser, Paul Hudak. Paul was an inspiring mentor whose enthusiasm for functional programming and well-designed systems had a strong impact on my academic career. I thank Paul for welcoming me into his research group and giving me the opportunity to work with him. Our discussions were formative for me and were instrumental in the preparation of this dissertation.

I am also indebted to Zhong Shao, who has welcomed me as a member of his group for years and who recently stepped in as my adviser for the end of my PhD.

I owe thanks to my committee members, Henrik Nilsson, Bryan Ford, and Ruzica Piskac, for their guidance and feedback on my work.

I would like to thank Shriram Krishnamurthi for introducing me to the idea of studying programming languages in the first place. I took his inspirational course as an undergraduate, and it changed my life. He has additionally been a reliable and available mentor, guide, and friend.

I am grateful to my colleagues, collaborators, and friends with whom I have had many long conversations. Notably, I must thank Shu-Chun Weng, who at this point probably knows the technical details of my research as well as I do and has helped me time and again rework them into a better state.

I am deeply grateful to my family for their love and support. They have encouraged, congratulated, and consoled me throughout my PhD work, and I cannot thank them enough.

Finally, I owe great thanks to Jane for her love and patience during this long process.

Computer Science Department

Dissertation Defense

Name: **Daniel Winograd-Cort**

Date of Defense: June 11, 2015

Title of Dissertation:

Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming

Advisor: Paul Hudak & Zhong Shao

Committee:

Name	Signature
Zhong Shao	
Bryan Ford	
Ruzica Piskac	Ruzica Piskac
Henrik Nilsson	Henrik Nilsson

Comments:

Passed: ✓ Failed: ___

Contents

Abstract i

Acknowledgements iii

Table of Contents v

List of Figures vii

1 Introduction 1
- 1.1 To Switch or Not To Switch . . . 2
- 1.2 Including Effects . . . 4
- 1.3 To Asynchrony and Beyond . . . 7
- 1.4 More Uses for Wormholes . . . 10

2 Background 11
- 2.1 Arrows . . . 11
- 2.2 Basic Language . . . 14

3 Choice and Settability 16
- 3.1 A Case for Non-Interfering Choice . . . 16
- 3.2 A Case for Settability . . . 22
- 3.3 An Alternative to *pSwitch* . . . 26
- 3.4 Implementing Settability . . . 29
- 3.5 Optimizations . . . 35
- 3.6 Other effects of switching from switch . . . 38

4 General Effects in FRP 40
- 4.1 Resource Types . . . 40
- 4.2 A Resource Typed Language . . . 44
- 4.3 Examples . . . 47
- 4.4 Delay and Loop . . . 51
- 4.5 Semantics . . . 53
- 4.6 Safety . . . 58
- 4.7 Haskell Implementation . . . 59

5 Asynchronous Functional Reactive Processes 67
- 5.1 Considering Asynchrony . . . 67
- 5.2 Motivating Examples . . . 69

5.3 The Language . . . 72
5.4 Concurrency Operators . . . 79
5.5 Language Properties . . . 80
5.6 Blocking . . . 82
5.7 Haskell Implementation . . . 83

6 UISF – A Case Study **88**
6.1 Arrowized User Interface . . . 88
6.2 Example: Time . . . 94
6.3 Example: Bidirectional Data Flow . . . 95
6.4 Example: Dynamically Active Widgets . . . 96
6.5 Example: Asynchronous Computation . . . 97
6.6 Differences From Theory . . . 99
6.7 Conclusions and Discussion of Similar Libraries . . . 100

Bibliography **102**

Appendix **107**
A.1 Proof That Non-Interference Implies Commutativity (and Exchange) . . . 107
A.2 Choice-Based Implementations of First-Order Switch . . . 107
A.3 Proofs of Preservation and Progress for Synchronous Semantics . . . 108
A.4 CFRP Properties . . . 111

List of Figures

2.1 The types of the arrow operators. . . . 12
2.2 The abstract syntax of $\mathscr{L}\{\rightarrow\times+\}$. . . . 15

3.1 The Arrow-choice Laws 18
3.2 The implementation of *runNTimes* 19
3.3 The implementation of *runDynamic* 20
3.4 The *settable* function and its laws. 24
3.5 The implementation of a drawing GUI 26
3.6 The drawing GUI using pswitch 27
3.7 Settability Transformations for various arrow combinators 30
3.8 Settability Transformation for RSwitch 32
3.9 The *State* data type and its two accessor functions. 33
3.10 *SA* implementations of the Arrow class functions. 34
3.11 Non-Interfering Arrow-Choice CCA Performance Results 38

4.1 The typing rules for arrow operators with resource types. 41
4.2 The resource type abstract syntax additions to $\mathscr{L}\{\rightarrow\times+\}$. . . . 45
4.3 Typing rules for resource typed expressions 46
4.4 The evaluation transition judgments for our extension to $\mathscr{L}\{\rightarrow\times+\}$. . . . 54
4.5 The functional transition judgments. 55
4.6 The temporal transition. 57
4.7 The Union type class. 61
4.8 Type Classes for Disjoint Union 62
4.9 SRemove Type Class for Set Removal 62
4.10 The Haskell Arrow classes redefined to permit resource types 63

5.1 The CFRP extension to $\mathscr{L}\{\rightarrow\times+\}$. . . . 72
5.2 CFRP Typing rules 73
5.3 The stack frames for the functional transition. 75
5.4 Functional Transition judgments 1 76
5.5 Functional Transition judgments 2 77
5.6 Functional Transition judgments 3 77
5.7 The frame application operator $\bowtie$ 78
5.8 The *buffer* function. 80
5.9 The *spar* function. 81
5.10 A potential definition of *brsf*. 83
5.11 The updated definitions of wormhole resources for CFRP. 84
5.12 The definition of *runSF* that can handle asynchrony. 86

6.1 UISF graphical input/output widgets . . . 89
6.2 UISF Mediators between continuous and discrete . . . 90
6.3 UISF Delays . . . 92
6.4 The Timer GUI. . . . 95
6.5 A screenshot of the Timer GUI. . . . 95
6.6 The Temperature Converter GUI. . . . 96
6.7 A screenshot of the Temperature Converter GUI. . . . 96
6.8 The Mind Map GUI. . . . 97
6.9 A screenshot of the Mind Map GUI. . . . 97
6.10 The compound widget for building a Pinochle hand. . . . 98
6.11 The Pinochle GUI. . . . 98
6.12 A screenshot of the Pinochle GUI. . . . 99

A.1 The implementation of *pChoice*. . . . 109

Chapter 1

Introduction

Functional Reactive Programming (FRP) is based on the idea of programming with *signals*, or time-varying values. Signals can be continuous, in which case they are defined for every moment in time, or they can be discrete event streams, in which case they are defined only at particular moments. Indeed, the signal is the fundamental and primary component of FRP, and the core purpose of FRP is that it provides a denotational semantics for signals as functions defined over time:

$$\mathit{Signal}\ \alpha = \mathit{Time} \to \alpha$$

Thus, FRP is designed to allow one to easily define behaviors for streams that react as the streams change over time.

This idea of using continuous modeling for dynamic, reactive behavior (that is, FRP) is due to Elliott, beginning with early work on TBAG, a C++ based model for animation [Elliott et al., 1994]. Subsequent work on Fran ("functional reactive animation") embedded the ideas in Haskell [Elliott and Hudak, 1997, Hudak, 2000], and more recently, there have been a plethora of FRP models. Because of its inherent signal-based design, FRP is a natural choice for real-time and continuous programming. Indeed, it has been used in such realms as animation, robotics, GUI design, networking, and audio processing, among others.

Due to its time-varying nature, one can think of FRP as a system that shifts the control flow from the structure of the program to the structure of the data itself. In a typical (say, imperative) program, the structure of the program itself governs how it will behave over time; as time moves forward, the program sequentially executes its statements, and at any line of code, one can make a clear distinction between code that has already been run (the past) and code that has yet to be run (the future). However, in FRP, the program acts as a signal transformer, and, as such, we are allowed to assume that the program executes *continuously* on its time-varying inputs—essentially, it behaves as if it is running infinitely fast and infinitely often. This design structure allows one to think of all data in the program as being *synchronized* in time, an idea consistent with the family of synchronous languages such as Lustre [Caspi et al., 1987], Esterel [Berry and Cosserat, 1984], and Signal [Gautier et al., 1987]. We consider this to be the core principle of the design and thus declare:

Key Principle (Fundamental Abstraction of FRP)**.** *A functional reactive program must be perceived to run simultaneously and continuously, and it is in the data itself that one can examine the past, present, and future.*

FRP systems that follow this abstraction have a number of practical, advantageous features. First, if the program has no sense of time, then it cannot describe sequential computation, which means that any effects that the program performs will be able to freely commute with each other within a given moment in time. Second, common temporal pitfalls that programmers fall into, such as race conditions and deadlocks, cannot occur. Lastly, if the program is deterministic on individual inputs, then its behavior over time will be deterministic as well.

A problem with classic FRP systems (such as Fran [Elliott and Hudak, 1997]) is their propensity toward space and time leaks [Liu and Hudak, 2007]. One method for addressing these leaks is by using *arrows* [Hughes, 2000, Lindley et al., 2010] in so called *arrowized* FRP (AFRP), which has been used in *Yampa* [Nilsson et al., 2002, Hudak et al., 2003, Courtney et al., 2003] (for animation, robotics, GUI design, and more), *Nettle* [Voellmy and Hudak, 2011] (for networking), and *Euterpea* [Hudak, 2014] (for audio processing and sound synthesis). In AFRP, instead of treating the signal as a first class value, one treats the *signal function* as the core component:

$$\alpha \rightsquigarrow \beta = \mathit{Signal}\ \alpha \rightarrow \mathit{Signal}\ \beta$$

The arrow structure then allows the signal functions to be composed quite naturally.

Furthermore, the arrow abstraction lends itself well to aggressive optimizations. An arrow's structure must be defined statically, and once defined, it cannot be altered mid-computation. Therefore, regardless of what data the signals contain, the arrow's overall behavior is fixed. Liu et al. [2011] use this restriction to design an optimization for a certain class of arrows, namely *causal commutative arrows* (CCA), which can often improve their performance in Haskell (using GHC) by an order of magnitude.

1.1 To Switch or Not To Switch

One problem with AFRP is that it does not naturally support the full range of capabilities that classic FRP provides. As mentioned, an arrow's structure must be fixed at compile-time, but classic FRP typically provides behavior-switching mechanisms. Thus, arrows are often augmented with a higher-order *switch* operator to recover this ability.

1.1.1 Switch

After the arrow framework was proposed by Hughes [2000], it was quickly adopted for use in FRP in the GUI language Fruit [Courtney and Elliott, 2001a, Courtney, 2004], which also introduced the first switch function (before then, higher-order signals were dealt with by a function typically called *until*). The design of Yampa [Nilsson et al., 2002, Hudak et al., 2003, Courtney et al., 2003] built off of this and expanded the idea to a variety of uses, eventually introducing fourteen different flavors of switch operators.[1]

Switching allows a program to accept and utilize a stream of signal functions, thus allowing for higher-order signal function expression in which the program can update its own structure during execution. Additionally, in the realm of signal functions, a higher-order ability like this provides the only means of starting and stopping signals mid-computation, which is often necessary for good performance. For instance, new signal functions can be provided at runtime and "switched on" to augment the current behavior of a program. Likewise, given an event that a certain signal is no longer needed, the program can "switch off" the portion of itself that is computing values for that signal, thus preventing unneeded computations from being performed. This ability to switch is very strong, and in fact, arrows with switch are equivalent to *ArrowApply* arrows, which themselves are equivalent to monads [Hughes, 2000].

Unfortunately, this power comes at a cost: the inherent higher-order nature of switch that allows it to run arbitrary signal functions from a stream makes certain compile-time optimizations and static guarantees much more difficult or even impossible. For example, arrows with switch cannot undergo the CCA optimizations. Likewise, in the realm of embedded systems, where static code is required due to strict time and resource constraints, switch can be an intolerable hole in a static guarantee.

[1] These fourteen operators were mostly convenience functions built atop a few primitive switchers, but they serve as an indication of the widespread use of switching.

1.1.2 An Alternative to Switch

One motivation of this research is to ask whether switch is really necessary. Most FRP programmers would be reluctant to give it up—indeed, some FRP programs would be inexpressible with just first-order arrows—but perhaps there is an operator that is powerful enough to replace switch in most cases while still being weak enough to allow for CCA-like optimizations. In order to consider this, we first must examine more closely exactly what switching provides.

Switch allows one to express two fundamental behaviors that are otherwise impossible with just arrows. First, it provides a way for signal functions to dynamically start and stop mid-computation, which is useful not just for expressing certain programs but also for achieving better performance. Second, it allows for higher-order signal expression, essentially providing a way to flatten a stream of streams into a single stream or insert a dynamic signal function into the arrow structure itself.

The second of these effects is impossible to replicate in classic (non-switchable) arrows, but there is some hope for the first. The ability to choose between whether to run a signal function or not is similar to what is provided by *arrow choice* [Hughes, 2000]. Arrows extended with choice can make a dynamic decision of how to process streaming data with the limitation that the possible choices must be statically defined. An additional difference lies in the fact that every effect from each possible choice will be executed regardless of the dynamic decision. This means that arrow choice cannot be used to entirely suspend a branch in the way that switch can suspend a "switched out" signal function because even effects from inactive branches will happen.

To address this, we can modify arrow choice by adding a new law in order to make it *non-interfering*. Non-interfering choice asserts that effects from only one branch of the choice will happen, and so if one branch is taken, it is as if the other does not exist.

Technically, non-interfering choice allows us only to pause signal functions and not actually start or stop them. For this reason, we additionally provide a method for making an arrow *settable*: a settable arrow's state can be saved, reloaded, and even reset.

Combining settability with non-interfering choice gives us the full power of the first effect of switch. That is, we can "start" a signal function by using choice and then resetting its state, and we can "stop" a signal function by indefinitely pausing it.

Interestingly, non-interfering choice allows for another unforeseen benefit: arrowized recursion. Because only one branch's effects can take place, we can do a form of recursion that allows behaviors that were previously only possible with switch. Combining this with settability allows for some surprising power.

1.1.3 Other Alternatives

Non-interfering choice and settability are not the only methods for trying to deal with the problems with switch. Patai [2011] presents an approach of embracing the higher-order mentality and shows a method for dealing with higher-order streams directly and efficiently using a monadic interface. In this way, switching becomes a core design property. Krishnaswami and Benton [2011] have a similar approach trying to bridge the divide between a synchronous, imperative style and an FRP-like, declarative style. Their work has a strong theoretical basis for handling causality when dealing with higher-order signals.

Another option is to allow switch in its normal form but then pursue other avenues of optimization. For instance Reactive [Elliott, 2009] and Elm [Czaplicki and Chong, 2013] focus on avoiding needless computation by using a "push" based design, which only recomputes values when changes are detected. Reactive additionally uses deterministic concurrency for even better performance.

From a different perspective, one can think of the ideas of settability and non-interfering choice not as a way to recover behavior after removing switching from an FRP language but instead as a way to add expressive power to a more traditional synchronous dataflow language. That is, there are plenty of FRP-like

languages that do not have the reactive power of switch, and these features can be used to extend them.

For instance, Esterel [Berry and Cosserat, 1984] is an imperative reactive language that allows programmers to create deterministic, synchronous control systems. It allows both parallel and sequential composition, which makes it suitable for many complex systems, but it has no concept of switching. Lustre [Caspi et al., 1987] and Signal [Gautier et al., 1987] are comparable.

Synchronous dataflow languages are often useful for describing hardware, which itself is deterministic and synchronous. Furthermore, hardware cannot support higher order signals for the obvious reason that hardware wires cannot themselves carry hardware operations. However, settability and non-interfering choice are both theoretically applicable to hardware domains.

1.2 Including Effects

An FRP program is still a pure functional program. That is, the signal-based computations are performed using pure functions, and the input and output of the program—which may include I/O commands—are handled separately, i.e. outside of the program. In this sense, there is an *I/O bottleneck* on either end of any signal function that represents a complete program. All of the input data must be separated from its source so that it can be fed purely into the appropriate signal function, and all of the output data must be separately piped to the proper output devices. We see this as an imperfect system, as ideally the sources and sinks would be directly connected to their data.

1.2.1 General Side Effects

A purely functional language does not admit side effects. Indeed, the original Haskell Report (Version 1.0) released in 1990, as well as the more widely publicized Version 1.2 [Hudak et al., 1992] specified a pure language, and the I/O system was defined in terms of both streams and continuations, which are equivalent (one can be defined straightforwardly in terms of the other). In 1989 the use of monads to capture abstract computations was suggested by Moggi [1989], subsequently introduced into Haskell by Wadler [1992], and further popularized by Peyton Jones and Wadler [1993].

Originally conceived as a pure algebraic structure, and captured elegantly using Haskell's type classes, it was soon apparent that monads could be used for I/O and other kinds of side effects. Indeed, Version 1.3 of Haskell, released in 1996, specifies a monadic I/O system. The inherent data dependencies induced by the operators in the monad type class provide a way to sequence I/O actions in a predictable, deterministic manner (often called "single threaded"). The Haskell I/O monad is simply named *IO*, and primitive I/O operations are defined with this monadic type to allow essentially any kind of I/O. A monadic action that returns a value of type a has type $IO\ a$.

To make this approach sound, a program engaged in I/O must have type $IO\ a$, and there can be no function, say $runIO :: IO\ a \to a$, that allows one to "escape" from the I/O monad. It's easy to see why this would be unsound. Consider the expression:

$$runIO\ m_1 + runIO\ m_2$$

If both m_1 and m_2 produce I/O actions, then it is not clear in which order the I/O actions will occur, since a pure language does not normally express an order of evaluation for $(+)$, and in general we would like $(+)$ to be commutative.

I/O is, of course, just one form of effect. For example, one might want to have mutable arrays (meaning that updates can be done "in-place" in constant time). A purely functional approach cannot provide constant-time performance for both reads and writes. Haskell has two solutions to this problem: First, Haskell defines an *IOArray* that can be allocated and manipulated in an imperative style. Predefined operations on the array

are defined in terms of the *IO* monad, and thus manipulating a mutable array becomes part of the single-threaded flow of control induced by the *IO* monad, as discussed earlier.

A problem with this approach is that it is common to want to define some local computation using an array and hide the details of how the array is implemented. Requiring that each such local computation inject the array allocation and subsequent mutations into the global I/O stream is thus not modular, and seems unnatural and restrictive.

What we would like is a monad within which we can allocate and manipulate mutable arrays (but not perform any I/O), and then "escape" from that monad with some desired result. Haskell's *ST* monad [Launchbury and Peyton Jones, 1994] does just that. Haskell further defines a type constructor *STArray* that can be used to define arrays that can be allocated and manipulated just like an *IOArray*. Once the programmer is done with the local computation, the *ST* monad can be escaped using the function:

$$runST :: (\mathsf{forall}\ s.\ ST\ s\ a) \rightarrow a$$

The "trick" that makes this sound is the use of the existential (phantom) type variable *s* within the *ST* monad and the operations defined on the arrays. For example, returning the value of an array reference would be unsound—it would mean that the mutable array could be further mutated in other contexts, with potentially unpredictable results. However, this is not possible in Haskell's *ST* monad, because the type of the array reference contains the hidden existential type, thus resulting in a type error.

1.2.2 Effects in FRP

Monads can be used for many pure computations as well as other kinds of effects, but the above has focused on two kinds of effects: I/O and mutable data structures. It is important to distinguish these two, since there are inherent conceptual differences. Mutable data structures can be created and allocated dynamically as required by the program. Because they have no external or observable effects, two different data structures can be guaranteed to be distinct, and we are only limited in their use by the bounds of the system's memory. In contrast, I/O devices are generally fixed—each printer, monitor, mouse, database, MIDI device, and so on, is a unique physical device—and they cannot be created on the fly. Although one could allocate multiple virtual instances of any given device, they would all eventually be mapped to the same physical device.

It is also worth noting that for both I/O devices and mutable data structures, the sequence of actions performed on each of them must generally be ordered, as it would be in an imperative language, but conceptually, at least, distinct actions on a printer, a MIDI device, or some number of separately allocated mutable data structures, could be performed concurrently.

So the question now is, how do we introduce these kinds of effects into FRP? Indeed, do these kinds of effects even make sense in an FRP language? Without effects, FRP has limited power and a constrictive design, but so far, work has only been on either the programmer interface level [Courtney and Elliott, 2001b] or the system's underlying connection to imperative-style effects libraries [Cooper and Krishnamurthi, 2006]. Can we bridge the gap between these by providing arbitrary effect usage directly to the front interface in a clear and safe way?

A normal Haskell variable is time-invariant, meaning that its value in a particular lexical context and in a particular invocation of a function that contains it, is fixed. In a language based on FRP, variables can be conceptually time-varying—their values in a particular lexical context and in a particular invocation of a function that contains them are not fixed, but rather depend on time.

A key insight is that the sequencing provided by a monad can be achieved in FRP by using the ordering of events in an event stream. In the case of I/O, another key insight is that each of the I/O devices can be viewed as a signal function that is a *virtualized* version of that device. We can guarantee the soundness of this at the type level by introducing *resource types*.

1.2.3 Safely Virtualizing Resources

Virtualizing a real world object or device is simply the concept of viewing that object as a piece of the program, or in FRP, as a signal function itself. For example, the console's input produces events with string values, the console's output takes events of strings as input, and a MIDI keyboard could take note events as input as well as generate note events as output. So it would seem natural to simply include these devices as part of the program in the form of signal functions—i.e. to program with them directly and independently rather than merge everything together as one input and one output for the whole program. In this sense, the real-world objects are being *virtualized* for use in the program.

The only problem is that one could easily duplicate these virtualized objects since after all, once virtualized, they are just values. This would cause the semantics of the program to become unclear. For example, a virtualized object may be duplicated such that each instance is provided with a different input event stream, but the object itself expects only one input stream—what do we do? We could non-deterministically merge the streams, but this seems imprecise and may not be the desired behavior. Really, we want to ensure that each of these virtualized devices is unique to the program. This seems difficult to achieve until we recognize that *uniqueness of signal functions can be realized at the type level*. In particular, we introduce the notion of a *resource type* to ensure that there is exactly one signal function that represents each real-world device. Because we are using arrows, we begin by re-typing the arrows themselves to include resource types, and then we introduce type families and classes that capture the idea of a disjoint unions of these resource types and update the arrow combinators to use them. For example, the keyboard could be virtualized into a signal function that produces keystroke events. We expect that every keystroke should produce a unique event, but if this signal function were duplicated, we can no longer easily guarantee that claim. Thus, we attach a resource type to this signal function that will propagate throughout the entire program upon composition and then restrict the program to allow only one *Keyboard* resource type. If a programmer attempts to use the signal function more than once in the same program, the resource type will appear twice, and the program will produce a type error.

Resource types share similarities to other type-and-effect systems. For instance, the language *Clean* [Brus et al., 1987, Plasmeijer and van Eekelen, 2002] has a notion of *uniqueness types*. In Clean, when an I/O operation is performed on a device, a value is returned that represents a new instantiation of that device; this value, in turn, must be threaded as an argument to the next I/O operation, and so on. This single-threadedness can also be tackled using *linear logic* [Girard, 1987], and various authors have proposed language extensions to incorporate linear types [Wadler, 1991, Hawblitzel, 2005, Tov and Pucella, 2011, Wadler, 1990]. In contrast, resource types are not concerned with single-threadedness since there is only one signal function to represent any particular I/O device. Rather, their purpose is to ensure that resources do not conflict. Additionally, they are specialized to handle FRP.

Resource types achieve their safety benefits by taking advantage of the temporal nature of FRP. That is, because of the fundamental abstraction of FRP, computations within single ticks of the clock cannot share the same resource, but a computation that uses a given resource will monopolize that resource for all clock ticks. Another way to constrain the temporal behavior of reactive programs is through linear-time temporal logic (LTL) [Jeffrey, 2012, Jeltsch, 2012] and the Curry-Howard correspondence between it and FRP. Indeed, Jeffrey [2012] lays out the basis for an implementation of a constructive LTL in a dependently typed language such that reactive programs form proofs of LTL properties.

The advantages of resource types include:

1. *Virtualization*. I/O devices can be treated conveniently and independently as signal functions that are just like any other signal function in a program. I/O is no longer a special case in the language design.

2. *Transparency*. From the type of a signal function, we can determine immediately *all* of the resources that it uses. In particular, this means that we know all the resources that an entire program uses (as

opposed to with the *IO* monad, where all we know is that some kind of I/O is being performed).

3. *Safety*. If used properly, a signal function engaged in I/O is *safe*—despite the side effects, equational reasoning is not compromised.

4. *Extensibility*. A user can add new resource types to the system that capture new kinds of effects or that represent new I/O devices.

1.2.4 Wormholes

In the realm of mutable data structures, we seem to come to a slightly different conclusion. We can start with a similar approach of lifting the interaction from individual actions to a signal function; for example, we could define:

$$sfArray :: Size \to (Event\ Request \rightsquigarrow Event\ Response)$$

such that *sfArray n* is a signal function encapsulating a mutable array of size *n*. That signal function would take as input a stream of *Request* events (such as read or write) and return a stream of *Response* events (such as the value returned by a read, acknowledgement of a successful write, or an index-out-of-bounds error). Note the similarity of this approach to the original stream I/O design in early Haskell [Hudak et al., 1992].

This design is also analogous to the *STArray* design, in that in-place updates of the array are possible in a sound way, and every invocation of *sfArray* creates a new mutable array. The difference between this and both the *STArray* design as well as the virtualized resources of the previous subsection is that no changes to the type system are required to ensure soundness (in particular, no hidden existential types are needed, nor are resource types). Using this idea, many kinds of mutable data structures are possible, as well as certain kinds of duplicable effects, for example, random number generation. These types of effects, being inherently local or duplicable, are readily available in other FRP formulations.

However, although functionally sound, this design is somewhat unsatisfying in that the requests and responses both need to be co-located. That is, these signal functions that represent mutable data structure are inherently complicated by the fact that they have both inputs and outputs all at once.

Thus, we next ask: What happens when we split the functionality into two separate signal functions, one for providing data and the other for producing output? In the simplest case, the data structure itself could simply be a single mutable data cell, but by splitting it into two components, it turns into a method for communicating data between otherwise unconnected parts of a program. We refer to the receiving signal function as the *blackhole*, the producing signal function as the *whitehole*, and the two together as a *wormhole*. By analogy, wormholes are a bit like *IORef*s in Haskell: one signal function provides the effect of writing and the other reading, but in the FRP framework, the details are considerably different.

It is notable that because a wormhole is no longer a single entity, we can no longer clearly distinguish two wormholes merely by them being invoked in different places. In fact, we must even face the question of what it may mean if two *of the same* whiteholes or blackholes are used in the same program. However, these questions lead to the same place they did when we were exploring virtualizing resources, and we resolve them in the same way as well: by using resource types. Upon constructing a wormhole, two fresh, virtual resources must be created that are then associated with the whitehole and blackhole of the wormhole.

At this point, wormholes may seem like something of a novelty—indeed, we seem to have built them solely to see if we can—but as we shall see, they have a variety of practical applications.

1.3 To Asynchrony and Beyond

As mentioned, FRP creates a synchronous model of programming, or one in which time cannot affect any portion of the program (or its data) without affecting its entirety. In this realm then, we can think of

asynchronous computation as another form of effect. For instance, perhaps we have a computation that runs unpredictably longer or shorter than others, and we would like to let it run freely. In another case, we may simply want two unrelated tasks to run separately, free of needing to synchronize with each other on any particular schedule.

This "asynchronous effect" seems fundamentally different than the effects discussed previously. Rather than dealing with how to achieve a clear ordering of events in time, we instead need to consider the nature of time itself, which should immediately give us pause. If an FRP program is expected to uphold the fundamental abstraction of FRP, that it is synchronous and can process instantaneous values infinitely fast, then what is the point of asynchrony? If we are using this abstraction so that we can assume that continuous signals behave continuously, then what does it mean for one to take longer than another? Thus, it appears that even attempting to address asynchrony in FRP will destroy the main reason to use FRP at all.

However, we need not lose all hope. Rather than think of time as a global constant that governs the entire program uniformly, we borrow an idea from physics and think of time as *relative*. In one process, time will appear to progress at one rate, but in another, time can move differently. Although we lose the global impact of the fundamental abstraction, this allows us to retain its effects on a per-process scale. That is, we can assume each process processes its inputs continuously despite the whole network having different notions of time.

1.3.1 Communicating Functional Reactive Processes

All that remains is a way for asynchronous processes to communicate as necessary. Because of the time difference between processes, this information must somehow be transformed as it moves from one "time stream" to another. With a proper design, a wormhole can be made to do exactly this[2].

The solution is to build the wormhole over a data structure that can allow for compression or expansion of the underlying signal, essentially allowing for the time dilation that may occur between two different signal rates. For instance, if a signal is sent from a fast process to a slow process, then the signal will appear sped up, or at a higher frequency on the receiving end compared to how it was submitted on the sending end.

These time dilating wormholes allow us to effectively create a new language of communicating functional reactive processes (CFRP) that can add asynchrony to FRP while retaining the fundamental FRP abstraction on a per-process scale. Thus, CFRP includes an operator to allow the creation of a new process with its own notion of time[3], and it uses the concept of wormholes to create bridges between those processes.

1.3.2 General Parallelism

Although we introduced the asynchronous effect to provide new means of expression, it provides a clear model to allow signal functions to run in parallel, thus opening a new opportunity for performance optimization. Indeed, the design allows us to take advantage of multi-core architectures, letting each functional reactive process proceed through time on its own core.

Thus, along with our future discussion of asynchrony, we will present a number of high-level parallel and concurrency operators that are built using the simple ideas of asynchrony and wormholes.

[2]The name "wormhole" may make more sense now, as it is a reference to the theoretical astronomical oddity, the "Einstein-Rosen bridge," a one-directional path through space-time such that matter can only flow in through the black hole and out through the white hole and that has the capacity to permit certain forms of time travel.

[3]In the physics-based space-time model, one could think of this as an operator that causes a new big bang or spawns a new universe.

1.3.3 Other Efforts

While our work considers communicating *functional reactive* processes, the seminal work on communicating *sequential* processes is [Hoare, 1978, Milner, 1982, 1999]. Our ideas are similar, but obviously different based on the domain.

In presenting CFRP, we will provide a full set of statics and dynamics to describe its functionality. However, there are other models that attempt to describe the behaviors of concurrent or asynchronous programs. For instance, the π-calculus [Milner, 1993] provides a model to describe concurrent programs through the use of name generation and sharing via channels. Our asynchronous use of resource types is similar, but while names can be sent through channels, resources cannot be sent through wormholes. This restriction allows us to maintain the fundamental abstraction of FRP by forcing all resources to be race-free.

Concurrency in functional languages has been explored previously, most notably in Concurrent ML [Reppy, 1993], concurrent and parallel Haskell [Jones et al., 1996, Li and Zdancewic, 2006, Jones and Hudak, 1993], and Erlang [Virding et al., 1996], among others.

The term *serializability* [Papadimitriou, 1979] typically refers to the idea that a parallel execution of a set of transactions over multiple items is equivalent to *some* serial execution, or in other words, that one can find a total ordering of transactions. The idea of *linearizability* [Herlihy and Wing, 1990] is that updates to an object can be thought of as acting instantaneously at some point during the update operation. Both of these notions are relevant to CFRP due to the fact that wormholes must be built atop data structures that are both serializable and linearizable.

Although our time dilating wormholes are a novel concept, using non-local communication channels to facilitate data transfer between multiple threads has been explored previously, as in teleport messaging [Thies et al., 2005]. However, teleport messaging is designed particularly for parallelized stream programs while CFRP is designed for asynchronous communication.

Reactive Concurrency

CFRP can be seen as an instance of a Globally Asynchronous Locally Synchronous (GALS) system [Chapiro, 1984]. Work on GALS systems tends to be in the realm of de-synchronizing synchronous programs to work on asynchronous architectures without sacrificing determinism or synchronous semantics [Sangiovanni-Vincentelli et al., 2000, Benveniste et al., 1999]. The design of CFRP was not led by architecture but rather by embracing the forms of computation introduced by asynchrony without allowing them to overwhelm the fundamental abstraction of FRP. The wormhole communication channel, which dilates data moving through it rather than being a typical FIFO queue, exemplifies this difference.

A related idea is that of synchronous programming with multiple clocks [Berry and Sentovich, 2001], which relies on the idea that clocks tick and that these ticks can be used as synchronization points. This idea of ticks forces an inherent discreteness which CFRP does not. Although CFRP cannot fully resynchronize two asynchronous processes, it can express continuous signals as well as discrete ones.

Another way to handle multiple clock rates is to use a clock calculus of multiple static clock rates with sampling between them as necessary, as first seen in Lustre [Caspi et al., 1987]. This has more recently been embedded in strongly typed functional languages (e.g. Euterpea [Hudak, 2014], Lucid Synchrone [Pouzet, 2006]) by leveraging the type system and type inference.

Parallel FRP [Peterson et al., 2000] allows concurrent signal processing by allowing multiple threads to perform the same function on a stream of inputs. The by-product of this design is that the ordering of events on that stream may not be preserved in the output stream. Although CFRP preserves the ordering of streams by default, we can achieve a similar non-preserving behavior in CFRP with event streams by asynchronizing functions and gathering their results when they are ready. Thus, if one event takes a long time to process, it will not hold up the rest of them.

Although not explicitly concurrent, Elliott [2009] presents an FRP implementation that makes use of

concurrency “under the hood” with an *unambiguous choice* operator. While this may enhance performance, it does not actually provide a method for asynchronous programming.

Elm [Czaplicki and Chong, 2013] is an asynchronous FRP language for creating GUIs. It provides built-in asynchronizing capabilities similar to ones that can be built in CFRP, but it does not provide access to the lower operators (e.g. wormholes, etc.) that would enable users to build their own custom concurrency operators.

1.4 More Uses for Wormholes

Wormholes will be discussed in much more detail later, but it is worth pointing out that they have many uses beyond being a means of non-local communication between asynchronous processes. To consider this, we will look at how wormholes must behave when both the whitehole and blackhole are in the same process.

As mentioned, a wormhole applies a time dilation over its data. The primary use for this is to allow data to be converted between two time streams as it is communicated between processes, but the dilation will occur even if both ends of the wormhole are in the same process. In this case, the dilation will appear as a *unit delay*. In a discrete-time context, this would be a delay of the smallest unit of time, and in a continuous model, it would be an infinitesimal delay, or a delay of change in time as that change approaches 0. This means that if the whitehole and blackhole are composed together in sequence, the resulting structure will act as a *delay* operator. One implication of this is that wormholes are strictly causal entities, in which the output of the whitehole is based on inputs to the blackhole that are strictly from the past.

More interestingly, one can consider the result of composing the blackhole to the whitehole. Of course, this would seemingly be a vacuous signal function, but with a suitable signal function between the two, we create a simple, causal feedback loop. Indeed, with wormholes in the language, one can typically forego the classic arrow looping mechanisms altogether in favor of the strictly causal looping of wormholes.

In total, this means that we can allow feedback and state within our signal functions while maintaining causality. Krishnaswami et al. [2012] also explore causality at the type level. They describe a language that uses non-arrowized FRP yet still manages to restrict space-leaks statically. This language seems somewhat more expressive than ours as it allows a more generic loop operator, but it is not clear whether it can be easily adapted to allow mutation or other side effects.

Chapter 2

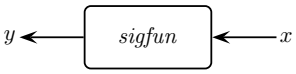
Background

2.1 Arrows

2.1.1 Signal Processing

Programming with AFRP is a lot like expressing signal processing diagrams. Where signal processing diagrams have lines, AFRP has *signals*, and where diagrams have boxes that act on those lines, AFRP has *signal functions*. These signals can represent either continuously-defined time-varying values or streams of discrete events.

Because AFRP is based on arrows, we can use Paterson's *arrow syntax* [Paterson, 2001] to make programming with it easier. For example, we can turn this simple signal processing diagram:



into just as simple a code snippet:

$$y \leftarrow \mathit{sigfun} \prec x$$

In this example, *sigfun* is a signal function that takes the input stream x and produces the output stream y.

We will use Haskell's arrow syntax and operators to express code examples. Thus, the above code fragment cannot appear alone, but instead must be part of a **proc** construct. The expression in the middle must be a signal function, whose type we write as $\alpha \rightsquigarrow \beta$ for some types α and β. The expression on the right may be any well-typed expression with type α, and the expression on the left must be a variable or pattern of type β.

The purpose of the arrow notation is to allow the programmer to manipulate the instantaneous values of the signals. For example, the following is a definition for *sigfun* that integrates a signal and adds one to the output:

$$\begin{aligned} \mathit{sigfun} = \mathbf{proc}\ x \rightarrow \mathbf{do} \\ y \leftarrow \mathit{integral} \prec x \\ \mathit{returnA} \prec y + 1 \end{aligned}$$

The notation "**proc** $x \rightarrow$ **do** ..." introduces a signal function, binding the name x to the instantaneous values of the input. The second line sends the input signal into an integrator, whose output is named y. Finally, we add one to the value and feed it into the signal function *returnA*, that returns the result. The last line of this notation has no binding component—instead, whatever value is produced in the last line is returned as the output stream.

$$
\begin{aligned}
\mathit{arr} &:: (\alpha \to \beta) \to (\alpha \rightsquigarrow \beta) \\
\mathit{first} &:: (\alpha \rightsquigarrow \beta) \to ((\alpha, \gamma) \rightsquigarrow (\beta, \gamma)) \\
(\ggg) &:: (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma) \\
(|||) &:: (\alpha \rightsquigarrow \gamma) \to (\beta \rightsquigarrow \gamma) \to ((\alpha + \beta) \rightsquigarrow \gamma) \\
\mathit{loop} &:: ((\gamma, \alpha) \rightsquigarrow (\gamma, \beta)) \to (\alpha \rightsquigarrow \beta) \\
\mathit{delay} &:: \beta \to (\beta \rightsquigarrow \beta)
\end{aligned}
$$

Figure 2.1: The types of the arrow operators.

Of course, one can use arrows without Haskell's arrow syntax. Arrows are made up of three basic operators: construction (*arr*), partial application (*first*), and composition ($\ggg$). Furthermore, we extend our arrows with choice ($|||$) [Hughes, 2000] to allow dynamic control flow, looping (*loop*) [Paterson, 2001] to allow value-level recursion, and delay (*delay*). The types of these operators are shown in Figure 2.1.

For example, the signal function *sigfun* defined earlier can be written without arrow syntax as follows:

$$\mathit{sigfun} = \mathit{integral} \ggg \mathit{arr}\ (\lambda y.\ y + 1)$$

Note that *returnA* is defined simply as *arr id*, which is why it is used for clarity to return values in the last line of arrow syntax but is omitted from the above definition of *sigfun*. In later chapters, we will also make use of the function $\mathit{constA} :: \beta \to (\alpha \rightsquigarrow \beta)$, which takes one static argument and returns a signal function that ignores its input stream and returns a constant stream of the given value.

Events and Event Streams

The classical interpretation of a signal of type α is that it is a function from time to α defined for all points in time. We call this a *continuous* signal. However, we frequently require the ability to define a signal that has values at only discrete points in time and is undefined elsewhere. These so-called *event streams* are represented by encapsulating the signal's type with an option type. We will use the following:

$$\textbf{data}\ \mathit{Event}\ \alpha = \mathit{Event}\ \alpha \mid \mathit{NoEvent}$$

Note that we are overloading the name *Event* such that it is both the general type as well as the constructor for an event. Thus, any signal that has type $\mathit{Event}\ \alpha$ is defined when it provides an *Event* and undefined when it provides *NoEvent*.

We will further make use of the fact that *Event* is a functor in the obvious way and freely *fmap* functions over *Event* values.

2.1.2 Strictly Causal Looping

Functional reactive programming itself does not need to be causal. That is, values along a signal can, in fact, depend on future values. Of course, in real-time systems, causality is forced to be preserved by the nature of the universe. For example, a program's current output cannot depend on a user's future input. Thus, in the world of effectful FRP, we limit ourselves to causal signal functions.

The main impact of this limitation has to do with fixed points and looping in the signal function domain. We restrict signal functions so that they cannot perform limitless recursion without moving forward in time. That is, all loops must contain a delay such that the input only depends on past outputs. We call this *strictly causal looping*.

We use the *delay* operator as an abstract form of causal computation[1]:

$$delay :: a \to (a \rightsquigarrow a)$$

Based solely on the type, the current output of *delay i* could depend on the previous, current, or even future inputs; however, the typical definition (and the one that we will use) is as a unit delay operator[2], and as such, the current output would depend on only the previous inputs. Used in tandem with the arrow *loop* operator from Figure 2.1, one can define strictly causal loops:

$$dLoop :: c \to ((c \times a) \rightsquigarrow (c \times b)) \to (a \rightsquigarrow b)$$

The *dLoop* operator takes an initial value for the looping parameter, which will update in time but always be slightly delayed. Notice that when *dLoop* is given the simple swapping function $(\lambda(x,y).(y,x))$ as its second argument, it reduces to an instance of the *delay* function acting as a unit delay.

2.1.3 State via loop and delay

A key component of FRP systems (AFRP included) is the ability to perform stateful computation. For example, Yampa includes the *integral* function that integrates its input signal, a process impossible without some form of internal state.

Although stateful signal functions can be achieved in a variety of ways, we follow Liu et al. [2011] in the use of the *delay* operator along with *loop* (or equivalently, the *dLoop* operator defined above). In this model, we use the loop as a feedback mechanism, allowing an auxiliary output containing the state to be fed back as an input, and we use the delay to prevent an infinite feedback loop. Indeed, Liu et al. [2011] even demonstrate that in a fixed rate, discrete time system, *integral* can be defined using this method:

$$\begin{aligned} integral = \mathbf{proc}\ x \to \mathbf{do} \\ \quad \mathbf{rec}\ v \leftarrow delay\ 0 \prec v + dt * x \\ \quad returnA \prec v \end{aligned}$$

Note here that the **rec** keyword in arrow syntax invokes the *loop* operator and that we assume *dt* is a global time step.

2.1.4 Switch

As discussed in the introduction, the ability to dynamically *switch* one signal function for another during the execution of a program is a staple of most FRP systems. Considering that one of our primary goals is to show an alternative to switching, here we will describe switch's capabilities.

The idea of switching was introduced along with the earliest models of FRP [Elliott and Hudak, 1997]. These non-arrowized FRP implementations had the ability to sequence periods of signal function execution, a process that is inherently monadic in nature. However, the move to the arrow abstraction would not allow this behavior, and to prevent any loss in expressiveness, Nilsson et al. [2002] introduced the *switch* function in Yampa.

Actually, Yampa includes some 14 different variations on the switch function ranging from the simplest switch to the recurring, parallel, batch-input, decoupled switch. We will briefly examine three of these switchers.

[1] Although the *delay* operator has been around for some time, Liu et al. [2011] introduced the concept of this operator as the basis of causal computation. That said, they referred to it as *init*.

[2] As mentioned in the introduction, we use the idea of a unit of time to refer to the smallest amount of time when in a discrete-time context and an infinitesimal delay in a continuous one.

Switch

The most basic switch function has the following type:

$$\begin{aligned} switch \;::\; & (\alpha \rightsquigarrow (\beta, Event\ \gamma)) \\ & \rightarrow (\gamma \rightarrow (\alpha \rightsquigarrow \beta)) \\ & \rightarrow (\alpha \rightsquigarrow \beta) \end{aligned}$$

The first argument is the initial signal function that the result will behave as. When that signal function produces an event, the switch will use the data from that event along with its second argument to produce a new signal function. From then on, it will behave as that new signal function.

Recurring Switch

A slightly more advanced version of switching allows for the signal function to be switched out more than once:

$$\begin{aligned} rSwitch \;::\; & (\alpha \rightsquigarrow \beta) \\ & \rightarrow ((\alpha, Event\ (\alpha \rightsquigarrow \beta)) \rightsquigarrow \beta) \end{aligned}$$

Here, the resulting signal function takes an event stream of signal functions along with the stream of input α values. When the event stream contains an event, it switches into the signal function contained in the event.

Parallel Switch

The parallel version of switch is significantly more intimidating from its type signature and likewise is also quite powerful:

$$\begin{aligned} pSwitch \;::\; & Functor\ col \\ & \Rightarrow col\ (\alpha \rightsquigarrow \beta) \\ & \rightarrow ((\alpha, col\ \beta) \rightsquigarrow Event\ \gamma) \\ & \rightarrow (col\ (\alpha \rightsquigarrow \beta) \rightarrow \gamma \rightarrow (\alpha \rightsquigarrow col\ \beta)) \\ & \rightarrow (\alpha \rightsquigarrow col\ \beta) \end{aligned}$$

The parallel switcher works on *collections* of signal functions, where a collection must be a *Functor* (perhaps a list). First, it is given an initial collection of signal functions to run and a signal function that produces update events. The third argument takes the current collection of signal functions and the value from an event in order to produce a new collection of signal functions. In total, *pSwitch* will run every signal function in its collection and produce as output a collection of their results.

Note that any one of these versions of switch is strong enough to implement the others. The reason for Yampa's many varieties of switch is not due to power differences, but rather due to ease of use. That is, for example, using *switch* to do an operation that requires *rSwitch* is tedious, so both varieties are provided.

2.2 Basic Language

In future chapters of this report, we will create new languages to demonstrate new features and design paradigms that we introduce. These languages all share a common basis, or ancestor language. We present this basic language here.

We specify our language in a similar manner to Lindley et al. [2010]. We start with the lambda calculus extended with product and sum types and general recursion, and when necessary, we will refer to it as $\mathscr{L}\{\rightarrow\times+\}$. We show the abstract syntax for this language in Figure 2.2. We let τs range over types, vs over variable names, es over expressions, and Γs over environments. A type judgment $\Gamma \vdash e :: \tau$ indicates that that it follows from the mappings in the environment Γ that expression e has type τ. Sums, products, and functions satisfy β- and η-laws. This is a well established language, so rather than repeat the typing rules,

Typ	τ	::=	$()$	unit
		|	$\tau_1 \times \tau_2$	binary product
		|	$\tau_1 + \tau_2$	binary sum
		|	$\tau_1 \to \tau_2$	function
Var	v			
Exp	e	::=	v	variable
		|	(e_1, e_2)	pair
		|	*fst e*	left-pair projection
		|	*snd e*	right-pair projection
		|	*left e*	left-sum injection
		|	*right e*	right-sum injection
		|	$case(e; x_1.e_1; x_2.e_2)$	case analysis
		|	$\lambda v.e$	abstraction
		|	$e_1\ e_2$	application
Env	Γ	::=	$v_1 :: \tau_1, ..., v_n :: \tau_n$	type environment

Figure 2.2: The abstract syntax of $\mathscr{L}\{\to\times+\}$.

it suffices to say that they are as expected. We also borrow an expected operational semantics that utilizes lazy evaluation.

Note that the *Event* data type we defined earlier is equivalent to the type $\alpha + ()$, but we use the event notation for readability.

Chapter 3

Choice and Settability

3.1 A Case for Non-Interfering Choice

We will begin this section by exploring one of the main uses of switchers: as a method to allow the dynamic starting and stopping of signal functions. We will present our first-order alternative and then demonstrate it in a few practical settings.

3.1.1 Pausable Signal Functions

At a basic level, switch is often used to improve performance of an AFRP program. Without switch, signal functions will last forever, and this typically means that they will compute future values indefinitely. Using switch, one can "turn off" signal functions that are not currently necessary and even turn them back on if they are required again in the future.

For example, consider the scenario where we would like to integrate a stream only when a certain condition holds. Naïvely, we can write the following program:

$$
\begin{aligned}
&\mathit{integralWhen}_{\mathit{Naive}} :: (\mathit{Double}, \mathit{Bool}) \rightsquigarrow \mathit{Double} \\
&\mathit{integralWhen}_{\mathit{Naive}} = \mathbf{proc}\ (i, b) \rightarrow \mathbf{do} \\
&\quad v \leftarrow \mathit{integral} \prec i \\
&\quad v_{\mathit{prev}} \leftarrow \mathit{delay}\ 0 \prec v \\
&\quad \mathbf{let}\ v_\Delta = v - v_{\mathit{prev}} \\
&\quad \mathbf{rec}\ \mathit{result} \leftarrow \mathit{delay}\ 0 \prec \mathbf{if}\ b\ \mathbf{then}\ \mathit{result} + v_\Delta\ \mathbf{else}\ \mathit{result} \\
&\quad \mathit{returnA} \prec \mathit{result}
\end{aligned}
$$

This program will only update the result when the boolean is *True*, but it is still unsatisfying that the integral is being computed at all when it is not being used. If integral were instead a costly signal function and the boolean were usually *False*, this could be seriously problematic to performance.

In cases like this, switch can be employed to prevent the integral from running when it is not needed:

$$
\begin{aligned}
&\mathit{integralWhen}_{\mathit{Switch}} :: (\mathit{Double}, \mathit{Event\ Bool}) \rightsquigarrow \mathit{Double} \\
&\mathit{integralWhen}_{\mathit{Switch}} = \mathbf{proc}\ (i, e_b) \rightarrow \mathbf{do} \\
&\quad \mathbf{rec}\ v \leftarrow \mathit{rSwitch}\ (\mathit{constA}\ 0) \prec (i, \\
&\qquad\quad \mathit{fmap}\ (\lambda b \rightarrow \mathbf{if}\ b \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ (\mathit{integral} \ggg \mathit{arr}\ (+v)) \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \ (\mathit{constA}\ v))\ e_b) \\
&\quad \mathit{returnA} \prec v
\end{aligned}
$$

For this version, we modified the type to make it more amenable to switching by converting the streaming boolean value to an event stream that will send events only when the stream would change from *True* to *False* or back. Internally, we use the *rSwitch* function that we introduced in Section 2.1.4 to switch between *integral* and a constant function. Each time we switch into *integral*, it is fresh and has no history from the last time we were using *integral*, so we additionally compose it with *arr* $(+v)$ so it can maintain its history.

3.1.2 Non-interfering Choice

Although the above example is a fairly common use for switch, careful examination of the problem reveals that switch is far more powerful that necessary. That is, while switch allows us to dynamically incorporate new signal functions into the running computation, here, we are simply making a *choice* of whether to run a component signal function based on a dynamic value. Our solution to this problem will thus be built around arrow choice, so we will begin by examining it more closely.

The general choice operator we use ($|||$ in Figure 2.1) can actually be built from a simpler component:

$$left :: (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha + \gamma) \rightsquigarrow (\beta + \gamma))$$

where *left* f calls f when the input signal contains *Left* values and acts as the identity function otherwise. With the *left* function, we can also define an analogous *right* function and then use the two together to define $|||$.

Choice also comes with a set of laws that we show in Figure 3.1. For us, the most notable law is the *exchange* law, which acts as a weak form of commutativity between *left* functions and *right* functions. One may ask why choice does not demand full commutativity (i.e. *left* $f \ggg$ *right* $g =$ *right* $g \ggg$ *left* f), and in the context of signal processing, this question is very sensible. After all, it seems intuitively obvious that either the *left* function or the *right* function will run, but in no case will both run. However, because arrows can have effects regardless of their dynamic inputs, and the compositional order of these effects can alter the program itself, choice is weakened. It is precisely this leniency that makes switching necessary in cases such as the above example.

In order to give choice the extra power it needs to be an adequate replacement for switch, we strengthen the *exchange* law into the more powerful:

$$\text{Non-interference} \qquad arr\ Right \ggg left\ f = arr\ Right$$

Indeed, *non-interference* implies exchange and even commutativity as it is stronger than either (see Appendix A.1 for details). It states that once the streaming value is tagged as a *Right* value, then it will not be applicable to *left* f, and so it should behave as if the *left* f is not even there. Thus, by including the non-interference law for choice, we assert that either signal functions cannot have static effects or that the choice operation has the power to dynamically choose which effects to perform.

We can see this in practice by considering a concrete example. Let's consider the case of the signal function:

$$left\ integral \ggg right\ integral$$

and we supply it with a signal that varies between *Left* 1.0 and *Right* 2.0 every second (that is, on the interval $[0.0, 1.0)$, the signal is *Left* 1.0, on $[1.0, 2.0)$, it is *Right* 1, and so on). Examining the output would reveal the following pattern:

$$\begin{aligned}
[0.0, 1.0) &: \quad Left\ 0.0 - Left\ 1.0 \\
[1.0, 2.0) &: Right\ 0.0 - Right\ 2.0 \\
[2.0, 3.0) &: \quad Left\ 1.0 - Left\ 2.0 \\
[3.0, 4.0) &: Right\ 2.0 - Right\ 4.0
\end{aligned}$$

In other words, when the "left" integral is inactive, it turns off, and behaves as if no time passes. The same is true for the "right" integral.

$$
\begin{array}{lrcl}
\text{Extension} & \mathit{left}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (\mathit{left}\ f) \\
\text{Functor} & \mathit{left}\ (f \ggg g) & = & \mathit{left}\ f \ggg \mathit{left}\ g \\
\text{Exchange} & \mathit{left}\ f \ggg \mathit{arr}\ (\mathit{right}\ g) & = & \mathit{arr}\ (\mathit{right}\ g) \ggg \mathit{left}\ f \\
\text{Unit} & f \ggg \mathit{arr}\ \mathit{Left} & = & \mathit{arr}\ \mathit{Left} \ggg \mathit{left}\ f \\
\text{Assoc.} & \mathit{left}\ (\mathit{left}\ f) \ggg \mathit{arr}\ \mathit{assoc}_+ & = & \mathit{arr}\ \mathit{assoc}_+ \ggg \mathit{left}\ f
\end{array}
$$

$$
\begin{array}{ll}
\mathit{assoc}_+\ (\mathit{Left}\ (\mathit{Left}\ x)) & = \mathit{Left}\ x \\
\mathit{assoc}_+\ (\mathit{Left}\ (\mathit{Right}\ y)) & = \mathit{Right}\ (\mathit{Left}\ y) \\
\mathit{assoc}_+\ (\mathit{Right}\ z) & = \mathit{Right}\ (\mathit{Right}\ z)
\end{array}
$$

$$
\text{Non-interference} \quad \mathit{arr}\ \mathit{Right} \ggg \mathit{left}\ f\ =\ \mathit{arr}\ \mathit{Right}
$$

Figure 3.1: The standard laws for arrow choice with our new non-interference law below.

3.1.3 Pausable Signal Functions Revisited

With non-interfering choice in our arsenal, we can define a new version of *integralWhen* in an even more intuitive and straightforward way:

$$
\begin{array}{l}
\mathit{integralWhen}_{\mathit{Choice}} :: (\mathit{Double}, \mathit{Bool}) \rightsquigarrow \mathit{Double} \\
\mathit{integralWhen}_{\mathit{Choice}} = \mathbf{proc}\ (i, b) \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ v \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \mathit{integral} \prec i \\
\qquad\qquad\ \ \mathbf{else}\ \ \mathit{returnA} \prec v \\
\quad \mathit{returnA} \prec v
\end{array}
$$

Because we are not actually switching out of the *integral* signal function, it will retain its state internally. When it is executed, it will calculate and add the latest delta of integral, and otherwise, it will simply wait.

3.1.4 A Single First-Order Switch

The most basic switching operation is to non-recursively switch out one signal function for another dynamically. For example, we could write a simple guessing game that accepted an event stream of guesses, and when the correct answer was provided, it would switch into a signal function that ignored its input and declared that the game was over:

$$
\begin{array}{l}
\mathit{guess} :: \mathit{Event}\ \mathit{Int} \rightsquigarrow () \\
\mathit{guess} = \mathit{switch}\ (\mathit{arr}\ f)\ (\lambda\ t \rightarrow \mathit{label}\ t) \\
\quad \mathbf{where}\ \ f\ (\mathit{Event}\ i) | (i == 3) = ((),\ \mathit{Event}\ \text{“You Win!”}) \\
\qquad\qquad\ \ f\ \_ \qquad\qquad\qquad\ \ = ((),\ \mathit{NoEvent})
\end{array}
$$

where *label* is a signal function widget that ignores its streaming input and displays the text it was given as its static argument. Note that we are using the plain, non-recurring, non-parallel version of switch that we presented in Section 2.1.4. In *guess*, when the event containing 3 is processed, the string “You win!” is given to the label, and the guessing is switched out for that label.

```
runNTimes :: Int → (α ⇝ β) → ([α] ⇝ [β])
runNTimes 0 _ = constA [ ]
runNTimes n sf = proc (b : bs) → do
  c ← sf ⊸ b
  cs ← runNTimes (n − 1) sf ⊸ bs
  returnA ⊸ (c : cs)
```

Figure 3.2: The implementation of *runNTimes* using structural recursion.

For this example again, switch is too strong. Notice that the argument given to the switched-in signal function is not itself a signal function. In fact, it's just a constant! We can rewrite this with non-interfering choice:

```
guess_choice :: Int ⇝ ()
guess_choice = proc i → do
  rec haveWon ← delay False ⊸ haveWon || (i == 3)
  if haveWon then label "You Win!" ⊸ ()
             else returnA ⊸ ()
```

Note that we changed the input stream to a continuous stream as opposed to an event stream simply to make the example clearer.

Reacting to dynamic events

The above versions of *guess* are quite primitive, and although we use switching in the first one, we are far from using its full power. We can make the example slightly more complex by adding an additional component to the input such that the program is actually reactive:

```
guess′ :: Event (Int, String) ⇝ ()
guess′ = switch (arr f) (λ t → label t)
  where f (Event (i, s))|(i == 3) = ((), Event s)
        f _                       = ((), NoEvent)
```

In *guess′*, the text to put in the label is no longer static and instead is part of the guess event, and in its current form, switching is a necessity as it is the only way to provide the dynamically streaming string to the static *label* function. However, we could once again lift the need for switching if we could redesign the label to instead take an *impulse*. An impulse is a one time event that initializes a signal function, so in this case, the type for *label* would change from $String \to (\alpha \rightsquigarrow ())$ to $(Event\ String) \rightsquigarrow ()$.

With an impulse driven label widget, we can once again convert the *guess′* function to a switch-free alternative:

```
guess′_choice :: (Int, String) ⇝ ()
guess′_choice = proc (i, s) → do
  rec haveWon ← delay False ⊸ haveWon || (i == 3)
  let imp = if not haveWon && i == 3
            then Event s else NoEvent
  if haveWon then label ⊸ imp
             else returnA ⊸ ()
```

```
runDynamic :: (α ⇝ β) → ([α] ⇝ [β])
runDynamic sf = proc lst → do
  case lst of
    [ ]      → returnA ⊸ [ ]
    (b : bs) → do c ← sf ⊸ b
                  cs ← runDynamic sf ⊸ bs
                  returnA ⊸ (c : cs)
```

Figure 3.3: The implementation of the choice-based *runDynamic* function using arrowized recursion.

3.1.5 Arrowized Recursion

As we have shown in the previous two examples, there is a direct usage for non-interfering choice, but the non-interference law also gives us a less obvious benefit. By restricting the arrow effects to only one branch, we open the door to the possibility of a new kind of recursion.

Typically, arrows can perform recursive behaviors in one of two ways. First, arrows can use the *loop* functionality to perform a value level recursion, or a sort of fix point recursion. After all, one of the laws for *loop* is:

$$loop\ (arr\ f) = arr\ (\lambda\ b \rightarrow fst\ (fix\ (\lambda(c,d) \rightarrow f\ (b,d))))$$

Second, there is *structural* recursion. Structural recursion happens when the host language's recursion is used to create an arrow in a recursive way. For instance, we might have a function like:

$$runNTimes :: Int \rightarrow (\alpha \rightsquigarrow \beta) \rightarrow ([\alpha] \rightsquigarrow [\beta])$$

When defining this function, we use Haskell's conditional syntax to recur on the value of the first argument: while it is greater than zero, we run the signal function and recur, and when it is equal to zero, we return a constant stream of the empty list. We show a definition of *runNTimes* using this form of recursion in Figure 3.2.

A key frustration with structural recursion is that the recursive argument is static as opposed to streaming. Thus, structural recursion is often performed in tandem with higher-order switching to allow a streaming value to be used in place of the static argument.

One may be inclined to perform recursion using arrow choice, but with the standard choice laws, this can be problematic. In general, if both branches of an arrow choice statement perform effects, then both of those effects must be applied statically regardless of the dynamic streaming values provided. In other words, recursion within arrows, even when guarded by arrow choice, can loop indefinitely in certain implementations.

The non-interference law forces us to delay effects until the dynamic values are ready, which in turn allows us to use arrow choice for recursion. We call this new form of recursion *arrowized* recursion. In practice, it is very similar to structural recursion except that instead of using the host language's conditional, we use arrow choice.

With arrowized recursion, we can write a function similar to the above *runNTimes* but that needs no static argument to perform its recursion. In fact, we can make the input stream of lists the recursive argument and eliminate the need for an "N" altogether. We call this function *runDynamic* and show it in Figure 3.3.

When used, *runDynamic* has exactly the behavior one would expect of using standard arrow choice. That is, any signal functions that are not in currently active branches are stopped. For example, if we were

to run *runDynamic integral* with a signal defined as:

$$\begin{aligned}
[0.0,1.0): &\ [1.0] \\
[1.0,2.0): &\ [2.0,3.0] \\
[2.0,3.0): &\ [1.0] \\
[3.0,4.0): &\ [2.0,3.0,4.0]
\end{aligned}$$

then we would see the following results:

$$\begin{aligned}
[0.0,1.0): &\quad [0.0] - [1.0] \\
[1.0,2.0): &\quad [1.0,0.0] - [3.0,3.0] \\
[2.0,3.0): &\quad [3.0] - [4.0] \\
[3.0,4.0): &\ [4.0,3.0,0.0] - [6.0,6.0,4.0]
\end{aligned}$$

It should be noted that using arrowized recursion creates new signal functions by need (i.e. the new *integral* that is created at $t = 3$ above), but once they are created they are kept around in perpetuity. This is discussed again in Section 3.6.2.

3.1.6 Dynamic GUI

One power of switch, showcased particularly in *Fruit* [Courtney and Elliott, 2001a], is the ability to allow a dynamic number of signal functions to execute. That is, by default, arrows have a fixed structure, and the streaming values moving through an AFRP program cannot affect that structure. However, switch allows one to dynamically alter the arrow at runtime based on the streaming values.

For example, one may desire a GUI that gathers the names of an unknown group of people. If the size of the group were fixed or at least known at compile time, then this is achievable trivially with arrows, but if the size is a parameter that is filled in by the user of the GUI, then standard arrows are stymied. One approach is to use a switching mechanism.

For this example, we will assume a few GUI widgets:

$$\begin{aligned}
&\mathit{label} &&:: \mathit{String} \to ((\,) \rightsquigarrow (\,)) \\
&\mathit{getInteger} &&:: (\,) \rightsquigarrow \mathit{Int} \\
&\mathit{getIntegerE} &&:: (\,) \rightsquigarrow \mathit{Event\ Int} \\
&\mathit{getName} &&:: (\,) \rightsquigarrow \mathit{String}
\end{aligned}$$

Note that we have both a regular and event-based version of *getInteger*: the event-based one, which produces an event each time the value changes, is useful for our example with switch, and we will use the regular one with choice.

We can use these widgets in combination with the *rSwitch* function to make our GUI:

$$\begin{aligned}
&\mathit{getNames} :: (\,) \rightsquigarrow [\mathit{String}] \\
&\mathit{getNames} = \mathbf{proc}\ (\,) \to \mathbf{do} \\
&\quad _ \leftarrow \mathit{label}\ \text{“How many people?”} \prec (\,) \\
&\quad e_n \leftarrow \mathit{getIntegerE} \prec (\,) \\
&\quad \mathit{rSwitch}\ (\mathit{constA}\ [\,])\ \prec (\mathit{repeat}\ (\,), \\
&\qquad \mathit{fmap}\ (\lambda\ n \to\ \mathit{runNTimes}\ n\ \mathit{getName})\ e_n)
\end{aligned}$$

where the *runNTimes* function is the one we discussed in the previous subsection (that uses structural recursion to run the given signal function the given number of times, as shown in Figure 3.2).

The above definition of *getNames*, although correct, is using the higher order nature of switch when it is not truly necessary. Switching gives the power to substitute in any new signal function for the currently running one, but here, the nature of the new signal function is already known: it will be some number of *getName* widgets. Because this fact is known at compile time, we can use arrowized recursion instead to create a simpler, switch-free GUI.

$$
\begin{aligned}
&\mathit{getNames} :: () \rightsquigarrow [\mathit{String}] \\
&\mathit{getNames} = \mathbf{proc}\ () \to \mathbf{do} \\
&\quad \_ \leftarrow \mathit{label}\ \text{“How many people?”} \prec () \\
&\quad n \leftarrow \mathit{getInteger} \prec () \\
&\quad \mathit{runDynamic}\ \mathit{getName} \prec \mathit{replicate}\ n\ ()
\end{aligned}
$$

Because *runDynamic* uses arrow choice to do arrowized recursion, we do not need to use any switching.

3.2 A Case for Settability

In this section, we will explore a second main use of switchers: the ability to start a signal function mid-computation with no prior state. Once again, we will begin with a simple yet canonical example before describing our first-order alternative and some further usage examples.

3.2.1 Restartable Computation

Although pausing signal functions is useful (as in the *integralWhen* example of Sections 3.1.1 and 3.1.3), there are times when we really do want to restart a signal function, resetting its state to its initial defaults. In fact, with switching, this is even easier than pausing considering that switch naturally starts its new signal function from the beginning.

For instance, let us consider the scenario where we would like to take the integral of a stream, but at any moment, we may be given an event that indicates that we should reset the integral's accumulation to its initial default. With switch, this is actually trivial: we simply lift the *integral* function into the resetting event, and send everything into a recurring switcher:

$$
\begin{aligned}
&\mathit{integralReset}_{\mathit{Switch}} :: (\mathit{Double}, \mathit{Event}\ ()) \rightsquigarrow \mathit{Double} \\
&\mathit{integralReset}_{\mathit{Switch}} = \mathbf{proc}\ (i, e) \to \mathbf{do} \\
&\quad \mathit{rSwitch}\ \mathit{integral} \prec (i,\ \mathit{fmap}\ (\mathit{const}\ \mathit{integral})\ e)
\end{aligned}
$$

Without switch, this seems like a tough problem, and nothing about non-interfering choice lends any help.

One idea is to try to simulate the behavior of a restart without actually touching *integral* itself. That is, because the function we are lifting is just an integral, we could take a snapshot of its output at the restarting moment and then continuously subtract that value from future outputs:

$$
\begin{aligned}
&\mathit{integralReset}_{\mathit{Basic}} :: (\mathit{Double}, \mathit{Event}\ ()) \rightsquigarrow \mathit{Double} \\
&\mathit{integralReset}_{\mathit{Basic}} = \mathbf{proc}\ (i, e) \to \mathbf{do} \\
&\quad o \leftarrow \mathit{integral} \prec i \\
&\quad \mathbf{rec}\ k \leftarrow \mathit{delay}\ 0 \prec k' \\
&\qquad\quad \mathbf{let}\ k' = \mathbf{if}\ \mathit{isEvent}\ e\ \mathbf{then}\ o\ \mathbf{else}\ k \\
&\quad \mathit{returnA} \prec o - k
\end{aligned}
$$

Although this is a valid solution to this particular situation, a similar solution cannot always be found. To do so requires a function that point-wise transforms the output to what it would have been if the signal function were started at the designated point in time, and this function must be computable from the output from that point in time forward.

3.2.2 Settability

At this point, the idea of lifting a signal function into the event stream, as we did in $integralReset_{Switch}$ above, should seem unnecessary. Indeed, we are not even switching into some dynamically given new signal function but rather just using a new instance of the same signal function again. Rather than switching, our first-order approach is to develop a notion of signal function *settability*, or a way to change the internal state of a signal function at arbitrary points.

Because we are dealing with state, we will begin with an even more primitive example and examine the *delay* operator directly. At first glance, it seems to suffer from the same problem as *integral*—the *delay* will always output old values, so what can we do to reset it? However, modifying it to be resettable requires only the addition of a single input event stream:

$$
\begin{aligned}
&resettableDelay :: \beta \to ((\beta, Event\ ()) \rightsquigarrow \beta) \\
&resettableDelay\ i = \mathbf{proc}\ (b, e) \to \mathbf{do} \\
&\quad out \leftarrow delay\ i \prec b \\
&\quad returnA \prec \mathbf{case}\ e\ \mathbf{of} \\
&\qquad NoEvent \to out \\
&\qquad Event\ () \to i
\end{aligned}
$$

Whenever *resettableDelay* is given an event, it will immediately output its initial value again, essentially behaving as if it has only just started. In fact, we can take this one step further and construct a version of *delay* that can be set to any value of our choosing:

$$
\begin{aligned}
&settableDelay :: \beta \to ((\beta, Event\ (Maybe\ \beta)) \rightsquigarrow \beta) \\
&settableDelay\ i = \mathbf{proc}\ (b, e) \to \mathbf{do} \\
&\quad out \leftarrow delay\ i \prec b \\
&\quad returnA \prec \mathbf{case}\ e\ \mathbf{of} \\
&\qquad NoEvent \qquad \to out \\
&\qquad Event\ Nothing \to i \\
&\qquad Event\ (Just\ s) \to s
\end{aligned}
$$

With *settableDelay*, the event stream can potentially carry a new value to set the internal state, and if there is no value, we perform a reset. It may seem superfluous to have an event of an option, but adding the ability to set the state does not make resetting the state obsolete.

A fortuitous bonus to this function is that, in addition to being able to set the state, we can also capture the current state. That is, because the input stream is necessarily setting the new current state, it can also be made to provide it directly. Thus, we can use *settableDelay* to both "store" and "load" state.

General Settability

Although a settable version of *delay* may be useful on its own, it would be much more useful to have any arbitrary signal function be settable. However, this would require manually changing every internal *delay* operator to its settable alternative and then properly routing the state-setting events to the appropriate places. Additionally, if capturing the state at a given moment were important, all of the inputs to the *delay* functions would also need to be grouped and appropriately routed to the output. This would be exceptionally cumbersome and not at all feasible. What we want is a function like:

$$settable :: (\alpha \rightsquigarrow \beta) \to ((\alpha, Event\ State) \rightsquigarrow (\beta, State))$$

that will automatically take a signal function and allow us to both pass in an optional new state as well as save its current state. For now, we will assume that the *State* type can encode an arbitrary type (along

$$settable :: (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha, Event\ State) \rightsquigarrow (\beta, State))$$

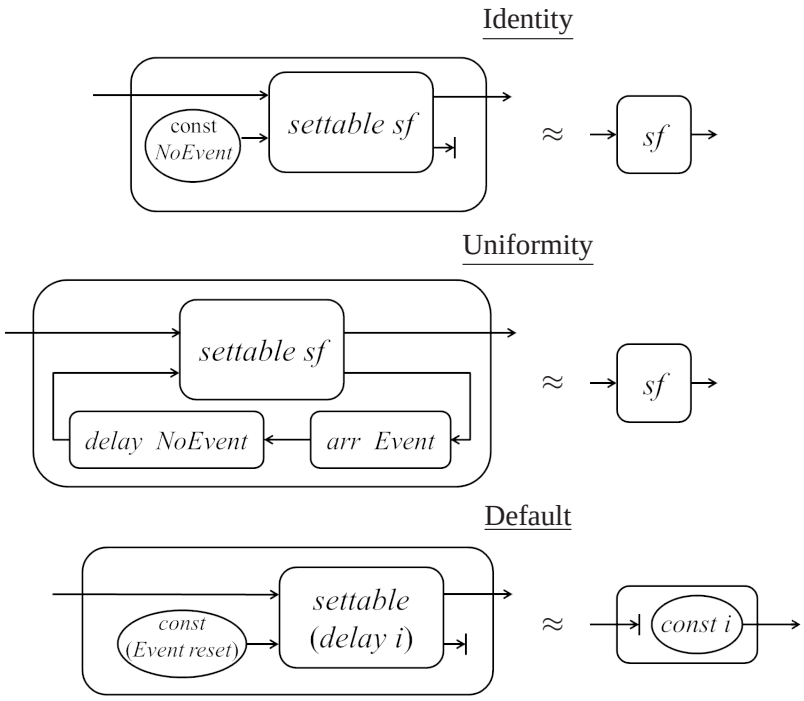


Figure 3.4: The *settable* function and its laws.

with a special "reset" value), and we will discuss it in more detail when we discuss the implementation of settability in Section 3.4.2.

This *settable* function should hold to certain principles of behavior. For example, if it is never provided with a state, then it should do nothing. Similarly, if the state it produces is used to set it, then there should be no observable difference in behavior. Additionally, there should be a particular value of *State* that acts as a *reset* (in our *settableDelay* function from earlier, this was *Event Nothing*). Thus, if one were to feed a constant stream of reset states, the output would always use the default values. We declare these principles as laws of behavior for *settable* and show them diagrammatically in Figure 3.4.

In fact, with an appropriate code transformation, any arrow can be extended with a *settable* function. We will explore the details of this transformation in Section 3.4, but for now, it suffices to state that it is possible and available in our examples.

3.2.3 Restartable Computation Revisited

With the *settable* function, defining *integralReset* is just as trivial as with switch:

$$\begin{aligned}
&integralReset :: (Double, Event\ ()) \rightsquigarrow Double \\
&integralReset = \textbf{proc}\ (i, e) \rightarrow \textbf{do} \\
&\quad (v, s) \leftarrow settable\ integral \prec (i,\ fmap\ (const\ reset)\ e) \\
&\quad returnA \prec v
\end{aligned}$$

Rather than lifting a dynamic signal function to the signal level just to be activated by switch as we did previously, we lift only a reset signal. The difference in the amount of code between this function and $integralReset_{switch}$ is negligible (it basically comes down to ignoring the state output of the settable signal function), but the conceptual difference is quite important: rather than needing to stop a currently running signal function to replace it with a new, fresh instance of itself, it is possible to refresh it while leaving it active.

3.2.4 Freezing and Duplicating

This *settable* function has applications beyond just resetting arbitrary, stateful signal functions. By separating the state from the signal function, we are essentially separating the current behavior from the structure. That is, the *settable* function gives us the power to *freeze* signal functions.

Typically freezing a signal function is thought of as a higher-order operation achievable only with a switch operator. Specifically, freezing is the process of stopping a running signal function mid-execution and providing it as a piece of data to reuse. Later, it can be resumed by using a switcher to reintegrate it into the structure of the program.

Rather than providing a copy of itself, a function made settable will provide a stream of its *essence* (i.e. its current state), which can then be reinserted at any time later. It is worth noting that this does not provide any advantage over switch in terms of resources or memory, but it does provide the ability to freeze and resume without actually needing switch in the language.

Example

For this example, we will construct a GUI for drawing. The main window will feature a drawing pane, but the user will be able to create new panes and switch between them. When a new pane is created, it is automatically populated with a copy of whatever is currently on the current pane.

For this example, we will assume a few widgets:

$$\begin{aligned} &\mathit{drawing} &&:: () \rightsquigarrow () \\ &\mathit{choosePane} &&:: () \rightsquigarrow \mathit{Int} \\ &\mathit{button} &&:: \mathit{String} \rightarrow (() \rightsquigarrow \mathit{Event}\ ()) \end{aligned}$$

The *drawing* widget is a stateful, effectful widget that provides a canvas and allows the user to draw; the *choosePane* widget returns an *Int* stream that represents the currently selected pane; and the *button* widget takes a static label and produces an event stream that indicates when the button has been pressed.

With these widgets, we can create the GUI we described (shown in Figure 3.5). The state for the GUI is kept as a list of drawing states, initialized in the sixth line as a one element list containing a *reset* state. This initial list describes a GUI with a single pane that has a blank drawing canvas. When a user wishes to duplicate the current pane, the current state is added to the list allowing the GUI to "save" the original pane while providing a duplicate state for the new one. The key here is that instead of keeping track of different instances of the signal function, each with its own state, we keep track of multiple states themselves and use them with a single signal function.

To make a version of this GUI with switching is surprisingly complicated. Instead of keeping track of multiple states for the single *drawing* widget, we keep a collection of multiple *drawing* widgets that we can switch between as necessary. The only version of switch that provides this information is the parallel *pSwitch*, which processes collections of signal functions. Therefore, we will start by using concepts borrowed from Giorgidze and Nilsson [2008] for the practical use of pswitch. However, because we are only actually running a single *drawing* widget at a time, we are forced to use some clever engineering:

```
gui :: () ⇝ ()
gui = proc () → do
  e_dup ← button "Duplicate pane?" ⪡ ()
  e_del ← button "Delete pane?" ⪡ ()
  i ← choosePane ⪡ ()
  rec stateLst ← delay [reset] ⪡ stateLst_new
      ((), state_new) ← settable drawing ⪡ ((), stateLst !! i)
      let stateLst_new = case (e_dup, e_del) of
            (Event (), _) → set i stateLst state_new ++ [state_new]
            (NoEvent, Event ()) → delete i stateLst
            _ → set i stateLst state_new
  returnA ⪡ ()
```

Figure 3.5: The implementation of the GUI from Section 3.2.4.

- First, in order to satisfy pswitch's requirement for a collection, we create a new indexed list data type *IList*. Applying *fmap* over it applies the given function only to the currently indexed element.
- We need an event every time the user selects a different pane, and we achieve this by using the helper function *unique*, which converts a continuous stream to a discrete one by providing an event containing the value of the stream whenever it changes.
- We need the switching to be repeatable, so we call *pSwitch* recursively.

The result is shown in Figure 3.6.

3.3 An Alternative to *pSwitch*

Here, we will pull together the ideas of both settability and non-interfering choice that we have highlighted in the previous sections to present a high power yet first-order version of a parallel switcher.

As we mentioned in Section 2.1.4, parallel switchers allow for whole collections of signal functions to be managed and switched in or out at once. One example of the usefulness of this kind of switcher can be seen in the musical realm where one might have a program that plays music with software "instruments" that are actually themselves signal functions. The music is given as a sequence of "On" and "Off" events, where the "On" events provide the instrument to play and some initializing data about what note to play, and the "Off" events tell which instrument to stop:

```
data NoteEvt = NoteOn UID Instr InitData
             | NoteOff UID Instr

type Instr = InitData → (() ⇝ Sound)

sumSound :: [Sound] ⇝ Sound
```

Note that the *UID* type is a unique identifier that is used to connect a given *NoteOn* event with its *NoteOff* counterpart, and the *Sound* data type represents the sound that an instrument produces. The *sumSound* signal function is for summing dynamic lists of sounds together.

```
data IList a = IList Int [a]
instance Functor IList where
  fmap f (IList i lst) = IList 0 [f (lst !! i)]

gui_switch :: () ⇝ ()
gui_switch = proc () → do
  e_dup ← button "Duplicate pane?" ⪦ ()
  e_del ← button "Delete pane?" ⪦ ()
  e_i ← unique ⋙ choosePane ⪦ ()
  pSwitch initialSFs (arr test) k ⪦ (e_dup, e_del, e_i)
  returnA ⪦ ()
  where initialSFs = IList 0 [drawing]
        test ((NoEvent, NoEvent, NoEvent), _) = NoEvent
        test (inp, _) = Event inp
        k (IList i_prev lst)(NoEvent, NoEvent, Event i) =
            pSwitch (IList i (set i_prev lst (lst !! i_prev))) (arr test) k
        k (IList i lst)(NoEvent, Event (), _) =
            pSwitch (IList i (delete i lst)) (arr test) k
        k (IList i lst)(Event (), _, _) =
            pSwitch (IList i (lst ++ [lst!!i])) (arr test) k
        k ilst_ = pSwitch ilst (arr test) k
```

Figure 3.6: The implementation of the GUI from Section 3.2.4 using switch instead of settability.

Although we will use the same *pSwitch* that we introduced in Section 2.1.4, for clarity, we will show its type signature again, this time with a few of the type variables instantiated for our example.

$$\begin{aligned} &pSwitch :: [UID, () \rightsquigarrow \beta] \\ &\quad \rightarrow (() \rightsquigarrow Event\ \gamma) \\ &\quad \rightarrow ([UID, () \rightsquigarrow \beta] \rightarrow \gamma \rightarrow [UID, () \rightsquigarrow \beta]) \\ &\quad \rightarrow (() \rightsquigarrow [\beta]) \end{aligned}$$

For our collection, we use a mapping of *UID* to signal function (which we implement as a list for simplicity), and we set the input type α to $()$.

For this musical example, the initial list of signal functions will be empty, the events to change that list will be *NoteEvt*s, and the function will use the *NoteEvt* data to add or remove signal functions from the list as necessary:

$$\begin{aligned} &maestro :: (() \rightsquigarrow Event\ [NoteEvt]) \rightarrow (() \rightsquigarrow Sound) \\ &maestro\ music = pSwitch\ [\,]\ music\ f \ggg sumSound \\ &\quad \textbf{where}\ f\ lst\ [\,] = lst \\ &\qquad f\ lst\ (NoteOn\ u\ i\ imp : rst) = f\ ((u, i\ imp) : lst)\ rst \\ &\qquad f\ lst\ (NoteOff\ u\ i : rst) = f\ (filter\ ((\neq u) \,.\, fst)\ lst)\ rst \end{aligned}$$

In order to remove our reliance on switch, we need to make a few small changes to the layout of the problem. First, as we did in Section 3.1.4, we will need to change the instruments from functions that take a "static" initializing argument to functions that take that argument as an impulse. Second, we need to know statically what the different signal functions are, so we make use of a finite data type and add one layer of indirection:

$$\textbf{data}\ Instr = Trumpet \mid FHorn \mid Trombone \mid Tuba$$

$$\textbf{type}\ Instrument = Event\ InitData \rightsquigarrow Sound$$

$$toInstrument :: Instr \rightarrow Instrument$$

It is critically important that the *Instr* type is finite because, due to the fact that choice is not actually higher order, we need to know exactly which *Instrument* signal functions can possibly be called. This technique of representing functions by a first-order data type and then interpreting them later is known as *defunctionalization* [Reynolds, 1972, Danvy and Nielsen, 2001] and has been established as a viable method of converting higher-order functions into first-order ones. Fortunately, in most situations where parallel switching is used, the possibilities of signal functions are known statically, so a transformation like this one is not difficult.

With these changes made, we can utilize the *pChoice* function. The idea behind *pChoice* is that as long as we know the possible signal functions that we may use, we can run each one a dynamic number of times. So, rather than keep a dynamic list of signal functions, we keep a static list of signal functions and a dynamic list of signal function *states*. We then use a combination of structural and arrowized recursion: structural recursion to provide access to each possible signal function and arrowized recursion to allow a dynamic number of runs per possibility.

The type of *pChoice* is:

$$\begin{aligned} &pChoice :: Eq\ key \Rightarrow [(key, Event\ \alpha \rightsquigarrow \beta)] \rightarrow \\ &\qquad ([(key, (UID, Event\ \alpha))] \rightsquigarrow [\beta]) \end{aligned}$$

and as it is somewhat complicated, we leave its implementation and a more detailed description of its inner-functioning to Appendix A.2.

We can use *pChoice* to reimplement our music program without switch:

$$
\begin{aligned}
&\mathit{maestro} :: [\mathit{NoteEvt}] \rightsquigarrow \mathit{Sound} \\
&\mathit{maestro} = \mathit{arr}\ (\mathit{map}\ f) \ggg \mathit{pChoice}\ \mathit{lst} \ggg \mathit{sumSound} \\
&\quad \textbf{where}\ \mathit{lst} = \mathit{map}\ (\lambda\ i \rightarrow (i, \mathit{toInstrument}\ i))\ \mathit{allInstrs} \\
&\qquad\quad f\ (\mathit{NoteOn}\ u\ i\ \mathit{imp}) = (i, (u, \mathit{Event}\ \mathit{imp})) \\
&\qquad\quad f\ (\mathit{NoteOff}\ u\ i) \qquad = (i, (u, \mathit{NoEvent}))
\end{aligned}
$$

where *allInstrs* is a complete list of all of the *Instr*s that might be played. In fact, one notable difference between this version of *maestro* and the switch-based alternative from earlier is this *allInstrs* list: the reason that we can write this program at all is because *allInstrs* can be defined statically.

3.4 Implementing Settability

As we mentioned in Section 3.2.2, we can achieve settability of any arrow with a code transformation. Here, we will provide a detailed description of the transformation process before presenting Haskell code that implements it.

3.4.1 Design

In essence, the idea of settability is the idea of having access to the internal state of an arrow. Thus, as we discussed previously, it is encapsulated by a function like:

$$\mathit{settable} :: (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha, \mathit{Event}\ \mathit{State}) \rightsquigarrow (\beta, \mathit{State}))$$

that will automatically take a signal function and allow us to both pass in an optional new state as well as save its current state. However, in order to achieve this, we will need to rewrite the underlying arrow to support this behavior. Therefore, we will describe a recursive transformation that will provide settable capabilities to ordinary arrows.

Intuitively, this settability transformation is a simple process of routing state update information in through the various arrow combinators so that it can be easily accessed by any internal delay operators and then routing current state data back out through the combinators to the level of the *settable* call. For each combinator, there is a transformation that achieves exactly this goal; we show circuit diagrams for these transformations in Figure 3.7 and describe them in detail below. Note that we use the notation $\overline{sf}$ to denote the signal function *sf* after having been transformed, and we assume that the *Event State* input stream and *State* output stream are always the lower input and output.

- We will begin at the lowest level by examining the *delay* operator itself. In Section 3.2.2, we showed a design for a settable version of delay, but we need to modify it just slightly in order for it to be general enough for our *settable* transformation: in addition to taking in an *Event State* stream, it also needs to emit its current *State* as a stream. This is rather trivial as its current state is identical to its own input stream, but this is important to the transformation as a whole. Thus, our circuit diagram shows the input stream both being sent to the embedded *delay* operator as well as being duplicated to the *State* output, and the output is determined by a case analysis of the *Event State* input with data from the *delay*'s output.

- The simplest transformation is that of the *arr* operator, which has no state and should essentially remain unaffected. In this case, we ignore the input *Event State* and return a constant stream of the null, or *reset*, state.

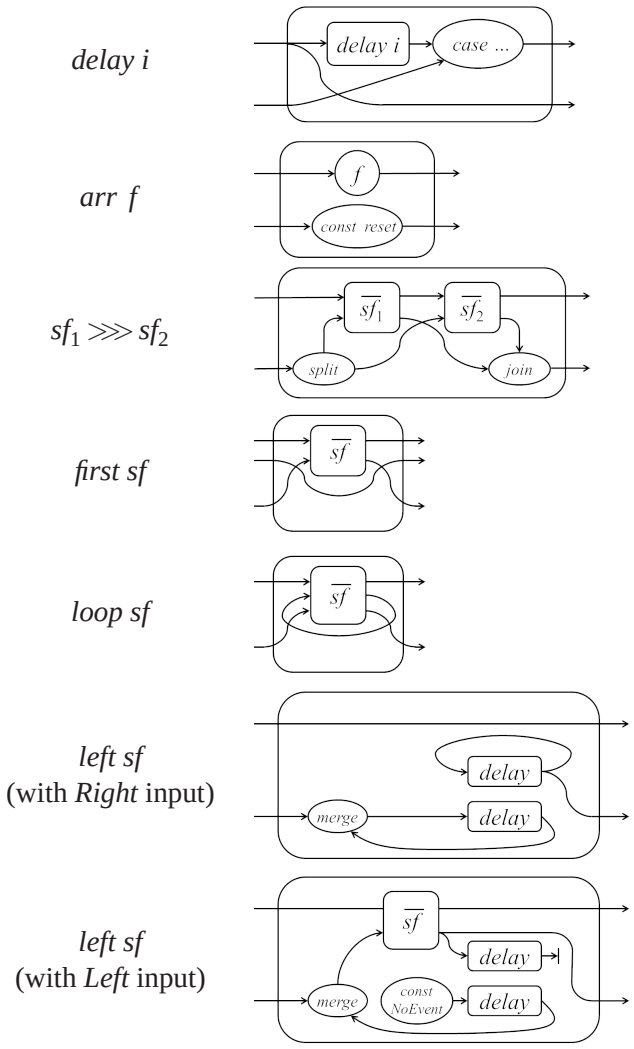


Figure 3.7: The circuit diagrams showing the settability transformations for the various arrow combinators.

- The composition of two functions is a little more interesting. Each of the two composed signal functions may have state, so we need to split the incoming *Event State* into two pieces and pass the first to the first signal function and the second to the second. We gather the resulting states together and join them into a single output state.
- Applying a partial application (*first*) is a simple matter of rerouting the state data and the unused input stream properly.
- Looping is handled similarly to partial application with a simple rerouting of streams.
- The most complicated transformation is for our non-interfering choice's *left* operator. This is because there are two difficult questions that we must address in designing this transformation. First, in the case of an input *Right* value, the embedded signal function is not executed, so where can we get a *State* value for the output *State* stream? And second, again in the case of an input *Right* value, if we are given an *Event State* that requires updating the embedded signal function, how can we get that event where it needs to go? The way to address both of these questions is to allow the transformed choice operator to contain some internal state, which we achieve with *loop* and *delay*.

  Furthermore, in an effort to clarify the behavior of the transformed choice, we provide two diagrams to describe its behavior: one that shows how it behaves when given a *Right* value and the other for when it is given a *Left* value. The *delay*s are shared between both diagrams: the upper *delay* should be assumed to be initialized with a *NoEvent* value and the lower with a null, or *reset*, state value. The *merge* function is a standard overwriting event merge that favors the left (newly incoming) event in the case of two events.

  When given a *Right* input, the input stream is identical to the output stream. The *Event State* input is merged with the stored *Event State* and stored once again, thus updating the store with any new setting events. The output *State* is the stored one.

  When given a *Left* input, we will execute the embedded signal function. We still merge the *Event State* input with the stored one, but the result goes directly into the embedded signal function, and the store is instead updated with a *NoEvent*, indicating that there are no past *Event State*s waiting to be delivered. The output of the transformed, embedded signal function, both the streaming *Left* value as well as the output *State*, become the output of the overall transformed signal function, but the output *State* is also stored for potential future use. The stored *State* value is discarded outright as it is now obsolete.

Setting Switch

Although the stated purpose of this chapter is to develop language constructs to allow us to remove switchers from FRP, the constructs themselves do not necessarily preclude switching. Indeed, we can extend the ideas of settability to include switching without much work at all.

We will begin by instead looking at the *app* operator from the *ArrowApply* class Hughes [2000]:

$$app :: (a \rightsquigarrow b, a) \rightsquigarrow b$$

This operator is very similar to a switch, but it is notably different in that it treats the new signal function input as a continuous stream instead of as discretely separate events. This is important because once the higher-order signal function is used, it is discarded, ready to be replaced at the next moment. This means that, in essence, there is no sense of state here, which in turn means that making it settable should have no impact on its behavior. Another way to reach this conclusion is to consider what it would mean to set the state of *app*. Whatever may happen, that state will be immediately overwritten by the streaming signal function component. Ultimately, for the purposes of settability, *app* is a pure signal function.

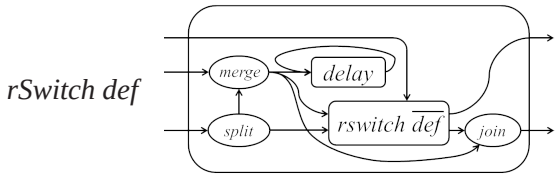


Figure 3.8: The circuit diagram showing the settability transformation for the repeating switcher.

Although *app* is stateless, switchers are not. However, the above explanation will come in handy as we consider the case for the moment that switching occurs.

Let us no consider applying the settability transformation to the repeating *rSwitch*. In general, the state of the switcher is the pair consisting of the currently running signal function and its state. Therefore, the output state will be a joined state of these two. Because we only see the currently running signal function when it is supplied as an event, the transformed *rSwitch* must keep its own internal state, which we can implement with a loop and delay. Finally, the input state event must be allowed to overwrite the current behavior except for when a switching event occurs, in which case that switching event takes priority. We show this all diagrammatically in Figure 3.8.

3.4.2 Haskell Implementation

Rather than relying on Haskell's rewrite rules or Template Haskell, we can perform the entire transformation with only type classes. Our method involves creating a wrapper for a generic arrow that itself instantiates the arrow classes. Then, any code that is an arbitrary arrow could just as well be this wrapper.

Thus, our goal will be to concretely define our types and then instantiate the arrow classes using them. We lay out the process in this section and also note that the code is available online as a Haskell package.[1]

Data Types

The first type we must choose a concrete representation for is the *State* data type. For a single *delay*, the definition of *State* seems obvious: it is a maybe type of the stored value (just as we saw in the *settableDelay* example from Section 3.2.2). One way to extend this to arbitrary signal functions would be to extend the idea of arrows to include an extra parameter that indicates what that arrow's state is. Then, when we compose two arrows, we combine the two component states into one joined state. Despite being cumbersome, this becomes especially challenging in the presence of arrowized recursion, where we would need some sort of coinductively defined state type to allow for type unification. Indeed, this should be technically possible using Haskell's type families and other features, but the complexity would detract from our point. Therefore, to keep the types simple, we make use of Haskell's *Dynamic* data type to store arbitrary state information from individual *delay* functions.[2] Also, rather than use an auxiliary option type to represent a default state or an absence of state (as we did in the *settableDelay* function in Section 3.2.2), we will build this directly into the type.

We show the definition of the *State* data type along with the few helper functions we need in Figure 3.9. Note that because *NoState* represents an absence of state information, trying to split it returns a similar lack of information.

[1] hackage.haskell.org/package/SettableArrow

[2] Technically, using *Dynamic* in this way enforces a *Typeable* restriction to the types of the individual state components, but this is of little consequence.

```
data State = NoState
           | DState Dynamic
           | PairState State State

reset = NoState

split :: Event State → (Event State, Event State)
split NoEvent                = (NoEvent, NoEvent)
split (Event NoState)        = (Event NoState, Event NoState)
split (Event (PairState l r)) = (Event l, Event r)

join :: State → State → State
join l r = PairState l r

merge :: Event → Event → Event
merge NoEvent  e = e
merge e        _ = e
```

Figure 3.9: The *State* data type and its two accessor functions.

With the *State* type defined, we next build our wrapper for a general arrow:

$$\textbf{data}\ SA\ (\rightsquigarrow)\ \alpha\ \beta = SA\ ((\alpha, Event\ State) \rightsquigarrow (\beta, State))$$

Already, we can see that this *SA* data type is merely hiding the extra piping that will be required to store and load the state.

Instantiating Arrow

Next, we show how $SA\ (\rightsquigarrow)$ can instantiate the arrow operators themselves. If it can, then any program written using the arrow operators could just as well be written for the generic arrow $(\rightsquigarrow)$ as for $SA\ (\rightsquigarrow)$. Thus, this instantiation will essentially provide a method to perform a code transformation to allow any arrow to behave as if it could be made settable. In fact, it will not even matter if this instantiation actually obeys the arrow laws; because the arrow it is built atop does, we can always strip off the wrapper and be left with an arrow that does satisfy the laws. The implementations are shown in Figure 3.10.

The implementations follow directly from the circuit diagrams from Figure 3.7, and thus we will omit any further description of how they function.

Settable

It feels like we could make an $SA\ (\rightsquigarrow)$ settable merely by removing the *SA* wrapper – after all, the underlying arrow will be of the appropriate type. However, this approach limits modularity by forcing the input and output arrows of the *settable* function to be different. Therefore, we instead write a *settable* function for *SA* directly:

```
settable (SA f) = SA $ proc ((b, e_s), e'_s) → do
  (c, s) ← f ≺ (b, merge e_s e'_s)
  returnA ≺ ((c, s), s)
```

This *settable* function is straightforward with one exception. If there is already a state-update event that is propagating a new state (shown here as e'_s), and the settable signal function is also given a state-update

```
arr f = SA $ arr (λ (b, _) → (f b, NoState))

first (SA f) = SA $ proc ((b, d), e_s) → do
  (c, s) ← f ⪦ (b, e_s)
  returnA ⪦ ((c, d), s)

(SA f) >>> (SA g) = SA $ proc (b, e_s) → do
  let (e_l, e_r) = split e_s
  (c, s_l) ← f ⪦ (b, e_l)
  (d, s_r) ← g ⪦ (c, e_r)
  returnA ⪦ (d, join s_l s_r)

loop (SA f) = SA $ proc (b, e_s) → do
  rec ((c, d), s) ← f ⪦ ((b, d), e_s)
  returnA ⪦ (c, s)

delay i = SA $ proc (s_new, e_s) → do
  s_old ← delay i ⪦ s_new
  returnA ⪦ (f s_old e_s, DState (toDyn s_new))
 where f s NoEvent = s
       f _ (Event NoState) = i
       f _ (Event (DState d)) = fromDyn d

left ~(SA f) = SA $ proc (bd, e_s) → do
  rec (s_old, e_old) ← delay (NoState, NoEvent) ⪦ (s_now, e_next)
      let e_now = merge e_s e_old
      (s_now, e_next, cd) ← case bd of
          Left b → do
              (c, s) ← f ⪦ (b, e_now)
              returnA ⪦ (s, NoEvent, Left c)
          Right d → returnA ⪦ (s_old, e_now, Right d)
  returnA ⪦ (cd, s_now)
```

Figure 3.10: *SA* implementations of the Arrow class functions.

event (e_s), which one takes precedence? In fact, the new one must take precedence in order to guarantee the laws we set out in Figure 3.4.

Implementation in Practice

The implementation described in this section has been fully achieved in Haskell and can be found at `https://github.com/dwincort/SettableArrow`. That said, the library's source code is actually slightly different than the code shown in the previous subsection because the code we have provided has a significant performance overhead.

The biggest problem with settability comes from the fact that making a signal function settable causes it to expand considerably, and that each composition especially has a costly overhead. For instance, if one uses the arrow function *second* and then attempts to make it settable, then *second* f is first expanded to:

$$arr\ swap \ggg first\ f \ggg arr\ swap$$

(for a pure definition of *swap*). Thus, one use of *second* causes two uses of composition, which each in turn need to be made settable.

Thus, the major difference between what we have shown here and the code in the SettableArrow library is that the library code has been hand-optimized to our best ability. First, we removed the arrow syntax, replacing it entirely with the arrow combinators themselves. Second, we additionally define the derivable arrow operators such as *second* and *right* (with the obvious, expected definitions). Third, we introduce a lifting operator:

$$\begin{aligned} &uncheckedSA :: Arrow\ (\rightsquigarrow) \Rightarrow (b \rightsquigarrow c) \to SA\ (\rightsquigarrow)\ b\ c \\ &uncheckedSA\ a = SA\ \$\ first\ a \ggg (second\ \$\ constA\ NoState) \end{aligned}$$

This function is useful in the special cases where the user knows that a function has no internal state (or potentially when internal state will never need to be set). Instead of recursively applying the settable transformation, this function simply ignores any incoming state and returns *NoState* as output. Thus, when the performance cost of using *settable* is otherwise too high, one can use *uncheckedSA* judiciously to reduce the amount of slower transformed code.

With these optimizations (excluding using *uncheckedSA*), we find that the performance cost is typically about 2-3x.

3.5 Optimizations

Providing such an expressive, first-order alternative to the higher-order switch function is a boon for optimizations as it allows the arrow structure to be fully determinable at compile time. For instance, Causal Commutative Arrows (CCAs) are a particular subclass of arrows that have been shown to be highly optimizable [Liu et al., 2011], but they are restricted to be only first-order. As a demonstration of the optimization capabilities of our work, we extend the Haskell CCA transformation to include non-interfering choice and show the promising results. We begin with a brief overview of CCAs.

3.5.1 Causal Commutative Arrows

Causal Commutative Arrows are arrows that have two additional laws: a commutativity law that essentially states that signal function effects can be reordered at will, and a product law that governs the behavior of the causal operator (the *init* or *delay* operator). With these two laws at their disposal, Liu et al. [2011] describe a transformation that allows an arrow to be reduced to a normal form, which they call the Causal Commutative Normal Form (CCNF), and then even stream fused into a standard function. The authors demonstrate that GHC can then aggressively optimize this, yielding performance increases of orders of magnitude.

The CCA transformation is of particular interest to us as it is what we will be extending to add support for non-interfering choice, but first, we must describe the CCNF. The CCNF of an arrow is either of the form:

$$arr\ f$$

or

$$loop\ (arr\ f \ggg second\ (delay\ i))$$

where f is a pure function and i is a state. We can express these more simply by calling them $Arr\ f$ and $LoopD\ i\ f$. The transformation, then, is the process of reducing an arrow built with the arrow operators into one of these two forms. It is a recursive transformation that applies a set of reduction rules until the normal form is produced.

For instance, if the transformation comes across an arrow of the form *first sf*, then it will recursively reduce *sf* and then choose one of the following two rules based on the result:

$$\begin{aligned} first\ (Arr\ f) &\mapsto Arr\ (f \times id) \\ first\ (LoopD\ i\ f) &\mapsto LoopD\ i\ (juggle\ .\ (f \times id)\ .\ juggle) \end{aligned}$$

where *juggle* is a pure helper function to reorder the inputs and outputs as necessary.

3.5.2 Extending CCA

CCAs already have a mechanism for dealing with choice, and at first glance, it appears to work with non-interfering choice too. However, it is the arrowized recursion that non-interfering choice allows, and not the choice operator directly, that actually poses a problem for the CCA transformation.

As is, the CCA transformation does not support arrowized recursion. Of course, as we mentioned when we introduced it in Section 3.1.5, the standard arrow laws are not guaranteed to support it, so its absense is perfectly sensible. However, the absense of recursion support is *not* due to inability – indeed, with the non-interfering choice law guarding the recursion, we can add that functionality in a straightforward manner.

Intuitively, the presence of arrowized recursion will present us with the following two scenarios:

$$Arr\ f = Arr\ (g\ f)$$

$$LoopD\ i\ f = LoopD\ (j\ i)\ (g\ f)$$

In the first case, we find that a signal function of the form $Arr\ f$ is defined based on that same function f, and the second is the same except for both f and its state i. However, because f and g (and j) are pure functions, this is a trivial relation to solve: indeed the solution to the first form is as simple as applying a fix point operator:

$$f = fix\ g$$

The second form is slightly more complicated as a precise definition would require the use of a coinductive data type for i. That is, we would want a data type such as:

$$\textbf{data}\ StateCCA\ k = S\ (k\ (StateCCA\ k))$$

However, for our purposes, it is acceptable to relax this requirement and instead assume a more powerful *State* data type that can encode arbitrary values (this would be a similar type to the *State* that we used when describing Settability in Section 3.2.2.

3.5.3 Haskell Implementation

We model the Haskell implementation off of the original CCA transformation design. We use Template Haskell along with a clever use of the Arrow type classes to perform a preprocessing step on only the arrowized components. Thus, rather than try to interfere with Haskell's native recursion support, we introduce a new type class to capture it only where we need it:

$$\begin{aligned}&\textbf{class } \mathit{ArrowFix}\ (\rightsquigarrow)\ \textbf{where}\\ &\quad \mathit{afix} :: (b \rightsquigarrow c \to b \rightsquigarrow c) \to b \rightsquigarrow c\end{aligned}$$

The *ArrowFix* type class introduces the *afix* function that acts as a fix point function particularly for arrowized recursion. In practice, we could merely define *afix* to be equivalent to the regular fix point operator, but we will make better use of it for the transformation.

Specifically, when the recursive transformation encounters an arrow of the form $\mathit{afix}\ f$, the first thing it will do is to produce a fresh, unique "hole". The hole (which we represent with $\bullet$) is a special internal data structure that acts like *Arr* or *LoopD* except that instead of holding the function f and state i, it keeps track of the modifying functions g and j. That is, if the hole is an *Arr* form, then we know that we will eventually come to a scenario such as

$$\mathit{Arr}\ f = \mathit{Arr}\ (g\ f)$$

and since f is unknown and will be deduced via the fix point operation, the hole instead keeps track of g. Applying this hole as the argument to f and then recursively running the transformation will reduce the result to one of the two forms we identified in the previous subsection, which we have already shown can be solved easily.

To facilitate this, we create a second set of transformation rules that are nearly identical to the original except that they expect an additional argument. For instance, if the transformation comes across a partial application of a hole, then it will follow one of the following two rules:

$$\begin{aligned}\mathit{first}\ (\bullet_{\mathit{Arr}}\ g) &\mapsto \bullet_{\mathit{Arr}}\ (\lambda\ f \to (g\ f \times \mathit{id}))\\ \mathit{first}\ (\bullet_{\mathit{LoopD}}\ j\ g) &\mapsto \bullet_{\mathit{LoopD}}\ j\ (\lambda\ f \to\\ &\qquad (\mathit{juggle}\ .\ (g\ f \times \mathit{id})\ .\ \mathit{juggle}))\end{aligned}$$

Note the similarities between this and the description for the non-hole version at the end of Section 3.5.1. They are almost identical except for the fact that the hole's arguments are functions of functions.

State

At the end of the previous subsection, we mentioned that we would use an *State* type to encode the arbitrary state component i of a CCNF arrow of the form $\mathit{LoopD}\ i\ f$. Just as we used for the Haskell implementation of Settability, we utilize Haskell's *Dynamic* data type as an all-purpose state wrapper here.

3.5.4 Performance Results

We followed the same procedure for performance testing that Liu et al. [2011] use. That is, for each program, we:

1. Compiled with GHC, which has a built-in translator for arrow syntax.
2. Translated the arrow syntax to arrow combinators using Paterson's *arrowp* pre-processor [Paterson, 2001] and then compiled with GHC.
3. Normalized into CCNF combinators and compiled with GHC.

Name	GHC	arrowp	CCNF	Stream
Dynamic Counters	1.0	1.66	10.91	12.73
Chained Adder	1.0	1.91	4.06	4.29
Chained Integral	1.0	2.17	13.27	15.40

Figure 3.11: Non-Interfering Arrow-Choice CCA Performance Ratio (higher is better)

4. Normalized into CCNF combinators, rewrote in terms of streams, and compiled with GHC using stream fusion.

The three benchmark programs we used are based on the examples from this paper but are simplified. The first uses the *runDynamic* function to run multiple stateful counters at the same time. The second and third use a function similar to *runDynamic* that runs a signal function multiple times but chains the output from one run to the input of the next, essentially linking them together. For the second, we link together a basic, stateless adder, and for the third, we link an integral function.

The programs were compiled and run on an Intel Core i7 machine with GHC version 7.6.3, using the `-O2` optimization. The results are shown in Figure 3.11, where the numbers represent normalized speedup ratios.

In general, the results show a similarly dramatic performance improvement compared with standard CCA. Notably, the performance of the chained adder, although improved in CCNF, does not show nearly the speedup that the others show. We believe this is because the chained adder has no internal state whatsoever, making the pre-processed performance better.

3.6 Other effects of switching from switch

As stated earlier, arrows with switch are fundamentally more powerful than those without. Thus, it was never our goal to demonstrate that non-interfering choice and state settability could provide the tools to replace switch outright, but rather that switch's power is often underutilized, and in those cases, switch can be replaced.

3.6.1 First order

The primary and most important difference between switch and non-interfering choice is that switch is truly higher order while choice is not. This means that while programs with switch can accept streams of signal functions and then run those signal functions, programs with only choice cannot.

3.6.2 Memory Use

One of the main reasons to use switch in a program is to improve performance. Rather than run a signal function when its results are not being used, we can switch it off, reducing unneeded computation. Signal functions that have been switched out will never be restarted and so can be garbage collected to free memory.

With non-interfering choice, we can similarly stop a signal function, but because it might be restarted, it cannot be garbage collected. Rather, once started, it will remain in memory forever. This is a fundamental reason for demonstrating state settability of signal functions: a signal function that is waiting in memory can have its state re-set so that it can behave as a fresh instance of itself. Thus, with proper management of state, we should never be creating new signal functions while others are left for dead but stranded in memory. Therefore, though our system will always use at least as much memory as a version with switch and often

times more, it should be capped by the maximum amount of memory that a comparable switch-based version would use at any one time.

Chapter 4

General Effects in FRP

4.1 Resource Types

As mentioned in the introduction, we wish to treat I/O devices as signal functions. Consider, for example, a MIDI sound synthesizer with type:

$$midiSynth :: Event\ Note \rightsquigarrow ()$$

midiSynth takes a stream of *Note* events as input and synthesizes the appropriate sound of each note. Now consider this code fragment:

$$\_ \leftarrow midiSynth \prec notes_1$$
$$\_ \leftarrow midiSynth \prec notes_2$$

midiSynth is intended to represent a single output device, but there are two occurrences of it above; so what happens? Are the event streams $notes_1$ and $notes_2$ somehow interleaved or non-deterministically joined together? Clearly, there is a problem.

Likewise, we can imagine a similar problem with input. Suppose *keyboard* is intended to produce events for every key press:

$$keyboard :: () \rightsquigarrow Event\ KeyPress$$

Now consider this code fragment:

$$inp_1 \leftarrow keyboard \prec ()$$
$$inp_2 \leftarrow keyboard \prec ()$$

What is the relationship between inp_1 and inp_2? Do they return the same result, or are they different? If they are the same, then individual key presses are generating multiple events, but if they are different, then which one should get the event? Again, there is a problem.

The solution to these problems is to somehow prevent duplication of certain signal functions like those above. To do this, we introduce the notion of a *resource type*. Resource types are essentially a phantom type parameter to each signal function that represents what resources that signal function accesses. We then assert that if two signal functions share even one resource, then they cannot be composed together. Because this is done at the type level, this check is static and can be caught before runtime.

As the resource type is part of the type signature of a signal function, we show it in the type signature as follows: the type $\alpha \overset{R}{\rightsquigarrow} \beta$ is a signal function that "consumes" the resources in set R, while converting a signal of type α into a signal of type β. For the two examples above, adding resource types yields the following signatures:

$$midiSynth :: Event\ Note \overset{\{MidiSynth\}}{\rightsquigarrow} ()$$
$$keyboard :: () \overset{\{Keyboard\}}{\rightsquigarrow} Event\ KeyPress$$

$$\textsc{Ty-Arr}\ \frac{\vdash e : \alpha \to \beta}{\vdash \mathit{arr}\ e : \alpha \overset{\emptyset}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-First}\ \frac{\vdash e : \alpha \overset{R}{\rightsquigarrow} \beta}{\vdash \mathit{first}\ e : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}$$

$$\textsc{Ty-Comp}\ \frac{\vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta \quad \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R \quad R_1 \cap R_2 = \emptyset}{\vdash e_1 \ggg e_2 : \alpha \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Chc}\ \frac{\vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \gamma \quad \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R}{\vdash e_1 \,|||\, e_2 : (\alpha + \beta) \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Loop}\ \frac{\vdash e : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}{\vdash \mathit{loop}\ e : \alpha \overset{R}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-Delay}\ \frac{\vdash e : \alpha}{\vdash \mathit{delay}\ e : \alpha \overset{\emptyset}{\rightsquigarrow} \alpha}$$

$$\textsc{Ty-Switch}\ \frac{\vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} (\beta, \mathit{Event}\ \gamma) \quad \vdash e_2 : \gamma \to (\alpha \overset{R_2}{\rightsquigarrow} \beta) \quad R_1 \cup R_2 = R}{\vdash \mathit{switch}\ e_1\ e_2 : \alpha \overset{R}{\rightsquigarrow} \beta}$$

Figure 4.1: The typing rules for arrow operators with resource types.

With these types, the above code snippets will not type check.

An additional benefit of resource types is that they provide a new level of transparency to the meaning of a function. Where before, the type of a signal function provided only the types of the inputs and outputs, now we also have easy, static access to the entire set of resources that a program may use.

4.1.1 Typing Rules

Because they exist only at the type level, we can conceive of resource types as having no runtime component; therefore, discussing their behavior should reduce to simply examining their effects on typing rules.

In 2.1.1 and Figure 2.1, we mentioned the standard arrow operators. We now must update these operators to include resource types, and we show the typing rules for these updated operators in Figure 4.1. Note that because we are viewing resource types as a purely type-level entity, the behaviors of these operators will not change. Rather, our new rules will simply apply extra restrictions to make programs that improperly use resources produce type errors.

We will examine each of these typing rules in depth:

- The Ty-Arr rule states that the set of resource types for a pure function lifted to a signal function is empty. Obviously, there can be no resource use in a pure function.
- The Ty-First rule states that transforming a signal function using *first* does not alter the resource type.

- The TY-COMP rule states that when two signal functions are composed, their resource types must be disjoint, and the resulting resource type set is the union of the two. This is exactly the behavior we outlined in the previous section, and it is in this rule that resource types show their power.

- The TY-CHC rule is for the choice operator. The resulting resource type set is the union of those of its inputs, which are not required to be disjoint. Unlike with composition, where the argument signal functions will both be used, at any given moment, only one argument to choice can be active. Therefore, there will be no resource conflicts between the two arguments, and we need not worry about whether the resources are disjoint. Of course, because we cannot know which branch will be active, the result's resource type set still must be the union of the arguments'.

- The TY-LOOP rule states that looping a signal function does not alter the resource type. Looping allows data to be fedback through a signal function, but it has no inherent resource usage.

- The TY-DELAY rule states that the set of resource types for a stateful delay function is empty. Although we may think of a delay operation as using memory and memory usage as a resource-worthy effect, each use of *delay* will create a new piece of memory, and there will never be contention between them.

- There are many different varieties of switch, as we described earlier, but they are all related. Therefore, it suffices to show the typing rule for the simplest one. The TY-SWITCH rule, similar to the rule for choice, states that the union of the resources of the two arguments make up the resource type of the result. The reasoning is much the same as for choice: the two signal functions in the arguments can never be active at the same time, so their resource types can overlap without contention.

4.1.2 Where Do Resources Come From?

When we introduced resource types above, we used an example with a MIDI synthesizer. We stated that the synthesizer could be represented as a signal function, and then to make it safer, we "tag" it with a *MidiSynth* resource type. Here we explore what that actually means and the connection between the resource type and the resource it indicates.

Concretely, we think of resources as devices that perform *effects*, or that collect some sort of input and provides some output to the world. In Haskell, one would typically achieve this sort of effect by utilizing the *IO* monad. Thus, the synthesizer might support a function such as $midiSynthM :: Note \rightarrow IO\ ()$; this monadic action would send individual notes to the synthesizer and return a unit response. To lift this to the realm of signal functions, we would make use of the Kleisli arrow (or equivalent).

From there, we must manually tag low level signal functions that access resources with the appropriate resource types. At first this may seem unsafe, but because this would be done by the language or library designer rather than the end programmer, our end programming safety guarantees are unaffected. Thus, the designer can create a basis of signal functions that are typed to overlap in resources where appropriate such that the programmer can use to build resource-safe applications.

Although this approach is possible, it has two irritating problems. First, it conflates the design of resource types with the domain in which they are being used. That is, a language specification will be forced to have many built-in signal functions to cover the range of resources that the language requires. Second, there is still a small disconnect between the resource types and the resource actions. When a program has a particular resource in its resource type, there is no clear definition of what effect that resource will have. This means that it is possible for a designer to mis-mark a signal function with the wrong resource type, and there is no easy way to detect the error. That is, confining resource types to solely the type level restricts their ability to connect to the program.

4.1.3 Activating Resources

In order to provide a clear connection between virtual resources and the signal functions that use them, we provide a direct operator that *activates* resources. We do this by introducing a new fundamental arrow operator: *rsf* (to be read as "resource signal function").

The *rsf* operator takes a resource as an argument and uses the nature of that resource to construct a signal function. This means that we can no longer think of resources strictly as phantom types, but rather, they will have real substance that will have an effect on the execution of a program.

This pattern brings to mind ideas of defunctionalization Reynolds [1972], and indeed, the process here is similar. All resources are declared statically as types, and using *rsf* is tantamount to choosing which fixed resource to activate. The key usefulness of this is that it provides a clear separation between a core component of the language (the *rsf* operator) and information about the environment (the resources themselves).

We can illustrate the behavior of *rsf* with an example, and thus we once again turn our attention to the scenario of a MIDI synthesizer. The point of *rsf* is that because the *MidiSynth* resource type is available in the environment, the user will be able to use the signal function:

$$\mathit{rsf}\ \mathit{MidiSynth} :: \mathit{Event}\ \mathit{Note} \overset{\{\mathit{MidiSynth}\}}{\rightsquigarrow} ()$$

What exactly is the type of *rsf* itself? That will depend on the resource. In this example, the resource is one that consumes *Note* events and produces $()$, but other resources may be different.

4.1.4 Virtual Resources

The resources we have examined so far are all associated with concrete, real-world devices. Thus, all the resource types are pre-defined and not dependent on any particular program. However, there is no reason why we cannot introduce "virtual" resources during execution, and in fact, this is precisely what we must do to support wormholes.

As mentioned in Section 1.2.4, a wormhole is a way to transfer information non-locally, and it behaves as a mutable reference in memory where the writing end (the whitehole) and the reading end (the blackhole) can be separated. In order to ensure their safe usage, these two ends must be accessed no more than once, and we use resource types to enforce this restriction. Thus, upon introducing a wormhole, we must also introduce two fresh resources.

Unlike global resources that represent real-world devices, these virtual resources have a limited scope in which they function, and outside of that scope, they should disappear. In this sense, their introduction is a lot like a **let** construct, and it is this similarity that leads us to name the wormhole introduction operator **letW**. Within the scope of the **letW** statement, there are two additional resources that can be used by *rsf*, and at its conclusion, those resources are removed from the resource environment.

One important contribution we make in the design of wormholes is recognizing that the order of execution of a wormhole affects program behavior. One could allow the read and write from a wormhole to happen in either order, but this allows two nearly identical programs to potentially have very different behaviors. We show that restricting wormholes such that the read always happens before the write allows sounder reasoning as well as introduces a new possibility for control flow. Intuitively, regardless of the structure of a program, we want the read to be immediate while the write takes place "between" time steps. In this way, we can be sure that any data read from a wormhole was generated in the previous time step, allowing us to use wormholes to create causal connections.

4.1.5 Resource Commutativity

We introduced resource types in order to address the question of what happens if the same resource is accessed more than once at the same time. However, we can broaden this question to ask what the observable

effect of accessing two different resources at the same time should be. The fundamental abstraction of FRP indicates that that functionality should be perceived as instantaneous; thus, any resource effects should also be instantaneous. This implies that the order of multiple resource interactions should not matter, or that resource access must be *commutative*.

This is a natural conclusion, and we can use it to describe resource types in a slightly different way: the purpose of resource types is to allow only effects whose ordering can be commuted. That is, if two effects require an ordering (for instance, if the same effect is performed twice in succession), then it cannot be permitted. This concept extends to wormhole resources as well: the ordering of the blackhole and whitehole must not matter, and thus they are specifically designed so that the whitehole only relies on *past* inputs to the blackhole.

This means that if an arrow itself is commutative, that adding resource interaction governed by resource types to it will not affect its commutativity. Indeed, much like how the causal commutative arrow (CCA) transformation [Liu et al., 2011] reorders an arrow to group all stateful (e.g. *init*) effects separate from pure computation, one could use the same techniques to group all resource effects. In fact, because wormholes can be used to store state (as we shall discuss in more detail in Section 4.4), a resource version of the CCA transformation would be strictly stronger than the traditional one.

4.2 A Resource Typed Language

Because we are storing key program information in the resource types and using them as both types and values, it no longer suffices to simply provide some new typing rules for our new operators (*rsf* and **letW**). In this section we will explore the foundations of a language that fully integrates resources.

4.2.1 Language Definition

We start with $\mathscr{L}\{\rightarrow\times+\}$, the basic lambda calculus extended with product and sum types and general recursion that we introduced in Chapter 2.2. From there, we add the type for resource-typed, arrow-based signal functions, and we add expressions for the three standard operators for them (*arr*, *first*, and $\ggg$) as well as choice ($|||$), loop, and delay. In the process, we also add resources as a new component to the language, complete with types for resources and a resource environment. Finally, we connect the resources by adding our new introduction (**letW**) and application (*rsf*) operators.

We show our extension to $\mathscr{L}\{\rightarrow\times+\}$'s abstract syntax in Figure 4.2 and the typing rules for the newly added expressions in Figure 4.3. In addition to the previous syntax, we let rs range over resources, ts over resource types, and $\mathscr{R}$s over resource environments. A type judgment $\mathscr{R} \vdash r :: t$ indicates that resource environment $\mathscr{R}$ contains an entry mapping resource r to resource type t. Typically, we will combine judgments to the form $\Gamma, \mathscr{R} \vdash \ldots$ indicating that both environments may be used.

Lastly, we make the following definition of programs that our language supports at the top level:

Definition 1 (Program)**.** *An expression p is a **program** if it has type $() \overset{R}{\rightsquigarrow} ()$ for some set of resources R.*

This restriction is actually rather minor. As our language is defined for FRP, it is reasonable to require that the expression being run is a signal function. Furthermore, as all input and output for a program should be handled through resources, the input and output streams of a program need not contain any information.

4.2.2 Resources

Resources can be thought of as infinite streams of data that correspond with real world objects, and the default resource environment, $\mathscr{R}_o$, is essentially the real world (i.e. user and outside data interaction) split

Res	r			
RTp	t	::=	$\langle \tau_{in}, \tau_{out} \rangle$	resource type
Typ	τ	::=	...	
		|	$\tau_1 \overset{\{r_1,\ldots\}}{\rightsquigarrow} \tau_2$	resource typed SF
Exp	e	::=	...	
		|	*arr e*	SF construction
		|	*first e*	SF partial application
		|	$e_1 \ggg e_2$	SF composition
		|	$e_1 \mid\mid\mid e_2$	SF choice
		|	*loop e*	SF looping
		|	*delay e*	SF delay
		|	*rsf r*	SF resource application
		|	**letW** r_w r_b e_i **in** e	wormhole introduction
REn	$\mathscr{R}$	::=	$r_1 :: t_1, \ldots, r_n :: t_n$	resource environment

Figure 4.2: The resource type abstract syntax additions to $\mathscr{L}\{\rightarrow\times+\}$.

up into discrete, quantized pieces, but new "virtual" resources can be added to resource environments via wormholes.

Resources are used at both the type level and the expression level. At the type level, resources are associated with the signal functions that use them. Specifically, they are included in the set of resources that is part of the type of signal functions.

At the expression level, resources can be accessed for input and output via the *rsf* expression, which essentially lifts a resource into a signal function tagged with a type level version of that resource such that the input type of the signal function is the input type of the resource and the output type is similarly the output type of the resource. All resource interaction, and thus all I/O, is done via *rsf*.

The purpose of resources is to track I/O; therefore, despite the fact that they are "usable" at the expression level, we do not want them to escape through an abstraction and so we do not think of them as typical first-class values.

4.2.3 Signal Function Expressions

It's worth noting that these typing rules are almost identical to the ones from Figure 4.1. The only change is that because we have a more well-specified language, we have typing environments Γ and $\mathscr{R}$ to use, and indeed, our rule for *rsf* makes use of the resource type environment $\mathscr{R}$. Specifically:

- The TY-RSF rule says that the input and output types of the signal function that interacts with a given resource must match the input and output types given by the form of the resource. Furthermore, the signal function created will have the singleton resource type set containing the used resource.

- The TY-WH says that the body of the wormhole is a signal function provided that two resources are added to $\mathscr{R}$: one of the form $\langle (), \tau \rangle$ (the whitehole) and one of the form $\langle \tau, () \rangle$ (the blackhole) where τ is the type of the initializing expression. The result of the whole expression is the same as that of the body except that the resources r_w and r_b are removed from the resource set. This omission is valid because the virtual resources cannot escape the wormhole expression.[1]

[1]This is similar to a trick used in Haskell to hide monadic effects by using the universal type quantifier forall to constrain the

$$\textsc{Ty-Arr}\ \frac{\Gamma, \mathscr{R} \vdash e : \alpha \to \beta}{\Gamma, \mathscr{R} \vdash \mathit{arr}\ e : \alpha \overset{\emptyset}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-First}\ \frac{\Gamma, \mathscr{R} \vdash e : \alpha \overset{R}{\rightsquigarrow} \beta}{\Gamma, \mathscr{R} \vdash \mathit{first}\ e : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}$$

$$\textsc{Ty-Comp}\ \frac{\Gamma, \mathscr{R} \vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta \quad \Gamma, \mathscr{R} \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R \quad R_1 \cap R_2 = \emptyset}{\Gamma, \mathscr{R} \vdash e_1 \ggg e_2 : \alpha \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Chc}\ \frac{\Gamma, \mathscr{R} \vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \gamma \quad \Gamma, \mathscr{R} \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R}{\Gamma, \mathscr{R} \vdash e_1 \,|||\, e_2 : (\alpha + \beta) \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Loop}\ \frac{\Gamma, \mathscr{R} \vdash e : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}{\Gamma, \mathscr{R} \vdash \mathit{loop}\ e : \alpha \overset{R}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-Delay}\ \frac{\Gamma, \mathscr{R} \vdash e : \alpha}{\Gamma, \mathscr{R} \vdash \mathit{delay}\ e : \alpha \overset{\emptyset}{\rightsquigarrow} \alpha}$$

$$\textsc{Ty-RSF}\ \frac{}{\Gamma, \mathscr{R}(r : \langle \tau_{in}, \tau_{out} \rangle) \vdash \mathit{rsf}\ r : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}$$

$$\textsc{Ty-WH}\ \frac{\Gamma, \mathscr{R}(r_w : \langle (), \tau \rangle, r_b : \langle \tau, () \rangle) \vdash e : \alpha \overset{R'}{\rightsquigarrow} \beta \quad \Gamma, \mathscr{R} \vdash e_i : \tau \quad R = R' \setminus \{r_w, r_b\}}{\Gamma, \mathscr{R} \vdash \mathbf{letW}\ r_w\ r_b\ e_i\ \mathbf{in}\ e : \alpha \overset{R}{\rightsquigarrow} \beta}$$

Figure 4.3: The typing rules for the new expressions.

4.2.4 No More Switching

An astute reader will note the removal of the switch operator. Indeed, with the addition of wormholes to the language, switching is no longer safe.

To illustrate this point, we can consider a simple example. Switch allows one to take values at the signal level and convert them into values at the signal function level. For instance, one could imagine a signal function such as the following:

$$switchSF :: \alpha \overset{\emptyset}{\rightsquigarrow} Event\ (\beta \overset{R}{\rightsquigarrow} \gamma)$$

This signal function takes values of type α and produces events of signal functions of type $\beta \overset{R}{\rightsquigarrow} \gamma$, all without using any resources itself. However, what if one of those produced signal function output events used one or both ends of a wormhole? The *switchSF* signal function could escape the **letW** scope, but then if we were to switch into its argument, we would be given access to the wormhole resources. Switch's ability to allow wormhole resources to escape their scope makes it dangerous to us.

There are ways to address this. For instance, we could restrict switch so that the switched in signal function is not allowed to use any resource types. Alternatively, we could refine the definition of a program to only allow resources that are in $\mathscr{R}_o$, forcing all wormhole resources to be "cleaned up" before the program could run. However, to enhance clarity in our further discussion, and because we proved in the previous chapter how so much of switch's behaviors can be achieved by choice, we instead choose to omit switch from our resource-typed language.

4.3 Examples

We have extended a simple arrowized model of FRP by introducing resource types as a means to achieve regulated side effects and wormholes to provide a form of non-local communication. We demonstrate the usefulness of these concepts with a few examples derived from two different FRP domains; the first examples will demonstrate resource types in general, and the last two will focus on wormholes.

The examples will generally use the arrow syntax rather than the more abstract arrow combinators to make their behavior clearer. Additionally, we will assume a few basic data types such as numbers and Boolean values as well as typical operators over them.

4.3.1 Composition

For our first example, we will look at how resource types behave under signal function composition. As the typing rules make clear, a signal function cannot compose with another signal function that it shares a resource with, but it is allowed to compose with one in which it does not share a resource. For this example, we will use music as our domain (common in e.g. Euterpea [Hudak, 2014]), and explore the practice of connecting multiple MIDI devices [2].

Although MIDI devices typically have separate unrelated streams for input and output, many devices can be set to act as stream transformers that instead add notes produced by the device in real time to the input stream. This is especially useful in cases where one has many MIDI devices but a limited quantity of ports to connect them to the computer. In cases like these, one can "daisy chain" the devices, or connect them together in sequence, to gather all of the MIDI events produced into one large set. In this model, MIDI resources would have the type:

$$\langle Event\ MidiData, Event\ MidiData \rangle$$

scope. Here, the resources are only available inside the body of the wormhole.

[2] MIDI stands for "Musical Instrument Digital Interface" and it is a standard protocol for communication between electronic instruments and computers.

Although we are not limited by number of ports in a virtual setting, we can still *virtually* daisy chain multiple devices in order to apply a single operation uniformly across all of the MIDI events.

For example, here is a signal function that daisy chains three MIDI keyboards together and then transposes all of the notes they produce by a given number of steps:

$$
\begin{aligned}
&\mathit{daisy} :: \mathit{Number} \to \left(\mathit{Event\ MidiData} \overset{\{\mathit{MIDI}_1,\mathit{MIDI}_2,\mathit{MIDI}_3\}}{\rightsquigarrow} \mathit{Event\ MidiData}\right)\\
&\mathit{daisy}\ n = \textbf{proc}\ \mathit{notes}_{in} \to \textbf{do}\\
&\quad \mathit{notes}_2 \leftarrow \mathit{rsf}\ \mathit{MIDI}_1 \prec \mathit{notes}_{in}\\
&\quad \mathit{notes}_3 \leftarrow \mathit{rsf}\ \mathit{MIDI}_2 \prec \mathit{notes}_2\\
&\quad \mathit{notes}_4 \leftarrow \mathit{rsf}\ \mathit{MIDI}_3 \prec \mathit{notes}_3\\
&\quad \mathit{returnA} \prec \mathit{transpose}\ n\ \mathit{notes}_4
\end{aligned}
$$

If we had accidentally used the same MIDI device more than once, the program would result in a type error. Thus, the disjoint resource types ensure that the different devices are kept distinct, just like in the real world.

4.3.2 Recursion

Sticking with MIDI and the musical domain, we can define a signal function that creates an "echo" effect for notes played on a MIDI device. We achieve this by delaying and looping the notes back through the device itself, attenuating each note by some percentage on each loop:

$$
\begin{aligned}
&\mathit{echo} :: (\mathit{Number}, \mathit{Number}) \overset{\{\mathit{MIDI}_1\}}{\rightsquigarrow} \mathit{Event\ MidiData}\\
&\mathit{echo} = \textbf{proc}\ (\mathit{rate}, \mathit{freq}) \to \textbf{do}\\
&\quad \textbf{rec}\ \ \mathit{notes}_{out} \leftarrow \mathit{rsf}\ \mathit{MIDI}_1 \prec \mathit{notes}\\
&\qquad\quad \mathit{notes} \quad \leftarrow \mathit{delayT} \quad \prec (1.0/\mathit{freq},\ \mathit{decay\ rate}\ \mathit{notes}_{out})\\
&\quad \mathit{returnA} \prec \mathit{notes}_{out}
\end{aligned}
$$

Note the use of the **rec** keyword, which will induce the loop operator and rule (from Section 4.2.3).

The *echo* signal function takes a decay rate and frequency as time varying arguments and uses them to add an echo to the notes played on the MIDI device. It uses two helper functions: *decay rate ns* attenuates each note in *ns* by *rate*, dropping notes when their volume falls below an audible threshold; and $\mathit{delayT} \prec (t, \mathit{ns})$ delays each event in *ns* by the amount of time t.

4.3.3 Conditionals

As mentioned earlier, signal function composition requires that the resource types of the arguments be *disjoint*. However, for conditionals (i.e. case statements), the proper semantics is to take the *natural union* of the resource types. Consider the following functions for sending sound data to speakers:

$$
\begin{aligned}
&\mathit{playLeft} \quad :: \mathit{Sound} \overset{\{\mathit{Speaker}_L\}}{\rightsquigarrow} ()\\
&\mathit{playRight} \quad :: \mathit{Sound} \overset{\{\mathit{Speaker}_R\}}{\rightsquigarrow} ()\\
&\mathit{playStereo} :: \mathit{Sound} \overset{\{\mathit{Speaker}_L,\mathit{Speaker}_R\}}{\rightsquigarrow} ()
\end{aligned}
$$

We can use these to define a signal function for routing sound to the proper speaker (often called a demultiplexer):

$$\textbf{data}\ \mathit{SpeakerChoice} = \mathit{Left} \parallel \mathit{Right} \parallel \mathit{Stereo}$$

$$\begin{aligned}
&\mathit{routeSound} :: (\mathit{SpeakerChoice}, \mathit{Sound}) \overset{\{\mathit{Speaker}_L, \mathit{Speaker}_R\}}{\rightsquigarrow} () \\
&\mathit{routeSound} = \textbf{proc}\ (\mathit{sc}, \mathit{sound}) \to \textbf{do} \\
&\quad \textbf{case}\ \mathit{sc}\ \textbf{of} \\
&\qquad \mathit{Left} \to \mathit{playLeft} \prec \mathit{sound} \\
&\qquad \mathit{Right} \to \mathit{playRight} \prec \mathit{sound} \\
&\qquad \mathit{Stereo} \to \mathit{playStereo} \prec \mathit{sound}
\end{aligned}$$

This is well typed, since the case statement in arrow syntax invokes the inference rule for the choice operator ($|||$).

The *routeSound* signal function may only make use of one speaker at a time, but it feels natural that it should acquire both the $\mathit{Speaker}_L$ and $\mathit{Speaker}_R$ resource types, because we cannot know at compile time which speakers will be used. Furthermore, even though different branches of the case statement use the *same* resources, those resources will never be used more than once *simultaneously*.

4.3.4 Clarifying Domains

An added feature of resource types is that they increase the transparency of code. For example, consider the following non-resource-typed program designed to control a simple robot:

$$\begin{aligned}
&\mathit{controlRobot} :: \mathit{Bool} \rightsquigarrow (\mathit{Double}, \mathit{Double}) \\
&\mathit{controlRobot} = \textbf{proc}\ b \to \textbf{do} \\
&\quad \mathit{returnA} \prec \textbf{if}\ b\ \textbf{then}\ (-5, 0)\ \textbf{else}\ (10, 10)
\end{aligned}$$

Without documentation, it is near impossible to discern what this program's purpose is. In fact, this program was designed to control a robotic car that has two motors (one to control each front wheel) and a bump sensor on the front. The sensor provides a *Bool* value to show its status, and the motors each take a *Double* argument that control their speed. On the whole, *controlRobot* makes the robot go straight unless its bump sensor is hit, at which point it does a brief turn in reverse before continuing straight again.

The problem with *controlRobot* is what we have referred to as the *I/O bottleneck*. Running the program that utilizes *controlRobot* probably looks something like:

$$\begin{aligned}
&\mathit{repeatForever}\ \$\ \textbf{do} \\
&\quad \mathit{inp} \leftarrow \mathit{runIO}_{\mathit{input}} \\
&\quad \mathit{runIO}_{\mathit{output}}\ (\mathit{tick}\ \mathit{controlRobot}\ \mathit{inp})
\end{aligned}$$

We have two runIO_* commands performing effects, and we are stepping the *controlRobot* signal function forward in a pure way. This creates a conceptual (and code-level) gap between where any data is produced and where it is used. With resource types, the input and output devices can be consolidated *into the signal function itself*, making the function of the program much clearer. Consider the following:

$$\begin{aligned}
\mathit{Motor}_L &:: \langle \mathit{Double}, () \rangle \\
\mathit{Motor}_R &:: \langle \mathit{Double}, () \rangle \\
\mathit{Sensor}_{\mathit{Bump}} &:: \langle (), \mathit{Bool} \rangle
\end{aligned}$$

$$
\begin{aligned}
&R = \{Motor_L, Motor_R, Sensor_{Bump}\}\\
&controlRobot_R :: () \overset{R}{\rightsquigarrow} ()\\
&controlRobot_R = \textbf{proc}\ () \to \textbf{do}\\
&\quad b \leftarrow rsf\ Sensor_{Bump} \prec ()\\
&\quad \textbf{if}\ b\ \textbf{then}\ \ rsf\ Motor_L \prec -5\\
&\qquad\quad \textbf{else}\ \ \textbf{do}\ rsf\ Motor_L \prec 10\\
&\qquad\qquad\quad\ \ rsf\ Motor_R \prec 10
\end{aligned}
$$

We can see clearly what $controlRobot_R$ does—the type shows us what resources are being used, and they are being used alongside where they are produced, within the signal function itself. Furthermore, we know that, for example, if we want to add a command to the right motor when the bump sensor is **True**, we can. However, if we want to do that when the bump sensor is **False**, we will have a type error—if we must, we know that we need to rewrite code rather than simply add it.

Let's now consider a more complicated program. In a monadic framework, functions controlling a robot might look like the following:

$$
\begin{aligned}
&moveArmUp &&:: IO\ ()\\
&moveArmDown &&:: IO\ ()\\
&moveArmLeft &&:: IO\ ()\\
&moveArmRight &&:: IO\ ()\\
&clawGrab &&:: IO\ ()\\
&clawRelease &&:: IO\ ()
\end{aligned}
$$

These functions activate various motors in the robot arm to move the arm as expected. We might also have a compound function: $toss :: IO\ ()$. The documentation for *toss* says that it moves the arm while releasing the claw to toss whatever the claw was holding. Perhaps we have tested *toss*, and now want to make the tossed object go higher, so we would like to additionally run *moveArmUp* in parallel with *toss*. Is this a good idea? We don't actually know how *toss* works; if it calls *moveArmDown* internally, that could result in a motor conflict. With resource types, these functions all become much clearer. Consider the following:

$$
\begin{aligned}
&moveArmUp_{SF} &&:: \alpha \overset{\{Motor_{Vertical}\}}{\rightsquigarrow} ()\\
&moveArmDown_{SF} &&:: \alpha \overset{\{Motor_{Vertical}\}}{\rightsquigarrow} ()\\
&moveArmLeft_{SF} &&:: \alpha \overset{\{Motor_{Horizontal}\}}{\rightsquigarrow} ()\\
&moveArmRight_{SF} &&:: \alpha \overset{\{Motor_{Horizontal}\}}{\rightsquigarrow} ()\\
&clawGrab_{SF} &&:: \alpha \overset{\{Motor_{Claw}\}}{\rightsquigarrow} ()\\
&clawRelease_{SF} &&:: \alpha \overset{\{Motor_{Claw}\}}{\rightsquigarrow} ()
\end{aligned}
$$

$$
toss_{SF} :: \alpha \overset{\{Motor_{Vertical}, Motor_{Claw}\}}{\rightsquigarrow} ()
$$

Now, not only is it clear which motors $toss_{SF}$ uses, but trying to run $toss_{SF}$ at the same time as $moveArmUp_{SF}$ will result in a type error.

4.3.5 Data transfer

One strength of wormholes is their ability to transfer data between two disparate parts of a program. Typically, this would involve rewriting signal functions so that they consume or produce more streams so that one can create a stream link between the two components to be connected. However, this work is unnecessary with wormholes.

We will consider the following two programs:

$$P_1 \;::\; R_1' \subseteq R_1 \Rightarrow (\mathit{Integer} \overset{R_1'}{\rightsquigarrow} \mathit{Integer}) \to (() \overset{R_1}{\rightsquigarrow} ())$$
$$P_2 \;::\; R_2' \subseteq R_2 \Rightarrow (\mathit{Integer} \overset{R_2'}{\rightsquigarrow} \mathit{Integer}) \to (() \overset{R_2}{\rightsquigarrow} ())$$

We will assume that as long as R_1' and R_2' are disjoint, then R_1 and R_2 are disjoint also. These two programs both do almost the same thing: they acquire a stream of *Integer*s from a source, apply a given signal function to them, and then send the result to an output device.

Our goal is to connect these two programs in order to cross their streams. That is, we would like the stream from P_1 to go to the output device of P_2 and vice versa. Without wormholes, we would be forced to examine and change the implementation and type of at least one of these two programs. However, instead, we can define:

$$\begin{aligned} \mathit{main} = \;&\mathbf{letW}\ r_{w_1}\ r_{b_1}\ 0\ \mathbf{in} \\ &\mathbf{letW}\ r_{w_2}\ r_{b_2}\ 0\ \mathbf{in} \\ &\quad P_1\ (\mathit{rsf}\ r_{b_1} \ggg \mathit{rsf}\ r_{w_2}) \ggg \\ &\quad P_2\ (\mathit{rsf}\ r_{b_2} \ggg \mathit{rsf}\ r_{w_1}) \end{aligned}$$

We pair two wormholes together almost like two *delay* expressions, except that we swap the inputs and outputs. This provides us with two functions that are able to communicate even when no streams seem readily available.

4.4 Delay and Loop

We have provided looping as a built-in feature via the *loop* arrow operator, and in our introduction (Section 2.1.2), we described that its use in FRP will always be paired with an associated use of *delay* to enforce causality. With wormholes, these two functions are no longer fundamental but instead can be constructed.

We start by showing that a strictly causal implementation of *delay* can be produced as syntactic sugar with a wormhole:

$$\textsc{Ty-Delay}\ \frac{\Gamma, \mathscr{R} \vdash e_i : \alpha}{\Gamma, \mathscr{R} \vdash \mathit{delay}\ e_i : \alpha \overset{\emptyset}{\rightsquigarrow} \alpha}$$

$$\mathit{delay}\ i \overset{\mathbf{def}}{=} \mathbf{letW}\ r_w\ r_b\ i\ \mathbf{in}\ \mathit{rsf}\ r_b \ggg \mathit{rsf}\ r_w$$

By attaching the blackhole and whitehole of a wormhole back to back, we create a signal function that accepts present input and returns output delayed by one step. Essentially, we see that the *delay* operator is the connection of two ends of a wormhole.

Interestingly, we can attach the wormhole ends the other way too. Obviously, this can lead to a trivial signal function of type $() \overset{\emptyset}{\rightsquigarrow} ()$ that does nothing, but if we provide a signal function to be run in between the connection, we can build the following:

$$\textsc{Ty-dLoop}\ \frac{\Gamma, \mathscr{R} \vdash e_i : \gamma \quad \Gamma, \mathscr{R} \vdash e : (\gamma \times \alpha) \overset{R}{\rightsquigarrow} (\gamma \times \beta)}{\Gamma, \mathscr{R} \vdash \mathit{dloop}\ e_i\ e : \alpha \overset{R}{\rightsquigarrow} \beta}$$

$$\begin{aligned} \mathit{dLoop}\ i\ e \overset{\mathbf{def}}{=} \mathbf{letW}\ r_w\ r_b\ i\ \mathbf{in}\ &\mathbf{proc}\ a \to \mathbf{do} \\ &x \leftarrow \mathit{rsf}\ r_w \prec () \\ &(y, b) \leftarrow e \prec (x, a) \\ &\mathit{rsf}\ r_b \prec y \\ &\mathit{returnA} \prec b \end{aligned}$$

We are able to achieve a delayed form of looping by a clever use of a wormhole. We first produce a new wormhole and provide the loop's initialization value as its initial value. We extract the loop data x from the wormhole by accessing the whitehole, feed that along with the input value a to the signal function e, we loop the resulting loop data y by sending it to the blackhole, and finally we return the generated value b. Due to the causal behavior of wormholes, b values that are output from e become new a input values to e on the next iteration. Thus, the input on the n^{th} iteration is given by the output on the $n-1^{\text{st}}$ iteration.

With the results of this example, we no longer need to provide looping or delay as fundamental operators in our language.

4.4.1 Wormhole Loop Syntax

There is one problem with using wormholes for looping, which is that doing it in practice often feels somewhat imperative. The nature of explicitly writing to blackholes and reading from whiteholes can obscure the underlying feedback that is occurring.

Actually, a similar problem happens with arrow loop itself, since the *loop* operator is not always easy to use. The solution in this case is that arrow syntax is extended with a **rec** keyword, which allows the programmer to write recursively defined streaming values. This **rec** syntax is then translated into an invocation of the *loop* operator [Paterson, 2001]. Of course, it is possible to rewrite non-causal loops with this syntax, and doing so can create an infinite loop, so one often needs to use a *delay* operator of some sort to prevent this.

We can create a similar system with wormholes. That is, we can create a custom syntax that eases program development and that desugars into standard wormhole creation and application. It will be nearly identical to the arrow loop **rec** syntax in appearance, but it will rely on a different underlying transformation. We will still have a **rec** block, and within that block, values are allowed to be recursively defined. However, rather than simply hoping that the user uses *delay* operators in the appropriate places, we provide a new operator, *introduce*, which we will use in the desugaring.

The *introduce* operator behaves to the user identically to *delay*. That is $introduce :: \alpha \to (\alpha \overset{\emptyset}{\rightsquigarrow} \alpha)$, and it must be used whenever a new recursive value is defined. For instance, if we have a signal function *sf*, and we would like its output fed back to itself as input, with an initial value of 0, then we could write the following in our **proc** syntax:

$$\begin{aligned} \textbf{rec}\ \ & x \leftarrow introduce\ 0 \prec y \\ & y \leftarrow sf \prec x \end{aligned}$$

Basically, the rule of thumb is that streaming values on the right side of an *introduce* do not need to have been defined yet. However, because this syntax is designed for resource typed FRP, which is commutative by design, the real rule is that the value to the right of the *introduce* must simply be defined within (or before) the **rec** block.

The desugaring is as expected. We create a new wormhole and we populate it with initial values gathered from each *introduce* from the **rec** block. Then, at the start of the **rec** block, we read the whitehole, and at the end of it, we write to the blackhole. All of the *introduce* statements are removed post-desugaring.

For the example above, the block will desugar to:

$$\begin{aligned} &\textbf{letW}\ r_w\ r_b\ 0\ \textbf{in} \\ &\quad x \leftarrow rsf\ r_w \prec () \\ &\quad y \leftarrow sf \prec x \\ &\quad () \leftarrow rsf\ r_b \prec y \end{aligned}$$

If there are more than one *introduce* statement in the block, then they can be grouped together: the value stored in the wormhole would be a tuple of all of the *introduce*d values, and they could be read all at once from the whitehole and written all at once to the blackhole.

4.5 Semantics

We provide a discrete, synchronous operational semantics for the resource typed arrowized FRP language we have built. As these semantics are somewhat complex, and in an effort to demystify them, we separate the functionality into three distinct transitions. At the highest level, we apply a temporal transition. This transition details how resources behave over time and explains how the signal function itself is "run". (Recall from Definition 1 that only expressions with type $() \overset{R}{\rightsquigarrow} ()$ are allowed as "runnable" programs.) Because our language is lazy and evaluation is performed when necessary, expressions may be able to simplify themselves over time. Therefore, this transition will return an updated (potentially more evaluated) version of the input program.

The temporal transition makes use of a functional transition to interpret the flow of data through the component signal functions of the program at a given point in time. Thus, the judgments in the functional transition handle how the instantaneous values of the signals are processed by signal functions.

Because the expressions to be run can contain arbitrary lambda calculus, the functional transition judgments make use of an evaluation transition when necessary to evaluate expressions when strictness points are reached. This is a fairly simple transition that performs as a typical, lazy semantics of a lambda calculus.

A top-down view of the three transitions is the most intuitive way to describe their functionality. However, to define them, it is easier to start with the evaluation transition and work up from there. Therefore, we present the following transitions:

$$\begin{array}{rl} e \mapsto e' & \text{Evaluation transition} \\ (\mathscr{V}, x, e) \Rrightarrow (\mathscr{V}', y, e', \mathscr{W}) & \text{Functional transition} \\ (\mathscr{R}, \mathscr{W}, P) \overset{t}{\mapsto} (\mathscr{R}', \mathscr{W}', P') & \text{Temporal transition} \end{array}$$

where

e and e'	are expressions
$\mathscr{V}$ and $\mathscr{V}'$	are sets of triples
x and y	are values
$\mathscr{W}$ and $\mathscr{W}'$	are sets of wormhole data
$\mathscr{R}$ and $\mathscr{R}'$	are resource environments, and
P and P'	are programs

In the following subsections, we discuss these transitions in more detail.

4.5.1 Evaluation transition

The evaluation transition is used to evaluate the non-streaming components of the language. We start by assuming a classic, lazy semantics for lambda expressions and application, product-type pairs and projection, and sum-type case analysis and injection as provided by $\mathscr{L}\{\rightarrow \times +\}$. We show our additional rules for the additional expressions of our language in Figure 4.4. Note that we leave out *delay* and *loop* due to them being implementable via wormholes.

We use the notation e val to denote that expression e is a value and needs no further evaluation.

Obviously, these rules are very straightforward: no evaluation is done on signal functions in this transition. This transition is important for the operations of $\mathscr{L}\{\rightarrow \times +\}$, but it is strictly a formality here.

The language $\mathscr{L}\{\rightarrow \times +\}$ has a standard Canonical Forms Lemma associated with it that explains that for each type, there are only certain expressions that evaluate to a value of that type. By simple examination of these new rules to the transition, we can extend the lemma as follows:

Lemma 1 (Canonical Forms)**.** *If e val and $e : \alpha \overset{R}{\rightsquigarrow} \beta$, then e is either an SF constructor, an SF partial application, an SF composition, an SF choice statement, an SF resource interaction, or a wormhole introduction.*

$$\textsc{Et-Arr}\ \frac{}{arr(e)\ \mathsf{val}}$$

$$\textsc{Et-First}\ \frac{}{first(e)\ \mathsf{val}}$$

$$\textsc{Et-Comp}\ \frac{}{(e_1 \ggg e_2)\ \mathsf{val}}$$

$$\textsc{Et-Chc}\ \frac{}{(e_1 \,|||\, e_2)\ \mathsf{val}}$$

$$\textsc{Et-RSF}\ \frac{}{rsf\ r\ \mathsf{val}}$$

$$\textsc{Et-WH}\ \frac{}{(\textbf{letW}\ r_w\ r_b\ e_i\ \textbf{in}\ \ e)\ \mathsf{val}}$$

Figure 4.4: The evaluation transition judgments for our extension to $\mathscr{L}\{\rightarrow\times+\}$.

4.5.2 Functional transition

The functional transition details how a signal function behaves when given a single step's worth of input. It is a core component of the temporal transition described in the next section as it essentially drives the signal function for an instant of time. It is a big step semantics. The functional transition judgments are shown in Figure 4.5.

Before we discuss the judgments themselves, it is important to examine the components being used. First, one will notice the set $\mathscr{V}$. $\mathscr{V}$ represents the state of the resources (both real and virtual) in the world at the particular moment in time that this transition is taking place. Each element of $\mathscr{V}$ is actually a triple of a resource, the value that resource is providing at this moment, and the value to be returned to that resource. At the start, we assume that all of the elements have the form $(r,x,\cdot)$, which indicates that resource r provides the value x and has no value to receive. It should be no surprise that the only judgments that read from or modify this set are FT-RSF and FT-WH, the judgments for resource interaction and virtual resource creation.

The second argument to each of the judgments (typically x in Figure 4.5) represents the streaming value being piped into the signal function. However, since the functional transition is only defined for an instant of time, rather than this value being an actual stream, it is the instantaneous value on the stream at this time step. Its partner is the second result, or the instantaneous value of the streaming output of the input signal function.

The third argument is the expression being processed. The purpose of the functional transition is to describe how signal functions behave when given values from their streaming input, and as such, it is only defined for signal functions (that is, expressions that have the type $\alpha \overset{R}{\rightsquigarrow} \beta$ for some set R). Notably, there are only judgments corresponding to the forms given in the updated canonical forms lemma (Lemma 1). On the output end, this term represents the potentially further evaluated form of the input expression. We prove later in Theorem 2 that this output expression is functionally equivalent to the input one.

The first three terms of the output correspond to the three terms of the input, but there is also an additional term $\mathscr{W}$, which contains data about any wormholes processed during this transition. In addition to adding the two virtual resources created by a wormhole expression to the resource environment, we need to separately keep track of the fact that they are a pair. Therefore, $\mathscr{W}$ contains elements of the form $[r_b, r_w, e]$ where r_b is the name of the blackhole end of the wormhole, r_w is the name of the whitehole end, and e is the value in the wormhole. We will use this information later to properly update wormholes over time in the temporal transition.

$$\textsc{Ft-Arr}\ \frac{}{(\mathscr{V},x,arr(e)) \Rrightarrow (\mathscr{V},e\ x,arr(e),\emptyset)}$$

$$\textsc{Ft-First}\ \frac{e \mapsto^* e' \quad (\mathscr{V},x,e') \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W})}{(\mathscr{V},(x,z),first(e)) \Rrightarrow (\mathscr{V}',(y,z),first(e''),\mathscr{W})}$$

$$\textsc{Ft-Comp}\ \frac{e_1 \mapsto^* e_1' \quad (\mathscr{V},x,e_1') \Rrightarrow (\mathscr{V}',y,e_1'',\mathscr{W}_1) \quad e_2 \mapsto^* e_2' \quad (\mathscr{V}',y,e_2') \Rrightarrow (\mathscr{V}'',z,e_2'',\mathscr{W}_2)}{(\mathscr{V},x,e_1 \ggg e_2) \Rrightarrow (\mathscr{V}'',z,e_1'' \ggg e_2'',\mathscr{W}_1 \cup \mathscr{W}_2)}$$

$$\textsc{Ft-Chc}_1\ \frac{x \mapsto^* left(x') \quad e_1 \mapsto^* e_1' \quad (\mathscr{V},x',e_1') \Rrightarrow (\mathscr{V}',y,e_1'',\mathscr{W})}{(\mathscr{V},x,e_1 \,|||\, e_2) \Rrightarrow (\mathscr{V}',y,e_1'' \,|||\, e_2,\mathscr{W})}$$

$$\textsc{Ft-Chc}_2\ \frac{x \mapsto^* right(x') \quad e_2 \mapsto^* e_2' \quad (\mathscr{V},x',e_2') \Rrightarrow (\mathscr{V}',y,e_2'',\mathscr{W})}{(\mathscr{V},x,e_1 \,|||\, e_2) \Rrightarrow (\mathscr{V}',y,e_1 \,|||\, e_2'',\mathscr{W})}$$

$$\textsc{Ft-RSF}\ \frac{}{(\mathscr{V} \cup \{(r,y,\cdot)\},x,rsf\ r) \Rrightarrow (\mathscr{V} \cup \{(r,\cdot,x)\},y,rsf\ r,\emptyset)}$$

$$\textsc{Ft-WH}\ \frac{e \mapsto^* e' \quad (\mathscr{V} \cup \{(r_w,e_i,\cdot),(r_b,(),\cdot)\},x,e') \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W})}{(\mathscr{V},x,\textbf{letW}\ r_w\ r_b\ e_i\ \textbf{in}\ \ e) \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W} \cup \{[r_b,r_w,e_i]\})}$$

Figure 4.5: The functional transition judgments.

Note also that we use the term $e \mapsto^* e'$ to denote continued application of the evaluation transition $\mapsto$ on e until it is evaluated to a value. That value is e'

As this is a critical piece of the overall semantics, we examine each of the judgments individually:

- The FT-ARR judgment does not touch the resources, so the input $\mathscr{V}$ is returned untouched in the output. The expression $e\ x$ does not need to be evaluated due to the lazy semantics, but it is the streaming output nonetheless. The final two outputs reveal that no further evaluation of the expression has been done and no wormhole data was created.

- The FT-FIRST judgment is only applicable when the input streaming value is a pair (which is assured by the type checker by using the TY-FIRST rule). The first element of the pair is recursively processed with the argument to *first*, and the output is formed by the updated $\mathscr{V}'$ and by re-pairing the output y. As the body of the *first* expression, e, was evaluated, its updated form is returned along with any wormhole data the recursion generated.

- The FT-COMP judgment first sends the streaming argument x through e_1 recursively. Then, with the updated $\mathscr{V}'$, it sends the result y through e_2. The resulting $\mathscr{V}''$ and z are returned. Once again, the updated expression is returned in the output. Lastly, the wormhole data from both recursive calls of the transition are unioned together and returned.

- The FT-CHC$_1$ judgment is applicable for a signal function choice operation when the streaming argument evaluates to a *left* value. This argument is defined in typing rule TY-CHC to be a sum type. The "left" expression, e_1 is evaluated and a recursive call is made. The output is formed by the updated $\mathscr{V}'$, the new streaming output, the choice operator applied to the updated e_1'' and the original, unevaluated e_2, and any wormhole data from the recursive call.

- The FT-CHC$_2$ judgment proceeds similarly to the FT-CHC$_1$ judgment, but when x is a *right* value instead of a *left* value.

- The FT-RSF judgment requires $\mathscr{V}$ to contain an element of the form $(r,y,\cdot)$, where r is the resource being accessed, y is the value the resource currently has, and no output has been sent to this resource yet. The streaming value x is put into the resource, and the result is the streaming value y from what was in the resource. The set $\mathscr{V}$ is updated, replacing the triple used here with a new one of the form $(r,\cdot,x')$ showing that this resource has essentially been "used up".
- The FT-WH judgment first evaluates its body e to the value e'. For its recursive call, it updates the set $\mathscr{V}$ with two new triples corresponding to the two new resources created in the wormhole operation: $(r_w,e_i,\cdot)$ and $(r_b,(),\cdot)$. These are two fresh, unused triples that *rsf* operators can make use of in the body e'. As triples are never removed, $\mathscr{V}'$ will include these two triples as well. The result is this $\mathscr{V}'$ with the new triples, the streaming value y, the updated body e'', and the wormhole data from the recursion updated with the element $[r_b,r_w,e_i]$ corresponding to this wormhole. Note that the returned expression is no longer a wormhole but has been replaced with the body of the wormhole. This is because now that this wormhole has been evaluated, its values live inside $\mathscr{V}$ and it has been cataloged in $\mathscr{W}$—it is no longer needed in the expression.

The following theorems provide some extra information about the overall functionality of this transition.

Theorem 1 ($\mathscr{V}$ Preservation). *If $(\mathscr{V},x,e)\Rrightarrow(\mathscr{V}',y,e',\mathscr{W})$, then $\forall(r,a,b)\in\mathscr{V}$, $\exists(r,a',b')\in\mathscr{V}'$ and $\forall[r_b,r_w,i]\in\mathscr{W}$, $\exists(r_b,a_b,b_b)\in\mathscr{V}'$ and $\exists(r_w,a_w,b_w)\in\mathscr{V}'$.*

This theorem states that the elements in the input $\mathscr{V}$ are preserved in the output. In fact, there is a direct correspondence between them such that if the input set has an element with resource r, then the output will too. Furthermore, when new values are added (as in FT-WH), they correspond to values in $\mathscr{W}$. The proof is straightforward and proceeds by induction on the functional transition judgments. It has been omitted for brevity.

Theorem 2. *[e Preservation] If $e:\alpha\overset{R}{\rightsquigarrow}\beta$ and $(\mathscr{V},x,e)\Rrightarrow(\mathscr{V}',y,e',\mathscr{W})$, then $e':\alpha\overset{R'}{\rightsquigarrow}\beta$ and e' has the same structure of sub-expressions as e with the exception that wormhole expressions may have been replaced by their bodies. For each so replaced, there is a corresponding element in $\mathscr{W}$ of the form $[r_b,r_w,i]$ such that r_b and r_w are the virtual resources of said wormhole. Furthermore, $R\subseteq R'$ and $\forall r\in(R'\setminus R)$, either $[r,\_,\_]\in\mathscr{W}$ or $[\_,r,\_]\in\mathscr{W}$.*

This theorem states exactly how the output expression e' can be different from the input expression e. Notably, it will still be a signal function with the same input and output types and it will still behave in essentially the same way, but its set of resource types may grow. Specifically, if the resource type set does grow, it is because a wormhole expression was reduced to its body and the virtual resources it introduced are now visible at a higher level. A notable corollary of this theorem is that if $\mathscr{W}=\emptyset$, then $e=e'$.

Proof. The proof follows by induction on the judgments and the typing rule TY-WH for wormholes. A cursory examination of the judgments reveals that the only one to change the form of the expression from input to output is FT-WH, which replaces the input expression with the body of the wormhole. The typing rule tells us that if $e:\alpha\overset{R}{\rightsquigarrow}\beta$ and e is a wormhole, then the body of e has type $\alpha\overset{R'}{\rightsquigarrow}\beta$ where $R=R'\setminus\{r_w,r_b\}$. Although the resource type set may have grown, it could only have grown by the addition of r_b, r_w, or both. Furthermore, the element $[r_b,r_w,e_i]$ is added to the output $\mathscr{W}$. □

Lastly, it may appear that multiple *rsf* commands on the same resource could be problematic; after all, the FT-RSF judgment initially requires the resource r to have a triple of the form $(r,y,\cdot)$, but it results in the third element of the triple being filled in. That is, there is no *rsf* command judgment where the triple has a value in the third element. However, as we prove later in Theorem 3, if the program has type $\alpha\overset{R}{\rightsquigarrow}\beta$, then it must have at most one *rsf* command for any given resource r.

$$\mathscr{V}_{in} = \{(r, \mathsf{read}\ r, \cdot) \mid r \in \mathscr{R}\} \cup \{(r_w, i, \cdot) \mid [r_b, r_w, i] \in \mathscr{W})\} \cup \{(r_b, (), \cdot) \mid [r_b, r_w, i] \in \mathscr{W})\}$$
$$(\mathscr{V}_{in}, (), P) \Rrightarrow (\mathscr{V}_{out}, (), P', \mathscr{W}_{new})$$
$$\mathscr{R}' = \{\mathsf{update}\ r\ o' \mid r \in \mathscr{R}, (r, \_, o) \in \mathscr{V}_{out}, o \mapsto^* o'\}$$
$$\mathscr{W}' = \{[r_b, r_w, \textbf{if}\ o = \cdot\ \textbf{then}\ i\ \textbf{else}\ o] \mid (r_b, \_, o) \in \mathscr{V}_{out}, [r_b, r_w, i] \in (\mathscr{W} \cup \mathscr{W}_{new})\}$$
$$\overline{(\mathscr{R}, \mathscr{W}, P) \overset{t}{\mapsto} (\mathscr{R}', \mathscr{W}', P')}$$

Figure 4.6: The temporal transition.

4.5.3 Temporal transition

Because signal functions act over time, we need a transition to show their temporal behavior. At each time step, we process the program, taking in the state of the world (i.e. all the resources) and returning it updated. There is only one temporal transition, but it is quite complicated. It is shown in Figure 4.6.

This transition says that the resource environment $\mathscr{R}$, the set of wormhole data $\mathscr{W}$, and a program P transition into an updated resource environment, an updated set of wormhole data, and a potentially more evaluated program.

Before we can begin to analyze how the transition behaves on a fine grain level, we first need a method of actually *interacting* with resources. This happens via the use of two functions:

$$\mathsf{read} :: \langle \tau_{in}, \tau_{out} \rangle \to \tau_{out}$$
$$\mathsf{update} :: \langle \tau_{in}, \tau_{out} \rangle \to \tau_{in} \to \langle \tau_{in}, \tau_{out} \rangle$$

The read function simply returns the current output value of the given resource, merely "peeking" at what is there. The update function takes a resource and the value to give to it and returns an updated version of the resource.

The first precondition extracts data from the resources and wormholes and compiles it into a form that the functional transition can use. For the resources, we create triples of the form $(r, \mathsf{read}\ r, \cdot)$ meaning that the resource r provides the value $\mathsf{read}\ r$ and is waiting for a return value. For wormholes, we actually create two triples, one for the blackhole and one for the whitehole. The whitehole uses the whitehole resource name r_w and the current value in the wormhole, and the blackhole uses r_b and produces only $()$.

This data is provided to the functional transition along with the program P. Because P has type $() \overset{R}{\rightsquigarrow} ()$ by definition, the streaming argument is set to $()$. The result of the functional transition is the updated value set $\mathscr{V}_{out}$, the streaming output of P (which the type guarantees to be $()$), the updated program, and a set of any new wormhole data encountered during execution.

The last two preconditions are analogous to the first one: they extract the resource and wormhole data from $\mathscr{V}_{out}$. For every element in $\mathscr{V}_{out}$ that corresponds to a resource in $\mathscr{R}$, we take the output value o, evaluate it, and push it to the resource. The resulting updated resources make up the new set $\mathscr{R}'$. It may be that o was never filled and is still empty—the update operation is executed regardless in order to push the resource one time step into the future. Note that because of the use of the evaluation transition, this step acts as a strictness point for the streaming values of the signal functions.

The wormhole data is extracted in much the same way. For every element in $\mathscr{V}_{out}$ that corresponds to a blackhole in either the original wormhole data set $\mathscr{W}$ or in the new additions $\mathscr{W}_{new}$, we examine the output value o. If o was filled in, then the updated wormhole entry contains the new value; otherwise, the wormhole keeps its old value.

Each application of the temporal transition is designed to represent one moment in time, or one unit time step. We could easily parametrize this transition with an actual δt, or change in time, but this is not

necessary. In fact, one can think of real time itself as a resource whose value can be probed at any moment, and in doing so, the semantic behavior of the transition is allowed to be independent of real time.

In total, we see that the temporal transition uses the program P to update the resources $\mathscr{R}$ and the wormhole data $\mathscr{W}$. Because of Lemma 1, we can see that $\mathscr{R}'$ contains all the resources that $\mathscr{R}$ did, and similarly, $\mathscr{W}'$ contains all of the elements from both $\mathscr{W}$ and $\mathscr{W}_{new}$. Therefore, if $(\mathscr{R},\mathscr{W},P) \overset{t}{\mapsto} (\mathscr{R}',\mathscr{W}',P')$, then this transition can repeat indefinitely. That is, the next step would be $(\mathscr{R}',\mathscr{W}',P') \overset{t}{\mapsto} (\mathscr{R}'',\mathscr{W}'',P'')$ and so on. Since each pass through the transition represents one moment in time, this makes sense as a valid way to represent program execution over time.

We can use the temporal transition to establish an overall semantics for a program P in our language. Recall that $\mathscr{R}_o$ is the default resource environment containing all the resources of the real world.

Definition 2 (Program Evaluation)**.** *If P is a program (that is, an expression of the form $() \overset{R}{\rightsquigarrow} ()$ for some set R), then P will have the infinite trace starting at state $(\mathscr{R}_o,\emptyset,P)$ that uses only the temporal transition $\overset{t}{\mapsto}$.*

4.6 Safety

Here we show the safety that resource typing provides. We intend to show that if a program is well typed, then no two components will compete for the same resource. To express this, we must first define what it means to interact with a resource.

Definition 3 (Resource interaction)**.** *A program P interacts **once** with a resource r at a given time step if it reads the value produced by r at that time step, returns a value to r at that time step, or does both simultaneously.*

With this definition, we can state our resource safety theorem:

Theorem 3 (Resource safety)**.** *If a program $P : \alpha \overset{R}{\rightsquigarrow} \beta$, then P will interact only with resources in R, and for each resource it interacts with, it will do so at most once per time step.*

This theorem tells us that any program that type checks will only use the resources in its type and never have the problem where two components are vying for the same resource. The program will be entirely deterministic in its resource management, and from the type alone, one will be able to see which resources it has the potential to interact with while it runs.

Proof. The proof of resource safety begins by examining the temporal transition. Because each element in $\mathscr{R}$ is a unique resource, we know that interacting once each with different elements in $\mathscr{R}$ will never cause a problem. Furthermore, as all we do to create $\mathscr{R}'$ is exactly one `update` operation on each resource, $\mathscr{R}'$ will likewise have unique resources. The concern, then, comes from the functional transition. We must prove that updates in $\mathscr{V}_{out}$ are not being overwritten by future updates during the functional transition.

Therefore, the bulk of the proof proceeds by induction on the functional transition where we must show that any elements in $\mathscr{V}$ are only being updated at most once. Based on the updated Canonical Forms Lemma (Lemma 1), we know that since $P : \alpha \overset{R}{\rightsquigarrow} \beta$, it must be one of the five SF operators. We examine each in turn:

- *SF constructor:* If P is of the form $arr(e)$, then by typing rule TY-ARR, $R = \emptyset$ and it will use judgment FT-ARR. There are no other transitions nor resource interaction being performed in this judgment, and since $R = \emptyset$, we trivially satisfy our conditions.
- *SF partial application:* If P is of the form $first(e)$, then by typing rule TY-FIRST, we know that if e has type $\alpha \overset{R'}{\rightsquigarrow} \beta$, then $R = R'$. Furthermore, we know that P will proceed via judgment FT-FIRST. By our inductive hypothesis, we know that e will interact with each resource in R' at most once, and since no resource interaction happens in this judgment, we satisfy our conditions.

- *SF composition:* When P is of the form $e_1 \ggg e_2$, it will proceed by the FT-COMP judgment. By typing rule TY-COMP, we know that e_1 has resource type set R_1 and e_2 has resource type set R_2 such that $R_1 \cup R_2 = R$ but $R_1 \cap R_2 = \emptyset$. By our inductive hypothesis, e_1 evaluates interacting with at most the resources in R_1 and e_2 evaluates interacting with at most the resources in R_2. However, R_1 and R_2 share no common resources, and together, they make up R. Therefore, P does not interact with any more resources than those in R, and any in R that it interacts with, it does so at most once.

- *SF choice:* When P is of the form $e_1 \,|||\, e_2$, it will proceed by the FT-CHC$_1$ or FT-CHC$_2$ judgment. Typing rule TY-CHC tells us that e_1 has resource type set R_1 and e_2 has resource type set R_2 such that $R_1 \cup R_2 = R$. By our inductive hypothesis, we know that either e_1 evaluates interacting with at most the resources in R_1 or e_2 evaluates interacting with at most the resources in R_2, but only one transition is used. We know that R is the set of all common resources in R_1 and R_2, so regardless of which transition P proceeds through (running e_1 or e_2), only resources in R will see interaction, and they will only be interacted with at most once. Therefore, we satsify our conditions.

- *SF resource interaction:* If P is of the form *rsf* r, then it will proceed by the FT-RSF judgment. Typing rule TY-RSF tells us that its type must be $\alpha \overset{\{r\}}{\rightsquigarrow} \beta$. The transition completes in one step with no preconditions making use of no further calls, but in fact, $\mathscr{V}$ is being modified, so resource interaction is taking place. We see that the element in $\mathscr{V}$ for resource r is the only one being accessed and it happens precisely once. The access is allowed because trivially $r \in \{r\}$.

- *wormhole introduction:* P will proceed by the FT-WH judgment when it is of the form **letW** $r_w\ r_b\ e_i$ **in** e. Typing rule TY-WH tells us that e has type $\alpha \overset{R}{\rightsquigarrow} \beta$ the same as P. First, we recognize that no resource interaction can be performed by e_i because it is never evaluated as a expression by the functional transition. Even though we add values to $\mathscr{V}$, we do not modify and existing values, so we are not doing any true resource interaction in this transition. Therefore, our inductive hypothesis tells us that only acceptable resource interaction is done in the transition of the precondition. □

This proof takes the progress and preservation of our semantics for granted. The proofs for these can be located in Appendix A.3.

4.7 Haskell Implementation

In addition to the typing rules and operational semantics, we built an implementation of arrowized FRP with resource types and wormholes within the Haskell language. This implementation differs slightly from the language design we have specified, but this is due to a few particular limitations in Haskell. At the end of this section, we will discuss possible extensions to the Haskell language specification (or, more likely, language extensions for GHC) that could allow us to overcome these limitations.

4.7.1 The Resource Type

We will begin by building a system to allow for resource types. Essentially, a resource is a type-level entity that can be accessed to perform a read or effectful update (recall the `read` and `update` functions from the previous section). We choose to represent the idea of resources as a type class, and then new resources can be created by allowing types to instantiate this class. We make use of GHC's functional dependencies and

multiple parameter type classes to write this:

$$\begin{aligned}&\textbf{class } \mathit{Resource}\ r\ a\ b \mid r \to a, r \to b \textbf{ where}\\ &\quad \mathit{read} \quad :: r \to \mathit{IO}\ b\\ &\quad \mathit{update} :: r \to a \to \mathit{IO}\ ()\\ &\quad \mathit{rsf} \quad :: r \to (a \overset{\{r\}}{\rightsquigarrow} b)\end{aligned}$$

Where in our theoretical model, the read and update functions were pure, here we allow their Haskell counterparts *read* and *update* to perform effectful *IO* actions. Thus, we additionally require resources to obey the following Resource Law:

$$\mathit{read}\ r \gg \mathit{read}\ r = \mathit{read}\ r$$

To instantiate this class with a resource type, one would provide as the three type parameters the resource, its input type, and its output type. Then, for the given resource, one can define the *read* and *update* functions which will perform the resource's I/O effects.

Rather than force the user to instantiate the *rsf* function, there should be a default implementation provided by the library author, or the one who defines the arrow type. For instance, one could build this resource type system on top of a simple Kleisli Automaton over the *IO* monad:

$$\textbf{data } a \rightsquigarrow b = \mathit{KA}\ (a \to \mathit{IO}\ (b, a \rightsquigarrow b))$$

In this case, a simple implementation of *rsf* could be:

$$\begin{aligned}&\mathit{rsf}\ r = \mathit{KA}\ \$\ \lambda\ a \to \textbf{do}\\ &\qquad\qquad \mathit{update}\ r\ a\\ &\qquad\qquad b \leftarrow \mathit{read}\ r\\ &\qquad\qquad \mathit{return}\ (b, \mathit{rsf}\ r)\end{aligned}$$

With this infrastructure in place, a user can define resources very easily. For instance, if a user wanted to declare a resource for printing lines of text to the terminal console, he could do so:

$$\begin{aligned}&\textbf{data } \mathit{Console} = \mathit{Console}\\ &\textbf{instance } \mathit{Resource\ Console\ String}\ () \textbf{ where}\\ &\quad \mathit{read}\ \_ = \mathit{return}\ ()\\ &\quad \mathit{update}\ \_ = \mathit{putStr}\end{aligned}$$

Thus, resources are extensible both over the nature of the underlying arrow type as well as by the user who wishes to add new resources to his environment.

4.7.2 Resource Type Sets

Resource types alone are not enough; next, we need a way to represent sets of resource types. Our implementation is inspired by Haskell's HList library [Kiselyov et al., 2004] for heterogeneous lists, and as such, resource type sets will actually be implemented using lists of types (which have an inherent order and may have duplicates). Our goal in this subsection will be to create type classes and families to allow us to perform our main set operations: union, disjoint union, and set removal. In addition to some of GHC's more well-known language extensions (multiple parameter type classes, etc.), we make use of the newer type families and data kinds extensions as well.

To begin, we use an updated version of the type equality class from HList, here with equality constraints and Boolean data kinds:

$$\begin{aligned}&\textbf{class } \mathit{TypeEq}\ (x :: k)\ (y :: k)(b :: \mathit{Bool}) \mid x\ y \to b\\ &\textbf{instance } (\mathit{True} \sim b) \Rightarrow \mathit{TypeEq}\ x\ x\ b\\ &\textbf{instance } (\mathit{False} \sim b) \Rightarrow \mathit{TypeEq}\ x\ y\ b\end{aligned}$$

```
class Union (xs :: [*]) (ys :: [*]) (zs :: [*]) | xs ys → zs
instance Union '[] '[] '[]
instance Union '[] ys ys
instance Union xs '[] xs
instance (ElemOf x ys b, res ~ IfThenElse b ys (x ': ys), Union xs res zs)
      ⇒ Union (x ': xs) ys zs
```

Figure 4.7: The Union type class.

An instance of *TypeEq* has its third type as *True* only when the first two types are equal. Although it is possible to write a similar construction using Haskell's closed type families, the two versions are not the same. This version of *TypeEq* will unify more eagerly in the case of type inequality, which will prove essential for how we intend to use it.

We will also need a way to make a type level decision based on whether types are equal, so we introduce the following closed type family:

```
type family IfThenElse (b :: Bool) (x :: k) (y :: k) :: k where
  IfThenElse True  x y = x
  IfThenElse False x y = y
```

If the first type argument is True, the result is the second (the "then" clause), and if it is False, the result is the third (the "else" clause).

Together, type equality and the conditional type family allow us to write a type class that computes type level list inclusion:

```
class ElemOf (x :: *) (ys :: [*]) (b :: Bool) | x ys → b
instance ElemOf x '[] False
instance (TypeEq x y b, ElemOf x ys z, r ~ IfThenElse b True z)
      ⇒ ElemOf x (y ': ys) r
```

The first instance states that a type is never an element of an empty type list. The second states that a type is an element of a type level list either if it is equal to the head of that list or if it is an element of the tail.

With these classes and family established as the basics, we can begin in earnest with the set operations we need. We present a type class that performs the union of two sets in Figure 4.7. The first three instances dictate that the union of a set with the empty set (in either order) is the set itself. The last states that we can find the union of two sets by examining the head of the first set. If it is an element of the second set, then the result is the union of the tail of the first set and the second set. Otherwise, the result is the union of the tail of the first set and the head of the first set added to the second set.

Another way of viewing our set union operation is that is is appending the two underlying lists together but skipping any elements that the two lists have in common.

Next, we can use this union operation together with a disjointness test to create our disjoint union type class, which we show in Figure 4.8.

Lastly, we create a type class that represents set removal, which we show in Figure 4.9. Unlike the other type classes, this class will have an associated function that performs the removal. The first instance states that removing an element from an empty set is just the empty set. The second states that removing an element from a set whose head element is that element is the tail of the set. The third states that removing an element from a set whose head element is not that element is the same as removing the element from the

```
class Disjoint (xs :: [*]) (ys :: [*])
instance Disjoint '[] '[]
instance Disjoint '[] ys
instance Disjoint xs '[]
instance (ElemOf x ys False, Disjoint xs ys)
      => Disjoint (x ': xs) ys

class UPlus (xs :: [*]) (ys :: [*]) (zs :: [*]) | xs ys -> zs
instance (Disjoint xs ys, Union xs ys zs) => UPlus xs ys zs
```

Figure 4.8: The Disjoint type class for establishing disjointness and the UPlus type class for performing the disjoint union.

```
class SRemove (x :: [*]) (ys :: [*]) (zs :: [*]) | x ys -> zs where
  sremove :: x -> (b ~>[ys] c) -> (b ~>[zs] c)
instance SRemove x '[] '[] where
  sremove _ = unsafeCoerce
instance (ys ~ zs) => SRemove x (x ': ys) zs where
  sremove _ = unsafeCoerce
instance (SRemove x ys ys', (y ': ys') ~ zs) => SRemove x (y ': ys) zs where
  sremove _ = unsafeCoerce
```

Figure 4.9: The SRemove type class removes resource types from resource type sets. It has a value-level function as well.

$$\textbf{class } \mathit{Category}\ (\rightsquigarrow) \textbf{ where}$$
$$\quad \mathit{id} :: b \overset{'[]}{\rightsquigarrow} b$$
$$\quad (\ggg) :: (\mathit{UPlus}\ r_1\ r_2\ r_3) \Rightarrow (b \overset{r_1}{\rightsquigarrow} c) \to (c \overset{r_2}{\rightsquigarrow} d) \to (b \overset{r_3}{\rightsquigarrow} d)$$

$$\textbf{class } \mathit{Category}\ (\rightsquigarrow) \Rightarrow \mathit{Arrow}\ (\rightsquigarrow) \textbf{ where}$$
$$\quad \mathit{arr}\ \ :: (b \to c) \to (b \overset{'[]}{\rightsquigarrow} c)$$
$$\quad \mathit{first} :: (b \overset{r}{\rightsquigarrow} c) \to ((b,d) \overset{r}{\rightsquigarrow} (c,d))$$

$$\textbf{class } \mathit{Arrow}\ (\rightsquigarrow) \Rightarrow \mathit{ArrowChoice}\ (\rightsquigarrow) \textbf{ where}$$
$$\quad (|||) :: (\mathit{Union}\ r_1\ r_2\ r_3) \Rightarrow (b \overset{r_1}{\rightsquigarrow} d) \to (c \overset{r_2}{\rightsquigarrow} d) \to (\mathit{Either}\ b\ c \overset{r_3}{\rightsquigarrow} d)$$

$$\textbf{class } \mathit{Arrow}\ (\rightsquigarrow) \Rightarrow \mathit{ArrowLoop}\ (\rightsquigarrow) \textbf{ where}$$
$$\quad \mathit{loop} :: ((b,d) \overset{r}{\rightsquigarrow} (c,d)) \to (b \overset{r}{\rightsquigarrow} c)$$

Figure 4.10: The Haskell Arrow classes redefined to permit resource types. Note that we are only including the primitive operations and leaving out any that can be defined in terms of them (e.g. *second*).

tail while including the head in the result. One may note that we use a trick similar to what we did in the *TypeEq* class to test for type equality.

We need the value-level function *sremove*, but it has no value-level computation, so we fix it for all instances as $\mathit{unsafeCoerce} :: a \to b$. Although seemingly overpowered, we are using this coercion in a safe way: not only is the function itself constrained by the class, but because it can only be used when the type set and the element to remove satisfy the class, we can be sure that we will not change the type into something inappropriate.

4.7.3 Re-Typing the Arrow Operators

We now have both type level resources as well as a method to represent and operate over sets of types. What remains is to use these types in the typing of the arrow operators, as we did in Figure 4.1.

In Haskell, arrows are represented by the *Arrow* type class, which itself is a subclass of the *Category* type class. In order to retain our ability to use arrow syntax in our code, we follow the same pattern.[3] Thus, instead of recreating our arrows from scratch, we simply modify the existing classes to suit our needs.

We show the code for these type classes as well as relevant portions of the *ArrowChoice* type class (which provides the choice operations) and the *ArrowLoop* type class (for loops) in Figure 4.10.[4] Note that we require the $(|||)$ function for choice rather than the more traditional and simpler *left*. Because choice allows for a union of types, we can on longer define $(|||)$ as a composition of *left* and *right*; however, we can define *left* and *right* in terms of $(|||)$.

[3] As we will discuss in Section 4.7.6, GHC does not currently support rebindable syntax for Arrows, which means no method of re-typing the Arrow classes will make using the arrow syntax with resource types possible. However, we show this process to future-proof the concept, or to prepare for when this feature is supported.

[4] Note that the code we present is not actually valid Haskell code because we are using the $\rightsquigarrow$ symbol for our arrow. In Haskell, a symbol like this can only be used as a binary type operator, but we use it as an operator over three types (input to the left, output to the right, and resources above). In Haskell, we are technically forced to use a prefix type operator instead, but for the sake of clarity and consistency with our examples, we take the liberty of using the symbol operator.

4.7.4 Wormholes

In our theoretical model, wormhole resources were subtly different from physical resources. Specifically, wormhole resources were handled with a separate virtual resource environment $\mathscr{W}$. In practice, it is simpler to instead make wormhole resources just the same as physical ones. Thus, implementing wormholes comes down to instantiating the *Resource* type class.

Ideally, we would be able to use local class instances such that we could generate new types for our resources and then locally declare them as instances of the *Resource* type class for the wormhole body. Although local type class instances have been discussed, no version has yet made it into GHC, so we are forced to take another approach.

We will make two new types to represent whiteholes and blackholes and provide each with a hole for a phantom type so that we can keep their types distinct in the presence of multiple wormholes:

```
newtype Whitehole r t = Whitehole (IORef t)
newtype Blackhole r t = Blackhole (IORef t)
```

Within every whitehole and blackhole is a reference (an *IORef*) to the piece of memory that they both non-locally share. Writing the *Resource* instances is straightforward:

```
instance Resource (Whitehole r t) () t where
  read (Whitehole ref) = readIORef ref
  update _ _ = return ()

instance Resource (Blackhole r t) t () where
  read _ = return ()
  update (Blackhole ref) t = writeIORef ref t
```

Now that we have wormhole resources, we can consider writing the *letW* operator to introduce them. We will make one change from the version we discussed earlier in the chapter. Because Haskell has no way to simply extend the type environment for a body, we turn the body argument into a function that takes two resource values (one for the whitehole and one for the blackhole). Thus, the type of this function will look like:

```
letW :: ∀ r_t r r' r'' t b c .
  (SRemove (Whitehole r_t t) r r', SRemove (Blackhole r_t t) r' r'') ⇒
  t → (Whitehole r_t t → Blackhole r_t t → (b ⇝[r] c)) → b ⇝[r''] c
```

To remove the wormhole resource types from the output resource type set, we use two instances of *SRemove*.

The implementation of *letW* would be straightforward to write were it not for the fact that creating a whitehole and blackhole requires an *IORef*. Therefore, we must introduce one more function that our arrow must be able to support to allow wormholes:

```
class Arrow (⇝) ⇒ ArrowIO (⇝) where
  initialAIO :: IO d → (d → (b ⇝[r] c)) → (b ⇝[r] c)
```

This class and its function allow an arrow to be built from the results of an initializing I/O action. A function like this is required to build wormholes, but it also clearly breaks the guarantees of resource safety if used incorrectly. Thus, we allow it for the implementation of *letW* but expect that it is not exported to library users.

We now give the implementation of *letW*:

```
letW t inner = sremove (undefined :: Blackhole r_t t) $
  sremove (undefined :: Whitehole r_t t) $
    initialAIO (newIORef t)
      (λ ref → inner (Whitehole ref) (Blackhole ref))
```

Notice that this is where we make use of the value-level *sremove* function from the *SRemove* class. Without it, we would not be able to make the output resource set match the one returned by the *inner* function.

4.7.5 An Arrow Instance

We have done our best up until this point to avoid choosing any particular instance of the arrow class—resource types should be relatively universal and applicable to many forms of arrows—but at this point in our discussion of implementation, we will provide a sample signal function implementation.

As hinted at earlier, our implementation will be similar to the Kleisli Automaton, but we will augment it slightly to properly deal with resources. Specifically, we will define our arrow data type as follows:

$$\textbf{data}\ SF\ r\ b\ c = SF\ (b \to IO\ (c, IO\ (), SF\ r\ b\ c))$$

The *SF* data type has two separate ways of dealing with *IO* actions rather than the single way that the Kleisli Automaton has: the actions can be performed *during* the arrow's execution, or they can be gathered up in the output $IO\ ()$ to be performed later (*between* time steps).

Instantiating the arrow classes is trivial with this data type, and our instances look identical to those for the Kleisli Automaton but with *return* $()$ filled in for the extra $IO\ ()$ outputs and a simple bind ($\gg$) whenever two of those actions need to be combined. Indeed, that extra output becomes relevant only for resources, such as when we are defining our default implementation of *rsf*:

$$\begin{aligned} rsf\ r = SF\ \$\ \lambda\ b \to \textbf{do}& \\ & c \leftarrow read\ r \\ & return\ (c, update\ r\ b, rsf\ r) \end{aligned}$$

As can be seen here, the extra $IO\ ()$ output allows us to delay executing the `update` actions until between time steps, which is critical for making wormholes behave properly.

Recall from Definition 1 that for a signal function to be a program, it must have type $() \overset{R}{\rightsquigarrow} ()$, and we can write a function that will allow us to run a program:

$$\begin{aligned} &runSF :: () \overset{R}{\rightsquigarrow} () \to IO\ () \\ &runSF\ (SF\ sf) = \textbf{do} \\ &\quad ((), action, sf') \leftarrow sf\ () \\ &\quad action \\ &\quad runSF\ sf' \end{aligned}$$

4.7.6 Limitations

There are two limitations with the implementation we have presented in this section.

Abstract Types With Wormholes

The implementation we have provided works with any concrete resource types, but it cannot always handle arbitrary resource type sets. This means that any full program can be defined, but functions that transform arbitrary other signal functions may be rejected by the type checker.

An example will help illustrate this point. Let's assume we have a signal function that does some arbitrary resource interaction with resources *R* and that has an output type that is the same as its input type:

$$\begin{aligned} &mySF :: a \overset{R}{\rightsquigarrow} a \\ &mySF = \ldots \end{aligned}$$

Perhaps we want to wrap this with a wormhole to create a feedback loop. We can define the following:

$$mySF_{loop} :: a \to ((), ()) \overset{R}{\rightsquigarrow}$$
$$mySF_{loop}\ a = letW\ a\ (\lambda\ w\ b \to rsf\ w \ggg mySF \ggg rsf\ b)$$

As long as *mySF* is defined and concrete, this will work fine.

However, as functional programmers, we may naturally want to abstract the behavior of "wrapping" a function with a wormhole, and so we desire to write:

$$wrap :: (a \overset{r}{\rightsquigarrow} a) \to a \to ((), ()) \overset{r}{\rightsquigarrow}$$
$$wrap\ sf\ a = letW\ a\ (\lambda\ w\ b \to rsf\ w \ggg sf \ggg rsf\ b)$$

With this, we could simplify $mySF_{loop} = wrap\ mySF$, which seems great. Unfortunately, this is impossible. The type checker does not know what *r* will be, and so it cannot verify critical steps such as whether the whitehole or blackhole resources we create might already be part of *r*. This is frustrating because we as programmers know that the resources of a wormhole we construct will be fresh and absent from any arbitrary *r*, but Haskell's type checker is currently not up to the task of deducing this.

This limitation only appears when combining signal function transformers (such as the above *wrap* function) with uses of wormholes. Thus, we see it as an acceptable limitation for our prototype of resource types. We believe the issue could be overcome in more dependently typed languages, so for a fully working implementation, we must either extend or abandon Haskell.

Rebindable Syntax

Currently, arrow syntax in GHC is activated by the "arrows" language pragma and is fairly restrictive: GHC expects all arrows to follow a particular form, and that form cannot change through a segment of arrow syntax. With resource-typed arrows, the arrow's type may change within arrow syntax—indeed it will if any *rsf* operations are within the syntax. GHC sees this as an error.[5]

Unlike the type removal from the previous limitation, this one is entirely surmountable. In fact, there is even a template from which to begin working: monadic rebindable syntax. The rebindable syntax extension to GHC allows one to declare his own Monad class, and then GHC will follow it exactly, even if it calls for the type of the monad to change within the syntax. Although no effort has been made to write this extension, it should not only be possible, but we expect it to be straightforward, if time consuming.

Additionally, this limitation even has a currently available workaround. Although GHC does not support rebindable syntax for arrows, Paterson's original `arrowp` arrow preprocessor[6] does. Therefore, a user who desires resource types along with arrow syntax can instead remove the "Arrows" language pragma from their source file and run it through `arrowp` before handing it off to GHC. The error messages tend to be more challenging to comprehend, and it is more of a hassle to use, but it is a technically working alternative.

[5] One may note that the code shown in this entire Haskell Implementation section never explicitly uses Haskell's arrow syntax, and this rebindable syntax restriction is the reason why.

[6] Available on Hackage at `https://hackage.haskell.org/package/arrowp`.

Chapter 5

Asynchronous Functional Reactive Processes

5.1 Considering Asynchrony

Arrowized functional reactive programming is naturally synchronous: components are connected via composition, and those connections describe a synchronous flow of information. In other words, if a signal function has an output type of α then it can be composed with another that has an input type of α, and every output from the first is synchronously fed to the second. In fact, even if no *data* is being conveyed along one of these connections, as would be the case if α were $()$, there is still a sense of *time* that is communicated. Thus, to create asynchronous signal functions, we need to consider an entirely new connection method: we must consider how we can construct a connection that somehow dissociates time from data.

5.1.1 Wormholes Revisited

Of course, we have already discussed the answer previously; rather than use default composition for our asynchronous connections, we use specially designed wormholes that apply a *time dilation* over their contents, warping those contents to match the timing of their output. Where synchronously there would be a unit delay between the blackhole's inputs and the whitehole's outputs, asynchronously, there must be a different sort of effect on time. Instead of the whitehole emitting exactly what the blackhole accepted, it will emit a stretched out or compressed image of it depending on the nature of time in its output. For example, if a sine wave were passed through a wormhole from one process to another running at half the speed (in a discrete realm, this would correspond to it experiencing half as many time steps per second), then the receiving process would perceive the frequency of that sine wave as twice as fast.

Of course, the fundamental nature of the wormhole will not change—even with this new model, if the two ends of the wormhole share the same notion of time (i.e. are in the same process), then this will simplify to the same unit delay that we started with.

Operationally, we can achieve this in a discrete model by allowing the underlying data structure of the wormhole to sometimes return nothing (indicating a stretching of time, or that no new data has been generated from the blackhole since the last whitehole access) and other times return multiple elements (a compression of time, in which multiple elements have "queued up"). Thus, we build our wormholes atop a queue structure, which we model using a list.

Interestingly, we generally do not need to worry about how large this list can get or other common issues of buffering. Because wormholes are designed to be read only by one source (the whitehole), we do not need to keep any buffer history between whitehole accesses. Thus, the amount that the buffer can

grow is governed by the number of times its blackhole is written to before its whitehole is read, which is typically predictable and fairly well bounded. Of course, one can design a pathological case where the process writing to the blackhole stalls indefinitely, but as long as the system behaves fairly, this should not be a realistic problem.

The *letW* operator changes slightly to reflect this. The blackhole resource remains the same, but the whitehole resource will now be of type $\langle (), List\ t \rangle$. Additionally, as the underlying data type is now a list, the initial value given to the wormhole will also be a list.

Because of this underlying queue, wormholes somewhat resemble channels from other languages. Although they are conceptually similar in that they both ferry data from one place to another, their use is somewhat different. Unlike the output of a channel, from which individual data can be popped off and used, the output from a wormhole consists of the time dilated data from another process.

5.1.2 Forking

With this model, communication between asynchronous components can, and in fact must, be done entirely at the resource level (i.e. with wormholes). Therefore, any asynchronous process will have the type $() \overset{R}{\rightsquigarrow} ()$. But still, we must be careful—we cannot simply compose two asynchronous signal functions together even if their input and output types are $()$ because any connection using standard composition would still enforce a synchronization point.

Therefore, we introduce a new operator:

$$fork :: () \overset{R}{\rightsquigarrow} () \to \alpha \overset{R}{\rightsquigarrow} \alpha$$

The *fork* operator will spawn a new process for the given signal function and allow it to run freely with its own sense of time. In its own process, it will behave as an identity.

As mentioned above, even when inputs and outputs seem to convey no information, they are still communicating a sense of time; this is the case with *fork* as well. Although the embedded signal function is specifically *not* synchronized with the input $()$ stream, that stream still provides a notion of time to the asynchronous process.

This may seem confusing or irrelevant, but it has a serious impact when considered in the presence of arrow choice. If a *fork* operation is in a branch of a choice that is not currently active, then it is not currently receiving any information, about time or otherwise. Therefore, it should not and must not provide any sense of time to the asynchronous process it has created. That is, if a *fork* is in an un-taken choice branch, then the forked process must not be active (or must be immediately terminated). In many ways, this is similar to the ideas of non-interfering choice discussed in Chapter 3, which dictate that if a branch is inactive, the program as a whole should behave as if that branch does not exist.

At first glance, this seems dangerous—What if we stop a process mid-execution, leaving it in some sort of unsafe state?—however, we can easily sidestep this issue by relying on the fundamental abstraction of FRP. Because we assume that instantaneous values are processed infinitely fast, then at any given point in time, the instantaneous value is either processed entirely or not at all. Thus, we can never be "in the middle of" anything.

Operationally, we handle this by treating each time step for a given process as a *transaction* that will either succeed completely or fail completely. Thus, if the process stops mid-execution, the transaction fails and no effect is observable.

In total, we have a system where multiple signal functions can run independently of each others' notions of time and yet still communicate when needed. That is, we have a system of *communicating functional reactive processes*, which we refer to as CFRP.

5.2 Motivating Examples

Before proceeding with the formal syntax and semantics of CFRP, we provide a few examples to help motivate the design we have chosen. In these examples, we demonstrate some high level, expressive operators that can be built using the CFRP tools. In Section 5.4, we will define these operators in more detail.

5.2.1 Fork

Forking a signal function to run with its own notion of time is the most primitive asynchronous computation that CFRP offers. That said, it is powerful and useful even on its own.

In the world of computer music, it is not uncommon to want to present a GUI to a user while simultaneously producing music, and indeed, FRP can be used for both. Unfortunately, the GUI will probably be running at around 30-60 frames per second, but in order to get high quality sound, the pitch produced must be rendered at more like 44,100 samples per second.

This thousandfold disparity would pose a problem for a synchronous FRP model, but it is exactly the problem that the fork primitive is designed to overcome. Furthermore, we can make use of fork's interaction with choice to provide the user with an option to dictate whether the music should be playing or not. If the user opts to silence the music, the forked process will stop executing.

We will assume two domain specific signal functions: a widget that produces a selection option ($select :: String \to (() \rightsquigarrow Bool)$) and a sound player ($playSound :: () \overset{\{Speakers\}}{\rightsquigarrow} ()$). With these, we define:

$$
\begin{aligned}
&musicGUI :: () \overset{\{Speakers\}}{\rightsquigarrow} () \\
&musicGUI = \textbf{proc}\ () \to \textbf{do} \\
&\quad b \leftarrow select\ \text{“Play music?”} \prec () \\
&\quad \textbf{if}\ b\ \textbf{then}\ fork\ playSound \prec ()\ \textbf{else}\ returnA \prec ()
\end{aligned}
$$

While the selection is *True*, the *playSound* function will proceed at its own time rate in its own process, but when the user sets it to *False*, it will stop and the process will stop as well.

5.2.2 Asynchrony in network packet maps

Although the forking from the above example is useful, CFRP can also handle the more interesting case where the asynchronous processes need to communicate as well. For example, in the realm of networking, one may have two signal functions, one for determining an incoming packet's destination and another for examining the packets to determine an optimal network map. Every network map is guaranteed to be correct, so it is acceptable to use an old map when routing, but it is essential that the system routes packets as fast as possible. Even though creating the network map may take a long time, a fully synchronous system will force packets to wait whenever the map is being recalculated, but with CFRP, we can construct new maps in an asynchronous process that experiences time more slowly, allowing us to retain fast routing performance.

We might describe this scenario with the data types *Packet*, *Dest*, and *Map* and the signal functions $route :: (Packet, Map) \rightsquigarrow Dest$ and $makeMap :: Packet \rightsquigarrow Map$. Using arrow syntax, we can write the synchronous version:

$$
\begin{aligned}
&router :: Packet \rightsquigarrow Dest \\
&router = \textbf{proc}\ p \to \textbf{do} \\
&\quad m \leftarrow makeMap \prec p \\
&\quad d \leftarrow route \prec (p, m) \\
&\quad returnA \prec d
\end{aligned}
$$

This router will execute *makeMap* for every *Packet*, slowing down packet routing severely. Even with a modified $makeMap' :: (List\ Packet) \rightsquigarrow Map$ that accepts batches of packets, we will get an intermittent

slowdown; for example, if *makeMap*′ ran on batches of ten packets, then every tenth packet would be delayed while the map was being created.

Instead, we can create an *async* operator that will automatically fork *makeMap*′ to its own asynchronous process and provide wormholes for communication between the processes. With *makeMap*′ running with its own, slower notion of time on the other end of a wormhole from the main process, the stream of *Packet*s it receives will be compressed and the *Map*s that it produces expanded. Operationally, this means that new *Packet*s will queue up in the input wormhole, and as soon as *makeMap*′ finishes one *Map*, it will collect the queued *Packet*s and begin working on the next one. Meanwhile, *route* will run quickly on each *Packet* and automatically see new *Map*s whenever they are created. The *async* operator has the type $async :: (List\ \alpha \rightsquigarrow \beta) \to (\alpha \rightsquigarrow List\ \beta)$ and we will define it from CFRP primitives in Section 5.4.1 once we present the language more fully. That said, we can use it now to code an asynchronous version of the router:

$$
\begin{aligned}
&router :: Packet \rightsquigarrow Dest \\
&router = \textbf{letW}\ r_w\ r_b\ (m_{default} : \varepsilon)\ \textbf{in proc}\ p \to \textbf{do} \\
&\quad m_{new} \leftarrow async\ makeMap' \prec p \\
&\quad m_{prev} \leftarrow rsf\ r_w \prec () \\
&\quad \textbf{let}\ m = \textbf{if}\ null\ m_{new}\ \textbf{then}\ head\ m_{prev}\ \textbf{else}\ head\ m_{new} \\
&\quad () \leftarrow rsf\ r_b \prec m \\
&\quad d \leftarrow route \prec (p, m) \\
&\quad returnA \prec d
\end{aligned}
$$

Note that we use a supplementary wormhole initially supplied with a default map simply to keep track of state during a time step. That is, the wormhole feeds the old map back to the beginning of the process in case no new map is ready by the next time step.

5.2.3 Speculative Parallelism

One of the benefits of non-deterministic asynchrony is the ability to perform multiple operations at once and observe which is fastest. In particular, we can start two tasks, and when one of them finishes, we can accept the value it returns and ignore or even cancel the other task. This is called *speculative parallelism*.

Let us assume we have two signal functions that represent our two tasks. In synchronous FRP, these two signal functions will both compute their results in one time step, and even if one finishes first, the program will wait for the other. One option to try to address this is to allow the signal functions to take multiple time steps to complete. That is, we can let the input be an event stream that is assumed to only ever provide a one-time "impulse" event and allow the output to likewise be an event stream that will only return a value when it has taken enough steps to have produced that value.

Even with events and the ability to delay returning a value, synchronous FRP models will not work properly because the two signal functions will still proceed in lock-step. Even though one may finish before the other, their synchronization on each step will prevent one with many fast steps from ever beating one with a few computationally intensive steps.

The asynchronous nature of forking can overcome this hurdle. Because we can allow each signal function to run with its own time, we can actually observe which is faster, even if it is the one that takes more time steps. Thus, we can provide a function for speculative parallelism:

$$
\begin{aligned}
spar :: &(Event\ \alpha \rightsquigarrow Event\ \beta) \\
&\to (Event\ \alpha \rightsquigarrow Event\ \gamma) \\
&\to (Event\ \alpha \rightsquigarrow Event\ (\beta + \gamma))
\end{aligned}
$$

When $spar\ e_1\ e_2$ is given an impulse, it will fork both e_1 and e_2. Eventually, some time later, it will produce an event that is either a $Left\ \beta$ if e_1 finished first or a $Right\ \gamma$ if e_2 did. At that point, it will stop both signal

functions and produce no more events. The *spar* function can be defined from CFRP primitives, and we will show this definition in Section 5.4.3.

As a practical case, we can once again consider the network routing maps from our previous example. Let us assume that there are two different implementations of *makeMap*′: one that spends a brief time step on each packet in its batch, incorporating the packets into the map one by one, and the other that creates a map out of the batch of packets and then merges that in its entirety to the older map all in one long time step. Which approach is faster may be indeterminable at the outset, so we would like to run both and use the result from whichever finishes first. With *spar*, this is trivial.

5.2.4 Parallel Composition

Although asynchrony typically implies nondeterminism (as in the previous examples), we can also use it to define a certain class of deterministic, concurrent scenarios such as a parallel, no-feedback function pipeline. As long as there is no feedback, the parallel composition of two signal functions is possible because the first one can begin work on the "next" value while the second one is still working on the "current" value. In fact, whole chains of signal functions can be parallelized in this way. Of course, this mentality assumes discrete events, and so we must restrict this procedure to only apply to event streams.

For this example, let us assume we have a signal function producing data to process, two signal functions for computation, and a signal function for delivering output:

$$\begin{aligned} \mathit{source} &:: () \overset{\{\mathit{src}\}}{\rightsquigarrow} \mathit{Event}\ \alpha \\ \mathit{sf}_1 &:: \mathit{Event}\ \alpha \rightsquigarrow \mathit{Event}\ \beta \\ \mathit{sf}_2 &:: \mathit{Event}\ \beta \rightsquigarrow \mathit{Event}\ \gamma \\ \mathit{sink} &:: \mathit{Event}\ \gamma \overset{\{\mathit{snk}\}}{\rightsquigarrow} () \end{aligned}$$

Note the use of resources *src* and *snk*.

Rather than simply compose these all in series, we can asynchronize each one, but instead of relying on a master thread to manage the others, we use the parallel composition operator:

$$(>|>) :: (\alpha \rightsquigarrow \mathit{Event}\ \beta) \to (\mathit{Event}\ \beta \rightsquigarrow ()) \to (\alpha \rightsquigarrow ())$$

which we will define in Section 5.4.2. Note that the type of $>|>$ is similar to that of simple composition. With it, we can connect the output of one signal function directly to the next one and still reap the benefits of parallelism:

$$\begin{aligned} &\mathit{pipeline} :: () \overset{\{\mathit{src},\ \mathit{snk}\}}{\rightsquigarrow} () \\ &\mathit{pipeline} = \mathit{source}\ >|>\ \mathit{sf}_1\ >|>\ \mathit{sf}_2\ >|>\ \mathit{sink} \end{aligned}$$

For a practical case, consider an FRP implementation of a program interpreter. Program interpretation proceeds through a number of steps: perhaps parsing, optimizing, and evaluating. While optimizing one piece of code, we could theoretically start parsing the next, but standard synchronous composition of these steps would force us to wait until each code event is optimized and even evaluated before we can begin parsing the next one. However, because parsing depends on neither optimization nor evaluation and optimization does not depend upon evaluation, we can connect these with $>|>$ instead of $\ggg$ and see a performance improvement.

Typ	τ	$::=$	$\ldots$
		$\mid \tau_1 \overset{\{r_1,\ldots\}}{\rightsquigarrow} \tau_2$	resource typed SF
Var	v		
Exp	e	$::=$	$\ldots$
		$\mid$ *arr* e	SF construction
		$\mid$ *first* e	SF partial application
		$\mid e_1 \ggg e_2$	SF composition
		$\mid e_1 \mid\mid\mid e_2$	SF choice
		$\mid$ *fork* e	SF fork
		$\mid$ *rsf* r	SF resource interaction
		$\mid$ **letW** $r_w\ r_b\ e_i$ **in** e	wormhole introduction
Env	Γ	$::= v_1 :: \tau_1, \ldots, v_n :: \tau_n$	type environment
Res	r		
RTp	t	$::= \langle \tau_{in}, \tau_{out} \rangle$	resource type
REn	Ψ	$::= r_1 :: t_1, \ldots, r_n :: t_n$	resource environment

Figure 5.1: The CFRP extension to $\mathscr{L}\{\rightarrow\times+\}$.

5.3 The Language

5.3.1 Syntax

Once again, we will start with $\mathscr{L}\{\rightarrow\times+\}$, the basic lambda calculus extended with product and sum types and general recursion that we introduced in Chapter 2.2 and extend it further with arrow operations, resources, and wormholes[1]. This extension is shown in Figure 5.1.

We let τ range over types, v over variable names, e over expressions, and Γ over environments. A type judgment $\Gamma \vdash e :: \tau$ indicates that it follows from the mappings in the environment Γ that expression e has type τ. Sums, products, and functions satisfy β- and η-laws. Further, we let r range over resources, t over resource types, and Ψ over resource environments.

Lastly, we define processes that CFRP supports, and note that CFRP can run any process as a top level program:

Definition 4 (Process)**.** *An expression e is a **process** if it has type $() \overset{R}{\rightsquigarrow} ()$ for some set of resources R.*

5.3.2 Typing Rules

The part of the language not associated with resources and signal functions (that is, $\mathscr{L}\{\rightarrow\times+\}$) is necessary but tangential to our discussion, and as such, we omit the typing rules and semantics. It suffices to say that they are as expected for a non-strict, functional language.

The seven signal function expressions allow the construction of complex signal functions in CFRP. The rules are presented in Figure 5.2, and we will examine the new or altered ones in more detail here:

- The TY-FORK rule states that a forked signal function must have type $() \overset{R}{\rightsquigarrow} ()$. The whole expression acts as the identity signal function to its streaming input.

[1] We will additionally use the notation ε for the empty list, : for construction, and $+\!\!+$ for appending two lists.

$$\textsc{Ty-Arr}\ \frac{\Gamma \vdash e :: \alpha \to \beta}{\Gamma;\Psi \vdash \mathit{arr}\ e :: \alpha \overset{\emptyset}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-First}\ \frac{\Gamma;\Psi \vdash e :: \alpha \overset{R}{\rightsquigarrow} \beta}{\Gamma;\Psi \vdash \mathit{first}\ e :: (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}$$

$$\textsc{Ty-Comp}\ \frac{\Gamma;\Psi \vdash e_1 :: \alpha \overset{R_1}{\rightsquigarrow} \beta \quad \Gamma;\Psi \vdash e_2 :: \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \uplus R_2 = R}{\Gamma;\Psi \vdash e_1 \ggg e_2 :: \alpha \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Chc}\ \frac{\Gamma;\Psi \vdash e_1 :: \alpha \overset{R_1}{\rightsquigarrow} \gamma \quad \Gamma;\Psi \vdash e_2 :: \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R}{\Gamma;\Psi \vdash e_1 \,|||\, e_2 :: (\alpha + \beta) \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-Fork}\ \frac{\Gamma;\Psi \vdash e :: () \overset{R}{\rightsquigarrow} ()}{\Gamma;\Psi \vdash \mathit{fork}\ e :: \alpha \overset{R}{\rightsquigarrow} \alpha}$$

$$\textsc{Ty-RSF}\ \frac{\big(r :: \langle \tau_{in}, \tau_{out} \rangle\big) \in \Psi}{\Gamma;\Psi \vdash \mathit{rsf}\ r :: \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}$$

$$\textsc{Ty-LetW}\ \frac{\Gamma;\Psi, r_w :: \langle (), \mathit{List}\ \tau \rangle, r_b :: \langle \tau, () \rangle \vdash e :: \alpha \overset{R'}{\rightsquigarrow} \beta \quad \Gamma;\Psi \vdash e_i :: \mathit{List}\ \tau \quad R = R' \setminus \{r_w, r_b\}}{\Gamma;\Psi \vdash \mathbf{letW}\ r_w\ r_b\ e_i\ \mathbf{in}\ e :: \alpha \overset{R}{\rightsquigarrow} \beta}$$

Figure 5.2: The typing rules for the CFRP signal functions.

- The TY-LETW rule is for wormhole introduction. It says that the body of the wormhole is a signal function provided that two resources are added to $\mathscr{R}$: one of the form $\langle (), List\ \tau \rangle$ (the whitehole) and one of the form $\langle \tau, () \rangle$ (the blackhole) where $List\ \tau$ is the type of e_i. The result of the whole expression is the same as that of the body except that the two new resources are removed from the resource set. This omission is valid because the virtual resources cannot escape the wormhole expression.

5.3.3 Operational Semantics

The operational semantics for CFRP are broken up into two main transitions: the *evaluation transition* for evaluating basic lambda calculus and the *functional transition* for interpreting the flow of data through the component signal functions of the program through time. Beyond these two, we have the *executive transition*, a one judgment transition that defines the course of program execution. Thus, we present the following three transitions:

$$\begin{array}{rll} e \mapsto e' & & \text{Evaluation transition} \\ (S, T, \mathscr{R}, \mathscr{W}) & \Rrightarrow & (S', T', \mathscr{R}', \mathscr{W}') \quad \text{Functional transition} \\ (T, \mathscr{R}, \mathscr{W}) & \Downarrow & (T', \mathscr{R}', \mathscr{W}') \quad \text{Executive transition} \end{array}$$

where e are expressions, S are program states, T are process maps, $\mathscr{R}$ are resource maps, and $\mathscr{W}$ are wormhole maps.

These semantics follow a similar three-tier system to the semantics for synchronous FRP with resource types from Chapter 4.5. However, where the synchronous semantics use a big-step transition for the middle tier, we use a small step, which allows us to merge the synchronous "temporal" transition right into the functional transition. Furthermore, the small step semantics are critical for being able to express the interleaving of processes that is necessary in describing the asynchrony inherent in CFRP.

Evaluation Transition

The evaluation transition is used to evaluate the non-streaming components of CFRP. n an effort to conserve space, we take as given a classic, non-strict, functional semantics for lambda expressions and application, product-type pairs and projection, sum-type case analysis and injection, and list-type construction and case analysis. Furthermore, we simply let all seven signal function expressions of CFRP go "unevaluated" so that they can be handled fully in the functional transition.

CFRP has a standard Canonical Forms Lemma associated with it that explains that for each type, there are only certain expressions that evaluate to a value of that type. The relevant portion of this lemma for our purposes is as follows:

Lemma 2 (Canonical Forms)**.** *If e is a value and* $e :: \alpha \overset{R}{\rightsquigarrow} \beta$*, then e is either an SF constructor, an SF partial application, an SF composition, an SF choice statement, a fork, a resource interaction, or a wormhole introduction.*

Functional Transition

The functional transition is a small-step, stack-based transition that details how signal function expressions behave. Not only does it describe how a signal function handles an instantaneous value of input, but it also governs the conceptual passage of time and applies a code transformation, updating the program itself when certain code segments are executed.

With four inputs and outputs, the functional transition looks rather complex, but in actuality, no single judgment uses all of the inputs. Therefore, we will occasionally omit matching inputs and outputs for a given judgment with the implication that the judgment holds them constant.

$$\textsc{Fr-First}\ \frac{}{(\cdot, z)\ \mathsf{frame}}$$

$$\textsc{Fr-Comp}_1\ \frac{}{(\cdot \ggg e_2)\ \mathsf{frame}} \qquad \textsc{Fr-Comp}_2\ \frac{}{(e_1 \ggg \cdot)\ \mathsf{frame}}$$

$$\textsc{Fr-Chc}_1\ \frac{}{(\cdot \,|||\, e_2)\ \mathsf{frame}} \qquad \textsc{Fr-Chc}_2\ \frac{}{(e_1 \,|||\, \cdot)\ \mathsf{frame}}$$

Figure 5.3: The stack frames for the functional transition.

The first argument to the transition is the program state, which is used by every judgment in the transition. The program state consists of a control stack K, an expression e, a streaming value v, and update data U and can come in one of two forms:

1. An *evaluation state* of the form $K \rhd (e, v, U)$ corresponds to the evaluation of signal function e with instantaneous streaming value v and update data U relative to a control stack K.

2. A *return state* of the form $K \lhd (e, v, U)$ corresponds to the evaluation of a stack K with possibly transformed signal function e, resulting instantaneous streaming value v, and update data U.

The control stack K represents the context of evaluation and is represented as a list of *frames*. The possible frames are shown in Figure 5.3, and we use the operator ; to add frames to the stack.

The set U contains pairs of resources along with input data for those resources. This is necessary because writing to resources happens conceptually *between* time steps, and so U acts as a buffer that accumulates resource writes until they are ready to be written.

The judgments for the most basic arrow expressions (SF construction, partial application, and composition) utilize only the program state. We show them in Figure 5.4 without the clutter of the other of the transition's arguments where it is assumed that the other parameters proceed through the transition unchanged. Furthermore, as these judgments are relatively straightforward, we omit a detailed description of them.

The second argument to the functional transition, T, is a mapping from process identifiers to program states and is used to represent the currently running processes. It is necessary for forking new processes, as seen in the following two judgments to handle the *fork* operator:

$$\textsc{Ft-Fork}\ \frac{p\ \mathsf{fresh}, \quad T' = T[p \mapsto \varepsilon \rhd (e, (), \emptyset)]}{(K \rhd (\mathit{fork}\ e, x, U), T) \Rrightarrow (K \lhd (\mathit{fork}\ e\ p, x, U), T')}$$

$$\textsc{Ft-Fork}_p\ \frac{T' = \textbf{if}\ p \in \mathit{Dom}(T)\ \textbf{then}\ T\ \textbf{else}\ T[p \mapsto \varepsilon \rhd (e, (), \emptyset)]}{(K \rhd (\mathit{fork}\ e\ p, x, U), T) \Rrightarrow (K \lhd (\mathit{fork}\ e\ p, x, U), T')}$$

Note that we allow the *fork* operator to take an optional additional process identifier argument. Thus, in the judgment Ft-Fork, we fork a new process with a fresh process ID, and in the $\textsc{Ft-Fork}_p$ judgment, the identifier is available. Although in this second judgment we know that we have forked before, we still must check to see if the forked process is in T in case it was terminated due to a choice statement. Also, note that this judgment assumes that $\mathscr{R}$ and $\mathscr{W}$ are held constant through the transition.

The parallel judgments to the Ft-Fork ones are those for governing the behavior of choice ($|||$). Recall that CFRP utilizes the ideas of non-interfering choice to allow the branching decision to affect the program behavior. Specifically, any forked processes from an un-taken branch of a choice expression are terminated. In order to express this behavior, we make use of the following meta-level function over T:

$$\mathsf{getChildrenOf} :: T \to (\alpha \overset{R}{\rightsquigarrow} \beta) \to T$$

$$\textsc{Ft-Eval}\ \frac{e \mapsto e'}{(K \rhd (e,x,U)) \Rrightarrow (K \rhd (e',x,U))}$$

$$\textsc{Ft-Arr}\ \frac{}{(K \rhd (arr\ e,x,U)) \Rrightarrow (K \lhd (arr\ e,e\ x,U))}$$

$$\textsc{Ft-First}_1\ \frac{x \mapsto x'}{(K \rhd (first\ e,x,U)) \Rrightarrow (K \rhd (first\ e,x',U))}$$

$$\textsc{Ft-First}_2\ \frac{}{(K \rhd (first\ e,(x,z),U)) \Rrightarrow (K;(\cdot,z) \rhd (e,x),U)}$$

$$\textsc{Ft-First}_3\ \frac{}{(K;(\cdot,z) \lhd (e,y),U) \Rrightarrow (K \lhd (first\ e,(y,z),U))}$$

$$\textsc{Ft-Comp}_1\ \frac{}{(K \rhd (e_1 \ggg e_2,x,U)) \Rrightarrow (K;(\cdot \ggg e_2) \rhd (e_1,x,U))}$$

$$\textsc{Ft-Comp}_2\ \frac{}{(K;(\cdot \ggg e_2) \lhd (e_1,y,U)) \Rrightarrow (K;(e_1 \ggg \cdot) \rhd (e_2,y,U))}$$

$$\textsc{Ft-Comp}_3\ \frac{}{(K;(e_1 \ggg \cdot) \lhd (e_2,z,U)) \Rrightarrow (K \lhd (e_1 \ggg e_2,z,U))}$$

Figure 5.4: The functional transition judgments for the standard arrow combinators. The process data (T), resource map ($\mathscr{R}$), and wormhole map ($\mathscr{W}$) are all assumed to be held constant.

The getChildrenOf function takes a process data map and a signal function and returns the set of process data that the given signal function has caused to be forked. It is used in the judgments describing the behavior choice as seen in Figure 5.5. The $\textsc{Ft-Chc}_*$ judgments show the functioning of the choice operator. For the most part, this behavior is typical of non-interfering choice, but we take one additional step. When choosing the left or right branch (in $\textsc{Ft-Chc}_{l1}$ or $\textsc{Ft-Chc}_{r1}$), we remove all processes from T that were produced by the other branch.

The last two arguments to the functional transition are associated with resources. The mapping $\mathscr{R}$ maps resources to resource data. Each resource may have a different type of resource data, but regardless, the resource and its data must implement the following two functions:

$$\begin{aligned} \mathsf{read} &:: \langle \tau_{in}, \tau_{out} \rangle \to \mathit{Data} \to \tau_{out} \\ \mathsf{update} &:: \langle \tau_{in}, \tau_{out} \rangle \to \mathit{Data} \to \tau_{in} \to \langle \tau_{in}, \tau_{out} \rangle \end{aligned}$$

where *Data* represents the associated data type for the given resource. The read function returns the current output value of the given resource, merely "peeking" at what is there without affecting it in any way. The update function takes an input value for the resource and returns an updated version of that resource. In practice, as can be seen in the transition judgments, calls to these functions will generally be of the form read $r\ \mathscr{R}(r)$ and similar for update. As might be expected, reading can happen at any time, but updating only happens at a time step.

Because wormholes need to share an internal sense of state, we cannot simply add two resources to $\mathscr{R}$. Instead, we use a layer of indirection in the form of $\mathscr{W}$. We create a dummy resource in $\mathscr{R}$ that contains the wormhole state and then use $\mathscr{W}$ to map both the blackhole and whitehole virtual resources to that dummy. We additionally include an identifier (either "W" or "B") to keep track of whether this resource is from a whitehole or blackhole.

$$\textsc{Ft-Chc}_e\ \frac{x \mapsto x'}{(K \rhd (e_1 \,|||\, e_2, x, U), T) \Rrightarrow (K \rhd (e_1 \,|||\, e_2, x', U), T)}$$

$$\textsc{Ft-Chc}_{l1}\ \frac{T' = T \setminus (\mathsf{getChildrenOf}\ T\ e_2)}{(K \rhd (e_1 \,|||\, e_2, \mathit{Left}\ x, U), T) \Rrightarrow (K; (\cdot \,|||\, e_2) \rhd (e_1, x, U), T')}$$

$$\textsc{Ft-Chc}_{l2}\ \frac{}{(K; (\cdot \,|||\, e_2) \lhd (e_1, z, U), T) \Rrightarrow (K \lhd (e_1 \,|||\, e_2, z, U), T)}$$

$$\textsc{Ft-Chc}_{r1}\ \frac{T' = T \setminus (\mathsf{getChildrenOf}\ T\ e_1)}{(K \rhd (e_1 \,|||\, e_2, \mathit{Right}\ y, U), T) \Rrightarrow (K; (e_1 \,|||\, \cdot) \rhd (e_2, y, U), T')}$$

$$\textsc{Ft-Chc}_{r2}\ \frac{}{(K; (e_1 \,|||\, \cdot) \lhd (e_2, z, U), T) \Rrightarrow (K \lhd (e_1 \,|||\, e_2, z, U), T)}$$

Figure 5.5: The functional transition judgments for choice. The resource map ($\mathscr{R}$) and wormhole map ($\mathscr{W}$) are assumed to be held constant.

$$\textsc{Ft-RSF}_r\ \frac{r \in \mathscr{R} \quad U' = (r, x) : U \quad y = \mathsf{read}\ r\ \mathscr{R}(r)}{(K \rhd (\mathit{rsf}\ r, x, U), \mathscr{R}, \mathscr{W}) \Rrightarrow (K \lhd (\mathit{rsf}\ r, y, U'), \mathscr{R}, \mathscr{W})}$$

$$\textsc{Ft-RSF}_w\ \frac{r \in \mathscr{W} \quad U' = (r, x) : U \quad y = \mathsf{read}\ r\ \mathscr{R}(\mathit{fst}\ \mathscr{W}(r))}{(K \rhd (\mathit{rsf}\ r, x, U)\mathscr{R}, \mathscr{W}) \Rrightarrow (K \lhd (\mathit{rsf}\ r, y, U')\mathscr{R}, \mathscr{W})}$$

$$\textsc{Ft-LetW}\ \frac{r\ \mathsf{fresh} \quad \mathscr{R}' = \mathscr{R}[r \mapsto (\varepsilon, e_i)] \quad \mathscr{W}' = \mathscr{W}[r_b \mapsto (r, \mathrm{B}), r_w \mapsto (r, \mathrm{W})]}{(K \rhd (\mathbf{letW}\ r_w\ r_b\ e_i\ \mathbf{in}\ e, x, U), \mathscr{R}, \mathscr{W}) \Rrightarrow (K \rhd (e, x, U), \mathscr{R}', \mathscr{W}')}$$

$$\textsc{Ft-Time}\ \frac{\begin{array}{l}\mathscr{R}_1 = \mathscr{R}\ \left[r \mapsto \mathsf{update}\ r\ \ \mathscr{R}(r)\ \ x \mid\ (r, x)\ \in U, r \in \mathscr{R}\right] \\ \mathscr{R}_2 = \mathscr{R}_1\ \left[r \mapsto \mathsf{update}\ r_b\ \mathscr{R}_1(r)\ x \mid (r_b, x) \in U, \mathscr{W}(r_b) = (r, \mathrm{B})\right] \\ \mathscr{R}_3 = \mathscr{R}_2\ \left[r \mapsto \mathsf{update}\ r_w\ \mathscr{R}_2(r)\ x \mid (r_w, x) \in U, \mathscr{W}(r_w) = (r, \mathrm{W})\right]\end{array}}{(\varepsilon \lhd (e, (), U), \mathscr{R}, \mathscr{W}) \Rrightarrow (\varepsilon \rhd (e, (), \emptyset), \mathscr{R}_3, \mathscr{W})}$$

Figure 5.6: The functional transition judgments that include resource and wormhole management. The process data (T) is assumed to be held constant.

The wormhole resource state consists of a pair of queues (b, w), where b is the accumulation of blackhole data and w is the next readable value for the whitehole. We define the read and update for wormhole resources as follows:

$$\begin{array}{l}\mathsf{read}\ r_w\ (b, w) = w \\ \mathsf{update}\ r_w\ (b, w)\ () = (\varepsilon,\ b) \\ \mathsf{read}\ \ r_b\ (b, w) = () \\ \mathsf{update}\ \ r_b\ (b, w)\ \ x\ = (b; x,\ w)\end{array}$$

The functional transitions that make use of $\mathscr{R}$ and $\mathscr{W}$ are the most critical transitions of CFRP, and they can be seen in Figure 5.6. These judgments behave as expected given our intuitive descriptions of the operations from the beginning of the chapter and the immediately preceding descriptions of the parameters, with two notes:

- Once virtual resources are added to $\mathscr{W}$ and $\mathscr{R}$ by the Ft-LetW judgment, they will not need to be added again, so the expression is modified to exclude the *letW* operator on return. Despite this,

$$
\begin{aligned}
\varepsilon \bowtie e &= e \\
K;(\cdot, z) \bowtie e &= K \bowtie \mathit{first}\ e \\
K;(\cdot \ggg e_2) \bowtie e &= K \bowtie e \ggg e_2 \\
K;(e_1 \ggg \cdot) \bowtie e &= K \bowtie e_1 \ggg e \\
K;(\cdot \,|||\, e_2) \bowtie e &= K \bowtie e \,|||\, e_2 \\
K;(e_1 \,|||\, \cdot) \bowtie e &= K \bowtie e_1 \,|||\, e
\end{aligned}
$$

Figure 5.7: The definition of the frame application operator $\bowtie$ used by the unwinding operator $\hookrightarrow$.

wormholes cannot be created entirely with a pre-processor due to the fact that a process that is forked, terminated, and then forked again must recreate its wormholes. As this is a dynamic operation, it must be handled here.

- Rather than deal with a particular form of an expression, the FT-TIME judgment handles the case where the program is in a *return* state with an empty control stack. This state signifies that the signal function has run its course for this time step. All update data in U is processed, updating $\mathscr{R}$ as necessary, and the program begins again with the program state changing from *return* to *evaluation* and a new empty set of update data.

Structural Preservation of the Functional Transition

The purpose of having an expression in the *return* program state in the functional transition is to allow the transition to apply a code transformation. This transformation is useful to allow wormholes that have been executed to not be executed again in the future. We assert that this transformation will not negatively impact the behavior or functionality of the code, but to state this more precisely, we first define a notion of *unwinding* a program state.

Definition 5 (Unwinding)**.** *If S is a program state of either the form $K \rhd (e,v,U)$ or $K \lhd (e,v,U)$, then $S \hookrightarrow e'$ (read S* unwinds *to e') where $e' = K \bowtie e$. (The $\bowtie$ operation is shown in Figure 5.7.)*

Basically, this gives us a way to examine the entire program so that we can compare it before and after any transformations. We use this to show that our transformation affects the program in only very specific ways.

Theorem 4 (Structural Preservation)**.** *If S is a program state, $S \hookrightarrow e :: \alpha \overset{R}{\rightsquigarrow} \beta$ and $(S,T,\mathscr{R},\mathscr{W}) \Rrightarrow (S',T',\mathscr{R}',\mathscr{W}')$, then $S' \hookrightarrow e' :: \alpha \overset{R'}{\rightsquigarrow} \beta$ such that e' will be identical to e except that:*

- *There may be code in e' that has been further evaluated by the evaluation transition than in e (as per* FT-EVAL*),*
- *Wormhole expressions in e may be replaced by their bodies in e' and there are corresponding updates in $\mathscr{W}'$ and $\mathscr{R}'$ (as per* FT-LETW*).*

The proof of this theorem follows directly from an analysis of the functional transition judgments.

Program Execution

With the functional transition well defined, we can discuss the overall execution of a program in CFRP. Because of the asynchrony inherent in the language, we intuitively want to think of the program as running

multiple signal functions at once. However, we describe it technically as only one process running at a time, with a non-deterministic choice between which one runs.

The following judgment describes the executive transition, which defines program execution:

$$\textsc{Exec}\quad \frac{(p,S) \in T \qquad (S, T \setminus \{(p,S)\}, \mathscr{R}, \mathscr{W}) \Rrightarrow (S', T', \mathscr{R}', \mathscr{W}')}{(T, \mathscr{R}, \mathscr{W}) \Downarrow (T' \cup \{(p,S')\}, \mathscr{R}', \mathscr{W}')}$$

Note that the choice of (p,S) from T is made non-deterministically and fairly. Thus, this transition arbitrarily chooses a process and runs one step of its execution. The returned process data is the set T', itself a result of the functional transition, extended with the process that just ran (p,S').

We can define the execution of a whole program as a sequence through this EXEC transition as follows:

Definition 6 (Program Execution)**.** *Program execution is the application of the reflexive transitive closure over the* EXEC *transition* $\Downarrow$ *starting with initial parameters* $T = \{(p, \varepsilon \triangleright (e, (), \varepsilon))\}$, $\mathscr{R} = \mathscr{R}_o$, *and* $\mathscr{W} = \emptyset$ *where* p *is a fresh process ID,* e *is a process, and* $\mathscr{R}_o$ *is an initial mapping of resources representing those of the real world.*

5.4 Concurrency Operators

In Section 5.2, we showed some examples of the higher level constructs that can be built in CFRP. Now that we have defined the syntax of CFRP, we will define these constructs and display their underlying principles.

5.4.1 Asynchrony

Asynchrony is trivially available in CFRP via the *fork* primitive. That said, it allows no direct communication between processes. However, by using *fork* in conjunction with wormholes, we can easily create an *async* operator:

$$\begin{aligned}
&\mathit{async} :: \left(\mathit{List}\ \alpha \overset{R}{\rightsquigarrow} \beta\right) \to \left(\alpha \overset{R}{\rightsquigarrow} \mathit{List}\ \beta\right) \\
&\mathit{async}\ \mathit{sf} = \textbf{letW}\ r_{wi}\ r_{bi}\ \varepsilon\ \textbf{in letW}\ r_{wo}\ r_{bo}\ \varepsilon\ \textbf{in}\ (\mathit{fork}\ g \ggg f) \\
&\quad \textit{where}\ f = \mathit{rsf}\ r_{bi} \ggg \mathit{rsf}\ r_{wo} \\
&\qquad\quad g = \mathit{rsf}\ r_{wi} \ggg \mathit{sf} \ggg \mathit{rsf}\ r_{bo}
\end{aligned}$$

Quite simply, *async* creates two wormholes, one for input values and one for output values, and then uses them like channels between the main process and the forked process.

5.4.2 Parallel Composition

In Section 5.2.4, we examined a signal processing pipeline that made use of parallelizing composition to link together different signal functions such that they could each process as fast as possible. This $>|>$ function essentially creates two forked processes: one to do the parallel job and one to accept the result of that job.

$$\begin{aligned}
&(>|>) :: (R_1 \cup R_2 = R, R_1 \cap R_2 = \emptyset) \Rightarrow \\
&\qquad \left(\alpha \overset{R_1}{\rightsquigarrow} \mathit{Event}\ \beta\right) \to \left(\mathit{Event}\ \beta \overset{R_2}{\rightsquigarrow} ()\right) \to \left(\alpha \overset{R}{\rightsquigarrow} ()\right) \\
&\mathit{sf}_1\ >|>\ \mathit{sf}_2 = \textbf{letW}\ r_w\ r_b\ \varepsilon\ \textbf{in}\ (\mathit{fork}\ g \ggg f) \\
&\quad \textit{where}\ f = \mathit{sf}_1 \ggg \mathit{rsf}\ r_b \\
&\qquad\quad g = \mathit{buffer}\ (\mathit{rsf}\ r_w) \ggg \mathit{sf}_2
\end{aligned}$$

To simplify the definition of this function, we have made use of the function *buffer*, which is shown in Figure 5.8. The *buffer* function takes a signal function that returns a *List* of *Event*s and converts it into a

```
buffer :: (α ~R~> List (Event β)) → (α ~R~> Event β)
buffer sf = letW r_w r_b (ε : ε) in proc x → do
  (b : _) ← rsf r_w ⪦ ()
  elements_new ← sf ⪦ x
  case (b ++ compress elements_new) of
    ε       → do _ ← rsf r_b ⪦ ε
                 returnA ⪦ r
    (y : ys) → do _ ← rsf r_b ⪦ ys
                 returnA ⪦ y

compress ε = ε
compress (NoEvent : rst) = compress rst
compress (Event x : rst) = Event x : compress rst
```

Figure 5.8: The *buffer* function.

signal function that returns *Event*s one at a time. Essentially, the list is compressed to just its events—if there are more than one, then they are buffered and returned one at a time, and if there are none and the buffer is empty, then *NoEvent* is returned.

As long as there are no wormholes or resources permitting the flow of information from e_2 back to e_1, then $>|>$ can provide something resembling deterministic parallelism even though it is made of only non-deterministic, asynchronous components.

5.4.3 Speculative Parallelism

In Section 5.2.3, we discussed the idea of speculative parallelism and that it can be achieved in CFRP. In Figure 5.9, we show the definition of the *spar* function.

The *spar* function starts by creating a new wormhole that it uses to keep track of whether it should continue or not. If it should continue, then it sends the impulse event stream to asynchronized versions of its two input signal functions and observes their buffered outputs. Because we only expect a single event in the output, we can buffer them simply to reduce the type from *List* (*Event* α) to *Event* α. If either signal function produces an event, then we return it and set *continue* to *False* by sending *False* into the blackhole. the wormhole. Otherwise, we set *continue* to *True* and output *NoEvent*.

If the wormhole indicates that the speculative parallelism is complete (that is, if *continue* is *False*), then we choose the second branch, which simply reasserts that we should not continue and outputs *NoEvent*. Note that by choosing the second branch, we are also effectively halting the progress of the asynchronized processes, preventing any unneeded computation if one of the processes is still trying to produce an output.

5.5 Language Properties

CFRP satisfies two important properties that we highlight in this section: resource safety and resource commutativity. We will provide an intuitive sense for these two properties, but a formal treatment can be found in Appendix A.4.

We begin with a concept of a *moment in time*. One moment is the computation between time steps that a given CFRP process executes. The idea of a moment in time comes from the fundamental abstraction of FRP, in that one moment represents the simultaneous execution of all data with the same time stamp.

```
spar :: (R1 ⊎ R2 = R) ⇒ (Event α ~>[R1] Event β) →
        (Event α ~>[R2] Event γ) → (Event α ~>[R] Event (β + γ))
spar sf1 sf2 = letW rw rb (False : ε) in proc a → do
  (continue : _) ← rsf rw ⪦ ()
  if continue || isEvent a then do
      el ← buffer (async (arr collapse >>> sf1)) ⪦ a
      er ← buffer (async (arr collapse >>> sf2)) ⪦ a
      case (el, er) of
        (Event b, _) → do _ ← rsf rb ⪦ False
                          returnA ⪦ Event (Left b)
        (_, Event c) → do _ ← rsf rb ⪦ False
                          returnA ⪦ Event (Right c)
        _            → do _ ← rsf rb ⪦ True
                          returnA ⪦ NoEvent
  else do
      _ ← rsf rb ⪦ False
      returnA ⪦ NoEvent

collapse ε = NoEvent
collapse (NoEvent : rst) = collapse rst
collapse (Event x : rst) = Event x
```

Figure 5.9: The *spar* function.

The notion of resource safety starts with the guarantee that a signal function of type $\alpha \overset{R}{\rightsquigarrow} \beta$ will not interact with any resource $r \notin R$. That is, resources that are not represented in the type of a signal function will not be read or updated by that signal function.

This idea extends in two directions. First, we can consider the ramifications of this in an asynchronous setting. Resource safety gives us the guarantee that no two processes can interact with the same resources, which in turn means that multiple processes cannot encounter any of the typical problems of resource contention.

In another direction, we can look at resource safety from a temporal perspective and state that within any given moment in time, a process cannot interact with any resource more than once. This satisfies the fundamental abstraction nicely: if the same resource were accessed twice at the same time, then there must be an ordering to its access, but any ordering would imply that the moment itself was not processed entirely simultaneously.

An extension to the idea of resource safety is that of resource *commutativity*. Once again, this comes from the fundamental abstraction. If the order of access of two resources within the same moment makes a difference, then this implies an ordering within the moment, which in turn implies a lack of simultaneity.

CFRP has both resource safety and commutativity.

5.6 Blocking

One might think that it would be useful to allow blocking in CFRP, and indeed, an ability to block would be critical to a high performance implementation of CFRP. Furthermore, blocking would allow two asynchronous processes to resynchronize, essentially providing the ability to do synchronized parallelism.

5.6.1 Blocking *rsf*

One method to achieve blocking would be to include a new version of the *rsf* operator: a *blocking* resource signal function, which we could call *brsf* and would have the following typing rule:

$$\textsc{Ty-BRSF}\ \frac{\left(r :: \langle \tau_{in}, \mathit{List}\ \tau_{out} \rangle\right) \in \Psi}{\Gamma; \Psi \vdash \mathit{brsf}\ r :: \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}$$

Notice that *brsf* requires its resource to have an output type that is a list but that it returns the output of only a single element. The idea here is that *brsf* will block while the resource provides an empty list and will then provide values one at a time as they appear.

If used on a standard resource r, *brsf* r will only progress through time and return a value when r produces a value. It cannot be used on a blackhole (the outputs type of a blackhole resource is not a list), but on a whitehole, it will provide one datum that was provided by the blackhole.

With the *brsf* operator, it becomes possible to resynchronize two asynchronous processes. Instead of having the two processes communicate via two wormholes accessed with *rsf*, like we did when defining the *async* function (in Section 5.4.1), we let the whitehole resources of both wormholes be accessed with blocking. This forces both processes to wait for each other to proceed, giving them a synchronous behavior.

The main drawback of the *brsf* operator and its design is that we lose our strongest guarantees in its presence: it breaks the fundamental abstraction of FRP. That is, to use the parlance of the previous section, a use of *brsf* will interrupt a *moment in time*. Thus, despite its apparent usefulness, we omit it and any other blocking operators from the language.

$$
\begin{aligned}
&\mathit{brsf} :: (r :: \langle x, \mathit{List}\ y\rangle) \to (y \overset{R}{\rightsquigarrow} z) \to (x \overset{R\cup\{r\}}{\rightsquigarrow} \mathit{Event}\ z)\\
&\mathit{brsf}\ r\ \mathit{sf} \overset{\textbf{def}}{=} \textbf{letW}\ r_w\ r_b\ \varepsilon\ \textbf{in proc}\ x \to \textbf{do}\\
&\quad y \leftarrow \mathit{rsf}\ r \prec x\\
&\quad \mathit{prev} \leftarrow \mathit{rsf}\ r_w \prec ()\\
&\quad \textbf{case}\ (\mathit{concat}\ \mathit{prev} +\!\!+ y)\ \textbf{of}\\
&\qquad \varepsilon \qquad\quad \to \mathit{returnA} \prec \mathit{NoEvent}\\
&\qquad (y' : ys) \to \textbf{do}\ () \leftarrow \mathit{rsf}\ r_b \prec ys\\
&\qquad\qquad\qquad\quad z \leftarrow \mathit{sf} \prec y'\\
&\qquad\qquad\qquad\quad \mathit{returnA} \prec \mathit{Event}\ z
\end{aligned}
$$

Figure 5.10: A potential definition of *brsf*. Note that the *concat* function will concatenate the elements of a list of lists into a single list.

5.6.2 Simulated Blocking

Because of the power of non-interfering choice, we can design a local version of blocking that does not disrupt our sense of time and thus preserves the fundamental abstraction.

Consider the definition of *brsf* given in Figure 5.10. In our semantics, resources cannot be treated as they are here (as first class values that could be arguments to functions), so this is not a function we could ordinarily define, but it presents an interesting idea. Once per moment in time (or in other words, on each iteration of this signal function), the given resource r is accessed. If it returns an empty list, this indicates that it has no ready data, and the first branch of the case statement takes effect; in this branch, the argument signal function is not run. However, if there is data, then the signal function is run. Furthermore, the data gets buffered by the wormhole, meaning that if accessing r returns too much, the data is fed to *sf* slowly, one element at a time.

This version of *brsf* does not break the fundamental abstraction, but it is also somewhat weaker than true blocking. This is because true blocking has a non-local effect on its process, preventing the entire process from doing anything while the blocking occurs. However, *brsf* will only prevent the argument signal function from acting, and it will have no effect on any other signal function in this process (e.g. the compositional context).

One consequence of *brsf*'s weak form of blocking is that it may often have a negative performance impact on the programs that make use of it. That is, when a *brsf* is blocking, its behavior mimics "busy-waiting," where it continues to loop even though it is accomplishing nothing. This is entirely necessary when the compositional context should run regardless of the blocking, but it seems unfortunate in most cases. Thus, implementations of CFRP are encouraged to provide a special combination of forking and blocking that can assure there is no compositional context to the blocking in the forked process and then use true blocking instead of *brsf*'s weak blocking to achieve better performance.

Thus, we have the ability to simulate blocking on a local scale, but we cannot achieve true process blocking in general without breaking the fundamental abstraction.

5.7 Haskell Implementation

Just as we have extended the theory of arrowized, resource typed signal functions to include asynchrony, so too have we extended our Haskell implementation. Thus, the implementation discussion here will build directly off of where we ended at the end of Section 4.7. Once again, we make certain modifications to the theory we have presented in order to satisfy Haskell's particular constraints. Note also that this

```
instance Resource (Whitehole r t) () [t] where
  read (Whitehole (_, w)) = readIORef w
  update (Whitehole (b, w)) _ = do
    b_data ← atomicModifyIORef b (λ l → ([], l))
    atomicModifyIORef w (λ _ → (reverse b_data, ()))

instance Resource (Blackhole r t) t () where
  read _ = return ()
  update (Blackhole (b, _)) t = atomicModifyIORef b (λ l → (t : l, ()))
```

Figure 5.11: The updated definitions of wormhole resources for CFRP.

implementation suffers from the same pitfalls and limitations as does the one in the previous chapter.

5.7.1 Updating Wormholes

Our first goal will be to update wormholes to satisfy our new specification for them. In the synchronous FRP that we described earlier, whiteholes read the value in the wormhole data structure but never write to that value. In CFRP, they cannot be this simple. Rather, when data is read, the whitehole needs to remove it from the underlying data structure so that it is not read a second time. Thus, the whitehole's *update* function will perform an effect.

Furthermore, it is imperative that whiteholes be processed *before* their corresponding blackholes if they are both in the same process. If the whitehole is processed *after*, then blackhole data may not be visible when the whitehole next reads. This was implicit in the design of the FT-TIME transition judgment, but we must make it explicit here.

We begin by updating the *Whitehole* and *Blackhole* types to more closely match the CFRP wormhole semantics:

```
newtype Whitehole r t = Whitehole (IORef [t], IORef [t])
newtype Blackhole r t = Blackhole (IORef [t], IORef [t])
```

A wormhole consists of two *IORef*s, which correspond to the *b* and *w* elements of the wormhole state from the semantics respectively. Both the whitehole and the blackhole resources each have access to both of them (i.e. the first *IORef* of the whitehole and the blackhole both point to the same data and likewise for the second).

The *Resource* instances will look familiar, but they are updated to deal with lists of data (as opposed to individual elements) as well as to include the whitehole's *update* behavior. They are shown in Figure 5.11. Note that we perform a typical functional queue optimization here of constructing the list of data in reverse and then reversing it when it is requested.

Now, both whiteholes and blackholes have an effect upon updating. The blackhole adds a new element to the underlying data structure, and the whitehole removes what it has read.

Next, we must address the ordering and assure that whiteholes are processed before blackholes. Recall from Section 4.7.5 that our running definition of the data type *SF* is:

```
data SF r b c = SF (b → IO (c, IO (), SF r b c))
```

where the *IO* () in the output tuple is used to carry the update actions. We can achieve the ordering we want by expanding the *SF* type so that instead of having a single action type (*IO* ()) to denote the updates, we

have a pair: $(IO\ (), IO\ ())$:

$$\textbf{data}\ SF\ r\ b\ c = SF\ (b \to IO\ (c, (IO\ (), IO\ ()), SF\ r\ b\ c))$$

We then update the default implementation of *rsf* to:

$$\begin{aligned} rsf\ r = SF\ \$\ \lambda\ b \to\ & \textbf{do} \\ & c \leftarrow read\ r \\ & return\ (c, (return\ (), update\ r\ b), rsf\ r) \end{aligned}$$

and in the *Resource* instance for *Whitehole*, we overwrite that definition with the following:

$$\begin{aligned} rsf\ r = SF\ \$\ \lambda\ b \to\ & \textbf{do} \\ & c \leftarrow read\ r \\ & return\ (c, (update\ r\ b, return\ ()), rsf\ r) \end{aligned}$$

When we run the signal function, we choose the proper ordering of events, and then we are guaranteed that whitehole updates will always be before blackhole ones.

Lastly, the *letW* operation looks almost exactly the same as before except that the initialization value for the wormhole is of type $[t]$ rather than simply t and there are two *IORef*s instead of one. This has the interesting effect of allowing one to create wormholes that have no initial value in them, a trick which we used earlier in some of the parallelism examples.

5.7.2 Forking New Processes

In the semantics we defined in this chapter, the *fork* operation makes use of the helper functions haveIForked and getChildrenOf to determine if it needs to fork a new process or not. In Haskell, these two functions are non-trivial, and we approach the problem from a different perspective.

Rather than actually terminate processes that should be inactive, we instead *freeze* them. That is, we keep them alive, but we prevent them from having any noticeable effects. Then, if they ever need to become active again, we can simply unfreeze them. This strategy allows us to sidestep the question of haveIForked, as any *fork* operation will only ever spawn one new process.

Rather than going into excruciating detail, it should suffice to say that we will extend our *SF* type to additionally include process status information: At any time, we can see an *MVar* which contains the current process's status as well as a reference to a list of the status *MVar*s of all child processes. The status will be one of the following:

$$\begin{aligned} \textbf{data}\ PStatus = \ & Proceed \\ \mid\ & ShouldFreeze \\ \mid\ & ShouldSkip \\ \mid\ & Frozen\ (MVar\ ()) \\ \mid\ & Die \end{aligned}$$

and the *SF* type will become:

$$\textbf{type}\ PState = (MVar\ PStatus, PChildren)$$

$$\textbf{data}\ SF\ r\ b\ c = SF\ ((b, PState) \to IO\ (c, (IO\ (), IO\ ()), SF\ r\ b\ c))$$

where *PChildren* represents the child processes.

By default, processes are in the *Proceed* state, which indicates that they should proceed as normal. If, in the course of execution, a choice branch is taken that would deactivate certain processes, then those

```
runSF :: PState → () ⇝^R () → IO ()
runSF (ps@(mvar, _)) (SF sf) = run sf where
  run sf = do
    ((), (action1, action2), sf') ← sf ((), ps)
    command ← takeMVar mvar
    case command of
      Proceed → action1 >>> action2 >>> putMVar mvar Proceed >>> run sf'
      ShouldFreeze → do
        wait ← newEmptyMVar
        putMVar mvar (Frozen wait)
        takeMVar wait
        run sf
      ShouldSkip → putMVar mvar Proceed >>> run sf
      Die → putMVar mvar Die
      Frozen _ → error "Impossible: Frozen in runSF''
```

Figure 5.12: The definition of *runSF* that can handle asynchrony.

processes' states are set to *ShouldFreeze*; a process that "should freeze" will freeze itself when it is next able by switching itself to the *Frozen* state, where it will additionally generate a new *MVar* that it will block on. If a choice branch is taken such that it activates processes, then any child processes that are currently in the *ShouldFreeze* state are set to *ShouldSkip*, and any that are *Frozen* are awakened (their *MVar*s are unblocked) and set to *Proceed*. Thus, if a process is frozen and unfrozen so quickly that it did not even have time to properly freeze itself, then it will be alerted to skip its current run and restart. Lastly, there is a state to terminate the process altogether that can be used for cleaning up asynchronous processes when the program ends.

These states allow us to update the *runSF* function, which we show in Figure 5.12. We can see that during the **case** statement in the body, we check to see what state the process is in before continuing. Only if it is in a *Proceed* state do we perform the effects ($action_1 >>> action_2$). In any other case, we consider the situation as a failed transaction and either block and wait (*ShouldFreeze*), restart (*ShouldSkip*), or abort altogether (*Die*).

Fork Itself

Forking is essentially a special internal use of running a signal function, so it follows that *fork* will make use of *runSF*. More precisely, the *fork* command will create a new *PState* for the child process by copying the current process's state, add the child to the list of children in its own state, and then use GHC's underlying thread forking to create a new thread for the process.

Internally, *fork* could use any one of GHC's mechanisms for creating a new process. We choose the standard *forkIO* operator, but variants to support OS threads (GHC's *forkOS* function) or specific cores (GHC's *forkOn*) are fine too. As long as GHC is run with the `-threaded` flag, we have found that these perform comparably for the simple tasks we have tested.

5.7.3 Controlling Forked Processes

As discussed previously in the chapter, we control forked processes not with a thread identifier or another such imperative representation, but by using the choice operator to freeze or resume threads. We discussed the machinery for this in the previous subsection, but here we will further explore the mechanism within

choice itself. To simplify this discussion, we will only explore the implementation of the *left* operator.

In the previous subsection, we mentioned a data type *PChildren*, which we use to store information about the states of any child processes. The actual definition is given by:

data *PChild* = *PForkChild PState* | *PChoiceChild PChildren*
type *PChildren* = *IORef* [*PChild*]

The idea here is that every time we fork, we create a new process with its own state, and every time we enter a choice statement, we create a new set of children that we can easily freeze or resume.

When we enter a *left* statement, we use *initialAIO* to create a new *PChildren* reference, which we will then pass to the body of the *left* (or just store for later if the incoming streaming value is *Right*). Then, if any component of the body forks a process, it will be added to that *PChildren* reference and we will be able to access it directly. However, before we do, we check to make sure that we, the parent process, are in a *Proceed* state. If we are not, then we should not make any active changes, but if we are, then we can tell the children to either freeze or proceed as necessary.

Other than this extra bookkeeping, choice proceeds in the typical fashion of the Kleisli Automaton fashion.

Chapter 6

UISF – A Case Study

One common application of functional reactive programming is in the design of *graphical user interfaces* (GUI). As a case study in the ideas of arrowized FRP and the concepts of non-interfering choice and resource types, we built the UISF library. UISF, which stands for "User Interface Signal Function" is built in Haskell on top of the GLFW graphics package and is currently being used as the main GUI toolkit for the computer music langauge Euterpea [Hudak, 2014]. The full UISF package is available on Hackage.[1]

This chapter will discuss the design principles behind UISF as well as demonstrate how it works with a few examples. Note that, as mentioned in the Haskell Implementation sections for resource types (Sections 5.7 and 4.7), Haskell's type system does not fully support resource types. Thus, although UISF has most of the operational capabilities discussed previously in this report, it cannot yet guarantee the same safety properties. However, in the absense of resource types, we can use the arrow syntax without any issues.

We will start with a technical description of the UISF interface in the next section. Once armed with the basics, we will build a few example GUIs, discussing the benefits of this design as we do.

6.1 Arrowized User Interface

The UISF library focuses on the *UISF* data type, which is an instance of *Arrow* (as well as *ArrowChoice*, etc.). In many ways, this data type is similar to the automaton models we have used in previous implementation sections of this report, but it is extended with further features specific to GUIs.

Using *UISF*, we can create "graphical widgets" using arrow syntax. Each signal function component of a *UISF* has the capacity to itself be a widget, such that upon composition, one can create compound widgets—in fact, it is in this fashion that the entire GUI is created.

Unlike in the rest of this report where we use the symbol $\rightsquigarrow$ to refer to an Arrow type, we will follow the library itself. Thus, instead of using types such as $a \rightsquigarrow b$, we will use *UISF a b*.

6.1.1 Graphical Input and Output Widgets

Some of UISF's basic widgets are shown in Figure 6.1. Note that each of them is, ultimately, a value of type *UISF a b*, for some input type *a* and output type *b*, and therefore may be used with the arrow syntax to help coordinate their functionality. The names and type signatures of these functions suggest their functionality, which we elaborate in more detail below:

- *label*: A simple (static) text string widget.

[1] The source can be found at `hackage.haskell.org/package/UISF`.

$$
\begin{array}{ll}
\mathit{label} & :: \mathit{String} \to \mathit{UISF}\ a\ a \\
\mathit{displayStr} & :: \mathit{UISF}\ \mathit{String}\ () \\
\mathit{display} & :: \mathit{Show}\ a \Rightarrow \mathit{UISF}\ a\ () \\
\mathit{textboxE} & :: \mathit{String} \to \mathit{UISF}\ (\mathit{Event}\ \mathit{String})\ \mathit{String} \\
\mathit{radio} & :: [\mathit{String}] \to \mathit{Int} \to \mathit{UISF}\ ()\ \mathit{Int} \\
\mathit{button} & :: \mathit{String} \to \mathit{UISF}\ ()\ \mathit{Bool} \\
\mathit{checkbox} & :: \mathit{String} \to \mathit{Bool} \to \mathit{UISF}\ ()\ \mathit{Bool} \\
\mathit{hSlider, vSlider} & :: \mathit{RealFrac}\ a \Rightarrow (a,a) \to a \to \mathit{UISF}\ ()\ a \\
\mathit{hiSlider, viSlider} & :: \mathit{Integral}\ a \Rightarrow a \to (a,a) \to a \to \mathit{UISF}\ ()\ a
\end{array}
$$

Figure 6.1: UISF graphical input/output widgets

- *displayStr*: A simple dynamic text string widget allowing a time-varying string to be displayed. For convenience, we also provide *display*, which "shows" the streaming argument:

$$\mathit{display} = \mathit{arr}\ \mathit{show} \ggg \mathit{displayStr}$$

- *textboxE*: A bidirectional text input widget. The input stream can be used to set the current text value, and the output stream provides that value. The *textboxE* keeps its state internally.

  There is a more primitive version:

$$\mathit{textbox} :: \mathit{UISF}\ \mathit{String}\ \mathit{String}$$

  which does not keep track of its current state but rather requires the manual use of a delay and a loop.

- *radio*, *button*, *checkbox*: These are three kinds of "push-buttons," suitable for retrieving input from the user in the form of choices between static options.

- **slider*: There are four different kinds of "sliders"—graphical widgets that looks like a slider control as might be found on a hardware device. The first two yield floating-point numbers in a given range, and are oriented horizontally and vertically, respectively, whereas the latter two return integral numbers. For the integral sliders, the first argument is the size of the step taken when the slider is clicked at any point on either side of the slider "handle." In each of the four cases, the other two arguments are the range and initial setting of the slider, respectively.

6.1.2 Widget Positioning

In addition to just creating widgets, we must determine where they will appear on the screen. UISF uses two mechanics to do this: *layout* and *flow*. A widget's layout determines its size, and its flow determines its relative position to its sister widgets.

All pre-built widgets (i.e. the ones presented in the previous subsection) have an already defined layout, but this can be altered with:

$$\mathit{setLayout} :: \mathit{Layout} \to \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b$$

and new layouts can be built using the following function:

$$
\begin{array}{l}
\mathit{makeLayout} :: \mathit{LayoutType} \to \mathit{LayoutType} \to \mathit{Layout} \\
\textbf{data}\ \mathit{LayoutType} = \mathit{Stretchy}\ \{\mathit{minSize} :: \mathit{Int}\} \\
\qquad\qquad\qquad\qquad\quad \mid\ \mathit{Fixed}\quad \{\mathit{fixedSize} :: \mathit{Int}\}
\end{array}
$$

$$
\begin{aligned}
&\mathit{unique} :: \mathit{Eq}\ a \Rightarrow \mathit{UISF}\ a\ (\mathit{Event}\ a)\\
&\mathit{edge} \quad :: \mathit{UISF}\ \mathit{Bool}\ (\mathit{Event}\ ())\\
&\mathit{hold} \quad :: a \rightarrow \mathit{UISF}\ (\mathit{Event}\ a)\ a\\
&\mathit{accum} :: a \rightarrow \mathit{UISF}\ (\mathit{Event}\ (a \rightarrow a))\ a
\end{aligned}
$$

Figure 6.2: UISF Mediators between continuous and discrete

The *makeLayout* function takes information for first the horizontal dimension and then the vertical. A dimension can be either stretchy (with a minimum size in pixels but that will expand to fill the space it is given) or fixed (measured in pixels).

The default flow for widgets is in a top-down format, where each widget will be placed from the top of the window sequentially. However, this can be changed by the following functions:

$$\mathit{topDown}, \mathit{bottomUp}, \mathit{leftRight}, \mathit{rightLeft} :: \mathit{UISF}\ a\ b \rightarrow \mathit{UISF}\ a\ b$$

whose names make clear their behavior.

One should note that this flow component means that the UISF arrows are *not* commutative. Indeed, reordering composition of widgets will likely cause their visual appearance to change. However, due to the **rec** keyword within arrow syntax (which uses arrow loop), this is rarely an issue: the widgets can be coded in whatever order makes them appear properly on screen, and the streams between them will connect properly.

Lastly, widget transformers can be nested, meaning that one part of a GUI can be in one flow while another portion is in another.

6.1.3 Non-Widget Signal Functions

Unlike the signal functions from Subsection 6.1.1, the signal functions presented in this subsection have no graphical effects. They are not pure—for pure signal functions, we could simply lift a pure function with *arr*—but their effects are all achieved with state rather than being visual. For this reason, many of the signal functions we will present here are not specific to *UISF* and can actually be used in other arrowized domains; however, for simplicity, we will express their types as specific to *UISF*. This also means that they can all be written manually using arrowized recursion, state via loop and delay, and other concepts discussed previously.

Mediators

Mediators are functions that *mediate* between discrete and continuous signals. A selection of UISF's mediators that we will use in our examples are shown in Figure 6.2 and described below:

- *unique*: Converts a continuous stream to a discrete one by providing an event containing the value of the stream whenever it changes.
- *edge*: Generates an event whenever the input changes from *False* to *True*.[2]
- *hold*: This signal function converts a discrete stream to a continuous one by "holding" the last value it was given.
- *accum*: Starting with the statically given value, applies the functions attached to the streaming input events to that value returning the result as a continuous stream.

[2] In signal processing this is called an "edge detector," giving rise to the name chosen here.

Folds

In regular functional programming, a folding, or reducing, operation is one that joins together a set of data. The typical case would be an operation that operates over a list of data, such as a function that sums all elements of a list of numbers.

The two primary folds in UISF are based on the ideas of structural or arrowized recursion as described in Section 3.1.5. For structural recursion, we have:

$$concatA :: [UISF\ b\ c] \rightarrow UISF\ [b]\ [c]$$

and for arrowized recursion, we have:

$$runDynamic :: UISF\ b\ c \rightarrow UISF\ [b]\ [c]$$

which is the very same function from Section 3.1.5.

The *concatA* fold takes a list of signal functions and converts them to a single signal function whose streaming values are themselves lists. For example, perhaps we want to display a bunch of buttons to a user in a single window. Rather than coding them in one at a time, we can use *concatA* to fold them into one operation that will return their results altogether in a list. In essence, we are *concat*enating the signal functions together.

As described earlier, the *runDynamic* signal function is similar except that it takes a single signal function as an argument rather than a list. Then, instead of folding over the static signal function list, it folds over the $[b]$ list that it accepts as its input streaming argument.

Timers

UISF has an implicit notion of elapsed time, but it can be made explicit by the following signal source:

$$getTime :: UISF\ ()\ Time$$

where *Time* is a type synonym for *Double*.

Although the explicit time may be desired, some UISF widgets depend on the time implicitly. For example, the following signal function creates a *timer*:

$$timer :: UISF\ DeltaT\ (Event\ ())$$

In practice, *timer* takes a stream that represents the timer interval (in seconds), and generates an event stream, where each pair of consecutive events is separated by the timer interval. Note that the timer interval is itself a stream, so the timer output can have varying frequency.

Because UISF is a pull-based AFRP system, this concept of time and timers is not perfectly precise or accurate. For instance, if the clock rate (e.g. the length of the unit time interval) is one hundredth of a second, then a timer may be triggered up to a hundredth of a second late.

Delays

Another way in which time can be used implicitly in UISF is in a *delay*. UISF comes with five different delaying widgets, which each serve a specific role depending on whether the streams are continuous or event-based and if the delay is a fixed length or can be variable. They are shown in Figure 6.3 and described below:

To start, we will examine the most straightforward one. The *delay* function creates what is called a "unit delay", which can be thought of as a delay by the shortest amount of time possible. This delay should be treated in the same way that one may treat a δt in calculus; that is, although one can assume that a delay

$$
\begin{aligned}
&\mathit{delay} &&:: a \to \mathit{UISF}\ a\ a\\
&\mathit{fcdelay} &&:: a \to \mathit{DeltaT} \to \mathit{UISF}\ a\ a\\
&\mathit{fdelay} &&:: \mathit{DeltaT} \to \mathit{UISF}\ (\mathit{Event}\ a)\ (\mathit{Event}\ a)\\
&\mathit{vdelay} &&:: \mathit{UISF}\ (\mathit{DeltaT}, \mathit{Event}\ a)\ (\mathit{Event}\ a)\\
&\mathit{vcdelay} &&:: \mathit{DeltaT} \to b \to \mathit{UISF}\ (\mathit{DeltaT}, b)\ a
\end{aligned}
$$

Figure 6.3: UISF Delays

takes place, the amount of time delayed approaches zero. Thus, in practice, this should be used only in continuous cases and should only be used as a means to initialize arrow feedback.

The rest of the delay operators delay by some amount of actual time, and we will look at each in turn. *fcdelay b t* will emit the constant value *b* for the first *t* seconds of the output stream and will from then on emit its input stream delayed by *t* seconds. The name comes from "fixed continuous delay."

One potential problem with *fcdelay* is that it makes no guarantees that every instantaneous value on the input stream will be seen in the output stream. This should not be a problem for continuous signals, but for an event stream, it could mean that entire events are accidentally skipped over. Therefore, there is a specialized delay for event streams: *fdelay t* guarantees that every input event will be emitted, but in order to achieve this, it is not as strict about timing—that is, some events may end up being over delayed. Due to the nature of events, we no longer need an initial value for output: for the first *t* second, there will simply be no events emitted.

We can make both of the above delay widgets a little more complicated by introducing the idea of a variable delay. For instance, we can expand the capabilities of *fdelay* into *vdelay*. Now, the delay time is part of the signal, and it can change dynamically. Regardless, this event-based version will still guarantee that every input event will be emitted. "*vdelay*" can be read "variable delay."

For the variable continuous version, we must add one extra input parameter to prevent a possible space leak. Thus, the first argument to *vcdelay* is the maximum amount that the widget can delay. Due to the variable nature of *vcdelay*, some portions of the input signal may be omitted entirely from the output signal while others may even be outputted more than once. Thus, once again, it is highly advised to use *vdelay* rather than *vcdelay* when dealing with event-based signals.

6.1.4 Asynchrony

Without resource types, wormholes are particularly dangerous, but UISF does allow certain forms of asynchronous, concurrent processing. Operationally, this is important due to system constraints on computational power. That is, there are two primary ways in which the illusion of continuity fails:

- Computations can be sensitive to the sampling rate itself such that a low enough rate will cause poor behavior.
- Computations can be sensitive to the variability of the sampling rate such that drastic differences in the rate can cause poor behavior.

These are two subtly different problems, and we address both with subtly different asynchronous operators:

$$
\begin{aligned}
&\mathit{asyncUISF}_E :: \mathit{NFData}\ b \Rightarrow \mathit{Automaton}\ (\to)\ a\ b \to \mathit{UISF}\ (\mathit{Event}\ a)\ (\mathit{Event}\ b)\\
&\mathit{asyncUISF}_V :: \mathit{NFData}\ b \Rightarrow \mathit{Double} \to \mathit{Automaton}\ (\to)\ a\ b \to \mathit{UISF}\ a\ [(b, \mathit{Time})]
\end{aligned}
$$

In fact, UISF has a few more asynchronizing operators, but we omit them in order to keep our discussion concise.

- *asyncUISF*$_E$: This takes an Automaton built over regular functions and makes it asynchronous, generally for the case where the given signal function is a slow running operation. This slow computation may have deleterious effects on the GUI, causing it to become unresponsive and slow, so we allow it to run asynchronously. The computation is lifted into the discrete, event realm, and for each input event given to it, a corresponding output event will be created eventually. Of course, the output event will likely not be generated immediately, but it will be generated eventually, and the ordering of output events will match the ordering of input events.

- *asyncUISF*$_V$: This function can convert a signal function with a fixed, virtual clockrate to a realtime UISF. The first input parameter is a buffer size in seconds that indicates how far ahead of real time the signal function is allowed to get, but the goal is to allow it to run at a fixed clockrate as close to realtime as possible. Thus, the output stream is a list of pairs providing the output values along with the timestamp for when they were generated. This should contain the right number of samples to approach real time, but on slow computers or when the virtual clockrate is exceptionally high, it will lag behind. This can be checked and monitored by checking the length of the output list and the time associated with the final element of the list on each time step.

In both cases, we require that the output types be instances of *NFData*, which is the Haskell way of declaring that they can be strictly evaluated. We do this to assure that the computations are actually performed asynchronously and not lazily returned to the main process and computed there.

UISF's asynchronous functions, although inspired by wormholes with fork, do not actually follow the design pattern from Chapter 5 very closely. Rather than use non-interfering choice to govern when forked processes are active or not, we use blocking, but because we have such rigid patterns for forking and communication between the forked processes (that is, one can only do this by using one of the async functions), this blocking cannot cause any sort of deadlock. Thus, we can use blocking without causing a perceivable violation to the Fundamental Abstraction of FRP.

This means that in their implementation, the async functions can use Haskell's *MVar*s, and indeed, they do. For instance, in the definition of *asyncUISF*$_V$, we fork a new thread and then communicate with it via *MVar*s in the data-sending direction and an *IORef* to retrieve computed values. If there is no data being sent to the forked thread in the *MVar*, then it will block, effectively stopping computation until it is asked to resume.

It is worth noting that the when using asynchrony in UISF, one is advised to compile the program with GHC's `-threaded` flag to allow for multi-core processor utilization. This is not strictly necessary as multi-threaded operations can be interleaved into a single-threaded computation, but it will often improve performance. Internally, we are using the *forkIO* operation to create new threads, which creates a lightweight Haskell thread for each asynchronous component. Because these threads stay alive for the length of the program, GHC can often schedule them effectively. However, in certain cases, a user may want better control over which cores are performing which operations, and thus we also provide "On" versions of the async operators. These "On" versions instead use GHC's *forkOn* operation, which allows the user to specify exactly on which core each forked thread should be run.

6.1.5 Settability

The UISF library has the concepts of non-interfering choice and settability built right into the design. Thus, UISF signal functions can also support the *settable* function established in Section 3.2:

$$settable :: UISF\ a\ b \rightarrow UISF\ (a, Event\ State)\ (b, State)$$

Any *UISF* signal function that is declared settable can have its state saved, loaded, or reset.

Because settability comes with a performance overhead, one has to make an active design decision to turn it on when it is required. In the future, we plan to improve the settability transformation by making it automatically apply when desired but incur no overhead when unused.

6.1.6 Putting It All Together

A Haskell program must eventually be a value of type *IO* () in order to run, and thus we need a function to turn a *UISF* value into an *IO* value—i.e. the *UISF* needs to be "run." We can do this using the following function[3]:

$$runUI :: UISF\ ()\ ()\ \rightarrow IO\ ()$$

Just like in the model langauges we described in previous sections, a full program is forced to do all of its effects internally, so its input and output streams must both be of type ().

Executing *runUI ui* will create a single UI window whose behavior is governed by the argument *ui* :: *UISF* () ().

6.2 Example: Time

For our first example, we will examine how easy it is to use time within the UISF framework. We will build a simple timer GUI that ticks forward for a user-specified amount of time (via a slider widget), displaying the elapsed time both graphically and textually. If the target time is greater than the elapsed time, the timer will continue, and if it is less than or equal to the elapsed time, then the timer will stop. A reset button at the bottom will reset the elapsed time to zero. The inspiration for this example comes from the 7GUIs project [Kiss, 2014], which in turn took the idea from Ignatoff et al. [2006].

Although we discussed a *timer* widget in the previous section, it is not useful for our current purposes, so the first thing we do is to create an alternative widget to help us keep track of time:

$$\begin{aligned}
&getDeltaTime :: UISF\ ()\ DeltaT\\
&getDeltaTime = \textbf{proc}\ ()\ \rightarrow \textbf{do}\\
&\quad t \leftarrow getTime \prec ()\\
&\quad t_{prev} \leftarrow delay\ 0 \prec t\\
&\quad returnA \prec t - t_{prev}
\end{aligned}$$

This function uses the *getTime* widget along with a unit delay to return the change in time on each tick of the underlying clock.

Next, although UISF has built-in widgets for displaying text, clickable buttons, and interactive sliders, there is no widget for displaying a "gauge" to graphically indicate the passing time. Although we did not discuss UISF's suite for manual widget construction, one still exists, and we will use it to create this gauge:

$$\begin{aligned}
&gauge :: Layout \rightarrow UISF\ (DeltaT, DeltaT)\ ()\\
&gauge = unique \ggg canvas'\ l\ draw\ \textbf{where}\\
&\quad draw\ (x,t)\ (w,h) = block\ ((0, padding),\\
&\qquad\qquad (min\ w'\ \ round\ \ x * (fromIntegral\ w')/t, h - 2 * padding))\\
&\qquad \textbf{where}\ w' = w - 2 * padding
\end{aligned}$$

This widget takes a pair of (elapsed time, total duration) and draws a black block on the screen of the appropriate size. Considering we have not discussed functions like *canvas′* or *block*, the point of showing this code is to demonstrate that creating a new widget like this can be done simply and concisely.

[3] Technically, this function is called *runUI′* in the UISF library as the actual *runUI* function takes an additional parameters argument that can be adjusted for special case fine-tuning. We will not use this argument in this report, so we drop it.

```
timerGUI :: UISF () ()
timerGUI = proc () → do
  rec leftRight (label "Elapsed Time:" >>> gauge) -< (e,d)
      display -< e
      leftRight (label "Duration:" >>> display) -< d
      d ← hSlider (0,30) 4 -< ()
      reset ← button "Reset" -< ()
      δt ← getDeltaTime -< ()
      e ← delay 0 -< case (reset, e >= d) of
            (True, _) → 0
            (False, True) → e
            _ → e + δt
  returnA -< ()
```

Figure 6.4: The Timer GUI.

Figure 6.5: A screenshot of the Timer GUI.

With these widgets written, constructing the timer GUI itself is straightforward. The GUI is shown in Figure 6.4 and a screenshot of it running can be seen in Figure 6.5.

6.3 Example: Bidirectional Data Flow

For our next example, we will consider the concept of *bidirectional* data flow. In many GUI systems, it is easy to link one widget's output to another's input, but it is not always as easy to link the second widget's output *back* to the first's input. However, this is a straightforward feature of arrows with loop, and the UISF library handles it easily.

Thus, for this example, we will build a lightweight utility to convert between temperatures in Celsius and temperatures in Fahrenheit. Once again, the idea for this example comes from the 7GUIs project [Kiss, 2014]. The complete code for this example can be seen in Figure 6.6 and a screenshot of it running can be seen in Figure 6.7.

The most complicated part of this GUI is actually the text parsing and conversion operations, which is all in pure Haskell. The part that actually presents the GUI is all found in the first four lines of arrow syntax. After that, we use the **let** construct to use pure Haskell for the parsing and conversions. Because the results of the conversions are used in the widgets themselves, the whole block is put through a loop with the **rec** keyword and the looping values are held back from infinitely looping with *delay*s.

```
tempConverter :: UISF () ()
tempConverter = leftRight $ proc () → do
  rec c ← unique ⋘ textboxE ⋘ delay Nothing ⊸ c_update
      label "degrees Celsius = " ⊸ ()
      f ← unique ⋘ textboxE ⋘ delay Nothing ⊸ f_update
      label "degrees Fahrenheit" ⊸ ()
      let c_num = join $ fmap (readMaybe :: String → Maybe Double) c
          f_num = join $ fmap (readMaybe :: String → Maybe Double) f
          c_update = fmap (λf → show $ round $ (f − 32) ∗ (5/9)) f_num
          f_update = fmap (λc → show $ round $ c ∗ (9/5) + 32) c_num
  returnA ⊸ ()
```

Figure 6.6: The Temperature Converter GUI. Note that ⋘ is the same as ⋙ but with its arguments flipped.

Figure 6.7: A screenshot of the Temperature Converter GUI.

6.4 Example: Dynamically Active Widgets

In the previous two examples, we demonstrated primarily UISF features that are inherited from the arrowized design. For this example, we will make use of UISF's adherence to non-interfering choice to use arrowized recursion to create a GUI that has widgets that can activate dynamically. We will build a text-based *mind map*, a structure to organize data.

Mind maps are typically used to help a person organize thoughts. They start with a single element (usually) that has connections to other elements, which in turn can have connections to others. We will represent our mind map data with a map from strings to lists of strings:

```
type MindMap = Map String [String]
```

Thus, elements are keys and the elements they connect to are their values.

Our GUI will allow a user to lookup keys or add elements to the mind map, and as the mapping grows, so too will the number of label widgets we display. In order to provide easy text entry, we will create a compound widget out of a textbox and a button:

```
textEntryField :: String → UISF () (Event String)
textEntryField txt = rightLeft $ proc () → do
  b ← edge ⋘ button txt ⊸ ()
  t ← textboxE "" ⋘ delay Nothing ⊸ fmap (const "") b
  returnA ⊸ fmap (const t) b
```

The *textEntryField* is given a label for the button it displays. When that button is pressed, this compound widget will produce an event of the current text in the textbox and then clear the textbox to prepare for the next entry.

We use two of these *textEntryField*s in the full program: one for looking up keys and the other for adding values. We store the map in an accumulator that updates every time we have an "add" event. Finally, we

```
mindmap :: UISF () ()
mindmap map_initial = proc () → do
  e ← textEntryField "Lookup" ⤙ ()
  a ← textEntryField "Add" ⤙ ()
  key ← hold "" ⤙ e
  m ← accum map_initial ⤙ fmap (λv → insertWith (++) key [v]) a
  leftRight (label "Key = " >>> displayStr) ⤙ key
  runDynamic displayStr ⤙ findWithDefault [] key m
  returnA ⤙ ()
```

Figure 6.8: The Mind Map GUI. Note that it requires the Haskell Map package to function, as that package provides the *insertWith* and *findWithDefault* functions that operate on Maps.

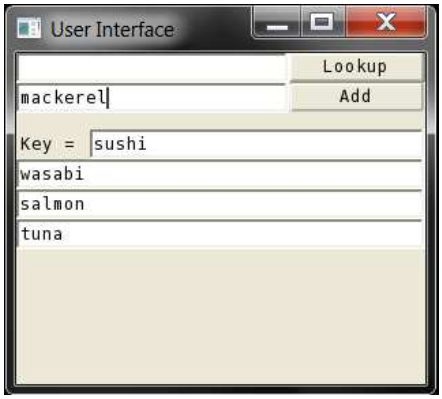


Figure 6.9: A screenshot of the Mind Map GUI.

display a dynamic number of *displayStr* widgets depending on how long the list is in the currently viewed key of the map.

The complete code can be seen in Figure 6.8 and a screenshot of it running can be seen in Figure 6.9.

6.5 Example: Asynchronous Computation

As we introduced in Section 6.1.4, UISF supports asynchronous operations, and here, we will build an example that makes use of the feature. Specifically, we will present a GUI for performing a complex and lengthy calculation, but when the calculation is requested, it will be performed asynchronously. Thus, the GUI will still continue to respond and behave normally.

The lengthy computation will be the calculation of potential meld in the card game Pinochle. In this game, players first get a hand of cards and then bid to receive a further 4 card "kitty". With the kitty added to the winner's hand, he plays his meld, which are specific combinations of cards. For instance, having one of each of the aces is worth 10 points, and having a King-Queen of the same suit is worth 2 points. Pinochle is played with a special deck of playing cards that has two of each card but only includes the cards from Nine to Ace in each suit.

The GUI will present a set of buttons for the user to enter his hand and will then calculate the average expected meld the user can expect if he wins the kitty. The calculation is performed asynchronously, and when it produces a result, the GUI displays it both plots it and displays it textually.

```
handSelector :: [Suit] → [Number] → UISF () Hand
handSelector [] _ = constA EmptyHand
handSelector (s : ss) ns = proc () → do
  bs ← leftRight $ slabel (show s) >>> concatA (map cardSelector ns) -< repeat ()
  hand ← handSelector ss ns -< ()
  returnA -< addToHand hand (map (s,) (concat $ zipWith replicate bs ns))
  where slabel str = setLayout (Fixed 75) (Fixed 30) $ label str
```

Figure 6.10: The compound widget for building a Pinochle hand.

```
pinochle :: UISF () ()
pinochle = proc () → do
  hand ← handSelector allSuits allNums -< ()
  event_update ← unique -< hand
  meld ← hold "" -< fmap calcMeld event_update
  leftRight $ label "Total meld = " >>> display -< meld
  b ← edge >>> button "Calculate meld from kitty" -< ()
  event_kitty ← (asyncUISF_E $ arr calcKitty) -< fmap (const hand) b
  let (meld_kitty, d) = case (event_kitty, b) of
        (Event (k, d), _) → (Event k, Event d)
        (_, Event _) → (Event "Calculating ...", Event NoHistogram)
        _ → (Nothing, Nothing)
  runDynamic display <<< hold [] -< meld_kitty
  histogram (makeLayout (Stretchy 10) (Fixed 15)) -< d
  returnA -< ()
```

Figure 6.11: The Pinochle GUI.

Before we can begin building this GUI, we must have some logic about Pinochle itself. Thus, we assume *Number*, *Suit*, and *Hand* data types that behave as expected as well as two functions for calculating a hand's meld and possible results from winning the kitty:

```
data Meld = String
calcMeld :: Hand → Meld
calcKitty :: Hand → ([Meld], HistogramData)
```

We will also make use of the following two custom UISF widgets:

```
cardSelector :: Number → UISF () Int
histogram :: Layout → UISF (Event HistogramData) ()
```

The *cardSelector* widget looks similar to a *button* widget but has some extra internal machinery to allow for selecting two of the same card. The *histogramWithSacle* widget displays a graphical histogram on screen.[4]

The first step in building this GUI is to provide an interface to allow the user to enter his hand. We do this by adding a *cardSelector* for each possible card. Because the Pinochle deck is totally static, we can achieve this with simple structural recursion. Thus, the *handSelector* compound widget is shown in Figure 6.10.

[4]Technically, the *histogram* widget is a built-in widget in UISF, but for brevity, we did not include a detailed discussion of it when we introduced the main UISF features earlier in this chapter, and we are using a simplified version of it here.

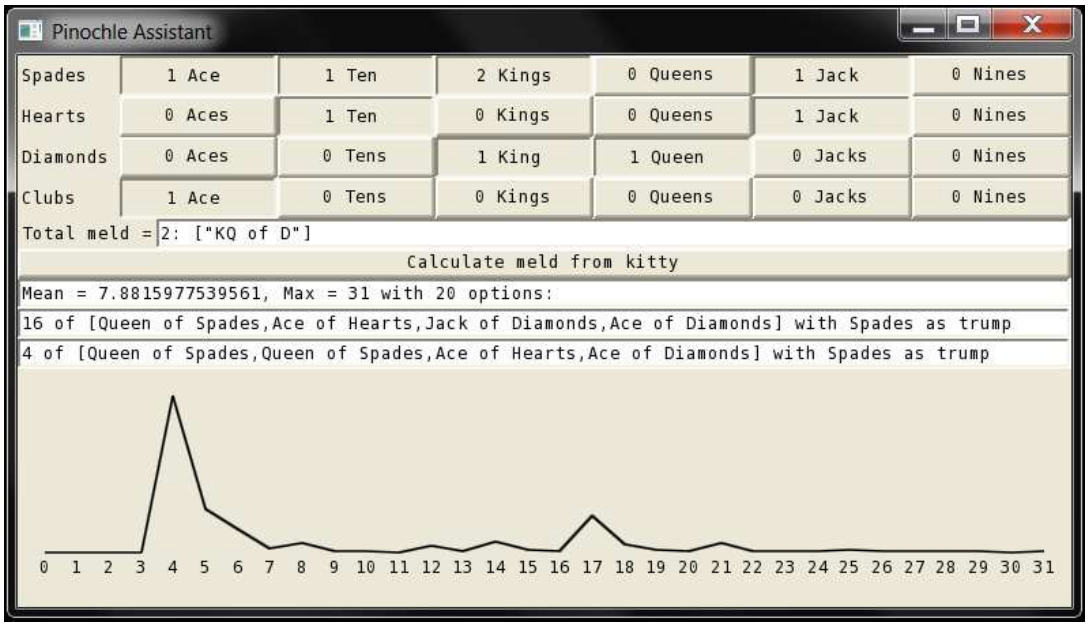


Figure 6.12: A screenshot of the Pinochle GUI.

In the Pinochle UISF itself, shown in Figure 6.11, we use the hand selector at the very beginning. We then display the meld just from the hand so far, and provide a button to calculate the potential meld from the kitty. When this button is pressed, an event is sent to the asynchronized *calcKitty* function, and the results are gathered, displayed, and plotted in the histogram. A screenshot of the program running can be seen in Figure 6.12.

6.6 Differences From Theory

Although UISF is inspired by CFRP and the theory contained in this report, it is different from that theory in a number of ways.

Resource Types

First and foremost, there are no resource types. This may seem like a critical omission considering that resource types are core to CFRP's safety guarantees, but it is a necessary evil. As mentioned in previous chapters, arrows with resource types are still not fully supported in GHC, and wormholes themselves cannot be implemented.

Of course, the biggest cost of omitting resource types is that we can no longer guarantee safe usage of arbitrary effects. One way to address this would be to remove effects altogether, but we find this to be too restrictive. Thus, we settle for allowing effects with a warning to the user to take care when using them.

The most common resources used in CFRP programs are actually those of wormholes. Mostly, this is because CFRP uses wormholes as its built-in concept of looping and state (recall from Section 4.4.1 that we remove the need for *loop* and *delay* in the presence of wormholes), but they are also central for any asynchronous operations.

To prevent UISF users from accidentally misusing wormholes (which becomes easy to do now that we have no resource types), we remove them entirely from the interface of the language. We revert to using the classic *loop* and *delay* operators for state, and we force all asynchronous communication to follow a few

specific patterns (i.e. it must be able to be expressed using one of the various "async" functions).

Removing wormholes is unfortunate for a few reasons. First, reverting to the classic *loop* and *delay* operators reintroduces the potential for unbounded looping, which is only detectable at runtime. Second, it restricts the forms that asynchrony can take (wormholes allow arbitrary communication channels between multiple processes, but the async functions enforce a sense of "parent" and "child" processes). Lastly, where once we had a single *letW* command for creating a wormhole, now we must have many different async functions. This bloating of the language is reminiscent of the many varieties of switch necessary before our concept of non-interfering choice. In some sense, though, UISF is better for not having wormholes. Although restrictive, the various async functions are optimized to run efficiently at their given tasks, and the specific options available may help users identify appropriate ways to write the programs they are trying to write.

Limitations

Another significant difference between UISF and CFRP has to do with the behavior of these async functions themselves. UISF is built atop the GLFW OpenGL library, and the current interface that it uses has little support for concurrent operations. Specifically, the GUI itself must be entirely single-threaded. That is, we can run multiple signal functions at multiple time rates, but all of the GUI behavior must be running in the same thread at the same rate. To prevent UISF from causing its GLFW back-end to throw errors, we restrict the async functions to fork *Automaton*s rather than other *UISF*s.

6.7 Conclusions and Discussion of Similar Libraries

UISF is a fully functioning, viable GUI library. The examples shown in this chapter are a sampling of what can be done with it, but perhaps an even better example is its integration with Euterpea. Within Euterpea, UISF itself has been extended to handle various sorts of MIDI input and output, and additional graphical widgets (a piano and guitar frets) are available for users.

Other GUI Libraries

GUI libraries come in many flavors and varieties. On one end of the spectrum, designs employ a callback structure, in which widgets register themselves as awaiting certain events, and when those events occur, the widgets are "called back." This design is typical of object-oriented languages, and there are far too many examples to cite.

In fact, many functional GUI libraries are simply built atop one of these imperative designs. For instance, FranTk Sage [2000], although built on top of the FRP system Fran Elliott and Hudak [1997], provides a fundamentally imperative design (with its *GUI* monad) for designing applications. Ignatoff et al. [2006] explore this interface between imperative GUI toolkits and functional languages by combining an object-oriented GUI toolkit into the FRP language FrTime in a principled way.

UISF shares many similarities with Fudgets [Carlsson and Hallgren, 1998], which uses stream processing as a central concept of design. Indeed, AFRP in general is clearly inspired by the Fudgets design. However, while AFRP is synchronous by default, Fudgets are instead asynchronous. UISF, built atop the AFRP framework, is obviously naturally synchronous, but it also has strong asynchronous support in the form of its async operators. Thus, we feel that it finds a good middle ground between these two approaches.

There are many other GUI libraries even in the category of FRP-based ones in Haskell [Apfelmus, 2012, Czaplicki, 2012, Giorgidze and Nilsson, 2008]. Grapefruit [Jeltsch, 2009] is a push-based FRP system that provides direct access to signals. Fruit [Courtney and Elliott, 2001b] introduced the first arrowized switch function and in general has a principled design to arrowized FRP that UISF models in many ways. Elm [Czaplicki and Chong, 2013] is an asynchronous FRP language for creating GUIs that uses both a

“traditional” and arrowized FRP design allowing the user to handle signals directly in basic cases or use signal functions for reactive or stateful computation.

Bibliography

H. Apfelmus. Reactive-banana, May 2012. URL `http://www.haskell.org/haskellwiki/Reactive-banana`.

A. Benveniste, B. Caillaud, and P. L. Guernic. From Synchrony to Asynchrony. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR 1999, pages 162–177, London, UK, 1999. Springer-Verlag.

G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, July 1984.

G. Berry and E. Sentovich. Multiclock Esterel. In *Correct Hardware Design and Verification Methods*, pages 110–125. Springer, 2001.

T. Brus, M. van Eekelen, M. van Leer, M. Plasmeijer, and H. Barendregt. CLEAN – A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer-Verlag, September 1987.

M. Carlsson and T. Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden, March 1998.

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188. ACM, January 1987.

D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, CA, 1984.

G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, March 2006.

A. Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, May 2004.

A. Courtney and C. Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001a.

A. Courtney and C. Elliott. Genuinely Functional User Interfaces. In *2001 Haskell Workshop*, September 2001b.

A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop*, Haskell '03, pages 7–18. ACM, August 2003.

E. Czaplicki. Elm: Concurrent FRP for functional GUIs, 2012.

E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, 2013.

O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM.

C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273. ACM, June 1997.

C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *21st Conference on Computer Graphics and Interactive Techniques*, pages 421–434. ACM, July 1994.

C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, September 2009. ACM.

T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer-Verlag, November 1987.

G. Giorgidze and H. Nilsson. Switched-on yampa. In *Proc. Practical Aspects of Declarative Languages*, pages 282–298. Springer Verlag LNCS, 2008.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

C. Hawblitzel. Linear types for aliased resources (extended version). Technical Report MSR-TR-2005-141, Microsoft Research, Redmond, WA, October 2005.

M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

P. Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, 2000.

P. Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.6), January 2014.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, August 2003.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

D. Ignatoff, G. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 259–276. Springer Berlin Heidelberg, 2006.

A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Sixth Workshop on Programming Languages meets Program Verification*, pages 49–60. ACM, January 2012.

W. Jeltsch. Signals, not generators! *Trends in Functional Programming*, 10:145–160, 2009.

W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. In *28th Conference on the Mathematical Foundations of Programming Semantics*, pages 215–228. Elsevier, June 2012.

M. P. Jones and P. Hudak. Implicit and explicit parallel programming in haskell. Technical report, Yale University, 1993.

S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages*, pages 295–308. ACM, 1996.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004.

E. Kiss. 7guis, 2014. URL `https://github.com/eugenkiss/7guis/wiki`.

N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, pages 257–266. IEEE Computer Society, 2011.

N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-Order Functional Reactive Programming in Bounded Space. In *39th Symposium on Principles of Programming Languages*, pages 45–58. ACM, January 2012.

J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, pages 24–35. ACM, June 1994.

P. Li and S. Zdancewic. A language-based approach to unifying events and threads. Technical report, University of Pennsylvania, 2006.

S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, January 2010.

H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *Journal of Functional Programming*, 21(4–5): 467–496, September 2011.

P. Liu and P. Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193(1):29–45, November 2007.

R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

R. Milner. The polyadic p-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series*, pages 203–246. Springer Berlin Heidelberg, 1993.

R. Milner. *Communicating and mobile systems: the pi calculus*. Cambridge University Press, 1999.

E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64. ACM Press, 2002.

C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

G. Patai. Efficient and compositional higher-order streams. In *Proceedings of the 19th International Conference on Functional and Constraint Logic Programming*, WFLP'10, pages 137–154, Berlin, Heidelberg, 2011. Springer-Verlag.

R. Paterson. A new notation for arrows. In *Sixth International Conference on Functional Programming*, pages 229–240. ACM, September 2001.

J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Practical Aspects of Declarative Languages*, pages 16–31. Springer, 2000.

S. Peyton Jones and P. Wadler. Imperative functional programming. In *20th Symposium on Principles of Programming Languages*. ACM, January 1993. 71–84.

R. Plasmeijer and M. van Eekelen. Clean – version 2.1 language report. Technical report, Department of Software Technology, University of Nijmegen, November 2002.

M. Pouzet. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

J. H. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 165–198. Springer Berlin Heidelberg, 1993.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

M. Sage. Frantk - a declarative gui language for haskell. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 106–117, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.

A. Sangiovanni-Vincentelli, M. Sgroi, and L. Lavagno. Formal Models for Communication-Based Design. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR 2000, pages 29–47. Springer Berlin Heidelberg, 2000.

W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 224–235. ACM, 2005.

J. A. Tov and R. Pucella. Practical affine types. In *38th Symposium on Principles of Programming Languages*, pages 447–458. ACM, January 2011.

R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2011.

P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. IFIP TC 2, April 1990.

P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 255–273. ACM, September 1991.

P. Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming languages*, pages 1–14. ACM, January 1992.

Appendix

A.1 Proof That Non-Interference Implies Commutativity (and Exchange)

Theorem 5 (Commutativity)**.**

$$right\ f \ggg left\ g = left\ g \ggg right\ f$$

Proof. This proof is straightforward. We will begin by assuming *Right* inputs only, and thus we can modify our assertion to:

$$arr\ Right \ggg right\ f \ggg left\ g = arr\ Right \ggg left\ g \ggg right\ f$$

Starting with the left hand side,

$$\begin{aligned}
& arr\ Right \ggg right\ f \ggg left\ g \\
= \quad & \{\ \text{Unit backwards}\ \} \\
& f \ggg arr\ Right \ggg left\ g \\
= \quad & \{\ \text{Non-Interference}\ \} \\
& f \ggg arr\ Right \\
= \quad & \{\ \text{Unit}\ \} \\
& arr\ Right \ggg right\ f \\
= \quad & \{\ \text{Non-Interference backwards}\ \} \\
& arr\ Right \ggg left\ g \ggg right\ f
\end{aligned}$$

For *Left* input values, the proof works in exactly the same way except that we must use non-interference's mirror:

$$arr\ Left \ggg right\ f = arr\ Left$$

which follows directly from non-interference and the definition of *right*. □

A.2 Choice-Based Implementations of First-Order Switch

Although using non-interfering choice and settability allows for a different paradigm for designing FRP programs, we can also use these tools to implement operators that are similar to the classic switchers. We show two such implementations in this appendix.

A.2.1 Standard Switch

The standard *switch* function can be implemented with non-interfering choice in a straightforward manner:

$$
\begin{aligned}
&\mathit{switch}_{\mathit{choice}} :: (\alpha \rightsquigarrow (\beta, \mathit{Event}\ \gamma)) \rightarrow ((\mathit{Event}\ \gamma, \alpha) \rightsquigarrow \beta) \\
&\qquad\qquad \rightarrow (\alpha \rightsquigarrow \beta) \\
&\mathit{switch}_{\mathit{choice}}\ \mathit{sf}_1\ \mathit{sf}_2 = \textbf{proc}\ a \rightarrow \textbf{do} \\
&\quad \mathit{onOne} \leftarrow \mathit{delay}\ \mathit{True} \prec \mathit{not}\ \mathit{onTwo} \\
&\quad (b, \mathit{et}) \leftarrow \textbf{if}\ \mathit{onOne}\ \textbf{then}\ \mathit{sf}_1 \prec a \\
&\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathit{returnA} \prec (\mathit{undefined}, \mathit{NoEvent}) \\
&\quad \textbf{let}\ \mathit{onTwo} = (\mathit{isEvent}\ \mathit{et})\ ||\ (\mathit{not}\ \mathit{onOne}) \\
&\quad \textbf{if}\ \mathit{onTwo}\ \textbf{then}\ \mathit{sf}_2 \prec (\mathit{et}, a) \\
&\qquad\qquad\ \textbf{else}\ \ \mathit{returnA} \prec b
\end{aligned}
$$

Here, we keep track of two internal state variables called *onOne* and *onTwo* that indicate whether we should be running the first or the second signal function. When the first produces an event, we set *onOne* to *False* so that we stop running it, and we set *onTwo* to *True*. Then, we pass the impulse generated from the first signal function to the second one, and for the future, the impulse stream contains only *NoEvent* values.

A.2.2 Parallel Switch

The *pChoice* function is somewhat more complicated and is shown in Figure A.1. *pChoice* takes a mapping of keys to signal functions (implemented here as a list for simplicity) as its static argument. For each element of this static list, we keep a dynamic list of states (the *states* variable in the figure). We check the input events for any that are keyed to the signal function we are currently processing and update the state list accordingly (by either adding or removing elements), and then we run the signal function for each state and recur. Note that the static signal functions are all impulse driven; thus, when new states are first added to the state list (which is done in the *update* helper function), they are given an impulse event, but otherwise, they are given *NoEvent* (i.e. in the definition of $\mathit{states}_{\mathit{new}}$). This restriction to strictly impulse driven signal functions is not fundamental – indeed, we could write a version of *pChoice* that accepts signal functions that also take a streaming input – but making it more generic would needlessly complicate this already dense definition.

It is also worth noting that there is a subtle difference in performance between *pChoice* and *pSwitch*. When the finite data type is large but rarely used, *pSwitch* may outperform *pChoice* because *pChoice* still has to iterate through its entire static list on each step while *pSwitch*'s dynamic list will be just the relevant signal functions. That said, their performance should be comparable when the finite data type is small compared to the number of currently running signal functions.

A.3 Proofs of Preservation and Progress for Synchronous Semantics

In order to prove preservation and progress for our semantics, we must show these properties for each of the transitions we have defined. Here we state and prove the relevant theorems.

A.3.1 Evaluation Transition

The evaluation transition is mostly lifted from a standard lazy semantics for $\mathscr{L}\{\rightarrow \times +\}$. The additions presented in Figure 4.4 simply explain that the new expressions are all values. Therefore, preservation and progress follow trivially.

```
pChoice :: Eq key ⇒ [(key, Event α ⇝ β)] →
                    ([(key, (UID, Event α))] ⇝ [β])
pChoice [ ] = constA [ ]
pChoice ((key, sf) : rst) = proc es → do
  rec states ← delay [ ] ≺ states_new
      let es_this = map snd $ filter ((== key) . fst) es
          states_inp = update states es_this
      output ← runDynamic (first (settable sf)) ≺ states_inp
      let states_new = map (λ ((_, s), uid) →
                              ((NoEvent, Event s), uid)) output
  rs ← pChoice rst ≺ es
  returnA ≺ (map (fst . fst) output) ++ rs
 where update :: [((Event α, Event State), UID)]
                 → [(UID, Event α)]
                 → [((Event α, Event State), UID)]
       update s [ ] = s
       update s ((uid, NoEvent) : rst) =
                    update (filter ((≠ uid) . snd) s) rst
       update s ((uid, i) : rst) =
                    update (((i, Event ), uid) : s) rst
```

Figure A.1: The implementation of *pChoice*.

A.3.2 Functional Transition

Preservation for the functional transition proceeds in a straightforward manner making sure that the streaming input is appropriately transitioned into a streaming output.

Theorem 6 (Preservation during functional transition)**.** *If* $e : \alpha \overset{R}{\rightsquigarrow} \beta$, $x : \alpha$, *and* $(\_, x, e) \Rrightarrow (\_, y, \_, \_)$, *then* $y : \beta$.

Proof. The proof of preservation proceeds by induction on the derivation of the transition judgment along with the knowledge of preservation for the evaluation transition. Each of the judgments can be proved trivially with a brief examination of the typing rules, so we omit the details. □

Progress for the functional transition is a somewhat more interesting concept. Because of the complexity of the transition, we are forced to make a few assumptions about the input data:

Theorem 7 (Progress during functional transition)**.** *If* $e : \alpha \overset{R}{\rightsquigarrow} \beta$, $x : \alpha$, *and* $\mathscr{V}$ *contains elements such that* $\forall r \in R, (r, a, \cdot) \in \mathscr{V}$ *where* $r : \langle \tau_{in}, \tau_{out} \rangle$ *and* $a : \tau_{in}$, *then* $\exists y : \beta, e' : \alpha \overset{R'}{\rightsquigarrow} \beta, \mathscr{V}', \mathscr{W}$ *such that* $(\mathscr{V}, x, e) \Rrightarrow (\mathscr{V}', y, e', \mathscr{W})$.

We require that in addition to the expression e being well-formed and the streaming argument x being of the right type, the set $\mathscr{V}$ must also be "well-formed". That is, for every resource that e might interact with (all resources in R), there is a triple in $\mathscr{V}$ corresponding to that resource that contains values of the appropriate types. Notably, they must all be resources that have not seen any interaction. This is not an unreasonable requirement as we proved in Theorem 3 that at any point during the functional execution, no resources see more than one interaction.

Proof. The proof of progress proceeds by induction on the derivation of the functional transition judgment. Based on the Canonical Forms Lemma (Lemma 1), we know that the functional transition need only apply to the five forms of a signal function, and we see by inspection that it does. We examine each judgment in turn:

- *SF constructor* (FT-ARR): When e is of the form $arr(e')$, typing rule TY-ARR tells us that $e' : \alpha \to \beta$. As $x : \alpha$, the streaming output $e\ x$ is of type β as necessary. The other outputs exist regardless of the form of e'.

- *SF partial application* (FT-FIRST): If e is of the form $first(e')$, then the typing rule TY-FIRST tells us that e' has resource type set R just as e does. Our inductive hypothesis tells us that outputs are available for our recursive transition. The streaming output (y,z) has the appropriate type, and the expression output, formed by applying *first* to the expression output of the recursive transition has the same type as e.

- *SF composition* (FT-COMP): e may be of the form $e_1 \ggg e_2$. By typing rule TY-COMP, we know that $e : \alpha \overset{R}{\rightsquigarrow} \gamma$, $e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta$, and $e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma$. The evaluation transitions progress, and by our inductive hypothesis, the functional transitions in the precondition progress as well. The output is formed from the results of the precondition with the streaming value z being of type γ as required. The expression output, made by composing the two expressions e_1'' and e_2'' has the same type as e.

- *SF choice* (FT-CHC1 and FT-CHC2): When e is of the form $e_1 \,|||\, e_2$, typing rule TY-CHC tells us that $e : \alpha + \beta \overset{R}{\rightsquigarrow} \gamma$, $e_1 : \alpha \overset{R_1}{\rightsquigarrow} \gamma$, and $e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma$. For either form of x, there is a judgment, and in either judgment the inductive hypothesis gives us output values where $y : \gamma$ as expected. The returned expression is also of the appropriate form considering that e_1'' from FT-CHC1 and e_2'' from FT-CHC2 have the same types as e_1 and e_2 respectively.

- *SF resource interaction* (FT-RSF): If e is of the form $rsf\ r$, then the typing rule TY-RSF tells us that its type must be $\alpha \overset{\{r\}}{\rightsquigarrow} \beta$ and $r : \langle \alpha, \beta \rangle$. By the conditions of our theorem, $\mathscr{V}$ must contain an element $(r, y, \cdot)$ such that $y : \beta$. Therefore, the streaming output y is of the right type. Lastly, the output expression is identical to the input expression.

- *Wormhole introduction* (FT-WH): We use typing rule TY-WH when e is of the form $\textbf{letW}\ r_w\ r_b\ e_i\ \textbf{in}\ e_{body}$; it tells us that e_{body} has type $\alpha \overset{R'}{\rightsquigarrow} \beta$ where $R = R' \setminus \{r_w, r_b\}$. Before using our inductive hypothesis, we must prove that the value set for the recursive call meets our requirements. We know that $(R \cup \{r_w, r_b\}) \supseteq R'$, so $\mathscr{V} \cup \{(r_w, e_i, \cdot), (r_b, (), \cdot)\}$ clearly satisfies the condition. Therefore, the streaming output y will be of type β. Furthermore, the output expression e'' must have the same type as e_{body} which satisfies our output requirement. □

A.3.3 Temporal Transition

By the definition of the overall operational semantics (Definition 2), we know that the trace of any program P is infinite. As long as we can prove progress, preservation is irrelevant. We make use of the preservation and progress theorems for the evaluation and functional transitions shown earlier to prove the following:

Theorem 8 (Progress of overall semantics)**.** *If P is a program with type $\alpha \overset{R}{\rightsquigarrow} \beta$ and $R \subseteq \mathscr{R}_o$ then the trace of P will always be able to progress via the temporal transition $\overset{t}{\mapsto}$ when starting from $(\mathscr{R}_o, \emptyset, P)$.*

Proof. The judgment for the temporal transition allows the input to progress so long as the preconditions are met. The first condition defines $\mathscr{V}_{in}$ to contain elements for each resource in $\mathscr{R}$ as well as for each whitehole and blackhole pair in $\mathscr{W}$. This is used in the second condition, which will progress only if we can prove that $(\mathscr{V}_{in}, (), P)$ will progress through the functional transition. P may access resources in R as well as any virtual resources introduced through wormholes. In the base case, the functional transition has never been run, and R does not contain any virtual resources. Then, because $R \subseteq \mathscr{R}_o$, $\mathscr{V}_{in}$ contains elements for every resource in R, so we meet the conditions of the functional progress theorem (Theorem 7). In the inductive case, we are dealing with a potentially further evaluated program P' with resources R', which may contain virtual resources. Then, all virtual resources will have been generated from previous passes through the functional transition, and all of the virtual resources will be represented by $\mathscr{W}$. Once again, $\mathscr{V}_{in}$ will contain elements for each resource in R', and the functional transition can progress.

The last two preconditions are simply definitions of $\mathscr{R}'$ and $\mathscr{W}'$ such that R' contains the same number of elements keyed by the same resource names as $\mathscr{R}$ and that $\mathscr{W}'$ contains the same whitehole and blackhole resource names as $\mathscr{W}$ as well as any new wormhole data entries from $\mathscr{W}_{new}$.

The output program P' is not the same as P. Notably, its type may have changed to $() \overset{R'}{\rightsquigarrow} ()$. From Theorem 2, we know that R' is the set R with up to two new virtual resources for each element of $\mathscr{W}_{new}$ corresponding to the whiteholes and blackholes of the elements of $\mathscr{W}_{new}$. This is fine for exactly the reason that these new resources are "documented" in $\mathscr{W}_{new}$ and $\mathscr{W}_{new}$ is unioned with $\mathscr{W}$ for the output of the transition. Therefore, when $\mathscr{V}$ is being generated in the next iteration, all of the resources of R' will be represented, both the original ones in $\mathscr{R}$ and any virtual ones created and documented in $\mathscr{W}$.

Finally, we must consider the overall base case. On the first iteration through the temporal transition, there can be no virtual resources because no wormhole expressions have been executed by the functional transition yet. Therefore, the initial wormhole set $\mathscr{W}$ can be the empty set. □

A.4 CFRP Properties

In order to express the ideas of resource safety and commutativity, we first need a way to discuss a given process's execution at a given moment in time.

In order to do this, we need a bit more access to the executive transition than we have. Specifically, we define the following slightly modified executive transition: $\Downarrow_p$. The behavior of $\Downarrow_p$ is identical to that of $\Downarrow$ except that when the transition internally invokes the functional transition $\Rightarrow$ on a process with process ID p, it must do so in a restricted form such that the FT-TIME judgment is not permitted. Furthermore, we use $\Downarrow_p^*$ to refer to the reflexive transitive closure over this transition .

This modified executive transition allows us to rigorously define the term "moment in time".

Definition 7 (Moment in Time)**.** *We say $(S, \mathscr{R}, \mathscr{W}) \hookrightarrow_p (S', \mathscr{R}', \mathscr{W}')$ represents the sequence of unique program states $S = S_0, S_1 \ldots S_n = S'$ if and only if $\exists T, T'$ such that $(T, \mathscr{R}, \mathscr{W}) \Downarrow_p^* (T', \mathscr{R}', \mathscr{W}')$ where $T(p) = S_0, T'(p) = S_n$, and for all intermediate T, $\exists i$ such that $T(p) = S_i$. We call this sequence of states part of the same **moment in time**.*

Essentially, the idea of a moment in time fits with the fundamental FRP abstraction, where we assume that the program executes infinitely fast. Thus, one "moment" is the sequence of steps on one process that occurs between any abstract passage of time.

Lemma 3. *The program states representing one moment in time cannot cause an update to a resource.*

This lemma is trivially provable due to the fact that resources can only be updated in the FT-TIME judgment, which is definitionally restricted from being one of the states of a moment in time. Furthermore,

it provides us with the knowledge that any process that is terminated while it is mid-execution will not affect any resources.

We will go on to show that no resource can be interacted with more than once in a given moment and that any data observed in a given moment will be the same regardless of the process's structure or what other processes are running asynchronously.

A.4.1 Resource Safety

In order to state that CFRP interacts with resources in a safe and predictable manner, we first must define what it means to interact with a resource.

Definition 8 (Resource interaction)**.** *Every program state S of the form $(K \triangleright (rsf\ r, x, U))$ for any control stack K, value x, and update set U is a* ***resource interaction*** *of resource r.*

With this, we can state the following trivial lemma regarding resource interaction over a sequence of states:

Lemma 4. *A sequence of program states $S_0 \ldots S_n$ will interact with a resource r exactly j times where j is the number of states in the sequence $S_0 \ldots S_{n-1}$ that interact with resource r.*

Together, we can use this definition and lemma to define resource safety:

Theorem 9 (Resource Safety)**.** *For a program $P :: \alpha \overset{R}{\rightsquigarrow} \beta$, we know:*

- *No program states will ever interact with a resource $r \notin R$.*
- *No two processes in P can interact with the same resource.*
- *No moment of time in P will ever interact with a resource more than once.*

This theorem has three components. The first statement asserts that a well typed program will not interact with resources not noted in its type. The next statement asserts that even with the asynchrony of multiple processes running simultaneously, resource access remains unique across the entire program. The last statement asserts that for any given process, within one moment in time, no resource will be accessed more than once.

Proof. The first two statements of this theorem follow from the typing rules. First, if P is well formed and has resource types R, then there can be no *rsf* construct in P for a resource $r \notin R$. the second follows naturally from the typing rules for fork and composition.

To prove the third statement we show that for all states of $(S, \mathscr{R}, \mathscr{W}) \hookrightarrow_p (S', \mathscr{R}', \mathscr{W}')$, no two can interact with the same resource. Let S_k be a state in the sequence that interacts with resource r. Then, $S_k = (K \triangleright (rsf\ r, \_, U_k))$, and S_{k+1} must be $(K \triangleleft (rsf\ r, \_, U_{k+1}))$. The only way to move from a *return* state like this to an *evaluation* state is through either the FT-TIME judgment or the FT-COMP$_2$ judgment. No state can move through the FT-TIME judgment by definition of a *moment in time*, and the FT-COMP$_2$ judgment will not allow code that has already been executed to run again, which we can be sure of due to Theorem 4 (Structural Preservation). Therefore, no state $S_{j>k}$ can repeat the same state as S_k.

Furthermore, due to the typing rules and the fact that every state S_i unwinds to a well typed expression e and that a moment contains no FT-TIME judgments that would allow the expression to begin again, there can be no more than one *rsf* command for r in this moment. Therefore, no resource can be interacted with more than once in this sequence. □

A.4.2 Resource Commutativity

The resource safety theorem tells us that a process will never perceive more than one resource interaction with the same resource in a given moment in time. We use this to make the following claim:

Theorem 10 (Commutativity)**.** *For any S and r, if $(S, \mathscr{R}, \mathscr{W}) \hookrightarrow_p (S', \mathscr{R}', \mathscr{W}')$ is the set of states $S_0 \ldots S_n$ and there exists $i < n$ such that $S_i = (K \rhd (rsf\ r, \_, U_i))$ and $S_{i+1} = (K \lhd (rsf\ r, x, U_{i+1}))$, then x will be the same for all S regardless of i.*

This theorem states that within a given moment in time for a given initial $\mathscr{R}$ and $\mathscr{W}$, regardless of where a resource is read or what comes before or after it within that moment, it will produce the same value. Thus, if two sequences of code both use the same resources and can execute in the same moment in time, they can be substituted and the values produced by their resources will not alter because of that change.

Thus, we state that the order of execution of the components of a signal function running at the same moment in time does not change the result of the program. This fits the model of our abstraction exceptionally well because it implies that we can really think of a moment in time as happening all at once—no one component needs to happen before another to produce the result.

Proof. We will prove this theorem by proving that it holds for any resource type.

First, if r is a blackhole, the theorem holds trivially. The FT-RSF$_w$ judgment applies, and due to the definition of `read` for blackholes and regardless of anything else, the value x will be $()$.

If r is a whitehole resource, then the transition from S_i to S_{i+1} must once again be FT-RSF$_w$, in which case the value x is determined uniquely by the element w of the wormhole's internal resource data, and it suffices to show that regardless of S and i, w will be the same. Resource data can only be changed by a resource update (`update`), and the only functional transition judgment that updates resources is FT-TIME. Furthermore, the FT-TIME judgment will only update if an element $(r, \_)$ is in the update data passed into it. However, $(r, \_)$ will only be in U for sets of update data in a process that has already processed the FT-RSF$_w$ judgment with the resource in question. By Theorem 9 (Resource Safety), we know that no other process can interact with r, so that element can only be in U for states in $S_0 \ldots S_n$. Lastly, because no states $S_0 \ldots S_n$ use the FT-TIME transition, we know that w cannot be changed during the moment.

Lastly, if r is a physical resource, then the transition from S_i to S_{i+1} is FT-RSF$_r$, and the value x is determined uniquely by the state of the resource r. This conclusion follows similarly to that for whiteholes. □