Abstract

Modeling User Interfaces in a Functional Language

Antony Alexander Courtney

2004

It is widely recognized that programs with Graphical User Interfaces (GUIs) are difficult to design and implement. One possible reason for this difficulty is the lack of any clear formal basis for GUI programming. GUI toolkit libraries are typically described only informally, in terms of implementation artifacts such as objects, imperative state and I/O systems.

In this thesis, we develop Fruit, a Functional Reactive User Interface Toolkit. Fruit is based on Yampa, an adaptation of Functional Reactive Programming (FRP) to the Arrows computational framework. Yampa has a clear, simple formal semantics based on a synchronous dataflow model of computation. GUIs in Fruit are defined compositionally using only the Yampa model and formally tractable mouse, keyboard and picture types. Fruit and Yampa have been implemented as libraries for Haskell, a purely functional programming language.

This thesis presents the semantics and implementations of Yampa and Fruit, and shows how they can be used to write concise executable specifications of common GUI programming idioms and complete GUI programs.

Modeling User Interfaces in a Functional Language

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Antony Alexander Courtney

Dissertation Director: Professor Paul Hudak

May 2004

Contents

Acknowledgments **ix**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Dissertation Overview 4

I Foundations 5

2 Yampa: A Synchronous Dataflow Language Embedded in Haskell **6**

2.1 Concepts 6

2.2 Composing Signal Functions 7

2.2.1 Lifted Functions 8

2.2.2 Serial Composition 9

2.2.3 Aggregation and Widening 9

2.3 Synchronization and Concurrency 10

2.3.1 Synchrony and Algebraic Laws 11

2.4 Composing Larger Dataflow Graphs 12

2.5 Events and Event Sources 14

2.6 Loops and Delays 15

2.7 History-Sensitive Signal Functions 17

2.8 Arrow Syntax 18

2.9 Switching 20

2.10 Examples 21

2.10.1 An Edge Detector 21

2.10.2 The *hold* Signal Function 23

2.10.3 A Bounded Counter 25

2.11 Animating Signal Functions 27

2.12 Chapter Summary 29

3 The Evolution of Yampa **30**

3.1 Origins: Fran 32

3.1.1 Switching in Fran 33

3.1.2 The User Type 34

3.1.3 Time and Space Leaks in Fran 34

3.2 From Fran to FRP 37

3.3 Understanding SOE FRP's Conceptual Model 39

3.3.1 Limits in SOE FRP's Expressive Power 41

3.4 Extending SOE FRP with runningIn 42

3.4.1 Difficulties with runningIn 43

3.5 Behaviors that "escape" 44

3.6 Chapter Summary 46

4 The Formal Semantics of Yampa **48**

4.1 Direct Denotational Semantics 49

4.2 Operational Semantics 54

4.3 Derived Combinators 59

4.4 Chapter Summary 61

5 Implementing Yampa **62**

5.1 Synchronized Stream Processors 62

5.2 Continuation-Based Implementation 64

5.3 Implementing Primitives 65

5.4 Encoding Variability 67

5.5 Simple Dynamic Optimizations 69

5.6 Chapter Summary 71

6 Haven: Functional Vector Graphics **72**

6.1 Introduction 72

6.1.1 A Functional Model of Vector Graphics 73

6.2 Basic Concepts 75

6.2.1 Points and Colors 75

6.2.2 Regions and Region Algebra 76

6.2.3 Images 77

6.3 Geometry 78

6.3.1 Paths 78

6.3.2 Rectangles 80

6.3.3 Paths, Regions and Bounding Rectangles 81

6.3.4 Shapes 82

6.3.5 Transforms 83

6.4 Rendering Model 84

6.4.1 Composition Operators 84

6.4.2 Cropping Images 85

6.4.3 Pens and Stroking 86

6.4.4 Fonts and Text 87

6.5 Layout 89
6.6 From Images to Pictures: A Rendering Monad 91
6.7 Examples 93
6.7.1 Sierpinski Gasket 93
6.7.2 A Logo for Haven 95
6.8 Chapter Summary 95

7 Fruit: A Functional GUI Library 97

7.1 Defining GUIs 97
7.1.1 What is a GUI? 99
7.1.2 Library GUIs 99
7.2 The GUIInput Type 101
7.3 Basic Layout Combinators 102
7.3.1 Transforming GUIs 103
7.4 Specifying Layout Using Arrows 106
7.5 Chapter Summary 109

8 Dynamic Collections in Yampa 110

8.1 The Need for Dynamic Collections 110
8.2 Parallel Composition and Local State 111
8.3 A First Attempt 113
8.4 Continuation-Based Switching 115
8.5 Parallel Switching and Signal Collections 117
8.6 Dynamic Interfaces Example 119
8.7 Chapter Summary 121

II Applications **122**

9 Proving Properties of Yampa Applications **123**

9.1 Preliminaries and Notation 124

9.1.1 Observing Signal Functions 124

9.1.2 The “always” Quantifier 125

9.2 An Invariance Theorem 126

9.3 Example: A Simple Bounded Counter 130

9.3.1 Implementation 130

9.3.2 A Rudimentary Proof 131

9.4 Chapter Summary 138

10 The Model / View / Controller (MVC) Design Pattern in Fruit **139**

10.1 “Shallow” MVC: Multiple Camera Views 140

10.1.1 Passive Views 141

10.1.2 Multiple Active Views 143

10.2 Model/View/Controller In Fruit 144

10.2.1 GUI Component Refactoring 145

10.3 Chapter Summary 148

11 Application Case Studies **149**

11.1 A Media Controller 149

11.2 A Web Browser with Interactive History 152

11.2.1 Basic Components 154

11.2.2 A History-less Browser 155

11.2.3 Modeling Interactive History 156

11.3 An Interactive Video Game 159

11.3.1 Game Play 159
11.3.2 Game Objects 161
11.3.3 The Game Proper 164
11.4 Evaluation 179
11.5 Chapter Summary 183

III Conclusions and Future Work 184

12 Related Work, Conclusions and Future Work 185

12.1 Related Work 185
12.1.1 Data Flow Languages 185
12.1.2 Imperative GUI Toolkits 187
12.1.3 Functional GUI Toolkits 188
12.1.4 Constraints 189
12.1.5 Formal Models of GUIs 190
12.2 Conclusions 191
12.3 Future Work 193
12.3.1 Incremental Implementation 193
12.3.2 Integration with Standard Widget Sets 194
12.3.3 Modeling Systems Software 196
12.3.4 Model-Based Interface Design 196

Bibliography 196

List of Figures

2.1 A Signal Function, **SF** $a\ b$ 7

2.2 Core Arrow Primitives 8

2.3 Other Arrow Operators 12

2.4 Arrow loop operator 15

2.5 Semantics of hold 18

2.6 Implementation of edge 22

2.7 Implementation of $dHold$ 24

2.8 Implementation of $boundedCounter$ 27

3.1 Alternatives for Semantics of Behaviors 35

4.1 Core Primitives of Yampa 48

4.2 Operational Semantics for Core Yampa 54

4.3 Flawed Semantics for Switching 59

4.4 Standard Yampa Utility Routines 60

6.1 Cropping a Monochrome Image to a Path 85

6.2 Stroking a Path with a Pen 88

6.3 The Sierpinski Gasket 94

6.4 A Logo for Haven 96

7.1 ballGUI Specification 98

7.2 Using besideGUI 102

8.1 Mozilla Thunderbird Search Interface 112

10.1 Implementation of a view 141

10.2 Multiple Views 142

10.3 A Shared Model with Local Editing 147

11.1 Basic Media Controller 150

11.2 Media Controller Finite State Machine 150

11.3 Media Controller Implementation 150

11.4 A Simple Web Browser 153

11.5 Screen-shot of Space Invaders 159

11.6 Dynamic collection of game objects maintained by dpSwitch. 165

11.7 Runtime Heap in Java/Swing Implementation 180

Acknowledgments

My work was funded primarily by a National Science Foundation Graduate Research Fellowship.

I am grateful to my advisor, Paul Hudak, for his constant support and good advice, and for giving me considerable latitude to pursue new ideas and research directions.

Conal Elliott, Paul Hudak, John Peterson, Henrik Nilsson and Zhanyong Wan developed the underlying ideas and early implementations of Fran and Functional Reactive Programming, and participated in lengthy discussions about the design and semantics of Yampa.

Great credit is due to Magnus Carlsson and Thomas Hallgren for their seminal work on Fudgets, the first genuinely functional user interface library I am aware of. Their work was a continual source of inspiration and ideas for the work presented here.

The members of my thesis committee, Paul Hudak, Conal Elliott, John Peterson and Zhong Shao, provided essential guidance and valuable feedback on my talks and papers.

Ross Paterson, Magnus Carlsson, Thomas Hallgren, Conal Elliott and anonymous reviewers provided extremely detailed comments and suggestions on the published papers on which much of this work is based.

I owe a special debt of gratitude to Henrik Nilsson, my direct collaborator on Yampa. His thorough, methodical work style set an example I continue to aspire to, and all of my work was greatly improved by his input.

I am also extremely grateful to Valery Trifonov for teaching me formal semantics, and for always having time to give helpful, insightful suggestions in response to my many questions on a wide range of technical matters.

On a personal note, thanks to my friends Killian, Scott, John, Michael, James, Wolfgang, Luciana, Stephanie, Rob, Bettina, Rachel, Henrik, Arvind, Valery, Chris, Stefan, Carsten and Molly for their unfailing friendship, being there for me when I really needed it, and ensuring I had a wonderful time while in graduate school. To John, for believing in me and offering generous contributions of hardware on which all of this work was developed. To my mother for her love, support and encouragement. To Sally Ann for her companionship in the early years. To Sandro, for teaching me to program and showing me why computers matter. And to my rock climbing partners Scott, Jacques, Lara, John, Luciana and Stephanie, for being such wonderful friends and providing an excellent and necessary diversion from the demands of graduate school.

Chapter 1

Introduction

1.1 Background and Motivation

It is widely recognized that programs with Graphical User Interfaces (GUIs) are difficult to design and implement [14, 52, 48]. Myers [52] enumerated several reasons why this is the case, addressing both high-level software engineering issues (such as the need for prototyping and iterative design) and low-level programming problems (such as concurrency). While many of these issues are clearly endemic to GUI development, the subjective experiences of many practitioners is that even with the help of state-of-the-art toolkits, GUI programming still seems extremely complicated and difficult relative to many other programming tasks.

Historically, many difficult programming problems became easier to address once the theoretical foundations of the problem were understood. To cite just one example, precedence-sensitive parsers became much easier to implement after the development of context-free grammars and the Backus Naur Formalism [57]. In contrast, while some formal models of GUIs have been proposed [17, 30, 16], these models have been largely divorced from the world of practical GUI toolkits. To see this, we need only ask the question "what is a GUI?" in the context of any modern GUI toolkit. In all toolkits that we are aware of, the answer is either entirely informal, or depends on artifacts of the toolkit implementation, such as objects, imperative state, non-deterministic concurrency or I/O systems, each of which has an extremely difficult and complicated formal semantics in and of itself [1, 15, 32, 66].

This situation lead us to pose the following questions:

- While a formal account of GUIs based on objects, imperative programming, and I/O systems is clearly *sufficient*, are such concepts *necessary*?
- Is there a simpler formal model of GUIs that is still powerful enough to account for GUIs in general?

To answer these questions, we have developed *Fruit* (a Functional Reactive User Interface Toolkit) based on a new formal model of GUIs. Fruit's foundational model (called *Yampa*) is based on two simple concepts: *signals*, which are functions from real-valued time to values, and *signal functions*, which are functions from signals to signals. GUIs are defined compositionally using only the Yampa model and simple *mouse*, *keyboard* and *picture* types.

While there are many possible formal models of GUIs, the Fruit model is compelling for a number of reasons:

- The concepts of *signal* and *signal function* in the Fruit model have direct analogues in digital circuit design and signal processing. This allows us to borrow ideas from these established domains, and also resonates with our own experience, in which programmers speak of "wiring up" a GUI to describe writing event handlers.

- Fruit specifications are *extremely concise*. Small interactive GUIs can be written with one or two lines of code, with minimal attention to the kind of boiler-plate that plagues modern GUI toolkits.

- The model is *purely declarative*. While declarative programming languages have failed to gain mainstream acceptance, the growing popularity of XML [19] and interactive, direct-manipulation authoring tools [69] has spawned something of a resurgence of interest in declarative models of interactive systems. But if we look at two of the more popular exemplar systems in each of these areas (the XML User-Interface Language XUL [54] in the first case, and Macromedia's Flash [45] in the latter), we find that these systems require an imperative programming language (JavaScript and ActionScript, respectively) to describe even fairly elementary interactive behaviors that aren't accounted for in the declarative modeling part of the system. A consequence of this arrangement is that automated tools can not easily trace or visualize how an event handler written in the imperative language for one part of the application affects other parts, and tools are forced to address the "round trip" problem [75] – keeping the textual imperative code synchronized with the declarative model understood by the tool.

- The Fruit model enables a clear account of the connection between the GUI and the non-GUI aspects of the application, and allows a clear separation of these aspects using the Model/View/Controller design pattern [42]. The design and implementation of the MVC pattern in Fruit is explored in detail in chapter 10.

- The Fruit model makes *data flow explicit*. As we will discuss in detail in chapter 9, capturing the pattern of data flow relationships explicitly is fundamentally useful when reasoning about implementations of graphical interfaces.

1.2 Dissertation Overview

This dissertation is organized into three parts: *Foundations*, *Applications* and *Conclusions*.

Part I, Foundations, presents the three libraries developed as part of this dissertation: *Yampa* – for programming reactive systems in a synchronous dataflow style, *Haven* – for creating 2D vector graphics images, and *Fruit* – for specifying interactive graphical user interfaces using Haven and Yampa. Chapters 2-5 cover the evolution, semantics and implementation of Yampa. Chapter 6 presents Haven and chapter 7 presents the Fruit GUI toolkit (based on Haven and Yampa). Chapter 8 describes the challenge of accounting for dynamic user interfaces in a dataflow programming model, and present our extensions to Yampa to address this issue.

Part II of this thesis explores applications of Yampa, Haven and Fruit. Chapter 9 demonstrates how we can verify properties of Yampa programs using equational reasoning and the formal semantics developed in chapter 4. Chapter 10 explores using Fruit to give a precise account of the Model / View / Controller design pattern in Fruit. Chapter 11 presents a couple of larger example applications written in Fruit, and contrasts the Fruit approach to user interface development with traditional imperative toolkits.

Part III summarizes related work, presents our conclusions, and outlines a number of ideas for future work based on the results presented here.

Part I

Foundations

Chapter 2

Yampa: A Synchronous Dataflow Language Embedded in Haskell

Yampa is a a language embedded in Haskell for describing reactive systems. Yampa is based on ideas from Fran [23, 21] and FRP [78]. This chapter gives a brief, informal introduction to Yampa; later chapters present the formal details.

2.1 Concepts

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to a value:

$$Signal\,\alpha \;=\; Time \rightarrow \alpha$$

$Time$ is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, if $Point$ is the type of a 2-dimensional point, then the time-varying mouse position might be represented with a value of type $Signal\ Point$.

Figure 2.1: A Signal Function, **SF** a b

A *signal function* is a function from *Signal* to *Signal*:

$$SF\,\alpha\,\beta \;=\; Signal\,\alpha \rightarrow Signal\,\beta$$

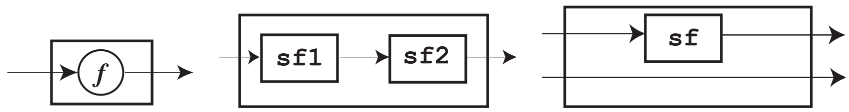
When a value of type $SF\,\alpha\,\beta$ is applied to an input signal of type $Signal\,\alpha$, it produces an output signal of type $Signal\,\beta$.

We can think of signals and signal functions using a simple analog or digital circuit analogy. Line segments (or "wires") represent signals, with arrowheads indicating the direction of flow. Boxes (or "components") represent signal functions, with one signal flowing into the box's input port and another signal flowing out of the box's output port, as shown in figure 2.1.

In order to ensure that signal functions are executable, we require them to be *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. All primitive signal functions in Yampa are causal and all combinators for composing signal functions preserve causality.

2.2 Composing Signal Functions

Programming in Yampa consists of defining *signal functions* that map an *input signal* of some type to an *output signal* of some type. In the Yampa implementation, *signal functions* are provided as values of type $SF\ \ a\ \ b$, but *signals* are never exposed directly to the programmer. Instead, the implementation provides the programmer with a small set of *primitive* signal functions, as well as a set of combinators



(a) *arr f* (b) *sf1* $\ggg$ *sf2* (c) *first sf*

Figure 2.2: Core Arrow Primitives

for composing signal functions into larger aggregate signal functions.

The decision to only expose signal functions (and not signals) to the programmer was based on experience implementing Fran [20] and FRP [35]. This experience taught us that allowing signals as first class values leads to "space-time leaks" [20] in the implementation. In contrast, by defining SF as an abstract type constructor, and only providing a fixed set of primitives and combinators for defining signal functions, we can show by structural induction that our implementation is free of space-time leaks. A full account of the issues and alternatives related to this design choice is given in the next chapter.

Yampa's signal functions are an instance of the arrows framework proposed by Hughes [39], and hence all of the arrow combinators may be used to define signal functions. The core arrow combinators are shown in figure 2.2. These core primitives all have simple, precise definitions in terms of the conceptual model presented in section 2.1. An informal description of these operators is given below; the precise semantics will be covered in chapter 4.

2.2.1 Lifted Functions

The most basic kind of primitive signal function is a *lifted* function. Lifted functions are constructed with the standard arrow operator arr, which lifts a static function from a to b to the level of a signal function, mapping a $Signal\ \ a$ to $Signal\ b$:

$$arr :: (a \to b) \to SF\ a\ b$$

Visually, lifting of some function f is depicted by drawing an ellipse around f, as shown in figure 2.2(a).

For any function f, $arr\ f$ denotes *pointwise* application of f. That is, at every time t, the output signal of $arr\ f$ is f applied to the input signal at t:

$$arr\ f = \lambda s \to \lambda t \to f\ (s\ t)$$

This definition of arr gives a concise account of the denotation of arr in terms of the conceptual model of section 2.1. However, it is important to note that this definition is for reference purposes only. This definition does not reflect the actual implementation (described in chapter 5), and because SF is an *abstract* type constructor, the programmer may not actually write such definitions directly.

2.2.2 Serial Composition

As shown in figure 2.2(b), two signal functions may be connected in series using the serial composition operator $\ggg$. Denotationally, serial composition is just reverse function composition:

$$\begin{aligned}
&(\ggg) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c \\
&sf1 \ggg sf2 = \lambda s \to \lambda t \to (sf2\ (sf1\ s))\ t \\
& = sf2 \circ sf1
\end{aligned}$$

That is, the overall output of $sf1 \ggg sf2$ at time t is determined by connecting the overall input signal to $sf1$, connecting $sf1$'s output signal to the input signal of $sf2$, and using $sf2$'s output as the overall output signal.

2.2.3 Aggregation and Widening

In Yampa, *tuples* are used to express aggregation of signals. In the simplest case, a signal carrying a *pair* of values is used to represent a *pair* of distinct signals. The simplest operator that acts on such aggregate signals is the *first* operator:

$$\mathit{first} :: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)$$

As illustrated in figure figure 2.2(c), given some signal function $sf :: SF\ a\ b$, $\mathit{first}\ sf$ is a signal function in which the input and output signal types of the argument sf have been "widened" to accomodate an extra signal line (of type $Signal\ \ c$). As illustrated in the diagram, *first* just passes the value of this extra signal line unmodified from input to output. Formally, we can define *first* as:

$$\mathit{first}\ sf = \lambda s \rightarrow pairZ\ (sf\ (fstZ\ s))\ (sndZ\ s)$$

where the following auxiliary functions are simple liftings of standard Haskell functions for manipulating pairs to the level of signal functions:

$$\begin{aligned}
&pairZ :: Signal\ a \rightarrow Signal\ b \rightarrow Signal\ (a, b)\\
&pairZ\ sa\ sb = \lambda t \rightarrow (sa\ t, sb\ t)\\
&fstZ :: Signal\ (a, b) \rightarrow Signal\ a\\
&fstZ = arr\ fst\\
&sndZ :: Signal\ (a, b) \rightarrow Signal\ b\\
&sndZ = arr\ snd
\end{aligned}$$

2.3 Synchronization and Concurrency

A crucial aspect of all functional reactive programming systems (including Yampa) is that they use a *synchronous* execution model. In general, the value of the output signal from a signal function at time t is determined precisely by the signal function's input signal on the interval $[0, t]$. This is a *closed* interval, and hence the output at time t may depend on the value of the input signal at t. That is, there is no observable "propagation delay" in Yampa[1]. While we often appeal to digital circuitry as a useful analogy for thinking about the data flow style of

[1]There are certain exceptions to this rule. In particular, certain primitive signal functions are provided by Yampa which deliberately introduce an infinitesimal delay between their input and output in order to ensure that feedback loops are well-formed. We defer a full discussion of delays and feedback loops to section 2.6.

programming, the absence of any observable propagation delay in Yampa is a marked difference between Yampa's programming model and the classical model of digital hardware circuits[2]. In the digital circuit model, every circuit element (including wires) are considered to have some arbitrary non-zero propagation delay, and explicit clocking must be introduced to provide synchronization. In contrast, Yampa presents an idealized model in which signal propagation takes zero logical time, synchronization is implicit, and delays are introduced explicitly as needed. Yampa uses a synchronous execution model because the resulting deterministic model of concurrency yields programs that are vastly easier to reason about than non-deterministic (i.e. multi-threaded) ones [7].

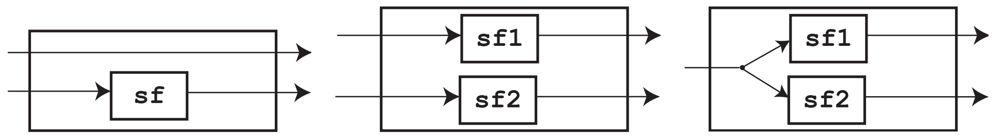
2.3.1 Synchrony and Algebraic Laws

A corollary of Yampa's synchronous execution model is that we can derive a number of useful algebraic laws from the denotational semantics of Yampa. For example:

$$\forall\, f :: a \rightarrow b,\ g :: b \rightarrow c\ .\ arr\ (g \circ f) \equiv arr\ f \ggg arr\ g$$

The above property states that the lifted form of two composed functions (using Haskell's standard function composition operator $\circ$) is equivalent to serial composition (using Yampa's $\ggg$ operator) of the lifting of each individual function. This is a true *identity* in the sense that it works for any functions f and g, and can be applied either right-to-left or left-to-right. Although the implementations of Yampa on real computers will always be discrete-time approximations of the

[2] An excellent introduction to digital logic design, including a full account of the classical model of combinational logic circuits and the implications for synchronous and asynchronous logic design, is given by Mano [46].



(a) *second sf* (b) *sf1 *** sf2* (c) *sf1 &&& sf2*

Figure 2.3: Other Arrow Operators

underlying continuous-time denotational semantics [78], any implementation of Yampa must ensure that non-divergent terms that are equivalent according to the denotational semantics are observationally equivalent in the implementation.

Yampa's synchronous execution model is crucial to enabling such algebraic identities. If either lifting (using arr) or serial composition (using $\ggg$) introduced any observable delay between input and output, the above identity would not hold, as serial composition of two lifted signal functions would have an observably different propagation delay than a single lifting. Providing an idealized execution model in which signal propagation takes zero logical time enables the programmer to focus on the logical relationships between input and output signals, without worrying about how the internal wiring structure of a particular signal function might affect the observable behavior.

2.4 Composing Larger Dataflow Graphs

In addition to the core arrow primitives shown in figure 2.2, there are a number of other arrow operators, some of which are shown in figure 2.3. These other operators enable the programmer to form arbitrary directed data flow graphs from signal functions. The operators in figure 2.3 have the following types:

$$\begin{aligned}
&second :: SF\ b\ c \rightarrow SF\ (a,b)\ (a,c) \\
&(***) \quad :: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a,c)\ (b,d) \\
&(\&\&\&) :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b,c)
\end{aligned}$$

The operator $second$ is just the dual of the primitive arrow operator $first$ (figure 2.2(c)): given a pair of signal lines, $second$ feeds the second signal line through its argument while leaving the first signal line untouched.

The operators $***$ and $\&\&\&$ provide two forms of parallel composition of signal functions. The $***$ operator provides true parallel composition in the sense that the input and output signals from the pair of signal functions being composed are independent. The $\&\&\&$ operator provides a kind of "broadcast" or "split" operation, since the same input signal is fed to both of the signal function arguments.

All of the operators of figure 2.3 are *derived*, in the sense that they can be expressed using only the core arrow primitives of figure 2.2. The implementations of each operator in terms of the core primitives are as follows:

$$
\begin{aligned}
&second\ f = arr\ swap \ggg first\ f \ggg arr\ swap\\
&f *** g = first\ f \ggg second\ g\\
&f \&\&\& g = arr\ dup \ggg (f *** g)\\
&swap\ (x, y) = (y, x)\\
&dup\ x = (x, x)
\end{aligned}
$$

At first glance, it might appear that the above definitions for $***$ and $\&\&\&$ are not true parallel composition, since in the definition of $f *** g$, the first input signal is fed to f "before" the second signal is fed to g. However, recall that a key aspect of Yampa's synchronization model (section 2.3) is that the primitive operators $first$ and $\ggg$ do not introduce any observable delays. Hence, the observable output of the $f *** g$ is exactly as if f and g executed in parallel. Becuase of this flexibility, one could just as easily define $***$ with:

$$f *** g = second\ g \ggg first\ f$$

and the observable behavior would be indistinguishable from the previous definition.

2.5 Events and Event Sources

While some aspects of reactive programs (such as the mouse position) are naturally modeled as continuous signals, other aspects (such as the mouse button being pressed) are more naturally modeled as *discrete events*. Conceptually, events are conditions that "occur instantaneously" – that is, they have no duration. To model discrete events, we introduce the $Event$ type, isomorphic to Haskell's $Maybe$ type:

$$\begin{aligned}\textbf{data}\ Event\ a &= EvOcc\ a\\ &\mid NoEvent\end{aligned}$$

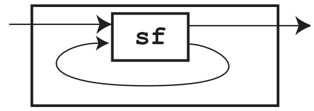
A signal function whose output signal carries values of type $Event\ \ T$ for some type T is called an *event source*. The $Event\ \ a$ type represents a *possible* event occurrence. At any time t, sampling a signal of $Event\ a$ either yields the value $EvOcc\ a$ (indicating an event occurence), or $NoEvent$ (indicating that no event occured).

Event Tagging and Mapping Individual event occurrences may carry a value, which is the purpose of the type parameter a in type $Event\ \ a$. Two operations are provided for manipulating these values: *tagging* and *mapping*, which allow the programmer to *enrich* each occurrence with extra information about the occurence that may be of interest to the observer of the event source: [3]

$$\begin{aligned}&tag :: Event\ a \rightarrow b \rightarrow Event\ b\\ &fmap :: Event\ a \rightarrow (a \rightarrow b) \rightarrow Event\ b\end{aligned}$$

For example, a mouse button press event has, by default, type $Event\ ()$, indicating that it carries no information other than the fact of its occurrence. However, we could tag each button press event occurrence with the mouse's current position, by applying tag point-wise (using arr) to the event source.

[3]The reason for the name *fmap* is because $Event$ is an instance of the Haskell's *Functor* type class.



(a) $loop\ sf$

Figure 2.4: Arrow loop operator

Event *mapping* (with $fmap$) applies a static function f (of type $(a \rightarrow b)$) to an $(Event\ a)$ to obtain an $(Event\ b)$. As with event tagging, mapping is almost always done point-wise, by using $fmap$ in conjunction with arr.

Event tagging is really just a special case of mapping that ignores the value carried in the original occurrence. Tagging is thus easily defined as:

$$tag\ e\ v = fmap\ e\ (const\ v)$$

where $const\ v$ is the standard Haskell *constant function*.

Event Merging Finally, possible event occurrences can be *merged* with $merge$:

$$\begin{aligned}
&merge :: Event\ a \rightarrow Event\ a \rightarrow Event\ a \\
&merge\ (EvOcc\ v1)\ e2 = (EvOcc\ v1) \\
&merge\ NoOcc\ e2 = e2
\end{aligned}$$

The definition of $merge$ specifies that an occurrence is favored over a non-occurrence, and the first argument is favored if both are occurrences. Like tag and $fmap$, $merge$ is typically applied point-wise.

2.6 Loops and Delays

In earlier sections, we have appealed to a digital circuit analogy for thinking about Yampa programs. In digital circuits, *feedback* is used to define *stateful* circuits, i.e. circuits whose reaction to an input stimulus at time t depends not just on the input at t, but on some or all of the previous history of the input signal.

Yampa provides a *loop* operator (illustrated in figure 2.4(a)) which enables the definition of signal functions that accumulate state in a manner analogous to that of a digital circuit. As in a digital circuit, Yampa's loop operator works by *feedback*: some portion of a signal function's output is made available as part of the input signal. Yampa's $loop$ operator has the following type:

$$loop :: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b$$

The argument to $loop$ is a signal function operating on a pair of signals. The $loop$ operator arranges for the second half of the signal function's output signal to be made available as the second component of the pair of input signals. We defer a precise definition of the loop operator to the formal semantics (chapter 4). Suffice it to say that the $loop$ operator, like the other Yampa primitives, does not introduce any delays; the fed back portion of the input signal at time t is exactly the output signal at t.

The definition of $loop$ thus enables the programmer to write ill-formed signal functions such as:

$$bad :: SF\ Int\ Int$$
$$bad = loop\ (arr\ (\lambda(x, y) \rightarrow (y + 1, y))$$

The above definition is ill-formed because at any time t, computing a sample value to use for the fed-back input sample (y) requires evaluating the expression ($y + 1$), which in turn requires evaluating the expression y itself. Since such a definition has no least fixed point, the definition is thus a "black hole", and the above Yampa program will diverge at runtime.

Returning to the digital circuit analogy, circuits involving feedback are well-formed precisely because of the presence of propagation delay. The fact that every circuit element (including wire!) has some finite propagation delay ensures that the fed-back input at a time t will in fact be the output signal at some time $t - \epsilon$. It is the *absence* of any observable delay in Yampa which causes definitions such as bad to diverge. To allow the programmer to write useful, well-formed definitions involving feedback, Yampa provides a primitive operator, $iPre$, that allows the programmer to introduce infinitesimal delays in to Yampa programs in a deliberate, controlled manner. The $iPre$ ("initialized previous element") primitive has the following signature:

$$iPre :: a \rightarrow SF\ a\ a$$

At time $t = 0$, sampling the output signal of $iPre\ x$ will yield the value x. At times $t > 0$, sampling the output of $iPre\ x$ will yield the input signal at time $t - \epsilon$ (for some implementation-defined, unspecified ϵ). The previous definition of bad could thus be rewritten as:

$$notBad :: SF\ Int\ Int$$
$$notBad = loop\ (arr\ (\lambda(x, y) \rightarrow (y + 1, y) \ggg second\ (iPre\ 1))$$

In this version, the use of $iPre$ ensures that the feedback loop is well formed. Initially (at time $t = 0$), the use of iPre ensures that the value 1 is used as the fedback value y, and hence the output at time $t = 0$ is 2. At subsequent sample times, the output at time t will just be the value of y at time $t - \epsilon$, which will just be 1. Thus the above definition is observationally equivalent to the signal function $constant\ 2$. While this is clearly a trivial example, in the following sections loops and delays will be used in conjunction with event sources to construct more interesting signal functions that can not be expressed without feedback.

2.7 History-Sensitive Signal Functions

Thus far, all of the core primitive signal functions we have presented are *instantaneous*: the value of the output signal at time t depends only on the value of the input signal at t. However, the Yampa model also provides primitive signal functions that are *history-sensitive*. Such primitives produce an output signal that may depend not just on the input signal at t, but at all times on $[0, t]$.

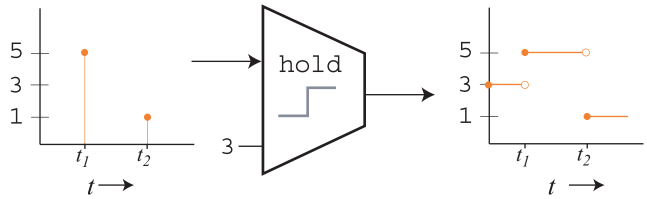


Figure 2.5: Semantics of hold

One of the most basic and useful such primitives is the *hold* primitive:

$$hold :: a \rightarrow SF\ (Event\ a)\ a$$

The *hold* primitive provides a continuous view of a discrete event source by "latching" (or *hold*ing) the value of the last event occurrence across a period of non-occurrences, as illustrated in figure 2.5. The graph on the left is a trace of an $Event\ Int$ signal over time. We use a vertical line terminated by a circle to indicate event occurrences; the height of the line indicates the value carried with the occurrence. The graph on the right is a trace of the output signal of $(hold\ 3)$ applied to the given event source. The signal starts out as 3 (the initial argument to $hold$). At time t_1, the input signal has an occurrence with value 5. The output signal becomes 5 at t_1, and remains 5 until the input signal's next occurrence at t_2, and so on.

2.8 Arrow Syntax

One benefit of using the arrow framework in Yampa is that it allows us to use Paterson's arrow notation [61]. Paterson's syntax (currently implemented as a preprocessor for Haskell) effectively allows signals to be named, despite signals not being first class values. This eliminates a substantial amount of plumbing resulting in much more legible code. In this syntax, an expression denoting a signal function has the form:

$$
\begin{array}{l}
\textbf{proc}\ pat \rightarrow \textbf{do} \\
\quad pat_1 \leftarrow sfexp_1 \prec exp_1 \\
\quad pat_2 \leftarrow sfexp_2 \prec exp_2 \\
\quad \ldots \\
\quad pat_n \leftarrow sfexp_n \prec exp_n \\
\quad returnA \prec exp
\end{array}
$$

This is just *syntactic sugar*: The arrow notation can be translated into plain Haskell that makes use of the arrow combinators.

The keyword **proc** is analogous to the λ in λ-expressions, pat and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, exp and exp_i are expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the **proc**-expression is bound by pat, and its output signal is defined by the expression exp. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions ($sfexp_i$). An optional keyword **rec**, applied to a group of definitions, permits signal variables to occur in expressions that textually precede the definition of the variable, allowing recursive definitions (feedback loops). Finally,

$$\textbf{let}\ pat = exp$$

is shorthand for

$$pat \leftarrow identity \prec exp$$

where $identity$ is the identity signal function ($arr\ id$), allowing binding of instantaneous values in a straightforward way.

The syntactic sugar is implemented by a preprocessor which expands out the definitions using only the basic arrow combinators arr, $\ggg$, first, and, if **rec** is used, $loop$.

For a concrete example, consider the following:

$$
\begin{array}{l}
\mathit{sf} = \mathbf{proc}\ (a, b) \rightarrow \mathbf{do} \\
\quad \mathit{c1} \leftarrow \mathit{sf1} \prec a \\
\quad \mathit{c2} \leftarrow \mathit{sf2} \prec b \\
\quad c \leftarrow \mathit{sf3} \prec (\mathit{c1}, \mathit{c2}) \\
\quad \mathbf{rec} \\
\qquad d \leftarrow \mathit{sf4} \prec (b, c, d) \\
\quad \mathit{returnA} \prec (d, c)
\end{array}
$$

Here we have bound the resulting signal function to the variable sf, allowing it to be referred by name. Note the use of the tuple pattern for splitting sf's input into two "named signals", a and b. Also note the use of tuple expressions for pairing signals, for example for feeding the pair of signals $\mathit{c1}$ and $\mathit{c2}$ to the signal function $\mathit{sf3}$.

2.9 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is switch:

$$
\begin{array}{l}
\mathit{switch} :: \\
\quad \mathit{SF}\ a\ (b, \mathit{Event}\ c) \rightarrow (c \rightarrow \mathit{SF}\ a\ b) \rightarrow \mathit{SF}\ a\ b
\end{array}
$$

switch switches from one subordinate signal function into another when a switching event occurs. The first argument to switch is an *initial embedded signal function*. This embedded signal function produces a pair of output signals. The first output signal defines the overall output of the switch while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, $switch$ applies its second argument to the value carried in the event occurrence and switches into the resulting signal function.

Yampa also includes *parallel* switching constructs that maintain dynamic collections of signal functions connected in parallel. Signal functions can be added to or removed from such a collection at runtime in response to events. The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible synchronous dataflow language. Yampa's parallel switching constructs are described in detail in chapter 8.

2.10 Examples

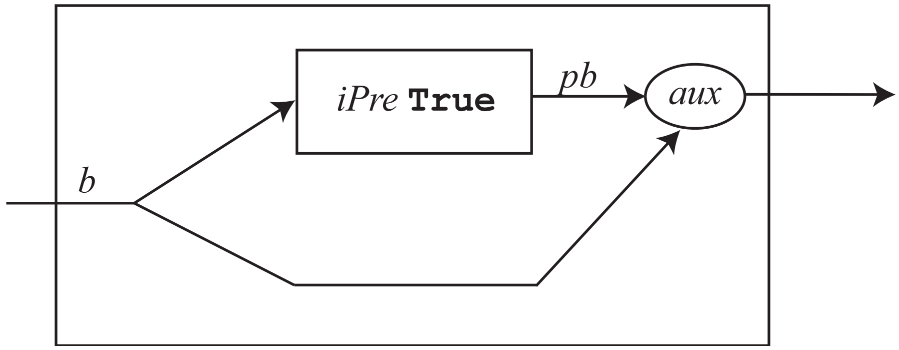
In this section, we present a few basic examples that demonstrate programming with Yampa. We first present direct implementations of a couple of the utility signal functions included in the standard Yampa utilities library. We then develop a simple "bounded counter" signal function that demonstrates the basic design and implementation techniques used in developing real applications.

The purpose of these examples is to provide a rudimentary introduction to writing simple Yampa programs. In later chapters (particularly in Part II), a number of full-scale Yampa applications will be presented in detail.

2.10.1 An Edge Detector

It is often useful to be able to detect so-called "predicate" events, i.e. events that are described by some boolean expression. For example, we might wish to detect when the mouse position crosses from the left half of the screen to right half of the screen. Yampa provides a standard utility signal function $edge$ (a "rising edge"

$edge =$



where
$\quad aux\ (False, True) = EvOcc\ ()$
$\quad aux\ \_ \qquad\qquad\quad\;\, = NoEvent$

Figure 2.6: Implementation of edge

detector) for this purpose:

$$edge :: SF\ \ Bool\ (Event\ ())$$

The $edge$ signal function takes a (continuous) signal of boolean values as input and produces a discrete event on output. An occurence of the output event indicates that $edge$ has detected a transition from $False$ to $True$ (a rising edge) on its input signal.

The data flow graph for the implementation of $edge$ is shown in figure 2.6. The boolean input signal is fed to $iPre$ to produce a delayed version of b (labeled pb). At any time t, the value of pb is just the value of b at time $t - \epsilon$. The argument to $iPre$ specifies the value of pb at time $t = 0$. Since $edge$ is a *rising* edge detector (i.e. sensitive to transitions from $False$ to $True$), the value $True$ is used as the initial value. the input signal happens to have the value

The concrete implementation of $edge$ (written using arrows syntax) is:

```
  -- rising edge detector:
edge :: SF Bool (Event ())
edge = proc b → do
```

$$
\begin{array}{l}
pb \leftarrow iPre\ True \prec b \\
returnA \prec \mathbf{if}\ ((pb, b) \equiv (False, True)) \\
\qquad\qquad\quad \mathbf{then}\ EvOcc\ () \\
\qquad\qquad\quad \mathbf{else}\ NoEvent
\end{array}
$$

The above is a fairly straightforward translation of the data flow graph of figure 2.6. The only notable difference is that the body of the lifted function aux has been written as an inline expression fed as the input to $returnA$ on the last line.

Using just the arrows primitives (without the arrows syntax), this example could be written as:

$$
\begin{array}{l}
edge :: SF\ Bool\ (Event\ ()) \\
edge = (iPre\ True \mathbin{\&\&\&} arr\ id) \ggg arr\ aux \\
\quad \mathbf{where} \\
\qquad aux\ (False, True) = EvOcc\ () \\
\qquad aux\ \_ \qquad\qquad\quad\ = NoEvent
\end{array}
$$

Or, equivalently:

$$
\begin{array}{l}
edge = arr\ dup \ggg first\ (iPre\ True) \ggg arr\ aux \\
\quad \mathbf{where} \\
\qquad aux\ (False, True) = EvOcc\ () \\
\qquad aux\ \_ \qquad\qquad\quad\ = NoEvent
\end{array}
$$

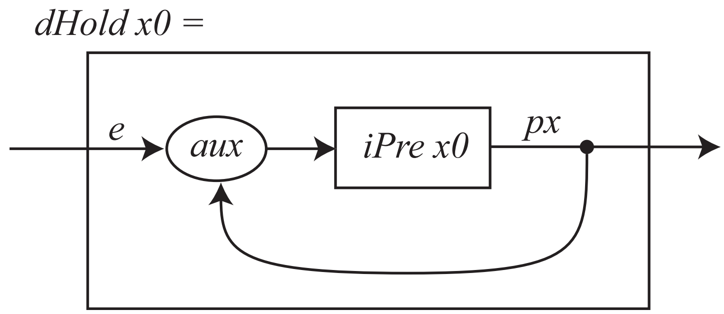
The latter version is just an in-line expansion of the definition of the definition of the $\&\&\&$ operator in the first version.

2.10.2 The *hold* Signal Function

In section 2.7 we described $hold$, a history-sensitive signal function provided in the Yampa standard utilities library.

The data flow graph for the implementation of a basic variant of hold, called $dHold$ (for "delayed" hold) is shown in figure 2.7. The corresponding code follows directly from the diagram:

$$
\begin{array}{l}
dHold :: a \rightarrow SF\ (Event\ a)\ a \\
dHold\ x0 = \mathbf{proc}\ e \rightarrow \mathbf{do} \\
\mathbf{rec}\ px \leftarrow iPre\ x0 \prec event\ px\ id\ e
\end{array}
$$



where
$aux\ e\ px = event\ px\ id\ e$

Figure 2.7: Implementation of $dHold$

$returnA \prec px$

The function $event$, used in the first line of the body of $dHold$, is just the Event analog of Haskell's standard $maybe$ function:

$event :: b \rightarrow (a \rightarrow b) \rightarrow Event\ a \rightarrow b$
$event\ \_\ f\ (EvOcc\ x) = f\ x$
$event\ x\ \_\ NoEvent\quad = x$

As with the $edge$ implementation in the previous section, the body of the auxiliary function has been expanded inline in the definition of $dHold$ for concision. The **rec** form of the arrows syntactic sugar is used to allow the point-wise sample px to be used as both the input and output of $iPre$.

This $dHold$ variant of $hold$ is so named because if an event occurs on the input signal at time t, this will only be latched to the output signal of $dHold$ at time t^+. Introducting an infinitesimal delay here is often useful, as it means that the output of $dHold$ can be used directly in some larger feedback loop without the need to introduce another explicit delay.

The non-delayed version of $hold$ simply defines x as the output signal instead of px:

-- non-delayed version:

$$
\begin{array}{l}
\mathit{hold} :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ a)\ a \\
\mathit{hold}\ x0 = \mathbf{proc}\ e \rightarrow \mathbf{do} \\
\mathbf{rec}\ px \leftarrow \mathit{iPre}\ x0 \prec x \\
\quad \mathbf{let}\ x = \mathit{event}\ px\ \mathit{id}\ e \\
\mathit{returnA} \prec x
\end{array}
$$

Of course, given this definition, dHold could also be implemented simply as:

$$\mathit{dHold}\ x0 = \mathit{hold}\ x0 \ggg \mathit{iPre}\ x0$$

2.10.3 A Bounded Counter

A useful circuit in the realm of digital electronics is a *counter*, which increments a number (stored in some register) in response to some external event (such as a button being pressed).

We can, of course, also easily develop a basic counter as a Signal Function in Yampa. The basic counter makes use of accum, another standard Yampa utility signal function:

$$\mathit{accum} :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ (a \rightarrow a))\ (\mathit{Event}\ a)$$

The accum signal function is similar to hold in the sense that it accumulates a value internally that is updated in response to input events. However, rather than merely latching the value carried with the event, accum expects each event occurence to carry a *function* that is *applied* to the current internal value to produce the next value. On the output side, accum produces an output event signal that is synchronized with the input event signal, but whose value is a snapshot of accum's internal state. If a continuous output signal is needed on the output side, accum can simply be composed with hold. This idiom is so common, in fact, that Yampa provides these as standard utilities:

$$
\begin{array}{l}
\mathit{accumHold} :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ (a \rightarrow a))\ a \\
\mathit{accumHold}\ x0 = \mathit{accum}\ x0 \ggg \mathit{hold}\ x0 \\
\mathit{dAccumHold} :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ (a \rightarrow a))\ a \\
\mathit{dAccumHold}\ x0 = \mathit{accum}\ x0 \ggg \mathit{dHold}\ x0
\end{array}
$$

A basic counter, then, can be implemented simply and directly as follows[4]:

$$\begin{aligned}&\mathit{counter} :: \mathit{Int} \rightarrow \mathit{SF}\ (\mathit{Event}\ ())\ \mathit{Int}\\&\mathit{counter}\ \mathit{x0} = \mathit{arr}\ (\text{`}\mathit{tag}\text{`}\mathit{incr}) \ggg \mathit{accumHold}\ \mathit{x0}\\&\quad \mathbf{where}\ \mathit{incr}\ x = x + 1\end{aligned}$$

Note in the above, that incr is a *function* that is used as a value to *tag* every input event occurence. The definition of incr is intended to make this explicit; However, the definition of $\mathit{counter}$ could be written slightly more concisely by replacing the use of incr with $(+1)$, making further use of Haskell's section notation.

To make this example slightly more interesting, we will develop a *bounded* counter, which will only increment if the counter's current value is less than or equal to some fixed maximum value. We will again appeal to a simple standard utility function, gate:

$$\begin{aligned}&\mathit{gate} :: \mathit{Event}\ a \rightarrow \mathit{Bool} \rightarrow \mathit{Event}\ a\\&\_\ \text{`}\mathit{gate}\text{`}\ \mathit{False} = \mathit{NoEvent}\\&e\ \text{`}\mathit{gate}\text{`}\ \mathit{True} = e\end{aligned}$$

Like the other utility functions operating on event signals fmap and tag, gate is intended to be applied pointwise. However, gate is intended to be applied pointwise to both an event signal and a boolean signal. When used in this way, gate acts as a kind of filter that will filter out all events that occur at times when the corresponding boolean signal is False.

Implementing the bounded counter is thus a simple matter of feeding back the output of the counter, comparing the counter's current value with the given maximum value, and using this pointwise computation to gate the external input event. The data flow graph for this example is shown in figure 2.8. The corresponding concrete syntax (using arrows syntactic sugar) is as follows:

$$\mathit{boundedCounter} :: \mathit{Int} \rightarrow \mathit{SF}\ (\mathit{Event}\ ())\ \mathit{Int}$$

[4]The use of $(\text{`}\mathit{tag}\text{`}\mathit{incr})$ instead of just $\mathit{tag}\ \mathit{incr}$ is because tag is usually used as an infix operator with the Event argument first. This argument order is almost always more convenient and readable in larger examples.

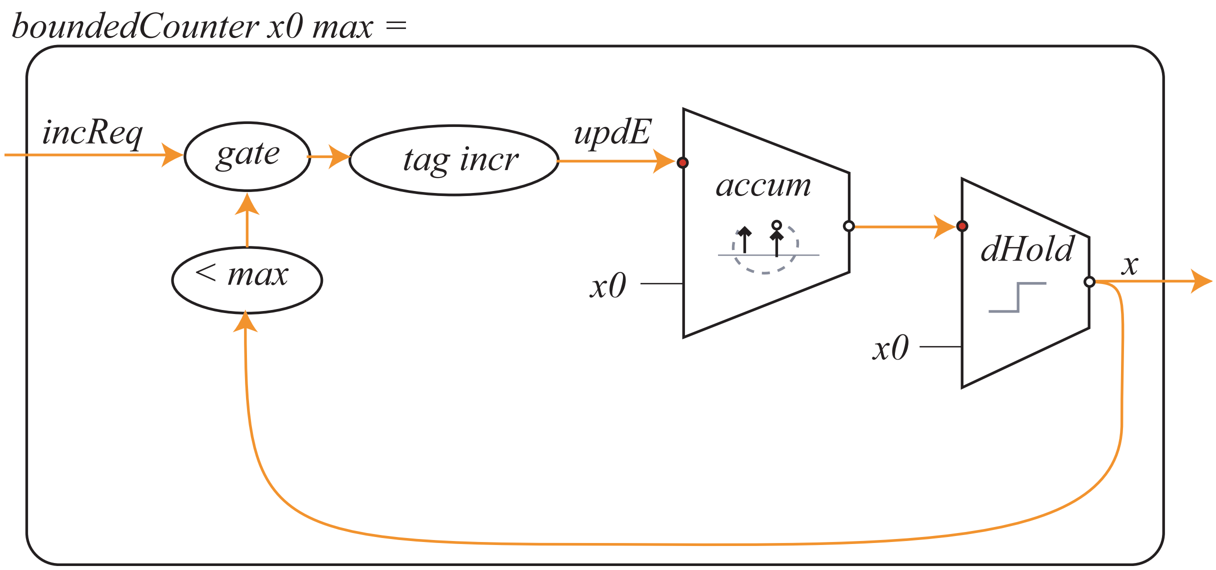


Figure 2.8: Implementation of *boundedCounter*

$$
\begin{array}{l}
\mathit{boundedCounter}\ \mathit{max} = \mathbf{proc}\ \mathit{incReq} \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ n \leftarrow \mathit{dAccumHold}\ 0 \prec \mathit{incCount} \\
\qquad \mathbf{let}\ \mathit{incCount} = (\mathit{incReq}\ `\mathit{gate}`\ (n < \mathit{max}))\ `\mathit{tag}`\ (+1) \\
\quad \mathit{returnA} \prec n
\end{array}
$$

Note in the above that we are using *dAccumHold* rather than *accumHold* to ensure that the feedback is well-formed. Of course, this has the consequence of introducing an infinitesimal delay between input event occurences and an observable change in the output signal.

2.11 Animating Signal Functions

Thus far we have seen a number of simple reactive programs realized as signal functions. One notable omission from these specifications was any explicit mention of the I/O system or a connection to the external world. This is quite deliberate, and is a hallmark of Yampa programming.

Instead of specifying an interactive application as an explicit sequence of I/O actions, the Yampa programmer defines a signal function that transforms an input signal of some type into an output signal of some type. This programming style ensures that Yampa programs have a simple, precise denotation independent of the (typically complex and underspecified) details of the I/O system or the external world.

To actually execute a Yampa program we need some way to connect the program's input and output signals to the external world. Yampa provides the function $reactimate$ (here slightly simplified) for this purpose [5]:

$$
\begin{array}{ll}
reactimate :: IO\ (DTime, a) & \text{-- sense} \\
\quad \rightarrow (b \rightarrow IO\ ()) & \text{-- actuate} \\
\quad \rightarrow SF\ a\ b & \\
\quad \rightarrow IO\ () &
\end{array}
$$

The first argument to reactimate ($sense$) is an IO action that will obtain the next input sample along with the amount of time elapsed (or "delta" time, $DTime$) since the previous sample. The second argument ($actuate$) is a function that, given an output sample, produces an IO action that will process the output sample in some way. The third argument is the signal function to be animated. Reactimate approximates the continuous-time model presented here by performing discrete sampling of the signal function, invoking $sense$ at the start and $actuate$ at the end of each time step. In the context of our video game, we use $sense$ and $actuate$ functions which read an input event from the window system and render an image on the screen, respectively.

[5] Yampa also provides a slightly more complicated version of this function, $reactimateEx$, which allows for more efficient interfacing with the operating system (by allowing $sense$ to perform blocking I/O), and enables $actuate$ to indicate that the application should terminate.

2.12 Chapter Summary

This chapter has presented a brief introduction to *Yampa*, a library for implementing reactive systems in Haskell based on a synchronous data-flow model of programming. Yampa is based on an adaptation of ideas from Fran and FRP to the arrows computational framework proposed by Hughes. We presented a few Yampa primitives and the arrow combinators, described some important aspects of the Yampa conceptual model (such as synchrony and tupling of signals), and examined a number of small examples. We also briefly presented the arrows syntactic sugar, and showed how this notation could be used to directly transliterate data flow diagrams into concrete textual syntax.

Chapter 3

The Evolution of Yampa

"Functional Reactive Programming" refers to a general conceptual framework for programming reactive systems. Although they vary in many details, all functional reactive programming systems share the following essential properties:

purely functional - reactive programs communicate with subprograms and the external world only via explicit functional interfaces. There is no implicit global "world" or "heap" available to functions, and functions only compute values; they may not have any hidden side-effects or perform I/O actions.

first-class time-varying values - a central polymorphic type (called a *signal* in Yampa, a *behavior* in Fran and FRP) defines the complete output of a reactive system over time.

declarative reactivity - a reactive program's response to external stimuli are expressed by *switching* from one time-varying value to another in response to input *events*.

There have been many concrete manifestations of functional reactive programming as libraries or stand-alone languages. The first of these systems, which introduced the essential ideas of the framework, was *Fran* [23, 21], implemented as a library for Haskell. Fran offered a simple, powerful programming model for specifying reactive programs. Unfortunately, this expressive power came at a substantial price in terms of performance: it was very easy to write programs that suffered from serious performance problems at run time (described shortly), and it proved extremely difficult to provide an implementation of Fran in Haskell that did not suffer from such problems.

A subsequent system, referred to as *FRP* or sometimes "SOE FRP"[1] was designed to alleviate some of the performance issues of the original Fran implementation. The key difference between FRP and Fran was that FRP redefined Fran's core "Behavior" type. While the FRP design appeared to ameliorate some of Fran's performance issues, it turned out to be fundamentally limited in expressive power: a number of programs that could easily be expressed in Fran could simply not be expressed in FRP. Once this became clear, an attempt was made to extend the basic FRP design to address this issue. Unfortunately, there were some practical difficulties in the design and implementation of this extension (to be described shortly).

Yampa was developed in order to explore an alternative approach to FRP for embedding functional reactive programming in Haskell, based on the benefit of experience implementing Fran and FRP. In the concrete implementations of Fran and FRP, a great deal of effort was devoted to syntactic matters. For example, the implementations of Fran and FRP contain tens or hundreds of one line type and function definitions that are simple liftings from the world of static values to the world of time-varying values. In contrast, Yampa almost completely ignores all

[1] A reference to the fact that the implementation of this system was first presented in the book "The Haskell School of Expression" by Paul Hudak.

matters of surface syntax, and simply tries to provide a basic set of combinators that are equivalent in expressive power to (extended) FRP. The end result is that Yampa is defined by a very small set of primitives with clear, precise semantics, and also clarifies the distinction between *signals* and *signal functions*. Unfortunately, this semantic simplicity comes at a price: the arrows syntactic sugar is needed to write Yampa programs of even modest size.

This chapter presents the historical background that led to Yampa. We briefly review Fran, and describe some of Fran's performance issues and implementation difficulties. We then present FRP, and discuss informally why FRP's design might address one of Fran's key performance issues. We then present an example application that illustrates the limits of FRP's expressive power, describe the extension to FRP to address this issue and explore the problems raised by this extension. In the next chapter, we will present *Yampa*, and show how it can accommodate some of the examples that proved troublesome for Fran and FRP.

3.1 Origins: Fran

Fran was the first functional reactive programming system, implemented as a library for Haskell. Fran was designed around two key polymorphic data types: *behaviors* and *events*.

In Fran, the polymorphic *Behavior* type is simply a function from time to a value:

$$Behavior\,\alpha = Time \rightarrow \alpha$$

$Time$ is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the behavior. For example, if $Point$ is the type of 2-dimensional points, then the time-varying mouse position might be represented with a value of type $Behavior\ Point$.

In Fran, Behaviors are first-class values, and thus may be passed as arguments, returned as results, stored in data structures, etc. For example, we can define the following:

$$wiggle :: Behavior\ Time$$
$$wiggle = sin\ (5 * time)$$

In the above example, $time$ is a predefined Behavior (of type $Behavior\ Time$), that represents the current time (in seconds). The Behavior type constructor is defined as an instance of Haskell's Num type class, so that the functions sin and $(*)$ can be applied to numeric Behaviors. That is, the above is really short-hand for:

$$wiggle = lift1\ sin\ (lift2\ (*)\ (constB\ 5)\ time)$$

where $liftN$ lifts a (static) function of arity N to operate on behaviors by pointwise application. The appropriate type signatures are:

$$constB :: a \to Behavior\ a$$
$$lift1 :: (a \to b) \to Behavior\ a \to Behavior\ b$$
$$lift2 :: (a \to b \to c) \to Behavior\ a \to Behavior\ b \to Behavior\ c$$

3.1.1 Switching in Fran

The original exposition of Fran [23] provided declarative reactivity via the $untilB$ combinator, with the following type signature:

$$untilB :: Behavior\ a \to Event\ (Behavior\ a) \to Behavior\ a$$

Informally, b ‘$untilB$‘ e behaves as b until e occurs, and then switches to the behavior carried with the event occurrence. For example, if lbp is an event source[2] that has occurrences whenever the left mouse button is pressed, then the following

[2] Throughout this thesis, we distinguish between event *sources* and individual event *occurrences*. An event *source* is a (time-varying) signal that may have distinct *occurrences* at discrete points in time. In a somewhat unconventional choice of terminology, Fran and FRP use the type name *Event* to refer to an event source. Yampa uses the type name *Event* to refer to a possible event occurrence, which is more consistent with typical usage in event-driven programming and signal processing literature.

definition:

$$myColB :: Behavior\ Color$$
$$myColB = (constB\ red)\ `untilB`\ (lbp \mathbin{-\!\!=\!\!>} constB\ blue)$$

is a piece-wise constant behavior whose value is red until the left mouse button is pressed, at which point its value switches to $blue$. Note that the operator $\mathbin{-\!\!=\!\!>}$ is a function that tags every event occurrence with a particular value. Its type is:

$$(\mathbin{-\!\!=\!\!>}) :: Event\ a \rightarrow b \rightarrow Event\ b$$

The $untilB$ switching construct reacts only to the *first* occurrence of the given event source. A variation, $switch$, reacts to *every* event occurrence.

3.1.2 The User Type

The above example assumed the existence of an event source, lbp, whose occurrences indicate the left mouse button being pressed. In actuality, lbp and other event sources that are derived from user input are defined as functions of $User$, an abstract type representing the totality of user input to the program over time. So lbp's actual type signature in Fran is:

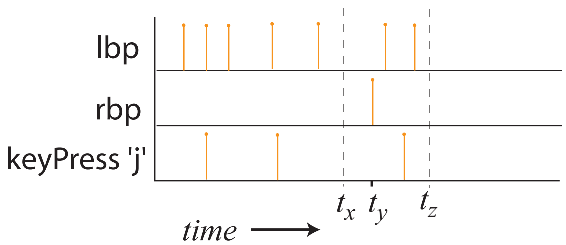
$$lbp :: User \rightarrow Event\ ()$$

This detail will be important in the forthcoming discussion concerning start times, and the workarounds for time and space leaks implemented in FRP and Yampa.

3.1.3 Time and Space Leaks in Fran

The previous switching example wasn't particularly interesting, because it merely switched from one constant behavior to another. Consider the following slightly more involved example:

$$lbpCount :: User \rightarrow Behavior\ Int$$
$$lbpCount\ u = stepAccum\ 0\ (lbp\ u \mathbin{-\!\!=\!\!>} (+1))$$
$$jCount :: User \rightarrow Behavior\ Int$$



(a) Input Trace

	t_x	t_z
lbpCount	5	7
jCount	2	3
counter	5	3

(b) Fran Semantics

	t_x	t_z
lbpCount	5	7
jCount	2	3
counter	5	1

(c) FRP Semantics

Figure 3.1: Alternatives for Semantics of Behaviors

$$
\begin{aligned}
&jCount\ u = stepAccum\ 0\ (keyPress\ u\ \texttt{'j'} \mathbin{-\!\!=\!\!>} (+1))\\
&counter :: User \rightarrow Behavior\ Int\\
&counter\ u = lbpCount\ u\ \text{`}untilB\text{`}\ (rbp\ u \mathbin{-\!\!=\!\!>} jCount\ u)
\end{aligned}
$$

The $lbpCount$ and $jCount$ behaviors are defined using Fran's $stepAccum$ combinator, which has the following type:

$$stepAccum :: a \rightarrow Event\ (a \rightarrow a) \rightarrow Behavior\ a$$

This combinator forms a piece-wise constant behavior by applying a function to a "current value" whenever an event occurs, and then holding this value until the next event occurrence. In this example, $lbpCount$ acts as a counter that is incremented every time the left mouse button is pressed. Similarly, $jCount$ is defined as a counter that is incremented every time the 'j' key is pressed on the keyboard. The behavior $counter$ is defined using Fran's $untilB$ switching construct to switch from $lbpCount$ to $jCount$ after the right mouse button is pressed.

The semantics of this example for a particular set of inputs is illustrated in figure 3.1. Figure 3.1(a) shows the occurrences of each user input event source over time. Each event occurrence is indicated by a vertical line with a circle at the top. Note that at time t_y, the right mouse button is pressed, which is the event that causes a switch in the definition of $counter$.

Figure 3.1(b) shows the result of sampling each behavior in this example if we observed them at times t_x and t_z. As we would expect, at each of these two sample times, the behaviors $lbpCount$ and $jCount$ reflect the total number of occurrences of the corresponding input event since the start of program execution. The behavior $counter$ has the same value as $lbpCount$ at time t_x (and, in fact, at all times up to t_y). At all times after the switching event occurrence (including time t_z), $counter$ has the same value as $jCount$.

Unfortunately, this semantics results in a serious operational problem called a *time leak*: at certain points in time, an implementation of Fran may need to do a significant amount of "catch up" computation to compute the output sample value.

To understand, at least informally, why this example suffers from a time leak, we need to consider how Fran is implemented. All functional reactive programming systems provide a top-level command (called $reactimate$ by convention), which will "animate" a particular behavior by repeatedly sampling it, and performing some application-specific action to convey the output sample to the user. Suppose that $reactimate$ is invoked with $counter$ as an argument. Then at all times up to t_y, $counter$ and $lbpCount$ are *active* or *running*, in the sense that they are being actively sampled. In contrast, executing $reactimate\ counter$ will not sample the behavior $jCount$ before time t_y, because at every point in time before t_y, the value of $jCount$ is not needed to compute a sample value for $counter$. At time t_y, $counter$ switches from $lbpCount$ to $jCount$. However, at every point in time, the value of $jCount$ depends on the complete history of user input. Since the implementation has not been actively sampling $jCount$ before time t_y, computing an accurate output sample at t_y requires sampling $jCount$ at all previous sample points up to t_y. Furthermore, since the value of $jCount$ at t_y depends on the complete history of key press events up to t_y, an implementation must maintain the trace of input events depicted in figure 3.1(a) in memory. Since the amount of time before the switching event occurs is unbounded, the amount of storage needed to store such input history is also unbounded, and hence this example will have a space leak as well as a time leak.

It might seem possible to avoid such time and space leaks simply by eagerly sampling every behavior that is defined in the program. In fact, an approach similar to this was attempted in an experimental implementation of Fran that was used in the FranTk project [68]. Unfortunately, this approach to avoiding time and space leaks is simply not feasible for a number of reasons. First, particularly in the context of an implementation based on embedding in a non-strict higher-order host language, it is simply not feasible to gather sufficient information about all of the behaviors defined in the program needed to do such eager sampling. But more significantly, there may be legitimate reasons for the programmer to define behaviors that produce $\perp$ as a sample value at certain points in time. In Fran, such programs will behave correctly as long as the top-level behavior being animated does not depend on the value of the bottom-producing behavior at those times when its output samples are $\perp$. In contrast, an implementation that attempted to avoid time leaks by eagerly sampling every behavior in the program would diverge on such examples.

3.2 From Fran to FRP

To alleviate the time and space leaks of Fran, Hudak developed another implementation of functional reactive programming embedded in Haskell, known as "SOE FRP" (or just "FRP"[3]). FRP differs from Fran by redefining the core Behav-

[3] A somewhat unfortunate choice of name, since it makes it difficult to distinguish between the functional reactive programming *style* and this particular implementation.

ior type to take a *start time*:

$$Behavior\,\alpha = Time \rightarrow Time \rightarrow \alpha$$

The first argument is a *start time*, whereas the second argument is the *sample time*. The start time represents an *epoch*, at which the Behavior begins execution. More precisely, given a Behavior b, start time t_{start}, and sample time t_{sample}, the sample $(b\ t_{start}\ t_{sample})$ is undefined if $t_{sample} < t_{start}$.

The use of start times is particularly significant in the definition of switching in FRP. Switching combinators (such as $untilB$) have the same type signature in FRP as they do in Fran. However, the semantics is quite different. In FRP, the time of the switching event, t_e, is used as the start time for the behavior that is switched in to.

The observable output of $counter$ under FRP semantics is shown in figure 3.1(c). Note that sampling $counter$ at time t_z yields 1 in this semantics. This is because the *switching event* (rbp) occurs time t_y. Thus t_y will be used as the *start time* for $jCount$, the Behavior being switched in to.

At first glance, it might appear somewhat suspicious that the samples for $jCount$ and $counter$ in figure 3.1(c) disagree at sample time t_z. What's really happening here is that figure 3.1(c) is somewhat incomplete, since it does not show the start times of the behaviors being sampled. What's actually being shown is each of the behaviors with start times of t_0, a global start time for the whole program (and also the origin of the horizontal axis in the input trace shown in figure 3.1(a)). Since t_y is the time of $counter$'s switching event in this input trace, t_y is passed as the start time to $jCount$ to obtain a time-varying value for use after the switch. So the value of $counter$ after the switch is given by $jCount\ t_y$, which is distinct from

$jCount\ t_0$ shown in the second row of this table.

Intuitively, it is easy to see why the FRP design avoids Fran's time and space leak for the $counter$ example. In Fran the problem was that at the time of the switching event (t_y), the system needed to perform a certain amount of *catch-up* computation. This is because, in Fran's semantics, the value of $jCount$ at time t_y depends on the complete history of user input over the interval $[0, t_y]$. In contrast, FRP passes t_y as the start time to $jCount$. By definition, $jCount$ can not be given a sample time before t_y, and hence does not depend on user input before t_y. Hence, the value of $(jCount\ t_y\ t)$ for all $t \geqslant t_y$ only depends on user input on the interval $[t_y, t]$; all user input before input t_y is irrelevant.

3.3 Understanding SOE FRP's Conceptual Model

Although SOE FRP addresses the operational issue of time- and space-leaks that arose in Fran, the change in the semantics of Behaviors has some significant consequences for application programmers. One of the compelling features of Fran is that systems of ordinary differential equations (which frequently arise in the natural sciences) could be easily realized as Fran programs. In many cases, the Fran program for such a system uses the same equations as are used in the underlying mathematical model. For example, the following is a Fran program that implements a physical model of a ball falling under the force of gravity:

$$
\begin{array}{l}
fallBall\ u = move\ (vector2XY\ x\ y)\ ball \\
\quad \textbf{where} \\
\qquad y = y0 + integral\ v\ u \\
\qquad y0 = 1 \\
\qquad v = integral\ a\ u \\
\qquad a = -9.8
\end{array}
$$

The code is a fairly direct translation of the laws of motion: the y position is the integral of velocity, and the velocity is the integral of acceleration due to gravity (both over time).

The variables that appear in differential equations are implicitly understood as functions of time. Sometimes variables are written in the more explicit form $y(t)$ instead of y to emphasize their true type. The semantic model of a Behavior as a function from time to value thus corresponds directly with the underlying mathematical notion of a time-varying quantity from differential equations.

This intuitive notion of time-varying values provides the user with a foundation for understanding and writing more complex Fran programs even when they involve switching or mutually recursive behaviors. For example, here is an extension of the above program in which the ball "bounces" once by negating the velocity immediately after the collision event:

```
bounceBall u = move (vector2XY x y) ball
  where
    y = y0 + integral v u
    y0 = 1
    bounce = predicate (y <* 0) u
    v = ia `untilB` (bounce -=> -ia)
    ia = integral a u
    a = -9.8
```

The use of *untilB* causes the object to bounce once, when the object collides with the ground. In the original Fran semantics (where behaviors are just time-varying values), the intended semantics of the above definitions corresponds directly to the basic differential equations from which they are derived, and all behaviors can be understood as simple functions from time to value.

The above Fran program can also be written (with minor syntactic differences) in SOE FRP:

```
bounceBall :: Beh Picture
```

$$
\begin{aligned}
&\mathit{bounceBall} = \mathit{moveXY}\ x\ y\ \mathit{ball} \\
&\quad \textbf{where} \\
&\qquad y = \mathit{y0} + \mathit{integralB}\ v \\
&\qquad \mathit{y0} = 1 \\
&\qquad \mathit{bounce} = \mathit{whenE}\ (y <\!*\ 0) \\
&\qquad v = \mathit{ia}\ \text{`}\mathit{till}\text{`}\ (\mathit{bounce} {-}{=}{>}\ {-\mathit{ia}}) \\
&\qquad \mathit{ia} = \mathit{integralB}\ a \\
&\qquad a = -9.8
\end{aligned}
$$

Since behaviors in SOE FRP denote *computations that produce signals* rather than just signals, the user must understand that any behavior is effectively *restarted* after a switch. In the above example, the "bounce" event will cause the velocity behavior computation (v) to start over (from 0) at the time of the switch. Part of the subtlety here is that the start time is critical to understanding the semantics of a behavior, yet there is no type-level or syntactic distinction between behaviors that have different start times. As a consequence, expressions such as $(y <\!* -0.8)$ or $-ia$ may produce different output signals depending on the context in which they appear.

3.3.1 Limits in SOE FRP's Expressive Power

In addition to the conceptual subtleties just described, the function-of-start-time model on its own also has some fundamental limitations in expressive power. Suppose, for example, that we wish to write an implementation of $counter$ from sec 3.1.3 with observable behavior as shown in figure 3.1(b). That is, we want to implement a counter that keeps count of both left mouse button presses and presses of the 'j' key. The counter should display the number of left mouse button presses until the right button is pressed, and then display the count of 'j' key presses *since the start of the program*. There is simply no way to define such a program directly in SOE FRP, since the behavior that counts presses of the 'j' key will only start at the time of the switching event occurrence.

3.4 Extending SOE FRP with runningIn

To work around SOE FRP's limitation in expressive power, Wan [77] extended SOE FRP with the $runningIn$ combinator. The $runningIn$ combinator allows the programmer to start a Behavior at a certain time, and then switch in to this (already running) behavior at some later time. Conceptually, the $runningIn$ combinator extends FRP with the following new language construct:

$$
\begin{array}{l}
\mathbf{letrun} \\
\quad b = b_1 \\
\mathbf{in} \\
\quad ..\, b \,..
\end{array}
$$

where b_1 is an expression of some Behavior type, and b is bound to a *running* version of that behavior. Since embedding in Haskell precludes defining such new binding constructs, a new combinator was introduced instead:

$$
\begin{array}{l}
runningIn :: Behavior\ a \\
\quad \rightarrow (Behavior\ a \rightarrow Behavior\ b) \\
\quad \rightarrow Behavior\ b
\end{array}
$$

The idea is that:

$$b1\ `runningIn`\ (\lambda b \rightarrow ...b...)$$

starts behavior $b1$ and binds this to b in the body of the function that is the second argument to $runningIn$. The body of the function defines a behavior (of type $Behavior\ \ b$). Within the body of the function, switching in to behavior b will switch in to a version of $b1$ that was started at the same time as the overall behavior.

Wan and Hudak [78] extended the original denotational semantics of Fran with start times. The meaning function, at gives the meaning of a Behavior in terms of a start time and a sample time:

$$\mathbf{at}[\![-]\!] :: \mathit{Behavior}\,\alpha \rightarrow \mathit{Time} \rightarrow \mathit{Time} \rightarrow \alpha$$

In the above type, the first Time argument is the *start time*, and the second is the *sample time*. Wan gives the denotation of the $\mathit{runningIn}$ construct as:

$$\mathbf{at}[\![b\ \text{`}\mathtt{runningIn}\text{`}\ f]\!]\ T\ t = f(\lambda T'.\mathbf{at}[\![b]\!]\ T)\ T\ t$$

The key to understanding the above semantic definition is to observe that T' is never used in the Behavior that is passed to f. Instead, the definition constructs a Behavior that is b started at the same time (T) as the overall behavior.

With the addition of $\mathit{runningIn}$, it becomes possible to express the Fran semantics shown in figure 3.1(b) in SOE FRP. This would be written as follows:

$$\begin{array}{l} \mathit{counter}'\ u = \mathit{jCount}\ u\ \text{`}\mathit{runningIn}\text{`} \\ \quad (\lambda \mathit{jcb} \rightarrow \mathit{lbpCount}\ u\ \text{`}\mathit{untilB}\text{`}\ \mathit{rbp}\ u {-\!\!=\!\!>}\ \mathit{jcb}) \end{array}$$

In this definition, an instance of the behavior jCount is started and bound to the identifier jcb. Within the body of the lambda expression given as the second argument to $\mathit{runningIn}$, all references to jcb refer to this *running behavior*. As a consequence, the switching event ($\mathit{rbp}\ u$) will result in switching in to a count of the number of presses of the 'j' key since $\mathit{counter}'$ was started.

3.4.1 Difficulties with runningIn

While adding the $\mathit{runningIn}$ combinator to SOE FRP recovers some of Fran's expressive power, its addition turns out to raise a number of substantial problems, both theoretical and practical. This section gives a brief summary (with examples) of some of the problems that were encountered when attempting to use $\mathit{runningIn}$

to construct large FRP applications.

Recursive Definitions

It is very common to write recursive definitions of behaviors in Fran and FRP. For example, in the bouncing ball example given in section 3.3, the ball's position is determined by its velocity, and its velocity reacts to bounce events, which are determined by the ball's position.

In Wan's implementation of SOE FRP with the *runningIn* extension, certain seemingly well-formed recursive definitions resulted in programs that diverged when executed. For example, behaviors of the form:

$$\textbf{let}\ a = \ldots \text{`}runningIn\text{`}\ (\lambda rb \rightarrow \ldots a \ldots)$$

would loop. Such patterns would occur naturally and commonly in attempting to express dynamic collections of behaviors, such as a collection of objects in a simulated world.

Unfortunately, it was never satisfactorily resolved whether the problems with recursion and *runningIn* were difficulties with the semantics of *runningIn* or the implementation. However, after considering many different examples of programs involving the combination of *runningIn* and recursion it quickly became apparent that there was no clear, simple model of what the semantics of runningIn should be in the presence of recursion.

3.5 Behaviors that "escape"

Another, purely implementation level issue with *runningIn* was that runningIn was implemented by folding a running behavior into the input type of a Behavior. While Fran's *Behavior* type constructor took only a single type parameter, FRP added an extra parameter representing the type of system input. This system input was fed to all Behaviors in FRP, and served a role similar to Fran's $User$ type.

The implementation of $runningIn$ in SOE FRP worked by starting a behavior and then extending the system input with a handle to this running behavior. Thus the actual type of the $runningIn$ combinator is:

$$runningIn :: Beh\ i\ a \rightarrow (Beh\ (i, a)\ a \rightarrow Beh\ (i, a)\ b) \rightarrow Beh\ i\ b$$

Note that, within the second (function) argument of $runningIn$, the input type parameter i has been extended to type (i, a) to account for the running behavior. Once the system has made the running behavior available on the system input, the actual "handle" to running behavior is simply an accessor for the appropriate signal that is present in the system input.

The problem with this approach is that, because behaviors are first class values, the behaviors that provide handles to running signals may escape their scope. For example:

$$\begin{aligned}
&b :: Beh\ i\ A\\
&a :: Beh\ i\ (Beh\ (i, A)\ A)\\
&a = b\ \text{`}runningIn\text{`}\ (\lambda rb \rightarrow lift0\ rb)
\end{aligned}$$

In the above code, rb is really an accessor or "reference" that refers to the running version of b fed in via the system input to the second argument of $runningIn$. Unfortunately, as this example illustrates, it is possible for such references to escape the scope of the function in which they are defined. The problem with such escaping references is that their use outside of their original scope will refer to some other running signal that appears in the surrounding context in which the reference is used, and not the original signal to which the identifier was bound.

Lack of Type Information

As discussed in section 3.3, one substantial change in the conceptual model from Fran to FRP is that the interpretation of Behavior changed from representing a simple signal (or time-varying value) to a *computation* – specifically, a function that produced a signal from a start time. The introduction of $runningIn$ partially restores access to Fran's simpler model of signals. However, it does so by providing accessors to running signals as behaviors, which can be freely mixed with other behaviors.

Unfortunately, our experience using SOE FRP with runningIn to implement actual applications was that this overloading of the $Behavior$ type was fundamentally confusing. Within any given expression, a subexpression with type $Behavior\ a$ might refer to a computation returning a signal, a signal, or some combination of the two. This made it very difficult for the programmer to construct a mental model of how the program would behave at runtime.

3.6 Chapter Summary

Fran was the first implementation of the functional reactive programming model. Unfortunately, Fran's implementation suffered from a number of substantial operational problems, most notably time and space leaks. Hudak's SOE FRP system was an attempt to resolve the operational issues of Fran by adopting a different programming model for Fran's core Behavior type. Unfortunately, this alternative model made it impossible to express certain kinds of programs. Attempts at extending the FRP model to regain Fran's expressive power (the addition of runningIn) raised its own set of design and implementation issues. This situation lead us to develop Yampa as an alternative framework for functional reactive programming which would retain both the expressive power and conceptual simplicity of Fran, but which would avoid Fran's fundamental operational difficulties.

Chapter 4

The Formal Semantics of Yampa

Yampa can be considered as a domain-specific language embedded in Haskell [34]. Embedding in Haskell allows us to leverage Haskell's type system, binding constructs, expression syntax and evaluation rules, enabling us to focus on the semantic details of our new language constructs. In this chapter, we define a core subset of Yampa and present its semantics.

-- Arrow combinators:
$arr :: (a \to b) \to SF\ a\ b$
$(\ggg) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c$
$\mathit{first} :: SF\ b\ c \to SF\ (b, d)\ (c, d)$

-- Feedback combinator (from ArrowLoop):
$loop :: SF\ (b, d)\ (c, d) \to SF\ b\ c$

-- Initialized delay element:
$iPre :: a \to SF\ a\ a$

$time :: SF\ a\ Time$
$integral :: (Floating\ a) \Rightarrow SF\ a\ a$

-- basic switching combinator:
$switch :: SF\ a\ (b, Event\ c) \to (c \to SF\ a\ b) \to SF\ a\ b$

Figure 4.1: Core Primitives of Yampa

4.1 Direct Denotational Semantics

Figure 4.1 gives the Haskell types for the primitive signal functions and combinators that form the core of Yampa.

The direct denotational semantics for Yampa is modeled on the denotational semantics of Fran [23] and FRP [78]. In order to focus on the semantics of Yampa constructs, we assume the existence of an underlying denotational semantics for Haskell expressions in which Haskell function definitions denote continuous partial functions.

Semantic Domains We define $Time$ to be a non-negative real number:

$$Time = \mathbb{R}^{+}$$

A *signal* is a continuous function from $Time$ to a value:

$$Signal\ \alpha \;\; = \;\; Time \rightarrow \alpha$$

A *signal function* is a function from $Signal$ to $Signal$:

$$SF\ \alpha\ \beta \;\; = \;\; Signal\ \alpha \;\rightarrow\; Signal\ \beta$$

We follow the Haskell convention that the function domains are implicitly lifted; we will need this property to account for the semantics of $loop$.

Semantic Equations Our denotational semantics simply maps Haskell signal function expressions to a value in the corresponding domain:

$$[\![-]\!] \ : \ \langle \mathit{sfexp} \rangle \rightarrow SF\ \alpha\ \beta$$

The arrow operators are all straightforward:

$$\begin{aligned}
[\![\mathit{arr}\ f]\!] &= \lambda s\ .\ \lambda t\ .\ [\![f]\!]\ (s\ t) \\
[\![\mathit{fa} \ggg \mathit{ga}]\!] &= ([\![\mathit{ga}]\!] \circ [\![\mathit{fa}]\!]) \\
[\![\mathit{first}\ \mathit{fa}]\!] &= \lambda s\ .\ \mathit{pairZ}\ ([\![\mathit{fa}]\!]\ (\mathit{fstZ}\ s))\ \ (\mathit{sndZ}\ s)
\end{aligned}$$

The semantics of arr relies on our assumption of an underlying denotation for the Haskell function f. The use of pairZ, fstZ, sndZ in the denotation of first are the straightforward lifting of the standard operations on tuples to the Signal domain:

$$\begin{aligned}
\mathit{pairZ} &= \lambda sa\ .\ \lambda sb\ .\lambda t\ .\ (sa\ t, sb\ t) \\
\mathit{fstZ} &= \lambda s\ .\ \lambda t\ .\ (\mathit{fst}\ (s\ t)) \\
\mathit{sndZ} &= \lambda s\ .\ \lambda t\ .\ (\mathit{snd}\ (s\ t))
\end{aligned}$$

In the above, fst and snd are the standard projection operators over primitive pairs[1]

Integration and Time The primitive signal function time provides access to the current time as a signal:

$$[\![\mathit{time}]\!] \ = \ \lambda s.\lambda t.t$$

The output signal of the primitive signal function $\mathit{integral}$ is the integration of

[1]Standard mathematical notation uses numeric subscripts for these projection functions. We use the explicit names instead to avoid confusion with subscripted variable names, at the possible cost of introducing confusion between the meta-language and object-language functions of the same name.

its input signal over time. Formally:

$$[\![integral]\!] \; = \; \lambda s.\lambda t. \int_0^t s(t)dt$$

ArrowLoop For signal functions, if fa has type $SF \;\; (b, d) \;\; (c, d)$, then $loop \;\; fa$ denotes a signal transformer that instantiates fa, and pairs the second half of fa's output signal with an external input signal to form fa's input signal. The output of $loop \; fa$ is the first half of fa's output signal.

Formally, we define $loop$ for signal functions as:

$$\begin{aligned} &[\![loop \; fa]\!] \; = \\ &\quad \lambda s \; . \; fstZ(\mathbf{Y}(\lambda r.[\![fa]\!](pairZ \; s \; (sndZ \; r)))) \end{aligned}$$

where $\mathbf{Y}$ is the standard least fixed point operator.

In order to ensure that loops are well-defined, the programmer must ensure that some signal function on the feedback path is capable of producing an output sample at time t without observing its input signal at t. In other words, somewhere on the feedback path there must be a signal function that is non-strict *at the current time*. There are many ways of introducing such non-strict signal functions. One possibility is simply to lift a non-strict function (such as $const$) using arr. Another way to break causality loops is to use a primitive signal function that is capable of producing a reaction to its output signal at time t without observing its input at t by introducing a *delay* between input and output. In Fran, such delays were implicitly provided by the $integral$ and $switch$ signal functions. However, our experience using Yampa for large scale applications revealed numerous cases where both delayed and non-delayed forms of switching were useful [56, 36], and other situations where it was convenient to define signal functions using feedback even though there was no obvious place for an *integral* or *switch* on the feedback path. Thus, Yampa provides a primitive delay element, *iPre* ("initialized" previous element), which introduces an infinitesimal delay between its input and output signal:

$$[\![iPre\ x]\!] \;=\; \lambda s.\lambda t. \begin{cases} [\![x]\!] & t \leqslant \epsilon \\ s\,(t\,-\,\epsilon) & otherwise \end{cases}$$

As in the denotation of *arr*, the denotation of *iPre* depends on the underlying denotational semantics of Haskell for the meaning of x. In the above definition, ϵ is an infinitesimally small unit of time. Following the approach of Wan and Hudak [78], the overall *meaning* of a Yampa program is the meaning of a signal function expression as this ϵ approaches 0:

$$\mathcal{M}[\![e]\!] \;=\; \lim_{\epsilon \to 0} [\![e]\!]$$

Switching and Event Occurrences Following the semantics of Fran, we define the following auxiliary function, **occ** that returns the first event occurrence on an event signal:

$$\begin{aligned} &\mathbf{occ}\ :\ Signal\,(Event\,\alpha)\ \rightarrow\ (Time, \alpha) \\ &\mathbf{occ}\, s\ =\ (t_e, \alpha) \\ &\qquad\quad \mathbf{where}\ \ (s\, t_e\ =\ Event\ \alpha)\ \wedge\ (\forall t \in [0, t_e).s\, t = NoEvent) \end{aligned}$$

The use of the universal quantifier over an interval of time in the above definition of **occ** is not computable. Aside from efficiency concerns, this choice prevents us from using the denotational semantics given here as the basis for a reference implementation in a functional language.

Given the **occ** function, the Yampa switching primitive is defined as follows:

$$[\![switch\ sf_0\ f]\!] \;=\; \lambda s.\lambda t. \begin{cases} (fstZ\ bes)\ t & t < t_e \\ (timeShift\ (t - t_e)([\![f]\!]\ c))\ s\ t & otherwise \end{cases}$$

$$\textbf{where} \quad bes \;=\; [\![sf_0]\!]\ s$$

$$(t_e, c) \;=\; occ\ (sndZ\ bes)$$

The *timeShift* function used above applies a simple translation *time transform* to a signal function:

$$timeShift \;=\; \lambda dt\ .\ \lambda sf\ .\ (ts\ (-dt)) \circ sf \circ (ts\ dt)$$

$$\textbf{where}\ ts \;=\; \lambda dt\ .\ \lambda s\ .\ \lambda t\ .\ s\ (t\ +\ dt)$$

This definition of time-transform is based on the general notion of how to apply a transform to a function used in Pan's hyper-filters [22]: To transform a function (in time or in space), apply the transform to the function's input, and apply the *inverse* transform to the resulting output.

The semantics of switching is defined in terms of time transformation in order to ensure *temporal modularity*. Temporal modularity is the property that all time-dependent values produced by a signal function are relative to some externally specified frame of reference, rather than relative to some (arbitrary) choice of global reference frame / start time. Although we do not currently support a general time transformation operator in Yampa, temporal modularity enables us to introduce such an operator in the future. Another advantage of using a time transform is that there is no arbitrary start time to consider; every signal function executes in some localized time frame that starts at $t = 0$.

4.2 Operational Semantics

The operational semantics of the primitives of Yampa is shown in figure 4.2. As we did with the denotational semantics, we assume the existence of an underlying semantics of Haskell expressions. The meta-variables e_n, f and g (and their primed forms) range over Haskell expressions, with f and g restricted to expressions of type signal function ($SF\ \ a\ \ b$). The meta-variable dt is a real number.

$$\frac{e \Rightarrow z \quad z \xrightarrow{dt,e_1} \langle f', e_2 \rangle}{e \xrightarrow{dt,e_1} \langle f', e_2 \rangle}\ (\textsf{sf-eval})$$

$$\frac{}{\textsf{arr}\ e_1 \xrightarrow{dt,e_2} \langle \textsf{arr}\ e_1, e_1\ e_2 \rangle}\ (\textsf{sf-arr})$$

$$\frac{f \xrightarrow{dt,\textsf{fst}\ e_1} \langle f', e_2 \rangle}{\textsf{first}\ f \xrightarrow{dt,e_1} \langle \textsf{first}\ f', (e_2, \textsf{snd}\ e_1) \rangle}\ (\textsf{sf-first})$$

$$\frac{f \xrightarrow{dt,e_1} \langle f', e_2 \rangle \quad g \xrightarrow{dt,e_2} \langle g', e_3 \rangle}{\textsf{comp}\ f\ g \xrightarrow{dt,e_1} \langle \textsf{comp}\ f'\ g', e_3 \rangle}\ (\textsf{sf-comp})$$

$$\frac{}{\textsf{dtime} \xrightarrow{dt,e_1} \langle \textsf{dtime}, dt \rangle}\ (\textsf{sf-dtime})$$

$$\frac{}{\textsf{iPre}\ e_1 \xrightarrow{dt,e_2} \langle \textsf{iPre}\ e_2, e_1 \rangle}\ (\textsf{sf-iPre})$$

$$\frac{x \text{ is a fresh variable} \quad f \xrightarrow{dt,(e_1,x)} \langle f', e_2 \rangle}{\textsf{loop}\ f \xrightarrow{dt,e_1} \langle \textsf{let}\ x = \textsf{snd}\ e_2\ \textsf{in loop}\ f', \textsf{let}\ x = \textsf{snd}\ e_2\ \textsf{in fst}\ e_2 \rangle}\ (\textsf{sf-loop})$$

$$\frac{f \xrightarrow{dt,e_2} \langle f', e_3 \rangle}{\textsf{switch}\ f\ e_1 \xrightarrow{dt,e_2} \langle \textsf{maybe}\ (\textsf{switch}\ f'\ e_1)\ e_1\ (\textsf{snd}\ e_3)\ ,\ \textsf{maybe}\ (\textsf{fst}\ e_3)\ (\textsf{sample}\ 0\ e_2\ \circ\ e_1)\ (\textsf{snd}\ e_3) \rangle}(\textsf{sf-switch})$$

$$\frac{f \xrightarrow{dt,e_1} \langle f', e_2 \rangle}{\textsf{sample}\ dt\ e_1\ f \Rightarrow e_2}(\textsf{prim-sample})$$

Figure 4.2: Operational Semantics for Core Yampa

Judgments of the form $e \Rightarrow z$ refer to the assumed operational semantics of Haskell, and are read as "expression e evaluates to canonical form z". Judgments of the form $f \xrightarrow{dt,e_1} \langle f', e_2 \rangle$ are to be read, "the signal function f, when sampled at delta-time dt with input sample e_1 yields output sample e_2, and continuation f'. Note that the input sample e_1 and the output sample e_2 are un-evaluated expressions. This is consistent with our reference implementation of Yampa in Haskell, and is crucial to ensuring that certain loop constructs are well formed.

As in RT-FRP, the overall execution, or "run" of a Yampa program under this operational semantics is modeled as an infinite sequence of sampling "time steps". Execution of each step occurs in the context of a *current signal function*, f. At each step, the time since the last step (dt) and an input sample (e_1) are obtained from the environment. These are fed to the evaluation relation to obtain an *output sample* for the current time step and a *next signal function* for use in the next time step. The output sample is made available to the environment via some mechanism not specified here, and execution proceeds at the next time step using the "next" signal function obtained from the evaluation relation of the previous time step.

An essential aspect of Yampa is that signal functions are first-class values. This point is formalized in the first rule of figure 4.2 (sf-eval). In any context where a signal function is required, an arbitrary base-language expression may be specified, so long as it reduces to one of the canonical signal function forms handled by one of the other rules of figure 4.2.

The next three rules, sf-arr, sf-first and sf-comp, formalize the standard Arrow operators [39] in the context of signal functions. The rule sf-arr *lifts* a static function e to the signal function level by applying the function point-wise to every input sample. Note the lack of strictness here: neither the input sample nor the static function nor the resulting expression for the output sample are evaluated.

The rule sf-first *widens* a signal function to one whose input and output sample types are pairs. The first component of the input sample ($\mathit{fst}\, e_1$) is fed to the embedded signal function f to obtain an output sample e_2. The overall output sample is produced by pairing this with the second component of the original input sample ($snd\ e_1$).

The rule sf-comp is serial composition, corresponding to the Arrow operator $\ggg$. The external input sample e_1 is fed to f to produce an output sample e_2, and this is used as the input sample for g to obtain the overall output sample e_3.

The operational semantics shown in figure 4.2 only specifies a $dtime$ ("delta time") primitive, and does not include explicit definitions for $time$ or $integral$ from figure 4.1. The $dtime$ primitive provides Yampa programs access to the amount of logical time elapsed since the last input sample, by making the value dt available as an output sample. In this semantics, $time$ and $integral$ can be defined in terms of the Yampa primitives $dtime$, $iPre$ and $loop$. For example, here is an implementation of $time$ using these primitives:

$$
\begin{array}{l}
time :: SF\ ()\ Double \\
time = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ dt \leftarrow dtime \prec ()\\
\qquad\quad t \leftarrow iPre\ 0 \prec t + dt \\
\quad returnA \prec t
\end{array}
$$

Note that this definition (written using the Arrows syntactic sugar) includes a recursive binding for t, since t occurs on both the input and output side of an arrow.

The next two rules, sf-iPre and sf-loop specify the semantics of the delay and feedback operators. The rule sf-iPre ("initialized" pre) produces as an output sample the input sample from the previous time step. Note that the input sample for the current time step, e_2, is captured in the signal function continuation for use in the next time step.

The sf-loop rule specifies Yampa's feedback operator, $loop$, which makes part of the output of a signal function available as input to that signal function. This is similar in spirit to the typical definition of a fixed point combinator in the non-strict λ-calculus:

$$\overline{\mathsf{fix}\; e \;\rightarrow\; e\;(\mathsf{fix}\; e)}$$

However, the interplay between the term level evaluation and sampling of signal functions requires a slightly more involved definition. We introduce a completely fresh variable identifier, x, that acts as a "placeholder" for the fed-back value, and pair this with the external input sample e_1. This un-evaluated pair is fed as the input sample to the embedded signal function f to obtain an output sample e_2 and signal function continuation f'. This definition depends crucially on the non-strictness of the signal function sampling relation, in that the relation must be capable of producing expressions for the signal function continuation and output sample *without evaluating the placeholder variable x*. We will have more to say about this point shortly when we examine the semantics of $switch$. To account for the possibility that x may occur as a free variable in either f' or e_2, the conclusion of sf-loop introduces a recursive **let** binding that binds x to the second component of the output sample e_2 (a pair).

Finally, the rule sf-switch defines Yampa's $switch$ construct. Recall that $switch$ has type:[2]

$$switch :: SF\;\; a\;\; (b, Event\;\; c) \rightarrow (c \rightarrow SF\;\; a\;\; b) \rightarrow SF\;\; a\;\; b$$

Sampling a $switch$ construct yields term-level expressions for both the signal function continuation and the output sample. Both of these expressions use the standard $maybe$ function applied to the possible event occurrence value ($\mathsf{snd}\; y$) obtained by sampling the embedded signal function (e_1). If there is no event occurrence, the overall signal function continuation is $switch$ applied to the embedded

[2]N.B. We're using the $Event\,/\,Maybe$ isomorphism here.

signal function's continuation (e_1'). Otherwise, the overall signal function continuation is given by applying function e_2 to the value carried with the event occurrence. Similarly, the overall output sample from the switch is either taken from the output sample of the embedded signal function ($\mathsf{fst}\ y$) if no event occurs, or is computed by sampling the signal function given by applying e_2 to the event occurrence value. This is consistent with Yampa's instantaneous switching semantics [56]; if we switch in to a new signal function at time t, the overall output sample at t results from sampling the *new* signal function at t.

The instantaneous switching semantics requires that we introduce a term-level sampling primitive, $sample$, whose semantics is given by the rule prim-sample. A $sample$ expression, when evaluated, yields the output sample (y) produced by sampling the given signal function (e) with the given input sample (x) and delta-time(dt). We had previously considered an alternative (but flawed!) formulation of the semantics for switch that does not involve this term-level $sample$ primitive. The rules for this alternative formulation are shown in figure 4.3. To see why this formulation is flawed, recall that our definition of sf-loop requires that the sampling relation must be able to produce a signal function continuation and output sample without evaluating any of its arguments. This is clearly violated by the two rules of figure 4.3, since the premises require evaluation of the embedded signal function's output sample to check for a possible occurrence of the switching event. We also considered an alternative formulation for $loop$ that did not involve a "placeholder" variable. However, this required the introduction of a term-level $pull$ operator to define $loop$ that behaved very much like our $sample$ primitive, yet still required the term-level $sample$ primitive in the definition of $switch$.

$$\frac{e_1 \xrightarrow{dt,e_1} \langle e_1', y\rangle \quad y \Rightarrow (z, \mathsf{Nothing})}{\mathsf{switch}\ e_1\ e_2 \xrightarrow{dt,x} \langle \mathsf{switch}\ e_1'\ e_2, z\rangle}\ (\textsf{sf-switch-ne})$$

$$\frac{e_1 \xrightarrow{dt,e_1} \langle e_1', y\rangle \quad y \Rightarrow (\_, \mathsf{Just}\ z) \quad (e_2\ z) \Rightarrow e_3 \quad e_3 \xrightarrow{dt,x} \langle e_4, y'\rangle}{\mathsf{switch}\ e_1\ e_2 \xrightarrow{dt,x} \langle e_4, y'\rangle}\ (\textsf{sf-switch-e})$$

Figure 4.3: Flawed Semantics for Switching

4.3 Derived Combinators

A number of important and useful functions and signal functions provided by Yampa are easily defined in terms of the core primitives of Yampa, and hence do not themselves need to be defined as primitives. Some of these utility combinators are shown (with their definitions) in figure 4.4.

Most of the utility combinators of figure 4.4 use feedback (loops) to accumulate state. The most common of these is the $hold$ signal function. The $hold$ signal function provides a continuous view of a discrete event signal by "latching" (or *hold*ing) the value of the last event occurrence across a period of non-occurrences, and was described in section 2.7.

Many of the utility signal functions such as $hold$ are provided in both delayed and non-delayed versions. The names of delayed versions of the utilities are prefixed by a 'd': $dHold$, $dAccum$, etc. The presence or absence of a delay determines when the reaction to an event on the input signal is observable on the output signal. In the non-delayed versions, the reaction to an input event at time t is directly observable on the output signal at t, whereas in the delayed versions, the reaction to an event at time t is not observable on the output signal until time $t + \epsilon$.

```
-- "sample and hold" or "zero order hold":
dHold :: a → SF (Event a) a
dHold x0 = proc e → do
rec px ← iPre x0 ⤙ event px id e
returnA ⤙ px

-- non-delayed version:
hold :: a → SF (Event a) a
hold x0 = proc e → do
rec px ← iPre x0 ⤙ x
  let x = event px id e
returnA ⤙ x

-- accumulating event processor:
accum :: a → SF (Event (a → a)) (Event a)
accum x0 = proc fe → do
  rec px ← iPre x0 ⤙ x
    let x = event x (app px) fe
  returnA ⤙ fe `tag` x
  where
    app :: a → (a → b) → b
    app x f = f x

-- rising edge detector:
edge :: SF Bool (Event ())
edge = proc b → do
  pb ← iPre True ⤙ b
  returnA ⤙ if ((pb, b) ≡ (False, True))
    then Event ()
    else NoEvent
```

Figure 4.4: Standard Yampa Utility Routines

4.4 Chapter Summary

This chapter presented the formal semantics of Yampa, giving both a denotational and operational semantics. The denotational semantics defines what a Yampa program "means" in the abstract, whereas the operational semantics determines how a Yampa program will actually behave at runtime. The semantics presented here will be used as a basis for implementing Yampa in the next chapter, and for precise reasoning about Yampa programs in chapter 9.

Chapter 5

Implementing Yampa

Elliott [20] described a number of different approaches to functional implementations of Fran, the first language in the FRP family. One approach is based on synchronized stream processors. This approach was pursued further by Hudak and Wan [35, 78], and later also formed the basis for an early implementation of Yampa. Our current implementation uses continuations, inspired by a similar encoding used in the implementation of Fudgets [10]. In this section, we briefly review the synchronous stream-based implementation, present our alternative continuation-based encoding, and describe some simple enhancements to the continuation-based encoding that enable dynamic optimizations.

5.1 Synchronized Stream Processors

In the stream-based representation of signal functions, signals are represented as time-stamped streams, and signal functions are just functions from streams to streams:

$$\textbf{type}\ \mathit{Time} = \mathit{Double}$$
$$\textbf{type}\ \mathit{SP}\ a\ b = \mathit{Stream}\ a \rightarrow \mathit{Stream}\ b$$

$$\textbf{newtype}\ SF\ a\ b = SF\ (SP\ (Time, a)\ b)$$

The $Stream$ type above can be implemented directly as a (lazy) list in Haskell, as described by Hudak [35].

In the above definition, each signal function ($SF\ \ a\ \ b$) is implemented as a Haskell function from a time-stamped stream of a values to a stream of b values. Time stamps may be omitted from the output stream because the implementation is *synchronous*: at every time step, a signal function will consume exactly one value from the head of its input stream and produce exactly one value on its output stream.

While a stream-based implementation is adequate for many purposes, it does have some substantial deficiencies:

- The synchronous nature of every primitive signal function is a critical requirement, but is not explicit in the implementation structure. That is, we require that each stream process consume exactly one input sample from its input stream and produce exactly one output sample on its output stream at every time step, but this is not explicit in the above type definition.

- At the implementation level, there is no way to identify signal functions that only react to *changes* in the input signal. As a consequence, sampling must occur at every time step, even though the program will only react to specific input events. Identifying signal functions that only react to changes in input would enable the implementation to make a blocking call to the operating system until an appropriate event occurs, a substantial performance improvement.

- The implementation does not retain enough information to do any runtime optimization of the dataflow graph.

- In chapter 8, we will introduce first-class signal function continuations, which provide a feature analogous to first-class continuations in functional languages [44, 67]. It does not seem to be possible[1] to implement first-class signal function continuations in a stream-based implementation, since the connection between a signal function and its input stream is hidden in a closure.

Since first-class signal function continuations are central to the way we handle structurally dynamic systems, we consider it essential to use an alternative representation of signal functions, presented below.

5.2 Continuation-Based Implementation

The Fudgets [10] graphical user interface toolkit is based on asynchronous stream processors. In [10], Carlsson and Hallgren present an alternative to the $\mathit{Stream}\ a \rightarrow \mathit{Stream}\ b$ representation of stream processors based on *continuations*. Inspired by this, we have adopted a continuation-based representation for Yampa. However, since we are working in a synchronous setting, there are substantial differences from the Fudgets implementation. A similar representation, called "residual behaviors", was explored as a possible implementation for Fran in [20].

We start by explaining a simplified version of the signal function representation, shown below. Optimizations will be discussed later.

$$\begin{array}{l} \mathbf{type}\ \mathit{DTime} = \mathit{Double} \\ \mathbf{data}\ \mathit{SF}\ a\ b = \\ \quad \mathit{SF}\{\mathit{sfTF} :: \mathit{DTime} \rightarrow a \rightarrow (\mathit{SF}\ a\ b, b)\} \end{array}$$

In this implementation, each signal function is encoded as a *transition function*. The transition function takes as arguments the amount of time passed since the

[1] At least not without stepping outside Haskell.

previous time step ($DTime$), and the current instantaneous value of the input signal (a). The time deltas are assumed to be strictly greater than 0. We will return to the question of what the first time delta should be below.

The transition function returns a pair of results:

- a *continuation* (of type $SF\ \ a\ \ b$), determining how the signal function will behave at the next time step;

- an *output sample* (of type b), determining the output at the current time step.

The top-level function responsible for animating a signal function (called $reactimate$) runs in an infinite loop: It reads an input sample and the time from the external environment (typically via an I/O action), feeds this sample value and corresponding $DTime$ to the SF's transition function to obtain an output sample and signal function continuation, and writes the output sample to the environment (also typically via an I/O action). The loop then repeats, but uses the *continuation* returned from the transition function on the next iteration.

5.3 Implementing Primitives

Most of the Yampa primitives have clear and simple implementations as continuations. For example:

$$
\begin{aligned}
&constant :: b \rightarrow SF\ \ a\ \ b \\
&constant\ \ b = SF\{\, sfTF = \lambda\_\ \_ \rightarrow (constant\ \ b, b)\} \\
&identity :: SF\ \ a\ \ a \\
&identity = SF\{\, sfTF = \lambda\_\ a \rightarrow (identity, a)\}
\end{aligned}
$$

Of course these are just special cases of the point-wise lifting operator, arr:

$$
\begin{aligned}
&sfArr :: (a \rightarrow b) \rightarrow SF\ \ a\ \ b \\
&sfArr\ f = SF\{\, sfTF = \lambda\_\ a \rightarrow (sfArr\ f, f\ \ a)\}
\end{aligned}
$$

The above primitives are all *stateless*. This fact is obvious from their definitions: the continuation returned from the transition function is exactly the signal function being defined. As an example of a *stateful* signal function, here is a simple implementation of $integral$:

$$
\begin{array}{l}
integral :: Fractional\ a \Rightarrow SF\ a\ a \\
integral = SF\{\, sfTF = sfAux\ 0 \} \\
\quad \mathbf{where} \\
\quad\quad sfAux :: Fractional\ a \Rightarrow \\
\quad\quad\quad a \rightarrow DTime \rightarrow a \rightarrow (SF\ a\ a, a) \\
\quad\quad sfAux\ acc\ dt\ a = (SF\{\, sfTF = tf \}, acc) \\
\quad\quad\quad \mathbf{where}\ tf = sfAux\ (acc + a * realToFrac\ dt)
\end{array}
$$

The auxiliary function $sfAux$ uses partial application to capture the internal state of the integral in the accumulator (acc) argument of the transition function.

Many of the higher-order primitives (those that accept signal functions as arguments) are also straightforward. For example serial composition:

$$
\begin{array}{l}
(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
(SF\{\, sfTF = tf1 \}) \ggg (SF\{\, sfTF = tf2 \}) = \\
\quad\quad SF\{\, sfTF = tf \} \\
\quad \mathbf{where} \\
\quad\quad tf\ dt\ a = (sf1' \ggg sf2', c) \\
\quad\quad\quad \mathbf{where} \\
\quad\quad\quad\quad (sf1', b) = tf1\ dt\ a \\
\quad\quad\quad\quad (sf2', c) = tf2\ dt\ b
\end{array}
$$

This definition follows naturally from the semantic definition of serial composition given in chapter 4. The transition function (tf) simply feeds the input sample and $DTime$ to the first signal function ($tf1$) to obtain $sf1'$ and b, feeds the resulting sample and $DTime$ to $tf2$ to obtain $sf2'$ and c, and returns a continuation that is the composition of the continuations $sf1'$ and $sf2'$, along with the output sample value c.

5.4 Encoding Variability

The continuation-based representation of SF allows for simple, precise operational definitions for the various combinators. However, this representation, while simple and general, hides some information that is potentially useful for optimization. For example, the concrete representation of SF makes no distinction between *stateless* and *stateful* signal functions.

To enable some simple runtime optimizations (described in the next section), we add extra constructors to the concrete representation of SF that encode certain properties of the signal function. We also add a separate type for the initial continuation, since there is no delta time to be fed in at the very first time step:

$$
\begin{array}{l}
\mathbf{data}\ SF\ a\ b = SF\{\mathit{sfTF} :: a \to (SF'\ a\ b, b)\} \\
\mathbf{data}\ SF'\ a\ b \\
\quad = \mathit{SFGen}\{\mathit{sfTF}' :: \mathit{DTime} \to a \to (SF'\ a\ b, b)\} \\
\quad \mid \mathit{SFArr}\{\mathit{sfTF}' :: \mathit{DTime} \to a \to (SF'\ a\ b, b), \\
\quad\quad \mathit{sfAFun} :: a \to b\} \\
\quad \mid \mathit{SFConst}\{\mathit{sfTF}' :: \mathit{DTime} \to a \to (SF'\ a\ b, b), \\
\quad\quad \mathit{sfCVal} :: b\}
\end{array}
$$

Each of the constructors still carries a transition function. The interpretation of the constructors is as follows:

SFGen denotes the most general case of a signal function, where there is no particular "extra" information known about the transition function.

SFArr denotes a point-wise or "pure" signal function. At any time t, the output signal at t depends only on the input sample at t (and not on the time since the last sample). Since a point-wise function is "stateless", the continuation is always just the same signal function regardless of the input sample or $DTime$.

SFConst denotes a signal function that has "gone constant". The output value and continuation for a constant signal function do not change from one sample to the next, regardless of input sample or $DTime$ value.

Formally, we can specify the properties captured by the constructors $SFArr$ and $SFConst$ by means of two predicates, $isArr$ and $isConst$, defined with respect to the original (non-optimized) definition of SF given in section 5.2:

$nextSF :: SF\ a\ b \rightarrow DTime \rightarrow a \rightarrow SF\ a\ b$
$nextSF\ sf\ dt\ a = fst\ ((sfTF\ sf)\ dt\ a)$
$sampleSF :: SF\ a\ b \rightarrow DTime \rightarrow a \rightarrow b$
$sampleSF\ sf\ dt\ a = snd\ ((sfTF\ sf)\ dt\ a)$

$$\begin{aligned}
isGen(sf) &= True \\
isArr(sf) &= \forall a.\forall dt.((nextSF\ sf\ dt\ a) = sf) \wedge \\
&\quad\ \forall a.\forall dt_1, dt_2.(sampleSF\ sf\ dt_1 a) = \\
&\qquad (sampleSF\ sf\ dt_2 a) \\
isConst(sf) &= isArr(sf) \wedge \\
&\quad\ \forall a_1, a_2.\forall dt_1, dt_2.(sampleSF\ sf\ dt_1 a_1) = \\
&\qquad (sampleSF\ sf\ dt_2 a_2)
\end{aligned}$$

The first part of the conjunction for $isArr$ asserts that sf's continuation is sf itself. The second part asserts that the sample value is the same regardless of the time delta (dt) between samples. The predicate $isConst$ extends $isArr$ with the requirement that the sample value is independent of delta time or input sample.

Since the first part of $isArr$ specifies that the same signal function is returned as the continuation for the next time step, it follows trivially by induction that $isArr$ and $isConst$ will hold for all subsequent samples of a signal function, and that same value will be returned for all subsequent samples of a constant signal function. Also note that the following implications hold:

$$isConst(sf) \implies isArr(sf) \implies isGen(sf)$$

Making the variability of signal functions explicit in the constructor enables two key optimizations:

- At the level where $reactimate$ interacts with the operating system, knowing that a signal function is $SFConst$ or $SFArr$ makes it possible to avoid redundant polling. For example, a signal function with variability $SFArr$ reacts only to changes in its input signal, not the progression of time. This enables $reactimate$ to make a blocking call to the operating system while waiting for an input sample, thus avoiding redundant polling. At present, the utility of this is limited, but the idea could be carried further by refining the constructors.

- The information about signal functions encoded in each of these constructors enables certain dynamic optimizations to the dataflow graph, based on some simple algebraic identities described in the next section.

5.5 Simple Dynamic Optimizations

As a signal function is animated, every signal function in the dataflow graph returns a new continuation at every time step. Encoding variability information in constructors enables the implementation to simplify the data flow graph if the graph reaches certain states as it is animated. For example, consider the Yampa primitive $once$:

$$once :: SF\ (Event\ a)\ (Event\ a)$$

This is a stateful filter that will only pass the *first* occurrence of its input signal to its output signal. Although the transition function for $once$ has variability $SFGen$ at initialization time, after the input signal has had an event occurrence, the continuation returned by $once$ will be equivalent to $constant\ NoEvent$, with variability

$SFConst$, and will therefore have no subsequent occurrences.

Such information can be used to optimize the dataflow graph dynamically by exploiting simple algebraic identities. For example, our implementation of serial composition exploits the following identities:

$$\begin{aligned}
&sf \ggg constant\ c = constant\ c \\
&constant\ c \ggg arr\ f = constant\ (f\ c) \\
&arr\ f \ggg arr\ g = arr\ (g \circ f)
\end{aligned}$$

Our optimized implementation of serial composition uses pattern matching to identify the above cases; the implementation follows directly from the above identities, with a default case to be applied when none of the above optimizations are applicable.

We also provide optimized versions of some of the other wiring combinators, such as first, using the identities:

$$\begin{aligned}
&\mathit{first}\ (constant\ b) = arr\ (\lambda(_, c) \rightarrow (b, c)) \\
&(\mathit{first}\ (arr\ f)) = arr\ (\lambda(a, c) \rightarrow (f\ a, c))
\end{aligned}$$

One special case that we would have liked to encode in our constructors was $SFId$, to indicate the special case of the lifted identity function, $arr\ \ id$. For example, consider the signal function

$$initially :: a \rightarrow SF\ a\ a$$

that behaves as the identity function, except at the instant $t = 0$, when its first argument (the initial value) is used as the output sample. If we could capture the fact that a signal function has become $(arr\ \ id)$ in the representation of SF', then we could exploit identities such as:

$$\mathit{first}\ (arr\ id) = arr\ id$$

Unfortunately, it appears that we would need dependent types to exploit such a constructor, since the type of the transition function ($sfTF'$) is too general. We could potentially keep around an extra function as a "proof" that we can perform the required coercions, but then the resulting code is no more efficient than using

$SFArr$ in the first place.

One last point is that we must be careful when propagating variability information. For example, even if both arguments to a switch are $SFArr$, the resulting signal function is still $SFGen$. This is because $switch$ in itself is a stateful operation. We have explored adding another constructor to SF' that basically would capture the case that something could be $SFArr$ for a while. This would yield more opportunities for blocking I/O. However, it is not part of the current implementation.

5.6 Chapter Summary

This chapter presented our implementation of Yampa. Our implementation is based directly on the operational semantics of chapter 4. We also described a number of dynamic runtime optimizations that we have implemented. These optimizations are based on using an explicit constructor representation to encode information about the "variability" of signal functions when it is known, and exploiting laws about composition of signal functions of known variability.

Chapter 6

Haven: Functional Vector Graphics

6.1 Introduction

A fundamental component of every Graphical User Interface system is, of course, graphics. This chapter presents *Haven*, a library for 2D vector graphics in Haskell. Haven provides the graphical primitives that form the basis of the Fruit GUI toolkit.

What is Vector Graphics? Most 2D graphics libraries provide a *bitmap* or *raster* model of graphics. The raster model of graphics implicitly assumes that the rendering area is comprised of a discrete two-dimensional array of *pixels* of finite size. While adequate for many tasks, the raster model has a number of substantial flaws. Most notably, the model is *resolution dependent*: programs implicitly depend on a particular hardware screen resolution (pixels per unit area). Changing the screen resolution will result in either the graphics being rendered at the wrong size or in undesirable rendering artifacts such as "jaggies" – jagged edges that appear on what should be rendered as a spatially continuous curve.

An alternative to the raster model of graphics is *vector* graphics. In the vector graphics model, geometric primitives are represented by *paths* that capture the idealized geometry of straight or curved shapes that the program wishes to display. The units used in the model are resolution-independent *points* that correspond directly with continuous physical spatial units (mm, inches, etc.) instead of discrete numbers of screen pixels. The most well-known instances of the vector graphics model are the Adobe PostScript language [3] and PDF file format [4]. The vector graphics model is also used by many popular drawing tools, such as Adobe Illustrator [2]) and some APIs, such as Java2D [29] and MacOS X's Quartz [5].

Why Vector Graphics? One of the goals of Fruit is to provide an abstract formal model of GUIs that is independent of any particular implementation. The primary advantage of vector graphics over raster graphics is that resolution independence makes the model independent of the underlying display hardware and rendering implementation.

6.1.1 A Functional Model of Vector Graphics

As noted, there are already libraries available that provide a vector model of graphics. However, all of these libraries are implemented in imperative programming languages, such as C or Java, and present a programming interface heavily biased towards the imperative programming model. For example, most vector graphics libraries use global mutable state to maintain various rendering parameters, such as the current transform or current paint color.

Since we wish to provide an implementation-independent model of GUIs, we want an underlying graphics model in which images to be displayed are simply values in a suitable semantic domain. To meet this goal, we developed a *functional* model of vector graphics.

Of course, there are many possible models we could have developed. Since our primary aim was to provide a model with a simple, precise denotational semantics, every effort was made to distill the model down to a small set of simple yet general primitives that can be composed to define many common visual effects. So, for example, the Haven model does not provide a single primitive for producing an image of a shape filled with a particular solid color. Instead, this is expressed by applying the $crop$ function to an infinite $monochrome$ image of the desired color. The advantage of this approach is that $crop$ is a much more general operation than a "fill" operator, and may be applied to sampled bitmap images, linearly interpolated color gradients, other cropped images, etc.

In the world of 2D image synthesis for functional languages, Pan [22] provides a concise, general model of spatially continuous 2D images. Pan is based on the idea of providing the programmer direct access to the functional model of images as functions of type $Point \rightarrow Color$. The programmer can specify images simply by defining and composing functions of this type. Pan is implemented using a staged compilation technique, in which a definition of a Pan image (written in Haskell) is used to generate a C program, which in turn renders the desired image.

Unfortunately, Pan's *off-line* implementation model makes it unsuitable for direct use as the graphics layer of an interactive GUI toolkit. Furthermore, although Pan provides spatially continuous *images* and a basic region algebra, there are some important aspects of the vector graphics model (such as computing a tight bounding rectangle of an arbitrary region) that Pan does not support.

Haven is a Haskell library that provides a functional model of 2D vector graphics geometry and images suitable for interactive use. Unlike Pan, Haven can be implemented efficiently without resorting to a distinct code generation or compilation step during image rendering. However, Haven sacrifices some of Pan's generality and expressive power to achieve this. While Pan allows the programmer to specify images and regions directly in terms of their denotation as functions, Haven provides a more limited set of primitives for composing images and regions from outline paths and constructive area geometry.

This chapter presents the functional programming interface to the Haven library. In many cases Haskell is used as a meta-language to specify the semantics of images and regions. However, these are reference definitions only. In all cases, we have chosen to give definitions that emphasize simplicity and clarity over concerns about efficiency. Haven appeals to Pan's semantic model wherever possible, but makes certain aspects of this model abstract and augments the model where necessary to offer support for paths and rectilinear layouts.

6.2 Basic Concepts

This section presents the basic types and functions that form the core of the Haven library.

6.2.1 Points and Colors

We use Pan's definitions of points and colors:

$$\mathbf{type}\ Point = (Float, Float)$$
$$\mathbf{type}\ Color = (Frac, Frac, Frac, Frac) \quad \text{-- RGBA}$$

A *Point* is an ordered pair of real numbers that identifies a point in continuous 2D space. A *Color* is a quadruple giving the intensities of each color channel, including an *alpha* channel representing opacity. As in Pan, we assume $Frac$ represents a real number on the interval $[0, 1]$, and use pre-multiplied alpha values [70]. We give the following definitions of some useful values and functions on points and colors:

$$
\begin{aligned}
&\mathit{origin} = (0, 0)\\
&\mathit{ptX}\ (x, \_) = x\\
&\mathit{ptY}\ (\_, y) = y\\
&\mathit{invisible} :: \mathit{Color}\\
&\mathit{invisible} = (0, 0, 0, 0)\\
&\quad \text{-- blend two colors, using alpha compositing rules:}\\
&\mathit{colorBlend} :: \mathit{Color} \to \mathit{Color} \to \mathit{Color}\\
&\mathit{colorBlend}\ (\mathit{r1}, \mathit{g1}, \mathit{b1}, \mathit{a1})\ (\mathit{r2}, \mathit{g2}, \mathit{b2}, \mathit{a2}) =\\
&\quad (h\ \mathit{r1}\ \mathit{r2}, h\ \mathit{g1}\ \mathit{g2}, h\ \mathit{b1}\ \mathit{b2}, h\ \mathit{a1}\ \mathit{a2})\\
&\quad \mathbf{where}\\
&\qquad h\ \mathit{x1}\ \mathit{x2} = \mathit{x1} + (1 - \mathit{a1}) * \mathit{x2}
\end{aligned}
$$

6.2.2 Regions and Region Algebra

A *region* is a set of points. Again following Pan, Haven uses the standard identification of regions with their characteristic functions [35, 22]:

$$\mathbf{type}\ \mathit{Region} = \mathit{Point} \to \mathit{Bool} \quad \text{-- abstract}$$

However, Region is an *abstract* type. Values of abstract types can not be constructed or used directly; instead, the implementation provides a finite set of functions for creating and manipulating values of the type.[1]

The most fundamental operation on regions is testing whether a Point is in a region:

$$
\begin{aligned}
&\mathit{inRegion} :: \mathit{Point} \to \mathit{Region} \to \mathit{Bool}\\
&\mathit{inRegion}\ \mathit{pt}\ r = r\ \mathit{pt}
\end{aligned}
$$

A primitive Region is constructed by specifying a Path that specifies the outline of the region. The functions for defining paths and obtaining regions from paths will be described in section 6.3.

[1] In Haskell, abstract types are implemented by declaring a **newtype**, and then only exporting the type name but not the constructor from the implementation module.

Standard set operations are provided for defining new regions from existing regions:

$$
\begin{aligned}
&\mathit{regUnion} :: \mathit{Region} \rightarrow \mathit{Region} \rightarrow \mathit{Region}\\
&\mathit{regUnion}\ \mathit{r1}\ \mathit{r2} = \lambda pt \rightarrow (\mathit{r1}\ pt) \vee (\mathit{r2}\ pt)\\
&\mathit{regIntersect} :: \mathit{Region} \rightarrow \mathit{Region} \rightarrow \mathit{Region}\\
&\mathit{regIntersect}\ \mathit{r1}\ \mathit{r2} = \lambda pt \rightarrow (\mathit{r1}\ pt) \wedge (\mathit{r2}\ pt)
\end{aligned}
$$

A couple of useful regions are the empty and infinite regions:

$$
\begin{aligned}
&\mathit{regEmpty} :: \mathit{Region}\\
&\mathit{regEmpty} = \mathit{const}\ \mathit{False}\\
&\quad \text{-- the infinite region:}\\
&\mathit{regInf} :: \mathit{Region}\\
&\mathit{regInf} = \mathit{const}\ \mathit{True}
\end{aligned}
$$

6.2.3 Images

As in Pan, an *Image* is a function from continuous 2D space to colors. However, we extend this basic definition with an optional *bounding frame* that delimits a region of interest within the image.

$$\mathbf{type}\ \mathit{Image} = (\mathit{Point} \rightarrow \mathit{Color}, \mathit{Region}) \quad \text{-- abstract}$$

As with regions, Image is *abstract*. In this section, we present a few simple image constructors and some basic operations on images. Later we will see how to construct images from paths to produce filled shapes and other common effects.

A useful operation on an Image is to recover its bounding frame:

$$
\begin{aligned}
&\mathit{imgFrame} :: \mathit{Image} \rightarrow \mathit{Region}\\
&\mathit{imgFrame}\ (\_, \mathit{frame}) = \mathit{frame}
\end{aligned}
$$

Bounding frames will be used later when composing images spatially, and are also used in the definition of image sampling.

Sampling is perhaps the most fundamental operation on an Image. Images are considered to be transparent when sampled outside of their frame:

$$
\begin{aligned}
&\mathit{imgSample} :: \mathit{Image} \rightarrow \mathit{Point} \rightarrow \mathit{Color}\\
&\mathit{imgSample}\ (\mathit{imgD}, \mathit{frame})\ pt =
\end{aligned}
$$

$$\textbf{if } (pt \text{ `}inRegion\text{` } frame) \textbf{ then } imgD\ pt \textbf{ else } invisible$$

The simplest image, of course, is the empty image:

$$imgEmpty :: Image$$
$$imgEmpty = (\bot, regEmpty)$$

Despite the use of $\bot$ for the first (function) component of $imgEmpty$, the definition of $imgSample$ ensures that sampling the empty image is always well-defined, since $imgEmpty$'s bounding frame ($regEmpty$) ensures that the the predicate tested in the body of $imgSample$ is always $False$.

A very basic $Image$ constructor is *imgMonochrome*, which produces an infinite image of a single color:

$$imgMonochrome :: Color \rightarrow Image$$
$$imgMonochrome\ c = (const\ c, regInf)$$

6.3 Geometry

Haven provides a small set of primitives for composing regions from straight and curved segments. This section presents the geometry primitives used to define regions, along with some other standard geometry types (such as rectangles) that are useful for layout.

6.3.1 Paths

Paths provide a very general notion of 2D geometry based on outlines. These outlines may be composed of straight line segments, smoothly curved segments, or some combination. Paths may be concave, convex, or have arbitrarily complicated self-intersections.

More precisely, a $Path$ is a set of (disconnected) *sub-paths*:

$$\textbf{data } Path = Path\ [SubPath]$$

Note that although we are using the Haskell convention of representing the sub-paths in a $Path$ as a list, this is really a set; ordering of sub-paths within a path is not significant.

A $SubPath$ always begins at a $Point$ and consists of a sequence of straight or curved segments:

$$\mathbf{data}\ SubPath = SubPath\ Point\ [Segment]$$

A $Segment$ is either a straight line segment, a second-order (quadratic) curved segment, or a cubic Beziér curve segment:

$$\begin{aligned}&\mathbf{data}\ Segment = LineTo\ Point\\ &\quad\mid QuadTo\ Point\ Point\\ &\quad\mid CubicTo\ Point\ Point\ Point\end{aligned}$$

Each segment begins at a starting point, P_0. In the representation used here, this starting point is not an explicit part of the $Segment$. Instead, the starting point for a segment is either the starting point of the $SubPath$ (for the first segment in the sub-path),or the endpoint of the previous $Segment$ in the $SubPath$. The $QuadTo$ constructor includes an off-segment *control point* that affects the shape of the curve; the $CubicTo$ constructor uses two such control points. In all constructors, the final $Point$ argument gives the endpoint of the segment, which is always considered to lie on the segment.

A $SubPath$ may intersect itself in arbitrary ways, and may be *open* or *closed*. A $SubPath$ is said to be *closed* if its starting point and the endpoint of its final segment are the same.

Haven provides a number of convenience routines for specifying paths constructively, with the following interface:

$$\begin{aligned}&pathEmpty :: Path\\ &pathEmpty = Path\ [\,]\\ &pathMoveTo :: Path \to Point \to Path\\ &pathMoveTo\ (Path\ sps)\ pt = Path\ ((SubPath\ pt\ [\,]) : sps)\end{aligned}$$

```
pathLineTo :: Path → Point → Path
pathLineTo (Path []) _ = error "pathLineTo without preceding moveto"
pathLineTo (Path ((SubPath p0 segs) : sps)) pt =
  Path ((SubPath p0 (segs ++ [LineTo pt])) : sps)
pathClose :: Path → Path
pathClose (Path []) = error "pathClose on non-open path"
pathClose (Path ((SubPath p0 segs) : sps)) =
  Path ((SubPath p0 (segs ++ [LineTo p0])) : sps)
pathQuadCurveTo :: Path → Point → Point → Path
pathQuadCurveTo (Path []) _ _ =
  error "pathQuadCurveTo without preceding moveto"
pathQuadCurveTo (Path ((SubPath p0 segs) : sps)) cpt pt =
  Path ((SubPath p0 (segs ++ [QuadTo cpt pt])) : sps)
pathCubicCurveTo :: Path → Point → Point → Point → Path
pathCubicCurveTo (Path []) _ _ _ =
  error "pathCubicCurveTo without preceding moveto"
pathCubicCurveTo (Path ((SubPath p0 segs) : sps)) cpt0 cpt1 pt =
  Path ((SubPath p0 (segs ++ [CubicTo cpt0 cpt1 pt])) : sps)
```

6.3.2 Rectangles

Although most common shapes can be precisely represented as paths or sub-paths, it is sometimes useful to know that a sub-path represents a shape that satisfies some well-defined set of invariants. Haven provides explicit representations for such geometric types, and makes it is easy to project values of such types into their corresponding path representation.

One such type that Haven defines is the *Rectangle*. A *Rectangle* is defined as a *Point* that specifies the upper-left corner of the rectangle, along with measures of the rectangle's *width* and *height* (measured in resolution-independent *points*).

```
type Rectangle = (Point, Dist, Dist)
```

The edges of a rectangle always lie parallel to the x- and y-axes.

6.3.3 Paths, Regions and Bounding Rectangles

An essential aspect of the vector graphics model is that there is a close relationship between regions and paths. With the exception of the infinite and empty regions, all other regions are defined as the interior of a closed path, or derived from set operations applied to other regions.

The interior region of a closed path is determined with $pathInterior$:

$$pathInterior :: Path \rightarrow Region$$

The algorithm used to implement $pathInterior$ is not specified here. It is expected that an implementation will use the standard "Nonzero Winding Number Rule" [4]. This algorithm is based on the idea of extending a ray from each point to infinity, examining the places where the ray intersects segments of the path, and considering the direction of the particular segment at each intersection.

The close relationship between paths and regions enables the implementation to provide access to the outline of any finite region:

$$regOutline :: Region \rightarrow Maybe\ Path$$

For any finite region r, $regOutline\ r$ returns $Just\ p$, where p is a $Path$ that specifies the outline of the region. The value $Nothing$ is returned for the infinite region.

Note that some set operations involving the infinite region may yield finite regions, while others will not. For example, for any path p, the implementation must obey:

$$regOutline\ (pathInterior\ p\ `regIntersect`\ regInf) \equiv Just\ p$$

On the other hand, subtracting a finite region from an infinite region is still an infinite region:

$$regOutline\ (regInf\ `regDiff`\ pathInterior\ p) \equiv Nothing$$

Finally, another important and useful operation on paths is computing a tight *bounding rectangle* (or *bounds*) for a path:

$$pathBounds :: Path \rightarrow Rectangle$$

The result of $pathBounds\ p$ for any path p is the smallest rectangle r whose interior contains all points that are in the interior of p.

We can easily define a function to obtain the bounds of an image's bounding frame, if the bounding frame is not infinite:

$$
\begin{aligned}
&imgBounds :: Image \rightarrow Maybe\ Rectangle \\
&imgBounds\ img = fmap\ pathBounds\ ((regOutline \circ imgFrame)\ img)
\end{aligned}
$$

6.3.4 Shapes

Specialized geometric types such as $Rectangle$ determine a corresponding $Path$, every $Path$ determines a corresponding $Region$, and every non-infinite $Region$ determines a $Path$.

While precise, it would be somewhat inconvenient to force the user to, for example, coerce a $Rectangle$ into a $Path$ before drawing its outline. For convenience, Haven uses the $Shape$ type class to represent values that determine regions:

$$
\begin{aligned}
&\textbf{class}\ Shape\ a\ \textbf{where} \\
&\quad region :: a \rightarrow Region \\
&\quad outline :: a \rightarrow Maybe\ Path
\end{aligned}
$$

There are suitable instances of $Shape$ for specialized geometric types such as $Rectangle$, as well as for $Path$ and $Region$:

$$
\begin{aligned}
&\textbf{instance}\ Shape\ Rectangle\ \textbf{where} \\
&\quad outline = Just \circ rectPath \\
&\quad region = rectRegion \\
&\textbf{instance}\ Shape\ Path\ \textbf{where} \\
&\quad outline = Just \\
&\quad region = pathInterior \\
&\textbf{instance}\ Shape\ Region\ \textbf{where} \\
&\quad region = id \\
&\quad outline = regOutline
\end{aligned}
$$

For convenience, most Haven primitives that require a $Region$ or $Path$ are defined to take any $Shape$.

6.3.5 Transforms

Haven provides a $Transform$ type that represents a mapping from points to points:

$$\textbf{type}\ Transform = (Point \rightarrow Point, Point \rightarrow Point) \quad \text{-- abstract}$$

A $Transform$ is represented with a pair of functions: the transform itself and its *inverse*. Haven restricts transforms to *affine* transforms (any combination of rotate, translate, scale and shear operations):

$$
\begin{aligned}
&scale :: Double \rightarrow Double \rightarrow Transform\\
&shear :: Double \rightarrow Double \rightarrow Transform\\
&translate :: Vector \rightarrow Transform\\
&\quad \text{-- rotate by a given angle}\\
&rotate :: Double \rightarrow Transform\\
&\quad \text{-- rotate by a given angle about a certain point:}\\
&rotateAbout :: Double \rightarrow Point \rightarrow Transform
\end{aligned}
$$

Restricting transforms to affine transforms ensures that every transform has an inverse:

$$
\begin{aligned}
&inverse :: Transform \rightarrow Transform\\
&inverse\ (t, invt) = (invt, t)
\end{aligned}
$$

Transforms can be composed with $compTransform$:

$$
\begin{aligned}
&compTransform :: Transform \rightarrow Transform \rightarrow Transform\\
&compTransform\ (t1, it1)\ (t2, it2) = (t1 \circ t2, it2 \circ it1)
\end{aligned}
$$

Since there are many types for which it makes sense to apply transforms, Haven provides a $Transformable$ type class:

$$
\begin{aligned}
&\textbf{class}\ Transformable\ a\ \textbf{where}\\
&\quad (\%\$) :: Transform \rightarrow a \rightarrow a
\end{aligned}
$$

Instances of $Transformable$ are defined for all of the standard types:

$$
\begin{aligned}
&\textbf{instance}\ Transformable\ Point\ \textbf{where}\\
&\quad (\%\$) = pointTransform
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Transformable\ Path} \textbf{ where} \\
\quad (\%\$) = \mathit{pathTransform} \\
\textbf{instance } \mathit{Transformable\ Image} \textbf{ where} \\
\quad (\%\$) = \mathit{imgTransform}
\end{array}
$$

The definition of $pointTransform$ simply applies the transform function to the given $Point$. Transforming a $Path$ just involves mapping $pointTransform$ over all points in the path. As in Pan, image transformation simply involves applying the inverse transform to a sample point before sampling the image.

6.4 Rendering Model

The previous sections defined Haven's model of images and path-based geometry. This section presents Haven's operator for composing and cropping images and for stroking paths. Haven differs from other vector graphics languages and libraries in its emphasis on providing a minimal, general set of primitives that can be composed to define more complex images.

6.4.1 Composition Operators

Images can be composed by placing one image over another with the *over* operator:

$$
\begin{array}{l}
\mathit{imgOver} :: \mathit{Image} \rightarrow \mathit{Image} \rightarrow \mathit{Image} \\
\mathit{imgOver\ imgA\ imgB} = \\
\quad (\lambda pt \rightarrow \mathit{colorBlend}\ (\mathit{imgSample\ imgA\ pt})\ (\mathit{imgSample\ imgB\ pt}), \\
\qquad \mathit{imgFrame\ imgA}\ \text{`}\mathit{regUnion}\text{`}\ \mathit{imgFrame\ imgB})
\end{array}
$$

Note that in the above definition, the frame of the resulting image is the union of the frames of the images being composed.

(a) Crop mask (closed Beziér path) (b) Monochrome Image (c) Cropped Image

Figure 6.1: Cropping a Monochrome Image to a Path

6.4.2 Cropping Images

Most vector graphics languages or libraries provide a primitive operation for filling a path in some particular color. Haven does not provide such a primitive, although the operation is provided as a utility function (for programmer convenience). Instead, Haven provides a general purpose *crop* operator, which may be used to restrict an image to a particular shape (the *crop mask*):

$$imgCrop :: (Shape\ a) \Rightarrow a \rightarrow Image \rightarrow Image$$

Cropping of a monochrome image with $imgCrop$ is illustrated in figure 6.1. The original monochrome image in figure 6.1(b) has an infinite bounding frame. However, after applying the crop operation, the resulting image will have a bounding frame that corresponds to the region determined by the path in figure 6.1(a), as illustrated by the dashed line in figure 6.1(c).

How, then, should we define $imgCrop$? In particular, what is the relationship between the crop mask and the source image's bounding frame? Two requirements seem natural:

- If the crop mask is entirely contained in (i.e. is a strict subset of) the frame of the image being cropped, then the frame of the resulting image should be the crop mask:

$$imgFrame\ (imgCrop\ m\ img) \equiv m$$

- If the image frame is a subset of the crop mask, then:

$$imgCrop\ m\ img \equiv img$$

These two requirements are satisfied by defining the result image's bounding frame as the intersection of the crop mask and source image's bounding frame:

$$imgCrop\ s\ (imgD, iframe) = (imgD, region\ s\ \text{`}regIntersect\text{`}\ iframe)$$

Although Haven does not provide "path fill" as a primitive operation, it is easy to define this using $imgCrop$ and $imgMonochrome$:

$$imgFill :: (Shape\ a) \Rightarrow Color \rightarrow a \rightarrow Image$$
$$imgFill\ c\ s = imgCrop\ s\ (imgMonochrome\ c)$$

6.4.3 Pens and Stroking

Another important operation in vector graphics is drawing the outline of a path, called *stroking* a path.

Haven supports the standard collection of attributes for describing how the outline of a path should be rendered. These attributes are collected in the Pen data type:

```
  -- join styles:
data Join = JoinMiter | JoinBevel | JoinRound
  -- end caps:
data Cap = CapButt | CapRound
  -- A pen specifies how the outline of a path is rendered:
data Pen = Pen{penWidth :: Float,
  penCap :: Cap,
  penJoin :: Join,
  penMiterLimit :: Float
  }
```

These attributes are the standard attributes used in the PostScript and PDF rendering model. The *pen width* is the width of the pen, measured perpendicular to the path being stroked. The *end caps* describes the decoration applied to unclosed subpaths. The *join style* describes the decoration applied at the intersection of two segments in a subpath. The *miter limit* specifies how much to trim the decoration in a $JoinMiter$ join style.

Haven's provides one primitive operation to stroke a path:

$stroke :: (Shape\ a) \Rightarrow Pen \rightarrow a \rightarrow Path$

The behavior of stroke is illustrated in figure 6.2. Using a pen (figure 6.2(a)) to stroke a path (figure 6.2(b)) yields a new path whose interior is the result of sweeping the pen along the path according to the pen attributes (figure 6.2(c)). This resulting path may be used with any other operation on paths or shapes. Figure 6.2(d) illustrates the result of applying $imgFill$ to the path from figure 6.2(c).

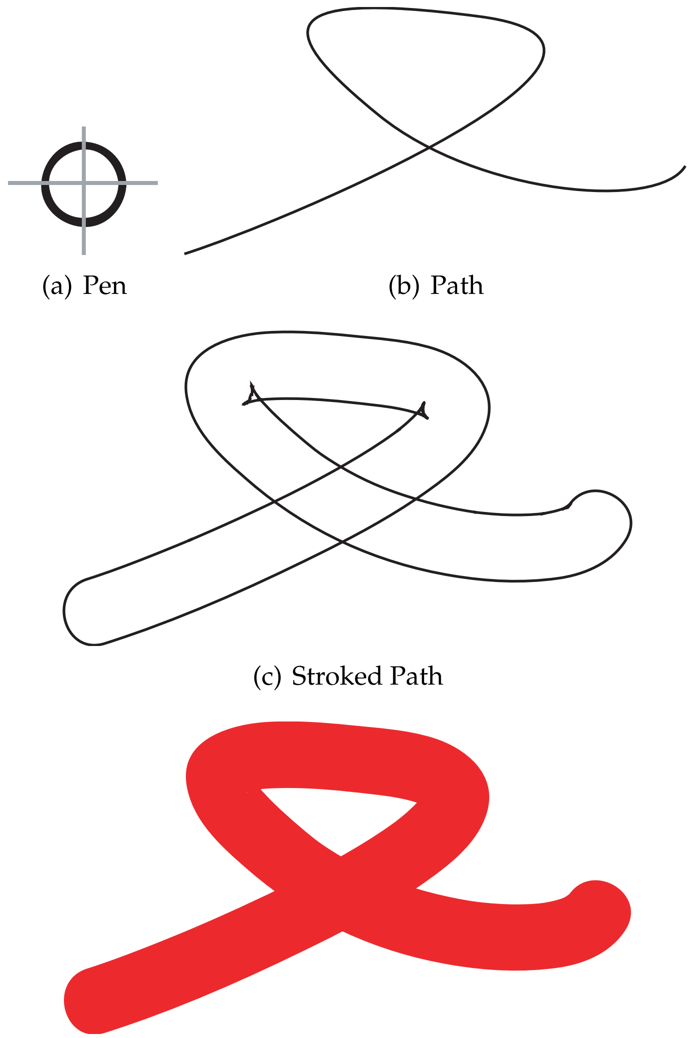
6.4.4 Fonts and Text

Haven provides access to *fonts*, which specify how to map strings to a visual representation. Fonts are identified by the font family name (a String), a *style* (interpreted as a set of style attributes), and a point size:

$\mathbf{data}\ StyleAttr = Bold \mid Italic$
$\mathbf{type}\ Style = [StyleAttr]$ -- interpreted as a set
$plain :: Style$
$plain = [\,]$
$\mathbf{data}\ Font = Font$
$font :: String \rightarrow Style \rightarrow Int \rightarrow Font$

For example, here is the definition of the default font:

$fontDefault :: Font$
$fontDefault = font$ `"SansSerif"` $plain\ 16$



(a) Pen (b) Path

(c) Stroked Path

(d) Filled, Stroked Path

Figure 6.2: Stroking a Path with a Pen

The exact behavior of the $font$ function is not specified, although it is expected to be a total function, returning some $Font$ value for any arguments.

The $textShape$ primitive is used for rendering text:

$$textShape :: Font \rightarrow String \rightarrow Path$$

The $textShape$ function returns a $Path$ representing the outline of the given $String$ rendered in the given $Font$. This path may be used with $imgCrop$, $imgStroke$, or any other function that operates on paths.

6.5 Layout

The Haven primitives presented thus far are adequate for producing a wide variety of vector graphics images. However, these primitives are not particularly convenient for composing images or geometric shapes to form larger images.

One very common pattern for composing images, particularly in graphical user interfaces, is a *rectilinear* layout: The bounding rectangles of images are used to arrange images horizontally or vertically.

Haven provides a set of utility combinators for composing paths or images into rectilinear layouts. Types that can be laid out (including $Path$ and $Image$) are instances of the $HasBounds$ and $Placeable$ type classes.

The $HasBounds$ type class is used for types for which a bounding rectangle can be determined:

```
class HasBounds a where
  bounds :: a → Rectangle
```

Suitable instances of $HasBounds$ are defined for $Path$ and $Rectangle$. Although we technically can not give an instance for $Image$ (since some images, such as those returned by $imgMonochrome$, may be infinite in extent), we can easily imagine a $BoundedImage$ type which is identical to $Image$ except that it requires the image to have a bounding frame. The current implementation of Haven doesn't provide such a $\mathit{BoundedImage}$ type. Instead, an instance of $\mathit{HasBounds}$ is provided for the Image type which will raise a runtime error if bounds is applied to an infinite image.

The only Haven primitive for combining two images is $\mathit{imgOver}$, defined in section 6.4.1. As defined, this function is both associative and has a left and right identity ($\mathit{imgEmpty}$), and hence forms a monoid. This fact is captured in the $\mathit{HMonoid}$ type class:

```
class HMonoid a where
  emptyPlaceable :: a       -- monoid unit
  (<++>) :: a → a → a       -- monoid compose
overs :: (HMonoid a) ⇒ [a] → a
overs = foldr (<++>) emptyPlaceable
instance HMonoid Image where
  emptyPlaceable = imgEmpty
  (<++>) = imgOver
```

A suitable instance of $\mathit{HMonoid}$ is also given for Path. The <++> operator for a Path is simply the union of all sub-paths in both arguments, and the unit is the empty path.

The type class $\mathit{Placeable}$ is defined for all types that support rectilinear layout:

```
class (HMonoid a, Transformable a) ⇒ Placeable a where
    -- translate a Placeable value so that the upper left corner
    -- of its bounding rectangle is located at the given Point
  place :: Point → a → a
    -- adjust a Placeable so that it is positioned on the given abscissa
  xPlace :: Double → a → a
    -- adjust a Placeable so that it is positioned on the given ordinate:
  yPlace :: Double → a → a
    -- compose two Placeables vertically, normalizing the result:
  vcomp :: a → a → a
    -- compose two Placeables horizontally, normalizing the result:
  hcomp :: a → a → a
```

A common instance of $Placeable$ is provided for all types that are instances of $HasBounds$, $Transformable$ and $HMonoid$ (such as $Path$ and $Image$):

$$
\begin{array}{l}
\textbf{instance}\ (HasBounds\ a, Transformable\ a, HMonoid\ a) \Rightarrow Placeable\ a\ \textbf{where} \\
\quad place = bPlace \\
\quad xPlace = bXPlace \\
\quad yPlace = bYPlace \\
\quad hcomp = bHComp \\
\quad vcomp = bVComp
\end{array}
$$

This use of the Haskell type class system enables us to avoid repeating the implementation of the $Placeable$ type class members for paths, images, and other types. Instead, we just need to define the much simpler $HasBounds$, $Transformable$ and $HMonoid$ instances for each type. For example, here is the common implementation of the $place$ function that is used for all of these derived instances:

$$
\begin{array}{l}
bPlace :: (HasBounds\ a, Transformable\ a, HMonoid\ a) \Rightarrow Point \rightarrow a \rightarrow a \\
bPlace\ pt\ obj = \\
\quad \textbf{let}\ offset = pt\ .-.\ rectPointA\ (bounds\ obj) \\
\quad \textbf{in}\ (translate\ offset)\ \%\$\ obj
\end{array}
$$

6.6 From Images to Pictures: A Rendering Monad

While the Haven primitives and utility routines presented thus far offer a simple, explicit interface to vector graphics, using these functions directly can be somewhat cumbersome. This is because the values of some function arguments (such as font, color, etc.) will have the same value in many places in the specification of a typical image.

One solution to this problem is to use an *environment monad* [76] to capture the common argument values. In an environment monad, common argument values are stored in a record type (the environment) which is hidden behind a monadic interface. For each function that depends on one of the standard argument values, a "lifted" version of the function is provided that has a monadic type. In the lifted form of the function, the explicit argument is omitted, and is instead taken from the environment monad. Wrapping the Haven functions in an environment monad sacrifices some of the precise typing of the primitives for syntactic convenience.

The rendering attributes (environment) provided in the rendering monad are:

$$
\begin{array}{l}
\textbf{data}\ RA = RA\{ raFont :: Font, \\
\quad raColor :: Color, \\
\quad raPen :: Pen \\
\quad \}
\end{array}
$$

The Rendering monad is then defined as a standard environment monad:

$$\textbf{type}\ RenderM\ a = EnvM\ RA\ a$$

A $Picture$ is defined as an Image value constructed using the rendering monad:

$$\textbf{type}\ Picture = RenderM\ Image$$

Values can be extracted from the rendering monad by providing an explicit set of rendering attributes:

$$
\begin{array}{l}
renderWith :: RA \rightarrow RenderM\ a \rightarrow a \\
renderWith\ ra\ pm = applyEnvM\ pm\ ra
\end{array}
$$

More common, however, is to simply use the global constant default rendering attributes:

$$
\begin{array}{l}
render :: RenderM\ a \rightarrow a \\
render = renderWith\ raDefault
\end{array}
$$

The "lifted" versions of various Haven primitives may omit parameters that can be obtained from the environment. For example:

$$
\begin{array}{l}
picMonochrome :: Picture \\
picMonochrome = \\
\quad \textbf{do}\ ra \leftarrow rmGetRA \\
\qquad return\ \$\ imgMonochrome\ (raColor\ ra)
\end{array}
$$

Similarly, many of the derived utility routines that operate over images have corresponding definitions in the context of pictures:

$$
\begin{array}{l}
picFill :: (Shape\ a) \Rightarrow a \rightarrow Picture \\
picFill\ s = picCrop\ s\ picMonochrome
\end{array}
$$

$$
\begin{aligned}
&\mathit{picOutline} :: (\mathit{Shape}\ a) \Rightarrow a \rightarrow \mathit{Picture}\\
&\mathit{picOutline}\ s = \mathit{rmStroke}\ s \ggg\!= \mathit{picFill}
\end{aligned}
$$

Functions are provided to explicitly specify values in the environment:

$$
\begin{aligned}
&\mathit{withFont} :: \mathit{Font} \rightarrow \mathit{RenderM}\ a \rightarrow \mathit{RenderM}\ a\\
&\mathit{withFont}\ f = \mathit{rmUpdateRA}\ (\lambda ra \rightarrow ra\{\mathit{raFont} = f\})
\end{aligned}
$$

This allows us to write, for example:

$$
\begin{aligned}
&\quad \text{-- font specified implicitly:}\\
&\mathit{test1} :: \mathit{Picture}\\
&\mathit{test1} = \mathit{picText}\ \texttt{"Hello, Haven!"}\\
&\quad \text{-- explicit font:}\\
&\mathit{test2} :: \mathit{Picture}\\
&\mathit{test2} =\\
&\quad \mathbf{let}\ f = \mathit{font}\ \texttt{"SansSerif"}\ (\mathit{bold}\ .|.\ \mathit{italic})\ 32\\
&\quad \mathbf{in}\ \mathit{withFont}\ f\ \mathit{test1}
\end{aligned}
$$

6.7 Examples

Finally, we present a few examples to illustrate the kinds of images that can be produced with Haven.

6.7.1 Sierpinski Gasket

Hudak's "The Haskell School of Expression" presents a rendering of the fractal "Sierpinski Gasket" as a demonstration of recursion. That version used the imperative "Draw" monad to render the image directly on to the screen. Here we use Haven's geometry operations to compose a path that describes the geometry of the gasket, which can then be used to produce the desired image:

First, we define a simple function to produce an equilateral right triangle:

$$
\begin{aligned}
&\mathit{rightTri} :: \mathit{Point} \rightarrow \mathit{Double} \rightarrow \mathit{Path}\\
&\mathit{rightTri}\ \mathit{pt0}\ \mathit{size} =\\
&\quad \mathbf{let}\ (x, y) = \mathit{pointXY}\ \mathit{pt0}\\
&\quad \mathbf{in}\ \mathit{polygon}\ [\mathit{pt0}, \mathit{point}\ (x + \mathit{size})\ y, \mathit{point}\ x\ (y - \mathit{size})]
\end{aligned}
$$

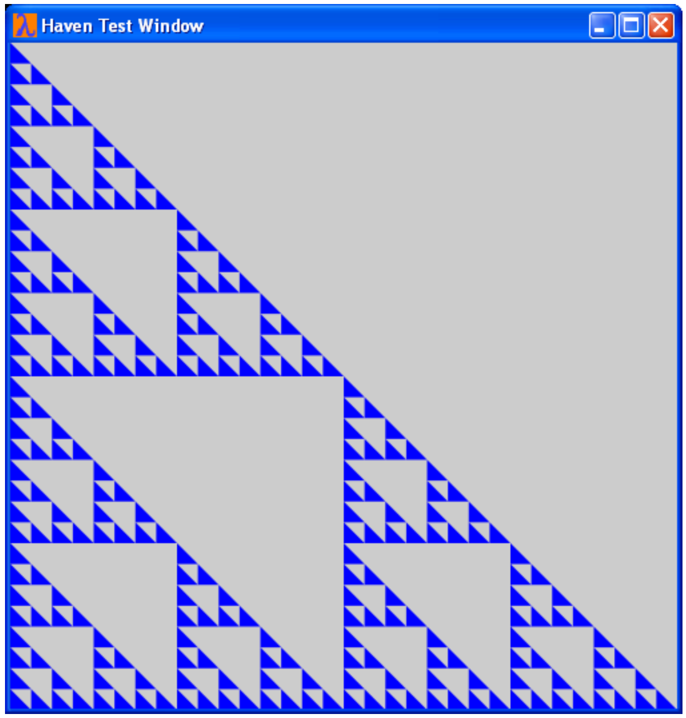


Figure 6.3: The Sierpinski Gasket

Using this definition, the path of the Sierpinski triangle is easily defined:

```haskell
sierpinskiTri :: Point -> Double -> Path
sierpinskiTri pt size =
  if size <= minSize
  then rightTri pt size
  else let size2 = size / 2
           (x, y) = pointXY pt
           t1 = sierpinskiTri pt size2
           t2 = sierpinskiTri (point x (y - size2)) size2
           t3 = sierpinskiTri (point (x + size2) y) size2
       in t1 <++> t2 <++> t3
```

To produce a complete image, we produce a filled version of the resulting path in some color:

```haskell
sierpinski :: Picture
sierpinski =
  let spath = sierpinskiTri (point 250 250) 500
  in place origin $ withColor blue $ picFill spath
```

The resulting picture is shown in figure 6.3

6.7.2 A Logo for Haven

We can use the Sierpinski Gasket from the previous section as a component of a more complicated picture that demonstrates many of Haven's features, including alpha-blending, gradients, high quality fonts and text effects.

Here is a small program to produce a logo for Haven:

```
-- produce a filled outline shape, using lpen:
outFillShape :: (Shape a) => a -> Picture
outFillShape s =
  (withColor black $ withPen lpen $ picOutline s) <++>
  (picFill s)
-- a partially transparent circle:
cpic = withAlpha 0.5 $ withColor yellow $
  outFillShape (circle origin 150)
rectPic = withAlpha 0.75 $ withColor red $
  outFillShape (rectangle origin 160 75)
bodyPic = withAlpha 0.75 $ sierBGPic2
tpslide = (xyPlace 160 20 ttext
   <++> decor
   <++> xyPlace 300 180 rectPic
   <++> xyPlace 220 150 cpic
   <++> xyPlace 160 100 btext
   <++> xyPlace 225 125 bodyPic)
logo = tpslide <++> bg
```

The output of the picture *logo* is shown in figure 6.4.

6.8 Chapter Summary

This chapter has presented *Haven*, a functional library for producing scalable vector graphics images in Haskell. Haven provides a clear separation between *images* and *geometry*, and defines a small, statically typed, compositional set of rendering primitives. This small kernel of primitives is augmented by utility libraries for producing common rectilinear layouts and a monadic interface that provides

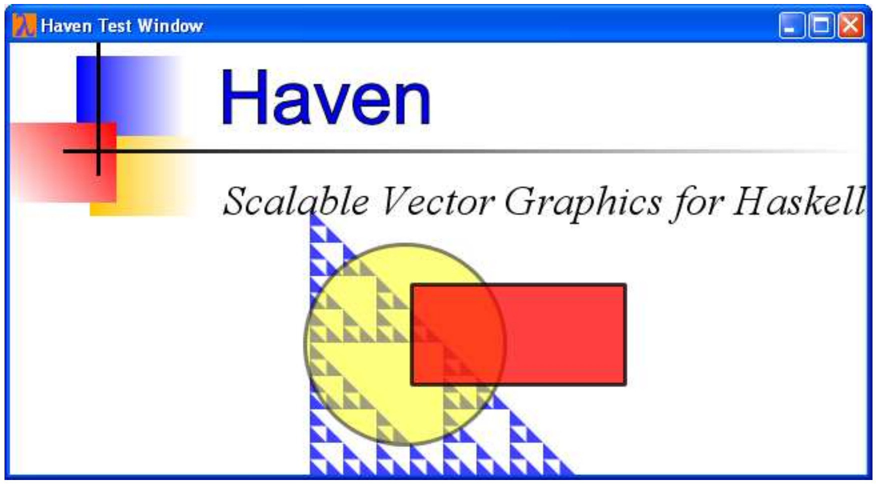


Figure 6.4: A Logo for Haven

syntactic convenience.

Chapter 7

Fruit: A Functional GUI Library

This chapter defines *Fruit*, a Functional Reactive User Interface Toolkit. Fruit is a library for composing graphical user interfaces in Haskell. Fruit makes no appeal to the IO monad or other imperative programming constructs, relying solely on the dataflow framework of Yampa developed in chapters 2 and 4, and the functional graphics model of chapter 6.

7.1 Defining GUIs

Fruit is a modest library of types and functions for specifying graphical user interfaces using Yampa. To illustrate the essence of composing a Fruit specification, consider the following type:

$$\textbf{type}\ \mathit{SimpleGUI} = \mathit{SF}\ \mathit{GUIInput}\ \mathit{Picture}$$

The $\mathit{GUIInput}$ type represents an instantaneous snapshot of the keyboard and mouse state (formally just a tuple or record). The $\mathit{Picture}$ type denotes a single, static visual image.

A $\mathit{SimpleGUI}$, then, is a signal function that maps a *Signal* of $\mathit{GUIInput}$ values to a *Signal* of $\mathit{Picture}$ values. As an example of a $\mathit{SimpleGUI}$, consider a challenge

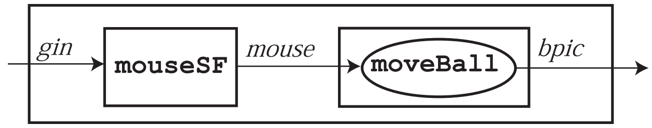


Figure 7.1: ballGUI Specification

for GUI programming languages posed by Myers many years ago [51]: a red circle that follows the mouse. For the moment we assume the Fruit library provides a signal function, $mouseSF$, that can extract the mouse's current position from the $GUIInput$ signal:

$mouseSF :: SF\ GUIInput\ Point$

We will rely on the Haven graphics library of chapter 6 for types and functions for static 2-D images, such as points, shapes, affine transforms and images. Using just Haven on its own (without Yampa), we can write:

-- a red ball positioned at the origin:
$ball :: Picture$
$ball = withColor\ red\ circle$
$moveBall :: Point \rightarrow Picture$
$moveBall\ p = translatePic\ ball\ p$

Given a point p whose components are x and y, $moveBall\ p$ is a picture of the red ball spatially translated by amounts given by x and y on the respective axes.

Note that $moveBall$ is a function over static values, not over signals. However, we can use Yampa's primitive *lifting* operator arr to *lift* the $moveBall$ function to obtain a signal function that maps a *time-varying* point to a *time-varying* picture (of type $SF\ Point\ Picture$). To allow the mouse to control the ball's position we connect the output signal of $mouseSF$ to the input signal of the lifted $moveBall$ using serial composition, as shown in figure 7.1.

Using the Arrows syntax of chapter 2, we would write $ballGUI$ as:

$ballGUI :: SimpleGUI$
$ballGUI = \textbf{proc}\ gin \rightarrow \textbf{do}$

$$gin \succ\!\!- mouseSF \rightarrow mouse$$
$$moveBall\ mouse \succ\!\!- returnA$$

Because the expression $moveBall\ \ mouse$ is computed point-wise, this specifies that, at every point in time, the output signal of the entire **proc** is $moveBall$ applied to $mouse$, where $mouse$ is the point-wise sample of the output signal of $mouseSF$. There is an important connection between such point-wise expressions and one-way constraints. We can interpret the last line as a *constraint* specifying that, at every point in time, the output picture produced by $ballGUI$ must be $ball$ translated by the current $mouse$ position.

7.1.1 What is a GUI?

The $SimpleGUI$ type is sufficient for describing $GUIs$ that map a $GUIInput$ signal to a $Picture$ signal. This accounts for the visual interaction aspects of a GUI, but real GUI-based applications connect the GUI to other parts of the application not directly related to visual interaction. To model these connections we expand the $SimpleGUI$ definition to:

$$\mathbf{type}\ GUI\ \ a\ b = SF\ (GUIInput, a)\ (Picture, b)$$

The input and output signals of $SimpleGUI$ have been widened by pairing each with a type specified by a type parameter. These extra *auxiliary semantic signals* enable the GUI to be connected to the non-GUI part of the application.

7.1.2 Library GUIs

The Fruit library defines a number a number of standard user interface components (or “widgets”) found in typical GUI toolkits as GUI values. Here we briefly present the programming interface to these components. Note, however, that there is nothing special or primitive about these components; they are just ordinary GUI values, defined using the Yampa primitives and graphics library.

Labels The simplest standard GUI components are labels, defined as:[1]

$$flabel :: LabelConf \rightarrow GUI\ LabelConf\ ()$$
$$ltext :: String \rightarrow LabelConf$$

A label is a GUI whose picture displays a text string taken from its auxiliary input signal, and produces no semantic output signal.

The behavior and appearance of a component at any point in time is determined by its *configuration options*. $LabelConf$ is the type of configuration options specific to the $flabel$ component. For labels, $LabelConf$ has just one constructor, $ltext$, which specifies the string to display in the label. Note, too, that $flabel$ is defined as a function that takes a $LabelConf$ argument and returns a GUI. The $LabelConf$ argument allows the user to specify an *initial default configuration* for the properties of the GUI, analogous to the role of constructor arguments in object-oriented toolkits. If a value for a particular property is specified by time-varying input signal to the GUI, the value specified in the input signal will override the initial configuration.

We use a trick from Fudgets [10] to specify configuration options. $LabelConf$, $ButtonConf$, etc. are simple $State \rightarrow State$ functions. These functions are very similar to the update functions generated by using Haskell's labeled field syntax, in that they will update one component of the state, but leave all others unchanged. This gives us a simple mechanism for composing property definitions (using the function composition operator '$\circ$') and for assigning default values for component properties. We will see an example of this shortly.

[1] Haskell's unit type (written ()) is the type with just one value, also called unit, and also written as (). Unit serves a similar role to the $void$ type in ANSI C.

Buttons A Fruit button (*fbutton*) is a GUI that implements a standard button control. The declaration of *fbutton* is:

$$
\begin{aligned}
&\mathit{fbutton} :: \mathit{ButtonConf} \rightarrow \mathit{GUI}\ \mathit{ButtonConf}\ (\mathit{Event}\ ())\\
&\mathit{btext} :: \mathit{String} \rightarrow \mathit{ButtonConf}\\
&\mathit{enabled} :: \mathit{Bool} \rightarrow \mathit{ButtonConf}
\end{aligned}
$$

There are two constructors for the $\mathit{ButtonConf}$ type: one to specify the string to display in the button, and another to control whether the button is enabled. A button that is disabled will have a grayed-out appearance, and does not react to mouse or keyboard input. A button is an event source that has an occurrence when the primary mouse button is pressed when the mouse is positioned over the button. Each event occurrence on the output signal carries no information other than the fact of its occurrence, hence the type $\mathit{Event}\ ()$.

7.2 The GUIInput Type

The $\mathit{GUIInput}$ type represents the part of the input to a GUI specifically related to its visual interactive characteristics. $\mathit{GUIInput}$ is essentially just a pair of records:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Mouse} = \{\ \mathit{mpos} && :: \mathit{Point},\\
&\qquad\qquad\qquad\ \ \mathit{lbDown} && :: \mathit{Bool},\\
&\qquad\qquad\qquad\ \ \mathit{rbDown} && :: \mathit{Bool}\}\\
&\mathbf{data}\ \mathit{Kbd} = \{\quad \mathit{keyDown} && :: [\mathit{Char}]\}\\
&\mathbf{type}\ \mathit{GUIInput} = (\mathit{Maybe}\ \mathit{Kbd}, \mathit{Maybe}\ \mathit{Mouse})
\end{aligned}
$$

The Kbd and Mouse types are wrapped in Maybe types to account for the *focus model*. In modern window systems, there is always a foreground application that receives the keyboard and mouse input from the window system to the exclusion of all other applications running in the background. The window system typically provides a lightweight gesture (such as mouse-over or click-to-type) that allows the user to shift the focus to another application. This concept of focus model is equally applicable within a window, as we can view moving the mouse between

Figure 7.2: Using besideGUI

two different visible components of a window as shifting the mouse focus from one component to the other. Keyboard focus traversal within a window (using the TAB key, for example) can be modeled analogously. Each of the $Maybe$ values in the $GUIInput$ signal to a GUI are $Nothing$ when the GUI does not have focus, and $Just\ x$ (for some x) when the component has the focus.

7.3 Basic Layout Combinators

To be able to build more interesting interfaces, we need a mechanism to compose multiple GUIs into a larger GUI. We provide two basic *layout combinators* for this purpose:

$$\begin{array}{l} aboveGUI :: GUI\ b\ c \rightarrow GUI\ d\ e \rightarrow GUI\ (b, d)\ (c, e) \\ besideGUI :: GUI\ b\ c \rightarrow GUI\ d\ e \rightarrow GUI\ (b, d)\ (c, e) \end{array}$$

The layout combinators produce a combined GUI that behaves as the two child GUIs arranged adjacent to one another. Here is a small example that illustrates the use of $besideGUI$:

$$\begin{array}{l} hello :: GUI\ ()\ (Maybe\ (), ()) \\ hello = \mathbf{proc}\ (inpS, \_) \rightarrow \mathbf{do} \\ \quad (fbutton\ \text{`}besideGUI\text{`}\ flabel) \prec \\ \qquad (inpS, (btext\ \texttt{"press me"}, \\ \qquad\quad ltext\ \texttt{" PLEASE! "})) \end{array}$$

The result of running this GUI in a top-level window with $runGUI$ is shown in figure 7.2. A *translation transformation* has been applied to the second argument

GUI to position it beside the first argument. The implementation of spatial transformation for GUIs will be described in detail in section 7.3.1.

In addition to transforming the second argument, the layout combinators must *demultiplex* the input signal into two disjoint signals to be passed to each child. This is achieved by *clipping* the $GUIInput$ signal based on the mouse position and the bounds of the picture output signals of the composed GUIs: The GUI under the mouse receives the (appropriately transformed) keyboard and mouse signals, while its sibling receives $Nothing$ values for the keyboard and mouse to indicate that it does not have focus.[2]

7.3.1 Transforming GUIs

One difference between Fruit and every other production user interface toolkit we are aware of (for either imperative or functional languages) is that Fruit provides a uniform model and programming interface for both "low-level" interactive graphics and "high level" user interface components such as buttons. Moreover, since GUIs are first class values that denote *pure functions*, we can use higher-order operators to manipulate GUIs in useful ways.

One of the most basic higher-order functions is the function composition operator ($\circ$); we use $\ggg$ instead, but the denotation is equivalent. (Recall that $(\ggg)$ is *reverse* composition, so $f \ggg g = g \circ f$ for the function space arrow.) Armed with just this operator, we can define *spatial transformation* of a GUI. We will define a generalized $transformGUI$ operator that applies an (affine) spatial transform to a GUI to produce a new GUI:

[2]Our current implementation of focus is based solely on mouse position. This is slightly simplistic, as modern user interface guidelines stipulate a keyboard focus cycle that is independent of the mouse focus. Extending our implementation to support such a split focus model is straightforward.

$transformGUI :: Transform \rightarrow GUI\ b\ c \rightarrow GUI\ b\ c$

Assuming that we have a basic understanding of spatial transformation for pictures, how shall we define spatial transformation of a GUI?

First, let's quickly review spatial transform for pictures. When we apply a spatial transform to a picture, it changes the size, position, or orientation of the picture. Consider translation of a picture by a displacement vector $(\Delta x, \Delta y)$. In general, this translation maps every (x, y) position in the original image to an (x', y') position in the new image by:

$$(x', y') = (x + \Delta x, y + \Delta y)$$

or, more generally, if tf represents the transformation, and $\%\$$ is the *apply-transform* operator:

$$(x', y') = tf\ \%\$\ (x, y)$$

Note that $\%\$$ is defined as part of the *Transformable* type class, so instance declarations may be given for any type that supports spatial transformation.

Since a GUI's visual output is a signal of $Picture$, and our graphics library supports applying affine transforms to $Picture$ values, we can transform a GUI's output by point-wise application of the transform to the picture output signal. But what about input?

If g is a GUI, point-wise transformation of g's picture signal will map every (x, y) position in g's coordinate system to (x', y'). In order to give an accurate input signal to g, $transformGUI$ must map every (x', y') mouse position back to its corresponding (x, y) position in g. This suggests a general principle for transforming functions: *To transform a function (in time or in space), apply the transform point-wise to the output, and apply the inverse transform point-wise to the input.* This idea corresponds exactly to Pan's spatial "hyper-filters" [22], i.e., spatial transformations of $Image \rightarrow Image$ functions. Note that this sample principle of applying a transform to a function was used to account for applying a time transform to a signal function in the denotational semantics of $switch$ in chapter 4.

The implementation of $transformGUI$ is then simply:

$$
\begin{array}{l}
transformGUI\ tf\ g = \mathbf{proc}\ (inp, b) \rightarrow \\
\quad (pic, c) \leftarrow g \prec (inverse\ tf\ \%\$\ inp, b) \\
\quad returnA \prec (tf\ \%\$\ pic, c)
\end{array}
$$

This model for transforming GUIs is used in the implementation of the layout combinators to reposition their second argument GUI. The transform to apply to the second argument is determined dynamically by applying a $bounds$ operation point-wise to the $Picture$ signal produced by the first argument GUI.

Spatial Scalability

While our basic layout combinators only use basic horizontal and vertical translations, the $transformGUI$ operator can apply *any* affine transform to a GUI. For example, here is a version of a Paddleball game that runs in a window $1/2$ the size of the original:

-- uniform scaling transform (from Graphics library):

$$
\begin{array}{l}
uscale :: Double \rightarrow Transform \\
minipb :: Double \rightarrow GUI\ ()\ (Maybe\ ()) \\
minipb\ vel = \\
\quad transformGUI\ (uscale\ 0.5)\ (pball\ vel)
\end{array}
$$

When run, $minipb$ displays a fully functional version of Paddleball shrunk down to postage stamp size. This type of zooming capability is obviously extremely useful for implementing vector or bitmap graphics editors, document previewers, etc. where zooming is a natural operation. But recent work in the Human/Computer Interaction (HCI) community has proposed continuous zooming can be a useful abstraction in its own right for many applications [63] [6]. Providing continuous zoom allows graphical interfaces to be designed so that users can "zoom out" for an overview of the data and "zoom in" for more detail. Pad [63] and Jazz [6] are two recent research projects that augment the widget set of a traditional imperative GUI toolkit with the abstraction of a continuously zoomable drawing surface.

The starting points for Pad and Jazz were the toolkits Tk and Swing, respectively. Because the Tk and Swing programming interfaces hide their connection with the graphics subsystem, Pad and Jazz are essentially new GUI toolkits, and require that existing applications be rewritten from scratch to take advantage of the zooming capabilities. In contrast, Fruit makes the connection to the interactive graphics subsystem seamless and explicit in the type of GUI. As $minipb$ demonstrates, this explicit connection to interactive graphics allows us to incorporate novel ideas (such as continuous zooming) without a major restructuring of the library or completely rewriting applications.

7.4 Specifying Layout Using Arrows

As the example from section 7.3 illustrates, a composed GUI has auxiliary semantic input and output signals whose types are the product of the corresponding types from the child GUIs. This has substantial syntactic consequences. Programs can become complicated rather quickly, because the type of a composed GUI will grow in proportion to the nesting depth of the layout.

To redress the syntactic overhead of specifying layout, Fruit exploits the ability of the Arrows framework to perform implicit plumbing based on the linear sequencing of arrow composition. Fruit defines two new type constructors, GA and Box, both of which are, like SF, declared as instance of $Arrow$.

The GA type is simply a **newtype** wrapper around the GUI type synonym given earlier:

$$\textbf{newtype}\ GA\ b\ c = GA\ (SF\ (GUIInput, b)\ (Picture, c))$$

The only interesting thing about the $Arrow$ instance declaration for GA is the compose operation ($\ggg$). The compose operation defined for GA assumes that GUIs are laid out, and clips the GUIInput signal of each GUI by the bounds of its output picture signal:

```
gaComp :: GA b c → GA c d → GA b d
gaComp (GA g1) (GA g2) = GA $ proc (gin, b) → do
  rec let g1bounds = rbounds g1pic
    let g1gin = clipRect g1bounds gin
    (g1pic, c) ← g1 ─≺ (g1gin, b)
    let g2gin = invClip g1bounds gin
    (g2pic, d) ← g2 ─≺ (g2gin, c)
  returnA ─≺ (g1pic <+> g2pic, d)
```

The Box type is used to specify a sequence of GUIs that should be laid out in a linear arrangement. Box is defined as an *environment* arrow, so that the specification of whether or not to lay out a box vertically or horizontally can take place outside of the box:

$$\textbf{newtype}\ Box\ b\ c = Box\ (LayoutF \rightarrow GA\ b\ c)$$

Here $LayoutF$ is a *layout function*. The layout function takes a $Rectangle$ that encloses all preceding GUIs in the sequence, and computes a Transform to apply to layout the next GUI:

$$\textbf{type}\ LayoutF = Rectangle \rightarrow Transform$$

We can now define a version of $gaComp$ that takes a $LayoutF$ and applies this to the bounding rectangle of the first GA to compute a transform to apply to the second GA:

```
gaGComp :: LayoutF → GA b c → GA c d → GA b d
gaGComp lf (GA g1) (GA g2) = GA $ proc (gin, b) → do
  rec let g1bounds = rbounds g1pic
    let g1gin = clipRect g1bounds gin
```

```
    (g1pic, c) ← g1 ⤙ (g1gin, b)
    let g2gin = invClip g1bounds gin
    let tf = lf g1bounds
    (g2pic, d) ← dynTransformGUI g2 ⤙ (g2gin, (tf, c))
  returnA ⤙ (g1pic <++> g2pic, d)
```

The arrow compose operator for *Box* simply inherits the layout function from the environment and uses this to layout its children:

```
compBox :: Box b c → Box c d → Box b d
compBox (Box b1f) (Box b2f) = Box $ λlf → gaGComp lf (b1f lf) (b2f lf)
```

The interaction between *Box* and *GA* ensures that a box is always laid out before it can be used as a *GA*. The functions for applying a layout to a *Box* to obtain a *GA* are *hbox* and *vbox*, defined as follows:

```
vlayout :: LayoutF
vlayout r = yTranslate (rectHeight r)

hlayout :: LayoutF
hlayout r = xTranslate (rectWidth r)

hbox :: Box b c → GA b c
hbox (Box bf) = bf hlayout

vbox :: Box b c → GA b c
vbox (Box bf) = bf vlayout
```

Finally, a number of useful *liftings* are provided for lifting ordinary signal functions and GUIs into *Box* and *GA*:

```
box :: GA b c → Box b c
box ga = Box (const ga)

boxSF :: SF b c → Box b c
boxSF = box ∘ gaSF

boxGUI :: GUI b c → Box b c
boxGUI = box ∘ GA
```

Using the *Box* and *GA* types in conjunction with the Arrows syntactic sugar makes specifying GUIs easy and concise. For example, here is a version of the "hello" GUI from figure 7.2 using the box layout arrow:

```
boxHello :: GA () ()
boxHello = hbox $ proc _ → do
  button_ ⤙ btext "press me"
```

$label\_ \prec$ $ltext$ `"please!"`

Here $button\_$ and $label\_$ are simple liftings of the corresponding GUI component into Box. The button and label will be arranged linearly in the order in which they are specified, and they will be laid out horizontally by virtue of passing the box to $hbox$. All of the plumbing of the $GUIInput$ and $Picture$ signals is now handled automatically by the Box arrow instance.

7.5 Chapter Summary

This chapter has presented *Fruit*, a Functional Reactive User Interface Toolkit. Fruit defines GUIs compositionally using only the Yampa framework and formally tractable types for the keyboard, mouse and pictures. GUI components provide input and output signals to model input devices and visual display, and auxiliary semantic input and output signals for connecting to the rest of the program. A set of layout combinators provide multiplexing and demultiplexing of window-system related input and output signals, and suitable arrow instances use the linear sequencing of arrows to automate many of the details of layout.

Chapter 8

Dynamic Collections in Yampa

The previous chapter presented the basic Fruit GUI model. In the basic model, each GUI component is represented as a signal function, and Fruit's layout combinators and Yampa's switching constructs are used to realize more complicated user interfaces.

While the basic Fruit model is adequate, it turns out to be non-modular for expressing a certain important class of user interfaces, specifically those in which user interface components are dynamically added to or removed from the interface in response to user input. This chapter describes this problem, and presents our solution, which involves extending Yampa with *first-class signal function continuations* and a set of *parallel switching combinators* for switching over signal function collections.

8.1 The Need for Dynamic Collections

In the Fruit model, each GUI component is represented as a signal function of type $GUI\ a\ b$. In addition to providing the programmer with facilities to define individual GUIs, Fruit provides a set of layout combinators that can combine two or more GUIs to form a larger aggregate GUI. Visually, two GUIs composed with layout combinators are tiled either horizontally or vertically. Temporally, however, two GUIs composed with layout combinators are executing in parallel.

Since GUIs are just signal functions, the standard Yampa switching combinators can be used to switch from one GUI to another in response to input events. Switching provides a kind of *sequential* composition over time: A GUI runs until some event occurs, and then switches to some other GUI.

8.2 Parallel Composition and Local State

An important property of signal functions is that they can accumulate local state internally. This is achieved either by direct use of feedback or through the many pre-defined accumulating signal function utilities provided in the standard Yampa utilities library ($accum$, $hold$, etc.).

When two state accumulating signal functions are composed in parallel, each signal function can accumulate its own internal state. For example, figure 8.1 shows the user interface for searching an email folder in the Mozilla Thunderbird email client. This interface is composed of a number of user interface components, such as buttons, text fields, drop down menus, etc. In Fruit, each such user interface component would be implemented as a GUI. Many of the user interface components in this example accumulate local state. In each row of search criteria, the drop down menus in the first two columns maintain the currently selected menu item, while the text fields in the final column maintain the string entered by the user. The parallel composition of signal functions allows each of these GUI components to accumulate its localized state independently of the others. For ex-

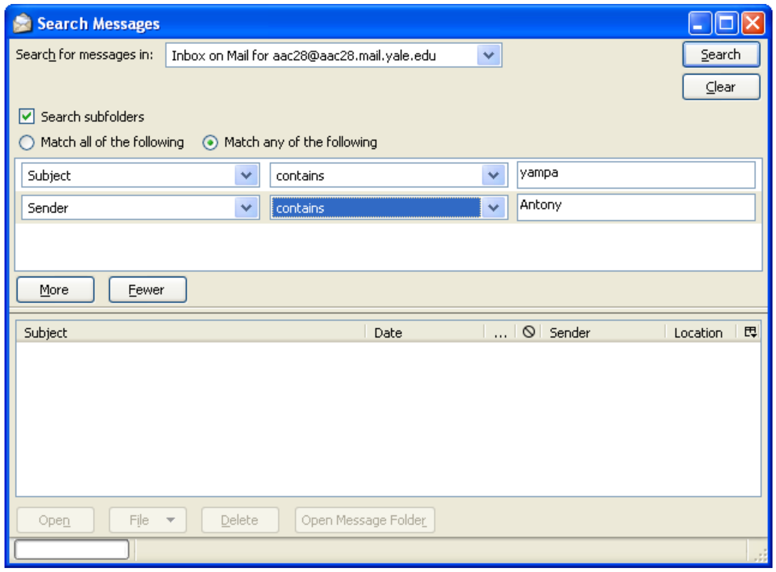


Figure 8.1: Mozilla Thunderbird Search Interface

ample, typing in the text field in one row affects only that text field; the other text fields and drop down menus in the interface retain whatever local state they have accumulated.

However, there is also a dynamic aspect to this particular user interface. Beneath the rows of search criteria in figure 8.1 are two buttons labeled “More” and “Fewer”. Pressing the “More” button adds another row of interactive GUI components to the list of search criteria. Pressing the “Fewer” button removes the last such row.

It is certainly possible to implement the search pane user interface in Fruit and Yampa, using only the primitives defined in chapters 4 and 7. Unfortunately, as we shall see, the “obvious” solution has a couple of substantial flaws.

8.3 A First Attempt

Let us consider how we might implement a simplified version of the dynamic collection of search attribute GUIs from the interface of figure 8.1 in Fruit. Let's assume that this interface consists of a number of *rows*, each of which is a GUI whose output signal is a representation of some predicate about one attribute of an email message:

$$\begin{array}{l} oneRow :: GUI\ ()\ MsgAttr \\ oneRow = ... \quad \text{-- an `hcomp` of two menus and a text field} \end{array}$$

The $MsgAttr$ type is the representation of the predicate specified by the user; it's details won't concern us here.

For simplicity, let us define a *dynamic grid* which contains a time-varying number of rows. For now we'll ignore the "Fewer" button (which removes a row from the grid), and just implement the "More" button which adds rows to the grid. The dynamic grid will have type:

$$dynGrid :: GUI\ (Event\ ())\ [MsgAttr]$$

The input signal is an event whose occurrence indicates that another row should be added to the grid. The output signal is the collection of message attribute predicates, one from each row.

If we have a value for the current collection of rows, it is easy to see how to produce a GUI that has another row:

$$\begin{array}{l} \quad \text{-- Add a row to a given search grid of arbitrary size:} \\ addRow :: GUI\ ()\ [MsgAttr] \rightarrow GUI\ ()\ [MsgAttr] \\ addRow\ curGrid = \mathbf{proc}\ (gin, \_) \rightarrow \mathbf{do} \\ \quad (pic, (sas, sa)) \leftarrow \\ \qquad curGrid\ `aboveGUI`\ oneRow \prec (gin, unit) \\ \quad returnA \prec (pic, sa : sas) \end{array}$$

The above code simply uses 'aboveGUI' to place the current grid above another row. Note that the auxiliary output signal of the composed grid is formed by cons'ing the output of the new row on to the list of outputs of the current grid.

The natural way to realize the grid itself is through use of $switch$. The type of $switch$:

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

In response to an event occurrence from the "More" button, the grid should $switch$ from the current grid to a grid with one more row (produced by $addRow$):

```
  -- Given the current grid, produce a GUI that will observe an
  -- event source input signal and add a new row when the event occurs:
mkGrid :: GUI () [MsgAttr] → GUI (Event ()) [MsgAttr]
mkGrid curGrid = switch aux
  (λ_ → mkGrid (addRow curGrid))
  where aux :: SF (GUIInput, Event ())
    ((Picture, [MsgAttr]), Event ())
    aux = proc (gin, moreE) → do
      (pic, mas) ← curGrid ⤙ (gin, ())
      returnA ⤙ ((pic, mas), moreE)
```

The $mkGrid$ functions takes a GUI that is the current collection of grid rows. This is wrapped in an auxiliary GUI that simply passes along the external event signal indicating a request for more rows, grouping the output signals in a way that is compatible with the type of $switch$. The second argument to $switch$ specifies the reaction to the switching event: a recursive call to $mkGrid$, passing $(addRow\ curGrid)$ as the new grid.

While this certainly gives a partial account of how to implement dynamic user interfaces in Fruit, this implementation has two significant flaws:

- Localized state accumulated in signal functions is not preserved across a switch. So although the above definitions implement a dynamic collection of GUIs, the collection will not exhibit the desired behavior: Every time the user presses the "More" button a new row will be added to the search grid, but *the internal state of all previous rows will be reset*.

- While rows can be dynamically added to the collection, there is no straightforward way to generalize this to the removal of rows, as we would need to do in order to implement the functionality of the "Fewer" button of figure 8.1. The problem is that, after each switching event, the entire collection is represented as a single value of type $GUI\ ()\ [MsgAttr]$. This type does not provide any access to the internal structure of the collection of GUIs which comprise this larger aggregate GUI.

We could potentially work around the first deficiency outlined above by making all internal state accumulated in each GUI available in the GUI's auxiliary output signal. We could then snapshot this auxiliary state at the time of the switching event occurrence, and use this to reconstruct the state after the switch. However, this would be both a tedious burden on the Yampa programmer and extremely non-modular, since it would require *all* internal state of all embedded signal functions to be exposed as an output signal, and every signal function constructor to accept an "initial state" argument.

8.4 Continuation-Based Switching

Fortunately, there is a more modular solution to the problem of preserving local state across a $switch$: continuation-based switching. Recall that the implementation in chapter 5 defines a signal function as a *transition function*:

$$\begin{array}{l}\mathbf{data}\ SF\ a\ b =\\ \quad SF\{sfTF :: DTime \to a \to (SF\ a\ b, b)\}\end{array}$$

The first component of the result of the transition function is a *signal function continuation* that determines how the signal function will behave at the next sample time (and hence, inductively, at all future sample times also). Hence, this signal function continuation must contain internally all of the localized state accumulated by the signal function.

The key to enabling this local state to be preserved across a switch is to provide the user with access to such signal function continuations as first-class values. The most basic form of this is $kSwitch$, a variation on $switch$ that provides the user with the signal function continuation at the time of the switch:

-- "Call-with-current-continuation" switch.

$$
\begin{aligned}
&kSwitch :: SF\ a\ b \rightarrow SF\ (a, b)\ (Event\ c) \\
&\quad \rightarrow (SF\ a\ b \rightarrow c \rightarrow SF\ a\ b) \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

The first argument to $kSwitch$ is a signal function to execute initially (called the *embedded* signal function, since it is embedded in the switch). The second argument is an *observer* signal function. The observer is given access to both the external input signal and the output signal of the embedded signal function, and produces an event signal whose occurrence causes the switch. The final argument to $kSwitch$ is a function that determines the signal function to switch into. As with the ordinary $switch$, this function is passed the value carried in the switching event. However $kSwitch$ also passes the *current signal function continuation*. This is a representation of the embedded signal function at the time of the switch.

The type of such signal function continuations is $SF\ a\ b$, the same type as any other signal function. This gives the programmer great flexibility in handling signal function continuations: the continuation can be switched back in immediately, composed with other signal functions, or saved in a data structure to be switched back in at some later time.

Using $kSwitch$ instead of $switch$ allows us to easily implement a version of the search grid that alleviates one of the problems of the previous version:

$$
\begin{aligned}
&mkGrid' :: GUI\ (Event\ ())\ [MsgAttr] \rightarrow GUI\ (Event\ ())\ [MsgAttr] \\
&mkGrid'\ curGrid = kSwitch\ curGrid\ (arr\ (snd \circ fst))
\end{aligned}
$$

$(\lambda grid\ \_ \rightarrow mkGrid'\ (addRow'\ grid))$

-- like addRow, but polymorphic on input type:

$addRow' :: GUI\ a\ [MsgAttr] \rightarrow GUI\ a\ [MsgAttr]$

$addRow'\ curGrid = \mathbf{proc}\ (gin, x) \rightarrow \mathbf{do}$

$\quad (pic, (sas, sa)) \leftarrow$

$\qquad curGrid\ \text{`}aboveGUI\text{`}\ oneRow \prec (gin, (x, unit))$

$\quad returnA \prec (pic, sa : sas)$

In this implementation, the current grid is captured in a signal function continuation when the user presses the "More" button. When the switching event occurs, the "current grid" is passed to the switching function (as the argument $grid$ in the λ expression), and this current grid is extended with a new row.

8.5 Parallel Switching and Signal Collections

There is still one problem with our implementation of the dynamic grid. Although the use of $kSwitch$ preserves the state of the grid across a $switch$, this still does not account for the "Fewer" button. The problem with implementing this feature is that, after each switch, the signal function being switched in to is formed by composing a new row with the current grid using $aboveGUI$. However, the internal structure of this composition is not exposed to the switching function. In order to allow switching functions to be able to manipulate the internal structure of a set of running signal functions, Yampa provides support for *signal function collections*.

Signal function collections are used to condense groups of signal functions into a single signal function. The collection type is arbitrary; it needs only to be in the $Functor$ class. The simplest operation groups signal functions sharing a common input:

$$parB :: Functor\ col \Rightarrow col\ (SF\ a\ b) \rightarrow SF\ a\ (col\ b)$$

The B suffix of $parB$ signifies that, in the resulting signal function, the input signal is *b*roadcast to all signal functions in the collection. Sometimes, however, it is desirable to specify a *routing function* that determines the input signal to be delivered to each element of the collection; the par primitive is provided for this purpose:

$$
\begin{aligned}
&par :: Functor\ col \Rightarrow \\
&\quad (forall\ sf \circ (a \rightarrow col\ sf \rightarrow col\ (b, sf))) \\
&\quad \rightarrow col\ (SF\ b\ c) \\
&\quad \rightarrow SF\ a\ (col\ c)
\end{aligned}
$$

This primitive will apply its first argument (the routing function) point-wise to the external input signal and the collection of signal functions to obtain a collection of (input sample, signal function) pairs. Each signal function in the collection can thus be given its own input sample. One way of thinking about the routing function is as a way of controlling *perception*: the routing function determines the view of the world seen by each element of the collection. This can be used to, for example, allow a set of objects to perceive only their closest neighbor, or only those that are located in some specified field of view.

While par and $parB$ give us basic facilities for maintaining collections of signal functions, the collections are fundamentally *static*: we cannot add or remove signal functions from the collection. For *dynamic* collections, we provide $pSwitch$, which allows the collection to be updated in response to events:

$$
\begin{aligned}
&pSwitch :: Functor\ col \Rightarrow \\
&\quad (forall\ sf \circ (a \rightarrow col\ sf \rightarrow col\ (b, sf))) \\
&\quad \rightarrow col\ (SF\ b\ c) \\
&\quad \rightarrow SF\ (a, col\ c)\ (Event\ d) \\
&\quad \rightarrow (col\ (SF\ b\ c) \rightarrow d \rightarrow SF\ a\ (col\ c)) \\
&\quad \rightarrow SF\ a\ (col\ c)
\end{aligned}
$$

The first two arguments are the routing function and initial collection, just as in par. The third argument is a signal function that observes the external input signal and the output signals of the collection, producing an event that triggers collection update. When the event occurs, the collection is reshaped by a function that produces a new collection given the value of the event.

The argument to the collection update function is the *collection* of signal function continuations captured at the time of the switch. As with $kSwitch$, these continuations are plain, ordinary signal functions, and thus can be resumed, discarded, stored, or combined with other signal functions.

$pSwitch$ is a "switch once" combinator; another version, $rpSwitch$ uses a recurring switch in the manner of $rSwitch$.

Although these switching combinators may appear complex and numerous, there is an underlying structure and relationship between all of the different switchers. For example, $rSwitch$ is defined in terms of $switch$ using a simple recursive definition:

$$
\begin{array}{l}
rSwitch :: SF\ a\ b \rightarrow SF\ (a, Event\ (SF\ a\ b))\ b \\
rSwitch\ sf = switch\ (first\ sf)\ rSwitch' \\
\quad \textbf{where} \\
\qquad rSwitch'\ sf = switch\ (sf *** notYet)\ rSwitch'
\end{array}
$$

In the above definition, $(***)$ is a derived combinator for parallel composition of two arrows, and $notYet$ is a primitive signal function that suppresses an event occurrence at time $t = 0$, but otherwise behaves as the identity signal function:

$$
\begin{array}{l}
(***) :: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d) \\
notYet :: SF\ (Event\ a)\ (Event\ a)
\end{array}
$$

The $pSwitch$ primitive is the most general of the switchers: all other switchers can be defined in terms of $pSwitch$.

8.6 Dynamic Interfaces Example

Finally, we present an implementation of the search grid portion of the interface of figure 8.1 using $pSwitchB$. This version addresses both of the deficiencies of the first implementation in section 8.3: rows may be both added *and* removed from the search grid with the "More" and "Fewer" buttons, and such dynamic updates will have no effect on the internal state of other rows in the grid.

$$
\begin{array}{l}
mkGrid'' :: [\,GUI\ (Event\ (), Event\ ())\ MsgAttr\,] \\
\quad \rightarrow SF\ (GUIInput, (Event\ (), Event\ ()))\ [(Picture, MsgAttr)] \\
mkGrid''\ curGrid = \\
\quad pSwitchB\ curGrid\ observe\ nextGrid \\
\qquad \mathbf{where} \\
\qquad\quad observe :: SF\ ((GUIInput, (Event\ (), Event\ ())), \\
\qquad\qquad [(Picture, MsgAttr)]) \\
\qquad\qquad (Event\ Bool) \\
\qquad\quad observe = \mathbf{proc}\ ((gin, (moreE, fewerE)), pmas) \rightarrow \\
\qquad\qquad returnA \prec (moreE\ \text{`}tag\text{`}\ True)\ \text{`}merge\text{`} \\
\qquad\qquad\quad (fewerE\ \text{`}tag\text{`}\ False) \\
\qquad\quad nextGrid :: [\,GUI\ (Event\ (), Event\ ())\ MsgAttr\,] \\
\qquad\qquad \rightarrow Bool \\
\qquad\qquad \rightarrow SF\ (GUIInput, (Event\ (), Event\ ())) \\
\qquad\qquad\quad [(Picture, MsgAttr)] \\
\qquad\quad nextGrid\ g\ True = mkGrid''\ (oneRow'' : g) \\
\qquad\quad nextGrid\ [\,]\ False = mkGrid''\ [\,] \\
\qquad\quad nextGrid\ (r : rs)\ False = mkGrid''\ rs \\
oneRow'' :: GUI\ (Event\ (), Event\ ())\ MsgAttr \\
oneRow'' = \mathbf{proc}\ (gin, \_) \rightarrow \mathbf{do} \\
\quad oneRow \prec (gin, ())
\end{array}
$$

In the above implementation, $observe$ merges the $moreE$ and $fewerE$ event signals into a single boolean event signal. An event occurrence carrying $True$ indicates that a row should be added to the grid, whereas $False$ indicates a row should be removed. The $nextGrid$ function takes the current collection of signal functions that form the grid (a list of rows) and the boolean event, and forms a new grid by adding or removing a row. As presented here, the result is not actually a proper GUI type; instead it is a signal function with a single $GUIInput$ signal and a collection of $(Picture, MsgAttr)$ output signals. A complete implementation would need to de-multiplex the $GUIInput$ signal to each GUI in the collection and merge the list of output pictures into a single picture by performing a point-wise $fold$ using $vcomp$ (the vertical composition operator). The details are straightforward and are omitted here for brevity.

8.7 Chapter Summary

This chapter described dynamic user interfaces and the problem they present for a dataflow programming model such as Yampa. We presented our extensions to Yampa to support such dynamic collections. Our solution is based on making the signal function continuations used in our operational semantics of chapter 4 and implementation of chapter 5 available to the Yampa programmer as first-class values, and to provide a switching primitive that can operate over collections of signal function continuations.

Part II

Applications

Chapter 9

Proving Properties of Yampa Applications

As noted in chapter 1, most graphical user interface toolkits are defined in terms of the imperative programming features of the language in which the toolkit was implemented. Fruit differs from these toolkits in that it does not appeal to imperative programming constructs or implementation artifacts for its definition. Instead, Fruit is defined entirely using Yampa, which provides a simple, precise formal model of reactive programming based on synchronous dataflow.

One benefit of functional programming in a pure language is that it enables simple formal proofs based on equational reasoning. Since Yampa is defined within the pure functional core of Haskell, we would expect that this benefit could be extended to Yampa applications, including Fruit. In this chapter, we explore basic equational reasoning proofs for Yampa.

9.1 Preliminaries and Notation

In this section, we define some types and logical formulas that will be useful when formulating propositions we wish to prove about Yampa programs.

In writing logical formulas, we will freely mix Haskell notation with standard mathematical notation. Haskell notation is usually preferred, but we will resort to mathematical notation where necessary. For example, we may write $\mathbb{N}$ to denote the set of natural numbers since the Haskell type system has no way to express this type. Haskell types appearing in logical quantifiers denote the set of all values of that type. For example:

$$\forall xs \in [\,T\,]....$$

is a logical formula in which xs ranges over all lists of type T.

We will use the convention that superscripts in lists types specify list length. So, for example:

$$\forall xs \in [\,T\,]^n....$$

is a formula in which xs ranges over all lists of T of length n.

9.1.1 Observing Signal Functions

Recall from the continuation semantics of chapter 4 that a signal function SF can be represented with type:

$$\begin{array}{l} \mathbf{data}\ SF\ a\ b = \\ \quad SF\{\, sfTF :: DTime \rightarrow a \rightarrow (SF\ a\ b, b)\} \end{array}$$

The $sfTF$ is a *transition function* that gives the reaction of a Yampa program to a single *input stimulus* (consisting of the delta time and a single sample of the input signal). The result is a *signal function continuation* that specifies how the rest of the input signal is processed, and a value for the signal function's output signal at the time of sampling.

Given a (potentially infinite) sequence of $(DTime, a)$ pairs, a *run* of a Yampa program is the result of applying the signal function to the first input stimulus, applying the resulting continuation to the second input stimulus, etc. Formally, we define a run as follows:

$$
\begin{aligned}
&runSF :: SF\ a\ b \rightarrow [(DTime, a)] \rightarrow [(SF\ a\ b, b)]\\
&runSF\ sf\ dtas = tail\ (scanl\ aux\ (sf, \bot)\ dtas)\\
&\quad \textbf{where}\ aux\ (sf, \_)\ (dt, a) = (sfTF\ sf)\ dt\ a\\
&runSF\_ :: SF\ a\ b \rightarrow [(DTime, a)] \rightarrow [b]\\
&runSF\_\ sf\ dtas = map\ snd\ (runSF\ sf\ dtas)
\end{aligned}
$$

where $scanl$ is the standard Haskell list-processing function:

$$scanl\ f\ z\ [x1, x2, ...] \equiv [z, z\ `f`\ x1, (z\ `f`\ x1)\ `f`\ x2, ...]$$

The variant $runSF\_$ is provided when only the sequence of samples of the output signal are of interest.

9.1.2 The "always" Quantifier

We often wish to show that some logical proposition *always* holds for the output signal of some signal function, regardless of its input signal. For this purpose, we use the $\Box$ ("always") quantifier of Lamport's Temporal Logic of Actions (TLA) [43]. In the context of Yampa, we define the always quantifier as:

$$
\begin{aligned}
\Box \quad &:: \quad (b \rightarrow Bool) \rightarrow SF\ a\ b \rightarrow Bool\\
\Box\ p\ sf \quad &\stackrel{\text{def}}{=} \quad \forall n \in \mathbb{N}, dtas \in [(DTime, a)]^n\ .\ all\ p\ (runSF\_\ sf\ dtas)
\end{aligned}
$$

Informally, this definition states that a predicate p *always* holds for some signal function sf if, for all possible input sequences to which sf is applied, the predicate p holds for all of the output samples.

9.2 An Invariance Theorem

In the continuation based operational semantics of chapter 4, signal functions are sampled at discrete sample points or *time steps*. It is often useful to express properties of Yampa programs as *invariants* – logical propositions about a signal function that, if true at the start of a given time step, will also hold for the continuation signal function returned for use on the next time step.

A simple but useful theorem about signal function invariants is the following:

Theorem 9.2.1 (Invariance Theorem for Signal Functions). *Let p be a logical proposition about signal functions and sf_i be a signal function parameterized by an integer i. Then:*

$$(\forall i \in \mathbb{N}, dt, a . \exists j \in \mathbb{N}, b.((sfTF\ sf_i\ dt\ a) = (sf_j, b)\ \wedge\ p\ b)) \quad \Rightarrow \quad \Box\, p\, sf_0$$

Informally, the invariance theorem states that if some property p is true of the output sample of signal functions of the form sf_i and we can show that the continuation of sf_i is of the form sf_j, then the given property always holds for any run beginning with sf_0.

Proof of Theorem 9.2.1: By induction on the length of a run, n.

Expanding the definition of $\Box$, we must show that:

$$\begin{aligned}(\forall i \in \mathbb{N}, dt, a . \exists j \in \mathbb{N}, b.((sfTF\ sf_i\ dt\ a) = (sf_j, b)\ \wedge\ p\ b)) \quad \Rightarrow \\ \forall n \in \mathbb{N}, dtas \in [(DTime, a)]^n\ .\ all\ p\ (runSF\_\ sf_0\ dtas) \qquad \mathbf{(1)}\end{aligned}$$

In each part of the inductive proof, we will take the propositions on the left-hand side of $\Rightarrow$ in (**1**) as assumptions and show that the right hand side always holds.

Base Case: $n = 1$:

$$\forall dtas \in [(DTime, a)]^1 \ . \ all\ p\ (runSF\_\ sf_0\ dtas)$$

By definition of list of length 1:

$$\equiv \ \forall dt, a.\ all\ p\ (runSF\_\ sf_0\ [(dt, a)])$$

By definition of $runSF\_$:

$$\equiv \ \forall dt, a.\ all\ p\ (map\ snd\ (runSF\ sf_0\ [(dt, a)]))$$

By definition of $runSF$:

$$\equiv \ \forall dt, a.\ all\ p\ (map\ snd\ (tail\ (scanl\ aux\ (sf_0, \bot)\ [(dt, a)])))$$

By definition of $scanl$:

$$\equiv \ \forall dt, a.\ all\ p\ (map\ snd\ (tail\ [(sf_0, \bot), aux\ (sf_0, \bot)\ (dt, a)]))$$

By definition of map, $tail$:

$$\equiv \ \forall dt, a.\ all\ p\ [snd\ (aux\ (sf_0, \bot)\ (dt, a))]$$

By definition of all:

$$\equiv \ \forall dt, a.\ p\ (snd\ (aux\ (sf_0, \bot)\ (dt, a)))$$

By definition of aux:

$$\equiv \ \forall dt, a.\ p\ (snd\ ((sfTF\ sf_0)\ dt\ a))$$

By assumption from l.h.s. of $\Rightarrow$ in (**1**):

$$\equiv \ True$$

Inductive Hypothesis: Assume that, for all $n \in \mathbb{Z}^+$:

$$(\forall i \in \mathbb{N}, dt, a. \exists j \in \mathbb{N}, b.((sfTF\ sf_i\ dt\ a) = (sf_j, b)\ \wedge\ p\ b)) \quad \Rightarrow$$
$$\forall dtas \in [(DTime, a)]^n\ .\ all\ p\ (runSF\_\ sf_0\ dtas) \qquad (\mathbf{IH})$$

Inductive Step: Given (**IH**), show that:

$$(\forall i \in \mathbb{N}, dt, a. \exists j \in \mathbb{N}, b.((sfTF\ sf_i\ dt\ a) = (sf_j, b)\ \wedge\ p\ b)) \quad \Rightarrow$$
$$\forall dtas \in [(DTime, a)]^{n+1}\ .\ all\ p\ (runSF\_\ sf_0\ dtas)$$

We will make use of the following lemma (along with some other obvious properties of lists and the map function):

Lemma 9.2.2 (Scan Lemma). *From the definition of $scanl$ in the standard Prelude:*

$$\begin{array}{l} scanl\ f\ z\ (xs +\!\!+\ ys) = \\ \quad (scanl\ f\ z\ xs) +\!\!+ \\ \qquad tail\ (scanl\ f\ (last\ (scanl\ f\ z\ xs))\ ys) \end{array}$$

$$\begin{array}{ll} & all\ p\ (runSF\_\ sf_0\ dtas) \\ & \quad \text{By definition of } runSF\_: \\ = & all\ p\ (map\ snd\ (runSF\ sf_0\ dtas)) \\ & \quad \text{By definition of } runSF: \\ = & all\ p\ (map\ snd\ (tail\ (scanl\ aux\ (sf_0, \bot)\ dtas))) \end{array}$$

Since $dtas$ is a list of length $n+1$, it must be of the form:

$$\begin{array}{rcl} dtas & = & [(dt_0, a_0), (dt_1, a_1), ...(dt_n, a_n), (dt_{n+1}, a_{n+1})] \\ & = & dtas_n\ +\!\!+\ [(dt_{n+1}, a_{n+1})] \end{array}$$

$$
\begin{aligned}
& \mathit{scanl}\ \mathit{aux}\ (\mathit{sf}_0, \perp)\ \mathit{dtas} \\
&= (\mathit{scanl}\ \mathit{aux}\ (\mathit{sf}_0, \perp)(\mathit{dtas}_n\ +\!\!+\ [(dt_{n+1}, a_{n+1})]) \\
&\quad \text{By scan lemma:} \\
&= sfbs_n +\!\!+ (\mathit{scanl}\ \mathit{aux}\ (\mathit{last}\ sfbs_n)\ [(dt_{n+1}, a_{n+1})]) \\
&\quad \textbf{where } sfbs_n = \mathit{scanl}\ \mathit{aux}\ (\mathit{sf}_0, \perp)\ \mathit{dtas}_n
\end{aligned}
$$

But note that $sfbs_n$ is the right hand side of a run of length n, and hence an instance of the inductive hypothesis. Let $(sf_n, b_n)\ =\ \mathit{tail}\ sfbs_n$. Then:

$$
\begin{aligned}
& \mathit{scanl}\ \mathit{aux}\ (\mathit{last}\ sfbs_n)\ [(dt_{n+1}, a_{n+1})] \\
&= \mathit{scanl}\ \mathit{aux}\ (\mathit{sf}_n, b_n)\ [(dt_{n+1}, a_{n+1})] \\
&= [sfTF\ \mathit{sf}_n\ dt_{n+1}a_{n+1}]
\end{aligned}
$$

By (IH),

$$
\exists\, j, b.(sfTF\ sf_n dt_{n+1}a_{n+1})\ =\ (sf_j, b)\ \wedge\ p\ b
$$

Hence:

$$
\begin{aligned}
& \mathit{all}\ p\ (\mathit{map}\ \mathit{snd}\ (\mathit{tail}\ (\mathit{scanl}\ \mathit{aux}\ (\mathit{sf}_0, \perp)\ \mathit{dtas}))) \\
&= \mathit{all}\ p\ (\mathit{map}\ \mathit{snd}\ (\mathit{tail}\ sfbs_n) +\!\!+ [b]) \\
&\quad \text{By definition of } \mathit{all}: \\
&= (\mathit{all}\ p\ (\mathit{map}\ \mathit{snd})\ (\mathit{tail}\ sfbs_n)) \wedge (p\ b) \\
&\quad \text{By } (IH)\text{:} \\
&= \mathit{True} \wedge \mathit{True} \\
&= True
\end{aligned}
$$

□

9.3 Example: A Simple Bounded Counter

In this section, we present an implementation of a simple "bounded counter" in Yampa. We wish to produce a signal function that takes an event signal as input and produces an integer signal as output. The input event signal indicates a request to increment the counter. In general, we want the integer on the output signal to be incremented whenever an event occurs on the input signal. However, such increment requests should be ignored once the counter reaches some specified maximum value.

9.3.1 Implementation

The implementation of the bounded counter in Yampa is as follows:

$$
\begin{array}{l}
bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int \\
bc\ ival\ max = loop\ ((arr\ gateReq) \ggg dAccumHold\ ival \ggg arr\ dup) \\
\quad \textbf{where}\ gateReq :: (Event\ (), Int) \rightarrow Event\ (Int \rightarrow Int) \\
\qquad gateReq\ (incReq, n) = \\
\qquad\quad (incReq\ \text{`}gate\text{`}\ (n < max))\ \text{`}tag\text{`}\ (+1) \\
\qquad dup\ x = (x, x)
\end{array}
$$

The bounded counter (bc) is realized as a function that takes an initial value ($ival$) and a maximum value (max) and returns a signal function of the appropriate type. The $loop$ combinator is used to feed back the integer output signal. The $gateReq$ function, which is applied point-wise to all input events (increment requests), uses the standard Yampa utility function $gate$ to filter increment requests: An increment request is allowed to pass through the filter so long as the current value of the counter (n) is less than the upper bound (max). In this case, the event is tag-ed with the increment function. The lifted $gateReq$ is composed with $dAccumHold$, which acts as a hold register.

The relevant definition of $dAccumHold$ (from the Yampa utilities) is:

$$dAccumHold :: a \to SF\ (Event\ (a \to a))\ a$$
$$dAccumHold\ a\_init = accum\ a\_init \ggg dHold\ a\_init$$
$$dHold :: a \to SF\ (Event\ a)\ a$$
$$dHold\ a0 = hold\ a0 \ggg iPre\ a0$$

The above definitions depend on the following Yampa primitives:

$$accum :: a \to SF\ (Event\ (a \to a))\ (Event\ a)$$
$$hold :: SF\ (Event\ a)\ a$$
$$iPre :: a \to SF\ a\ a$$

Finally, returning back to the definition of the bounded counter (bc), the output of $dAccumHold$ is fed in to $arr\ dup$ so that the integer output signal is used for both the overall and fed-back output of the signal function.

9.3.2 A Rudimentary Proof

As a simple example of a Yampa proof, let us prove that if the initial value of the counter is bounded, then the counter is always bounded:

We want to show that:

Theorem 9.3.1.

$$\forall max \in \mathbb{N}, i \in [0, max]\ .\ \Box\ (is\_bounded\ max)\ (bc\ i\ max)$$

where
$$is\_bounded\ n\ x = (x \leqslant n)$$

Note that obviously:

$$i \in [0, max]\ \Rightarrow\ is\_bounded\ max\ i$$

This fact will be useful in our proof.

Auxiliary Definitions and Lemmas

We will need to expand $bc\ i\ max$ using the definition of bc in numerous places in our proof, so we do so here:

$$\begin{aligned}
& bc\ i\ max \\
& \qquad \text{by definition of } bc\text{:} \\
= \ & loop\ (arr\ gateReq_{max} \ggg dAccumHold\ i \ggg arr\ dup) \quad (1)
\end{aligned}$$

Note that the definition of $gateReq$ refers to the value max passed as an argument to bc. Rather than expand the definition of $gateReq$, we will simply write $gateReq_{max}$ for brevity.

Let $bcAux$ refer to the body of the argument to $loop$ in (**1**):

$$\begin{aligned}
& bcAux\ i :: SF\ (Event\ (), Int)\ (Int, Int) \\
= \ & arr\ gateReq_{max} \ggg dAccumHold\ i \ggg arr\ dup \\
& \text{by definition of } dAccumHold\text{:} \\
= \ & arr\ gateReq_{max} \ggg accum\ i \ggg dHold\ i \ggg arr\ dup \\
& \text{by definition of } dHold\text{:} \\
= \ & arr\ gateReq_{max} \ggg accum\ i \ggg hold\ i \ggg iPre\ i \ggg arr\ dup
\end{aligned}$$

To prove that the counter is bounded, we will prove the following invariant about the bounded counter implementation. With the invariant theorem (theorem 9.2.1), this is sufficient to show that the counter is always bounded.

This invariant essentially asserts that for any i that is bounded by max, sampling $bcAux\ i$ results in a continuation of the form $bcAux\ j$ where j is also bounded by max.

Lemma 9.3.2 (Bounded Counter Invariant).

$$\forall dt, a, i \in [0, max] \,.\, (\exists j \in [0, max].\, \mathit{fst}\ (\mathit{sfTF}\ (\mathit{loop}\ (\mathit{bcAux}\ i))\ dt\ a) = (\mathit{loop}\ (\mathit{bcAux}\ j)))$$

Proof of Lemma 9.3.2: By equational reasoning and case analysis:

$$\begin{aligned}
& \mathit{fst}\ (\mathit{sfTF}\ (\mathit{loop}\ (\mathit{bcAux}\ i))\ dt\ a) \\
& \quad \text{by definition of } \mathit{loop}\text{:} \\
= \ & \mathit{fst}\ (\mathit{loop}\ \mathit{bcAux}', b) \\
& \quad \textbf{where } (\mathit{bcAux}', (b, c)) = (\mathit{sfTF}\ (\mathit{bcAux}\ i))\ dt\ (a, c) \\
& \quad \text{by definition of } \mathit{fst}\text{:} \\
= \ & \mathit{loop}\ \mathit{bcAux}' \\
& \quad \textbf{where } (\mathit{bcAux}', (b, c)) = (\mathit{sfTF}\ (\mathit{bcAux}\ i))\ dt\ (a, c)
\end{aligned}$$

Working out the right-hand side of the **where** clause:

By definition of $bcAux$, definition of $\ggg$ and associativity of $\ggg$, we have:

$$\begin{aligned}
& (\mathit{sfTF}\ \mathit{bcAux}\ i)\ dt\ (a, c) = \\
& \quad (\mathit{sf1}' \ggg \mathit{sf2}' \ggg \mathit{sf3}' \ggg \mathit{sf4}' \ggg \mathit{sf5}', \mathit{b5}) \\
& \quad \textbf{where} \\
& \qquad (\mathit{sf1}', \mathit{b1}) = (\mathit{sfTF}\ (\mathit{arr}\ \mathit{gateReq}_{max}))\ dt\ (a, c) \\
& \qquad (\mathit{sf2}', \mathit{b2}) = (\mathit{sfTF}\ (\mathit{accum}\ i))\ dt\ \mathit{b1} \\
& \qquad (\mathit{sf3}', \mathit{b3}) = (\mathit{sfTF}\ (\mathit{hold}\ i))\ dt\ \mathit{b2} \\
& \qquad (\mathit{sf4}', \mathit{b4}) = (\mathit{sfTF}\ (\mathit{iPre}\ i))\ dt\ \mathit{b3} \\
& \qquad (\mathit{sf5}', \mathit{b5}) = (\mathit{sfTF}\ (\mathit{arr}\ \mathit{dup}))\ dt\ \mathit{b4}
\end{aligned}$$

Working backwards, we have:

By definition of arr:

$$(\mathit{sf5}', \mathit{b5}) = (\mathit{arr}\ \mathit{dup}, (\mathit{b4}, \mathit{b4}))$$

By definition of $iPre$:

$$(\mathit{sf4}', \mathit{b4}) = (\mathit{iPre}\ \mathit{b3}, i)$$

Hence we have $b = i$ (sample of overall output signal) and $c = i$ (sample of feedback signal). However, we still need to work out values for $\mathit{sf1}'$ .. $\mathit{sf3}'$. To work out these values, we appeal to the following two lemmas, which follow by simple equational reasoning from the reference definitions given in chapter 4:

Lemma 9.3.3 (Hold Lemma). *From the definition of $hold$:*

$$
\begin{aligned}
&(\mathit{sfTF}\ (\mathit{hold}\ n))\ dt\ \mathit{mbe} = (\mathit{hold}\ x, x)\\
&\quad \textbf{where}\ x = (\mathit{event}\ n\ \mathit{id})\ \mathit{mbe}
\end{aligned}
$$

Lemma 9.3.4 (Accum Lemma). *From the definition of $accum$:*

$$
\begin{aligned}
&(\mathit{sfTF}\ (\mathit{accum}\ n))\ dt\ \mathit{mbe} = (\mathit{accum}\ x, y)\\
&\textbf{where}\ x = (\mathit{event}\ n\ (\mathit{app}\ n))\ \mathit{mbe}\\
&\quad y = \mathit{fmap}\ (\mathit{app}\ n)\ \mathit{mbe}\\
&\quad \mathit{app}\ x\ f = f\ x
\end{aligned}
$$

Returning to our proof of lemma 9.3.2:

$$
\begin{aligned}
(\mathit{sf3}', b3) &= (\mathit{sfTF}\ (\mathit{hold}\ i))\ dt\ b2\\
&\quad \text{by Hold Lemma:}\\
&= (\mathit{hold}\ x3, x3)\\
&\quad \textbf{where}\ x3 = (\mathit{event}\ i\ \mathit{id})\ b2
\end{aligned}
$$

$$
\begin{aligned}
(\mathit{sf2}', b2) &= (\mathit{sfTF}\ (\mathit{accum}\ i))\ dt\ b1\\
&\quad \text{by Accum Lemma:}\\
&= (\mathit{accum}\ x2, y2)\\
&\quad \textbf{where}\ x2 = (\mathit{event}\ i\ (\mathit{app}\ i))\ b1\\
&\quad\quad\quad y2 = \mathit{fmap}\ (\mathit{app}\ i)\ b1
\end{aligned}
$$

$$
\begin{aligned}
(sf1, b1) &= (\mathit{sfTF}\ (\mathit{arr}\ \mathit{gateReq}_{max}))\ dt\ (a, c)\\
&\quad \text{by definition of } arr\text{:}\\
&= (arr gateReq, gateReq(a, c))
\end{aligned}
$$

hence:

$$
\begin{aligned}
b1 &= gateReq\ (a, c) \\
&\quad \text{By definition of } gateReq_{max} : \\
&= (a \text{ `} gate \text{` } (c < max)) \text{ `} tag \text{` } (+1) \\
&\quad \text{By definition of } gate : \\
&= (\mathbf{if}\ (c < max)\ \mathbf{then}\ a\ \mathbf{else}\ NoEvent) \text{ `} tag \text{` } (+1) \\
&\quad \text{By definition of } tag\text{:} \\
&= fmap\ (const\ (+1))\ (\mathbf{if}\ (c < max)\ \mathbf{then}\ a\ \mathbf{else}\ NoEvent) \\
&\quad \text{Substituting value of } c\text{:} \\
&= fmap\ (const\ (+1))\ (\mathbf{if}\ (i < max)\ \mathbf{then}\ a\ \mathbf{else}\ NoEvent)
\end{aligned}
$$

We now do a case analysis on (a, i). There are three cases to consider:

case 1: $a = NoEvent$:

$$
\begin{aligned}
b1 &= fmap\ (const\ (+1))\ (\mathbf{if}\ (i < max)\ \mathbf{then}\ a\ \mathbf{else}\ NoEvent) \\
&\quad \text{substituting value of } a\text{:} \\
&= fmap\ (const\ (+1))\ (\mathbf{if}\ (i < max)\ \mathbf{then}\ NoEvent\ \mathbf{else}\ NoEvent) \\
&\quad \text{by case analysis of } \mathbf{if}\text{:} \\
&= fmap\ (const\ (+1))\ NoEvent \\
&\quad \text{by definition of } fmap\text{:} \\
&= NoEvent \\
y2 &= fmap\ (app\ i)\ b1 \\
&\quad \text{by definition of } fmap\text{, } b1\text{:} \\
&= NoEvent
\end{aligned}
$$

$$
\begin{aligned}
x2 &= (event\ i\ (app\ i))\ b1 \\
&\quad \text{by definition of } event,\ b1\text{:} \\
&= i \\
(sf2', b2) &= (accum\ i, NoEvent) \\
x3 &= (event\ i\ id)\ b2 \\
&\quad \text{by definition of } event,\ b2\text{:} \\
&= i \\
(sf3', b3) &= (hold\ i, i) \\
sf4' &= iPre\ i
\end{aligned}
$$

hence:

$$
\begin{aligned}
& sfTF\ (bcAux\ i)\ dt\ (a, c) \\
&\quad = (arr\ gateReq \ggg accum\ i \ggg hold\ i \ggg iPre\ i \ggg arr\ dup, (i, i))
\end{aligned}
$$

hence:

$$
\begin{aligned}
& fst\ (sfTF\ (loop\ (bcAux\ i))\ dt\ a) \\
&\quad = loop\ (arr\ gateReq \ggg accum\ i \ggg hold\ i \ggg iPre\ i \ggg arr\ dup) \\
&\quad = loop\ (bcAux\ i)
\end{aligned}
$$

which, if we simply choose $j = i$, satisfies lemma 9.3.2.

case 2: $a = Event\ ()$, $i < max$:

$$
\begin{aligned}
\mathit{b1} &= \mathit{fmap}\ (\mathit{const}\ (+1))\ (\mathbf{if}\ (i < \mathit{max})\ \mathbf{then}\ a\ \mathbf{else}\ \mathit{NoEvent}) \\
&\ \text{by definition of } \mathbf{if}\text{, value of } a\text{:} \\
&= \mathit{fmap}\ (\mathit{const}\ (+1))\ (\mathit{Event}\ ()) \\
&\ \text{by definition of } \mathit{fmap}\text{:} \\
&= \mathit{Event}\ (+1) \\
\mathit{y2} &= \mathit{fmap}\ (\mathit{app}\ i)\ \mathit{b1} \\
&\ \text{by definition of } \mathit{fmap}\text{, } \mathit{app}\text{, } \mathit{b1}\text{:} \\
&= \mathit{Event}\ (i+1) \\
(\mathit{sf2}', \mathit{b2}) &= (\mathit{accum}\ (i+1), \mathit{Event}\ (i+1)) \\
\mathit{x3} &= (\mathit{event}\ i\ \mathit{id})\ \mathit{b2} \\
&\ \text{by definition of } \mathit{event}\text{, } \mathit{b2}\text{:} \\
&= i+1 \\
(\mathit{sf3}', \mathit{b3}) &= (\mathit{hold}\ (i+1), i+1) \\
\mathit{sf4}' &= \mathit{iPre}\ (i+1)
\end{aligned}
$$

hence:

$$
\begin{aligned}
&\mathit{sfTF}\ (\mathit{bcAux}\ i)\ \mathit{dt}\ (a, c) \\
&= (\mathit{arr}\ \mathit{gateReq} \ggg \mathit{accum}\ (i+1) \\
&\ \ggg \mathit{hold}\ (i+1) \ggg \mathit{iPre}\ (i+1) \ggg \mathit{arr}\ \mathit{dup}, (i, i))
\end{aligned}
$$

hence:

$$
\begin{aligned}
& \mathit{fst}\ (\mathit{sfTF}\ (\mathit{loop}\ (\mathit{bcAux}\ i))\ dt\ a) \\
& \quad = \mathit{loop}\ (\mathit{arr}\ \mathit{gateReq} \ggg \mathit{accum}\ (i+1) \\
& \qquad\qquad\qquad \ggg \mathit{hold}\ (i+1) \ggg \mathit{iPre}\ (i+1) \ggg \mathit{arr}\ \mathit{dup}) \\
& \quad = \mathit{loop}\ (\mathit{bcAux}\ (i+1))
\end{aligned}
$$

which, if we choose $j = i+1$, satisfies lemma 9.3.2, since $i < max \Rightarrow (i+1) \leqslant max \Rightarrow j \in [0, max]$.

case 3: $a = \mathit{Event}\ ()$, $i >= max$:

$$
\begin{aligned}
b1 &= \mathit{fmap}\ (\mathit{const}\ (+1))\ (\mathbf{if}\ (i < max)\ \mathbf{then}\ a\ \mathbf{else}\ \mathit{NoEvent}) \\
&\quad \text{by definition of } \mathbf{if}\text{, value of } a,\ i\text{:} \\
&= \mathit{fmap}\ (\mathit{const}\ (+1))\ \mathit{NoEvent} \\
&\quad \text{by definition of } \mathit{fmap}\text{:} \\
&= \mathit{NoEvent}
\end{aligned}
$$

Since value of $b1$ is the same as case 1, rest of this case is identical to case 1.

□

9.4 Chapter Summary

This chapter explored how the operational semantics of chapter 4 can be used as a basis for reasoning about Yampa programs. We presented an *invariance theorem* for signal functions, and used this in a small example to prove that a signal function implementing a bounded counter is always bounded.

Chapter 10

The Model / View / Controller (MVC) Design Pattern in Fruit

The Model-View-Controller (MVC) design pattern [42], originally developed in the context of Smalltalk-80, has been widely employed in many modern object-oriented GUI libraries and applications, such as Java Swing [72] and Microsoft's Foundation Classes [47]. As a *design pattern* [24], MVC is a set of programming conventions to be followed when designing a program or library interface. The MVC design pattern makes it easy to add *multiple views* of the same data to an application. For example, a spreadsheet application might allow one view as a traditional two-dimensional table of numbers and another view as a bar graph of the same data. Ideally, both views should be interactive, so that changes made to the underlying data set in any view are reflected in all views. In the object-oriented setting, the MVC design pattern suggests that the implementation should be organized into three kinds of classes:

Models maintain the program state, provide a well-specified interface for updating and reading the program state, and allow registration of *listeners* to be notified when the state has changed.

Views map the program state to a visual representation. Views register as listeners on their corresponding models, to ensure that the view and the model are always consistent.

Controllers handle input events (such as mouse and keyboard presses) and map these primitive events into appropriate updates to the corresponding model.

The MVC design pattern is closely tied to the imperative, object-oriented programming model for which it was developed. This chapter explores the implementation of the MVC design pattern in Fruit. State-encapsulating signal transformers in Yampa (such as $accumHold$) naturally correspond with MVC's notion of *models*, *lifted functions* correspond naturally with *views*, and the event algebra corresponds naturally to *controllers*. Nevertheless, there are still many important open questions to explore. For example: what is the granularity of a particular MVC-style library design? Do individual GUI components like buttons and sliders act as independent view/controller pairs, or may such components be composed into larger user interfaces that are then connected to a more abstract model? Is the *model* organized monolithically, or can it be broken down into modular pieces? This offers a basic account of MVC in Fruit that answers these questions.

10.1 "Shallow" MVC: Multiple Camera Views

We can really distinguish two kinds of views: *passive* and *active*. A *passive* view observes the underlying data set, but does not provide any means for interacting directly with the data set. The preview window of the icon editor is an example

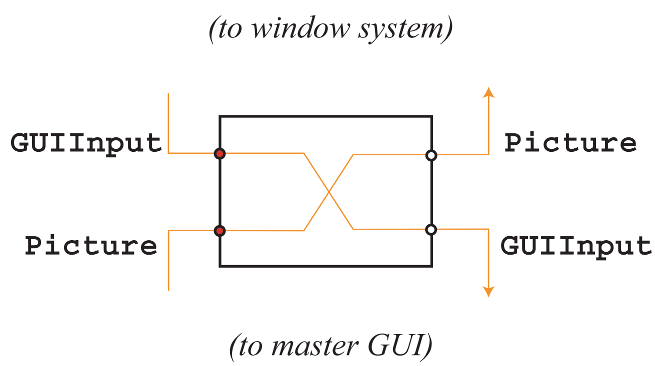


Figure 10.1: Implementation of a view

of such a passive view. In contrast, an *active* view is interactive: user actions in either view are reflected in other views and the underlying data set.

For many practical applications (such as icon editors, illustration programs, etc.), the multiple views provided by the application are views of the same underlying (time-varying) picture, with different affine transformations applied to produce the view. For example, a zoomed-in view of an icon is a view of the same picture as a zoomed-out view; the pictures differ only by a scaling transformation. For simple cases such as this, multiple views may be added in Fruit to *any* GUI, without any pre-meditation on the part of the original GUI programmer.

10.1.1 Passive Views

A view can be though of as “a GUI with no mind of its own”, as shown in figure 10.1. A view obtains its $Picture$ signal from some external source and delivers its $GUIInput$ signal to some external source. Concretely, a $view$ is a GUI that takes a $Picture$ signal as its auxiliary semantic signal, and uses this signal as its own $Picture$ signal. Similarly, it delivers its $GUIInput$ signal as its auxiliary output signal. This describes a simple crossover configuration that leads to the following definition:

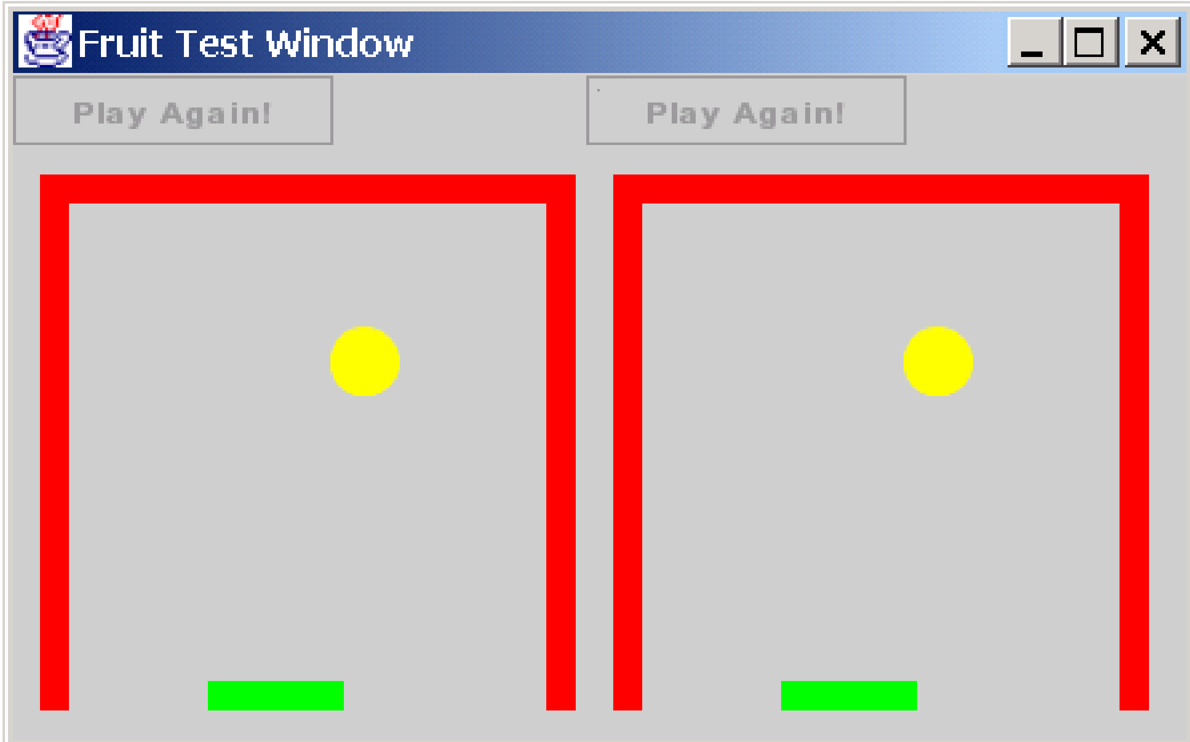


Figure 10.2: Multiple Views

$$
\begin{aligned}
&\mathit{view} :: \mathit{GUI}\ \ \mathit{G.Picture}\ \ \mathit{GUIInput}\\
&\mathit{view} = \mathit{arr}\ \ \mathit{swap}
\end{aligned}
$$

Given this definition, we can implement a version of Paddleball that has two views next to each other, as shown in figure 10.2:

$$
\begin{aligned}
&\mathit{pbview} :: \mathit{Double} \rightarrow \mathit{GUI}\ \ ()\ \ ()\\
&\mathit{pbview}\ \mathit{vel} = \mathbf{proc}\ (\mathit{inpS}, \_) \rightarrow \mathbf{do}\\
&\quad \mathbf{rec}\ (\mathit{picS}, (\mathit{activeIn}, \_)) \leftarrow\\
&\qquad (\mathit{view}\ \text{`}\mathit{besideGUI}\text{`}\ \mathit{view}) \prec\\
&\qquad\quad (\mathit{inpS}, (\mathit{gamePic}, \mathit{gamePic}))\\
&\qquad (\mathit{gamePic}, \_) \leftarrow (\mathit{rpball1}\ \ \mathit{vel}) \prec\\
&\qquad\quad (\mathit{activeIn}, \mathit{Nothing})\\
&\quad \mathit{returnA} \prec (\mathit{picS}, ())
\end{aligned}
$$

In this implementation, there are two views adjacent to each other. The view on the left is an *active* view, as its auxiliary input signal ($\mathit{activeIn}$) is fed as the input signal to the actual $\mathit{rpball1}$ GUI. The view on the right is *passive*: Its picture signal is the same ($gamePic$) signal as the active view on the left, but its $GUIInput$ signal is not connected to anything.

10.1.2 Multiple Active Views

Adding passive views to a GUI is certainly useful for many applications. But it is much more interesting, useful and symmetric to provide multiple *active* views, so that the user can interact with any view.

Recall from section 7.2 that we defined $GUIInput$ to account for a *focus model*: At every point in time, the visual input signal to a GUI is either $(Nothing, Nothing)$ (when the component does not have focus), or $(Just\ \ kbd, Just\ \ mouse)$ when the GUI has mouse focus. Further, as described in section 7.3.1, the layout combinators perform *clipping* as well as transformation to ensure that only the GUI under the mouse receives (a transformed view of) the $GUIInput$ signal. Recall, too, that our programming model includes a set of *event source combinators* that operate on signals of $Maybe$ values.

Armed with this knowledge, we can now consider how to implement active views. In the implementation of $pbview$, each $view$ is passed to the $besideGUI$ layout combinator. The $besideGUI$ combinator uses clipping and transformation to *demultiplex* its input signal into two signals, one for each child. At every point in time, one child's input signal is $(Just\ \ kbd, Just\ \ mouse)$ while the other's is $(Nothing, Nothing)$). Regardless of which GUI has focus, the input signal will be transformed into the child's local coordinate system. Given this knowledge, it is a simple matter to define a $mergeGUIInput$ combinator that will merge two disjoint $GUIInput$ signals back in to a single signal by favoring the $Just$ values and discarding the $Nothing$ values. We define $mvpball$ ("multi-view paddleball") as:

```
-- combine two maybe values, favoring Just over Nothing:
```

$$
\begin{array}{l}
\mathit{merge} :: \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \\
\mathit{merge}\ \mathit{mbeL}\ \mathit{mbeR} = \mathit{maybe}\ \mathit{mbeR}\ \mathit{id}\ \mathit{mbeL} \\
\mathit{mergeGUIInput} :: \mathit{GUIInput} \rightarrow \mathit{GUIInput} \rightarrow \\
\quad \mathit{GUIInput} \\
\mathit{mergeGUIInput}\ (\mathit{mbkA}, \mathit{mbmA})\ (\mathit{mbkB}, \mathit{mbmB}) = \\
\quad (\mathit{mbkA}\ `\mathit{merge}`\ \mathit{mbkB}, \\
\qquad \mathit{mbmA}\ `\mathit{merge}`\ \mathit{mbmB}) \\
\mathit{mvpball} :: \mathit{Double} \rightarrow \mathit{GUI}\ ()\ () \\
\mathit{mvpball}\ \mathit{vel} = \mathbf{proc}\ (\mathit{inpS}, \_) \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ (\mathit{combinedPic}, (\mathit{leftIn}, \mathit{rightIn})) \leftarrow \\
\qquad (\mathit{view}\ `\mathit{besideGUI}`\ \mathit{view}) \prec \\
\qquad\quad (\mathit{inpS}, (\mathit{masterPic}, \mathit{masterPic})) \\
\qquad \mathbf{let}\ \mathit{mergedIn} = \mathit{mergeGUIInput}\ \mathit{leftIn}\ \mathit{rightIn} \\
\qquad (\mathit{masterPic}, \_) \leftarrow (\mathit{rpball1}\ \ \mathit{vel}) \prec \\
\qquad\quad (\mathit{mergedIn}, ()) \\
\quad \mathit{returnA} \prec (\mathit{combinedPic}, ())
\end{array}
$$

In this version of Paddleball, both views are treated symmetrically: The user can play or press the restart button in either view, and the action is reflected in both views.

Fruit is the only toolkit we are aware of that provides multiple active views "for free", without requiring any extra forethought or planning by the programmer of the original GUI.

10.2 Model/View/Controller In Fruit

While the provision of multiple camera views described in the previous section is useful for some applications, camera views are a rather "shallow" interpretation of MVC, since the different views are always views of the same underlying picture signal. In true MVC, the model is completely separate from the view and controller. This makes it is possible to have entirely different visual presentations and interaction styles in the multiple views, but have these views share a common underlying model. A change made in one of the interactive views should be reflected in all of the other models.

10.2.1 GUI Component Refactoring

The Fruit programming interfaces to GUI components described in chapter 7 are simple and convenient: Each component is just a single signal function with auxiliary input and output signals used to control and observe the component. Complete GUIs are formed by specifying visual organization using layout combinators and specifying behavior by wiring of auxiliary input and output signals.

Unfortunately, the simple GUI component programming interfaces presented earlier are not quite capable of supporting a true Model/View/Controller separation. One source of difficulty is that the programming interfaces to Fruit components in chapter 7 often use continuous signals to convey auxiliary semantic signals. The use of continuous signals is one of the attractions of the FRP / Yampa model, as a continuous signal is a simple, natural way to represent dependencies between different parts of a reactive program. For example, the $label$ GUI takes a continuous time-varying $String$ as its input signal. The key benefit of this design is that it *reduces redundant state*. In a conventional toolkit, a label would maintain its own internal value for the string to appear in the label, and the programmer would need to ensure that the localized state in the GUI is always synchronized with the program state to which it corresponds. In contrast, the use of a continuous input signal to labels in Fruit enables a Fruit label to avoid having *any* internal state; the string that appears in a label is simply the current value of the label's input signal.

However, the pervasive use of continuous signals raises problems for GUI components that provide multiple views on to a single underlying model. The problem is that, in many cases, the GUI component may provide a style of interaction that requires some amount of *local* state that is only periodically synchronized against the underlying model state. For example, a text field allows editing of the text within the field, but usually does not commit changes to the underlying model until the user presses the "Enter" key.

The problem with providing a continuous signal is that, in the MVC paradigm, any given GUI component needs to both observe the shared model state and allow the user to update this model interactively. We might consider giving a text field view/controller pair a type such as:

$$textFieldVC :: GUI\ String\ String$$

with the intention that that the auxiliary input signal would come from the model and the auxiliary output signal would be connected to the model. The problem with this approach is that there is no obvious way to merge the input and output signals of multiple $textFieldVC$ so that they share a common underlying model.

The approach that we have adopted in Fruit is to instead use *events* as the input and output signals. The actual signature of a text field view/controller pair is thus:

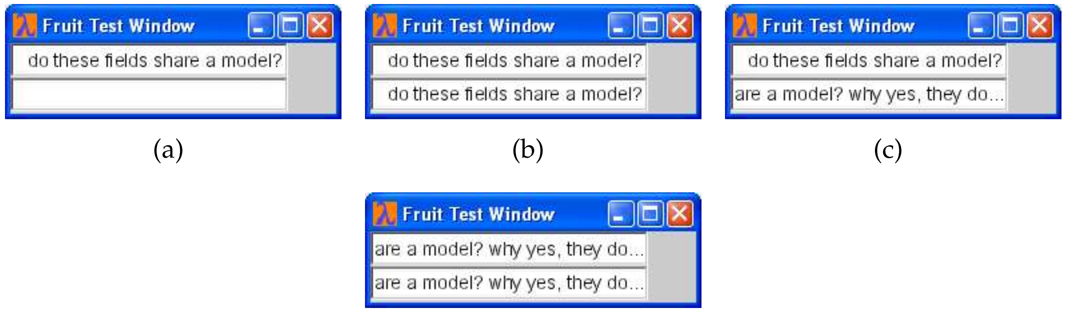
$$fTextFieldVC :: Int \rightarrow GUI\ (Event\ String)\ (Event\ String)$$

The (static) argument gives the width of the field. In the resulting GUI, the auxiliary input signal is an event whose occurrence indicates that the field's internal state should be updated with the given $String$. The auxiliary output signal is an event signal whose occurrence indicates that the model should be updated with the given string.

The underling *model* for a simple $String$ can be implemented as follows:

$$\begin{aligned}&fModel :: a \rightarrow SF\ (Event\ (a \rightarrow a))\ (Event\ a, a)\\&fModel\ v0 = accum\ v0 \ggg (arr\ id\ \&\&\&\ hold\ v0)\end{aligned}$$

The model takes an initial value as its static argument. The input signal of the resulting signal function is an event signal that specifies how to update the model's



(a) (b) (c)

(d)

Figure 10.3: A Shared Model with Local Editing

internal state. The output is a pair of signals: An event indicating when the model has been updated as well as a continuous view of the model. This latter signal enables the model's output to be fed directly to signal functions or GUI components (such as labels) that expect a continuous input signal.

It is now fairly straightforward to create GUIs that share a common model. For example, here is a pair of *textFieldVC* wired to a common input signal:

```
vcPairGA :: GA (Event String) (Event String, Event String)
vcPairGA = vbox (vc &&& vc)
  where vc = box $ GA $ fTextFieldVC 40
```

These can then be wired to share a common model as follows:

```
sharedModelFields :: GA () ()
sharedModelFields = vbox $ proc _ → do
  rec (vcChE1, vcChE2) ← box vcPairGA ≺ mChE
      (mChE, _) ← boxSF (fModel "") ≺ fmap const (vcChE1 `merge` vcChE2)
  returnA ≺ ()
```

The result of executing *sharedModelFields* is shown in figure 10.3. Figure 10.3(a) shows the resulting of typing some text in to the first text field. Note that characters typed in to the first field are not immediately reflected in the second field. Figure 10.3(b) shows the result of hitting *Enter* in the first text field: the contents of the text field are committed to the shared model, which results in immediately updating the second field. Figures 10.3(c) and 10.3(d) reflect the symmetric nature of this wiring: Figure 10.3(c) shows the result of typing some text in to the second text field, and figure 10.3(d) shows the result immediately after pressing *Enter* in the second text field.

10.3 Chapter Summary

This chapter described the Model / View / Controller (MVC) design pattern commonly used in imperative object-oriented GUI toolkits. We presented a variation on MVC, multiple camera views, and showed that a unique property of Fruit's explicit dataflow framework is that multiple camera views can be provided without any effort by the programmer. We then showed how the programming model for Fruit GUI components presented in chapter 7 can easily be adapted to support the more classical notion of MVC, and presented a small example to demonstrate multiple text field components with local editing sharing a common model.

Chapter 11

Application Case Studies

This chapter presents a number of application programs written using Fruit, Yampa and Haven. The first example, a simple three button media controller, is small enough to allow us to present the complete source code for the Fruit specification and (for contrast) to examine the essential details of a traditional imperative, object-oriented implementation. The next example, a web browser with interactive history, illustrates using Fruit to write a precise specification of how components of a user interface interact with the underlying time-varying application state. Finally, we present a complete interactive video game ("Space Invaders") implemented in Yampa, to illustrate Yampa's scalability, support for dynamic collections, and the use of signal functions in constructing simulations.

11.1 A Media Controller

As a concrete example of a Fruit specification, consider the classic VCR-style media controller illustrated in figure 11.1. The control panel provides a user interface to the simple finite state machine shown in figure 11.2. Each button is only en-

Figure 11.1: Basic Media Controller

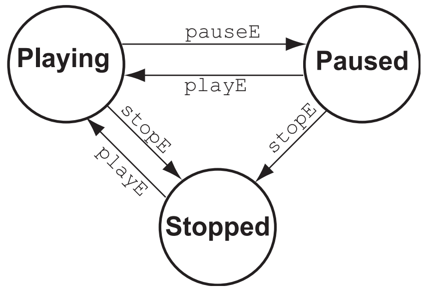


Figure 11.2: Media Controller Finite State Machine

abled if pressing the button is a valid action in the application's current state.

The implementation of the media controller in Fruit is easily derived from the state machine in figure 11.2, and is shown in figure 11.3. The implementation uses an enumerated type to encode the current state[1]:

$$\mathbf{data}\ \mathit{MediaState} = \mathit{Playing} \mid \mathit{Paused} \mid \mathit{Stopped}$$

Each of the three buttons is made by *fbutton*, and *playE*, *pauseE* and *stopE* are

[1]In C, this might be written as: `typedef enum {PLAYING, PAUSED, STOPPED} MediaState;`

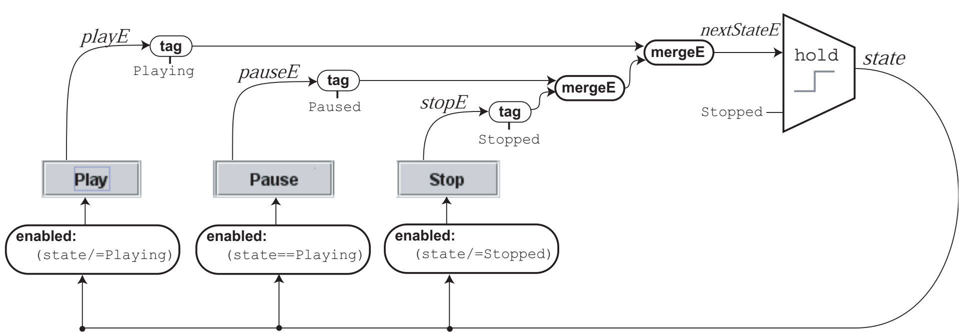


Figure 11.3: Media Controller Implementation

the output signals from each button (of type $Event\ ()$). Each event occurrence is *tagged* with its corresponding state, and these event signals are *merged* to form a single event signal, $nextStateE$, whose occurrences carry the next state. The $nextStateE$ event signal is fed to the $hold$ primitive to form a continuous signal, $state$, representing the current state. Recall from chapter 4 that the $hold$ primitive provides a continuous view of a discrete event signal by "latching" (or *hold*ing) the value of the last event occurrence across a period of non-occurrences, as illustrated in figure 2.5. Finally, the enabled property of each button is determined by a simple predicate applied point-wise to the $state$ signal. Each equation is derived directly from the state transition diagram of figure 11.2 by inspection.

Note that this diagram only illustrates the wiring of the auxiliary semantic signals of each button; the $GUIInput$ and $Picture$ signals have been omitted, because they are handled implicitly by the Box arrow of section 7.4.

The textual syntax for the media controller corresponds directly to the visual syntax of figure 11.3:

```
playerCtrl :: GUI () MediaState
playerCtrl = hbox (proc _ → do
   enabled (state ≢ Playing)
      ≻— fbutton (btext "Play") → playE
   enabled (state ≡ Playing)
      ≻— fbutton (btext "Pause") → pauseE
   enabled (state ≢ Stopped)
      ≻— fbutton (btext "Stop") → stopE
   (mergeE (tag playE Playing)
      (mergeE (tag pauseE Paused)
         (tag stopE Stopped)))
      ≻— boxSF (dHold Stopped) → state
   state ≻— returnA)
```

This definition makes use of *recursive* bindings. In this case, $state$ is used in the expression for the input signals on the first three lines, but is not defined until the line preceding $\ldots \succ\!\!- returnA$. In this example, the $dHold$ primitive introduces a delay between its event signal input and continuous output to ensure that the feedback loop is well formed. While the introduction of delays might appear subtle and arbitrary at first glance, in practice it is almost always obvious where to introduce the delays in a specification.

To complete the interface of figure 11.1, we place $playCtrl$ and a label in a vertical box, and connect the $state$ output signal of $playCtrl$ to the input signal of the label:[2]

$$
\begin{array}{l}
player :: GUI\ ()\ () \\
player = vbox\ (\mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad ()\ \succ\!\!- box\ playerCtrl \rightarrow state \\
\quad (ltext\ (\texttt{"state: "} +\!\!+ (show\ state)))\ \succ\!\!- label)
\end{array}
$$

A connection between point-wise computations and one-way constraints is apparent in the specifications of $playCtrl$ and $player$: We can interpret the input signal to each button as a constraint specifying the relationship between the $enabled$ property of the button and a predicate applied to the $state$ signal. Similarly, we can interpret the input signal to the $label$ as constraint that specifies that, at every point in time, the label's text property must be equal to the given string expression computed from $state$.

11.2 A Web Browser with Interactive History

In this section, we develop another, slightly larger example application: a web browser with a navigable interactive history mechanism. This example will illustrate how applications are composed from existing components in Fruit, and demonstrate the utility of signals and signal functions for giving a concise, rigorous specification of the connection between an application's internal time-varying

[2] Here $show$ has type $(MediaState \rightarrow String)$; the $+\!\!+$ operator is Haskell's string concatenation operator.

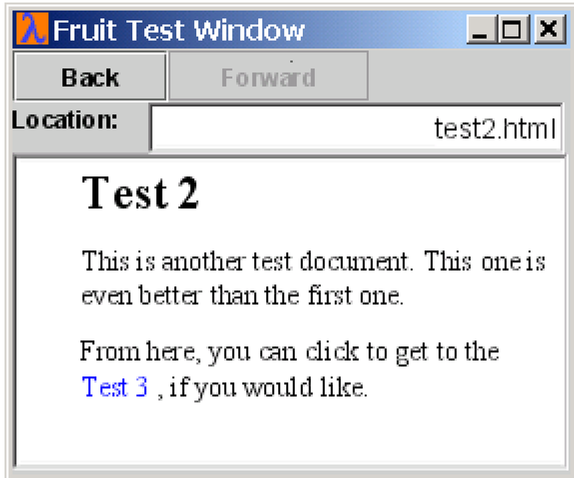


Figure 11.4: A Simple Web Browser

state and the graphical user interface presented to the user.

The user interface to the web browser is shown in figure 11.4. The (editable) text field adjacent to the “Location:” label displays the URL of the document shown in the text pane. The user may navigate by:

- Pressing either of the “Back” or “Forward” buttons will navigate to the previous or next document in the history list. (Note that the “Forward” button is *disabled* in figure 11.4, which indicates that the browser is positioned at the end the history list. Similarly, the “Back” button is disabled when the browser is positioned at the start of the history list.)

- Typing an URL explicitly in the “Location:” text field will jump to the document referenced by that URL.

- Clicking on a hypertext link in the document will jump to the document referenced by the URL embedded in the hypertext link.

Note that this informal specification is rather imprecise, since it does not specify exactly how these actions affect the history list or current position in that list. We will rectify this situation shortly.

11.2.1 Basic Components

To start with, we will assume definitions for a *text field* and *html pane* as basic GUI components:

$$\begin{array}{l} textField :: Int \rightarrow GUI\ (Event\ String)\ (Event\ String) \\ htmlPane :: GUI\ (Event\ String)\ (Event\ String) \end{array}$$

The $textField$ function takes a static value (giving the width of the text field), and returns a GUI. Internally, the text field component maintains a *local edit buffer*, that the user may edit using key strokes, and edit keys (backspace, etc.). The auxiliary input signal to the text field GUI is an event source that resets the text field's local edit buffer to the string carried with the event occurrence. The output signal of the text field is an event source that occurs whenever the user *commits* the contents of the text field (by typing the $ENTER$ key, for example). The event occurrence carries the contents of the text field's local edit buffer.

The $htmlPane$ is a GUI capable of loading and rendering a document written in the HyperText Markup Language (HTML). The auxiliary input signal to $htmlPane$ is an event source whose occurrences carry a string that gives the Uniform Resource Locator (URL) of a document to display in the pane. The output signal is an event that occurs when the user clicks on a hyper-text link appearing in the pane with the mouse. The event occurrence carries the URL associated with the link the user selected.

To start with, we can combine the text label ("Location:") with the text entry field to form a larger GUI component that obeys the same interface as a $textField$:

$$\begin{array}{l} locBar :: GUI\ (Event\ String)\ (Event\ String) \\ locBar = hbox\ \backslash\$\ \mathbf{proc}\ setUrlE \rightarrow \mathbf{do} \\ \quad ltext\ \texttt{"Location: "} \succ\!\!- label \end{array}$$

$setUrlE \succ\!\!- textField\ 40$

11.2.2 A History-less Browser

As a first experiment, we can compose the location bar component with the htmlPane to form a rudimentary browser that doesn't support interactive history:

```
  -- A simple web browser (no history mechanism)
browser0 :: GUI () ()
browser0 = vbox $ proc _ → do
  rec setUrlE >− box locBar → urlTypedE
      setUrlE >− htmlPane → linkClickE
      let setUrlE = urlTypedE `mergeE` linkClickE
  () >− returnA
```

This actually forms a complete working mini-browser. When executed, *browser0* appears similar to figure 11.4, without the "Back" and "Forward" buttons at the top. The user may navigate by typing an URL or clicking on a link. In either case, the URL of the document displayed in the html pane appears in the text field of the location bar. This is controlled by a simple *feedback loop* in the wiring structure: the event source *setUrlE* is formed by point-wise merging of the event sources *urlTypedE* and *linkClickE*. This merged event source is then fed as the input signal to both the location bar and the html pane.

Since we would like to use the user interface of *browser0* in our history-sensitive browser, with a slight modification we make a re-usable *browserPane* component:

```
browserPane :: GUI (Event String) (Event String)
browserPane = vbox \$ proc setUrlE → do
  setUrlE >− box locBar → urlTypedE
  setUrlE >− htmlPane → linkClickE
  let urlEnteredE = urlTypedE `mergeE` linkClickE
  urlEnteredE >− returnA
```

This is almost exactly the same as *browser0*, except that the *setUrlE* event source is taken from the component's input signal, and *urlEnteredE* is delivered as the component's output signal. (Of course, we could easily derive $\mathit{browser0}$ from $\mathit{browserPane}$ by simply wiring $\mathit{browserPane}$'s output signal to its input.)

11.2.3 Modeling Interactive History

We wish to develop a model of an interactive history list, and relate this to the GUI of our web browser. In this section, we will show that the history list can be modeled simply and naturally as a signal function, and show how the implementation can be derived systematically by answering a few basic questions. This approach corresponds closely to the design methodology typically used to specify (and derive) synchronous digital circuits [46]:

1. *What is the application state?* We will model the history list as a time-varying pair whose components are: the list of URLs that comprise the history, and an integer index giving the browser's "current position" in the history list. A snapshot of history list state at any point in time is thus:

   ```
     -- The history state is a history position and
     -- a list of URLs:
   type HistState = (Int, [String])
   ```

2. *What signals from the GUI affect the application state?* This determines the *input signals* to our history list model. For this example, the application state depends on *events* from the GUI. We can divide GUI events into two kinds: events in which the user selects an URL (either by typing it explicitly or clicking on a hypertext link), and navigation events (clicking on the "Forward" or "Back" buttons). Events of the former type carry a String (giving the URL); and we will require events of the latter type to carry an Int giving the amount (either $+1$ of -1) to adjust the history position.

3. *How do the signals from the GUI affect the application state?* The following functions specify how the history list state reacts to each kind of input event. Each reaction is specified as a Curried function, which takes the event datum to a $State \rightarrow State$ function.

   -- go to a specified URL
   $goUrl :: String \rightarrow HistState \rightarrow HistState$
   $goUrl\ url\ (pos, hList) = (0, url : (drop\ pos\ hList))$
   -- Adjust the current position in the history list
   -- by the specified offset:
   $hStep :: Int \rightarrow HistState \rightarrow HistState$
   $hStep\ off\ (pos, hList) = (pos + off, hList)$

4. *How does the GUI depend on the application state?* This question determines the *output signals* of our model. In this case, we need to provide two connections to the GUI: A *next URL* event that specifies the next URL to display in the $browserPane$, and the *current position* and *maximum valid index* (call it $histMax$) in the history list. These latter two signals are used to determine the *enabled* state of the forward and back buttons.

With these questions answered, we can thus derive the history list as a signal function:

$histList :: SF\ (Event\ String, Event\ Int)$
$\quad (Event\ String, Int, Int)$
$histList = \mathbf{proc}\ (urlSelectE, navStepE) \rightarrow \mathbf{do}$
$\quad \mathbf{rec}$
    -- bind an appropriate state update
    -- function to each event source:
  let $stepE = fmap\ hStep\ navStepE$
  let $locE = fmap\ goUrl\ urlSelectE$
    -- merge navigation event sources into a
    -- single event source:
  let $navE = locE\ `mergeE`\ stepE$
    -- accumulate navE events over time:
  $navE \succ\!\!- accum\ state0 \rightarrow nextStateE$

```
    -- use a hold operator to obtain a continuous
    -- view of the state:
  nextStateE ≻— dHold state0 → (histPos, histList)
  let histMax = (length histList) − 1
    -- map next state event to next url event:
  let nextUrlE = fmap getUrl nextStateE
(nextUrlE, histPos, histMax) ≻— returnA
```

Composing the navigation bar (consisting of the "Forward" and "Back" buttons) is now straightforward:

```
navBar :: GUI (Int, Int) (Event Int)
navBar = hbox $ proc (histPos, histMax) → do
  (btext "Back" ∘
    enabled (histPos < histMax)) ≻— button → bPressE
  (btext "Forward" ∘
    enabled (histPos > 0)) ≻— button → fPressE
  let navStepE = (bPressE ‘tag‘ 1) ‘mergeE‘ (fPressE ‘tag‘ (−1))
  navStepE ≻— returnA
```

Once again, the connection between point-wise computations and one-way constraints is apparent in this definition: We can interpret the *enabled* part of the input signal to the "Back" button as a constraint specifying that, at every point in time, the enabled state of the button must be equal to the predicate $(histPos < histMax)$. The output signal of the navigation bar is formed by tagging the event from each button with an integer offset to be applied to the current position in the history list and merging these event sources.

Finally, we can combine the navigation bar, browser pane and history list into a complete browser with interactive history:

```
hbrowser :: GUI () ()
hbrowser = vbox $ proc _ → do
  rec (histPos, histMax) ≻— box navBar → navStepE
    nextUrlE ≻— box browserPane → urlSelectE
    (urlSelectE, navStepE)
       ≻— boxST histList →
         (nextUrlE, histPos, histMax)
  () ≻— returnA
```

Figure 11.5: Screen-shot of Space Invaders

This completes the implementation of the browser shown in figure 11.4.

11.3 An Interactive Video Game

In this section, we present an implementation of Space Invaders, a complete interactive video game. In addition to illustrating that Yampa can be used for full scale applications, the game makes extensive use of dynamic collections, and the parallel and continuation-based switching primitives described in chapter 8.

11.3.1 Game Play

Our version of Space Invaders is based on the classic 2-D arcade game of the same name. This section briefly describes the game, and will serve as a highly informal requirements specification. A screen-shot of our Space Invaders is shown in figure 11.5.

Aliens are invading a planet in a galaxy far, far away, and the task of the player is to defend against the invasion for as long as possible. The invaders, in flying saucers, enter the game at the top of the screen. Each alien craft has a small engine allowing the ship to maneuver and counter the gravitational pull. The thrust of the engine is directed by turning the entire saucer, i.e. by adjusting its *attitude*. Aliens try to maintain a constant downward landing speed, while maneuvering to horizontal positions that are picked at random every now and again.

The player controls a gun (depicted as a triangle) that can move back and forth horizontally along the bottom of the screen. The gun's horizontal position is controlled (indirectly) by the mouse.

Missiles are fired by pressing the mouse button. The initial position and velocity of the missile is given by the position and velocity of the gun when the gun is fired. The missiles are subject to gravity, like other objects. To avoid self-inflicted damage due to missiles falling back to the planet, missiles self-destruct after a preset amount of time.

There are also two repelling force fields, one at each edge of the screen, that effectively act as invisible walls.[3] Moving objects that happens to "bump" into these fields will experience a fully elastic collision and simply bounce back.

The shields of an alien saucer are depleted when the saucer is hit by a missile. Should the shields become totally depleted, the saucer blows up. Shields are slowly recharged when below their maximal capacity, however. Whenever all aliens in a wave of attack have been eliminated, the distant mother ship will send a new wave of attackers, fiercer and more numerous than the previous one. The

[3] It is believed that they are remnants of an ancient defense system put in place by the technologically advanced, mythical Predecessors.

game ends when an alien successfully lands on the planet.

11.3.2 Game Objects

As described in section 11.3.1, there are three essential entities or *objects* in the space invaders game: the gun, the invaders and the missiles. They all react to external events and stimuli, such as collisions with other objects and control commands from the player; i.e., they are *reactive*. This section explains how to develop and test such reactive objects in Yampa by presenting an implementation of a somewhat simplified gun. This serves as a good introduction to section 11.3.3, where the implementation of the real game is presented in detail. However, this section also exemplifies a useful Yampa development strategy where individual reactive objects are first developed and tested in isolation, and then wired together into more complex systems. The last step may require some further refinement of the individual object implementations, but the required changes are usually very minor.

Implementing a Gun

Each game object produces time-varying output (such as its current position) and reacts to time-varying input (such as the mouse-controlled pointer position). The gun, for example, has a time-varying position and velocity and will produce an output event to indicate that it has been fired. On the input side, the gun's position is controlled with the mouse, and the gun firing event is emitted in response to the mouse button being pressed. We can thus represent the gun with the following types:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{SimpleGunState} = \mathit{SimpleGunState}\{ \\
\quad \mathit{sgsPos} :: \mathit{Position2},
\end{array}
$$

```
    sgsVel :: Velocity2,
    sgsFired :: Event ()
  }
type SimpleGun = SF GameInput SimpleGunState
```

where $GameInput$ is an abstract type representing a sample of keyboard and mouse state, and $Position2$ and $Velocity2$ are type synonyms for 2-dimensional vectors.

A simple physical model and a control system for the gun can be specified in just a few lines of Yampa code:

```
simpleGun :: Position2 → SimpleGun
simpleGun (Point2 x0 y0) = proc gi → do
  (Point2 xd _) ← ptrPos −≺ gi
  rec
      -- Controller
    let ad = 10 ∗ (xd − x) − 5 ∗ v
      -- Physics
    v ← integral −≺ clampAcc v ad
    x ← (x0+) ^ << integral −≺ v
  fire ← leftButtonPress −≺ gi
  returnA −≺ SimpleGunState{
    sgsPos = (Point2 x y0),
    sgsVel = (vector2 v 0),
    sgsFired = fire
    }
```

In the first line in the body of $simpleGun$, the horizontal position of the pointer is extracted from the $GameInput$ signal using the $ptrPos$ signal function (provided by the bindings to the graphics library). We could, of course, simply define the position of the gun to be that of the pointer. However, to add some degree of physical realism, the model of the gun includes inertia and adds bounds on the acceleration and velocity. The position of the pointer is thus interpreted as the gun's *desired* horizontal position, xd, and a control system is then used to compute the *desired* acceleration, ad, based on the difference between the current and desired position and the current velocity. The goal of the control system is to make the gun reach the desired position quickly subject to the physical constraints.[4]

The bounds on the acceleration and velocity are imposed through the auxiliary function $clampAcc$:

$$
\begin{array}{l}
clampAcc\ v\ ad = \\
\quad \mathbf{let}\ a = symLimit\ gunAccMax\ ad \\
\quad \mathbf{in\ if}\ (-gunSpeedMax) \leqslant v \wedge v \leqslant gunSpeedMax \\
\qquad \vee\ v < (-gunSpeedMax) \wedge a > 0 \\
\qquad \vee\ v > gunSpeedMax \wedge a < 0 \\
\qquad \mathbf{then}\ a \\
\qquad \mathbf{else}\ 0 \\
limit\ ll\ ul\ x = max\ ll\ (min\ ul\ x) \\
symLimit\ l = limit\ (-abs\ l)\ (abs\ l)
\end{array}
$$

The equations for the velocity v and horizontal position x are simply Newton's laws of motion, stated in terms of integration:

$$integral :: VectorSpace\ a\ k \Rightarrow SF\ a\ a$$

Testing the Gun

Individual game objects such as $simpleGun$ can be tested in isolation using $reactimate$ (described in section 2.11). To test $simpleGun$ we simply define an ordinary Haskell function to render a gun state as a visual image, and compose (using $\ggg$) $simpleGun$ with a lifted (using arr) version of this function:

$$
\begin{array}{l}
renderGun :: SimpleGunState \rightarrow G.Graphic \\
renderGun = ... \\
gunTest :: IO\ () \\
gunTest = runGame\ (simpleGun \ggg arr\ renderGun)
\end{array}
$$

$runGame$ is defined using $reactimate$ and suitable IO actions for reading an input event from the window system and rendering a graphic on the screen. Executing $gunTest$ will open a window in which the gun can be positioned using the mouse.

[4]The control system coefficients in this example have not been mathematically optimized.

11.3.3 The Game Proper

Game Structure

In section 11.3.2 we showed that individual game objects could be implemented as signal functions. However, in order to form a complete game, we will need a *collection* of game objects (the gun, aliens and missiles) that are all active simultaneously. We will also need some facility to add or remove objects from the game in response to events such as missiles being fired and missiles hitting targets. Thus the collection has to be *dynamic*. Moreover, the implementation of the gun in section 11.3.2 only reacted to external mouse input, whereas in the actual game objects will also need to react to each other.

The addition and deletion of signal functions constitute structural changes of the network of interconnected active signal functions. In Yampa, structural changes are effectuated by means of switching between modes of continuous operation. In particular, Yampa provides a family of *parallel* switching combinators that allow collections of signal functions to be simultaneously switched in to and out of the network of active signal functions as a group. When switched in, the signal functions in these collections are connected in parallel (hence the name parallel switch). The collections are allowed to change at the points of switching, in effect allowing them to be dynamic. Combining parallel switching with feedback enables game objects to interact in arbitrary ways. Figure 11.6 shows the resulting overall structure of the game.

The design and implementation of Yampa's parallel switching combinators is described in detail elsewhere [56]. Here we will focus on how to use one particular parallel switching combinator, $dpSwitch$, in the context of the Space Invaders game. As illustrated in figured 11.6, there are two key aspects of maintaining the

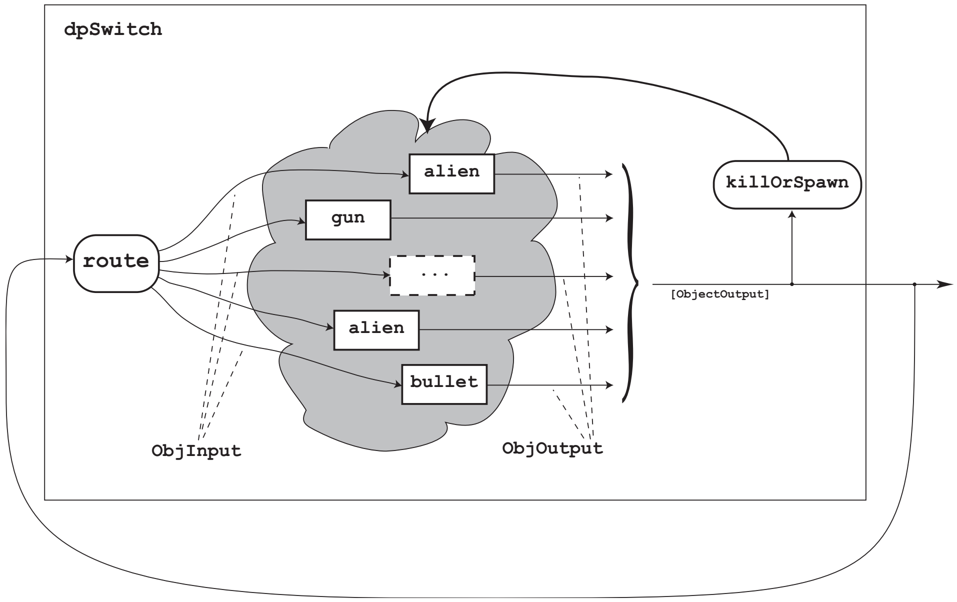


Figure 11.6: Dynamic collection of game objects maintained by dpSwitch.

dynamic collection which are under control of the user of *dpSwitch*:

- The *route* function, which specifies how the external input signal to the collection is distributed (or "routed") to individual members of the collection.
- The *killOrSpawn* function, which observes the output of the collection (and possibly the external input signal) and determines when signal functions are added to or removed from the collection.

In order to specify these two functions, we must first develop a clear understanding of how the different members of the collection interact with one another and the outside world, and under what circumstances signal functions are added to or removed from the collection. We will do that in the next section.

The Object Type

The type of *dpSwitch* requires that all signal functions in the dynamic collection to be maintained by *dpSwitch* have a uniform type. For our Space Invaders game, we use the following type for all game objects:

$$\textbf{type}\ \mathit{Object} = \mathit{SF}\ \mathit{ObjInput}\ \mathit{ObjOutput}$$

We must ensure that the types for the input and output signal ($\mathit{ObjInput}$ and $\mathit{ObjOutput}$) of this common type are rich enough to allow us to implement all necessary interactions between game objects. Based on a careful reading of the requirements in section 11.3.1, a concise specification of the interactions between game objects is as follows:

- A missile is spawned when the gun is fired.
- Missiles are destroyed when they hit an alien craft or when they self-destruct after some finite time.
- An alien craft is destroyed when it has been sufficiently damaged from a number of missile hits.

The analysis makes it clear that game objects react only to collisions with other objects and external device input. Hence we can define the $\mathit{ObjInput}$ type as follows:

$$\begin{array}{l}\textbf{data}\ \mathit{ObjInput} = \mathit{ObjInput}\{\\ \quad \mathit{oiHit} :: \mathit{Event}\ (),\\ \quad \mathit{oiGameInput} :: \mathit{GameInput}\\ \}\end{array}$$

On the output side, we observe that there are really two distinct kinds of outputs from game objects. First, there is the *observable object state*, consisting of things like the position and velocity of each object that must be presented to the user and that must be available to determine object interactions (such as collisions). Second, there are the events that cause addition or removal of signal functions from the dynamic collection, such as the gun being fired or an alien being destroyed. This leads to the following type definition:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ObjOutput} = \mathit{ObjOutput}\{\\
\quad \mathit{ooObsObjState} :: !\mathit{ObsObjState},\\
\quad \mathit{ooKillReq} \quad :: \mathit{Event}\ (),\\
\quad \mathit{ooSpawnReq} :: \mathit{Event}\ [\mathit{Object}]\\
\}
\end{array}
$$

The $\mathit{ooObsObjState}$ field contains the observable object state of type $\mathit{ObsObjState}$:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ObsObjState} =\\
\qquad \mathit{OOSGun}\{\\
\qquad\quad \mathit{oosPos} \quad :: !\mathit{Position2},\\
\qquad\quad \mathit{oosVel} \quad :: !\mathit{Velocity2},\\
\qquad\quad \mathit{oosRadius} :: !\mathit{Length},\\
\qquad \}\\
\quad\ \mid \mathit{OOSMissile}\{\\
\qquad\quad \mathit{oosPos} \quad :: !\mathit{Position2},\\
\qquad\quad \mathit{oosVel} \quad :: !\mathit{Velocity2},\\
\qquad\quad \mathit{oosRadius} :: !\mathit{Length}\\
\qquad \}\\
\quad\ \mid \mathit{OOSAlien}\{\\
\qquad\quad \mathit{oosPos} \quad :: !\mathit{Position2},\\
\qquad\quad \mathit{oosHdng} :: !\mathit{Heading},\\
\qquad\quad \mathit{oosVel} \quad :: !\mathit{Velocity2},\\
\qquad\quad \mathit{oosRadius} :: !\mathit{Length}\\
\qquad \}
\end{array}
$$

Note in the above that the constructors differentiate the different kinds of game objects. However, the oosPos, oosVel and $\mathit{oosRadius}$ fields are common to each alternative. This simplifies the implementation of many functions (such as collision detection), since collision detection can simply apply oosPos and $\mathit{oosRadius}$ to any observable object state value, without concern for the kind of game object that produced this value.

The $\mathit{ooKillReq}$ and $\mathit{ooSpawnReq}$ fields are object destruction and creation events, passed to $\mathit{killOrSpawn}$ by $\mathit{dpSwitch}$. On an occurrence of $\mathit{ooKillReq}$, the originating object is removed from the collection. On an occurrence of $\mathit{ooSpawnReq}$, the list of objects tagged to the event will be spliced in to the dynamic collection. A list rather than a singleton object is used for the sake of generality. For example, this allows an exploding alien craft to spawn a list of debris.

Gun Behavior

We now turn to describing the behavior of the game objects. This section covers the gun; the next section deals with the aliens. We leave out the missiles since that code is basically a subset of the code describing the alien behavior. Section 11.3.2 explained the basic idea behind modeling game objects using signal functions. With that as a starting point, we have to develop objects that conform to the *GameObject* type and interact properly with the world around them and the object creation and destruction mechanism.

The resulting code for the gun is very similar to what we have already seen:

```
gun :: Position2 → Object
gun (Point2 x0 y0) = proc oi → do
  let gi = oiGameInput oi
  (Point2 xd _) ← ptrPos ≺ gi
  rec
      -- Controller
    let ad = 10 * (xd − x) − 5 * v
      -- Physics
    v ← integral ≺ clampAcc v ad
    x ← (x0+) ^ << integral ≺ v
  fire ← leftButtonPress ≺ gi
  returnA ≺
    ObjOutput{
      ooObsObjState = oosGun (Point2 x y0)
        (vector2 v 0),
      ooKillReq     = noEvent,
      ooSpawnReq =
        fire `tag`
          [missile
            (Point2 x (y0 + (gunHeight / 2)))
            (vector2 v missileInitialSpeed)]
    }
missile :: Position2 → Velocity2 → Object
missile p0 v0 = proc oi → do ...
```

The only significant change is the interaction with the object creation and destruction mechanism. Note how *noEvent* is used to specify that a gun never removes itself from the dynamic collection, and how a signal function representing a new missile is tagged onto the *fire* event, yielding an object creation event.

Alien Behavior

This section presents the code describing the behavior of aliens. Like the gun, the description contains a simple physical model along with a control system, the only difference being that aliens move in two dimensions. Unlike the gun, the target for the control system is generated internally, partly through a random process. Also unlike the gun, aliens can be destroyed, which happens when their shields become depleted.

```
alien :: RandomGen g ⇒
  g → Position2 → Velocity → Object
alien g p0 vyd = proc oi → do
  rec
      -- Pick a desired horizontal position
    rx ← noiseR (xMin, xMax) g ⤙ ()
    smpl ← occasionally g 5 () ⤙ ()
    xd ← hold (point2X p0) ⤙ smpl `tag` rx
      -- Controller
    let axd = 5 * (xd − point2X p)
          − 3 * (vector2X v)
        ayd = 20 * (vyd − (vector2Y v))
        ad = vector2 axd ayd
        h  = vector2Theta ad
      -- Physics
    let a = vector2Polar
          (min alienAccMax
            (vector2Rho ad))
          h
    vp ← iPre v0 ⤙ v
    ffi ← forceField ⤙ (p, vp)
    v  ← (v0 ^+^) ^<< impulseIntegral
           ⤙ (gravity ^+^ a, ffi)
```

$$
\begin{array}{l}
\quad p \;\; \leftarrow (p0 \,.+\, \hat{}\,)\;\hat{}\,<< integral \prec v \\
\quad\quad \text{-- Shields} \\
\quad sl \;\leftarrow shield \prec oiHit\ oi \\
\quad die \leftarrow edge \prec sl \leqslant 0 \\
returnA \prec ObjOutput\{ \\
\quad ooObsObjState = oosAlien\ p\ h\ v, \\
\quad ooKillReq \quad\;\; = die, \\
\quad ooSpawnReq = noEvent \\
\quad \} \\
\textbf{where} \\
\quad v0 = zeroVector
\end{array}
$$

Picking a desired position is accomplished by occasionally sampling a noise signal. The signal function $noiseR$ generates a random signal (noise) in the specified interval. The signal function $occasionally$ is an event source for which the *average* event occurrence density can be specified. In this case, events occur on average once every 5 seconds. Thus, on each such occurrence, the noise source is sampled by tagging its value rx to the sample event $smpl$. A piecewise continuous signal indicating the desired horizontal position at all points in time is then obtained by feeding the discrete noise samples through the signal function $hold$. The typings for these signal functions are as follows:

$$
\begin{array}{l}
noiseR :: (RandomGen\ g, Random\ a) \Rightarrow \\
\quad (a, a) \rightarrow g \rightarrow SF\ ()\ a \\
occasionally :: RandomGen\ g \Rightarrow \\
\quad g \rightarrow Time \rightarrow b \rightarrow SF\ a\ (Event\ b) \\
hold :: a \rightarrow SF\ (Event\ a)\ a
\end{array}
$$

The output from the control system is the desired acceleration ad, a 2-dimensional vector. The horizontal component of this vector axd is computed based on the difference between the current horizontal position and desired horizontal position, along with the current velocity. The vertical component ayd is given by a simple proportional controller that tries to maintain a constant vertical velocity by comparing the desired vertical speed with the actual vertical speed. The direction of the desired acceleration vector in turn gives the attitude h of the alien craft.[5]

In the physical model, the real acceleration a is obtained by limiting the desired acceleration according to the capability of the craft. The velocity v and position p of the craft are then given by integration of the sum of the acceleration from the craft's engines and the gravity in two steps according to Newton's laws of motion. The force field is modeled by the auxiliary signal function $\mathit{forceField}$. It outputs an impulsive force ffi, modeled as an event, whenever an alien craft bumps into the force field. The orientation and magnitude of this impulsive force is such that the craft experiences an instantaneous, fully elastic collision. The effect of impulsive forces acting on the craft, a discontinuous change in velocity, is taken into account through the use of the signal function $\mathit{impulseIntegral}$ rather than $\mathit{integral}$ in the equation for the velocity.[6]

$$\begin{aligned}&\mathit{impulseIntegral} :: \mathit{VectorSpace}\ a\ k \Rightarrow\\&\quad \mathit{SF}\ (a, \mathit{Event}\ a)\ a\end{aligned}$$

The shield is modeled by the auxiliary signal function shield. Its output is the present shield level sl. Its input are events indicating that the craft has been hit by a missile, which causes the shield level to drop. Countering this, the shield is recharged at a constant rate, up to a maximal level. The shield level is monitored, and should it drop below zero, an event die is generated that indicates that the craft has been destroyed. The die event is defined using Yampa's *edge detector* primitive: $\mathit{edge} :: \mathit{SF}\ \mathit{Bool}\ (\mathit{Event}\ ())$.

Finally, the output signal is defined. The position p, attitude h, and velocity v make up the observable object state. The die event is made into a kill request, while $\mathit{noEvent}$ is used to specify that alien crafts do not spawn any other objects.

[5]Rotational inertia is not modeled: it is assumed that the alien craft can change direction instantaneously.

[6]The second author has developed a more systematic alternative based on Dirac impulses [55].

Maintaining Dynamic Collections

We will now put the pieces we have developed thus far together into a complete game, as outlined in figure 11.6. Thus we have to use $dpSwitch$ to maintain a dynamic collection of game objects, we have to design the control mechanism for adding and deleting objects ($killAndSpawn$), and we have to set up the proper interconnection structure by means of the $dpSwitch$ routing function and feedback.

$dpSwitch$ has the following signature:

$$
\begin{aligned}
& dpSwitch :: Functor\ col \Rightarrow \\
& \quad (forall\ sf \circ (a \rightarrow col\ sf \rightarrow col\ (b, sf))) \\
& \quad \rightarrow col\ (SF\ b\ c) \\
& \quad \rightarrow SF\ (a, col\ c)\ (Event\ d) \\
& \quad \rightarrow (col\ (SF\ b\ c) \rightarrow d \rightarrow SF\ a\ (col\ c)) \\
& \quad \rightarrow SF\ a\ (col\ c)
\end{aligned}
$$

The first argument is the routing function. Its purpose is to pair up each running signal function in the collection maintained by $dpSwitch$ with the input it is going to see at each point in time. The rank-2 universal quantification of sf renders the members of the collection opaque to the routing function; all the routing function can do is specify how the input is distributed. The second argument is the initial collection of signal functions. The third argument is a signal function that observes the external input signal and the output signals from the collection in order to produce a switching event. In our case, this is going to be the $killAndSpawn$ function alluded to in figure 11.6. The fourth argument is a function that is invoked when the switching event occurs, yielding a new signal function to switch into based on the collection of signal functions previously running and the value carried by the switching event. This allows the collection to be updated and then switched back in, typically by employing $dpSwitch$ again.

The collection argument to the function invoked on the switching event is of particular interest: it captures the *continuations* of the signal functions running in the collection maintained by $dpSwitch$ at the time of the switching event, thus making it possible to preserve their state across a switch. Since the continuations are plain, ordinary signal functions, they can be resumed, discarded, stored, or combined with other signal functions.

In order to use $dpSwitch$, we first need to decide what kind of collection to use. In cases where it is necessary to route specific input to specific signal functions in the collection (as opposed to broadcasting the same input to everyone), it is often a good idea to "name" the signal functions in a way that is invariant with respect to changes in the collection. For our purposes, an association list will do just fine, although we will augment it with a mechanism for generating names automatically. We call this type an *identity list*, and its type declaration along with the signatures of some useful utility functions, whose purpose and implementation should be fairly obvious, are as follows:

$$
\begin{array}{l}
\textbf{type}\ \mathit{ILKey} = \mathit{Int} \\
\textbf{data}\ \mathit{IL}\ a = \mathit{IL}\{\mathit{ilNextKey} :: \mathit{ILKey}, \\
\quad \mathit{ilAssocs} :: [(\mathit{ILKey}, a)]\} \\
\mathit{emptyIL} :: \mathit{IL}\ a \\
\mathit{insertIL\_} :: a \rightarrow \mathit{IL}\ a \rightarrow \mathit{IL}\ a \\
\mathit{listToIL} :: [a] \rightarrow \mathit{IL}\ a \\
\mathit{elemsIL} :: \mathit{IL}\ a \rightarrow [a] \\
\mathit{assocsIL} :: \mathit{IL}\ a \rightarrow [(\mathit{ILKey}, a)] \\
\mathit{deleteIL} :: \mathit{ILKey} \rightarrow \mathit{IL}\ a \rightarrow \mathit{IL}\ a \\
\mathit{mapIL} :: ((\mathit{ILKey}, a) \rightarrow b) \rightarrow \mathit{IL}\ a \rightarrow \mathit{IL}\ b
\end{array}
$$

IL is of course also an instance of $Functor$. Incidentally, associating some extra state information with a collection, like $ilNextKey$ in this case, is often a quite useful pattern in the context of $dpSwitch$.

Let us use $dpSwitch$ to implement the core of the game:

$$
\begin{array}{l}
\mathit{gameCore} :: \mathit{IL}\ \mathit{Object} \\
\quad \rightarrow \mathit{SF}\ (\mathit{GameInput}, \mathit{IL}\ \mathit{ObjOutput}) \\
\quad\quad (\mathit{IL}\ \mathit{ObjOutput}) \\
\mathit{gameCore}\ \mathit{objs} =
\end{array}
$$

$$
\begin{array}{l}
\mathit{dpSwitch}\ \mathit{route} \\
\quad \mathit{objs} \\
\quad (\mathit{arr}\ \mathit{killOrSpawn} \ggg \mathit{notYet}) \\
\quad (\lambda \mathit{sfs}'\ f \rightarrow \mathit{gameCore}\ (f\ \mathit{sfs}'))
\end{array}
$$

We will return to the details of the routing function and $\mathit{killOrSpawn}$ below. But the basic idea is that the switching event from $\mathit{killOrSpawn}$ carries a function that when applied to the collection of continuations yields a new signal function collection to switch into. That in turn is achieved by invoking $\mathit{gameCore}$ recursively on the new collection.

$\mathit{killOrSpawn}$ in a plain Haskell function that is lifted to the signal function level using arr. The resulting signal function is composed with $\mathit{notYet} :: \mathit{SF}\ (\mathit{Event}\ a)\ (\mathit{Event}\ a)$ that suppresses initial event occurrences. Thus the overall result is a source of kill and spawn events that will not have any occurrence at the point in time when it is first activated. This is to prevent $\mathit{gameCore}$ from getting stuck in an infinite loop of switching. The need for this kind of construct typically arises when the source of the switching events simply passes on events received on its input in a recursive setting such as the one above. Since switching takes no time, the new instance of the event source will see the exact same input as the instance of event source that caused the switch, and if that input is the actual switching event, a new switch would be initiated immediately, and so on for ever.

The routing function is straightforward. Its task is to pass on the game input to all game objects, and to detect collisions between any pair of interacting game objects and pass hit events to the objects involved in the collision:

$$
\begin{array}{l}
\mathit{route} :: (\mathit{GameInput}, \mathit{IL}\ \mathit{ObjOutput}) \rightarrow \mathit{IL}\ \mathit{sf} \\
\quad \rightarrow \mathit{IL}\ (\mathit{ObjInput}, \mathit{sf}) \\
\mathit{route}\ (\mathit{gi}, \mathit{oos})\ \mathit{objs} = \mathit{mapIL}\ \mathit{routeAux}\ \mathit{objs} \\
\quad \mathbf{where} \\
\quad\quad \mathit{routeAux}\ (k, \mathit{obj}) = \\
\quad\quad\quad (\mathit{ObjInput}\{\ \mathit{oiHit} = \mathbf{if}\ k \in \mathit{hs} \\
\quad\quad\quad\quad \mathbf{then}\ \mathit{Event}\ ()
\end{array}
$$

```
          else noEvent,
          oiGameInput = gi},
          obj)
    hs = hits (assocsIL
          (fmap ooObsObjState oos))
```

route invokes the auxiliary function *hits* that computes a list of keys of all objects that are involved in some collision. For all game objects, it then checks if the key of that object is contained in the list of colliding objects. If so, it sends a collision event to the object, otherwise not. *hits* performs its computation based on the fed-back object output. This gives the current position and velocities for all game objects. Two objects are said to collide if they partially overlap and if they are approaching each other. However, alien crafts do not collide in this version of the game.

killOrSpawn traverses the output from the game objects, collecting all kill and spawn events. If any event occurs, a switching event is generated that carries a function to update the signal function collection accordingly:

```
killOrSpawn :: (a, IL ObjOutput)
    → (Event (IL Object → IL Object))
killOrSpawn (_, oos) =
  foldl (mergeBy (∘)) noEvent es
  where
    es :: [Event (IL Object → IL Object)]
    es = [mergeBy (∘)
      (ooKillReq oo
         `tag` (deleteIL k))
      (fmap (foldl (∘) id
         ∘ map insertIL_)
         (ooSpawnReq oo))
      | (k, oo) ← assocsIL oos]
```

A kill event is turned into a function that removes the object that requested to be deleted by partially applying *deleteIL* to the key of the object to be removed. A spawn event is turned into a function that inserts all objects in the spawn request into the collection using *insertIL_*. These individual functions are then simply composed into a single collection update function. We have found this approach to collection updating to be quite useful and applicable in a wide range of contexts [56].

Closing the Feedback Loop

We can now take one more step toward the finished game by closing the feedback loop. We also add features for game initialization and score keeping. The function *game* plays one round of the game. It generates a terminating event carrying the current (and possibly final) score either when the last alien craft in the current wave of attack is destroyed, or when the game is over due to an alien touch down:

```
game :: RandomGen g ⇒
  g → Int → Velocity → Score →
  SF GameInput ((Int, [ObsObjState]),
    Event (Either Score Score))
game g nAliens vydAlien score0 = proc gi → do
  rec
    oos ← gameCore objs0 ⤙ (gi, oos)
  score ← accumHold score0
          ⤙ aliensDied oos
  gameOver ← edge ⤙ alienLanded oos
  newRound ← edge ⤙ noAliensLeft oos
  returnA ⤙ ((score,
    map ooObsObjState
          (elemsIL oos)),
    (newRound `tag` (Left score))
    `lMerge` (gameOver
       `tag` (Right score)))
  where
    objs0 =
      listToIL
        (gun (Point2 0 50)
           : mkAliens g (xMin + d) 900 nAliens)
```

The central aspect of this function is the closing of the feedback loop using the recursive arrow syntax. The arguments to *game* are a random number generator (used to seed the signal functions representing the alien crafts), the number of alien crafts in this wave of attack, the desired landing speed of the aliens, and an initial score carried over from any preceding rounds. Score is kept by simply counting kill requests from aliens in the object output, and events signaling a new round and game over is obtained by applying edge detectors to predicates over the object output looking for the absence of alien crafts and the landing of an alien craft, respectively.

An unsatisfying aspect of the current design of the Yampa switching combinators is that the exact choice of combinator in $gameCore$ (here $dpSwitch$) is critical to the design of $game$. This point is discussed further in section 11.3.3.

Playing Multiple Rounds

Finally, a multi-round game can be built on top of $game$. After each successful defeat of a wave of invaders, $game$ is reinvoked with more and faster alien crafts, passing on the current score. Once an alien has landed, the game starts over from the beginning.

$$
\begin{array}{l}
\mathit{multiRoundGame} :: \mathit{RandomGen}\ g \Rightarrow \\
\quad g \rightarrow \mathit{SF}\ \mathit{GameInput}\ (\mathit{Int}, [\mathit{ObsObjState}]) \\
\mathit{multiRoundGame}\ g = \mathit{rgAux}\ g\ \mathit{nAliens0}\ \mathit{vydAlien0}\ 0 \\
\quad \textbf{where} \\
\qquad \mathit{nAliens0} = 2 \\
\qquad \mathit{vydAlien0} = -10 \\
\qquad \mathit{rgAux}\ g\ \mathit{nAliens}\ \mathit{vydAlien}\ \mathit{score} = \\
\qquad\quad \mathit{switch}\ (\mathit{game}\ g'\ \mathit{nAliens}\ \mathit{vydAlien}\ \mathit{score}) \\
\qquad\qquad \$\ \lambda \mathit{status} \rightarrow \\
\qquad\quad \textbf{case}\ \mathit{status}\ \textbf{of} \\
\qquad\qquad \mathit{Left}\ \mathit{score}' \rightarrow \\
\qquad\qquad\quad \mathit{rgAux}\ g'' \\
\qquad\qquad\qquad (\mathit{nAliens} + 1) \\
\qquad\qquad\qquad (\mathit{vydAlien} - 10) \\
\qquad\qquad\qquad \mathit{score}' \\
\qquad\qquad \mathit{Right}\ \mathit{finalScore} \rightarrow \\
\qquad\qquad\quad \mathit{rgAux}\ g''\ \mathit{nAliens0}\ \mathit{vydAlien0}\ 0 \\
\qquad\quad \textbf{where}
\end{array}
$$

$$(g', g'') = split\ g$$

All that then remains is to connect the top-level signal function to the outside world. This involves feeding in a signal of mouse positions and button presses, and mapping the output signal pointwise to a suitable graphic representation.

Which Switch?

Yampa provides a family of parallel switching combinators. Two members are $pSwitch$ and $dpSwitch$ that have exactly the same type signature, and as mentioned in section 11.3.3, which one is chosen can have a significant impact on the design of a program.

The difference between $pSwitch$ and $dpSwitch$ is that the output from the switcher *at the point of a switching event* in the former case is determined by the signal function being switched into, which in turn usually means from the outputs of the signal functions in a *new*, updated collection, whereas the output is the latter case is given by the output from the signal functions in the *old* collection. This allows $dpSwitch$ to be *non-strict* in the switching event; i.e., the output from $dpSwitch$ at any point in time can be determined without demanding the switching event at that same point in time. The "d" in the name of $dpSwitch$ stands for "delayed", meaning that the effect of a switch cannot be observed immediately. All Yampa switchers have delayed versions, and all those are non-strict in the switching event.

Employing a switcher that is non-strict in the switching event may be enough to make it possible to close a feedback loop without any unit delay on the feedback path. In our case the lazy demand structure is such that this indeed is the case, and hence there is no unit delay ($iPre$) on the feedback path in $game$ in section 11.3.3.

Using *dpSwitch* also means that the requests for removal from the objects are going to be visible outside the switcher. This was exploited for the score keeping mechanism. Had *pSwitch* been used, we would only be able to observe the output from the objects *remaining* after a switch. This does of course not include the output from objects that just removed themselves by emitting kill requests, and these requests are exactly what is counted for keeping score. A more robust alternative would be to associate extra state information with the collection type (like the counter used for naming in the identity list IL of section 11.3.3) and use that to keep score.

11.4 Evaluation

Fruit provides a formal model of user interfaces, and demonstrates that this model can be used as the basis for a GUI toolkit. But is there any practical benefit to functional modeling? After all, an experienced GUI programmer could implement the media player example in a few minutes using their favorite imperative language and GUI toolkit. At first glance, the specification in figure 11.3 (or its corresponding textual syntax) may even seem somewhat *more* complicated than a corresponding imperative program, since it involves both an explicit *hold* operator to introduce local state and a feedback loop.

To see why Fruit specifications are useful, consider how the media controller might be implemented in a modern, object-oriented imperative toolkit, such as Java/Swing. A good object-oriented design would encapsulate the current state of the media controller into a *model* class that supports registration of *listener* classes to be notified when the model's state is updated. At initialization time, the implementation would create the model and the button instances, register listeners on

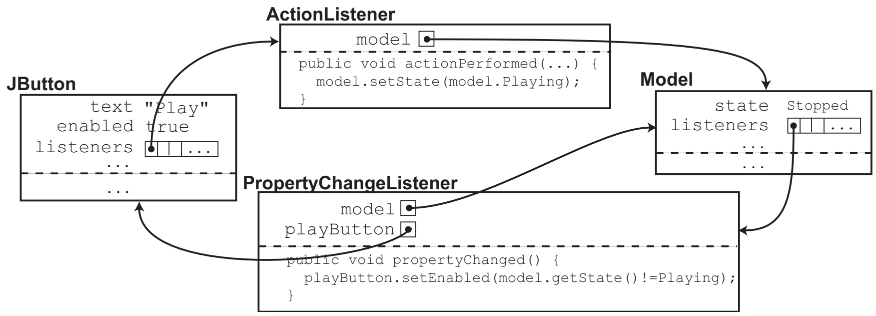


Figure 11.7: Runtime Heap in Java/Swing Implementation

the model instance that update the enabled property of the buttons, and register listeners on each button instance that update the state of the model, as illustrated in figure 11.7. As this diagram illustrates, a feedback loop exists at runtime in this object-oriented imperative implementation, just as it does in the Fruit specification. In fact, a more accurate diagram would repeat this cyclic graph structure once for each of the other two buttons, with each sub-graph sharing the same model instance – a considerably more complex structure than figure 11.3.

The key difference between figures 11.3 and 11.7 is that the former is a diagram of a static specification, while the latter is a visualization of a partial snapshot of the heap at runtime. In the Swing implementation, the feedback loops are hidden from the programmer in the listener lists in the implementation of the model and button classes. Even with whole program analysis, there is no reliable, systematic way for either the programmer or a programming environment to recover figure 11.7 directly from the source code of the Java/Swing implementation. In contrast, figure 11.3 is isomorphic to the (static) text of the specification. In short, a Fruit specification differs from an imperative implementation by making data flow dependencies explicit in the specification.

So why is it useful to specify data flow dependencies explicitly?

First, explicit dependencies encourage programmers to think in terms of time-invariant *relationships* between different components of the application. The considerable literature on constraints has made the case quite well that this is a higher-level view of user interfaces. Instead of writing event handlers that update mutable objects in response to individual events, the Fruit model encourages writing declarative *equations* that specify the relationships between components of the interface that *must hold* at every point in time.

The data flow style also eliminates a small but important class of programming errors. In traditional imperative event handlers, every event handler must include code to update all of the appropriate objects in response to the event. A common source of subtle bugs in imperative GUI programs is forgetting to update some particular object in response to a particular event, or (even worse) updating the local state of an object, but forgetting to notify registered listeners. In contrast, point-wise dependencies in Fruit are propagated automatically by the implementation.

Making data flow dependencies explicit also enables precise specification of *design patterns* related to data flow. For example, the classic Model / View / Controller (MVC) design pattern [42] enables multiple interactive views of the same underlying data set, and has become the cornerstone of modern object oriented GUI toolkits. The essence of MVC is decoupling of the time-varying application state (the *model*) from the graphical interface, so that the model may be observed and updated by multiple user interface objects. This decoupling can be expressed in Fruit by simply decoupling the state accumulation primitive ($hold$, in the media controller example) from the rest of the GUI. Multiple views and controllers may then be wired to share the same model, and this sharing will be manifest in the specification itself.

Finally, using data flow dependencies as the exclusive mechanism for communication between components of the application enables simple, precise reasoning about causal relationships directly from the specification. For example:

- (forward reasoning): *"What effect does pressing the 'Play' button have on the program state?"* This is easily determined from figure 11.3 by tracing the path from the play button to the $state$ signal.

- (backwards/dependency reasoning): *"What GUI components affect the state?"* This is easily determined by tracing backwards along all signal lines that enter the $hold$ primitive that forms the $state$ signal.

- (component interactions): *"How does the 'Play' button affect the 'Pause' button?"* This is determined by tracing the directed path from the first component to the second. Note that if the second component is not reachable from the first, then, since a functional specification can have no hidden side effects, the first component has no effect whatsoever on the second component.

In an imperative setting (even an object-oriented one), this kind of reasoning is simply not tractable. Imperative GUI implementations coordinate their activities via *side effects*: One callback writes to a variable or property that is subsequently read by others. Since any callback may directly or indirectly invoke a method or function that updates the global mutable state used by some other callback, there is no practical method for reasoning about or controlling interactions between different parts of the user interface.

11.5 Chapter Summary

This chapter presented three examples of increasingly complexity of interactive graphical applications developed with Fruit and Yampa: a media controller, a web browser with interactive history, and a video game. These examples provide evidence that Fruit is capable of modeling realistic user interfaces. Section 11.4 compared a Fruit-style specification with a traditional imperative implementation. One benefit of the Fruit approach is that dependencies between components are made explicit in the static specification. This makes it easy for the programmer to see and understand causal relationships between interface components in the Fruit specification, while such relationships would be difficult or impossible to derive from a traditional imperative program.

Part III

Conclusions and Future Work

Chapter 12

Related Work, Conclusions and Future Work

12.1 Related Work

12.1.1 Data Flow Languages

Data flow models and languages date back to the sixties [64, 41]. Esterel [7], Lustre [11, 27], Lucid Synchrone [13, 65] and Signal [26] are examples of synchronous data flow languages oriented towards control of real-time systems.

Yampa is intended to be a robust and expressive implementation of FRP, capable of describing reactive systems with a highly dynamic structure, such as graphical user interfaces or vision-based robot control systems [62], while retaining the fundamental advantages of the synchronous programming paradigm. Performance *guarantees* for space and time have so far been a secondary concern, although we have gone to great lengths to ensure that the system runs as smoothly as possible in practice. This puts Yampa in marked contrast to synchronous languages, as well as the RT-FRP line of work [79], where central aspects are guaranteed reactivity, execution in bounded space and time, and efficient implementation (including compilation to digital circuits [8]), but at the expense of requiring a fairly rigid system structure. For example, the closest thing there is to a `switch`-like construct in Lucid Synchrone is a reset operator [28], which causes a stream computation to start over.

Yampa shares with hybrid systems languages an inherent notion of continuous time and event-like abstractions for capturing discrete aspects of a system. However, the underlying numeric machinery of Yampa is currently much more simplistic than what is typical for hybrid modeling and simulation languages. For example, accurate location of the time of an event occurrence is often considered critical and requires complex algorithms in combination with language restrictions to be computationally tractable. Similarly, these languages often use sophisticated algorithms for integration with variable step size to ensure rapid computation as well as accurate results. The Yampa implementation does not currently do any of this. However, the ability of Yampa to express structurally dynamic systems, which typical hybrid modeling languages cannot deal with cleanly, makes it an interesting topic of future research to attempt to address such numerical concerns within Yampa.

Fudgets [10] has been a source of inspiration for the Yampa implementation. However, the asynchronous nature of Fudgets make it fundamentally different from Yampa. There is certainly an overlap of possible application domains (such as graphical user interfaces), but for areas where time and synchrony is inherent (animation, hybrid systems), we believe that a synchronous language is the obvious choice.

The continuation-based implementation of Yampa also bears a clear resemblance to other work in the area, such as the co-iterative characterization of Lucid

Synchrone [12], the operational semantics of RT-FRP [79], and the "residual behaviors" implementation of Fran [20].

Jacob et al [40] propose a data flow model for user interfaces, including both continuous variables and discrete event handlers. However, their model focuses on modeling "post-WIMP" user interaction, and is cast in an imperative, object-oriented setting. In contrast, the Fruit model demonstrates that the data flow model is applicable even in the classical WIMP setting, and and does not depend on objects or imperative programming. As discussed in section 11.4, we believe that using data flow as the sole basis for our specifications makes reasoning about specifications much more tractable.

12.1.2 Imperative GUI Toolkits

Java / Swing

Java's Swing User Interface Toolkit [72] represents the current industrial state-of-the-art for a GUI Toolkit. Swing is a GUI toolkit library implemented entirely in Java, depending only on the Java2D rendering engine for 2D graphics. Swing is a heavily object-oriented design, in that it makes extensive use of inheritance to capture relationships between similar GUI component types, follows a number of *design patterns* [24] (such as Model-View-Controller [42]), and is built around the Java Beans component model [18].

Component Models and Java Beans

In the Java community, recent work has produced the Java Beans component model [18]. The Java Beans component model prescribes a set of programming conventions for writing re-usable software components. A programmer writes a

Java Beans component by defining a Java class that specifies a set of *events* ("interesting" conditions which result in notifying other objects of their occurrence) and *properties* (named mutable attributes of the component that may be read or written with appropriate methods). A visual builder tool uses Java's introspection facilities [71] to discover the events and properties exported by the component class. Many of the classes in the standard Java class libraries (such as those of AWT and Swing) are defined as Java Beans components.

Interactors and Garnet

Interactors [50] are a set of objects responsible for handling input in the Garnet toolkit. Myers observed that most toolkits communicate using a stream of low-level device-dependent input events, and identified a set of six high-level parameterized interactor classes that cover the input requirements of most GUI-based applications. For example, different parameters enable the Move-Grow interactor to serve as the input controller for a scroll-bar elevator, or to allow dragging of objects in an interactive drawing program.

12.1.3 Functional GUI Toolkits

eXene

Gansner and Reppy developed eXene [25], an X Window System Toolkit for ML. The eXene library is based on Reppy's Concurrent ML (CML), which adds concurrency primitives to ML, and first-class event channels to ML. The event model in CML is quite similar to the event algebra of FRP, but eXene relies on ML's imperative programming features to handle state, whereas Fruit models mutable state explicitly using Yampa's facilities for feedback and state accumulating signal functions.

Fudgets

Fudgets [10] is a functional GUI toolkit for Haskell based on stream processors. Fudgets extends the stream-based I/O system of older versions of Haskell with request and response types for the X Window System. The programming model of Fudgets is very similar to that of Yampa, although Fudgets is based on discrete, asynchronous *streams*, whereas FRP is based on continuous, synchronous *signals*.

FranTk

FranTk [68] uses the Fran reactive programming model (and its combinators) to specify the connections between user interface components. However, FranTk uses an imperative model for creating widgets, maintaining program state (with mutable variables or "MVars"), and wiring of cyclic connections (which occur in most GUIs).

12.1.4 Constraints

There has been a great deal of work in the CHI community in the area of constraints for graphical user interfaces. In general, constraints allow the declarative specification of relationships that should be maintained in the user interface. Constraints have been successfully used for flexible visual layout [37], maintaining connections between data objects and views of those objects [33], controlling animation [9] and as the basis for composing interactors [50].

The use of constraints for user interfaces was pioneered in Sketchpad [73], and used in the Garnet [49], Amulet [53] and SubArctic [38] toolkits. However, these toolkits used constraints to augment an essentially imperative, object-oriented programming model. Our work differs from this previous work in the use of point-wise functions (i.e. one-way constraints) as the basis of a formal model that does not include objects or imperative state.

12.1.5 Formal Models of GUIs

There have been numerous previous proposals for formal models of graphical user interfaces, such as User Action Notation (UAN) [31] or Paterno's ConcurTaskTrees [59]. The emphasis in most of these formalisms is typically on modeling *user tasks* [60], i.e. a logical description of actions to be performed by the user to achieve certain goals. Such *task models* fit somewhere between requirements specification and design in the classical software engineering process. In contrast, the Fruit model is focused solely on user interface *implementations*. Task models are typically very high level, focused on the (human) user, and are not directly executable. In contrast, Fruit specifications are comparatively low level, make no direct mention of the user, and *are* directly executable.

Another formalism for modeling user interface implementations is Palanque's Petri net based Interactive Cooperative Objects (ICO) [58]. Like Fruit, ICO enables the programmer to give a precise specification of the run-time behavior of a graphical user interface. The core model of ICOs is Petri Nets, a simple formalism with well-understood semantics. ICOs allow Petri net models to be organized into object-oriented interfaces, in which an object's reaction to method invocations is specified by the Petri net. One key difference between Fruit and ICOs is that where ICOs use objects to organize core models into higher level abstractions, Fruit uses a (functional) host language to provide abstraction capabilities and general computation. An important consequence of embedding in a functional language is that Fruit models retain reasoning power and semantic clarity while still being directly executable.

12.2 Conclusions

The benefit of functional programming is that its underlying formal basis enables simple, precise reasoning about programs using equational reasoning. However, most modern functional languages in widespread use make some provision for imperative programming, either directly via imperative features (such as ML's mutable references) or in a more restricted way (such as Haskell's IO monad). The provision of imperative programming constructs in functional languages has made it very easy to provide functional languages with access to the operating system and I/O facilities, which are usually assumed to be somehow *inherently* imperative. With only one notable exception beyond the work presented in this thesis[1], all toolkits for programming graphical user interfaces that we are aware of, even those for functional programming languages, present the programmer with an imperative programming interface.

This dissertation has presented three libraries: *Yampa* – for programming reactive systems in a synchronous dataflow style, *Haven* – for creating 2D vector graphics images, and *Fruit* – for specifying interactive graphical user interfaces using Haven and Yampa. The unifying theme of these libraries is *pure* functional programming: they allow the programmer to specify systems using only the functional core of Haskell, and make no appeal to the I/O monad or other imperative programming constructs.

One contribution of this thesis is simply to demonstrate that the functional

[1] Carlsson and Hallgren's Fudgets system [10]

reactive programming model can be used to write declarative executable specifications of graphical user interfaces. The examples presented in chapters 10 and 11 provide concrete evidence that we have achieved this goal. However, considerable revision of the original functional reactive programming model and implementations were necessary to achieve this goal. To address the limits of expressive power and scalability of Fran and FRP described in chapter 3, we had to create Yampa, an adaptation of the functional reactive programming model to the Arrows computational framework, as described in chapters 2 and 4. Chapter 5 presented the concrete implementation of the operational semantics of chapter 4, as well as some basic dynamic optimizations. To account for certain dynamic user interfaces, we extended Yampa with features for parallel switching and dynamic collections, as described in chapter 8.

A secondary contribution of this thesis is to show that our purely functional model allows us to give a precise account of common GUI programming idioms and enables simple practical reasoning about GUI programs. Chapter 7 included an account of applying spatial transforms to a GUI, for both layout and zooming. Chapter 10 showed how variations on the Model / View / Controller design pattern could be specified in just few lines of Yampa code. Chapter 9 described how the operational semantics of chapter 4 could be used to prove properties of Yampa programs using equational reasoning and induction, and chapter 11 explored how the explicit data flow dependencies enable important causal relationships in a user interface to be derived from the data flow graph by inspection.

12.3 Future Work

This section describes ideas for future work that can build on what has been presented in this thesis.

12.3.1 Incremental Implementation

Large graphical user interfaces may involve tens or hundreds of GUI components (buttons, sliders, etc.), all of which are executing in parallel. In the Fruit GUI model, each such component produces an output signal consisting of the component's visual appearance and auxiliary semantic output. As described in chapter 5, Yampa is implemented by discrete sampling. At any given sample time, most of the signals in a large GUI program will have the same value for the current time step as they had in the previous time step, since the user typically only interacts with one GUI component at a time. Unfortunately, however, the current implementations of FRP and Yampa use a "pull" based implementation model, which results in sampling every signal function in the Yampa program at every sample time. This results in several significant sources of inefficiency, including the computational overhead of redundant sampling, and the memory/video bandwidth required to redraw the complete screen image on every time step.

An alternative approach would be to use a "push" model for propagation of signal values, based on the idea of *change propagation*. Instead of lazily "pulling" sample values from the output of the data flow graph, the implementation would "push" information about how a signal has changed since the last sample through the data flow graph, starting at the inputs.

We invested considerable time attempting to develop such an implementation within the Haskell embedding of Yampa. This turned out to require a small but rather significant modification of the Yampa programming model of chapter 5. The difficulty arose with the implementation of $loop$, since there is no way to "push" information through the feedback path of a loop without first "pulling" to know the previous value of the fed-back signal. A solution to this problem was to abandon the general purpose $loop$ combinator, and instead require the programmer to explicitly use a $loopIntegral$ or $loopDelay$ combinator that provided sufficient information about how to break such causality loops. Unfortunately, this meant that the Arrows syntactic could no longer be used to specify feedback loops, and it was not entirely clear what the consequences would be of nesting loop constructs.

It would be extremely interesting to explore direct compilation of the Mini-Yampa language presented in chapter 4, as it may be possible to achieve an incremental implementation of Yampa if one were not operating within the constraints of embedding Yampa in Haskell. If such a project were successful, an interesting next step would be to explore whether, by means of sophisticated preprocessing, one might return to using all of Haskell as a base language, instead of just a simple non-strict λ-calculus.

12.3.2 Integration with Standard Widget Sets

The Fruit toolkit presented in chapter 7 is implemented using only the Yampa and Haven libraries and the pure functional core of Haskell. Restricting ourselves to these purely functional models is what enables precise reasoning about implementation artifacts, since every GUI component is assured of having a clear denotation in terms of the Yampa model.

While this direct approach to the implementation of Fruit enables formal reasoning, it has significant drawbacks as a practical implementation technique. First, implementing individual GUI components such as buttons, sliders, etc. requires considerable effort, which is why Fruit only implements a tiny fraction of the components found in other modern GUI toolkits. Second, it is simply not possible for such a direct implementation of GUI components to completely and accurately implement the native platform look and feel, even for a single platform. There are two reasons for this. First, platform look and feel is only documented partially and informally in "style guides". Because such specifications are highly informal and usually written more for application developers than toolkit implementors, it seems unlikely that a style could guide could ever provide sufficient detail to accurately specify all relevant details of component behavior. Second, the set of components on a platform and the interaction styles they support are necessarily a moving target. New components and interaction styles are constantly being invented and added to modern window system toolkits, often without adding relevant detail to the corresponding style guide. These limitations prevent Fruit from being a production toolkit suitable for practical application development.

For Fruit to be useful as a production toolkit, it would be necessary to find a way to implement Fruit's GUI components using an existing (imperative) GUI toolkit. This would not necessarily mean abandoning Fruit's purely functional model or programming interface. For example, it should be possible to defer invoking the imperative GUI toolkit routines to the level of $reactimate$. Communication between GUI components and layout combinators would involve request/response style interaction using Yampa's Events, very much akin to the approach of Fudgets. The most challenging aspect of this approach will be in maintaining an accurate mapping between the Yampa data flow graph and the (imperative) widget hierarchy, particularly in the presence of Yampa's flexible switching constructs.

12.3.3 Modeling Systems Software

Most real GUI-based applications need to interact with the outside world in some way, by (for example) reading or writing files to the filesystem, making network connections, etc. These other aspects of the operating system do not have purely functional programming interfaces. A purely functional model of these external systems seems implausible because of issues such as non-determinism and exotic operational properties (like UNIX filesystem semantics). How should an FRP program interact with such services? Can we define some reasonable interface to such services that is consistent with the FRP programming model?

12.3.4 Model-Based Interface Design

Many tools have been proposed to assist in deriving executable user interface implementations from high-level task model specifications [74]. Unfortunately, in most cases the precise semantics of such tools and the code that they generate is an implicit property of the tool's implementation. In contrast, Fruit has a clear, implementation-independent semantics, but operates at a relatively low level of abstraction: graphics and input devices. It would be very interesting to use Yampa as a foundation for writing high-level task models, and to develop a tool that can systematically map such a high-level model into a lower-level executable Fruit specification.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Adobe Systems. *Adobe Illustrator*. Classroom in a book. Adobe Press, Mountain View, CA, USA, 1996.

[3] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley, Reading, MA, USA, third edition, 1999.

[4] Adobe Systems Incorporated. *PDF reference: Adobe portable document format, version 1.4*. Addison-Wesley, Reading, MA, USA, third edition, 2001.

[5] Apple Computer, Inc. Beyond quickdraw: Quartz. *Apple Developer Connection*, June 2001.

[6] Ben Bederson, Jon Meyer, and Lance Good. Jazz: An extensible zoomable user interface graphics toolkit in java. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, pages 171–180. ACM, 2000.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):217–248, 1992.

[8] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing Series. MIT Press, 2000.

[9] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, 1986.

[10] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, March 1998.

[11] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[12] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.

[13] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Submitted for publication, 2000.

[14] Antony Courtney. Engineering insights from an interactive imaging application. *The X Resource: A Practical Journal of the X Window System*, Fall 1991.

[15] R. L. Crole and A. D. Gordon. A sound metalogical semantics for input/output effects. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer science logic: 8th workshop, CSL '94*, volume 933, pages 339–353, Berlin, Heidelberg, and New York, 1995. Springer-Verlag.

[16] Alan Dix and Colin Runciman. Abstract models of interactive systems. In *Proceedings of the HCI'85 Conference on People and Computers: Designing the Interface*, The Design Process: Models and Notation for Interaction, pages 13–22, 1985.

[17] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):C25–C36, 1993.

[18] Graham Hamilton (editor). *Java Beans API Specification 1.01*. Sun Microsystems, 1997.

[19] Tim Bray (editor). *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium (W3C), October 2000.

[20] Conal Elliott. Functional implementations of continuous modelled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[21] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[22] Conal Elliott. Functional images. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Palgrave, 2003.

[23] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[25] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In *User Interface Software, Bass and Dewan (Eds.)*, volume 1, pages 61–80. John Wiley & Sons, 1993.

[26] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[27] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[28] Grégoire Hamon and Marc Pouzet. Modular resetting of synchronous data-flow programs. In *Principles and Practice of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[29] Vincent J. Hardy. *Java 2D API Graphics*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[30] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[31] H. Rex Hartson and Philip D. Gray. Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, 7(1):1–45, 1992.

[32] Matthew C. B. Hennessy and Gordon D. Plotkin. Full abstraction for a simple parallel programming language. In *Mathematical Foundations of Computer Science*, pages 108–120, 1979.

[33] Ralph D. Hill. The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *Proceedings of CHI'92*, pages 335–342, 1992.

[34] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[35] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, Cambridge, UK, 2000.

[36] Paul Hudak, Antony Courtney, Henrik Nilsson, and John 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, 2003.

[37] Scott E. Hudson and Shamim P. Mohamed. Interactive specification of flexible user interface displays. *Information Systems*, 8(3):269–288, 1990.

[38] Scott E. Hudson and Ian E. Smith. Ultra-lightweight constraints. In *ACM Symposium on User Interface Software and Technology*, pages 147–155, 1996.

[39] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, (37):67–111, 2000.

[40] Robert J. K. Jacob, Leonidas Deligiannidis, and Stephen Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

[41] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM J. Applied Math. 14*, (6):1390–1411, November 1966.

[42] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system, 1988.

[43] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[44] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998.

[45] Macromedia Corporation. *Macromedia Flash MX Product Information*, 2002.

[46] M. Morris Mano. *Digital Design*. Prentice Hall, 2001.

[47] Microsoft Corporation. *Microsoft Foundation Classes (MFC) Documentation*, 1997.

[48] Brad Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the Fourth Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, November 1991.

[49] Brad A. Myers. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, 1990.

[50] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, 1990.

[51] Brad A. Myers, editor. *Languages for Developing User Interfaces*. Jones and Bartlett Publishers, 1992.

[52] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Computer Science Department, Carnegie-Mellon University, July 1993.

[53] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997.

[54] Netscape Communications Corporation. *XUL Programmer's Reference Manual*. Mozilla Project, 2002. `http://www.mozilla.org/xpfe/xulref/`.

[55] Henrik Nilsson. Functional automatic differentiation with dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, August 2003. ACM Press.

[56] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[57] David Padua. The fortran I compiler. *Computing in Science and Engineering*, 2:70–75.

[58] Phillipe Palanque and Rémi Bastide. Interactive Cooperative Objects : an Object-Oriented Formalism Based on Petri Nets for User Interface Design. In *IEEE / System Man and Cybernetics 93*, pages 274–285. Elsevier Science Publisher, October 1993.

[59] Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer-Verlag, 1999.

[60] Fabio Paterno. Task models in interactive software systems. In S. K. Chang, editor, *Handbook of Software Engineering & Knowledge Engineering*. World Scientific Publishing Co., 2001.

[61] Ross Paterson. A new notation for arrows. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, September 2001.

[62] Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, October 2002.

[63] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. *Computer Graphics*, 27(Annual Conference Series):57–72, 1993.

[64] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut f¨ur Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[65] Marc Pouzet, Paul Caspi, Pascal Couq, and Grégoire Hamon. Lucid Synchrone v2.0 – tutorial and reference manual. `http://www-spi.lip6.fr/lucid-synchrone/lucid_synchrone_2.0_manual.ps`, April 2001.

[66] John Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[67] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

[68] Meurig Sage. FranTk: A declarative GUI system for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.

[69] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[70] Alvy Ray Smith. Image compositing fundamentals. Technical Report Technical Memo #4, Microsoft Research, July 1995.

[71] Sun Microsystems. *Java Core Reflection API and Specification*. Sun Microsystems, 1997.

[72] Sun Microsystems. *The Swing Java Connection*. Sun Microsystems, 1997.

[73] Ivan E. Sutherland. Sketchpad a man-machine graphical communication system. In *on Twenty-five years of electronic design automation*, pages 507–524. ACM Press, 1988.

[74] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *EHCI*, pages 120–150, 1995.

[75] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, Reading, MA, 1998.

[76] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[77] Zhanyong Wan. Re: more on frp. Electronic Mail Message to Functional Reactive Programming Mailing List (frp@cs.yale.edu). `http://http://netra.cs.yale.edu/mailman/listinfo/frp`.

[78] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[79] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming*, 2001.