Extensible and Robust Functional Reactive Programming

Iván Pérez Domínguez, MSc

Thesis submitted to The University of Nottingham

for the degree of Doctor of Philosophy, February 2018

Abstract

Programming GUI and multimedia in functional languages has been a long-term challenge, and no solution convinces the community at large. Purely functional GUI and multimedia toolkits enable abstract thinking, but have enormous maintenance costs.

General solutions like Functional Reactive Programming present a number of limitations. FRP has traditionally resisted efficient implementation, and existing libraries sacrifice determinism and abstraction in the name of performance. FRP also enforces structural constraints that facilitate reasoning, but at the cost of modularity and separation of concerns.

This work addresses those limitations with the introduction of Monadic Stream Functions, an extension to FRP parameterised over a monad. I demonstrate that, in spite of being simpler than other FRP proposals, Monadic Stream Functions subsume and exceed other FRP implementations.

Unlike other proposals, Monadic Stream Functions maintain purity at the type level, which is crucial for testing and debugging. I demonstrate this advantage by introducing FRP testing facilities based on temporal logics, together with debugging tools specific for FRP.

I present two uses cases for Monadic Stream Functions: First, I show how the new constructs improved the design of game features and non-trivial games. Second, I present Reactive Values and Relations, an abstraction for model-view coordination in GUI programs based on a relational language, built on top of Monadic Stream Functions. Comprehensive examples are used to illustrate the benefits of this proposal in terms of clarity, modularity, feature coverage, and its low maintenance costs. The testing facilities mentioned before are used to encode and statically check desired interaction properties.

A Susana e Jose, polo seu constante apoio, e por todo o que fan polos demais.

Acknowledgements

This thesis would have never been completed without the help and support of many colleagues and friends.

My supervisor, Dr. Henrik Nilsson, was a constant source of motivation. His advice, guidance and constructive criticism have helped me turn ideas and intuitions into robust theories and implementations. As a supervisor, his door has always been open to me, and he has always gone to great lengths to help me make the best of this period of my career, creating opportunities for me to present my work in public, to participate in conferences, and to collaborate with others. Thank you. I could not have been luckier.

I would like to thank Professor Graham Hutton, my second supervisor. His advice has always been concise, simple and accurate. He has always been able to give the simplest guideline that would make the biggest improvement to my work, encouraging me to pay attention to detail, to always do my best, and to try, even when I could fail. He has been patient, willing to read drafts of my work, to take the time to explain things in detail, and to help me clarify my thinking and improve my writing.

I thank Thorsten Altenkirch and Koen Claessen for agreeing to be my examiners on such short notice, for taking the time to discuss my work with me, and for their useful criticism. I also thank Venanzio Capretta for agreeing to be my internal examiner over the course of my studies.

I thank all other members of the Functional Programming Laboratory, for all the fun times, and for a number of technical discussions. For the many reviews of early drafts of my papers and proposals, I thank Paolo Capriotti, Ambrus Kaposi, Thorsten Altenkirch, Venanzio Capretta, Jan Bracker, Gabe Dijkstra, Jenny Hackett, Jakob von Raumer, Jonathan Thaler, Neil Sculthorpe, David McGillicuddy, Robert Mitchelmore, Florent Balestrier and Nuo Li. I thank the University of Nottingham for the funding that allowed me to complete this work, and ACM Sigplan for funding to attend multiple conferences.

This work has benefited from years of discussions with Manuel Bärenz. Thank you for your time, for your patience, for your work, for your reviews, for all the chocolate that you conveniently carry with you at all times and are always willing to share, and for introducing me to climbing.

For many technical discussions, for reviewing different versions of my papers and thesis, and for general feedback, I thank Arsen Kostenko, Héctor Fuertes, Jonathan Thaler, Simon Thompson, Lennart Augustsson, Jeremy Gibbons, Neil Krishnaswami, John Hughes, Koen Claessen, Conal Elliott, Simon Peyton Jones, Atze van der Ploeg, Baltasar Trancón y Widemann, Harald Steinlechner, Luite Stegeman, Philip Hölzenspies, Csaba Hruska, Niki Vazou, Derek Wright, Nicolas Rolland, Ángel Herranz-Nieva, Emilio Gallego Arias, the people who attended my talks, and anonymous paper reviewers. I thank Paul Hudak and Conal Elliott for inspiring me to explore interactive multimedia using a functional language.

A number of friends have personally helped me over this period, sometimes unknowingly. Miriam de la Campa, thank you for your continuous support, for believing in me more than I did, and for always being there when I needed you most. Anna Dodson, I thank you for always finding time in your schedule and a room in your home for me, for always having something nice to say, and for always managing to cheer me up. Natalia, thank you for rescuing me, for loving me, and for encouraging me to work on these ideas just for fun. Arsen, thank you for telling me that my work was worth it and interesting, even when I did not think so myself; thank you for always being there. Dani, thank your help and for the opportunity to work with you that led to a position that led to a position that took me here. Heather, thank you for listening and for your advice. Roza, thank you for the fun times and for always being there. Amanda, thank you, for every minute that you shared with me. Nessa, thank you for your support and your help. Estela, thank you for being who you are, for always making me smile, even when you are far. I love you.

Berto, thank you for being the best brother anyone could possibly have. Thank you for always being there, even when you do not really have the time. Ma, thank you for telling me to work hard, and for the many, many, many times that you helped me, even when I did not deserve it. Pa, I wish you could have seen this. Thank you for so many valuable lesons that still resonate. I also thank my uncles and aunts, my cousins, my grandparents, and all my friends.

Last, but not least, I would like to thank the many teachers and supervisors I have had over the years, including Ángel Herranz-Nieva, Oscar Corcho, Jan Kuper, Nieves, Pepe Parraños, Josefina, and Miguel Bernal. Thank you, to the elementary school Seis do Nadal, for including Programming in Logo as part of their curriculum.

Contents

List of Figures

Chapter 1

Introduction

Functional Programming promises highly declarative code, efficient and parallelisable execution, modularity, and reusability. These promises have already materialised in multiple fields of application. Yet when it comes to programming with effects, sticking to a purely functional style, and obtaining the benefits of doing so, is arguably much harder. This holds true, in particular, in interactive software.

Programming interactive software has been explored by the functional programming community, and proposals on the best way to do it abound. However, the lack of multiple examples of large real-world applications has limited the understanding of the impact of those solutions at a larger scale. In this thesis I explore the area of interactive application programming in functional languages, focusing, specifically, on graphical user interfaces and games.

1.1 Context

Programming interactive software is known to be hard [1], irrespective of the programming paradigm. Ideally, one would like to write interactive applications that are well-structured, easy to reason about, visually appealing and computationally efficient. There is often a trade-off to be made between these objectives, and both imperative and functional solutions find difficulties meeting these goals.

1.1.1 Limitations of Imperative GUI Programming

Interactive applications, and those with Graphical User Interfaces (GUIs), are often large and intrinsically complex, with many inter-related elements [1]. This complexity lies both in the application domain and in common usability features, such as being able to cancel long operations

(which requires *concurrency*, and undoing partially applied transactions) and undo-redo (with potentially unbounded *memory requirements*). Users expect these features, but they are orthogonal to the main problem addressed by the program.

Due to this complexity, both structuring GUI application code well and reasoning about it are hard [2, 3, 4, 5, 6, 7, 8]. These tasks are made more difficult by the fact that GUI applications are inherently stateful, as they interact with the outside world and usually react differently to the same user action depending on previous history. In imperative programming, this state is implicit, and functions can have side effects and affect the internal application state in any way.

Reasoning about the application requires that programmers keep mental track of how the state changes as the execution progresses, for which a deep understanding of the potential side effects of each function, GUI-related or not, is necessary. GUI toolkits—libraries used to present interactive visual elements, like windows and buttons—have a logic of their own, often loosely defined and lacking precise operational semantics.

The standard GUI toolkits used to implement applications also have structural constraints that make reasoning about programs harder. Reacting to user input is done by associating certain computations to specific widgets and events, which results in programs broken up in multiple inter-related parts, an event-oriented programming style [9] that inverts control [10, p. 36-37] and is hard to reason about [11].

To address modularity and structural concerns, multiple GUI application architectures have been suggested. The widely used Model-View-Controller [12] splits the application into a *model* (problem abstraction), a *view* (user interface) and a *controller* (keeps model and view in synchrony and executes effectful operations). For efficiency and visualisation reasons[1], the controller "tries" to update only minimal parts of the view, for which it needs to know which specific parts of the model change with each operation. This results in poor separation of concerns, an error-prone style and codebases that grow quadratically with every new feature or UI element[2].

To address the problems of MVC and similar architectural patterns, languages have incorporated mechanisms to monitor changes in data structures [13, p. 326], giving controllers the option to define separate event handlers to update the view when the model changes. This results in less code duplication and smaller codebases, but it requires manually installing and handling even

[1] Refreshing the entire screen usually results in *flickers*.

[2] GUI applications have synchronisation invariants between the model and the view. Several widgets may reflect the same part of the internal model. Delegating model-to-view updates to view-to-model event handlers means that each widget's event handler needs to propagate changes also in model-to-view direction to keep *all other widgets in synchrony*, duplicating code. If we add one more widget synchronised with the same part of the model, all other widgets need to keep that new widget in synchrony, and the event handlers of the new widget need to do the same for all others widgets, resulting in quadratic growth.

more callbacks[3].

Reactive Programming [14, 15] takes this approach one step further by ubiquitously using objects with support to notify other objects when they change. Some implementations promote a more declarative programming style by eliminating the need to install callbacks manually, and by allowing users to apply functions that operate on "plain" values onto reactive, changing values.

1.1.2 Limitations of Functional GUI Programming

As one reads the above, the potential benefits of Functional Programming (FP) become apparent. Abstracting features in orthogonal solutions that can be composed and combined freely seems like the kind of advantage that FP might provide. Referential transparency enables equational reasoning and subsequent program transformation, while strong, static type systems can provide substantial compile-time guarantees. Efficient parallelisation with no worries about deadlocks or rolling back unfinished transactions is part of the promise of this paradigm [16]. There are dozens of functional libraries to implement GUIs, ranging from low-level bindings to high-level functional GUI abstractions.

Low-level functional bindings to existing GUI toolkits provide competitive efficiency and a visually attractive appearance, but impose an imperative programming style. Functional Languages make state explicit and require that we pass it around as we modify it, always in the right order. Abstractions like monads [17, 18, 19] simplify such code by making state-passing implicit and guaranteeing its linearity. This, however, results in an imperative programming style which is difficult to reason about [20], to parallelise [21, p. 309], to optimise (both manually and by the compiler), and suffers from the kinds of modularity concerns described earlier.

Alternatively, high-level compositional widget toolkits, like Fudgets [22] and Gadgets[23], bring ease of reasoning and static guarantees, but have higher maintenance costs, impose an unnatural visual appearance, and may not always scale well in terms of efficiency and/or code modularity [24].

GUI toolkits are notoriously large[4] and, despite the similarities, each operating system promotes its own library. A toolkit that covers most, if not all, of users' needs would encompass a very large codebase. Furthermore, one would need to implement several such toolkits, one per platform, to obtain a natural look-and-feel in cross-platform applications, exacerbating their cost. Additionally, the interactive tools used by domain experts to define static GUIs visually would not be available for these re-implementations[5].

[3] In languages like Java, up until version 7, callbacks require the definition of an interface and an anonymous Object, a notoriously verbose hindrance.

[4] As an example, the standard GTK+ bindings for Haskell currently export 6185 symbols.

[5] Compare, for instance, the fully visual GUI design environment for Android (`http://developer.android.com/tools/help/adt.html`) integrated in Eclipse with the code-based strategy

An alternative to address the concerns raised so far would be to define a functional abstraction for reactive processes that facilitates reasoning but can also be used in combination with existing low-level GUI bindings. Functional Reactive Programming [25, 26, 27] is one such abstraction. FRP addresses interactive application concerns by defining reactive, dynamic programs as collections of inter-dependent time-varying values or *signals*. FRP focuses on describing time-varying values denotationally by specifying data dependencies, and leverages the responsibility of linearising state.

While FRP addresses most of the issues of the proposals presented before, it also has a number of limitations. The programming style that FRP imposes, based on specifying data dependencies and relations among signals upon signal creation, leads to initialisation problems [28] and modularity concerns ([29]) in the presence of cyclic dependencies, which are extremely common in GUI programs [30]. To address some of these issues, as well as to meet the requirements of performance-demanding applications and games, some FRP implementations use I/O, explicitly or implicitly. However, in doing so, they may also sacrifice abstraction and make FRP programs difficult to test in a deterministic and systematic way, since the conditions that lead to a bug when I/O is allowed may be hard or impossible to reproduce.

Pure vs Impure FRP The distinction between FRP implementations that use any form of I/O and those that do not is central in this dissertation. To simplify the discussion, we will use the term *pure* to refer to FRP variants and implementations that separate the I/O computations from the FRP logic completely, and we will use the term *impure* for those that allow I/O within FRP expressions, explicitly or implicitly. The intuition behind this choice of terminology is that, in pure FRP, signals and the functions that transform them can be seen as pure functions themselves.

1.2 Contributions

This thesis addresses the aforementioned limitations by proposing a more general abstraction that, on the one hand, allows for limited side effects, and, on the other hand, maintains the static guarantees needed for reliable testing and debugging.

This work contributes to the field of Functional Reactive Programming in the following ways:

- The thesis proposes Monadic Stream Functions (MSFs) as a simpler, more general abstraction for reactive programming. Monadic Stream Functions capture the essence of reactive systems, that is, transforming varying input values into varying output values, in a monadic context.

used for Dynamic GUIs (`http://developer.android.com/training/basics/fragments/index.html`), with only small fragments being designed visually.

This is based on a simple coinductive type, together with operations that allow us to preserve desired properties of these transformations.

- The thesis demonstrates that this minimal framework of Monadic Stream Functions is powerful enough to capture desired features in reactive programming like termination, switching, parallelism and sinks, all of which emerge naturally by changing the monad in which one is operating. Effects can be composed at a stream level by means of monad transformer stacks, which demonstrates the power and versatility of this abstraction.

- The thesis also demonstrates that Monadic Stream Functions subsume and extend what is currently possible with Reactive Programming and Functional Reactive Programming. Using the right combination of monads, one can implement reactive systems that vary over an explicit notion of time. This allows us to define both Classic and Arrowized FRP on top of Monadic Stream Functions. Through the use of monad stacks, the power of these FRP systems can be extended, enabling novel features useful in real games.

- The thesis shows that MSFs have strong mathematical properties that enable equational reasoning, and that further transformations are possible for monads with certain properties.

- The thesis demonstrates that traditional FRP has limitations that affect modularity and separation of concerns, and proposes Reactive Values (RVs) and Reactive Relations (RRs) as an abstraction that is better suited for discrete, event-oriented GUI programming. The thesis shows that these concepts, much closer to the discrete Reactive Programming paradigm, can also be implemented on top of Monadic Stream Functions, and their use improves the architecture of GUI applications.

- The thesis shows that some forms of FRP implementable with MSFs have desirable properties that facilitate testing and debugging. The thesis uses two logics to operate on reactive systems in different settings. A forward-time temporal logic over finite traces is used to test FRP systems in combination with existing testing tools like QuickCheck. A past-time variant of the same logic is used to insert monitors inside FRP programs and detect property violations during execution.

- The thesis presents an API and new debugging tools to test and monitor properties, and to connect to existing testing frameworks. The use of these tools is presented both with comprehensive examples and with an evaluation of several non-trivial bugs in real interactive applications.

1.3 Thesis Overview

The remainder of this dissertation is structured as follows:

- Chapters 2 and 3 present the background and identify the specific problems that we shall address.
- Chapter 4 introduces a new formalism of Monadic Stream Functions that enables simpler and more versatile reactive abstractions. This chapter is based on joint work with Manuel Bärenz and Henrik Nilsson [31]. Section 4.3.2 is mainly due to Manuel Bärenz, with input from Paolo Capriotti.
- Chapter 5 demonstrates a use case of Monadic Stream Functions that makes them more versatile than traditional Classic and Arrowized FRP. The definition of FRP abstractions in terms of MSFs is based on joint work with Manuel Bärenz and Henrik Nilsson [31]. The introduction of new features not present in existing RP and FRP frameworks is the author's own work [32].
- Chapter 6 presents the use of Monadic Stream Functions in real-world applications and games. This is the author's own work, based on previously published work, partly with Manuel Bärenz and Henrik Nilsson [31], as well as individually [32].
- Chapter 7 presents an abstraction more suitable for event-based GUI programming, which can be expressed in terms of Monadic Stream Functions. This is the author's own contribution, based on work published jointly with Henrik Nilsson [29].
- Chapter 8 introduces a novel testing approach for Arrowized FRP. This is the author's original work, based on work published jointly with Henrik Nilsson [33].
- Chapter 9 presents how this approach can be used to debug FRP programs. This is the author's original work, based on work published jointly with Henrik Nilsson [33].
- Chapter 10 presents how the testing and debugging facilities introduced in previous chapters were used to detect non-trivial bugs in real games. This is the author's original work, based on work published jointly with Henrik Nilsson [33].
- Chapter 11 presents related work. This is partly based on previous reviews published jointly with Henrik Nilsson and Manuel Bärenz [31, 33, 32, 29].
- Chapter 12 presents future lines of work.

1.4 Conventions

This thesis uses the programming language Haskell for all code examples and implementations, unless otherwise stated. We use the font *sample* when referring to programming constructs, and the standard text font when referring to the underlying concept that they represent. For example, I/O denotes input/output, whereas *IO* denotes the *IO* monad in Haskell.

Due to the general convention of using American spelling in programming, this dissertation uses different spellings when referring a general concept and when referring to the corresponding Haskell implementation. For example, in Chapter 7 we describe some entities as writeable, but use the type class *Writable* to capture that notion in the implementation.

Part I

Background

Chapter 2

Functors, Monads and Arrows

Throughout the rest of the thesis, we assume basic knowledge of Haskell. Nevertheless, in the interest of making this thesis self-contained, and to facilitate understanding the material in future chapters, this chapter presents several concepts that are key to understand the rest of the thesis.

This chapter presents four notions that are omnipresent in Haskell programming: *functors*, *applicative functors*, *monads*, and *arrows*. The concepts presented in this chapter are deeply rooted in Category Theory, a field that is not explored in this thesis, but the view presented here is that of a Haskell programmer. For a more complete reference on the Haskell constructs presented here, the reader can consult an introductory Haskell book [34, 35]. For an introduction to *Arrows*, [36] and [37] are recommended. For an understanding of the meaning of these terms in the field of Category Theory, the reader is directed to [38, 39].

While the explanations in this chapter are my own, the ideas and concepts explained are not. An effort has been made to avoid descriptions that are too similar to other texts and references, but certain influence and similitudes are unavoidable. The images for the diagrams for the *Arrow* combinators in Figure 2.1 are from [40], and reproduced with permission. Other references will be provided along with the text. Any errors in this chapter should be attributed only to myself.

2.1 Functors

Let us begin by presenting a function used to transform lists in Haskell. Lists are normally written as $[1, 2]$, $[5, 5, 5, 5]$, $[]$, and $[1..]$ (the latter meaning the infinite list of ones), although they can also be written using the data constructors explicitly, as in $(1 : 2 : [])$.

Haskell provides numerous operations to work with lists, one of which is the function *map*, which applies a transformation to every element of a list:

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to [a] \to [b] \\
&\mathit{map}\ f\ [\,] \qquad = [\,] \\
&\mathit{map}\ f\ (x : xs) = f\ x : \mathit{map}\ f\ xs
\end{aligned}
$$

For example, if we have the list $[1, 2]$, then $\mathit{map}\ (+1)\ [1, 2]$ is equal to $[2, 3]$.

The idea of traversing a (polymorphic) data structure and applying a transformation to every element is applicable not just to lists, but to other types as well. For example, for a type that represents binary trees:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

we can define an analogous mapping function that applies a transformation to every node:

$$
\begin{aligned}
&\mathit{treemap} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b \\
&\mathit{treemap}\ f\ \mathit{Leaf} \qquad\qquad\qquad\quad = \mathit{Leaf} \\
&\mathit{treemap}\ f\ (\mathit{Node}\ x\ \mathit{tLeft}\ \mathit{tRight}) = \mathit{Node}\ (f\ x)\ (\mathit{treemap}\ f\ \mathit{tLeft})\ (\mathit{treemap}\ f\ \mathit{tRight})
\end{aligned}
$$

And so, if we have a tree with strings of characters, we can create a tree with the same shape that contains their length, as follows:

$$
\begin{aligned}
&\mathit{treemap}\ \mathit{length}\ (\mathit{Node}\ \texttt{"Haskell"}\ (\mathit{Node}\ \texttt{"types"}\ \mathit{Leaf}\ \mathit{Leaf})\ \mathit{Leaf}) \equiv \\
&\quad (\mathit{Node}\ 7\ (\mathit{Node}\ 5\ \mathit{Leaf}\ \mathit{Leaf})\ \mathit{Leaf})
\end{aligned}
$$

This idea can be generalised to work for other polymorphic types, such as *Maybe*, *Either*, Sets, arrays, etc. In the context of Haskell, the type class *Functor* captures that idea, and describes types that can be *mapped over*:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\
&\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

*Functor*s must preserve identity and composition, so instances must satisfy the following laws:

$$
\begin{aligned}
&\mathit{fmap}\ \mathit{id} \qquad \equiv \mathit{id} \\
&\mathit{fmap}\ (f \circ g) \equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g
\end{aligned}
$$

Many Haskell types are *Functor*s. Lists and trees are instances of *Functor*, with the definitions for *fmap* that we saw above with slightly different names. So are the types $\mathit{Either}\ e$ (for a given e), and IO. The instance for *Maybe* is:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\
&\quad \mathit{fmap}\ f\ \mathit{Nothing} = \mathit{Nothing} \\
&\quad \mathit{fmap}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x)
\end{aligned}
$$

As an example, if we have a function from *String* to *Int*, we can apply it to a value of type *Maybe String* to transform the *String* inside, if any:

```
maybeLen :: Maybe String → Maybe Int
maybeLen maybeString = fmap length maybeString
```

As we can see in the definition of *fmap* for *Maybe*, if *maybeString* is *Nothing*, then this will be *Nothing*. If *maybeString* is *Just x* for some *String x*, then the result will be *Just* (*length x*).

The functor laws allow us to perform some transformations to our programs. For example, the following three definitions are equivalent:

```
maybeLenUpTo6  :: Maybe String → Maybe Int
maybeLenUpTo6  maybeString = fmap (min 6) (fmap length maybeString)

maybeLenUpTo6' :: Maybe String → Maybe Int
maybeLenUpTo6' maybeString = fmap (min 6 ∘ length) maybeString

maybeLenUpTo6'' :: Maybe String → Maybe Int
maybeLenUpTo6'' = fmap (min 6 ∘ length)
```

Sometimes we may find it more convenient to use the infix *fmap* operator (<$>), and write:

```
maybeLenUpTo6''' :: Maybe String → Maybe Int
maybeLenUpTo6''' maybeString = (min 6 ∘ length) <$> maybeString
```

This notation is similar to what we obtain if we use the infix function application operator for normal functions, ($) :: $(a \to b) \to a \to b$.

2.2 Applicative Functors

The function *fmap* from the *Functor* class is useful when the function we apply is unary. However, if we apply a binary function, we "trap" a function inside the functor, something we can inspect asking GHCi for the type of a value:

```
> :t fmap (++) (Just "Hello ")
fmap (++) (Just "Hello ") :: Maybe (String → String)
```

In order to apply this function to a second value of type *Maybe String*, we would need an auxiliary function with signature *Maybe* (*String* → *String*) → *Maybe String* → *Maybe String* that takes the second argument and passes it to the function inside. We can build such a function as follows

```
app :: Maybe (String → String) → Maybe String → Maybe String
app (Just f) (Just v) = Just (f v)
app _        _        = Nothing

> (fmap (++) (Just "Hello ")) `app` (Just "Haskeller")
Just "Hello Haskeller"
```

We use infix notation because it will be more convenient in the following. We could then use the same mechanism to work with a function on three arguments, except that now it becomes harder to work with the parenthesis:

```
> ((fmap (λx y z → x ++ y ++ z) (Just "a")) `app` (Just "b")) `app` (Just "c")
Just "abc"

> ((fmap (λx y z → x ++ y ++ z) (Just "a")) `app` Nothing) `app` (Just "c")
Nothing
```

This useful pattern is captured by the class of applicative functors [41], also known as *Applicative*s, defined as follows:

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

The function (<*>) is the generic version of *app*, defined above. The function *pure* simply "puts" a value inside the *Applicative*, and it is useful to lift functions to an applicative context, or to pass values to functions in an *Applicative* functor.

Like functors, applicative functors have associated laws they must fulfill:

```
pure id  <*> x                = x
pure (∘) <*> f <*> g <*> x = f <*> (g <*> x)
pure f   <*> pure x           = pure (f x)
f        <*> pure y           = pure ($y) <*> f
```

Many *Functor*s are also *Applicative*s. For example:

```
instance Applicative Maybe where
  pure x = Just x

  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

This instance says[1] that, if we have an unapplied function inside a *Just*, and pass it an argument inside a *Just*, then we can collapse the two by applying the value to the function and putting the

[1] For simplicity, we abuse terminology and say that a value is "inside a constructor" if it is an argument to that data constructor. For example, in the value *Just* 7 of type *Maybe Int*, the value 7 is "inside a *Just*".

result inside a *Just*. If either or both the function or the argument are *Nothing*, then the result is also *Nothing*.

Now we can apply functions with two arguments to *Maybe* values:

```
saluteReader :: Maybe String → Maybe String
saluteReader maybeReader = pure (++) <*> (Just "Hello, ") <*> maybeReader
```

A useful result that relates *Functor*s and *Applicative*s is that *fmap f x* = *pure f* <*> *x*. Using the infix *fmap* operator (<$>), we simplify the last construction and write:

```
saluteReader :: Maybe String → Maybe String
saluteReader maybeReader = (++) <$> (Just "Hello, ") <*> maybeReader
```

In this case, one could also say:

```
saluteReader :: Maybe String → Maybe String
saluteReader maybeReader = (++) <$> pure "Hello, " <*> maybeReader
```

Finally, we can write the complicated function based on *app* in the previous page as:

```
(λx y z → x ++ y ++ z) <$> pure "a" <*> pure "b" <*> pure "c"
```

Just like in the case of *Functor*s, many useful monads are also *Applicative*s, including *Either c* (for a given type *c*), lists, and the type of input/output computations, *IO*.

2.3 Monads

Monads in Haskell capture the notion of computation in a context [19], and are defined by the following type class:

```
class Applicative m ⇒ Monad m where
  return :: a → f a
  (>>=) :: m a → (a → m b) → m b
```

A polymorphic type in Haskell is a monad if it implements the *Monad* type class. The function *return* applied to a value is a computation that returns that value in the context, and the function bind, written (>>=), combines a computation with a function that consumes a value and produces a second computation.

Common examples of simple monads include, for instance, the *Maybe* monad:

```
instance Monad Maybe where
  return x = Just x
```

$$Nothing \gg\!= \_ = Nothing$$
$$(Just\ x) \gg\!= f = f\ x$$

We can use this interface to sequence several computations that may *fail* (returning *Nothing*), for instance, if we have to check a username and a password against a list of known credentials in a map, we can use the following function that handles failure (e.g., user not found, password not available, argument not provided), implicitly:

$$checkUsername :: [(String, String)] \to Maybe\ String \to Maybe\ String \to Maybe\ String$$
$$checkUsername\ userTable\ maybeUsername\ maybePassword =$$
$$\quad maybeUsername \gg\!= \lambda username \to$$
$$\quad maybePassword \gg\!= \lambda password \quad \to$$
$$\quad lookup\ username\ userTable \gg\!= \lambda realPassword \to$$
$$\quad \textbf{if}\ (password \equiv realPassword)\ \textbf{then}\ return\ \texttt{"Authorised"}\ \textbf{else}\ Nothing$$

Like *Applicative*s and *Functor*s, *Monad*s are also expected to fulfill some laws. In particular, *return* cannot have any side effects, and needs to act as an identity for bind on both sides:

$$return\ a \gg\!= f \equiv f\ a$$
$$m \gg\!= return \quad \equiv m$$

Also, the bind operator must be associative:

$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)$$

There are occasions in which we do not really need to do anything with the value inside the *Applicative*, but we do need to perform the "side effects". In the case of *Maybe*, the side effect is that the whole computation fails (is *Nothing*) if any computation in the sequence fails. In the case of the *IO* monad, which handles input/output, a side effect may be a change in the real world (a value shown on the screen, or a package sent through the network interface). A useful operation that captures performing these side effects is ($\gg$):

$$(\gg) :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$$
$$m1 \gg m2 = m1 \gg\!= (\backslash\_ \to m2)$$

At first, it might seem that the value inside *m1* is discarded, and so this might be the same as only *m2*. However, the *side effect* of the computation *m1* still persists. For example, the expression $Nothing \gg Just\ 7$ is equal to *Nothing*, whereas if we discard the computation to the left of ($\gg$), in this case *Nothing*, we obtain a different value: *Just* 7. The expression $getLine \gg print\ \texttt{"Hi"}$ waits for user input and then prints a line, whereas the expression $print\ \texttt{"Hi"}$ does not wait for user input. You can think of ($\gg$) as combining the side effects of both computations, in order.

Writing long chains of monadic actions using this syntax can be daunting. Haskell supports writing these sequentially and vertically using what is known as *do notation*. For example, given the following functions:

$$putStrLn :: String \to IO\ ()$$
$$getLine\ \ :: IO\ String$$

and knowing that IO is also a monad, the monadic expression:

$$expr = putStrLn\ \texttt{"Write your name?"} \gg getLine \gg\!= \lambda l \to putStrLn\ (\texttt{"Hi, "} +\!\!+ l)$$

which can be indented as follows:

$$\begin{aligned} expr = {} & putStrLn\ \texttt{"Write your name?"} \gg \\ & getLine \gg\!= \lambda l \to \\ & putStrLn\ (\texttt{"Hi, "} +\!\!+ l) \end{aligned}$$

can also be written, in *do notation*, as:

$$\begin{aligned} expr = {} & \mathbf{do}\ putStrLn\ \texttt{"Write your name?"} \\ & \quad l \leftarrow getLine \\ & \quad putStrLn\ (\texttt{"Hi, "} +\!\!+ l) \end{aligned}$$

Many standard operations can be lifted to operate with values that are the result of monadic computations. For example, the standard function *map*[2]:

$$map :: (a \to b) \to [a] \to [b]$$

has a monadic counterpart:

$$mapM :: (a \to m\ b) \to [a] \to m\ [b]$$

If we simply apply *map* to the auxiliary monadic function, the result will have type $[m\ b]$. When we only care about the side effects (common when we operate in the IO monad), we can also use the alternative construct:

$$mapM\_ :: (a \to m\ b) \to [a] \to m\ ()$$
$$mapM\_\ f\ xs = mapM\ f\ xs \gg return\ ()$$

For example, if we have a list of names, we can print them all, each in one line, with:

$$printNames :: [String] \to IO\ ()$$
$$printNames\ xs = mapM\_\ putStrLn\ xs$$

[2]This function is the same as *fmap* from the *Functor* instance for [] (lists). Its current type in Haskell is much more general, and expressed in terms of other classes like *Foldable*. Here, we give only the simplest definition.

Commutative Monads Monads can have additional properties that allow us to transform the code in other ways, and to optimise it. For example, in general, we cannot expect that computations be commutative. This is clear in the expression given in the previous page, in which we do not obtain the same result if we change it to:

$$
\begin{aligned}
&expr = \textbf{do}\ l \leftarrow getLine \\
&\qquad\quad putStrLn\ (\texttt{"Hi, "} +\!\!+\ l) \\
&\qquad\quad putStrLn\ \texttt{"Write your name?"}
\end{aligned}
$$

We say that a monad m is commutative if, for any two computations $m1 :: m\ a$ and $m2 :: m\ b$, and a function $f :: a \to b \to m\ c$, the following two expressions are equivalent:

$$m1 \gg\!= \lambda x1 \to m2 \gg\!= \lambda x2 \to f\ x1\ x2 \equiv m2 \gg\!= \lambda x2 \to m1 \gg\!= \lambda x1 \to f\ x1\ x2$$

Or, in do notation:

$$
\begin{aligned}
&\textbf{do}\ x1 \leftarrow m1 \\
&\quad x2 \leftarrow m2 \\
&\quad f\ x1\ x2
\end{aligned}
\qquad\qquad
\begin{aligned}
&\textbf{do}\ x2 \leftarrow m2 \\
&\quad x1 \leftarrow m1 \\
&\quad f\ x1\ x2
\end{aligned}
$$

That is, if inverting the order in which two computations appear in a sequence of monadic computations does not alter the result. Monads like *IO* are not commutative, as we saw before, but monads like *Maybe* are.

There is a whole family of type classes that empower monads with additional operations and laws, like *MonadFix* (monads with fixpoints), and *MonadPlus*.

2.4 Arrows

Arrows [36, 42] are an abstraction for computations, and a generalisation of monads. Just like monads capture the notion of a computation that creates values of a given type, arrows capture the notion of computations that take values of a given type, and transform them into values of a possibly different type.

For example, we might want to define the type of causal synchronous stream transformations in Haskell, that is, functions that transform infinite streams. We could opt to define streams as follows:

$$\textbf{type}\ Stream\ a = (a, Stream\ a)$$

And stream processors as follows:

$$\textbf{type}\ StreamProcessor\ a\ b = Stream\ a \to Stream\ b$$

However, this construction is too permissive. It allows us to create transformations that are not synchronous, producing more outputs than inputs, such as:

$$dupSP :: StreamProcessor\ a\ a$$
$$dupSP\ (a, sp) = (a, (a, (dupSP\ sp)))$$

We could also define non-causal stream processors, in which the n-th output sample depends on the $n + 1$-th input sample:

$$skipSP :: StreamProcessor\ Int\ Int$$
$$skipSP\ (\_, (b, sp)) = (b * 2, skipSP\ sp)$$

While these may be operationally sound, they do not represent the notion we wanted to capture of synchronous stream processors. What we want instead, is to provide a restricted API that maintains synchronicity and causality.

As an example that will be useful in later chapters, let us present a type that captures a synchronous function that processes an infinite stream of values:

$$\textbf{newtype}\ SP\ a\ b = SP\ \{ unSP :: a \to (b, SP\ a\ b) \}$$

A stream processor takes one sample, processes it, and produces an output sample and a continuation. This continuation is meant to be used to process the next input sample, and so on.

We can, for example, define a function that applies the same transformation to every input sample:

$$liftSP :: (a \to b) \to SP\ a\ b$$
$$liftSP\ f = SP\ \$\ \lambda v \to (f\ v, liftSP\ f)$$

or one that delays the input stream by one sample, with a given default value for the first output:

$$delay :: a \to SP\ a\ a$$
$$delay\ a0 = SP\ \$\ \lambda a1 \to (a0, delay\ a1)$$

*SP*s are not plain functions in the traditional sense. For example, if we have an SP that adds one unit to the input value:

$$spAdd1 :: SP\ Int\ Int$$
$$spAdd1 = liftSP\ (+1)$$

we cannot use the $(\circ)$ function composition operator and define:

$$spAdd2 :: SP\ Int\ Int$$
$$spAdd2 = spAdd1 \circ spAdd1$$

A notion of composition does exist for signal processors, though, and we can define it as follows:

$$
\begin{aligned}
&compSP :: SP\ a\ b \to SP\ b\ c \to SP\ a\ c \\
&compSP\ sp1\ sp2 = SP\ \$\ \lambda v0 \to \mathbf{let}\ (v1, sp1') = (unSP\ sp1)\ v0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (v2, sp2') = (unSP\ sp2)\ v1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (v2, compSP\ sp1'\ sp2')
\end{aligned}
$$

These definitions preserve causality and synchronicity, and so we know that the following *SP* is causal and synchronous:

$$
\begin{aligned}
&spAdd10Mult2 :: SP\ Int\ Int \\
&spAdd10Mult2 = compSP\ (liftSP\ (+10))\ (liftSP\ (*2))
\end{aligned}
$$

Arrows capture a notion of transformation that is slightly more general than that of a plain function. An instance of the *Arrow* type class requires only three definitions[3]:

$$
\begin{aligned}
&\mathbf{class}\ Arrow\ a\ \mathbf{where} \\
&\quad arr\ \ :: (b \to c) \to a\ b\ c \\
&\quad first\ :: (b \to c) \to a\ (b, d)\ (c, d) \\
&\quad (\ggg) :: a\ b\ c\ \ \ \to a\ c\ d \to a\ b\ d
\end{aligned}
$$

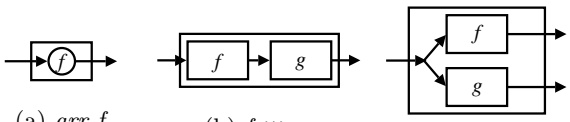
That is, we must be able to *lift* a pure function into an *Arrow*, we must be able to compose transformations with $(\ggg)$, and we must be able to apply an arrow to one element of a pair, leaving the other unchanged.

There are a number of additional operations on arrows that are defined in terms of the ones above. The complete definition, specialised for the case of *SP*, is as follows:

$$
\begin{aligned}
arr\ \ \ \ &:: (a \to b) \to SP\ a\ b \\
(\ggg)\ \ &:: SP\ a\ b\ \ \to SP\ b\ c \to SP\ a\ c \\
first\ \ \ &:: SP\ a\ b\ \ \to SP\ (a, c)\ (b, c) \\
second &:: SP\ b\ c\ \ \to SP\ (a, b)\ (a, c) \\
(***)\ \ &:: SP\ a\ b\ \ \to SP\ c\ d \to SP\ (a, c)\ (b, d) \\
(\&\&\&) &:: SP\ a\ b\ \ \to SP\ a\ c \to SP\ a\ (b, c)
\end{aligned}
$$

A diagramatic representation of some of these combinators, inspired by digital circuits is shown in Figure 2.1.

[3]This follows the description in [36]. In the version in the corresponding module in Haskell as of the time of this writing, only *arr* and *first* are required, but *Arrow*s are required to be instances of *Category*, and hence have to have a notion of identity and composition.



(a) *arr f* (b) $f \ggg g$ (c) $f \mathbin{\&\&\&} g$

Figure 2.1: Basic arrow combinators.

The implementations of *arr* and ($\ggg$) are, respectively, the ones for *liftSP* and *compSP* given previously, and are part of the instance definition. The implementation for *first* is given below:

```
instance Arrow SP where
  ...
  first sp = SP $ λ(x, z) → let (y, sp') = unSP sp x
                            in ((y, z), first sp')
  ...
```

We define *second* analogously. Combinators like ($***$) and (&&&) combine two arrows (in this case, stream processors) to form a new processor that executes both processors in parallel. They can be defined in terms of other combinators:

```
(***) :: Arrow a ⇒ a b c → a d e → a (b, d) (c, e)
sp1 *** sp2 = first sp1 >>> second sp2
```

The combinator (&&&) passes the same input to two parallel arrows:

```
(&&&) :: Arrow a ⇒ a b c → a b d → a b (c, d)
sp1 &&& sp2 = arr (λx → (x, x)) >>> (sp1 *** sp2)
```

Arrows have many associated laws, which follow:

```
arr id                        = id
arr (f >>> g)                 = arr f >>> arr g
first (arr f)                 = arr (first f)
first (f >>> g)               = first f >>> first g
first f >>> arr fst           = arr fst >>> f
first f >>> arr (id *** g)    = arr (id *** g) >>> first f
first (first f) >>> arr assoc = arr assoc >>> first f
  where assoc = ((a, b), c) = (a, (b, c))
```

Just like with *Monads*, many Arrow-related classes exist, like *ArrowLoop*, *ArrowPlus*, and *ArrowChoice*, each of which has its own set of laws and properties. There are also interesting notions not captured by type classes, like that of a commutative arrow, which means that $first\ f \ggg second\ g \equiv second\ g \ggg first\ f$. Arrows will be revisited in later chapters, with multiple examples, and will form a central part of this thesis.

2.4.1 Arrow Notation

To make it more convenient to work with arrows, special notation can be enabled in GHC. The *Arrows* extension lets us write arrow blocks in a notation similar to the monad do notation. For example, we can define the stream processor $spAdd10Mult2 = arr\ (+10) \ggg arr\ (*2)$ also as:

```
spAdd10Mult2 :: SP Int Int
spAdd10Mult2 = proc v0 → do
  v1 ← arr (+10) -≺ v0
  v2 ← arr (*2)  -≺ v1
  returnA -≺ v2
```

We can regard **proc** as the counterpart of the lambda in lambda functions, applied in an Arrowized setting, and *returnA* as the counterpart of *return*, and is defined as $returnA = arr\ id$.

Chapter 3

Functional Programming of Interactive Applications

This chapter presents an overview of the field of functional programming applied to interactive software. Existing solutions for functional interactivity and multimedia range from the imperative and low-level to the abstract and high-level. Some solutions provide functional libraries for multimedia or interactive Graphical User Interfaces (GUIs); others provide an abstraction suitable to be used with *existing* multimedia and GUI frameworks.

There are dozens of proposals for functional multimedia and GUIs, only a subset of which are presented here. While programming interactive applications often requires advanced graphics, we do not explore functional graphics in depth but, rather, focus on *interactivity*. The text aims at brevity, hopefully introducing just enough detail to help understand existing open problems and the motivation to focus on those issues. A more comprehensive exploration of related work is presented in Chapter 11.

In Section 3.1 we explore existing low-level solutions for multimedia and GUI programming in Functional Languages. In Section 3.2 we present high-level functional abstractions specific for GUIs and multimedia. In Section 3.3 we examine Functional Reactive Programming, an abstraction often used to write reactive and interactive applications.

3.1 Low-level Interactivity in Functional Languages

Interactivity and multimedia in functional languages are often performed using libraries written in C/C++ and imported as *effectful computations* via a Foreign Function Interface [43, 44]. Many of those underlying libraries work at a very low-level, such as Hcwiid [45], used to access Wiimote

devices, and OpenGL [46], a 3D graphics library. Functional bindings usually do minimal work, limited to basic type conversions and calls to C code, resulting in low-level APIs that resemble the underlying C interface.

In functional languages, these effectful computations are defined using monads and applicative functors, which are first-class entities [19, 41]. Together with their associated laws, this enables some forms of equational reasoning [47, 48, 49]. However, without higher-level abstractions, large programs that do input/output still tend to *look* imperative [24, p. 11–12]. The sequential execution order of the effectful computations imposes a need to mentally track the implicit program state. This makes reasoning about these programs hard [50].

As an example, regard the following code taken from the world-state updating function of the Haskell game Raincat[1]. It makes use of the aforementioned monadic abstractions; for example, the third line from the bottom uses $mapM\_ :: Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ ()$, introduced in Section 2.3 to apply the same monadic action to a list of elements, and chain only the side effects. In this case, the auxiliary function passed as argument to $mapM\_$ draws a texture on the screen. In spite of having these monadic abstractions available, this function is implemented in a low-level and imperative manner, and there is no clear way to write the function more declaratively.

```
gameDraw :: IORef WorldState → IO ()
gameDraw worldStateRef = do
  worldState ← readIORef worldStateRef
  Nxt.Graphics.begin
  let (cameraX, cameraY) = MainPanel.cameraPos (mainPanel worldState)
  Nxt.Graphics.worldTransform 0.0 0.0
    -- draw background
  Nxt.Graphics.drawTexture 0.0 0.0
    (MainPanel.backgroundTexture (mainPanel worldState))
    (1.0 :: GLdouble)
  Nxt.Graphics.worldTransform cameraX cameraY
    -- draw foreground
  mapM_
    (λ((x, y), tex) → Nxt.Graphics.drawTexture x y tex (1.0 :: GLdouble))
    (levelBackgrounds $ levelData $ curLevel worldState)
  ...
```

GUI Toolkits To abstract over visual elements and provide standardised appearance and behaviour, developers use predefined collections of interactive elements or *widgets*. Examples of

[1] `https://github.com/styx/Raincat/blob/master/src/Game/GameGraphics.hs`

common widgets include text boxes, buttons, windows, lists and menus.

Widgets have associated properties and events. For example, a text box has a property that contains the string of text presented by the text box. The value of this property changes when the user modifies the contents of the text box. Developers can associate operations to changes or actions on widgets, called *event handlers*, executed those changes take place.

Widgets are collected in GUI libraries, which define consistent behaviour and visual appearance for all interactive elements. Examples of such libraries include GTK+ [51], wxWidgets [52] and Qt [53]. While some of those libraries are cross-platform, many operating systems define their own interaction and appearance guidelines and provide a default GUI toolkit.

As an example of functional code interacting with a GUI library, the following snippet builds a simple one-window GTK+ application with a button that has an icon and a label [54], and installs an event handler using *onClicked* (line 8) to print a text when the button is clicked:

```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window ← windowNew
  set window [windowTitle := "Pix", containerBorderWidth := 10]
  button ← buttonNew
  onClicked button (putStrLn "button clicked")   -- Installs event handlers
  box ← labelBox "info.xpm" "cool button"
  containerAdd button box
  containerAdd window button
  widgetShowAll window
  onDestroy window mainQuit
  mainGUI

labelBox :: FilePath → String → IO HBox
labelBox fn txt = do
  box ← hBoxNew False 0
  set box [containerBorderWidth := 2]
  image ← imageNewFromFile fn
  label ← labelNew (Just txt)
  boxPackStart box image PackNatural 3
  boxPackStart box label PackNatural 3
  return box
```

Code that uses these GUI libraries feels as imperative [55, p. 522-527] as the one presented

earlier. Furthermore, event-driven architectures [9], here exemplified by the event handler on line 8, result in inversion of control [10, p. 36-37] and programs fragmented in many small computations, both of which make reasoning about programs hard [11]. Architectural design patterns created to faciliate structuring interactive applications, like the family of Model-View-Controller [12] patterns, move too much logic into one component, the controller, causing quadratic growth of the size of its codebase with new features [24, p. 8] and leading to what is informally known as "callback hells" [56, p. 2].

Reasoning about the behaviour of widgets themselves is also hard. Widget properties are not plain mutable variables [24, p. 13], and the semantics of GUI toolkits is often poorly defined. Furthermore, many of these libraries (including GTK+ and wxWidgets) are *not thread-safe*: GUI functions must be called from the UI thread, controlled by the toolkit itself[2]. Applications that need to control the execution loop or do background work must handle concurrency explicitly, making the code even more complex.

On the bright side, the readability and declarativeness of the resulting functional code is *not substantially worse* than its C/C++ equivalent, and sometimes can be better [57]. Moreover, the performance can be comparable to that obtained using imperative languages [58, 59, 60], making low-level bindings a customary choice for CPU-demanding multimedia.

Some functional GUI implementations try to define an intermediate layer that wraps the underlying effects. Examples of low-level functional GUI libraries include TkGofer [61, 62] and Haggis [63]. TkGofer implements a GUI toolkit and window manager on top of Tcl/Tk. It is powerful and low-level enough to accommodate other more abstract libraries like Fudgets [22]. Haggis is substantially simpler, but makes use of concurrent access to shared memory to implement asynchronous UIs. In both cases, the resulting code is monadic.

3.1.1 Limitations of Low-level Interactivity in Functional Languages

As presented so far, using bindings to existing GUI toolkits one can write functional applications that adhere to a standard look and feel of each platform without a substantial performance loss.

However, this low-level approach limits the benefits expected from using a functional language. Interacting with GUI toolkits imposes an imperative programming style, an event-oriented structure and require dealing with concurrency. Furthermore, GUI toolkits keep part of the application's state in a separate layer, out of the programmer's control, and do not give a clear semantics. Additionally, traditional design patterns to separate the UI and the program logic do not always scale well in terms of modularity and separation of concerns, and can lead to code duplication.

[2] `https://developer.gnome.org/gtk3/stable/gtk3-General.html`

3.2 Purely Functional Interactivity

One possible solution to the problems outlined in the previous section is to define a functional API to model the domain of interest. Such an API does not need to resemble the underlying bindings: an evaluation function can traverse the pure data structures and do the "dirty" work, projecting the changes onto the screen or other devices. Contrary to the imperative case, this kind of code looks declarative, compositional and abstract.

Multiple libraries follow this approach, such as Diagrams [64], Chart [65] and Gloss [66]. The following illustrative example of purely functional Gloss code [67] uses combinators like *Translate* and *Pictures*, to build animations from smaller and simpler ones:

```
main = animate (InWindow "machina" (800, 600) (10, 10)) black frame
frame time = Scale 0.8 0.8 $ Rotate (time * 30) $ mach time 6
mach t 0 = leaf
mach t d =
  Pictures
    [ leaf
    , Translate 0 (-100) $ Scale 0.8 0.8
                         $ Rotate (90 + t * 30)
                         $ mach (t * 1.5) (d - 1)
    , Translate 0 100    $ Scale 0.8 0.8
                         $ Rotate (90 - t * 30)
                         $ mach (t * 1.5) (d - 1)
    ]
...
```

Since interactive widgets must handle user actions, produce a visualisation and notify of changes, some purely functional solutions adopt a function-like view of widgets themselves. Fudgets [22] is an asynchronous programming library built on top of stream processors. In the context of GUIs, a fudget is a widget with *one input* and *one output*[3]. A number of combinators connect fudgets and determine their visual layout. However, and just for that reason, there is no way to connect visually non-adjacent widgets.

Gadgets [23], a similar approach, tries to work around Fudgets' limitation of one input and output channel per process. Gadgets are visual components built from more elementary types representing processes and wires (connections between components). Gadgets are asynchronous, and their types do not give enough information about the rates at which data is being produced

[3]In general terms, a fudget is a process which can communicate with other concurrently running fudgets, and the outside world.

```
import Fudgets

main = fudlogue (shellF "Up Counter" counterF)

counterF = intDispF          >==<
           mapstateF count 0 >==<
           buttonF "Up"

count n Click = (n + 1, [n + 1])
```

Figure 3.1: A small Fudgets program and a screenshot of the GUI it generates. *intDispF* is an integer display text fudget, *mapstateF* keeps a counter, and *buttonF* is a button fudget. (>==<) chains fudgets from right to left and places them next to one another in the GUI.

or consumed. Furthermore, gadgets that represent on-screen elements take the element attributes as an input data structure, which requires more resources for elements with many or with complex attributes, as it is common in GTK+. Both in Gadgets and Fudgets, code is more declarative than in imperative toolkits, but both are limited in terms of feature coverage.

The freedom to define a purely functional abstraction for the domain is both a blessing and a curse: to cover the whole domain, one needs to implement all possible types and operations that may be needed. For GUIs, this implies a type, an implementation and set of operations for each kind of widget supported. GUI toolkits are notoriously large[4], and this results in *very large codebases with high maintenance costs*[5], rendering some projects unrealistic in the long term. Furthermore, different platforms behave slightly differently. Creating a unique GUI abstraction that provides all the features of each platform under a common, clean interface has been challenging. The opposite, maintaining several (similar) GUI backends, only exacerbates the maintenance costs.

Another problem is that some of these solutions require that the GUI be coded manually, which is suboptimal and expensive in large projects [1] (expert UI designers tend to prefer to use visual tools to design GUIs [68]). It also affects the adaptability of the interface to new OSs like Android and iOS: changes to the environment (e.g. device orientation) [69] may require switching to a completely different UI without closing the application.

A limited form of abstraction over user interface programming is pursued by implementations that generate GUIs automatically based on the types of expressions. GuiTV [70], a Haskell im-

[4]As an example, GTK2hs, the Haskell bindings to GTK+ [51], currently exports 6185 symbols, including widget constructors, widget properties, event handler installers and other necessary types and functions.

[5]This extra cost is not exclusive to functional languages, but mainly the result of defining a new, large API for the domain. In languages that allow effects everywhere there hardly is a motivation to define such an expensive intermediate GUI abstraction.

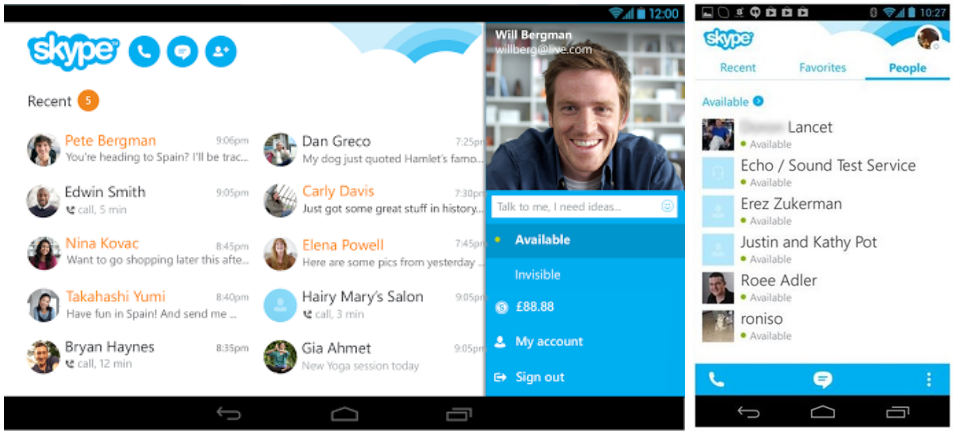


Figure 3.2: A side-by-side view of Microsoft's Skype running on a tablet (left) and a mobile phone (right). The program is the same, only the GUIs, specified in two separate XML files, change.

plementation of type-based GUI generators, can construct widget compositions for functions, to provide the arguments and show the result, eliminating one level of indirection between models and visualisations (see Figure 3.3).

$$reverseT :: CTV\ (String \to String)$$
$$reverseT = tv\ (oTitle\ \texttt{"reverse"}\ defaultOut)\ reverse$$

Figure 3.3: A Tangible Value and the GUI generated for it based on the (function) type of the CTV, with text boxes being used to interact with *String*s.

A similar idea is used in iTask [71], a client-server, task-oriented [72], state-transforming, monadic web programming framework for the Clean [73] language. Tasks produce values that change over time. They are stateful event-processors that may be composed in parallel and sequentially. User interfaces are generated automatically based on types [74], and then rendered on a browser. iTasks tries to address large-scale architectural concerns in GUI applications, relying on automatic GUI generation to cover low-level implementation details and focusing on the definition of data-oriented tasks that define access points to parts of the underlying model or to subsets that need to be provided in sequence (using monadic combinators to define these sequences). It is unclear, at this point, whether iTasks could target other platforms, including desktop and mobile, connect to existing GUI toolkits like GTK+, or whether it includes all the necessary features that exist in GUI toolkits.

Generating GUIs based on the model's type definition is similar to Ruby on Rails' runtime *scaffolding* [75]. A well-known limitation of this approach was that views cannot be fully adapted[6]. Dynamic (runtime) scaffolding was substituted in new versions of Ruby on Rails by generators that create UI code in different files, which developers are encouraged to adapt and modify [76].

Similarly, the idea of uniquely mapping one type to one kind of widget seems inflexible, as there may be more than one right way to interact with a specific value in our program. To circumvent this problem, one can use type wrappers (e.g. Haskell's **newtype**), at the expense of additional boilerplate code[7].

3.2.1 Limitations of Purely Functional Interactivity

Using a purely functional abstraction for interactive elements provides the reasoning and composability advantages that would be expected from functional languages.

However, as we have seen, many libraries that follow this approach need to introduce definitions for each and every kind of interactive or visual element that can be represented on the screen. As GUI and multimedia toolkits are large and complex, this makes providing a functional toolkit that includes all kinds of widgets, properties, events, and layout control for all platforms extremely expensive.

Furthermore, GUIs must be realisable using visual tools, with full control over the representation. Hard-coded GUIs are not flexible enough, for instance, they may not adapt well to landscape or portrait orientations or to devices of multiple sizes. This makes creating a comprehensive functional GUI toolkit even more expensive, as the toolchain that accompanies it would also be required.

3.3 Functional Reactive Programming

Functional Reactive Programming [25, 26, 27] is a programming paradigm to describe systems that operate on time-varying data. FRP is structured around the concept of signal, which conceptually can be seen as a function from time to values of some type:

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

Time is (notionally) 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, the type of an animation

[6] iTasks enable custom layout using style sheets. The authors also acknowledge that automatic UI generation is useful when prototyping, but customisation must be possible in real-world applications [71].

[7] All the operations defined for the type, including necessary instances of (type) classes, must also be reimplemented, making the impact of this approach quite palpable.

would be *Signal Picture* for some type *Picture* representing static pictures. Signals can also represent input data, like the mouse position.

There are multiple variants of FRP, all built around a notion of time-variance and reactivity (for a more exhaustive review, see [77]). Like Reactive Programming [14, 15], FRP focuses on defining *what* time-varying entities are and their relationships (declarative programming), as opposed to *how* they must be updated when one of them changes (imperative programming).

There are an even larger number of different *implementations* of FRP frameworks catering for different settings in terms of platforms and languages, application needs, and ideas about how to structure such frameworks in the most appropriate way. Particular differences include whether a discrete or hybrid (mixed discrete and continuous) notion of time is adopted, how systems are simulated or sampled, and how to handle I/O and other effects in a way that is both notationally convenient and scalable.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP [25] and Arrowized FRP [27, 40]. Classic FRP programs are structured around signals or a notion representing internal and external time-varying data. In contrast, Arrowized FRP programs are defined using causal functions between signals, or *signal functions*.

FRP implementations try to maximise clarity, reliability and efficiency, but they may improve one aspect at the expense of another. Some forms of Classic FRP introduce I/O, explicitly or implicitly, to improve performance [78] or versatility [79]. However, in doing so, some sacrifice determinism and static guarantees. In contrast, some forms of Arrowized FRP separate I/O from the FRP logic itself. This facilitates reasoning, and has the additional benefit that tests are fully reproducible. However, it also introduces structural and performance limitations. We refer to those FRP variants that use I/O as impure, and to those that separate I/O from logic as pure.

This thesis proposes a unifying reactive framework for Reactive Programming and FRP, designed by generalising Arrowized FRP. In the remainder we turn to Arrowized FRP for an introduction and understanding of the limitations we wish to address. A more in-depth exploration of the field and other related work is presented in Chapter 11.

3.3.1 Arrowized FRP

Additional constraints are required to make signals, as described before, executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be causal: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time.

Arrowized FRP is a variant in which systems are described as synchronous dataflow networks composed of *signal functions*. Signals in FRP are not first-class citizens, and they cannot be sampled at arbitrary times. Conceptually:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

Signal Functions must maintain signal causality, that is, the output signal at a time cannot depend on the input signal at any future time.

In the following, we explore Arrowized FRP as realised by Yampa [26, 27], an implementation that separates I/O from the FRP logic completely and will serve as a starting point for the abstraction proposed in the next chapter.

Basic SF Combinators

Signal Functions are Arrows [36, 42], and we can use standard arrow combinators to manipulate them, with the following types:

$$\begin{array}{lll}
arr & :: (a \to b) & \to SF\ a\ b \\
(***) & :: SF\ a\ b & \to SF\ c\ d \to SF\ (a, c)\ (b, d) \\
(\&\&\&) & :: SF\ a\ b & \to SF\ a\ c \to SF\ a\ (b, c) \\
first & :: SF\ a\ b & \to SF\ (a, c)\ (b, c) \\
(>>>) & :: SF\ a\ b & \to SF\ b\ c \to SF\ a\ c \\
loop & :: SF\ (a, c)\ (b, c) & \to SF\ a\ b
\end{array}$$

Example We can apply a transformation at every point in time to increase an input signal in one unit with the definition:

$$\begin{array}{l}
plusOne :: SF\ Int\ Int \\
plusOne = arr\ (+1)
\end{array}$$

We can combine this signal function with itself and apply it twice, as follows:

$$\begin{array}{l}
plusTwo :: SF\ Int\ Int \\
plusTwo = plusOne >>> plusOne
\end{array}$$

The following construct duplicates the input signal and applies *plusOne* to one signal and *plusTwo* to the other, producing an output signal that carries tuples:

$$\begin{array}{l}
plusTuple :: SF\ Int\ (Int, Int) \\
plusTuple = plusOne\ \&\&\&\ plusTwo
\end{array}$$

Time-Variant Signal Functions: Integrals and Derivatives

Signal Functions must remain leak-free. To prevent memory leaks, we must limit how much of the past values of other signals can be explored and retained during execution, and Yampa introduces limited ways of depending on past values of other signals. One such way is via *integral* and *derivative*, which approximate the calculation of the integral or derivative from the beginning of the execution while ensuring that it does not produce time or space leaks. Integrals and derivatives are important for application domains like games, multimedia and physical simulations, and they have well-defined continuous-time semantics. Their types in Yampa are as follows:

$$
\begin{aligned}
&\mathit{integral} \quad :: \mathit{VectorSpace}\ v\ s \Rightarrow \mathit{SF}\ v\ v \\
&\mathit{derivative} :: \mathit{VectorSpace}\ v\ s \Rightarrow \mathit{SF}\ v\ v
\end{aligned}
$$

Time-aware primitives like the above make FRP specifications highly declarative. For example, the position of a falling mass starting from a position $p0$ with initial velocity $v0$ is calculated as:

$$
\begin{aligned}
&\mathit{fallingMass} :: \mathit{Double} \to \mathit{Double} \to \mathit{SF}\ ()\ \mathit{Double} \\
&\mathit{fallingMass}\ p0\ v0 = \mathit{arr}\ (\mathit{const}\ (-9.8)) \\
&\quad \ggg \mathit{integral} \ggg \mathit{arr}\ (+v0) \\
&\quad \ggg \mathit{integral} \ggg \mathit{arr}\ (+p0)
\end{aligned}
$$

This resembles well-known physics equations (i.e. *the position is the integral of the velocity with respect to time*) even more when expressed using Paterson's Arrow notation [37]:

$$
\begin{aligned}
&\mathit{fallingMass} :: \mathit{Double} \to \mathit{Double} \to \mathit{SF}\ ()\ \mathit{Double} \\
&\mathit{fallingMass}\ p0\ v0 = \mathbf{proc}\ () \to \mathbf{do} \\
&\quad v \leftarrow \mathit{arr}\ (+v0) \lll \mathit{integral} \multimap (-9.8) \\
&\quad p \leftarrow \mathit{arr}\ (+p0) \lll \mathit{integral} \multimap v \\
&\quad \mathit{returnA} \multimap p
\end{aligned}
$$

Events and Event Sources

While some signals exist and vary continuously (like *fallingMass* or the mouse position), we can define signals that only exist or change at discrete points in time (like *fallingMass* being exactly 0, or the mouse button being clicked).

We use *discrete* events to capture the notion of signals that change only at discrete points in time [27]:

$$
\mathbf{data}\ \mathit{Event}\ a = \mathit{NoEvent} \mid \mathit{Event}\ a
$$

Example We can modify the previous *fallingMass* to output also an event signal that indicates when the mass hits the ground. When the event takes place, we use the value in the event to indicate the velocity at the time of impact.

$$
\begin{aligned}
&\mathit{fallingMass'} :: \mathit{Double} \to \mathit{Double} \to \mathit{SF}\ ()\ (\mathit{Double}, \mathit{Event}\ (\mathit{Double}, \mathit{Double})) \\
&\mathit{fallingMass'}\ \mathit{p0}\ \mathit{v0} = \mathbf{proc}\ () \to \mathbf{do} \\
&\quad v \leftarrow \mathit{arr}\ (+\mathit{v0}) \lll \mathit{integral} \rightarrowtail (-9.8) \\
&\quad p \leftarrow \mathit{arr}\ (+\mathit{p0}) \lll \mathit{integral} \rightarrowtail v \\
&\quad \mathbf{let}\ e = \mathbf{if}\ p \leqslant 0\ \wedge\ v \leqslant 0\ \mathbf{then}\ \mathit{Event}\ (p, v)\ \mathbf{else}\ \mathit{NoEvent} \\
&\quad \mathit{returnA} \rightarrowtail (p, e)
\end{aligned}
$$

Switching

Switching is a mechanism that allows us to apply a transformation until a certain condition is met, and then start applying a different transformation. Switching uses events to detect the time when the switching must take place, and to determine the initial information made available to the signal function turned on after switching.

$$\mathit{switch} :: \mathit{SF}\ a\ (b, \mathit{Event}\ c) \to (c \to \mathit{SF}\ a\ b) \to \mathit{SF}\ a\ b$$

Example Using switching we can restart *fallingMass′* to invert the velocity when the mass hits the ground, making it go up instead of down:

$$
\begin{aligned}
&\mathit{bouncingMass} :: \mathit{Double} \to \mathit{Double} \to \mathit{SF}\ ()\ \mathit{Double} \\
&\mathit{bouncingMass}\ \mathit{p0}\ \mathit{v0} = \mathit{switch}\ (\mathit{fallingMass'}\ \mathit{p0}\ \mathit{v0})\ (\lambda(\mathit{p1}, \mathit{v1}) \to \mathit{bouncingMass}\ \mathit{p1}\ (-\mathit{v1}))
\end{aligned}
$$

Simulation

Signal Functions are connected to the outside world only at the top level, by connecting them to input signals and by consuming their outputs. The process of animating signals is called, in FRP, *reactimating*. A simplified *reactimate* function is as follows:

$$
\begin{aligned}
\mathit{reactimate}\ ::\ & \mathit{IO}\ (\mathit{DTime}, a) && \text{-- Sense time delta and input} \\
& \to (b \to \mathit{IO}\ ()) && \text{-- Actuate/render} \\
& \to \mathit{SF}\ a\ b \\
& \to \mathit{IO}\ ()
\end{aligned}
$$

Signal Functions are executed by sampling the input signal at different times (*sensing*), processing the input using the signal function, and consuming the output sample (*actuating*). This

approximates the continuous-time ideal definition of signal functions, which becomes more accurate as the sampling frequency increases.

Input samples are paired with the time deltas, or times passed between consecutive samples. We maintain causality at a signal level by requiring that these time deltas be strictly positive, so that simulations always progress towards the future.

3.3.2 Limitations of FRP and Arrowized FRP

The Arrowized FRP variant presented so far has a number of limitations. Most of these also apply to other current AFRP systems, and some also apply to other non-Arrowized FRP variants.

Fixed Time Domain and Clock Yampa has a global clock that progresses as driven by an external time-sensing monadic (I/O) action. This is a serious limitation, as some games require the use of nested clocks (game clock versus application clock), and others require that time progresses at different speeds or with different precisions for different parts of the game. A consequence of having an external global clock for the whole system is that Continuous Collision Detection (CCD) in Yampa is very complex, since CCD requires that we have local control of time and the clock to calculate a nearest future time of impact and simulate the system at that specific time.

The time domain in Yampa is fixed to *Double*, which is not always the most appropriate. Many games run on a discrete clock, while others require a rational clock with arbitrary precision or no clock at all. In such cases, keeping and passing an additional continuous clock becomes an unnecessary nuisance.

I/O Bottleneck Yampa's input and output is connected to the external world once at the top level, in the invocation of the function that runs a signal function. This helps keep Signal Functions pure and referentially transparent across executions, meaning that Signal Functions will return the same output it provided with the same input and polled at the same sampling time. However, this design requires that all input data be polled every time, and leads to more complex data structures in the output [19].

Explicit Wiring Pure (I/O-free) AFRP implementations do not allow communication between signal functions except through explicit input/output signals. All data that a signal function needs must be manually routed down, and outputs manually routed up. In practice, we often want to make part of that wiring implicit.

A manifestation of this problem is that it is not possible to debug from within signal functions, except by adding explicit output signals carrying debugging information or by using functions like

Debug.Trace.trace, which output to standard output directly (using *unsafePerformIO* under the hood). Code that uses *trace* is not portable, for instance, to platforms like Android, as debug messages must be printed to a special debug log facility.

Referential Transparency Across Executions FRP programs use I/O to gather input data and render their output. Unlike in pure Arrowized FRP, some FRP implementations hide I/O behind the scenes, and reading a signal may result in I/O actions being executed [80]. While they use low-level adjustments to guarantee that polling a signal at the same conceptual time delivers temporally consistent results across the whole program, without additional adjustments, these implementations lack referential transparency *across executions*.

Even in IO-free forms of FRP, like the one presented in this section, simply sampling a simulation at different points in time can render different results. The non-determinism introduced by the sampling policy is demonstrated by the force-directed graph simulation in pure FRP depicted in Figure 3.4. In this simulation, nodes repel each other, but also attract those they are directly connected to, creating a joint that behaves like a spring. Implemented in Pure Arrowized FRP, this simulation is not interactive, but the numerical methods used to calculate forces, velocities and positions depend on the sampling time, and are highly affected by small variations. As a result, the layouts obtained when the simulation stabilises are very dependent on the sampling step.

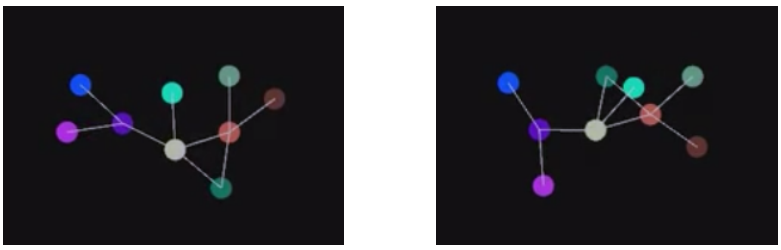


Figure 3.4: Two runs of the same graph layout algorithm. Even with identical initial conditions, two executions converge onto different stable configurations, demonstrating that even pure FRP systems exhibit non-determinism.

Because pure Arrowized FRP separates all data processing from all side effects, including the *time sampling policy*, we could truly run the program twice with the exact same input and sample at the same times, obtaining the same results. This constitutes a form of referential transparency across executions suitable to debug complex interactive applications, especially games.

This possibility of adding testing and debugging at the layer that separates the UI or multimedia framework from the logic itself makes pure Arrowized FRP extremely suitable to define robust multimedia, provided that we have the mechanisms to define interactive properties, to test them

and to find bugs in our programs.

Style and code modularity FRP tries to shift the direction of data-flow from *message passing* to *data dependency*. This helps reason about *what things are over time*, as opposed to *how changes propagate*.

One way to impose such restriction is to have one, and only one, place in our code, where a signal can be defined. That is, signals are defined *by* their values over time: we cannot declare a signal in one place and define its value over time somewhere else. This has traditionally been seen as an advantage, as it results in clear and declarative specifications. There are, however, at least two aspects in which FRP may be improved.

First, and in spite of the wide interest in FRP, multiple reports claim that it is difficult to understand and to model systems using FRP [81]. This may be due to lack of tutorials and examples, and not a deficiency of FRP as such, but it pays to try to use clearer programming style and avoid code redundancy whenever possible[8].

A second problem is that, in user interfaces, FRP may limit our ability to modularise our code appropriately. Consider, for instance, the screenshot from Xournal [82] shown in Figure 3.5. There are four different ways to move from one page to the next: with the toolbar buttons (top), by dragging the central area with the mouse (centre left), by scrolling down the page (centre right), and with the bottom toolbar controls. Each of these acts *both as an input and an output*. No matter which of the four methods we use, the central area will show different contents, and scroll bar will be at a different position, the toolbar buttons will be enabled/disabled depending on whether there are more pages before/after, and the bottom toolbar page selection text entry will show a different number. The following pseudo-FRP code illustrates their mutual dependencies:

```
toolbarButtonRight ← button "rightarrow.png"
                         [enabled := liftA2 (not ∘ isLast) currentPage numPages]
pageSelectionEntry ← numEntryText [value := currentPage]
pageArea           ← renderPage file currentPage
currentPage        ← accum 0
                         [(clickOn toolbarButtonRight `tag` (+1))
                          `merge` (enterText pageSelectonEntry `tag`
                                       (const (value pageSelectionEntry)))
                          `merge` ...
                         ]
```

[8] In particular, I have found, in myself and in multiple FRP users that I have spoken to, that dynamism (mode switching) and looped signal definitions (using let or *ArrowLoop*) tend to pose serious challenges.

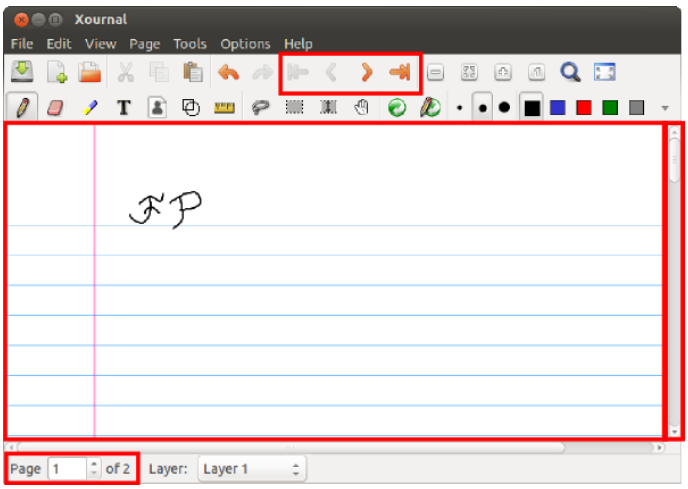


Figure 3.5: Screenshot of Xournal highlighting four different ways to change the page number.

We need *currentPage* to define *toolbarButtonRight* (line 1), *pageSelectionEntry* (line 3) and *pageArea* (line 4), but we need all three (and probably many others) to define the value of *currentPage* (line 5). These mutually dependent elements need to be in scope of one another. In languages like Haskell, with limited support for circular imports, this translates into all those elements being defined in the same module, and possibly in the same function, thus impairing modularity and separation of concerns. This only gets worse as the codebase grows.

Some FRP and FRP-inspired implementations and languages offer mechanisms to work around this problem. Elm [83], for instance, offers *handles* to push specific changes onto widgets, thus helping to break cycles involving interactive visual elements. Reactive Banana [84] offers *sinks* for each WX widget property, to which a signal can be attached. These mechanisms are not general solutions applicable to every reactive element, but ad hoc solutions to enable pushing changes to specific kinds of resources.

This pattern is common enough to motivate a more general solution. The proposal presented in later chapters has been tested in real-world applications to validate that it enables modularity and scalability as applications grow in size and complexity.

3.4 Summary

In this chapter we have explored three approaches to programming interactive software in functional languages: low-level imperative programming, high-level functional UI toolkits, and Functional Reactive Programming.

Low-level programming provides advantages in terms of feature coverage and speed, but re-

quires a sacrifice in terms of reasoning that limits the benefits of using functional programming. Functional multimedia and UI toolkits provide those benefits, but carry with them an increased production cost that makes them hard to maintain for practical and economic reasons.

Functional Reactive Programming provides a middle ground between the two, allowing us to describe interactive elements in an abstract way, to reason about them, and to connect them to existing multimedia frameworks. As currently realised, some FRP frameworks have inflexible designs that limit their modularity, performance, testability and their application to multiple domains of interest in GUI and interactive software.

In the next chapter we explore a proposal for an extensible functional abstraction that overcomes many of these limitations, and provides a suitable middle ground on top of which other Reactive and Functional Reactive notions can be implemented without sacrificing modularity or robustness.

Part II

Extensibility

Chapter 4

Monadic Stream Functions

As discussed in Chapter 3, there are a very large number of different FRP implementations catering for different platforms, languages, application needs, and ideas about how to structure such frameworks in the most appropriate way [85]. Particular differences include whether a discrete or hybrid (mixed discrete and continuous) notion of time is adopted, and how to handle I/O and other effects in a way that is both notationally convenient and scalable.

While this diversity makes for a rich and varied FRP landscape, it does raise practical obstacles for the FRP user in terms of which system to pick, especially if the needs are diverse or can be anticipated to change.

Given the underlying conceptual similarity between the various implementations, this raises the question of whether, through appropriate generalisations, it might be possible to provide a single framework that covers a significantly broader set of applications and needs than what any one FRP implementation up until now has done, and that is easily extensible.

This chapter proposes such a unifying framework by adopting stream functions parametrised over monads as the central reactive abstraction. Through composition of standard monads like *Reader*, *Either*, and *State*, any desirable combination of time domain, dynamic system structure, flexible handling of I/O and more can be obtained in an open-ended manner. In Chapters 5 and 7 we will see how such a minimalistic framework subsumes and exceeds existing FRP frameworks.

4.1 Basic Monadic Stream Functions

Monadic Stream Functions (MSFs) are a minimal abstraction to represent synchronous, effectful, causal functions on streams. MSFs are a generalisation of Yampa's Signal Functions (SFs), with additional combinators to control and stack side effects. While the fundamentals of the abstrac-

tion are standard (continuations, parametrisation on a monad [27, 42]), this approach enables for systematic effect extensions and subsumes existing FRP frameworks, addressing shortcomings mentioned in Chapter 3.

Notation The definitions that follow are simplified for clarity; performance and memory footprint of the implementation are discussed in Chapter 12. The first argument of an *MSF* is always a *Monad*, and consequently also a *Functor* and an *Applicative*. The corresponding *Monad* constraint is omitted from type signatures to reduce clutter.

4.1.1 Definitions

Monadic Stream Functions are defined by a polymorphic type *MSF* and an evaluation function that applies an *MSF* to an input and returns, in a monadic context, an output and a continuation:

$$\textbf{newtype}\ \mathit{MSF}\ m\ a\ b = \ldots$$

$$\mathit{step} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \rightarrow a \rightarrow m\ (b, \mathit{MSF}\ m\ a\ b)$$

Functions to build and combine MSFs will be provided in the rest of the chapter. We purposefully hide the details of the implementation of MSF, which will be shown later on[1].

The type *MSF* and the *step* function alone do not represent causal functions on streams. It is only when we successively apply the function to a stream of inputs and consume the side effects that we get the unrolled, streamed version of the function. Causality, or the requirement that the n-th element of the output stream only depend on the first n elements of the input stream, is obtained as a consequence of applying the MSF continuations *step by step*, or sample by sample. Using *step*, from the first MSF and the first sample we obtain, in a monadic context, an output sample and a new MSF, which is used to transform the second input sample, and so forth. Therefore, the n-th MSF, used to transform the n-th input sample, has been built from the $n-1$-th MSF, and so on.

For the purposes of exposition, we will use the following function to apply an MSF to a *finite list* of inputs, with effects and continuations chained sequentially. This is merely a debugging aid, not how MSFs are actually executed:

$$\mathit{embed} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \rightarrow [a] \rightarrow m\ [b]$$

[1] The use of the function *step* to define the behaviour of MSFs, as opposed to giving a model implementation, may seem a misleading attempt at appearing rigurous. The purpose of hiding the constructor is simply to limit users to the API that will be provided in the rest of the chapter, which has been designed to preserve causality and avoid leaks. Exposing the data constructor would also make all the following definitions use it explicitly, which would confuse readers and open the door to ill-behaved definitions.

4.1.2 Lifting

The simplest kind of transformation we can apply to a stream is point-wise to every sample. For the trivial case, when there are no side effects in that transformation, we define:

$$arr :: (a \to b) \to MSF\ m\ a\ b$$

This applies a state-less transformation to every sample, which we can describe by showing how it transforms the head and that the transformation applied to future samples is always the same:

$$step\ (arr\ f)\ a \equiv return\ (f\ a, arr\ f)$$

That is, the first output is f applied to the input, and the continuation remains unchanged.

More generally, we can apply an effectful computation to every sample:

$$liftS :: (a \to m\ b) \to MSF\ m\ a\ b$$

We describe the meaning of *liftS* by showing that it acts point-wise on the head, and that the continuation is the same *MSF* unchanged:

$$step\ (liftS\ f)\ a \equiv f\ a \gg\!= \lambda b \to return\ (b, liftS\ f)$$

It can be trivially shown that $arr\ f \equiv liftS\ (return \circ f)$, which makes *arr* not primitive in this framework.

Example One trivial way of using these combinators is the stream function that adds a constant number to the input:

$$\begin{aligned} &add \quad :: (Num\ n, Monad\ m) \Rightarrow n \to MSF\ m\ n\ n \\ &add\ n0 = arr\ (\lambda n \to n + n0) \end{aligned}$$

which we test in a session (in GHCi, monad-parametric computations are run in the IO monad and the results printed, if possible):

$$\begin{aligned} &> embed\ (add\ 7)\ [1, 2, 3] \\ &[8, 9, 10] \end{aligned}$$

4.1.3 Widening

We can also define new MSFs that only affect the first or second components of pairs, passing the other component completely unmodified:

$$\begin{aligned} &first \quad :: MSF\ m\ a\ b \to MSF\ m\ (a, c)\ (b, c) \\ &second :: MSF\ m\ b\ c \to MSF\ m\ (a, b)\ (a, c) \end{aligned}$$

The meaning of *first*, using the function *step*, is as follows:

$$
\begin{aligned}
&\mathit{step}\ (\mathit{first}\ \mathit{msf})\ (a, c) \equiv \\
&\quad (b, \mathit{msf}') \leftarrow (\mathit{step}\ \mathit{msf})\ a \\
&\quad \mathit{return}\ ((b, c), \mathit{first}\ \mathit{msf}')
\end{aligned}
$$

This produces an *MSF* that applies the auxiliary *MSF* only to one component of the input, obtaining an internal output and continuation. This result is used to build the output stream, putting it together with the unused input component, and applying *first* to the continuation.

Analogously, the meaning of *second* can be described as:

$$
\begin{aligned}
&\mathit{step}\ (\mathit{second}\ \mathit{msf})\ (a, b) \equiv \\
&\quad (c, \mathit{msf}') \leftarrow (\mathit{step}\ \mathit{msf})\ b \\
&\quad \mathit{return}\ ((b, c), \mathit{first}\ \mathit{msf}')
\end{aligned}
$$

It can be trivially shown that $\mathit{second}\ \mathit{msf} = \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ \mathit{msf} \ggg \mathit{arr}\ \mathit{swap}$, where $\mathit{swap}\ (x, y) = (y, x)$.

Examples Extending the previous example, we can write:

$$
\begin{aligned}
&> \mathit{embed}\ (\mathit{second}\ (\mathit{add}\ 7))\ [(1,1),(2,2),(3,3)] \\
&[(1,8),(2,9),(3,10)]
\end{aligned}
$$

4.1.4 Serial and Parallel Composition

*MSF*s can be composed serially using the combinator ($\ggg$). Composing $f \ggg g$ first applies f to the input producing an output, and applies g to that output, producing a final result. Side effects are sequenced in the same order, which we can express succinctly using *step*:

$$
\begin{aligned}
&(\ggg) :: \mathit{MSF}\ m\ a\ b \to \mathit{MSF}\ m\ b\ c \to \mathit{MSF}\ m\ a\ c \\
&\mathit{step}\ (f \ggg g)\ a \equiv \mathbf{do}\ (b, f') \leftarrow (\mathit{step}\ f)\ a \\
&\qquad\qquad\qquad\qquad\quad (c, g') \leftarrow (\mathit{step}\ g)\ b \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (c, f' \ggg g')
\end{aligned}
$$

This description of the behaviour of MSF composition highlights why we need a *Monad* type constraint on m, instead of just a *Functor* or an *Applicative*, in the definition of *MSF*, for the function *step*: we cannot write the application of the second *MSF* to the result b using just *Applicative* combinators [86]. Using *Applicative*s, the application of the second *MSF* to the first *MSF* would result in a result of type $m\ (m\ (c, \mathit{MSF}\ m\ a\ c))$ in the last line in the previous

definition, which would require the use of the monadic function *join* to flatten the monadic result down one layer of ms, making it of the required type $m\ (c, MSF\ m\ a\ c)$.

MSFs can be composed in parallel with the *Arrow* functions ($***$), which applies one MSF to each component of a pair, and ($\&\&\&$), which broadcasts an input to two MSFs applied parallely:

$$(***) :: MSF\ m\ a\ b \to MSF\ m\ c\ d \to MSF\ m\ (a, c)\ (b, d)$$
$$(\&\&\&) :: MSF\ m\ a\ b \to MSF\ m\ a\ c \to MSF\ m\ a\ (b, c)$$

These combinators are not primitive; their standard definitions are based on combinators defined previously:

$$f\ ***\ g = first\ f \ggg second\ g$$
$$f\ \&\&\&\ g = arr\ (\lambda x \to (x, x)) \ggg (f *** g)$$

When composed in parallel like above, the effects of the f are produced before those g. This is of prime importance for non-commutative monads. For a discussion, see Chapter 11.

Example The following example of palindromes demonstrates both monadic lifting and composition combinators:

```
testSerial :: MSF IO () ()
testSerial = liftS (λ() → getLine) >>> (arr id &&& arr reverse) >>> liftS print
```

We explore the behaviour in a session:

```
> embed testSerial (replicate 2 ())
Text
("Text","txeT")
A second input text
("A second input text","txet tupni dnoces A")
```

4.1.5 Depending on the Past

To keep state by producing an extra value or *accumulator* accessible in future iterations we use:

$$feedback :: c \to MSF\ m\ (a, c)\ (b, c) \to MSF\ m\ a\ b$$

This combinator takes an initial value for the accumulator, runs the MSF, and feeds the new accumulator back for future iterations:

```
step (feedback c msf) a ≡ do
  ((b, c'), msf') ← step msf (a, c)
  return (b, feedback c' msf')
```

Unlike in the definition of *loop* from the *ArrowLoop* type class, the input and the output accumulator are not the same during a simulation step. Conceptually, there is a one-step delay in the wire that feeds the output c back as input for the next run.

Example The following calculates the cumulative sum of its inputs, initializing an accumulator and using a feedback loop:

```
sumFrom :: (Num n, Monad m) => n -> MSF m n n
sumFrom n0 = feedback n0 (arr add2)
  where
    add2 (n, acc) = let n' = n + acc in (n', n')
```

A counter can now be defined as follows:

```
count :: (Num n, Monad m) => MSF m () n
count = arr (const 1) >>> sumFrom 0
```

4.1.6 Example: One-Dimensional Pong

Before moving on to control structures and monadic MSFs, let us examine part of an example of a game of pong in one dimension.

The game state is just the position of the ball at each point. We assume the players sit at fixed positions:

```
type Game = Ball
type Ball = Int

rightPlayerPos = 5
leftPlayerPos  = 0
```

The ball will move in either direction, one step at a time:

```
ballToRight :: Monad m => MSF m () Ball
ballToRight = count >>> arr (\n -> leftPlayerPos + n)

ballToLeft  :: Monad m => MSF m () Ball
ballToLeft  = count >>> arr (\n -> rightPlayerPos - n)
```

We can detect when the ball should switch direction with:

```
hitRight :: Monad m => MSF m Ball Bool
hitRight = arr (>= rightPlayerPos)
```

$$
\begin{aligned}
&\mathit{hitLeft} &&:: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ \mathit{Ball}\ \mathit{Bool}\\
&\mathit{hitLeft} &&= \mathit{arr}\ (\leqslant \mathit{leftPlayerPos})
\end{aligned}
$$

Switching itself will be introduced in Section 4.2.3, when we explore control flow.

4.1.7 Mathematical Properties

Monadic Stream Functions are Arrows [36] when applied to any monad. Using the Arrow abstraction to describe MSFs helps capture their conceptual meaning of composable transformations, with a notion of identity and widening, in a functional and declarative fashion, in spite of MSFs being effectful.

A proof of one of the arrow laws, as well as links to proofs of other laws and additional properties, are included in Appendix A. By constraining the monad we can obtain additional guarantees about MSFs. MSFs are *Commutative Arrows* [87] when applied to commutative monads (Section 2.3), a result we could exploit for optimisation purposes. For monads that are instances of *MonadFix*, the implementation provides a well-behaved MSF instance of *ArrowLoop*. The implementation also defines instances that facilitate writing declarative code, like *ArrowChoice* and *ArrowPlus*. MSFs partially applied to inputs are functors and applicative functors.

There are several isomorphisms between Monadic Stream Functions and Monadic Streams defined as $\mathit{Stream}\ m\ a = m\ (a, \mathit{Stream}\ m\ a)$. Some properties are easier to prove or to express when MSFs are seen as streams, something we elaborate on in Section 4.3. In turn, abstractions defined in terms of streams can be expressed using MSFs, an idea we use in Chapters 5 and 7 to obtain implementations of reactive frameworks for free.

4.2 Monads, Modularity and Control

This section motivates and explores the use of different monads with Monadic Stream Functions. Monads like *Reader* and *State* help modularise and increase the expressiveness of programs. Others like *Maybe*, *Exception* and the list monad give rise to control combinators for termination, higher order and parallelism.

Temporal Execution Functions Monads and monad transformers [88] have associated *execution* functions to run computations and extract results, consuming or trapping effects in less structured environments. For instance, in the monad stack $\mathit{ReaderT}\ e\ m$ we can eliminate the layer $\mathit{ReaderT}\ e$ with the function $\mathit{runReaderT} :: e \to \mathit{ReaderT}\ e\ m\ a \to m\ a$, obtaining a value in the monad m.

Analogously, MSFs applied to monads have associated *temporal execution functions*, which limit effects to part of a reactive network. In this section we introduce MSF lifting/running combinators for concrete monads, exploring how they augment expressiveness, help modularise code and give rise to known structural reactive combinators. In Section 4.3 we discuss a general way of combining effects in MSFs. In Chapter 5 we use these combinators to express FRP abstractions like Arrowized FRP.

4.2.1 Reader

Imagine now that we want to make our example MSFs parametric on the player positions. One option is to pass the player position as an argument, as in $ballToRight :: Monad\ m \Rightarrow Int \rightarrow MSF\ m\ ()\ Ball$. However, this complicates the implementation of every MSF that uses *ballToRight*, which needs to manually pass those settings down the reactive network. We can define such a parametrisation in a modular way using a *Reader* monad with the game preferences in an environment (we use *ReaderT* for forward compatibility):

```
type GameEnv = ReaderT GameSettings

data GameSettings = GameSettings
  { leftPlayerPos  :: Int
  , rightPlayerPos :: Int
  }
```

We rewrite the game to pass this environment in the context:

```
ballToRight :: Monad m => MSF (GameEnv m) () Ball
ballToRight = count >>> liftS (\n -> (n+) <$> asks leftPlayerPos)

hitRight    :: Monad m => MSF (GameEnv m) Ball Bool
hitRight    = liftS (\i -> (i >=) <$> asks rightPlayerPos)
```

To run a game with a fixed environment, we could use the function from the *ReaderT* monad transformer $runReaderT :: ReaderT\ r\ m\ a \rightarrow r \rightarrow m\ a$ as before. We test the expression $testMSF = ballToRight \ggg (arr\ id\ \&\&\&\ hitRight)$ with different settings as follows:

```
> runReaderT (embed testMSF (repeat 5 ())) (GameSettings 0 3)
[(1, False), (2, False), (3, True), (4, True), (5, True)]
> runReaderT (embed testMSF (repeat 5 ())) (GameSettings 0 2)
[(1, False), (2, True), (3, True), (4, True), (5, True)]
```

This execution, however, occurs outside the invocation of *embed*, so we cannot make the game settings vary during runtime. To keep the *ReaderT* layer local to an MSF, we define a *tempo-*

ral execution function analogous to *runReaderT* (implemented using an unwrapping mechanism presented in Section 4.3):

```
runReaderS :: MSF (ReaderT r m) a b → r → MSF m a b
```

Now we can run two games in parallel with different settings:

```
> embed (      runReaderS testMSF (GameSettings 0 3)
        &&& runReaderS testMSF (GameSettings 0 2))
        (repeat 5 ()))
[((1, False), (1, False)), ((2, False), (2, False)), ((3, False), (3, True)), ((4, True), (4, True))
, ((5, True), (5, True))]
```

We could run the *MSF* obtaining a new *Reader* environment from the input signal at every iteration, giving us $runReaderS :: MSF\ (ReaderT\ r\ m)\ a\ b \rightarrow MSF\ m\ (r, a)\ b$. In Section 4.3 we see how both definitions follow from the type of the run function of the *Reader* monad, and that there is a systematic way of defining various temporal monadic execution functions.

We could also use the *Reader* monad to give access to an environment containing external elements, such as handlers or GUI widgets. This is also the approach followed by libraries not based on MSFs like Fudgets [89], which implements a form of stream-based GUIs in which the *Window* is accessible to interactive data processing elements via a *Reader* monad.

4.2.2 Writer

We can use a similar approach to introduce monads like *Writer* or *State*, for instance, to log debug messages from MSFs.

We first extend our environment with a *WriterT* wrapper:

```
type GameEnv m = WriterT [String] (ReaderT GameSettings m)
```

We now modify *ballToRight* to print a message when the position is past the right player (indicating a goal):

```
ballToRight       :: Monad m ⇒ MSF (GameEnv m) () Ball
ballToRight       = count ⋙ liftS addLeftPlayerPos ⋙ liftS checkHitR
  where
    checkHitR   :: n → GameEnv m Int
    checkHitR n = do
      rp ← asks rightPlayerPos
      when (rp > n) $ tell ["Ball at " ++ show n]
```

Notice that we have changed the monad and *ballToRight*, but the rest of the game remains unchanged. Having used the transformer *ReaderT* instead of *Reader* in the previous step now pays off in the form of added modularity.

Like with the *Reader* monad, we may be interested in consuming the context (e.g., to print accumulated messages and empty the log). We can do so with the *temporal execution function*:

$$runWriterS :: Monad\ m \Rightarrow MSF\ (WriterT\ r\ m)\ a\ b \rightarrow MSF\ m\ a\ (b, r)$$

We can test this combinator as follows:

$$> embed\ (runWriterS\ (runReaderS\ testMSF\ (GameSettings\ 0\ 3)))\ (repeat\ 5\ ()))$$
$$[((1, False), []), ((2, False), []), ((3, True), []), ((4, True), [\texttt{"Ball at 4"}]), ((5, True), [\texttt{"Ball at 5"}])]$$

Similarly, we could have used a *State* monad to define configurable game settings (for instance, settings that can be adjusted using an options menu, but remain immutable during a game run).

4.2.3 Control Flow

MSFs can use different monads to define control structures. One common construct is *switching*, that is, applying a transformation until a certain time, and then applying a different transformation.

We can implement an equivalent construct using monads like *Either* or *Maybe*. For instance, we could define a potentially terminating *MSF* as an *MSF* in a *MaybeT m* monad. Following the same pattern as before, the associated execution function would have type:

$$runMaybeS :: Monad\ m \Rightarrow MSF\ (MaybeT\ m)\ a\ b \rightarrow MSF\ m\ a\ (Maybe\ b)$$

The evaluation function *step*, instantiated for this particular monad, would have the type $MSF\ Maybe\ a\ b \rightarrow a \rightarrow Maybe\ (b, MSF\ Maybe\ a\ b)$, indicating that it may produce *no continuation*. *runMaybeS* outputs *Nothing* continuously once the internal *MSF* produces no result. "Recovering" from failure requires an additional continuation:

$$catchM :: Monad\ m \Rightarrow MSF\ (MaybeT\ m)\ a\ b \rightarrow MSF\ m\ a\ b \rightarrow MSF\ m\ a\ b$$

We can now make the ball bounce when it hits the right player:

```
ballBounceOnce :: MSF (GameEnv m) () Ball
ballBounceOnce = ballUntilRight `catchM` ballLeft
ballUntilRight :: MSF (MaybeT (GameEnv m)) () Ball
ballUntilRight = liftST (ballToRight >>> (arr id &&& hitRight)) >>> liftS filterHit
  where
    filterHit (b, c) = MaybeT $ return $ if c then Nothing else Just b
```

The utility function *liftST* is defined in Section 4.3.

We define *ballUntilLeft* analogously and complete the game:

```
game :: Monad m ⇒ MSF m () Ball
game = ballUntilRight `catchM` ballUntilLeft `catchM` game
```

Let us interpret the game by inserting a list as input stream. The output shows the ball position going up and bouncing back between 10 (the right player's position) and 0 (the left player's position).

```
> embed game $ replicate 23 ()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3]
```

The actual implementation of switching in the library is based on a more general monad *ExceptT c m*, but the key idea is the same.

4.2.4 Parallelism

In games, it is often necessary to create and destroy "objects". For example, a gun may fire a bullet, or a target vanish when hit. Such dynamicity requires specific combinators in other reactive frameworks. In the proposed framework, the list monad provides the sought-after behaviour.

A stream function over the list monad can produce zero, one or several output values and continuations. How we continue depends on our interpretation or evaluation of that list. If we explore all continuations simultaneously, we will be implementing parallel broadcasting.

To produce multiple outputs we can use *zeroA*, which produces no outputs or continuations, and <+>, which concatenates lists produced by two *MSF*s in a list monad. (In the implementation, these are available for any instance of *Alternative* and *MonadPlus*.)

```
zeroA :: Monad m ⇒ MSF (ListT m) a b
(<+>) :: Monad m ⇒ MSF (ListT m) a b → MSF (ListT m) a b → MSF (ListT m) a b
```

We now change the game logic such that, each time the ball starts moving left, it splits into two balls (note the use of (<+>)):

```
type GameEnv m = ReaderT GameSettings (ListT m)
ballLeft :: Monad m ⇒ MSF (GameEnv m) () Ball
ballLeft = singleBallLeft <+> singleBallLeft
  where
    singleBallLeft = count >>> liftS (λn → (λp → p − n) <$> asks rightPlayerPos)
```

We can escape the list monad or the *ListT* transformer by collecting the outputs from all continuations into a list:

```
runListS :: Monad m ⇒ MSF (ListT m) a b → MSF m a [b]
```

This approach proves to be very modular, and we only need to modify our top function slightly to extract the list effect:

```
mainMSF :: MSF IO () ()
mainMSF = runListS    (   runReaderS game (GameSettings 20 17)
                      &&& runReaderS game (GameSettings 10 4))
          ⋙ liftS print
```

Note that the *ReaderT* layer is inside the *ListT* layer in the definition of *GameEnv*, and therefore both games are duplicated when either ball starts moving left. Running the above MSF prints the following output (presented in two columns):

```
[(18, 5)]                    [(18, 9), (18, 9)]
[(19, 6)]                    [(17, 10), (17, 10)]
[(20, 7)]                    [(18, 9), (18, 9), (18, 9), (18, 9)]
[(19, 8), (19, 8)]           ...
```

The standard implementation of *ListT* is only valid for commutative monads. Alternative implementations exist, but the discussion is beyond the scope of this thesis.

4.2.5 State

Keeping type signatures parametric, like we did with the above stream function *ballLeft* :: *Monad m* ⇒ *MSF* (*GameEnv m*) () *Ball*, renders more reusable definitions, since we can stack more monads on top. For example, it is easy to introduce a global state using the *State* monad, with a counter of the number of rounds played. The counter can be increased with:

```
incOneRound :: Monad m ⇒ StateT Integer m ()
incOneRound = modify (+1)
```

which we use in the game, accounting for the side effect in the type:

```
game :: Monad m ⇒ MSF (GameEnv (StateT Integer m)) () Ball
game = ballToRight `untilM` hitRight `catchM`
       ballToLeft  `untilM` hitLeft  `catchM`
       (lift incOneRound `andThen` game)
```

```
  -- Defined elsewhere
andThen :: Monad m ⇒ m () → MSF m a b → MSF m a b
lift    :: (MonadTrans t, Monad m) ⇒ m a → t m a
```

The function *andThen* performs the monadic action in the first argument once and immediately carries on processing the MSF in the second argument. The function *lift*, from the transformers package, lifts the state modification into the *GameEnv* monad.

To run this reactive program we have to pass the initial state, in a similar way to $runStateT :: StateT\ s\ m\ a \to s \to m\ (a, s)$, which passes an initial state to a monadic state transformation and extracts the computed value and the final state. The corresponding function for streams has type:

```
runStateS :: Monad m ⇒ MSF (StateT s m) a b → s → MSF m a (s, b)
```

Using this function in the main loop is a simple change:

```
mainMSF :: MSF IO () ()
mainMSF = runStateS parallelGame 0 ⋙ liftS print
  where
    parallelGame = runReaderS game (GameSettings 20 17)
                 &&& runReaderS game (GameSettings 10 4)
```

The output of running this MSF (presented in two columns) is:

```
(0, (18, 5))        (1, (19, 8))
(0, (19, 6))        (1, (20, 7))
(0, (20, 7))        (1, (19, 6))
(0, (19, 8))        (1, (18, 5))
(0, (18, 9))        (1, (17, 4))
(0, (17, 10))       (3, (18, 5))
(1, (18, 9))        ...
```

The first value is the counter of total rounds, the other two values are the positions of the ball in the two games, respectively.

We have introduced this change without altering any code that did not use the state variable. In standard Arrowized FRP, we would have had to alter all the signal functions and pass the state manually.

4.3 Monadic Lifting/Running Combinators

In the last section we saw functions to combine MSFs on different monads, all of which conform to one of the following patterns:

- Lifting a purer MSF into an MSF defined over an impurer monad (for example, $MSF\ Identity\ a\ b \to MSF\ IO\ a\ b$).
- Running a more structured computation inside a less structured monad (for example, $MSF\ (t\ m)\ a\ b \to MSF\ m\ (I\ a)\ (F\ b)$).

This section explores these transformations and presents a way of thinking about MSFs over different monads and monad stacks.

4.3.1 Lifting Combinators

Whenever we use *monad transformers* or other effect stacking mechanism, we may be interested in embedding an $MSF\ m\ a\ b$ into a larger $MSF\ (t\ m)\ a\ b$ (where $t\ m$ denotes a monad that encloses the effects of m, such as a transformer t applied to m).

We can lift between any two arbitrary monads, provided we have a monad morphism $(Monad\ m, Monad\ n) \Rightarrow m\ a \to n\ a$.

$$liftLM :: (Monad\ m, Monad\ n) \Rightarrow (\forall\ a\ .\ m\ a \to n\ a) \to MSF\ m\ a\ b \to MSF\ n\ a\ b$$

Users are responsible for providing an actual monad morphism that retains any information in m inside the context in n. For Monad Transformers, with a monad morphism $lift :: m\ a \to t\ m\ a$, we define the convenience lifting function $liftST = liftLM\ lift$.

4.3.2 Temporal Execution Functions

Monad stacks are a common design pattern. If we introduce a transformer in the stack, we frequently want to remove it after performing some effects in it, by executing it. An example is the "running function" $runReaderT :: ReaderT\ r\ m\ a \to r \to m\ a$.

In Section 4.2, we introduced temporal execution functions that remove a transformer from the monad stack *inside* the MSF, such as $runReaderS :: MSF\ (ReaderT\ r\ m)\ a\ b \to r \to MSF\ m\ a\ b$. Similar examples were given for $WriterT$, $MaybeT$, $ListT$ and $StateT$. Essentially, they are implemented by commuting the transformer past MSF and then applying the running function of the transformer. This will be explained in the following.

Monadic Streams and MSFs To simplify the implementation of the temporal running functions, we are going to exploit an isomorphism between Monadic Stream Functions and certain Monadic Streams. Internally, MSFs are defined as follows:

$$\textbf{newtype}\ \mathit{MSF}\ m\ a\ b = \mathit{MSF}\ \{\mathit{step} :: a \to m\ (b, \mathit{MSF}\ m\ a\ b)\}$$

Depending on the monad m, we may have one, none, or several outputs and continuations. Moving from a monad like *Maybe* or *Either c* to another monad requires retaining the possibility of providing no output, or recovering from a *termination* or *exception*.

In turn, Monadic Streams can be defined as:

$$\mathit{Stream}\ m\ a\ \cong\ m\ (a, \mathit{Stream}\ m\ a)$$

It is easy to see that MSFs are isomorphic to Monadic Streams in a *Reader* context:

$$\mathit{MSF}\ m\ a\ b\ \cong\ \mathit{Stream}\ (\mathit{ReaderT}\ a\ m)\ b$$

The existing research on streams, monadic streams and causal stream functions makes establishing this isomorphism useful in its own right [90].

Note that *Stream* is a transformer, with $\mathit{lift} :: m\ a \to \mathit{Stream}\ m\ a$ given by the infinite repetition of the same effect. Because $\mathit{MSF}\ m\ a\ \cong\ \mathit{Stream}\ (\mathit{ReaderT}\ a\ m)$, this makes *MSF* with the second argument a pre-applied also a transformer.

These monadic *Stream*s are *Functor*s and *Applicative*s, but not necessarily *Monad*s. As transformers they can still be applied and, with some constraints, commuted, so the lack of a general monad instance does not invalidate our argument.

Implementing Temporal Execution Functions Commuting a transformer t past *MSF* can be done in two steps, by commuting t past *ReaderT* and subsequently past *Stream*. We can capture all this in a type class that contains the definitions we need to implement a function that transforms an $\mathit{MSF}\ (t\ m)\ a\ b$ into a $t\ (\mathit{MSF}\ m\ a\ b)$:

$$\begin{aligned}&\textbf{class}\ \mathit{MonadTrans}\ t \Rightarrow \mathit{Commutation}\ t\ \textbf{where}\\ &\quad \mathit{commuteReader} :: \mathit{Monad}\ m \Rightarrow \mathit{ReaderT}\ r\ (t\ m)\ a \to t\ (\mathit{ReaderT}\ r\ m)\ a\\ &\quad \mathit{commuteStream} :: \mathit{Monad}\ m \Rightarrow \mathit{Stream}\ (t\ m)\ a \quad \to t\ (\mathit{Stream}\ m)\ a\\ &\quad \mathit{preserveMH} \quad :: (\mathit{Monad}\ m1, \mathit{Monad}\ m2) \Rightarrow (\forall\ a\ .\ m1\ a \to m2\ a) \to t\ m1\ b \to t\ m2\ b\end{aligned}$$

This type class gives rise to a commutation function, that commutes the t transformer past *MSF* (past the *Reader* and past the *Stream*):

$$\mathit{commute} :: \mathit{Commutation}\ t \Rightarrow \mathit{MSF}\ (t\ m)\ a\ b \to t\ (\mathit{MSF}\ m\ a)\ b$$

As a technicality, we need to assume that the transformer t preserves the isomorphism $MSF\ m\ a \cong Stream\ (ReaderT\ a\ m)$. This is because, to implement *commute*, we need to turn an *MSF* into a *Stream* on a $ReaderT$, commute t past the $ReaderT$, commute past the *Stream*, and then turn the continuation (a *Stream* on a $ReaderT\ r\ m$ monad), back into an *MSF*.

Many transformers, like $ReaderT$, $StateT$ and $MaybeT$, preserve monad homomorphisms[2]. There is no requirement that *commute* be an isomorphism, and, in cases like $ListT$ and $MaybeT$, a change of effects is actually desired.

Defining a temporal execution function is now as simple as defining an instance of the *Commutation* type class and composing *commute* with any running function. For example:

$$
\begin{aligned}
&runReaderS \equiv runReaderT \circ commute \\
&runWriterS \equiv runWriterT \circ commute \\
&runStateS \equiv runStateT \circ commute \\
&catchM\ msf\ handler \equiv fromMaybe <\$> handler <*> runMaybeT\ (commute\ msf)
\end{aligned}
$$

Transformers commuting with $ReaderT$ are abundant, and the implementation of the abstraction proposed in this chapter defines *Commutation* instances for common transformers.

Running with a Changing Input Other execution functions are also useful in practice. For example, we may be interested in obtaining a *Reader* context at every input sample, requiring:

$$runReaderS\_ :: MSF\ (ReaderT\ r\ m)\ a\ b \to MSF\ m\ (r, a)\ b$$

This kind of definition is trivial once we observe that the *Reader* monad commutes with itself, and that the following are isomorphic:

$$ReaderT\ r1\ (ReaderT\ r2\ m)\ a \equiv ReaderT\ (r1, r2)\ m\ a$$

and therefore:

$$
\begin{aligned}
&MSF\ (ReaderT\ r\ m)\ a\ b \\
&\equiv \{\ \textit{MSFs as Streams}\ \} \\
&\quad Stream\ (ReaderT\ a\ (ReaderT\ r\ m))\ b \\
&\equiv \{\ \textit{Commutativity of reader}\ \} \\
&\quad Stream\ (ReaderT\ r\ (ReaderT\ a\ m))\ b \\
&\equiv \{\ \textit{Reader r1 (Reader r2 a)} \equiv \textit{Reader (r1, r2) a}\ \} \\
&\quad Stream\ (ReaderT\ (r, a)\ m)\ b \\
&\equiv \{\ \textit{Streams as MSFs}\ \} \\
&\quad MSF\ m\ (r, a)\ b
\end{aligned}
$$

[2] A more exhaustive list of which ones do and do not preserve homomorphisms remains as future work.

4.4 Summary

In this chapter we explored Monadic Stream Functions (MSFs), a mathematical abstraction for causal stream functions in a monadic context. We saw that, when combined with different monads, they give rise to useful features existing in other FRP variants, like switching, logging and parallelism with broadcasting.

We also saw that MSFs have interesting mathematical properties, and that these sometimes can be established simply by examining the properties of the monad they are applied to.

Furthermore, because the monad is explicit in the type, we can always ensure purity of the implementation just by looking at the type of an *MSF*. This allows us to extend Monadic Stream Functions introducing controlled side effects. In Part III we explore constraints that make MSFs robust, and facilitate testing and debugging.

In the following Chapters 5, 6 and 7 we shall see that Monadic Stream Functions subsume existing FRP variants, and can be used to implement different forms of Reactive and Functional Reactive Programming.

Chapter 5

Extensible Functional Reactive Programming

In this chapter we explore how Monadic Stream Functions, introduced in the previous chapter, can be used to implement Reactive Programming and Functional Reactive Programming. As MSFs are polymorphic in the monad, the versions of RP and FRP that we implement are more versatile and extensible that their original counterparts.

In Section 5.1 we present a form of Reactive Programming based on a notion of Streams and Sinks built on top of Monadic Stream Functions. In Section 5.2 we extend this abstraction with time, thus implementing Classic FRP. In Section 5.3 we follow a similar approach to define an extensible form of Arrowized FRP. In Section 5.4 we demonstrate that this form of Arrowized FRP based on Monadic Stream Functions not only subsumes existing variants, but also makes it possible to add new features useful in real games. This discourse is limited to the programming abstractions; performance and comparison with other FRP variants are covered in Chapters 6 and 11.

5.1 Reactive Programming and Monadic Streams

Reactive Programming [91] is a programming paradigm organised around information producers and consumers or, respectively, *streams* and *sinks*, combined and connected to one another via a series of transformations [92, 93].

5.1.1 Streams

Streams model the concept of information producers. We can capture that idea with Monadic Stream Functions that do not depend on their input, as shown in Section 4.3.2, which we recall below:

$$\textbf{type } \mathit{Stream}\ m\ b = \mathit{MSF}\ m\ ()\ b$$

These streams can depend on their monadic context to produce information. This is made explicit in the type.

It is easy to see what this definition means by applying unit as the only possible argument to the function in the definition of MSF (if we disregard bottoms):

$$\begin{aligned}
\mathit{Stream}\ m\ b &\cong \mathit{MSF}\ m\ ()\ b \\
&\cong \{\, \mathit{definition\ of\ MSF} \,\} \\
&\quad () \to m\ (b, \mathit{Stream}\ m\ b) \\
&\cong \{\, \mathit{applying\ unit} \,\} \\
&\quad m\ (b, \mathit{Stream}\ m\ b)
\end{aligned}$$

Monadic streams applied to the *Identity* monad are isomorphic to coinductive infinite streams, which we can easily illustrate by unfolding the definition several steps[1]:

$$\begin{aligned}
\mathit{Stream\ Identity}\ b &\cong \{\, \mathit{definition\ of\ Stream} \,\} \\
&\quad \mathit{Identity}\ (b, \mathit{Stream\ Identity}\ b) \\
&\cong \{\, \mathit{definition\ of\ Identity} \,\} \\
&\quad (b, \mathit{Stream\ Identity}\ b) \\
&\cong \{\, \ldots \mathit{repeat\ for\ second\ element\ in\ pair} \ldots \,\} \\
&\quad (b, (b, \mathit{Stream\ Identity}\ b)) \\
&\cong (b, (b, (b, \ldots)))
\end{aligned}$$

There is an isomorphism between monadic streams and Monadic Stream Functions. As stated before (Section 4.3), the *Reader* monad can be used to define Monadic Stream Functions from Monadic Streams.

Example We can use this definition of *Stream* to represent, for instance, a stream of all the natural numbers starting from zero, using the function *feedback* defined in Section 4.1.5:

$$\begin{aligned}
&\mathit{naturalNumbers} :: \mathit{Stream}\ m\ \mathit{Int} \\
&\mathit{naturalNumbers} = \mathit{feedback}\ 0\ (\mathit{arr}\ (\mathit{dup} \circ (+1) \circ \mathit{snd})) \\
&\quad \textbf{where}
\end{aligned}$$

[1] See Appendix A for a proof.

```
    dup :: a → (a, a)
    dup a = (a, a)
  -- Defined elsewhere
feedback :: c → MSF m (a, c) (b, c) → MSF m a b
```

Recall that the function *feedback* creates a well-formed loopback by feeding the second component of the last output as the second component of the next input, and having an initial value for that accumulator. In the case above, the initial value in the accumulator is 0, to which we add one and duplicate it, making it both the output of the *Stream* and the next accumulator.

The previous *Stream* is fully polymorphic in the monad. In some cases, however, the stream may be created from a specific context. For instance, we can use *Streams* to obtain lines from standard input, which requires that the monad be *IO*:

```
lineStream :: Stream IO Int
lineStream = liftS (λ_ → getLine)
```

5.1.2 Sinks

Just like streams can be seen as MSFs that do not depend on their inputs, *sinks* can be seen as MSFs that do not produce any output:

```
type Sink m b = MSF m b ()
```

These represent information consumers or dead-ends of information flow. They are useful when only the side effects are of interest. Sinks are present in reactive implementations like reactive banana [84], Wormholes [94], and Reactive Values [29], the last of which is presented in the next chapter.

Example Information consumers can be used to produce side-effects. For example, the following sink prints lines to standard output:

```
printSink :: Sink IO String
printSink = liftS putStrLn
```

Streams and sinks are MSFs, so we can use *MSF* combinators to transform them and connect them. The following reactive program chains a stream and a sink to reverse input lines:

```
revInput :: MSF IO () ()
revInput = lineStream ⋙ arr reverse ⋙ printSink
```

Note that this *MSF* would block if the function *getLine*, used to implement *lineStream*, blocks in absence of input.

Syntax Monadic Streams as defined above are *Functors* and *Applicatives*[2]:

```
instance Monad m ⇒ Functor (Stream m) where
  fmap f  = (⋙ arr f)

instance Monad m ⇒ Applicative (Stream m) where
  pure x  = arr (const x)
  f <*> x = (f &&& x) ⋙ arr (uncurry ($))
```

Sinks, in turn, are *contravariant functors*:

```
instance Contravariant (Sink m) where
  contramap :: (a → b) → Sink m b → Sink m a
  contramap f msf = arr f ⋙ msf
```

Providing a *Monad* instance for monadic streams has proved difficult, and further research is needed. Defining a monad instance for monadic streams would require defining an operation $join :: Stream\ m\ (Stream\ m\ a) \rightarrow Stream\ m\ a$, which takes a stream of streams and flattens the values, producing only one stream. Some type-correct implementations of *join* do not satisfy the *monad laws*, such as, for instance, taking just the first stream in the stream of streams.

For pure streams, that is, when the monad is the identity, a well-understood definition is taking the diagonal: the first sample from the first stream, the second sample from the second stream, and so on. While this implementation is type correct, it requires applying the tail function to the n-th stream n times in order to obtain the n-sample, which, depending on the implementation, may present $O(n)$ computational complexity on the index of the desired stream element [78].

The difficulty for the case of *monadic streams* is possibly greater, as obtaining the n-sample of the n-stream requires discarding the first $n - 1$ samples, for which the monadic side effects of obtaining and discarding those samples, in order to access the tail of the stream that provides the next sample, need to be computed. In other words: calculating the n-th sample of the stream of streams will have the side effects of calculating the first sample of the first stream, the first two samples of the second stream, the first three samples of the third stream, and so on. In the case of Monadic Stream Functions as defined in this text, this additional execution of side effects would increase computational complexity and might produce undesired side effects.

Examples The previous instances allow us to write more declarative code. For example, given a stream $mouseX :: Stream\ IO\ Int$, representing the changing X coordinate of the mouse position,

[2]These instances use a *Monad* constraint for readability, as it results in simpler and shorter type constraints; the real implementation only needs *Functor* and *Applicative* constraints on m. For the same reasons, this text defines the instances in terms of the type synonym *Stream*. The actual implementation defines them for the more general type $MSF\ m\ a$, parameterised over any input type.

we can use the following applicative syntax to define the mouse position with respect to the right margin of the screen (for a fixed resolution of 1024 pixels):

```
mirroredMouseX :: Stream IO Int
mirroredMouseX = (1024−) <$> mouseX
```

We can sometimes simplify code further. For example, using the previous instances we can give a *Num* instance for *Num*-carrying *Stream*s:

```
instance (Monad m, Num b) ⇒ Num (Stream m b) where
  f + g           = (+) <$> f <*> g
  f * g           = (*) <$> f <*> g
  abs             = fmap abs
  signum          = fmap signum
  fromInteger x   = pure (fromInteger a)
  negate          = fmap negate
```

Now it is possible to overload numeric operators and write *mirroredMouseX* simply as:

```
mirroredMouseX :: Stream IO Int
mirroredMouseX = 1024 − mouseX
```

In this expression, 1024 has type *Stream IO Int* and it is the stream that constantly produces the value 1024.

5.2 Classic FRP and Monadic Stream Functions

Functional Reactive Programming (FRP), introduced in Chapter 3, is a paradigm to describe systems that change over time. Time in FRP is explicit, *conceptually continuous*, and non-negative. Systems are described by defining and combining *signals*, which represent time-varying values:

```
type Signal a ≃ Time → a
type Time     ≃ ℝ+
```

Signals normally represent internal or external information sources, like, for example, the varying mouse position, or the position of a dot moving around in circles with a one-second period.

While this conceptual definition enables giving FRP denotational semantics, execution is still carried out by sampling signals progressively. The ideal semantics is approximated more precisely as the sampling frequency increases [95].

This sampling-based execution is, in principle, very similar to our previous definition of streams, augmented with time information. Using Monadic Stream Functions we can add time to the monadic context. This can be achieved using the *Reader* monad, which can be defined as:

$$\textbf{type}\ \mathit{Reader}\ e\ a\ \simeq\ e \to a$$

To ensure causality at the level of FRP signals, time must always progress towards the future. We could guarantee, externally, that streams are executed with increasing time deltas for each cycle. Instead, like Yampa [26, 27], we use a type *DTime* to represent the strictly positive time passed between consecutive samples, which is provided in a *Reader* monad environment:

$$\textbf{type}\ \mathit{Signal}\ m\ a = \mathit{Stream}\ (\mathit{ReaderT}\ \mathit{DTime}\ m)\ a$$

This represents a signal that can depend on its past, as it provides a continuation to be used for future samples. By limiting the API and, in particular, the possibility of sampling other signals at arbitrary times, we can prevent signals from depending on other signals at future times, enforcing causality at a system level.

We can use the monad to extend this definition with additional side effects. To add external information sources we nest the monad transformer *ReaderT* with the *IO* monad, obtaining:

$$\textbf{type}\ \mathit{SignalIO}\ a = \mathit{Stream}\ (\mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO})\ a$$

We can visualise the dependency on the time delta, the presence of side effects, and the explicit continuation by expanding the definition:

$$\begin{aligned}
\textbf{type}\ \mathit{SignalIO}\ a\ &\cong\ \{\mathit{Definition\ of\ SignalIO}\}\\
&\quad\ \mathit{Stream}\ (\mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO})\ a\\
&\cong\ \{\mathit{Definition\ of\ Stream}\}\\
&\quad\ \mathit{MSF}\ (\mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO})\ ()\ a\\
&\cong\ \{\mathit{Definition\ of\ MSF}\}\\
&\quad\ () \to \mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO}\ (a, \mathit{MSF}\ (\mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO})\ ()\ a)\\
&\cong\ \{\mathit{Definition\ of\ Stream}\}\\
&\quad\ () \to \mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO}\ (a, \mathit{Stream}\ (\mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO})\ a)\\
&\cong\ \{\mathit{Definition\ of\ SignalIO}\}\\
&\quad\ () \to \mathit{ReaderT}\ \mathit{DTime}\ \mathit{IO}\ (a, \mathit{SignalIO}\ a)\\
&\cong\ \{\mathit{Definition\ of\ ReaderT}\}\\
&\quad\ () \to \mathit{DTime} \to \mathit{IO}\ (a, \mathit{SignalIO}\ a)\\
&\cong\ \{\mathit{Applying}\ ()\}\\
&\quad\ \mathit{DTime} \to \mathit{IO}\ (a, \mathit{SignalIO}\ a)
\end{aligned}$$

Example A simple simulation of a ball moving around the mouse position could be written in Classic FRP style as follows:

```
ballInCirclesAroundMouse :: SignalIO (Int, Int)
ballInCirclesAroundMouse = addPair <$> mousePos <*> ballInCircles

ballInCircles :: SignalIO (Double, Double)
ballInCircles = (λx → (rad * cos x, rad * sin x)) <$> time
  where
    rad = 45  -- radius in pixels

mousePos    :: SignalIO (Int, Int)
mousePos    = liftS (λ() → lift getMousePos)

  -- Predefined
addPair     :: Num a ⇒ (a, a) → (a, a) → (a, a)
getMousePos :: IO (Double, Double)
time        :: SignalIO Time
time        = liftS ask ⋙ sumFrom 0

  -- Defined in previous chapters
liftS       :: Monad m ⇒ m a → MSF m () a
sumFrom     :: (Monad m, Num a) ⇒ MSF m a a
```

With non-commutative monads, like *IO*, additional measures must be taken to ensure referential transparency at the same conceptual time. If several signals depend on, for example, *mousePos*, the mouse position might actually be sampled twice, producing different results at the same conceptual time. This can be addressed with a monad that caches results and enables garbage collection [78].

5.3 Arrowized FRP and Monadic Stream Functions

Arrowized FRP (AFRP) is an FRP formulation structured around the concept of *signal functions*. Unlike in Classic FRP, in AFRP signals are not first-class citizens.

Conceptually, we can define signal functions, or *SF*s, as follows:

```
type SF a b ≃ Signal a → Signal b
```

Signals in FRP must be causal. Consequently, signal functions are required to maintain causality of signals: the value of an output signal at a given time cannot depend on the value of the any signal at a future time.

Multiple AFRP implementations exist. In the following we see how to implement the core of Yampa [26, 27], used to program multimedia and games. In Section 5.4 we see that this definition

based on MSFs addresses some limitations of pure Arrowized FRP, and we implement systems that combine time domains, clocking speeds, and a limited form of Continuous Collision Detection.

Core Definitions

To make signal functions (SFs) executable and causal, Yampa uses the same approach that we used with Classic FRP: signal functions are executed by sampling them at increasing times, using strictly positive time deltas, or times between samples. Yampa's causal SFs are defined as follows[3]:

$$\textbf{type } SF'\ a\ b = DTime \to a \to (b, SF'\ a\ b)$$

We can realise this type using MSFs by passing time deltas in a *Reader* monad environment, just like we did with Classic FRP. Unlike in previous sections, we do not allow for arbitrary side effects in the monad:

$$\textbf{type } SF\ a\ b = MSF\ (Reader\ DTime)\ a\ b$$
$$\textbf{type } DTime = Double$$

Most of Yampa's core primitives [96], like *arr*, $(\ggg)$ or *switch*, are time-invariant, and the definitions in the previous chapter implement the same behaviour as Yampa. We only add:

```
integral :: VectorSpace a s ⇒ SF a a
integral = eulerSteps ⋙ sumFrom zeroVector
  where
    eulerSteps = liftS $ λx → do
      timeDelta ← ask
      return $ x ^* timeDelta
```

This function implements a simple approximation of the integral using the Euler method for numerical integration [97]. The signal function *eulerSteps* multiplies the input by the current time delta. The auxiliary function $ask :: Reader\ e\ e$ returns the environment in the *Reader* monad, and $(\hat{}*)$ represents multiplication of a vector by a scalar. The MSF $sumFrom :: Num\ a \Rightarrow MSF\ m\ a\ a$ sums its inputs and produces the accumulated result at each step. The value *zeroVector* represents a vector in which all coordinates are zero.

We can use this definition of integral to define another time-varying core Signal Function, *time*, in terms of it: $time = integral\ 1$.

[3] Yampa defines a type SF for signal functions at the initial sampling time 0, and a type SF' for signal functions that are already "turned on". Here we obviate that matter and discuss only an equivalent for SF'.

Example To facilitate understanding, let us demonstrate how systems can be defined in this variant of pure FRP. We can define the position and velocity of a falling mass as follows:

```
fallingBall :: Double → Double → SF () (Double, Double)
fallingBall p0 v0 = proc () → do
  v ← arr (v0+) ⋘ integral ⤙ (−9.8)   -- m/s
  p ← arr (p0+) ⋘ integral ⤙ v
  returnA ⤙ (p, v)
```

For simplicity, we focus only on the vertical axis. We use Paterson's Arrow notation [37] to facilitate reading. This definition is time-dependent and input-independent. We now define a signal function that outputs the position of a ball moving in circles around an input position:

```
circlingBall :: SF (Double, Double) (Double, Double)
circlingBall = proc (baseX, baseY) → do
  t ← time ⤙ ()
  let radius = 100   -- pixels
      x      = baseX + radius * cos t
      y      = baseY + radius * sin t
  returnA ⤙ (x, y)
```

This definition is both valid Yampa and valid in our MSF-based Yampa replacement.

Reactimating Signal Functions

The second part of our Yampa replacement is a *reactimation* or simulation function. The signature of Yampa's top-level simulation function[4] demonstrates the bottleneck effect mentioned in Chapter 3:

```
reactimate :: IO (DTime, a) → (b → IO ()) → SF a b → IO ()
```

The first argument produces inputs and time deltas, the second consumes outputs and produces side effects. We can implement this function as follows:

```
reactimate sense actuate sf = MSF.reactimate (senseSF ⋙ sfIO ⋙ actuateSF)
  where
    sfIO     :: MSF IO a b
    sfIO     = liftLM (return ∘ runIdentity) (runReaderS_ sf)

    senseSF  :: MSF IO () (DTime, a)
```

[4] Yampa's *reactimate* has a more complex signature for reasons beyond the scope of this dissertation that do not invalidate these claims. The implementation published in the library Dunai/BearRiver follows Yampa's specification.

$$senseSF \quad = liftS\ (\lambda() \rightarrow sense)$$
$$actuateSF :: MSF\ IO\ b\ ()$$
$$actuateSF = liftS\ actuate$$

The function *MSF.reactimate* used inside the implementation of *reactimate* has a simpler type signature, $MSF\ m\ ()\ () \rightarrow m\ ()$. This indicates that all I/O is done inside the Monadic Stream Function provided as argument, and that all we care about, in the end, are the side effects.

There are two notable aspects of *sfIO*. First, the auxiliary temporal execution function $runReaderS\_ :: MSF\ (ReaderT\ s\ m)\ a\ b \rightarrow MSF\ m\ (s, a)\ b$, defined in Section 4.2, provides the time deltas in a *Reader* environment for the Yampa monad. Second, after extracting the *ReaderT* layer, we use *liftLM*, defined in Section 4.3.1, to lift a computation in the *Identity* monad into the *IO* monad. At the top level, IO effects are consumed and presented to the user progressively.

The implementation of AFRP using MSFs has been validated with multiple Yampa games. This is discussed in Chapter 6.

5.4 Extensions to Arrowized FRP

The definition of Signal Functions introduced above fixes the time domain and a monad to conform to traditional AFRP constructs. The API presented so far prevents us from transforming the time deltas within SF definitions, effectively fixing the sampling step for the complete system. Yampa's complete API includes two functions that enable limited time manipulations: *embed*, which allows one to apply a limited stream to a signal function, obtaining a list of results but resetting the signal function at every step, and *embedSynch*, which allows for a limited form of synchronous embedding with fixed time deltas. None of these mechanisms are as versatile as what MSFs can offer.

The implementation of signal functions using MSFs could, in principle, use a different monad or time domain, or even expose them in the type constructors. This would make this FRP definition more versatile and better suited for other kinds of applications.

This section demonstrates the extensibility of MSF-based FRP by introducing features to produce controlled side effects, manipulate time and customise the clocking or sampling frequency.

5.4.1 Monadic FRP Extensions

The definition of Signal Functions given so far fixes the monad to the *Reader* monad, preventing additional side effects and making it rather limited:

type $SF\ a\ b = MSF\ (Reader\ DTime)\ a\ b$

The evaluation function that runs the main signal function is also the only one that can connect input and output signals to the rest of the system. All data that an internal signal function may need must be manually routed down, and its outputs manually routed up. This helps keep Signal Functions pure (free from any I/O) and referentially transparent *across executions*, but at the expense of having to poll all input data every time and handle more complex data structures in the output.

A manifestation of this problem is that it is not possible to debug from within signal functions, except by adding explicit output signals carrying debugging information, or by using functions like *Debug.Trace.trace*, which uses *unsafePerformIO* to print to standard output. Code that uses *trace* is not portable, for instance, to platforms like Android, as debug messages must be printed to a special debug log facility.

We can overcome these limitations by exposing the monad in the type of signal functions, provided that certain constraints are met. A more versatile signal function definition would be the following:

$$\textbf{type}\ ESF\ m\ a\ b = MSF\ (ReaderT\ DTime\ m)\ a\ b$$

In principle, these Effectful Signal Functions, or ESFs, would allow for arbitrary side-effects, which hinders the benefits of pure AFRP. We can constrain the monad with a type-class constraint to minimise the adverse effects of introducing this extensibility.

Examples A convenient use of monad-polymorphism of *ESF*s is to provide access to system properties that seldom or never change, like the screen size, using a *Reader* monad.

To make the screen size available to a signal function in traditional FRP we could pass it as an input signal or an argument to a function that produces a signal function:

$$\begin{aligned}&posFromBottomMargin :: SF\ ((Double, Double), (Double, Double))\ Double\\&posFromBottomMargin = arr\ \$\ \lambda((width, height), (x, y)) \rightarrow height - y\end{aligned}$$

or, alternatively:

$$\begin{aligned}&posFromBottomMargin1 :: (Double, Double) \rightarrow SF\ (Double, Double)\ Double\\&posFromBottomMargin1\ (width, height) = arr\ \$\ \lambda(x, y) \rightarrow height - y\end{aligned}$$

If we now have a signal function that uses either of these versions, that signal function needs to manually pass that data down to make it available:

$$\begin{aligned}&game :: SF\ ((Double, Double), GameInput)\ Double\\&game = \textbf{proc}\ (screenSize, gi) \rightarrow \textbf{do}\end{aligned}$$

```
    mouse   ← mousePos                 -≺ gi
    objectY ← posFromBottomMargin -≺ (screenSize, mouse)
    ...
    -- Defined elsewhere
mousePos = ...
type GameInput = ...
```

This affects both the types and the implementation of *game*. By using the *Reader* monad, we can make the "piping" implicit:

```
type GameMonad = Reader (Double, Double)   -- resolution
game :: ESF GameMonad GameInput Double
game = proc (gi) → do
    mouse   ← mousePos                 -≺ gi
    objectY ← posFromBottomMargin -≺ mouse
    ...
posFromBottomMargin :: ESF GameMonad (Double, Double) Double
posFromBottomMargin = proc (x, y) → do
    h       ← screenHeight -≺ ()
    returnA -≺ (h - y)
screenHeight :: ESF GameMonad () Double
screenHeight = liftS (asks snd)
```

While the code of *posFromBottomMargin* becomes slightly more complex, every function that uses *posFromBottomMargin*, directly or indirectly, becomes simpler.

Another useful example is the possibility of adding logging facilities to signal functions. This case is particularly interesting because logs are not commutative monoids, making the monad not commutative. However, because the *Writer* log is not accessible and cannot be used to affect signals, these effectful signal functions are still signal functions, or causal functions between signals, and therefore referentially transparent:

```
type GameMonad = Writer [String]   -- resolution
game :: ESF GameMonad GameInput Double
game = proc (gi) → do
    mouse ← mousePos -≺ gi
    ()    ← liftS (put ("Current mouse coordinates are " ++ show mouse)) -≺ ()
    ...
```

These two very useful examples, based on similar descriptions presented in Section 4.2, show

that using MSFs to describe FRP constructs allows us to easily extend its capabilities.

The implementation of the MSF library Dunai[5] includes a Yampa replacement library, called BearRiver[6], built this way. BearRiver's SF replacement for Yampa's main SF is just a monad-polymorphic *SF*, like *ESF* above, instantiated with the *Identity* monad.

5.4.2 Time Transformations

The nature and progress of time is central to FRP. In most FRP implementations, the time domain is the same for the whole system, time always advances towards the future, and synchronicity or asynchronicity is global to the framework. For instance, in Yampa [27, 40], access to other signals is limited to the present[7], time has floating-point precision and the whole system advances in unison on a synchronous clock. In other FRP variants, time is hidden, and different subsystems run asynchronously, being recalculated as the values that they depend on change [98].

Time Dilation

In the version of pure Arrowized FRP presented so far, the clock is controlled externally and globally for the whole simulation. All signal functions progress with the same sampling step. For example, let us define a signal function that represents the position of a ball going around in circles:

```
circlingBall :: SF a (Double, Double)
circlingBall = proc (_) → do
  t ← time ⤙ ()
  let radius = 100  -- pixels
      x      = radius * cos t
      y      = radius * sin t
  returnA ⤙ (x, y)
```

No mechanism seen so far allows us to run *circlingBall* faster or slower than it normally would without affecting other signal functions. If we wanted to show two balls, one moving twice as fast as the other, we would need to modify the implementation of *circlingBall*.

We can make a signal function progress at a different speed by directly transforming those time deltas, or time differences between consecutive samples. If the transformation does not depend on the past state or the global time, then we can define:

```
timeTransform    :: (DTime → DTime) → SF a b → SF a b
timeTransform f = liftLM (withReaderT f)
```

[5] `http://hackage.haskell.org/package/dunai`

[6] `http://hackage.haskell.org/package/bearriver`

[7] Except for the limited combinator *embedSynch*, mentioned before.

The function $liftLM :: (\forall\ a\ .\ m\ a \to m'\ a) \to MSF\ m\ a\ b \to MSF\ m'\ a\ b$ lets us apply a monad transformation to an *MSF*, and the function from the *ReaderT* monad transformer $withReaderT :: (e \to e1) \to ReaderT\ m\ e\ a \to ReaderT\ m\ e1\ a$ modifies the environment in a *ReaderT*.

Note that this does not make the argument signal function "tick" or be sampled more or less frequently than others, but simply makes it "believe" that more or less time has passed between samples. These signal functions are still synchronous.

Example Given *circlingBall* defined before, we run two such animations in parallel, one twice as fast as the other, as follows:

$$twoBalls :: SF\ a\ ((Double, Double), (Double, Double))$$
$$twoBalls = circlingBall\ \&\&\&\ timeTransform\ (*2)\ circlingBall$$

We can also make signal functions tick with a fixed step, irrespective of the actual sampling time (e.g. $timeTransform\ (const\ (1\ /\ 60))$), or with a minimum or maximum time step (e.g. $timeTransform\ (max\ (1\ /\ 120))$.

In all of these cases, we are assuming that time transformation functions produce positive time deltas. The FRP constructs presented so far would not be referentially transparent, or *temporally consistent*, if time could halt or flow backwards. There are mechanisms to extend the constructs given so far and enable travelling back in time under certain circumstances. The analysis of signal functions that behave consistently when time is reversed has been the briefly studied in [32].

State-dependent Time Transformations

The previous transformation is independent of the input and of previous history. One might want the time transformation to be applied only in some circumstances, for instance, when the player collects a special "power-up" or presses a certain key. This feature is common in games; a notable example is Max Payne's *Bullet Time*[8], which gives the player the ability to slow time down and aim weapons with more precision.

To facilitate this task, we could define a more general combinator:

$$timeTransformSF :: SF\ a\ (DTime \to DTime) \to SF\ a\ b \to SF\ a\ b$$

At every point in time, the first signal function produces a time transformation operating on the time deltas that can depend on the global time or the history of the system. This transformation is applied to the current time delta, which is then used to progress the second signal function.

[8] `http://www.rockstargames.com/maxpayne/main.html`

The function *timeTransform* provided earlier in this section can be defined as $timeTransform\ f = timeTransformSF\ (arr\ (\lambda\_ \rightarrow f))$.

Example Given a user input controller with an action button to determine if time should run in slow-motion:

$$\textbf{data}\ Input = Input\ \{slow :: Bool\}$$

We adapt our example to use that feature as follows:

$$game :: SF\ Input\ ((Double, Double), (Double, Double))$$
$$game = timeTransformSF\ timeProgression\ twoBalls$$
$$timeProgression :: SF\ Input\ (DTime \rightarrow DTime)$$
$$timeProgression = arr\ (\lambda c \rightarrow \textbf{if}\ slow\ c\ \textbf{then}\ (0.1*)\ \textbf{else}\ id)$$

We can use *timeProgression* to make time deltas smaller upon user request. This example shows that time transformations are composable: *twoBalls*, which is affected by the time transformation in *game*, internally applies another time transformation to one of the balls.

In Chapter 6 we explore more sophisticated ways of using these transformations. The features presented so far in this chapter enable a wide range of transformations, but could also be too permissive. For instance, certain transformations, like the inverse ($\lambda t \rightarrow 1\ /\ t$), would make systems less accurate the more finely they are sampled, violating the expectation that they should converge as the sampling frequency increases. It might be possible to limit the transformations allowed using a data type that defines "well-behaved" time manipulations. Using zero as the time delta or making it negative would also be allowed by this implementation, although some SF combinators are not time-reversible: for example, *switch* forgets the original signal function being applied before the time when the switch takes place, and cannot go back beyond that time. It is possible to define a reversible switch, and there are mechanism to limit possible memory leaks due to keeping too much history in order to make systems time-reversible [32].

Time Transformations between Time Domains

FRP requires that time be continuous, which does not accommodate all kinds of games well. For example, many puzzle and board games are inherently discrete, so using the natural numbers as the time domain or simply having no time at all might be enough to define them and, potentially, more efficient. Other games may have compound and non-linear representations of time.

The definitions presented so far follow the convention that the implicit type *DTime* represents time deltas. In Yampa, *DTime* is a *Double* and is required to be positive.

One could use the generality of Monadic Stream Functions to include more sophisticated time domains, like branching time, or simpler time domains, like the natural numbers. The following combinator, a generalization of *timeTransformSF*, allows us to embed a stream function in one time domain in a larger system with a different time domain:

```
timeTransformMSF :: MSF (ReaderT t1 m) a (t1 → t2)
                  → MSF (ReaderT t2 m) a b
                  → MSF (ReaderT t1 m) a b
timeTransformMSF timeSF sf = MSF $ λa → do
  (f, timeSF') ← step timeSF a
  (b, sf')     ← step (withReader f (sf a))
  return (b, timeTransformMSF timeSF' sf')
```

The two other time transformation functions are now straightforward, using the type synonym $SF\ a\ b = MFS\ (Reader\ DTime)\ a\ b$:

```
timeTransform     :: (DTime → DTime) → SF a b → SF a b
timeTransform f   = timeTransformSF (constant f)
timeTransformSF   :: SF a (DTime → DTime) → SF a b → SF a b
timeTransformSF   = timeTransformMSF
```

Example A common use of *timeTransformMSF* would be to transform between the different types and units that multimedia frameworks like Simple Direct-media Layer (SDL) use to represent time. For instance, we can run a game in a framework that uses integer numbers for time (for instance, representing ticks or milliseconds since the beginning of the execution) and include a system with continuous time as follows:

```
game :: MSF (Reader Int) GameInput GameVisuals
game = discreteGame
     ⋙ timeTransformMSF (λx → (fromIntegral x) / 1000) animations
discreteGame :: MSF (Reader Int)      GameInput GameState
animations   :: MSF (Reader DTime) GameState GameVisuals
```

Care must be taken when using cross-domain time transformations. As pure functions from time to values, signals are expected to return the same value if applied twice to the same conceptual time. Because signal functions may keep memory, if we allow negative or zero time deltas, they might transform the same input signal differently for the same conceptual time. Thus, a requirement to maintain referential transparency at a signal level is that time deltas returned by the time

transformation function always be positive. In the previous example, we would need to adapt the time transformation to make the value not zero. For discrete numeric types, it may be possible to capture the constraint that time deltas be strictly positive in the type itself, with a Peano encoding of positive numbers. However, when dealing with floating-point numbers, this guarantee may escape what can be expressed efficiently using Haskell types.

5.4.3 Asynchronous FRP

The construct *timeTransformMSF* runs both the discrete game and the continuous animations synchronously, which is not always ideal.

Consider, for example, the case of having a discrete *board* game which is animated in continuous time (Figure 5.1). In such a setting, changes to the board are applied at a discrete point, but may take some time to animate and be presented to the user. The top row in that figure shows how users might see an animation for the left piece being moved to the right position (the animation would be continuous, and the piece travel progressively left to right). However, conceptually, positions are discrete and the piece cannot be between cells. Therefore, the piece can move in our application's conceptual model representation before the animation finishes (second row), or when the animation starts (third row). In both cases, what the user sees does not match the conceptual board state.

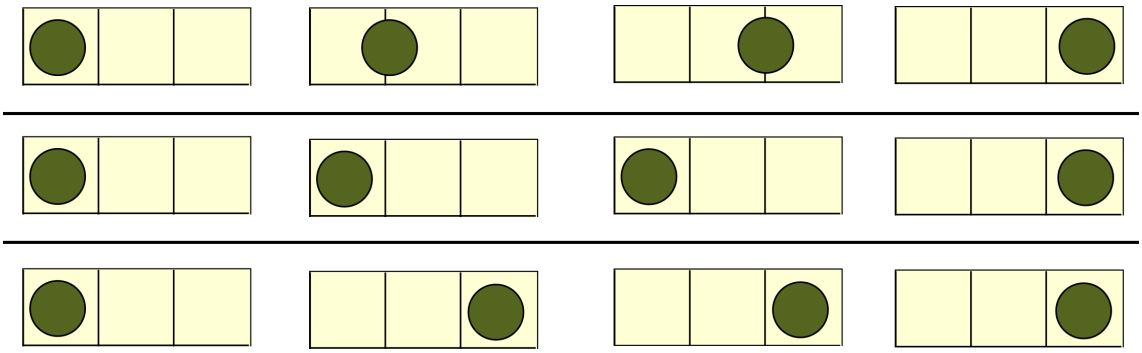


Figure 5.1: Illustration of board game with animations depicting moving the left piece to the right position. The top row shows the visualisation as users would experience it, the middle row shows the conceptual board game if changes are applied only after the animation is completed, and the bottom row shows the conceptual model if changes are applied immediately without waiting for the animation to complete. Time flows left to right.

Input events would be applied to a discrete game that reflects an old or a future state, which is not what the user is seeing at that time. If the user acts on the board during a transition period, the result may be counter-intuitive. This problem is common in GUI frameworks, and different

implementations opt for different design decisions. On Android, for instance, UI events are queued. Using a model-view abstraction, we can consider that events are applied on the conceptual model regardless of whether the view has "caught" up. The lack of visual feedback to indicate that the system is waiting or processing data in the background may generate frustration on the part of the user, whose actions will act on UI elements not present when the action took place. Windows, on the other hand, shows both visual feedback (hourglass mouse cursor) and discards events until the model and the view agree.

In this section we explore the coordination of signal functions running with different sampling frequencies. We explore this extension in two parts: *subsampling*, or the ability of a signal function to sample multiple times when the wrapping signal function was sampled only once, and *freezing*, or the ability of a signal function not to sample even when the wrapping signal function was sampled.

To simplify the discourse, we refer to these Effectful Signal Functions simply as signal functions, except when doing so may lead to confusion. It is important to remind that, if I/O side-effects are allowed, the resulting Effectful Signal Function may not be a pure or true Signal Function in the traditional sense.

Self-clocking Signal Functions

The function *twoBalls* seen in Section 5.4 runs two signal functions with the same precision: both "tick" in synchrony, even if one advances twice as much. We could, however, make one signal function sample more often than another:

```
twiceAsFast msf = MSF $ λa → do
  dt ← ask
  (_, msf1) ← runReaderT (step msf  a) (dt / 2)
  (b, msf2) ← runReaderT (step msf1 a) (dt / 2)
  return (b, twiceAsFast msf2)
```

The introduction of subsampling mechanisms like the latter needs to be addressed with care. Arbitrarily fine subsampling can lead to inherent memory leaks.

The previous construct is a special case of a more general construct that lets an extensible signal function control its own time step. The basic construct for local time control is a self-clocking signal function, defined as follows:

```
type SelfClockingMSF m a b = MSF (StateT (DTime, Bool) (ReaderT DTime m)) a b
```

This kind of signal function gets the time delta from a *Reader* context, and stores in the state the actual time delta used, and whether it could run again within the time left. We only require the

write capacity of $StateT$, but we avoid $WriterT$ due to its additional *Monoid* constraint. In terms of time we are giving this signal function self-timing abilities with the *State* monad, and we are extending the time domain with a *Bool* that indicates whether a simulation cycle has completed.

Example Imagine that we want to run a signal function with a maximum time delta of $16ms$ (60 times per second), possibly sampling multiple times within each simulation step. We could define our self-clocking *MSF* as follows:

```
game60FPS' :: MSF (Reader DTime) a b → SelfClockingMSF Identity a b
game60FPS' game = MSF $ λa → do
  dt ← lift $ ask
  let realDT  = min dt (1 / 60)
      lastStep = dt < (1 / 60)
  put (realDT, lastStep)
  (output, game') ← lift $ withReader (const realDT) (step game a)
  return (output, game60FPS' game')
```

Within each step, this *MSF* puts in the state the actual time delta used for the simulation, and whether it was the last simulation step.

As an example, if the actual delta is $44ms$, we should be able to run it twice within that time. The first iteration would write (16, *True*) in the state, indicating that there are iterations pending and that it could run again. A second iteration of the *MSF* would then write $(16, False)$ in the state, with *False* indicating that not enough time has passed to run a third time.

Executing self-clocking MSFs We define the following combinator that runs an *MSF*, possibly multiple times, accumulating the outputs at those times:

```
runSelfClocking :: SelfClockingMSF m a b → MSF (ReaderT DTime m) a [(DTime, b)]
```

A more convenient way of running a self-clocking function, by merging all outputs, is as follows:

```
resample :: (DTime → [(DTime, b)] → b)
         → SelfClockingMSF m a b
         → MSF (ReaderT DTime m) a b
```

The first argument to *resample* might return only the last sample, the average, or extrapolate it in some other way.

We now run a game with a minimum sampling rate as follows:

```
game60FPS :: MSF (Reader DTime) Input GameState
game60FPS = resample takeLast (game60FPS' realGame)
```

```
takeLast :: DTime → [(DTime, b)] → b
takeLast _ ss = snd $ last ss

  -- Defined elsewhere
realGame :: MSF (Reader DTime) Input GameState
```

The type signature of *realGame* does not change, showing that time control is truly local.

Continuous Collision Detection Physics simulations in Yampa are normally implemented by detecting overlaps between objects at each sampling time. This leads to *tunnelling* or *bullet-through-paper* effects, in which moving objects can pass through other objects if the simulation is not executed at a time when they actually overlap [99].

With MSFs we can implement some forms of Continuous Collision Detection (CCD) [99], by letting signal functions tell the top-level simulation when the next sampling should occur. The top-level *reactimate* can then decide whether the total time delta based on the actual CPU clock should be used for the next simulation step, or whether several steps and higher precision are needed at a particular stage.

This is a slight variation of self-clocking MSFs. In contrast, now the whole system is *synchronously* simulated with the smaller time delta, but an internal signal function can affect the global clock frequency.

Similarly to how we defined self-clocking signal functions, we can implement a form of CCD with a *Writer* monad. We use the monoid of time deltas with minimum as the associative binary operation and infinity as the identity element:

```
data FutureTime = AtTime DTime | Infinity
  deriving (Eq, Ord)
instance Monoid FutureTime where
  mempty  = Infinity
  mappend = min
```

Signal functions that try to change the *Writer* state with a requested sampling time will only do so when such time is smaller than one currently in the *Writer* state. At the end of each simulation iteration, the log will contain the closest suggested sampling time.

Using this approach, we can define a bouncing ball that never goes below the floor. We do so by calculating the expected time of impact ($nextT$) and by recording that time in the *Writer* context ($liftS \circ lift \circ tell$). For clarity, we use Paterson's Arrow notation [37]:

```
type CCDGameMonad m = ReaderT DTime (WriterT FutureTime m)
bouncingBall :: Double → Double → MSF CCDGameMonad () Double
```

```
bouncingBall p0 v0 = switch
  (proc () → do
    (p, v)    ← fallingBall p0 v0 -≺ ()
    bounce    ← edge                -≺ (p ⩽ 0 ∧ v < 0)
    let nextT = if v < 0 then sqrt (2 * p / gravity)   -- Down
                         else 2 * v0 / gravity         -- Up
    liftS (lift (tell nextT)) -≺ ()   -- Put "next" time
    returnA -≺ ((p, v), bounce `tag` (p, v)))
  (λ(p, v) → bouncingBall p (−v))
```

While, in this case, it is easy to calculate the time of impact locally, this problem becomes much harder in the presence of multiple objects, especially when accelerations are not constant, shapes are convex and/or rotations are allowed. It is worth noting that adjusting the time step based on how much a signal varies to increase the accuracy of the simulation is standard practice in simulation frameworks.

Freezing

The second asynchronous feature we explore is the possibility of sampling less often than the enclosing signal function. What this requires is, in principle, the possibility for a signal function to not change even when time has passed.

An ad hoc way to synchronise the two layers in traditional FRP can be achieved by making the visual layer output events that indicate when it has caught up with the conceptual changes. This, however, would require extending the type for input events to include visual events, and having the conceptual game be aware of and deal with data from the visual layer.

A rudimentary form of suspending a signal function could be achieved simply by caching the last output and outputting the same value untouched if the input, or the time passed, do not meet certain conditions. Implementing this behaviour using feedback is trivial.

We can define this problem in terms of the time domains of different signal functions and implement a general mechanism for discrete-continuous synchronization. This requires the introduction of an extension to make it possible to temporarily pause the game (i.e. allowing time deltas to be zero), and to make the visual layer determine the time delta of the conceptual layer.

First, we make the discrete part of the game run on a monad that indicates whether the game has stepped forward or not. In terms of time, we are extending the type of the time deltas, from being always exactly one, to being zero or one.

```
type DiscreteMonad m = ReaderT Bool m
```

Referential transparency in FRP would require that a zero time delta produce the same output as in the previous iteration. We can provide a mechanism to step a game forward only if the clock has a non-zero time delta (*True* in our *DiscreteMonad*), as follows:

```
possiblyRun :: b → MSF m a b → MSF (ReaderT Bool m) a b
possiblyRun b0 msf = MSF $ λa → do
  run ← ask
  if run then do (b, msf') ← lift (step msf a)
                 return (b, possiblyRun b msf')
         else return (b0, possiblyRun b0 msf)
```

This function "freezes" the internal signal function if the boolean in the *Reader* environment is *False*. When that happens, it returns the last known output or a default initial output.

Example Suspending a signal function lets us synchronise the conceptual and visual layers of an application by making the conceptual layer progress only when the visual layer is done animating previous changes.

We can now enclose the visual signal function in a similar wrapping, with the following monad:

```
type ContinuousMonad m = StateT Bool (ReaderT DTime m)
```

This lets the visual layer determine the time step that the conceptual layer should use, which corresponds to whether it is done animating the last change (*True*) or not (*False*).

If we now make the *Reader* monad of the discrete part (the conceptual board model), take the boolean from the state environment, and reset such state to *True* before every step of the evaluation of the visual layer, then we will effectively have synchronised both:

```
game :: MSF (ContinuousMonad m) Input VisualState
game = readerToState (possiblyRun defGameState game)
     ⋙ withSideEffect_ (put True)
     ⋙ visualLayer

  -- Auxiliary definitions
defGameState  :: GameState
game          :: MSF m Input GameState
visualLayer   :: MSF (ContinuousMonad m) GameState VisualState
readerToState :: MSF (Reader s m) a b → MSF (StateT s m) a b
withSideEffect_ :: m c → MSF m a a
```

The real-time animation detects changes to the discrete part and decides whether it should animate them and take some time to do so. When it does, it changes the state of the monad to *False*, indicating that it is trying to catch up, and so the rest of the game does not run.

More generally, this kind of synchronization, with one signal function deciding when to step another signal function and by how much, requires that the former writes in a context the time deltas of the latter, and the appropriate wiring functions to convert between contexts or monads.

While it is technically possible to achieve discrete-continuous synchronisation in a framework like Yampa using extra inputs/outputs to let each part indicate its internal progress and manually caching results for parts that do not need to progress, the resulting code stops being declarative, mixes model and visualisation aspects, and limits the benefits of using a purely functional language.

5.5 Summary

In this chapter we have seen that Monadic Stream Functions introduced in Chapter 4 can be used to implement different forms of Reactive Programming and Functional Reactive Programming.

The implementation of FRP using MSFs makes it possible to implement both Classic and Arrowized FRP, making it more versatile than other reactive programming variants. Furthermore, by exposing and constraining the type of the MSF using type classes we can make this variant of FRP more extensible and expressive, accommodating for controlled side effects.

We have also seen that, with minor modifications, the proposed framework can accommodate time transformations and different forms of local time control. Using additional combinators we can also introduce limited forms of asynchronous FRP.

Chapter 6 presents different games and game features implemented using the proposals in this and the previous chapter. Chapter 7 introduces a mid-level abstraction for reactive programming that encourages modularity and separation of concerns, which can be defined in terms of Monadic Stream Functions extended with a push-based *reactimate* function.

The proposed abstraction can expose the monad in signal functions at the type level, which can be used both to add and to limit possible side effects. Chapters 8 and 9 demonstrate that IO-free Signal Functions retain useful testing properties, and that Monadic Stream Functions provide additional debugging facilities over what traditional Arrowized FRP can provide.

Chapter 6

Evaluation

The ideas presented in Chapter 4 have been implemented in the library *Dunai*[1]. This library proposes a minimal core defining Monadic Stream Functions, the required functor, applicative functor and arrow instances, and a series of combinators and monadic-based *MSF* transformations.

Additionally, the library *BearRiver*[2] constitutes a substitute for Yampa, a standard pure Arrowized FRP library. BearRiver is built on top of Dunai, following the proposal in Chapter 5. BearRiver's Effectful Signal Functions are parameterised over a monad, making it more versatile than Yampa. An additional module *FRP.Yampa* reexports all of BearRiver and instantiates its Signal Function with the *Identity* monad to provide an API-compatible replacement for Yampa[3].

The time transformations mechanisms proposed in Chapter 5 have been implemented in Yampa and in Dunai. The latter also includes asynchronous *MSF* coordination facilities.

Section 6.1 presents the evaluation of the *MSF* framework Dunai and the FRP library BearRiver. Section 6.2 explores how time transformations aid in the development of interesting features in non-trivial games. Section 6.3 presents a brief description of how the asynchronous synchronization facilities were used to implement a tile-matching game. Section 6.4 presents a brief subjective

[1] `http://hackage.haskell.org/package/dunai`

[2] `http://hackage.haskell.org/package/bearriver`

[3] Both names for these libraries have a deeper significance in relation to the name Yampa. Yampa is a river in the state of Colorado, USA, with "river with long calmly flowing sections and abrupt whitewater transitions in between", making it "[a] good metaphor for hybrid systems" [100]. Bear River is a tributary of the Yampa river, "giving it its water", similarly to how the library Bear River captures the essence of the Yampa library and can power Yampa programs. Dunai, (also Dunay), is the transliteration of the Ukrainian name of the Danube river, which starts in the south-west of Germany and crosses Europe, west to east, uniting many countries along the way, meeting the Black Sea on the border between Ukraine and Romania. This name has been chosen following the tradition of using river names, and due to the connection of Dunai's authors to these countries and the deeper significance of uniting people with otherwise very different backgrounds.

review of how using MSFs affected programs and the task of programming itself from the author's perspective.

The implementation of Dunai and BearRiver has been accomplished jointly with Manuel Bärenz. The evaluation of those libraries for the examples presented in this chapter has been carried out mainly by the author. The addition of time transformations and asynchronous communication facilities to Yampa and Dunai has been carried out by the author, and has benefited from multiple discussions with Henrik Nilsson, Manuel Bärenz and others.

6.1 Monadic Stream Functions

The implementation of Monadic Stream Functions in Dunai and the Yampa replacement BearRiver has been validated by compiling and executing two existing Haskell games.

Magic Cookies In the Yampa game Magic Cookies [101, 102] the user needs to remove all the cookies from a board following a rule: touching any position toggles surrounding positions in a cross-like shape: up, down, left, right and centre, the cookies that were present disappear, and new ones appear where there were only cookie crumbles. The game uses SDL2 for multimedia, works for mobile devices running Android and iOS, and is available on iTunes and on Google Play for Android.

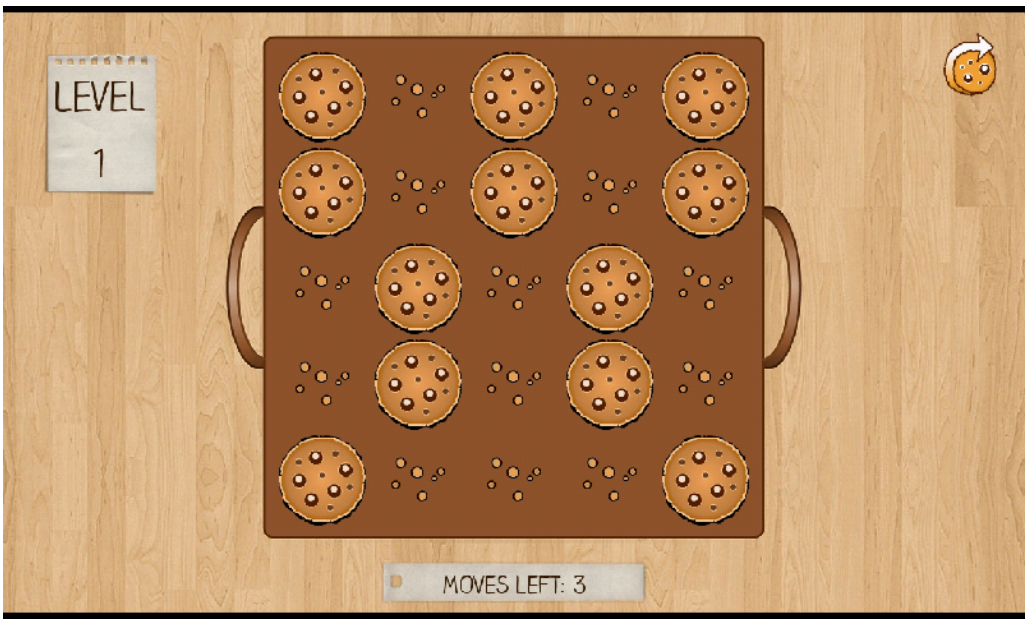


Figure 6.1: Screenshot of the Android and iOS game Magic Cookies (c) Keera Studios Ltd.

Because BearRiver is API-compatible with Yampa for the combinators it defines, the selection of which library to use to link the game can be made directly during compilation by using a flag that changes the program's dependencies during installation.

With BearRiver, the game runs in near constant memory (Figure 6.2). Enabling compiler optimisations, Magic Cookies uses approximately 325KB of heap memory with BearRiver and 300KB with Yampa.

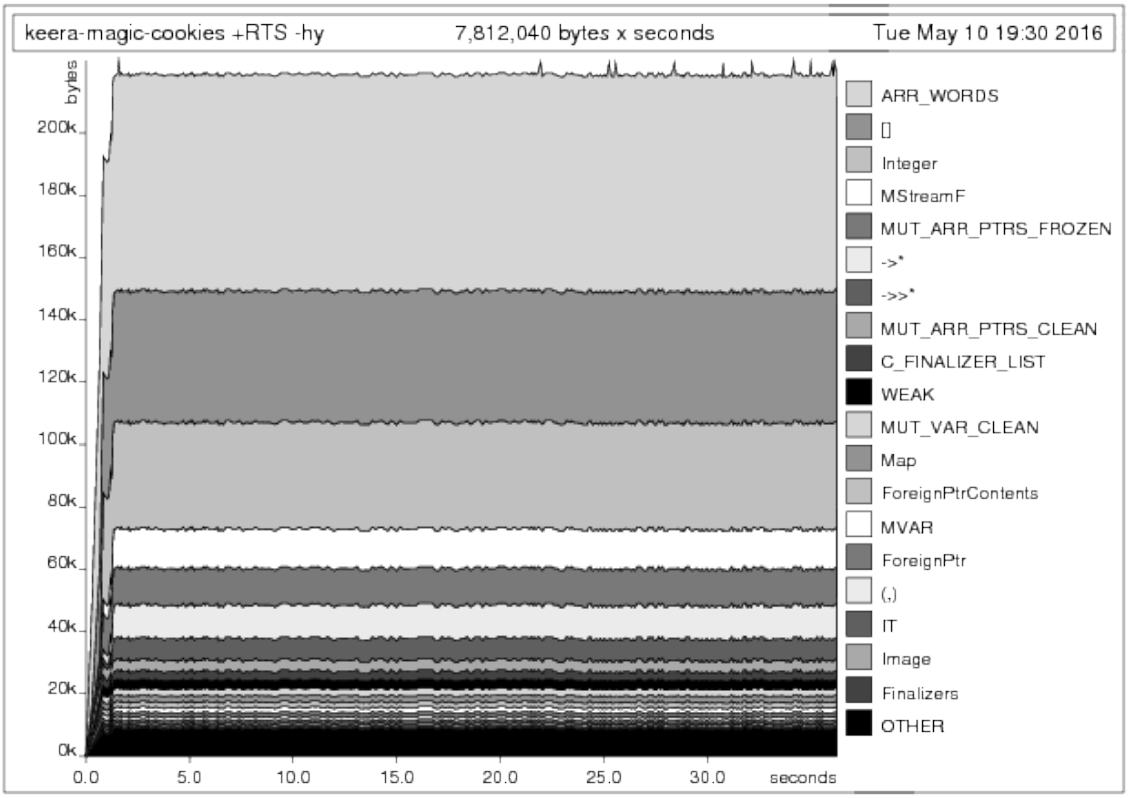


Figure 6.2: Heap profile of Magic Cookies running with BearRiver / Dunai.

Haskanoid The second game used to validate Dunai and BearRiver is Haskanoid [103], an open-source implementation of Arkanoid written in Haskell using Yampa, SDL multimedia, and supporting input devices like Kinects, Wii remotes and phone accelerometers. The game runs on Windows, Linux, Mac, iOS and Android with SDL2, and on Web with a small change of backend (Figure 6.3).

This game explores more features of Yampa and is slightly more CPU-demanding than Magic Cookies. It includes a basic collision system with convex shapes like rectangles, circles, and semi-planes. Rotations are not allowed. At any given point, there may a few dozen different elements

on the screen that must be checked for collisions. For efficiency, the game distinguishes between moving game objects and static objects, and only checks for collisions between moving objects and all objects. As this game was written in Yampa, not Dunai, the techniques for continuous collision detection presented in Chapter 5 cannot be used. Collisions are detected but not predicted.

Memory consumption remains relatively constant during a level in both cases, decreasing as blocks are removed. The game consumes approximately 2.2MB of memory with BearRiver, and 2MB with Yampa. While this shows that Yampa performs slightly better, memory consumption remains similar, and we could expect good improvements by introducing GADTs to distinguish between different kinds of Monadic Stream Functions and optimise networks.

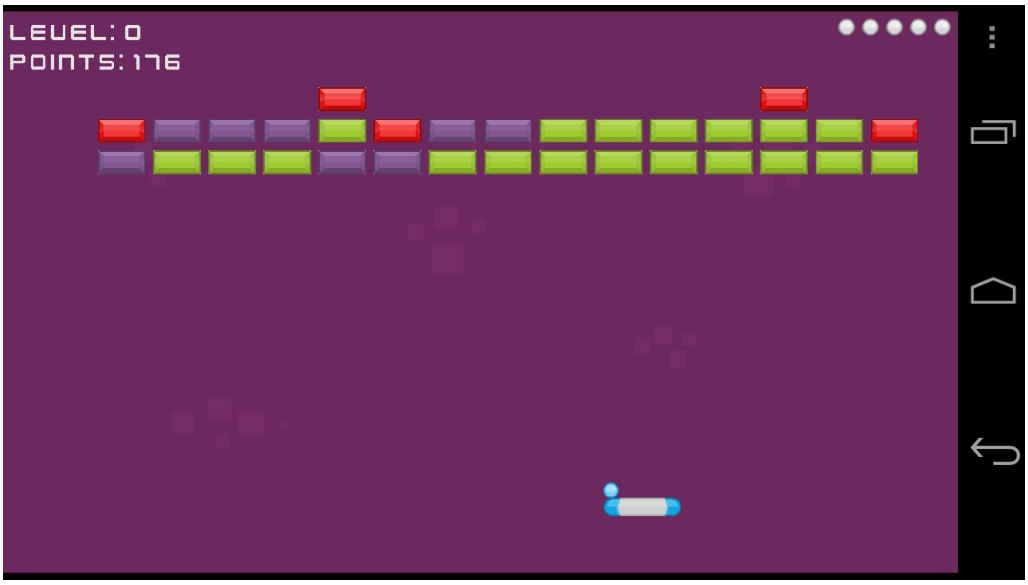


Figure 6.3: Screenshot of the open-source Yampa game Haskanoid.

6.2 Time Transformations

As presented in Chapter 5, time transformations in FRP enable a whole range of possibilities, making code more modular and also facilitating the introduction of interesting game features. To explore these new ideas, I modified an existing Yampa game, pang-a-lambda.

Pang-a-lambda (Figure 6.4) is a 2D platform game in which a player shoots balls bouncing around the screen. When hit by a bullet, balls split in two smaller ones, unless they are smaller than a minimal size, in which case they are removed. The goal is to eliminate all the balls. This game works on iOS and Android, and a version for desktop is available on hackage[4].

[4] `http://hackage.haskell.org/package/pang-a-lambda`

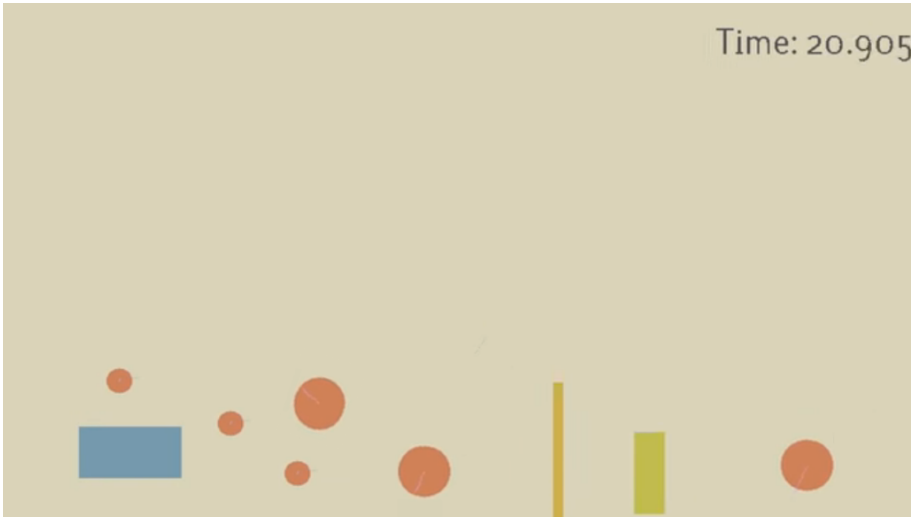


Figure 6.4: Screenshot of Pang-a-lambda.

This game has many opportunities to manipulate the progress of time to implement different effects. In this section we focus on two features: slowing time down and pausing specific elements.

6.2.1 Game Definition

Following the design of other Yampa games [40], both the player and balls are elements of a type *ObjectSF*, an *SF* from *ObjectInput* to *Object*. The type *ObjectInput* contains the input from the player plus information about previous collisions, needed for the physics and for the gameplay.

```
type ObjectSF = SF ObjectInput Object

data ObjectInput = ObjectInput
  { userInput :: Controller
  , collisions  :: Collisions }

data Object = Object
  { objectKind :: ObjectKind
  , objectPos  :: Pos2D
  , objectVel  :: Vel2D
  , ...
  }

data ObjectKind
  = Ball   Double     -- radius
  | Player Int Bool   -- lives x vulnerable
  | Projectile
```

```
  | Block   Size2D
  | Side    Side
```

6.2.2 Slowing Time Down

The player can decide when time should slow down by pressing a special key. This required modifying the type of the input to this game to know if time should be transformed:

```
data Controller = Controller
  { controllerSlow :: Bool
  , ...
  }
```

The player is only allowed to slow time down for a limited period of (real) time. This feature was implemented with a signal that counts how long it has been in use. When the feature is used, the remaining period during which the player can slow down time is gradually reset to its maximal duration. The following SF produces time transformations that need to be applied to game objects:

```
slowDown :: SF ObjectInput (DTime → DTime)
slowDown = proc (oi) → do
    rec let c      = userInput oi
            slow   = controllerSlow c
            unit   = if | power' ⩾ 0 ∧ slow    → (−1)
                        | power' ⩾ maxPower → 0
                        | otherwise         → 1
        power ← (maxPower+) ^≪ integral —≺ unit
        let power' = min maxPower (max 0 power)
            dtF    = if slow ∧ (power' > 0) then (0.1∗) else id
    returnA —≺ dtF
  where
    maxPower = 5
```

The power to slow time down should decrease over time when it is being used, and increase ("recharge") when not in use. Time in FRP can be defined as the integral of 1 over time, so power can be defined as the integral of 1, 0 or -1 over time, depending on whether the player has power left and whether it is being used or not. Power is always capped between 0 and *maxPower*. The function applied to the time deltas is $(0.1*)$ if the user wants to slow time down and there is power left, or no transformation otherwise. The use of recursive arrow syntax lets us keep the power bounded within the desired limits.

Our definitions for the player, game objects, etc. remain unchanged by the introduction of this time transformation. However, we must select which elements we want affected by slowing down the clock: while balls, items and the game time limit should be affected by this feature, the player still moves at a normal speed. I did this by manually surrounding those elements that can be slowed down with *timeTransformSF*.

This ability to affect time without having to modify the implementations of game signal functions clearly demonstrates the modularity of the proposal and constitutes one of its main strengths.

6.2.3 Pausing Time

We want to be able to also pause time under certain circumstances, for instance, when the in-game character collects a token or power-up. To achieve more interesting game effects, we can introduce this feature so that stopping time only affects balls and moving blocks, but not the player or the ability to fire bullets. When bullets hit a ball, the ball still splits normally and the new smaller balls start moving, even when others are not. This makes the feature useful for the player, while still challenging and fun.

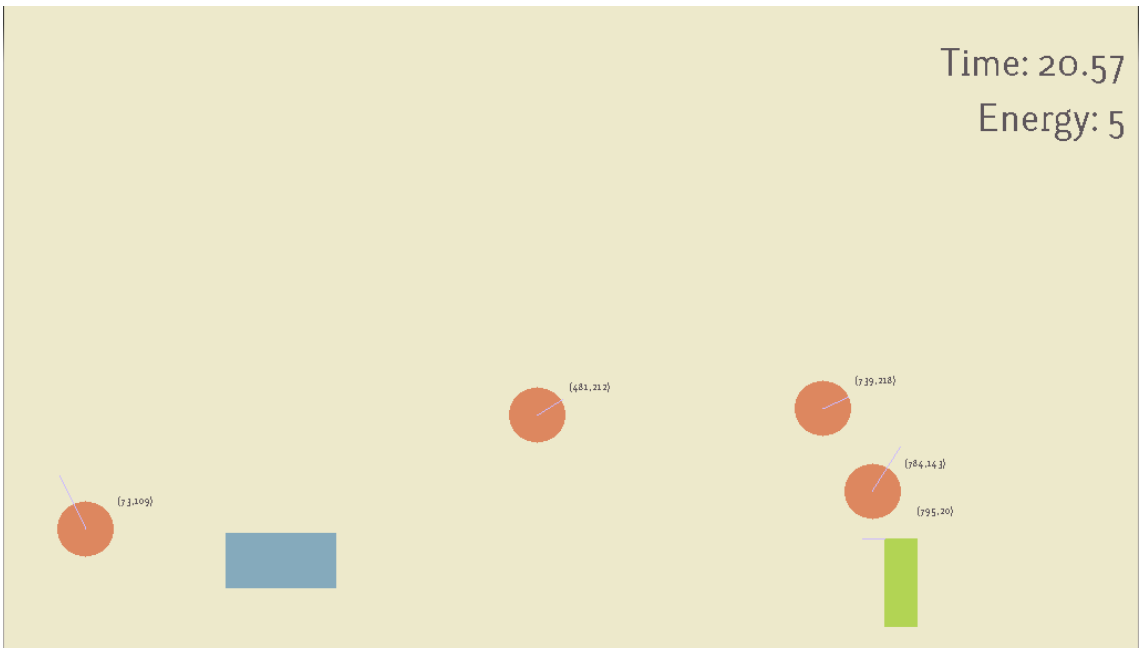
While some Yampa combinators check that time deltas are strictly positive and may throw an error otherwise, most functions work irrespective of the direction of time. The exposition of these features and an evaluation of their use in Pang-a-lambda was introduced in [32].

With a traditional FRP-based interpretation of signals and signal functions, using 0 as the time delta could lead to a violation of (temporal) referential transparency. Signal functions should transform signals so that, when the output signal after a transformation is evaluated twice for the same conceptual time, its value is the same. However, signal functions in Yampa and in the proposed framework are implemented with the assumption that flows strictly towards the future, and the current implementation could return two different outputs at successive steps that occur at the same conceptual time, even if the input is the same. Restoring referential transparency at the signal level would require defining FRP for other ordered, more structured time domains.

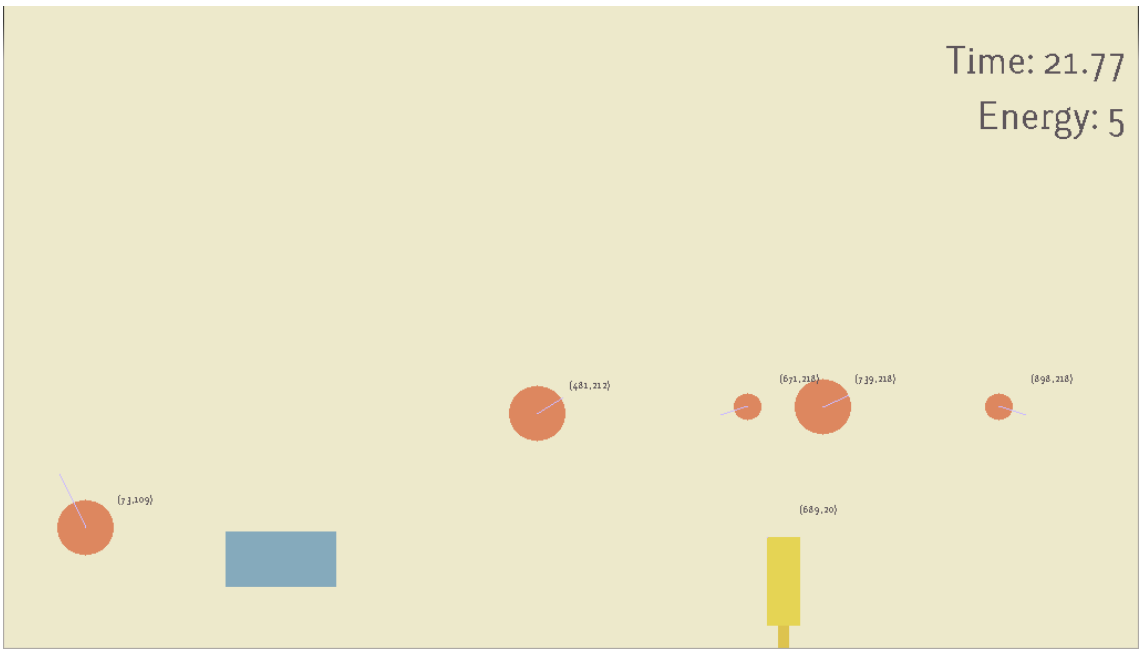
We introduce this feature in the implementation by using a time progression function that stops the clock for five seconds. The game clock keeps ticking, only the ball halts in the air:

```
timeProgressionStop :: SF ObjectInput (DTime → DTime)
timeProgressionStop = arr userInput ⋙ stopClock

stopClock :: SF Controller (DTime → DTime)
stopClock = switch (arr (λc → if controllerHalt c then (const 0, Event ()) else (id, noEvent)))
                   (λ_ → switch (constant (const 0) &&& after 5 ()) (λ_ → stopClock))
```



(a)



(b)

Figure 6.5: (a) Game with balls stopped mid-air, with player depicted in green; (b) game after shooting one ball while the others remain still, with player depicted in yellow. The mid-sized ball on the bottom-right, originally closest to the player (top image), is split in two smaller balls bouncing in opposite directions while the rest of the balls remain frozen in mid-air (bottom image).

The clock proceeds normally (*id*) unless the user activates the halting power (*controllerHalt*). If so, then the signal function switches. After the switch, it constantly halts time (*const* $0 :: DTime \rightarrow Dtime$), for five seconds (*after* 5 ()), and then it starts over again.

The player can stop the clock and move around while the balls are frozen in the air (Figure 6.5a). Balls react to user events and their arrow signal functions are being executed but, in terms of their movement, no time has passed. Yet, if a ball is hit while it is frozen, it splits in two (Figure 6.5b).

Apart from the exploration of pausing and forward time transformations seen here, I have also explored and implemented time reversal in Dunai and, partially, in Yampa. The implementation of reversible switching, caching, and all temporal aids requires keeping part of the history (a signal function, an input stream, a global clock) and modifying it to either get elements from the history when times goes backwards or adding new elements when time flows forward.

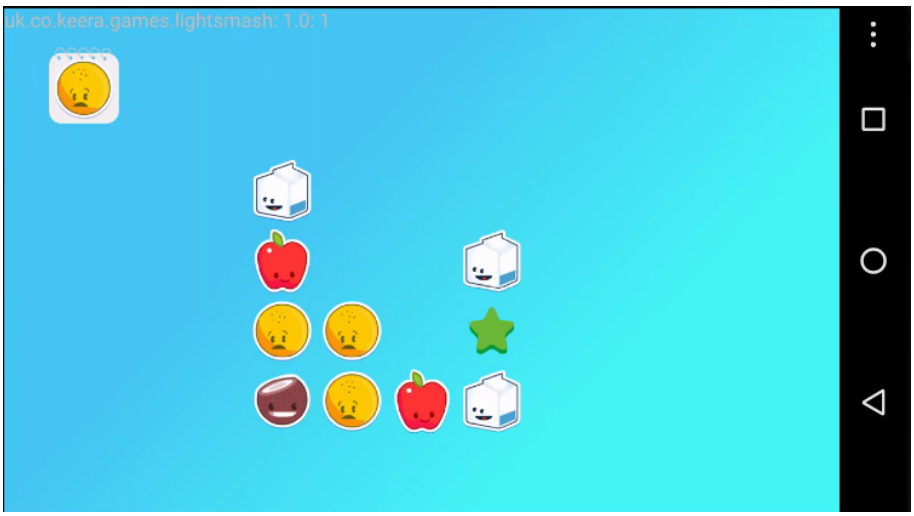


Figure 6.6: Screenshot of Lightsmash running on an Android device. (c) Keera Studios Ltd.

6.3 Asynchronous FRP

Synchronization of elements running on multiple clocks and time domains is straightforward in Dunai, as shown in the previous sections. The only two facilities that are primitive are time-polymorphic variants of *possiblyRun* and *runSelfClocking*. To drive and validate the implementation, a discrete-time mobile game was used.

Lightsmash (Figure 6.6) is a tile-matching board game with discrete changes and continuous animations. In this game there are two kinds of animations: one when tiles are dropped, and another when a group of tiles needs to be eliminated. In the first case, tiles increase in size as they fall and also travel vertically towards their destination. While they pass through other discrete

board positions, the model representation is not affected during these transitions. In the second case, tiles that are eliminated progressively decrease in size.

A goal with the design of this game was to keep the part of the program that implements the discrete board game unaware of visualization aspects. One special adaptation was required for the special case in which one animation (pieces disappearing) follows the other (pieces falling). In that situation, while both dropping and eliminating pieces occurred at, conceptually, the same time, the game was modified to make them take place at successive points in time. Otherwise, the discrete part of the game has no knowledge of visualization aspects.

6.4 Experience

Using Monadic Stream Functions is not, in principle, any harder than using Signal Functions in Yampa. A case that I have found becomes easier is passing data down using a *Reader* or *State* monad and extracting data from a stream function using a *Writer* monad.

Personally, I have found that using monad stacks with multiple transformers makes MSFs hard to work with, in spite of the multiple benefits of being able to describe switching in terms of *EitherT* or parallelism in terms of *ListT*. With this in mind, it is perhaps still convenient to add layers of abstraction on top of MSFs to describe Functional Reactive Programming or other mathematical constructs, as opposed to expressing programs directly in terms of Monadic Stream Functions (even if they are still implemented that way). Note, however, that programmers with a more mathematical background may find working with monad stacks more straightforward.

In my experience, an additional benefit of using MSFs, as opposed other FRP formulations, is that reasoning about their operational nature becomes easier, as they are, by definition, expressed in operational terms. In particular, this facilitates debugging and thinking about MSFs networks, by perceiving them as *discrete stream* functions (as opposed to continuous-time signals or signal functions).

Another benefit I have experienced when using MSFs is that the introduction of new effects and features, like the time transformations and support for asynchronous MSFs, can be done locally, even in an external library or as part of the program that needs that feature. This facilitates the task of programming itself, by making the development and installation process less cumbersome (there's no need to modify and publish new or alternative versions of pre-existing libraries), and generally increases the confidence in the library, as it can always be extended to suit one's needs.

6.5 Summary

This chapter introduced two libraries that implement the facilities described in previous chapters: Dunai, which includes Monadic Stream Functions, and BearRiver, which is an Arrowized FRP implementation built on top of Dunai.

We have validated the implementation of BearRiver and, indirectly, Dunai, by analysing the behaviour and performance of different Yampa games when compiled against BearRiver.

While Yampa performs slightly better in terms of memory consumption, both implementations are comparable. Because Yampa is optimised using GADTs to simplify FRP networks, we might expect that similar optimisations could be added to Dunai and render similar results.

We have also explored the time control facilities introduced in the previous chapter with two games: a continuous-time based game that features local time control and pausing facilities, and a board game with animations that requires asynchronous communication between the discrete model and the continuous view.

These examples demonstrate that the work presented in previous chapters is suitable for non-trivial games and that it enables declarative and more modular implementations and new and interesting time-based game features.

Chapter 7

Use Case: Reactive Values

As explained in Section 3.3.2, GUI applications implemented using FRP may lead to poor separation of concerns, poor modularity and code duplication.

Monadic Stream Functions, presented in previous chapters, address some of these concerns by introducing *monads* in a lazy stream-based functional abstraction, supporting extensibility and versatility by means of different monad stacks. This makes MSFs have the benefits of existing FRP implementations while being easier to extend and adapt.

In spite of the gain in versatility, FRP and MSFs have limitations in terms of separation of concerns and unnecessary code duplication when used to implement event-driven GUI applications.

This chapter proposes a higher-level abstraction specifically tailored to GUI applications and widget frameworks, reducing code duplication and focusing on declarativeness and modularity.

The proposed solution is based on two concepts: *Reactive Values* (RVs) and *Reactive Relations* (RRs). RVs can be used to represent widget properties, model fields, files on disk, network sockets and other changing entities. They can be combined and adapted using a series of combinators. RVs can be connected using declarative uni- or bi-directional *Reactive Relations*.

On the one hand, this proposal enables easy, uniform access to arbitrary existing GUI toolkits and other external resources. On the other hand, it blends seamlessly with the functional programming paradigm and with FRP concepts, it scales well, and it enables modularity.

The relation between RVs and MSFs is two-fold[1]. First, MSFs can be used to describe RVs and to implement them, so MSFs can serve as a suitable abstraction on top of which higher-level

[1]Historically, Reactive Values appeared before MSFs, as a way to define an abstraction that facilitated operating with event-driven toolkits with minimal overhead and maintenance costs. It was only later, when the use of RVs was shown to be beneficial, that, together with Henrik Nilsson and Manuel Bärenz, we decided to explore the addition of controlled side effects to FRP's signal functions, eventually creating MSFs. After that, RVs were re-visited to express their execution and evaluation in terms of MSFs, a contribution that is presented in this chapter.

notions like RVs and RRs can be expressed. Second, and supported by the previous point, RVs can be connected using RRs, but also MSFs and other FRP frameworks, which facilitates combining discrete, event-driven static systems represented using Reactive Values to dynamic, time-driven transformations described using MSFs.

The rest of this chapter is structured as follows. We first explore, in more detail, the problems we seek to address. We then examine Reactive Values and Relations, illustrating with concrete examples and explaining how the proposal addresses the problems identified. We continue by analysing the relation with Monadic Stream Functions and the Reactive Programming notions introduced in Chapter 5. Finally, we conclude with details about the implementation, real-world applications and the impact that this abstraction had on their architecture.

7.1 Motivation

Let us very briefly recall the specific limitation of FRP that we are trying to address in this chapter.

FRP tries to shift the direction of data-flow, from *message passing* onto *data dependency*. This helps reason about *what things are over time*, as opposed to *how changes propagate*.

A way to impose such restriction is to have one and only one place in our code where a signal can be defined. That is, signals are defined *by* their values over time: we cannot declare a signal in one place and define its value over time somewhere else. This has traditionally been seen as an advantage, as it results in clear and declarative specifications. There is, however, one limitation as a consequence of this feature of FRP.

Consider, for instance, the screenshot from Xournal [82] shown in Figure 7.1. There are at least four different ways to move from one page to the next: with the toolbar buttons (top), by dragging the central area with the mouse (centre left), by scrolling down the page (centre right), and with the bottom toolbar controls. Each of these acts *both as an input and an output*. No matter which of the four methods we use, the central area will show different contents, and scroll bar will be at a different position, the toolbar buttons will be enabled/disabled depending on whether there are more pages before/after, and the bottom toolbar page selection text entry will show a different number. The following pseudo-FRP code illustrates their mutual dependencies:

$$
\begin{aligned}
&\mathit{toolbarButtonRight} \leftarrow \mathit{button}\ \texttt{"rightarrow.png"} \\
&\qquad\qquad\qquad\qquad [\mathit{enabled} := \mathit{liftA2}\ (\mathit{not} \circ \mathit{isLast})\ \mathit{currentPage}\ \mathit{numPages}] \\
&\mathit{pageSelectionEntry} \leftarrow \mathit{numEntryText}\ [\mathit{value} := \mathit{currentPage}] \\
&\mathit{pageArea} \qquad\quad \leftarrow \mathit{renderPage}\ \mathit{file}\ \mathit{currentPage} \\
&\mathit{currentPage} \qquad \leftarrow \mathit{accum}\ 0 \\
&\qquad\qquad\qquad\qquad [(\mathit{clickOn}\ \mathit{toolbarButtonRight}\ `\mathit{tag}`\ (+1))
\end{aligned}
$$

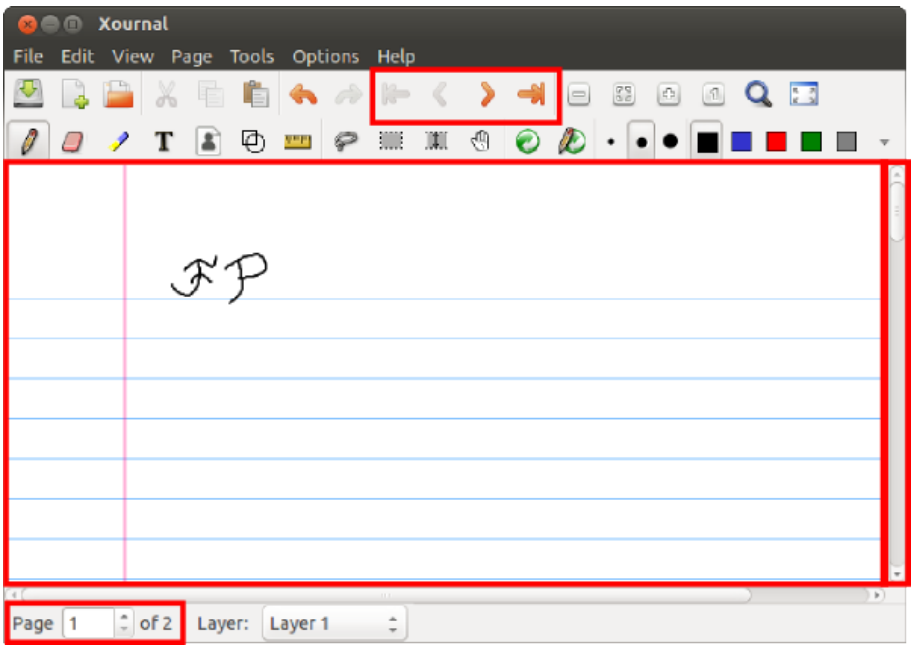


Figure 7.1: Screenshot of Xournal highlighting four different ways to change the page number.

```
    ‘merge‘ (enterText pageSelectonEntry ‘tag‘
                (const (value pageSelectionEntry)))
    ‘merge‘ ...
]
```

We need *currentPage* to define *toolbarButtonRight* (line 1), *pageSelectionEntry* (line 3) and *pageArea* (line 4), but we need all three (and probably many others) to define the value of *currentPage* (line 5). These mutually dependent elements have to be in scope of one another, which, in practice, requires that we define them together in the same module. This impairs modularity and separation of concerns, a problem that is exacerbated as the codebase grows.

Some FRP variants offer mechanisms to work around this problem. Reactive Banana [84], for instance, offers *sinks* for each WX widget property, to which a signal can be attached. Monadic Stream Functions can be used to implement sinks and address some of these limitations, but, because the whole system ticks in synchrony in a pull-based fashion, it would require the evaluation of all signal functions or manually freezing specific parts using the mechanisms introduced in previous chapters. The formalism proposed in this chapter is more suitable to abstract event-based GUI frameworks.

7.2 Reactive Values and Relations

To address the issues discussed before, let us introduce *Reactive Values* (RVs), which are *typed mutable value with access rights and subscribable change notification.*

RVs provide a *uniform interface* to GUI widgets, other external components such as files and network devices, and application models (the conceptual problem representation in MVC [12]). Each entity is represented as a collection of RVs, each of which encloses an individual property.

RVs can be transformed and combined using a range of combinators including (n-ary) function lifting and lens application [104].

To specify how the values of RVs are related, allowing them to be synchronised during execution in response to changes, we introduce *Reactive Relations* (RRs). A Reactive Relation is either *uni-* or *bi-directional*. RRs can be thought of as connecting RVs, such that a change to a value will produce an update of other(s).

RRs are defined separately from RVs. Indeed, relations involving the same RVs can even be defined in separate modules. This is in contrast to FRP signals, which are defined by their dependencies on other signals. Allowing RRs to be defined separately is a *key strength* of this approach, as this promotes separation of concerns and modularity (Section 7.1). The work presented here addresses static RRs and provides no way of removing installed relations. So far we have not found this to be a major limitation.

MVC controllers [12] can thus be seen as sets of Reactive Relations. Because the model is reactive and notifies of changes to it, the controller no longer needs to know how changes propagate within the model. This allows us to move more of the problem's logic into the model (whenever it conceptually belongs there), while minimising data propagation from the model to the view.

The API allows RVs to be created from pure models, widget properties and external elements. As we cannot cover every possible use case, we provide a low-level API that can be used to support other widgets and external entities, implement further synchronisation abstractions and introduce other RV combinators.

7.2.1 Definition

Conceptually, a *Reactive Value* (RV) is a *typed mutable value* that can be *transformed*, and *connected* to external I/O entities (like widgets or files) or to other Reactive Values. In this chapter we capture this idea with a series of types, type classes, operations and combinators.

Intuition We formally define RVs as a *type* equipped with operations to read it, modify it, and execute a monadic computation when the value changes.

As a first attempt, and to help convey the intuition, let us introduce a basic definition of an RV as the following type and operations[2]:

```
data ReadWrite a = ...
read        :: ReadWrite a → IO a
onCanRead   :: ReadWrite a → IO () → IO ()
write       :: ReadWrite a → a → IO ()
```

The operations *read* and *write* are a getter and a setter for the value held in the RV. The function *onCanRead* helps us install a computation to be executed when the value in the RV changes.

In absence of interaction from other RVs, processes and threads, we might have certain expectations about the behaviours of such RVs. For instance, if we see them as plain mutable variables with change propagation, we might expect that writing the same value already present in the RV should not affect it ($\forall\ rv\ .\ read\ rv \gg\!= write\ rv \equiv return\ ()$), or that writing a value and immediately overwriting it is the same as only writing the second value ($\forall\ rv\ a1\ a2\ .\ write\ rv\ a1 \gg write\ rv\ a2 \equiv write\ rv\ a2$). These properties, which resemble the lens laws, hold for some basic RVs so long as they are not connected to external entities or to other RVs in a network.

Example The following examples belong to a posture monitoring application (Section 7.4.2) that uses a webcam to monitor the user's sitting posture and trigger a warning when it differs too much from a reference posture.

Users can customise how much time needs to pass until a warning is triggered. In the GUI (Figure 7.2) this is configured using a spin button (text entry for numbers).

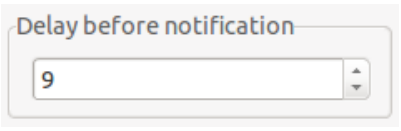


Figure 7.2: Entry used to configure the warning delay.

Following the MVC architectural pattern we would expect to have definitions like the following[3]:

```
-- UI (GTK+) module
delaySpinButton :: SpinButton
delaySpinButton = ...
```

[2]We purposefully omit the details of the abstract data type *ReadWrite*, as we only need the operations below to work with it.

[3]The implementation is omitted to avoid unnecessary details at this point.

```
  -- Model module
data Model = Model {
  notificationDelay :: Int
  ...
}
```

A reactive application would typically *also* include the following two definitions:

```
delayEntry :: ReadWrite Int                -- UI: Entry value
delayEntry = ...
notificationDelayField :: ReadWrite Int    -- Model
notificationDelayField = ...
```

The first of the two Reactive Values represents the numeric value held in *delaySpinButton*; the second one represents the field *notificationDelay* of the model.

In Section 7.2.4 we describe how to connect Reactive Values to keep them in sync during program execution. The two Reactive Values above are *both read/write* and *have the same type*, which will make connecting them bi-directionally straightforward. But it may not always be so easy. For example, in the same posture monitoring application, a text entry gives users the possibility of customising the sound played when the posture is "incorrect". We need to connect the following two RVs:

```
soundEntryText :: ReadWrite String          -- UI
soundEntryText = ...
soundField :: ReadWrite (Maybe FilePath)    -- Model
soundField = ...                            -- Nothing: Default sound
                                            -- Just fp: Sound in file fp
```

which have different types. In Sections 4.3 and 7.2.4 we explore see how to adapt the types when they do not match and how to connect different kinds of RVs so that they stay in sync during execution.

General definition Not all RVs are readable and writeable. Additionally, some GUIs already support operations analogous to the ones given above, which makes the additional type wrapper unnecessary.

RVs are best defined by type classes that describe how we can operate with them:

```
class Readable r a where
  read       :: r → IO a
  onCanRead  :: r → IO () → IO ()
```

```
class Writable r a where
  write :: r → a → IO ()
class (Readable r a, Writable r a) ⇒ ReadableWritable r a
```

An RV carrying values of some type a is a type r that implements one or more of these type classes. In these definitions, the variable r stands for the RV itself, and the variable a stands for the type of value it contains (*Int*, *Bool*, etc.). For simplicity, we use the *IO* monad in this chapter, which is the most common case in practice. In the complete implementation, these type classes are parameterised over the monad, enabling easier integration with different backends.

To facilitate the exposition, we introduce three basic types for RVs that implement the above type classes, although many others can be defined:

```
data ReadOnly  a = ...
data WriteOnly a = ...
data ReadWrite a = ...
```

ReadOnly is an instance of *Readable*, *WriteOnly* is an instance of *Writable*, and *ReadWrite* instantiates all three. The use of ellipsis (...) merely indicates that, at that point the discourse, the details that would follow in place of the ellipsis should not be highlighted.

7.2.2 Creation

Reactive Values are created from and linked to an underlying entity. These "backing" entities can be external (widgets, files, etc.) or internal (pure values and application models).

In this section we limit our discussion of GUIs to GTK+, but the same approach can be used with other toolkits such as wxWidgets or Qt. The implementation [105] includes examples of reactive applications written with different toolkits.

Externally Backed Reactive Values

Some Reactive Values represent mutable entities that exist outside our functional code. These could be, for instance, GUI widgets and their properties, files, network connections or hardware devices.

Graphical User Interfaces In GTK+ terminology, widgets (graphical components) have attributes (properties) with access rights (read/write). Widgets may trigger *signals* to execute event handlers when attributes change or when other events take place (clicks, key presses, etc.).

Checkboxes, for instance, have attributes such as the state (checked/unchecked) and whether users can interact with them (enabled/disabled). Clicking on an enabled checkbox toggles its state and fires an event that can be handled programmatically (Figure 7.3).

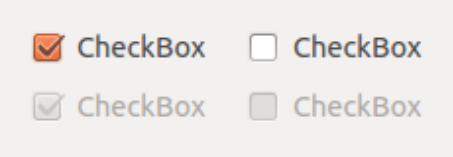


Figure 7.3: Possible states of a checkbox. Checkboxes can be checked/unchecked (left/right columns) and enabled/disabled (top/bottom rows).

There is a one-way correspondence between GTK+'s signals and attributes and our Reactive Values. In most cases, an attribute defines a Reactive Value, possibly accompanied by the signal (event) triggered when the attribute changes (Figure 7.4).

The API provided by the implementation covers most of the essential widget properties in gtk2hs [106]. It also provides a generic signal/attribute-based interface, suitable for widget properties not specifically supported. Additionally, there are experimental reactive layers for wxWidgets, Qt, Android, iOS and web [105].

Example (GUIs) To access a text entry's text, we use:

$$entryTextReactive :: (EditableClass\ e, EntryClass\ e) \Rightarrow e \rightarrow ReadWrite\ String$$

which, for a given GTK+ text entry, returns an RV representing the text in the entry. The text can be accessed and modified via the Reactive Value, which fires notifications when it changes.

We may also be interested in detecting events that do not correspond to any property or carry

GTK+	Reactive equivalent
Read-only attribute (+ associated event)	*ReadOnly* Reactive Value
Write-only attribute	*WriteOnly* Reactive Value
Read-write attribute (+ associated event)	*ReadWrite* Reactive Value
Signal or event	*ReadOnly* Reactive Value

Figure 7.4: Correspondence between GTK+ and Reactive Values.

data (for instance, a button being clicked). Events can always be seen as read-only RVs carrying no information, e.g.:

$$buttonActivateField :: Button \rightarrow ReadOnly\ ()$$

Given a button, this function defines an RV that fires a notification when the button is clicked.

Files Reactive Values can be used to interact with files and other sources/sinks of information. The predefined function *fileReactive* creates an RV enclosing a file:

$$fileReactive :: FilePath \rightarrow ReadWrite\ String$$

We use a file monitoring library[4] that relies on OS facilities to monitor files for changes without polling. This results in an RV that will notify dependent RVs when the file changes on disk.

Example (Files) The following RV is connected to the file *myFile.txt*. When the RV is written to, so is the file. When the file changes, RV subscribers are notified:

$$myFile :: ReadWrite\ String$$
$$myFile = fileReactive\ \texttt{"myFile.txt"}$$

Network Similarly, an experimental network reactive layer allows sockets to be seen as RVs. For instance, the function:

$$udpSink :: HostName \rightarrow String \rightarrow IO\ (WriteOnly\ String)$$

creates a writeable Reactive Value that sends any text written to it to the specified host and port using User Datagram Protocol (UDP).

Internally Backed Reactive Values

Library users have access to the value constructors of different RVs, and can thus define RVs that enclose pure values. In most applications we want to be able to detect when values change, update other RVs accordingly and guarantee thread-safe access to the Reactive Value.

The implementation includes a library with a definition of "very light" RVs that fulfils all of these requirements. The library offers several RV constructors, of which the default one compares the value of an RV with the previous one before setting it, to break unnecessary propagation loops.

This solution works well for simple programs, but it is suboptimal for very large applications: a change to only one part of a value (for instance, the first component of a tuple) will provoke forward propagations to RVs that depend only on other parts that did not change (for instance, the second component of the same tuple).

[4] `https://hackage.haskell.org/package/fsnotify`

Protected Models To address the aforementioned scalability concerns we define an abstraction that encloses an application's model, called Protected Model, implemented as a polymorphic, thread-safe mutable variable with change detection and an event dispatching thread, parameterised over two types[5].

The first type argument of *ProtectedModel* represents the type of values stored in it, that is, the pure model it encloses. The second argument acts as an identifiable reference to a part of the model and is used to identify what has changed[6]:

```
data (Ord e, Eq e) ⇒ ProtectedModel a e =
  ProtectedModel
    { reactiveModel :: TVar (ReactiveModel a e)
    ...
    }
data (Ord e, Eq e) ⇒ ReactiveModel a e =
  ReactiveModel
    { basicModel        :: a
    , eventHandlers     :: M.Map e [IO ()]
    , pendingHandlers :: [IO ()]
    ...
    }
```

The use of STM TVars [16], guarantees exclusive access and atomic modifications.

Protected Models can be created with the function *startProtectedModel*. This function also starts a dispatcher thread that executes pending handlers:

```
startProtectedModel :: (Ord e, Eq e) ⇒ a → IO (ProtectedModel a e)
```

In the following we see how to define RVs that give access to only parts of a Protected Model.

Projecting Protected Models to Reactive Values The main difference between a plain RV and a Protected Model is that the latter is intended to be a collection of RVs. Thus, one should define RVs that project specific parts of the Protected Model.

To make that extra layer as simple as possible for the users of our library, we provide a high-level API that uses Template Haskell to define RVs that represent projections of fields of the model. For instance, given:

[5]The actual implementation uses the type class *Event* for identifiable changes, which is an instance of *Ord* and *Eq*. Events have additional, orthogonal uses in our framework. To facilitate understanding, we use a simpler version here.

[6]In some of our programs, we overload the type e with semantic information about the nature of the change itself. See [24, p. 33] for more details.

```
data Model = Model {
  language :: Maybe Language
  ...
}

data ModelChange = LanguageChanged
                 |      ...
  deriving (Eq, Ord)
```

the following call to Template Haskell in a module (all the referred types, and additional RV libraries, must be in scope):

```
protectedField "Language" [t | Maybe Language |] '' Model '' ModelChange
```

generates a definition with signature:

```
languageField :: ProtectedModel Model ModelChange → ReadWrite (Maybe Language)
```

Of course, a lower level API to Protected Models and Reactive Values is also available, which can be used in case the given Template Haskell is not adequate for the user's needs.

Protected Models can incorporate more machinery than simply change detection and event dispatching. For instance, in the SoOSiM UI[7] and Gale IDE (Section 7.4.1), the Protected Model incorporates a change-aware undo/redo queue. The model is extended with three operations to control the queue, which can be used by the controller. The RVs generated using *protectedField* are the same.

Protected Models allow us to hide other design decisions, such as having a global event dispatcher vs executing events in new threads as they come in. This ability to introduce orthogonal features without affecting the rest of the codebase is another key strength of this proposal.

7.2.3 Transformation

Reactive Values can only be connected if they have compatible types. We can transform and combine Reactive Values by lifting (n-ary) functions, by applying lenses, and by letting one control the other (*governance*).

Unary lifting A function of type $a \to b$ can be applied to a Reactive Value in one of two ways:

- To write values of type a into a writeable RV of type b (converting the a into a b *before* writing).

[7] `https://github.com/ivanperez-keera/SoOSiM-ui`

- To read values of type b from a readable RV of type a (converting the values *after* reading).

This implies that:

1. Lifting *one* function onto a read-write Reactive Value will render a read-only or write-only Reactive Value.
2. To produce a read-write Reactive Value, we need to lift *two* functions, one for each direction of the transformation (Figure 7.5).

We thus define three unary lifting combinators:

$$
\begin{aligned}
&\mathit{liftR} &&:: \mathit{Readable}\ r \Rightarrow (a \to b) \to r\ a \to \mathit{ReadOnly}\ b \\
&\mathit{liftW} &&:: \mathit{Writable}\ r \Rightarrow (b \to a) \to r\ a \to \mathit{WriteOnly}\ b \\
&\mathit{liftB} &&:: (a \to b, b \to a) \to \mathit{ReadWrite}\ a \to \mathit{ReadWrite}\ b
\end{aligned}
$$

Read-only RVs are covariant in the read end, write-only RVs are contravariant in the write end.

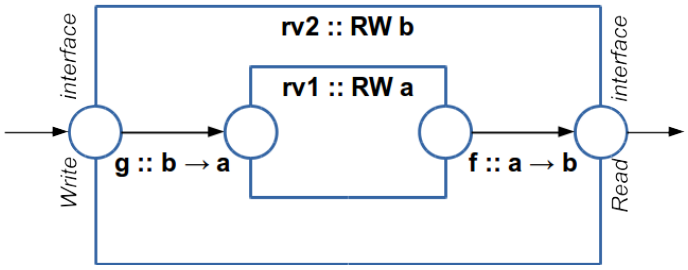


Figure 7.5: Applying a bijection to a read-write RV: $rv2 = \mathit{liftB}\ (f, g)\ rv1$

Example (lifting) Continuing with the previous example, we might want to render the language selection in a label, for which we need to transform the *Maybe Language* from the model into a *String*. We might do so as follows:

```
showLangSelection :: Maybe Language → String
showLangSelection (Just English) = "EN"
showLangSelection (Just Spanish) = "ES"
showLangSelection Nothing        = "--"

langText :: ReadWrite (Maybe Language) → ReadOnly String
langText lang = liftR showLangSelection lang
```

If we are given a function to parse the language selection, then we can easily make the Reactive Value writeable as well:

```
readLangSelection :: String → Maybe Language
readLangSelection "EN" = Just English
readLangSelection "ES" = Just Spanish
readLangSelection _    = Nothing

langTextRW :: ReadWrite (Maybe Language) → ReadWrite String
langTextRW lang = liftB (showLangSelection, readLangSelection) lang
```

Read-only Reactive Values are *Functor*s and *Applicative*s. Write-only Reactive Values are *Contravariant* functors. Using the *Applicative* and *Contravariant* infix lifting operators, we can write clearer, less verbose code.

```
langText :: ReadOnly (Maybe Language) → ReadOnly String
langText lang = showLangSelection <$> lang
```

As seen in the signature of *liftB*, transformations applied to read-write Reactive Values require a contravariant transformation for the write end (input), and a covariant one for the read end (output). We capture this idea with a new class, *GFunctor*, which defines a generic mapping function. For the case of bi-directional transformations applied to Read-write RVs, the *GFunctor* instance implements *profunctors* (bifunctors which are contravariant in one argument and contravariant in the other). A (simplified) definition of *GFunctor* is:

```
class GFunctor f where
  gmap :: (a → b, b → a) → r a → r b
```

The instance for Read-write RVs could be defined as:

```
instance GFunctor ReadWrite where
  gmap = liftB
```

To make code more readable, we can use the infix *gmap*, named < $$ >, analogous to <$> for functors. For example, the following is a more succinct definition of *langTextRW*:

```
langTextRW :: ReadWrite (Maybe Language) → ReadWrite String
langTextRW lang = (showLangSelection, readLangSelection) < $$ > lang
```

When lifting functions onto read-write Reactive Values, it is often desirable that the transformation be an *isomorphism* (in which case we would lift the function by the functor and the inverse by the contrafunctor). Given the limitations of Haskell, we cannot but trust users in this respect, providing only a small facility for *involutions*:

```
reversedText :: ReadWrite String
reversedText = (involution reverse) <$$> textValue
  where
    textValue :: ReadWrite String
    textValue = ...
```

Not using real isomorphisms may impact performance. The default setters compare the new values to the old ones (if they are instances of *Eq*). This stops unnecessary data propagation and breaks loops. However, if the inverse provided is not the true inverse, the value that propagates in the inverse direction after a change may cause a new propagation. It is therefore necessary to provide inverses that will lead to a fixed point. This is discussed further in Chapter 12.

N-ary lifting Similarly to how we can achieve unary lifting, we can lift n-ary functions into RVs using analogous functions (e.g. *liftR2*, *liftW2*, *liftB2*, etc.). The signatures of *liftR2* and *liftW2*, for instance, are:

```
liftR2 :: (Readable r1 a, Readable r2 b) ⇒ (a → b → c) → r1 → r2 → ReadOnly c
liftW2 :: (Writable r1 b, Writable r2 c) ⇒ (a → (b, c)) → r1 → r2 → WriteOnly a
```

N-ary lifting onto read-only values can also be achieved using applicative syntax [41].

Example (n-ary lifting) We could, for instance, render several configuration parameters in a tuple, to later show them in a label, as follows:

```
notificationDelay :: ReadWrite Int
notificationDelay = ...
correctionFactor :: ReadWrite Int
correctionFactor = ...
configurationPair :: ReadOnly String
configurationPair = liftR2 (λd f → show (d, f)) notificationDelay correctionFactor
```

Lenses Lenses [104] provide a way to *focus* on subparts of data structures by means of a getter and a setter that read from and inject values into an existing structure. Lenses are compositional (they can be combined using a notation similar to function composition), and can be derived automatically for some type definitions.

Lens application onto RVs is a specific form of lifting bijections. We provide a specific lens lifting combinator:

```
(<$$$>) :: Lens' a b → ReadWrite a → ReadWrite b
```

Example (Lenses) Given the lens $(\_1) :: Lens'\ (a, b)\ a$, which focuses on the first component of a pair, one can write:

$$
\begin{aligned}
&window1Top &&:: ReadWrite\ Int\\
&window1Top &&= \_1 <\$\$\$> window1Position\\
&window1Position &&:: ReadWrite\ (Int, Int)\\
&window1Position &&= \ldots
\end{aligned}
$$

Governance Another possible way of combining RVs is by letting one control another. Consider, for instance, the case in which one wants changes to a text box to be "reported" *only* when the button is depressed. If we use *liftR2* to combine them, both clicks on the button and text entry changes will trigger notifications. To address these situations we can use the function:

$$governing :: Readable\ r\ a \Rightarrow r \rightarrow ReadWrite\ b \rightarrow ReadWrite\ b$$

which defines a new Reactive Value that encloses the value in the second argument, and notifies of changes only when the first RV changes. An analogous function is provided for read-only RVs.

Examples (governance) Following the case described above, we often want the text of an entry not to be synchronised or passed around, except when a button is clicked. We can use governing to create a RV that encloses the entry's text, but whose changes are only propagated when the user clicks the button:

$$
\begin{aligned}
&buttonAndEntry &&:: ReadWrite\ String\\
&buttonAndEntry &&= button1Clicks\ `governing`\ textEntry1Text\\
&button1Clicks &&:: ReadOnly\ ()\\
&button1Clicks &&= \ldots\\
&textEntry1Text &&:: ReadWrite\ String\\
&textEntry1Text &&= \ldots
\end{aligned}
$$

7.2.4 Reactive Relations

So far we have seen ways to *create* Reactive Values, but we have not seen any way to *relate* readable and writeable RVs to allow changes to be propagated correctly to achieve overall consistency (for instance, to synchronise two text boxes, or an RV that represents a Protected Model field with one that encloses a widget attribute).

We introduce rule-building functions to capture the idea that two Reactive Values must be "in sync" for all time. The functions <:= and =:> (depending on the direction of change propagation)

build directional synchronisation relations. The source value (the origin of the change) must be *readable*, the destination must be *writeable*, and they must contain values of the same type. To simplify code further we also have at our disposal the function =:=, syntactic sugar for two directional relations. Their types are as follows:

$$(<:=) :: (Writable\ r1\ a, Readable\ r2\ a) \Rightarrow r1 \rightarrow r2 \rightarrow IO\ ()$$

$$(=:>) :: (Readable\ r1\ a, Writable\ r2\ a) \Rightarrow r1 \rightarrow r2 \rightarrow IO\ ()$$

$$(=:=) :: (ReadableWritable\ r1\ a, ReadableWritable\ r2\ a) \Rightarrow r1 \rightarrow r2 \rightarrow IO\ ()$$

Example (reactive relations) The posture monitoring application needs a GUI to manipulate the delay until a notification is presented. We have already seen how to define each Reactive Value (one for the GUI and one the model). To keep them in sync we write:

```
delayEntry =:= notificationDelayField
```

This combinator installs the necessary change listeners on each RV, so that the other RV is updated when either changes. For another example, based on *configurationLabel* defined earlier, we can show the correction factor and the warning delay in a label:

```
configurationPair :: ReadOnly String    -- Model
confLabel         :: GtkLabel           -- UI
confLabelString   :: ReadWrite String   -- UI
confLabelString   = labelString confLabel
rule = confLabelString <:= configurationLabel
```

where *configurationLabel* was defined in the last example that accompanies Section 4.3. Note that the rule is directional: there is no need to update the model from the view in this case because labels are not interactive.

To answer a question posed at the end of Section 7.2.1, we can use lifting and a relation to synchronise two RVs of different types:

```
soundEntryText :: ReadWrite String                -- UI
soundEntryText = ...
soundField     :: ReadWrite (Maybe FilePath)      -- Model
soundField     = ...
stringToMaybe :: String → Maybe FilePath
stringToMaybe "" = Nothing   -- Default sound
stringToMaybe fp = Just fp   -- Sound in file fp
```

```
rule =
  soundEntryText =:= (fromMaybe "", stringToMaybe) < $$ > soundField
```

7.2.5 Choreographies

GUI programs often contain common, re-occurring patterns. By parameterising Reactive Relations over Protected Models and Views containing certain Reactive Values, we can describe sets of relations in separate libraries that can be reused within the same application and across multiple applications. These abstract patterns are referred to as *choreographies*.

Let us examine a very simple one that has been found both illustrative and useful: showing the file name in the title bar of the main window. Type classes capture the requirements of model and view.

```
class ModelWithFilename m e where
  filenameField :: ProtectedModel m e → ReadWrite FilePath
class ViewWithMainWindow v where
  mainWindow :: v → Window
composeTitleBar :: (ModelWithFilename model event, ViewWithMainWindow view)
                ⇒ String → model → view → IO ()
composeTitleBar programName model view =
    (composeTitle `liftR` filenameField model) =:> (windowTitleReactive (mainWindow view))
  where
    composeTitle fp = fp ++ " - " ++ programName
```

Choreographies are usually more complex and not limited to one relation. They can contain internal models and views, and spawn threads. For example, the choreography that offers to save files when a program is closed contains two relations (one to present the confirmation dialog,'linebreak one to save the file), introduces one additional model type class and contains a view of its own (the dialog).

7.3 Relation to Monadic Stream Functions

External mutable entities can be regarded as both sources and sinks. For instance, in programs with Graphical User Interfaces, text boxes can be seen as *String*-carrying streams and sinks.

We could represent Reactive Values simply as pairs of a stream and a sink on the same monad and type, as done in Chapter 5, and connect them using the standard Monadic Stream Function and Arrow combinators that we saw before.

However, Reactive Relations that connect RVs in a circular way would lead to incorrect results. For example, imagine that we try to synchronise two text boxes, *textField1* and *textField2*, each of which defined only one string-based RV. With the framework defined so far, we would a reactive rule like the following:

$$textField1 =:= textField2$$

To implement this rule using the Arrow-based interface with streams and sink, and assuming that *textField1Stream* represents the read end of the Reactive Value *textField1*, and similarly for the write end (Sink) and *textField2*, then we might write the following:

$$\begin{aligned}&(\quad (textField1Stream \ggg textField2Sink)\\ &\&\&\& \ (textField2Stream \ggg textField1Sink))\end{aligned}$$

However, because these expressions are defined in the *IO* monad, which is not commutative, the first data propagation would always be executed before the second. This would mean that this kind of connection would always update *textField2* with the text from *textField1* box, regardless of which one actually changed.

To address this problem, we need individual push-based evaluation, for which we can follow the original design of the Reactive Values and tuple a stream, a sink, and a change event handler installer that executes an arbitrary monadic action whenever the value changes:

$$\begin{aligned}&\textbf{type } ReactiveValueRO\ \ m\ a = (Stream\ m\ a, m\ () \to m\ ())\\ &\textbf{type } ReactiveValueWO\ m\ a = Sink\ m\ a\\ &\textbf{type } ReactiveValueRW\ \ m\ a = (Stream\ m\ a, Sink\ m\ a, m\ () \to m\ ())\end{aligned}$$

Additionally, we need to modify *reactimate* to run only one step of the MSF each time a source stream changes:

```
pushReactimate :: MSF IO () () → IO (IO ())
pushReactimate msf = do
  msfRef ← newIORef msf
  return $ do msf' ← readIORef msfRef
              msf'' ← snd <$> step msf' ()
              writeIORef msfRef msf''
```

Note that this is only defined for *MSFs* on the *IO* monad, and therefore will constrain this implementation to systems connecting *IO RVs*.

With this interface, push-based connections need to be specified at the monadic level. Both directional and bi-directional connections can be expressed declaratively using rule-binding combinators. Recall that, in the proposed design, RR have type *IO* () when they connect Reactive Values defined in the *IO* monad:

```
(=:=) :: ReactiveValueRW IO a → ReactiveValueRW IO a → IO ()
(sg1, sk1, h1) =:= (sg2, sk2, h2) = do
  (sg1, h1) =:> sk2
  (sg2, h2) =:> sk1

(=:>) :: ReactiveValueRO IO a →  ReactiveValueWO IO a → IO ()
(sg, h) =:> sk = h =≪ pushReactimate (sg ⋙ sk)
```

Note that, unlike the (⋙) combinator, (=:>) does not actually update the right-hand side if the left-hand side has not changed. We can now express the example above as:

```
do txtField1Stream =:> txtField2Sink
   txtField2Stream =:> txtField1Sink
```

Defining RVs in terms of MSFs simplifies the implementation, as we can use MSF combinators to implement RV transformations. At a conceptual level, this also shows that MSFs are a suitable abstraction to reason about bi-directional connections between reactive entities.

7.4 Evaluation

An implementation of Reactive Values is available online as a collection of libraries, as part of the Haskell GUI Programming toolkit Keera Hails [105]. They provide definitions of Reactive Values and Reactive Relations, and bindings for a series of backends, including Protected Models, GTK+ widgets and properties, files, network sockets, FRP signal functions (using Yampa [27]) and Nintendo Wii Controllers. It also includes libraries to simplify common architectural patterns (MVC) as well as choreographies often needed in different kinds of applications. At the time of this writing, the libraries comprise over 7K lines of code.

I have used this approach to develop several real-world applications, currently amounting to slightly over 25K lines of Haskell (not counting comments, empty lines or code generated automatically by the library, using Template Haskell or the Keera Hails project generator, which generates an initial project skeleton).

Examples of the software created include an interactive tool[8] to visualise Supercomputer Operating System node simulations [107], a webcam-based posture monitor (Section 7.4.2), a OCR-based PDF renamer and a Graphic Adventure IDE (Section 7.4.1). I have also published several demonstration applications and small examples, such as an SMS sender[9] and a replacement for

[8] `https://github.com/ivanperez-keera/SoOSiM-ui`

[9] `https://github.com/ivanperez-keera/keera-diamondcard-sms-trayicon`

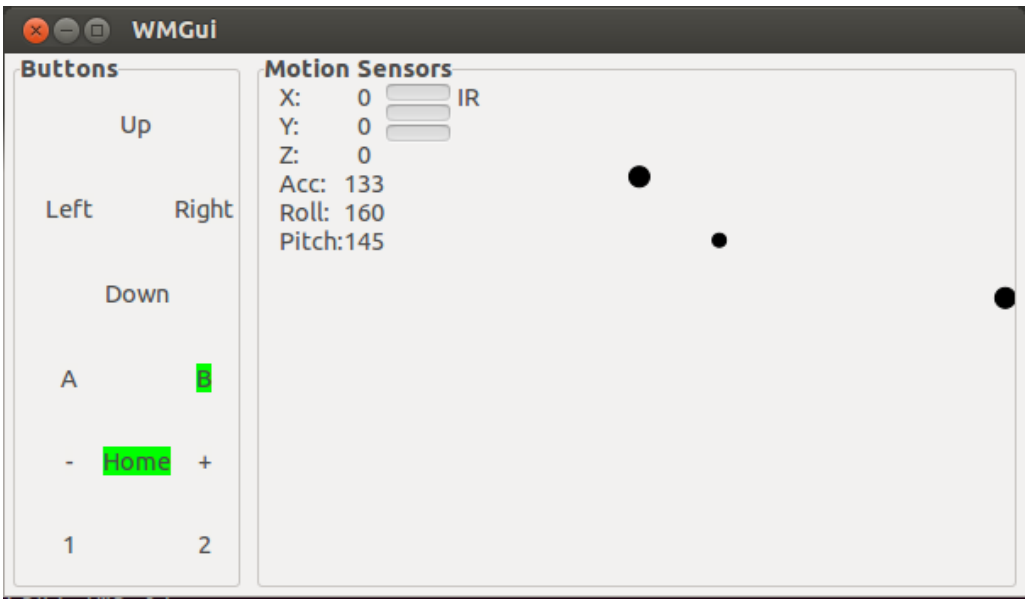


Figure 7.6: A demo that shows the state of a Nintendo "Wiimote".

WMGUI[10], the Nintendo Wii Controller debugging GUI available on most Linux distributions (Figure 7.6).

7.4.1 Gale IDE

Keera Gale [108] is a graphic adventure development IDE written entirely in Haskell[11]. The IDE uses GTK+ for the user interface, and allows users to create graphic adventure games without prior knowledge of programming. Users can define screens, characters, actions and reactions, enable conversations and customise the game interface. Other stages of the game design process, such as storyboarding and state transition diagrams, are also supported. The IDE is accompanied by a graphic adventure engine written in Haskell using SDL2, that has been tested on Windows, Linux, Android and iOS. The final distributable file can be generated directly from the IDE using only the GUI. Games created using Gale IDE are currently being beta-tested on Google Play.

The program uses MVC as its main architecture. The IDE features, at the current time, 385 modules of Haskell code, without including the engine or other auxiliary components. 228 of those modules conform the application's controller and contain 50 per cent of the code. Template Haskell is used to generate the View (from glade files) and the reactive layers of the model, decreasing the number of lines of code further.

[10] `https://github.com/keera-studios/hails-reactive-wiimote-demo`
[11] `http://keera.co.uk/blog/products/gale-studio/`

A separate thread is used to handle a responsive progress dialog when the distributable files for the game are being generated. The controller starts that thread, but further communication occurs only indirectly through the protected model (Figure 7.9).

The controller contains over 70 Reactive Relations. I have ported imperative MVC Haskell code to this new reactive interface, and using Reactive Values and Relations makes the controller's modules between 50 and 66 per cent smaller (in lines of code, without comments or empty lines) compared to code that had already been optimised to avoid code duplication due to bi-directional synchronisation[12]. Some of these Relations merely connect Reactive Values, while others include also connections between Reactive Values and low-level monadic code.

The controller makes heavy use of choreographies to eliminate boilerplate code. Re-occurring patterns include synchronising the selection on a tree view and on a tab page group (using the tree view to change the tab page), and efficient interaction with dynamic lists of elements (scenes, objects, etc.).

Furthermore, because Reactive Values encapsulate both bi-directional access and the relevant notification subscription mechanisms in one unique entity, we have observed that we are less likely to make errors such as installing handlers on the wrong events.

Keera GALE IDE clearly shows that this approach addresses all three problems introduced in Section 7.1: it uses a standard GUI toolkit (GTK+), it enables functional style through the use of reactive relations, and it is large and complex enough to prove that this approach scales well in terms of code modularity, and even enhances it.

7.4.2 Keera Posture

Keera Posture is a posture monitor written in Haskell using OpenCV for image recognition and GTK+ for the GUI[13].

The program works by comparing the current position of the user (estimating the distance based on the size of the head) with a reference position given during program calibration. When both differ "too much", a warning is shown.

Users can customise the sensitivity, the form of the warning (popup message, message bubble and/or sound), the sound being played, the language and the webcam being used. The initial calibration uses a step-by-step assistant. Users can pause the program by clicking on its icon, located in the system tray bar.

[12] In bi-directional synchronisation one needs to obtain the values on both sides, compare them and possibly update one side. The original code already received the direction of the update as a parameter, so that the code that polled the view and the model could be shared for both directions.

[13] `http://github.com/keera-studios/keera-posture`

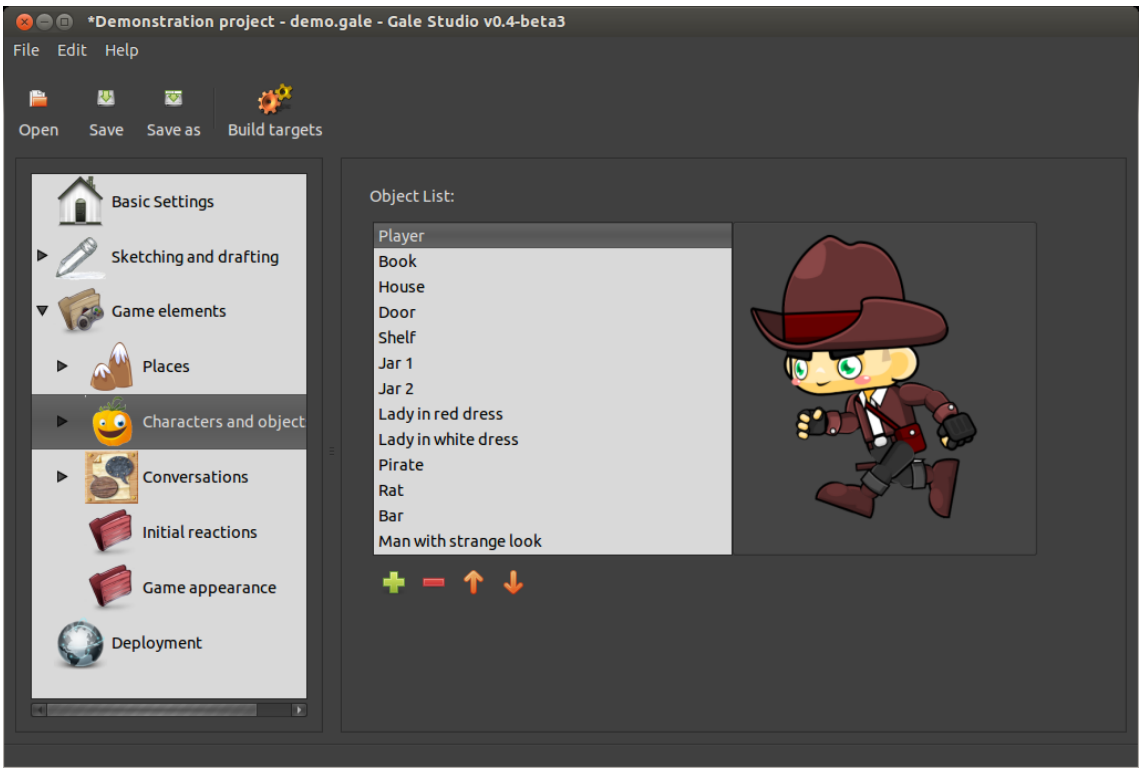


Figure 7.7: GALE IDE's object preview screen, showing an animation with each character's default state. Double clicking on any object opens the object details screen, where users can modify the object properties by selecting states, animations, actions applicable to them and reactions to those actions.

The program has been implemented for end-users and thus care has been placed on providing common usability features and an intuitive user interface. Both Windows and Linux are supported.

Like Gale IDE, Keera Posture runs several threads to keep the GUI responsive while doing image recognition. Changes in the posture are communicated to the GUI only indirectly, through the protected, reactive model.

Of the 53 modules included in the program, the Model contains 13 (plus 4 which are generated automatically). The Model constitutes 30% of the code (measured in lines, without comments or empty lines) and exposes 16 Protected Model fields (projections of model parts onto Reactive Values). The Controller contains 30 modules, which constitute 50% of the code and comprise 29 Reactive Relations. The image recognition module contains 10% of the code, and the View (generated during compile time from a GTK+ Glade file, using Template Haskell) contains 4%.

Keera Posture is a clear demonstration of how, using the right abstraction, one can write software that addresses real problems, in a purely functional way, with minimal boilerplate code.

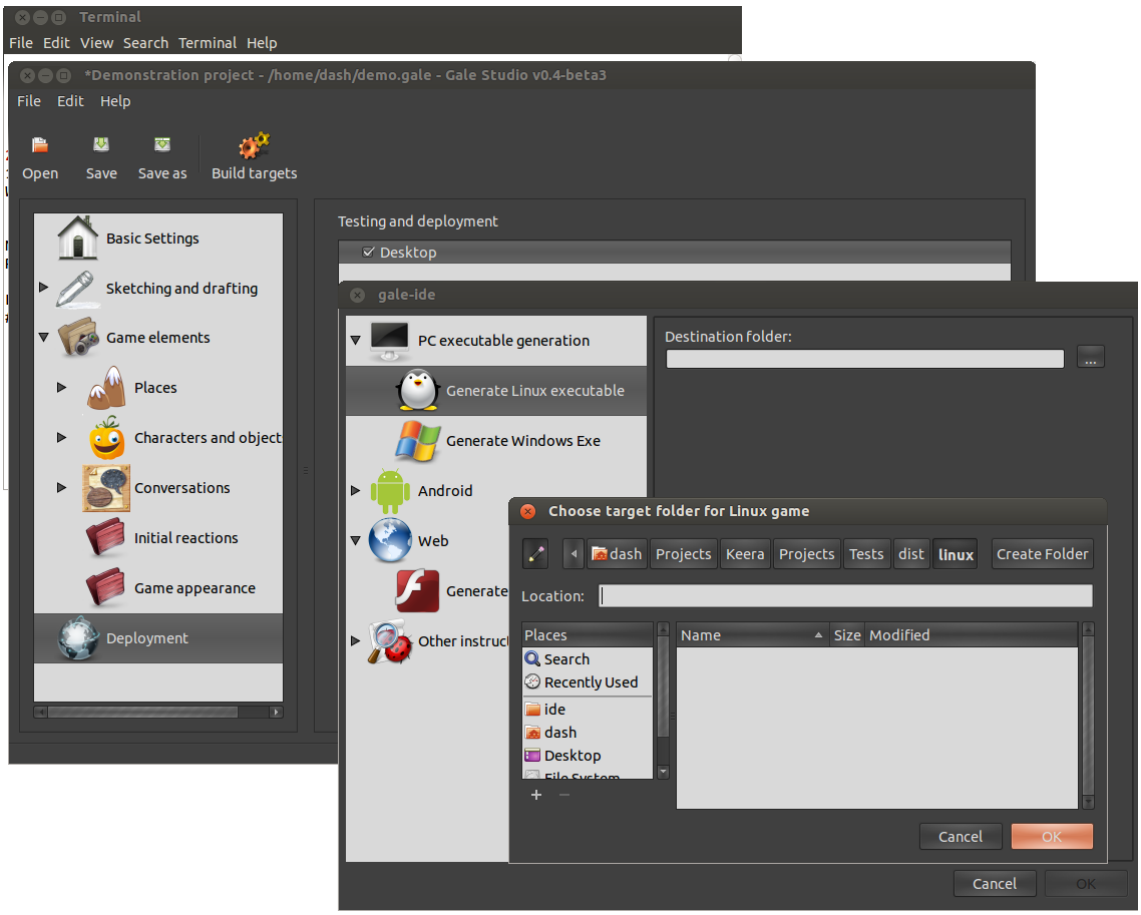


Figure 7.8: GALE IDE's target configuration screen. GALE IDE can target Windows, Linux, Android and Web. This screenshot of the running application shows three nested windows: the main application, the target/distributable selection window, and the target directory selection dialog.

Also, through the use of Reactive Values and Relations, it exemplifies how one can limit the side effects of using imperative bindings mainly to the GUI, without sacrificing any of the features that standard GUI toolkits offer.

7.4.3 Experience

From a programmer's perspective, I have found that using Reactive Values and Reactive Relations helps simplify and structure the code of applications, reduces code duplication, and facilitates debugging and modularity.

RVs can be used to define both reactive GUIs and reactive models, which already suggests a clear separation between the two. While this is also possible with any other paradigm, RVs both

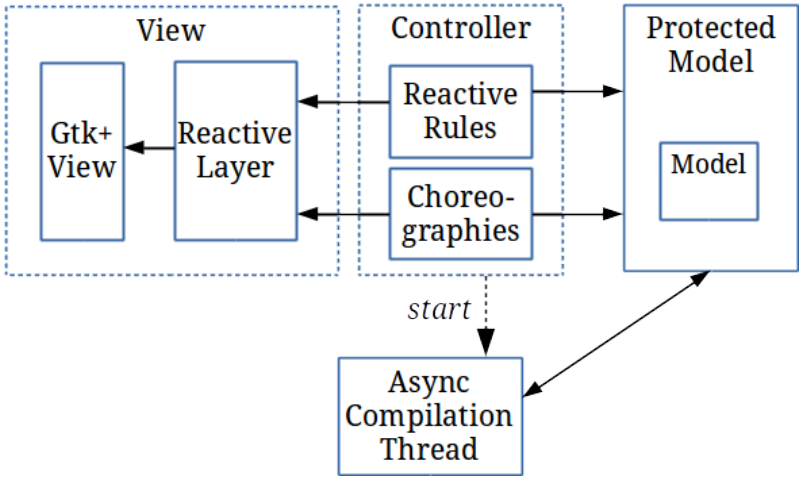


Figure 7.9: GALE IDE's architecture. The compilation thread is started by the Controller, but further communication takes place only indirectly, through the Protected Reactive Model.

facilitate and encourage that division.

Reactive Relations and the ability to use the same RV in multiple relations enables structuring the controller as a collection of relations, each of which deals with a specific feature, or a part of the model or the GUI. The main controllers of many of these reactive applications become just a large monadic **do** block that enumerates a collection of relations imported from many separate modules. This facilitates debugging and re-structuring, since it becomes trivial to disable a feature by not installing the related relations from the main controller.

By following minimal standards and using the model or the GUI as a map of the application to structure the collections of Reactive Relations, I have also found that code becomes much simpler to understand. Some programs have been in production for years without any changes and without any bugs ever reported. Others have been improved over time, but these changes have been much easier to introduce, compared to the alternative of using MVC in Haskell. In particular, I have found that it becomes much easier to understand where an issue could be or where a change needs to be introduced even years after the code was originally written.

In principle, the ability to use the same Reactive Values as sinks from different relations adds complexity, due to not knowing what value an RV will have at a given point during the execution. However, in practice I have not found this to be a big concern, since few Reactive Relations write directly with the same RV, and all can be located by searching for the RV's name in a text file. Furthermore, this problem already exists for GTK+ widgets regardless of the abstraction used. Nevertheless, exploring the impact of using an RV as sink in many relations remains an open problem.

7.5 Summary

We began this chapter by identifying a limitation in Functional Reactive Programming that leads to code duplication, and poor modularity and separation of concerns.

We proposed a different abstraction to deal with the visual interactive elements that form most Graphical User Interface toolkits, based on regarding them as collections of mutable values that we can connect to one another via transformation functions. We proposed the abstraction of Reactive Values for the mutable properties, and Reactive Relations for the uni- or bi-directional connections between them.

The proposed abstraction is also useful to deal with the application's model, sockets, files and other external systems, making it a suitable mechanism to define the connections between multiple parts in an application and to reason about them.

While this work seems to overlap with Monadic Stream Functions, proposed in Chapter 4, we have also seen that the Reactive Programming concepts presented in Chapter 5 help express notions of Reactive Values and define their implementation.

Finally, we also discussed different applications built using this framework, and how the new notions impacted their design. The fact that RVs are, in principle, so low-level, makes this framework easy to connect to any existing GUI toolkit. However, as we have also seen in those applications, the formalism is high-level and abstract enough to enable building modular and well-structured applications with tens of thousands of lines of code.

Part III

Robustness

Chapter 8

Testing

Testing software thoroughly is hard in general [109]. Testing games raises specific additional difficulties due to their interactive and real-time aspects [110]. For example, just specifying game input so as to ensure adequate test coverage is daunting. As a result, game developers often leave much of the testing to testers and even players. Another difficulty is the lack of reproducibility: in general, the exact same input may produce different results at different times. This is due to both deliberate random elements in many games and the effects of interacting with a constantly changing outside world at points in time that are difficult to control exactly due to real-time considerations. Consequently, the conditions that led to a bug can be hard to replicate.

These challenges apply generally, regardless of what language is used to implement a game. Thus, while using a pure functional language can offer benefits at the level of unit testing thanks to referential transparency, just using a pure language offers few if any additional benefits over commonly used languages for game programming when it comes to testing a game as a whole.

Functional Reactive Programming (FRP) [25, 27, 40] and Monadic Stream Functions help express interactive software, such as games, declaratively, in a way that arguably brings the benefits of pure functional programming to the *system level*. In this part, we demonstrate that *Arrowized* FRP in particular provides enough structure to address whole-program testing and debugging challenges such as those outlined above. The presented work applies to FRP programs in general, but we put a particular emphasis on games, physical simulations, and related applications in the following chapters, as these tend to highlight the difficulties of interest here.

There are two sources of uncertainty in FRP programs that may affect system-level referential transparency. First, some FRP implementations use *IO* inside core definitions to increase versatility and performance. This makes those systems susceptible to the problems discussed earlier. We refer to those implementations as *impure*, and those that separate logic from I/O as *pure*. Second, even

if we run a simulation twice with the same initial state and user input, we may obtain different outputs if our simulations are sampled at different points in time, due to differences in system load and OS scheduling decisions outside our control, for example. This is demonstrated by the force-directed graph simulation depicted in Figure 8.1. This simulation does not depend on user input, yet different stable configurations are reached depending on the sampling step, due to small differences in floating-point calculations.

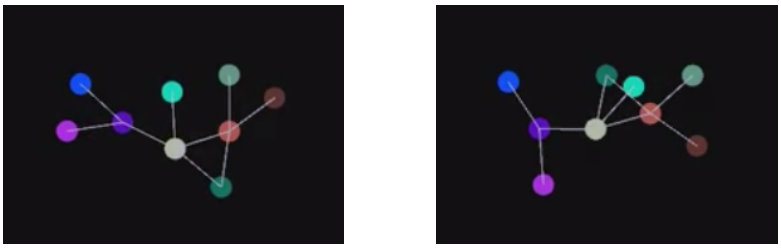


Figure 8.1: Two runs of the same graph layout algorithm in which nodes repel each other, unless directly connected, in which case they also attract. Even with identical initial conditions, two executions converge to different stable configurations, demonstrating that pure FRP *systems* exhibit non-determinism.

Pure Arrowized FRP [26, 27] completely separates all side effects and time sampling from the data processing, providing referential transparency across executions. In this variant we can truly run a program twice with the same input, poll it at the same times, and obtain the same output, enabling a form of game testing unparalleled by other languages and paradigms. Given the same architecture the results will be guaranteed by the type system and the compiler to be the same, *even* across different devices. This is a key aid for game development, especially on mobile platforms, since players often find bugs that developers cannot reproduce.

In this and the following chapters we explore how Arrowized FRP enables game testing and debugging to be approached systematically in pure functional languages. The contributions of this part are as follows:

- We extend FRP constructs with a notion of temporal predicates based on Linear Temporal Logic, equipped with an evaluation function, and demonstrate how they can be used to express temporal game properties.
- We provide random input data generators, and demonstrate how they can be used effectively to test FRP games using QuickCheck.
- We present a *causal* subset of Linear Temporal Logic that can be used to insert temporal assertions into FRP programs for revealing bugs during execution.

- We present an extension to an FRP implementation that allows users to record and replay input traces, together with a Graphical User Interface that can be used to control and debug running games.

The implementation works for PC, iOS and Android games written with the FRP framework Yampa. It is capable of recording, replaying, manipulating and visualizing execution traces from real users, as well as counter-example traces generated by QuickCheck. Our application lets us traverse an input trace and find points where assertions are violated, moving back and forth along the execution timeline and performing hot-swapping of the application to verify whether changes to the game fix existing bugs. We demonstrate the use of this setup to find bugs in real games.

Limitations of Current FRP Systems

FRP programs use I/O to gather input data and render their output. Unlike in pure Arrowized FRP, some FRP implementations hide I/O behind the scenes, and reading a signal may result in *IO* actions being executed [80]. While they use low-level adjustments to guarantee that polling a signal at the same conceptual time delivers temporally consistent results across the whole program, without additional adjustments these implementations lack referential transparency across executions.

Even in *IO*-free forms of FRP, simply sampling a simulation at different points in time can render different results. The non-determinism introduced by the sampling policy is demonstrated by the force-directed graph simulation in pure FRP depicted in the introduction, in Figure 8.1. Implemented in Pure Arrowized FRP, this simulation is not interactive, but the numerical methods used to calculate forces, velocities and positions depend on the sampling time, and are highly affected by small variations. As a result, the layouts obtained when the simulation stabilises are very dependent on the sampling step.

Because pure Arrowized FRP separates all data processing from all side effects, including the *time sampling policy*, we could truly run the program twice with the exact same input and sample at the same times, obtaining the same results. This constitutes a form of referential transparency across executions suitable to debug complex interactive applications, especially games.

In the following we take advantage of this feature of Arrowized FRP to test and debug FRP games. We present an extension to Arrowized FRP with Temporal Propositions, and introduce two ways of using temporal properties: in a black-box manner, to test the behaviour of an FRP construct for some input data, and inside FRP programs as assertions, to monitor and debug FRP systems from the inside. We provide APIs to test these programs in a referentially transparent way, using tools like QuickCheck. We also present a debugger for Yampa games, together with an

advanced GUI for live debugging that connects to running games on PC, iOS and Android devices via the network and allows controlling them, recording, replaying and manipulating input traces.

8.1 Temporal Logic

Temporal Logic is a branch of Modal Logic in which propositions are quantified over time [111, 112]. Just like the value of a signal in FRP varies over time, so does the truth value of a proposition in Temporal Logic.

Temporal Logic constitutes a suitable framework to express temporal properties of changing and reactive systems (for details, see Chapter 12). In this subsection we introduce a formal definition of temporal propositions in the same domain used to give denotational semantics to FRP, that of *time-varying values*, or *functions of time*. This will help us understand which Temporal Logic constructs can be defined in terms of FRP constructs.

8.1.1 Semantic Domain

In the following we will consider time abstract and assume only that it is linear (that is, that there is a total order and progresses towards the future) with a starting point denoted 0. In Temporal Logic, the nature of time (bounded vs unbounded, discrete vs continuous, etc.) plays a central role, determining both which modalities can be defined and which propositions hold. In FRP, time is conceptually continuous, but always discrete during execution. We consider a temporal proposition to be a time-dependent function, that is, a boolean whose value changes over time:

$$TProp \cong Time \to Bool$$

8.1.2 Point-wise Operators

Defining the standard logical operators, like *and* and *not*, is straightforward:

$$\begin{aligned}
&neg : TProp \to TProp \\
&neg\ \phi\ t = not\ (\phi\ t) \\
&and : TProp \to TProp \to TProp \\
&and\ \phi\ \psi\ t = \phi\ t\ \wedge\ \psi\ t \\
&or : TProp \to TProp \to TProp \\
&or\ \phi\ \psi\ t = \phi\ t\ \vee\ \psi\ t
\end{aligned}$$

$$impl : TProp \to TProp \to TProp$$
$$impl\ \phi\ \psi\ t = neg\ \phi\ t\ \vee\ \psi\ t$$

The previous logical operators act point-wise (or time-wise), and their meaning should be obvious. For instance, *neg* ϕ is true at a time t if ϕ is not true at that time.

8.1.3 Temporal Modalities

In temporal logic we also want to define propositions that refer to other points in time, for which we define four temporal operators: *always* (at every point in the future), *eventually* (at some point in the future), *history* (at every point in the past), and *ever* (at some point in the past). These correspond to the well-known Priorean operators Global, Future, History and Past [113].

$$always : TProp \to TProp$$
$$always\ \phi\ t = \forall\ t'\ \in\ Time\ ((t' > t) \to \phi\ t')$$
$$eventually : TProp \to TProp$$
$$eventually\ \phi\ t = \exists\ t'\ \in\ Time\ ((t' > t)\ \wedge\ \phi\ t')$$
$$ever : TProp \to TProp$$
$$ever\ \phi\ t = \exists\ t'\ \in\ Time\ ((t' < t)\ \wedge\ \phi\ t')$$
$$history : TProp \to TProp$$
$$history\ \phi\ t = \forall\ t'\ \in\ Time\ ((t' < t) \to \phi\ t')$$

We can use these operators in combination with the previous ones to describe properties such as that ϕ eventually becomes true forever, as *eventually* (*always* ϕ), or that ϕ will always be true in the future, but has not always been true in the past, as *ever* (*neg* ϕ) ‘*and*‘ *always* ϕ. Note that these operators have non-strict counterparts, in which the value of the predicate argument ϕ at the current time t is also taken into account.

The above temporal operators are sometimes defined in terms of more fundamental operators, namely *Until* and *Next*. Until is defined as:

$$until : TProp \to TProp \to TProp$$
$$until\ \phi\ \psi\ t = \exists\ t'\ \in\ Time\ (t' \geqslant t \to (\psi\ t'\ \wedge\ \forall\ t''\ \in\ Time\ (t'' < t' \to \phi\ t'')))$$

That is, ϕ holds at every point until ψ does.

Next expresses a temporal property that holds at the next time sample. This requires that time be discrete, which is true in our implementations, although time in FRP can be conceptually continuous. We use the function N to refer to the successor or next time.

$$next : TProp \rightarrow TProp$$
$$next\ \phi\ t = \phi\ (N\ (t))$$

The semantic domain of temporal logic, as given above, is bigger than that of FRP. While FRP Signal Functions must be *causal*, functions over Temporal Propositions may not be so. Modalities like *always*, for example, make the present depend on the future, and thus cannot be implemented easily and efficiently in Arrowized FRP as Boolean-carrying signals. Even if we limit our attention to modalities that look only to the present or the past, *history*, for example, depends on *all the past*, which could lead to memory leaks depending on how it is implemented or evaluated.

In Section 8.2 we present an encoding of Linear Temporal Logic with *always*, *eventually* and *next* modalities, which we use to test temporal predicates for a given input signal and time domain. In Chapter 9 we introduce a causal subset of the LTL presented before that can be implemented efficiently in FRP in the form of boolean-carrying signals, letting us define temporal assertions checked as games run live.

8.2 Temporal Logic Applied to FRP

In this subsection we present an encoding of Linear Temporal Logic operating on Signal Function Temporal Propositions. This allows us to test temporal properties of SFs, considered as black boxes. In subsection 8.3 we use this encoding to demonstrate how to test temporal predicates using existing tools like QuickCheck. Chapter 9 explores the inverse question, that is, which temporal operators can be implemented as causal FRP constructs and used to include assertions in FRP programs checked during execution.

8.2.1 Temporal Predicates

The following implementation of Linear Temporal Logic allows us to look into the present and the future. We introduce a basic value constructor to make Signal Functions unapplied temporal propositions. We also provide the two fundamental temporal operators *Next* and *Until*, and the two non-fundamental *Eventually* and *Always*, which correspond to the non-strict counterparts of the temporal modalities with the same names described in the previous subsection.

```
data TPred a where
Prop    :: SF a Bool → TPred a
And     :: TPred a → TPred a → TPred a
Or      :: TPred a → TPred a → TPred a
```

$$
\begin{aligned}
&\mathit{Not} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \\
&\mathit{Implies} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \to \mathit{TPred}\ a \\
&\mathit{Always} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \\
&\mathit{Eventually} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \\
&\mathit{Next} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \\
&\mathit{Until} && :: \mathit{TPred}\ a \to \mathit{TPred}\ a \to \mathit{TPred}\ a
\end{aligned}
$$

8.2.2 Evaluation

To evaluate a *TPred* we need to provide an input signal. Conceptually, we can think of a $\mathit{TPred}\ a$ for some type a as a possibly non-causal function that takes a signal of type a defined on a time domain and returns a Temporal Proposition (a signal of type *Bool*) on the same time domain.

In our implementation we provide both the signal and the time domain in the form of a finite input *stream* or sample *stream*:

$$
\begin{aligned}
&\textbf{type}\ \mathit{SignalSampleStream}\ a = (a, \mathit{FutureSampleStream}\ a) \\
&\textbf{type}\ \mathit{FutureSampleStream}\ a = [(\mathit{DTime}, a)]
\end{aligned}
$$

SignalSampleStream represents the signal starting from the first sample, always at time 0, and *FutureSampleStream* represents streams in which the samples are spaced by strictly positive delays.

We can now give an evaluation function that takes a Temporal Predicate and a sample stream and evaluates the temporal predicate at the initial time:

$$\mathit{evalT} :: \mathit{TPred}\ a \to \mathit{SignalSampleStream}\ a \to \mathit{Bool}$$

When we evaluate a *TPred* by providing a stream of input samples, we are effectively restricting the time domain to a subset defined implicitly by the times of the samples in the input stream. Note that the function *evalT* only lets us query the value of a *TPred* at the initial time, but we can always refer to specific future times using *Next*.

8.2.3 Examples

Let us now introduce an example of an FRP program, define temporal assertions and verify them. We start with a simple animation of a falling ball, considering only the vertical axis:

$$\mathit{fallingBall} :: \mathit{Double} \to \mathit{SF}\ ()\ \mathit{Double}$$

$$
\begin{aligned}
&\mathit{fallingBall}\ p0 = \mathbf{proc}\ () \to \mathbf{do} \\
&\quad v \leftarrow \mathit{integral0} \multimap -9.8 \\
&\quad p \leftarrow \mathit{integral0} \multimap v \\
&\quad \mathit{returnA} \multimap (p0 + p)
\end{aligned}
$$

where *integral0* is an implementation of Euler integration [97].

To check, for example, that the position of the falling ball at any time is lower than the initial position, one can define the following property:

$$
\begin{aligned}
&\mathit{ballFellLower} :: \mathit{Double} \to \mathit{TPred}\ () \\
&\mathit{ballFellLower}\ p0 = \mathit{Prop}\ (\mathit{fallingBall}\ p0 \ggg \mathit{arr}\ (\lambda p1 \to p1 \leqslant p0))
\end{aligned}
$$

We now test this property in a session using an input stream *stream*01 of 42 samples spaced by 0.1 seconds, with no data (that is, a stream $((), [(0.1, ()), (0.1, ()), \ldots]$ with 42 samples in total).

$$
\begin{aligned}
&> \mathit{evalT}\ (\mathit{ballFellLower}\ 100)\ \mathit{stream01} \\
&\mathit{True}
\end{aligned}
$$

However, with the given definition we are only checking this proposition at time 0. In order to check that the proposition always holds, we define:

$$
\begin{aligned}
&\mathit{ballFallingLower} :: \mathit{Double} \to \mathit{TPred}\ () \\
&\mathit{ballFallingLower}\ p0 = \mathit{Always}\ (\mathit{ballFellLower}\ p0)
\end{aligned}
$$

Testing it now tests it for every position in the stream:

$$
\begin{aligned}
&> \mathit{evalT}\ (\mathit{ballFallingLower}\ 100)\ \mathit{stream01} \\
&\mathit{True}
\end{aligned}
$$

To obtain further guarantees, we may want to check that the ball is always falling lower and lower, that is, that the new position is always lower than the previous one. We can express that idea by introducing a one-sample delay in a signal with *iPre*, and pairing the new ball position with the previous value:

$$
\begin{aligned}
&\mathit{fallingBallPair} :: \mathit{Double} \to \mathit{SF}\ ()\ (\mathit{Double}, \mathit{Double}) \\
&\mathit{fallingBallPair}\ p0 = \mathit{fallingBall}\ p0 \ggg (\mathit{identity}\ \&\&\&\ \mathit{iPre}\ p0)
\end{aligned}
$$

We can now compare the two, and check that the difference is always negative:

```
ballTrulyFalling :: Double → TPred ()
ballTrulyFalling p0 = Always (Prop (fallingBallPair p0 ⋙ arr (λ(pn, po) → pn < po)))
```

where *po* and *pn* represent the last known and new known position of the falling ball. However, this predicate does not hold at time zero, since the position at time zero is compared with itself, as we introduced it as argument to *iPre* to fill in the gap left by the shifting of the sampling stream.

```
> evalT (ballTrulyFalling 100) stream01
False
```

We can skip the first sample and check only afterwards with *Next*:

```
ballTrulyFalling' :: Double → SF () Double
ballTrulyFalling' p0 = Next (Always (Prop (fallingBallPair p0 ⋙ arr (λ(pn, po) → pn < po))))
> evalT (ballTrulyFalling' 100) stream01
True
```

Let us now consider the case in which the ball changes direction and bounces up when it hits the floor:

```
bouncingBall :: Double → Double → SF () (Double, Double)
bouncingBall p0 v0 = switch (fallingBall'' p0 v0 ⋙ (identity &&& hit))
                            (λ(p0', v0') → bouncingBall p0' (−v0'))
fallingBall'' :: Double → Double → SF () (Double, Double)
fallingBall'' p0 v0 = proc () → do
  v ← arr (v0+) ⋘ integral −≺ −9.8
  p ← arr (p0+) ⋘ integral −≺ v
  returnA −≺ (p, v)
hit :: SF (Double, Double) (Event (Double, Double))
hit = arr (λ(p0, v0) → if (p0 ⩽ 0 ∧ v0 < 0) then Event (p0, v0) else NoEvent)
```

The signal function *bouncingBall* starts with a ball falling down, until it hits the floor, and restarts the signal from the point of collision, inverting the direction of the velocity. The signal function *fallingBall''* is the same as *fallingBall*, except that it also takes an initial velocity and

outputs the current velocity. The signal function *hit* creates an event when the ball hits the floor, that is, when the position is not positive and the ball is going down.

If we were to translate the property *ballFellLower* to this new definition, we could assume that, with perfect elasticity and if the initial velocity is 0, the ball would never bounce higher than the initial position:

$$
\begin{aligned}
&ballLower :: Double \rightarrow TPred\ () \\
&ballLower\ p0 = Always\ (Prop\ (bouncingBall\ p0\ 0 \ggg arr\ (\lambda(p1, v1) \rightarrow p1 \leqslant p0)))
\end{aligned}
$$

If we now test this predicate with a stream, we see that it does not hold in our program. To give the ball enough time to bounce back up, we use a stream of 42 samples spaced by 0.5 seconds.

$$
\begin{aligned}
&> evalT\ (ballBouncingLower\ 100)\ stream05 \\
&False
\end{aligned}
$$

If we print the ball position at every time, we obtain:

$$
\begin{array}{llll}
[100.0, & 100.0, & 97.55, & 92.65 \\
, 85.3, & 75.5, & 63.25, & 48.55 \\
, 31.400000000000006, & 11.800000000000011, & -10.249999999999986, & 14.25000000000001 \\
, 36.30000000000001, & 55.900000000000006, & 73.05, & 87.74999999999999 \\
, 99.99999999999999, & 109.79999999999998, & 117.14999999999998, & 122.04999999999997 \\
, 124.49999999999996, & 124.49999999999996, & 122.04999999999997, & 117.14999999999996, \ldots
\end{array}
$$

The ball, dropped from 100 points with no vertical velocity, bounces up to 124.5 points. This is not the result of small floating-point inaccuracies or rounding errors, which could be accounted for by introducing a margin of error in the equality for floating-point numbers. Instead, it is caused by the errors introduced by the implementation of integral using the rectangle rule. In our simulation the Signal Function *integral* overestimates the result, and *integral0* underestimates it. Using *integral* the ball will bounce higher and higher every time. We could address this particular issue by providing a more accurate integral, or by manually capping *bouncingBall* to be lower than *p0*. This, however, is out of the scope of this dissertation.

Another problem with this simulation is that physics is not continuous, and the ball is not pulled out of the wall when it collides, so it will be temporarily rendered into the wall. This can be seen in the previous trace, with the ball position being -10.25 on the 11th sample. We test that the ball is always above the floor with:

```
ballOverFloor :: Double → TPred ()
ballOverFloor p0 = Always (Prop (bouncingBall p0 0 ⋙ arr (λ(p1, v1) → p1 ⩾ 0)))

> evalT (ballOverFloor 100) stream05
False
```

For reasons already explained, this property evaluates to *False*. For a proposal on how to address this problem by implementing continuous physics in Arrowized FRP, see the proposals for Continuous Collision Detection in Section 5.4.3 and [31].

8.3 QuickCheck Applied to FRP

In the previous subsection we saw how to express and test Temporal Properties of an FRP program. The input streams, however, were manually generated, which limits the coverage of our tests. For instance, if we test the predicate *ballOverFloor* dropping the ball from a slightly higher height, the property would not be violated for a stream *stream05′* with 21 samples spaced by 0.5 seconds, but it would be for one stream with 42 samples or one with 0.1-second delays instead:

```
> evalT (ballOverFloor 110.24999999999999) stream05′
True
> evalT (ballOverFloor 110.24999999999999) stream05
False
```

QuickCheck [114] is a tool that combines random data generation with a property language in order to test our code more thoroughly. QuickCheck testing is accomplished via a combination of a language to describe properties and a language to generate input data.

Properties are defined using combinators to express conditions on values. For example, a property stating that reversing a list twice leaves it unchanged could be defined and tested as follows:

```
propReverseTwice :: [Int] → Property
propReverseTwice xs = reverse (reverse xs) ≡ xs

> quickCheck propReverseTwice
+++ OK, passed 100 tests.
```

If we state an incorrect predicate, for instance, that *reverse* is the identity function on lists, QuickCheck finds and prints a counter-example that invalidates our assertion (in this case, the predicate is false for the input $[1, 0]$):

```
propReverseOnce :: [Int] → Property
propReverseOnce xs = reverse xs ≡ xs


> quickCheck propReverseOnce
*** Failed ! Falsifiable (after 3 tests and 1 shrink) :
[1, 0]
```

The confidence we can place on tests against randomly generated data depends on the nature of such data. We may want to constrain data to meet a function's precondition or mimic expected user input. Pre-conditions can be defined by means of *filters*, like the following, which states that the property defining how the functions *head* and *tail* relate to one another only makes sense for non-empty lists:

```
propHeadTail :: [Int] → Property
propHeadTail xs = not (null xs) ➞ (head xs) : (tail xs) ≡ xs
```

Filtering data can make the search inefficient when most of the data generated does not meet the preconditions. Additionally, randomly generated data may not explore the corner cases of our solution. To address these concerns, QuickCheck defines a language of generators, consisting of a series of types, classes and combinators operating on values that can be randomly generated.

In this section we demonstrate how to use the Temporal Language described in Subsection 8.2 to test temporal properties of FRP programs with QuickCheck. We do so by providing a series of input stream generators, together with combinators to constrain the kinds of streams generated. In Chapter 10 we demonstrate how this approach helped find bugs in real games.

8.3.1 Stream Generators

Generating suitable inputs for our tests requires that we provide random, but reasonable, input signal values and sampling times, constructing what we called a sample stream or *input stream*.

Our basic definition of input streams are signal samples paired with strictly positive time deltas. The data corresponding to each input is application-specific, but we can provide general-purpose generators to create random sampling times.

In order to generate lists of time samples, we use the following general function:

$$generateDeltas :: Distribution \rightarrow Range \rightarrow Length \rightarrow Gen\ DTime$$

where *Gen* is QuickCheck's type constructor for value generators. This function takes three parameters whose types are defined as follows:

$$\begin{aligned}
&\textbf{data}\ Distribution = DistConstant \mid DistNormal\ (DTime, DTime) \mid DistRandom \\
&\textbf{type}\ Range\ = (Maybe\ DTime, Maybe\ DTime) \\
&\textbf{type}\ Length = Maybe\ (Either\ Int\ DTime)
\end{aligned}$$

The type *Distribution* can be a constant function with a randomly generated time delta (which can be constrained or even fixed using the *Range* parameter), a normal (Gaussian) distribution with a given average and sigma coefficients, or a uniform distribution. The type *Range* describes possible lower and upper boundaries for the values generated. The type length describes the length of the stream, either in number of samples or in time length. To make code simpler, we provide the facility functions:

$$\begin{aligned}
generateStream\ &:: Arbitrary\ a \\
&\Rightarrow Distribution \rightarrow Range \rightarrow Length \rightarrow Gen\ (SignalSampleStream\ a) \\
generateStreamWith\ &:: Arbitrary\ a \\
&\Rightarrow (Int \rightarrow DTime \rightarrow Gen\ a) \rightarrow Distribution \rightarrow Range \rightarrow Length \\
&\rightarrow Gen\ (SignalSampleStream\ a)
\end{aligned}$$

which allow us to generate complete streams, either with purely random data or with an auxiliary generator that depends on the sample number and the absolute time. We provide additional facilities to make our code shorter:

$$\begin{aligned}
uniDistStream\ &:: Arbitrary\ a \Rightarrow Gen\ (SignalSampleStream\ a) \\
uniDistStream\ &= generateStreams\ DistRandom\ (Nothing, Nothing)\ Nothing \\
uniDistStreamMaxDT\ &:: Arbitrary\ a \Rightarrow DTime \rightarrow Gen\ (SampleStream\ a) \\
uniDistStream\ maxDT\ &= generateStreams\ DistRandom\ (Nothing, Just\ maxDT)\ Nothing \\
fixedDelayStream\ &:: Arbitrary\ a \Rightarrow DTime \rightarrow Gen\ (SignalSampleStream\ a) \\
fixedDelayStream\ dt\ &= generateStreams\ DistConstant\ (Just\ dt, Just\ dt)\ Nothing
\end{aligned}$$

8.3.2 Stream Combinators

The previous API lets us generate random streams, but in order to pre-feed existing data or express complex properties such as that a signal function behaves the same regardless of how often it is

sampled, we need additional ways to constrain streams. Recall that streams in our framework are defined as follows:

```
type SignalSampleStream a = (a, FutureSampleStream a)
type FutureSampleStream a = [(DTime, a)]
```

Streams can be concatenated, as well as merged, in which case we need an auxiliary function to decide what to do if the two streams provide a sample for the same time:

```
sConcat :: SignalSampleStream a → DTime → SignalSampleStream a
        → SignalSampleStream a
sMerge  :: (a → a → a) → SignalSampleStream a → SignalSampleStream a
        → SignalSampleStream a
```

We also provide clipping functions that allow us to drop samples before or after a specific time:

```
sClipAfterFrame  :: Int → SignalSampleStream a → SignalSampleStream a
sClipAfterTime   :: DTime → SignalSampleStream a → SignalSampleStream a
sClipBeforeFrame :: Int → SignalSampleStream a → SignalSampleStream a
sClipBeforeTime  :: DTime → SignalSampleStream a → SignalSampleStream a
```

8.3.3 Examples

We can define the QuickCheck property quantified over input streams as follows:

```
propTestBallOverFloor = forAll myStream (evalT (ballOverFloor 110.24999999999999))
  where
    myStream :: Gen (SignalSampleStream ())
    myStream = uniDistStream
```

QuickCheck finds a counter-example in only four tries:

```
> quickCheck propTestBallOverFloor
*** Failed ! Falsifiable (after 4 tests) :
((), [(4.437746115172792, ()), (1.079898766664353, ()), (3.0041170922472684, ())])
```

The counter-example generated by QuickCheck contains very large time deltas. In a realistic scenario, with a screen refresh rate of 60Hz standard on PC and phones, time deltas would approximate $0.016s$. We can test the previous property with this different generator and see how it behaves in ideal conditions:

```
propTestBallOverFloorFixed = forAll myStream (evalT (ballOverFloor 110.24999999999999))
  where
    myStream :: Gen (SignalSampleStream ())
    myStream = fixedDelayStream (1 / 60)
> quickCheck propTestBallOverFloorFixed
+++ OK, passed 100 tests.
```

The fact that this test passes is, however, a result of exploring few and small traces. Before, the ball was not bouncing until several seconds into the simulation, and with a time delta of barely 16ms, it takes several hundred samples to reach the floor. If we explore more cases and larger input streams, we again find situations in which the program misbehaves indicating a bigger problem with our physics simulation:

```
> quickCheckWith (stdArgs {maxSuccess = 1000, maxSize = 300}) propTestBallOverFloorFixed
*** Failed! Falsifiable (after 897 tests):
((), [(1.6666666666666666e-2, ()), (1.6666666666666666e-2, ()), (1.6666666666666666e-2, ()), ...
```

While the generator *fixedDelayStream* simulates ideal conditions, in realistic scenarios we cannot expect every frame to last exactly 1/60 seconds. In practice, we find it useful to have more control over how small or uniform delta times are, by means of $uniDistStreamMaxDT$, setting instead a maximum time delta and accepting that, if the game runs slower, the physics may not be as realistic.

We can use the given interface to generate sampling streams prefixed with real user input, and have QuickCheck generate random samples after a particular point in time when we suspect a bug may exist. The following generator completes a user's game run after 1 minute for approximately 10 additional seconds with 16ms delays:

```
tenAdditionalSecondsAfterMinute :: SignalSampleStream () → Gen (SignalSampleStream ())
tenAdditionalSecondsAfterMinute userStream = do
  qcStream ← generateStream (DistNormal (0.016, 0.002)) (Nothing, Nothing) (Right 10)
  let clippedUserStream = sClipAfterTime 60 userStream
  return (sConcat clippedUserStream 0.016 qcStream)
```

In Chapter 10 we will see how to use QuickCheck to test properties of more complex games, and explain how it helped find and address a fundamental problem in a real game.

8.3.4 Testing Abstract Properties

Our approach to purely functional reactive testing is also useful to test properties of FRP implementations, general properties of Signal Functions, such as statelessness, or check that the Temporal Logic is sound by testing tautologies.

We begin with a simple test of equality between two signal functions. This is useful for checking that an optimised implementation fulfils a specification:

```
alwaysEqual :: Eq b ⇒ SF a b → SF a b → TPred a
alwaysEqual sf1 sf2 = Always (Prop ((sf1 &&& sf2) ⋙ (arr (uncurry (≡)))))
propTestAlwaysEqual = forAll myStream (evalT (alwaysEqual (arr (**2)) (arr (^ 2))))
  where
    myStream :: Gen (SignalSampleStream Double)
    myStream = uniDistStream
```

We can use this generic predicate to test if two specific signals functions perform the same transformation:

```
> quickCheck propTestAlwaysEqual
+++ OK, passed 100 tests.
```

This approach can be used to test underlying properties of Yampa. For instance, we could check whether certain arrow laws [36, 37] hold for signal functions:

```
propArrowIdentity = forAll myStream (evalT (alwaysEqual identity (arr id)))
  where
    myStream :: Gen (SignalSampleStream Double)
    myStream = uniDistStream


> quickCheck propArrowIdentity
+++ OK, passed 100 tests.
```

The above property expresses, for a limited input domain, that the identity for arrows (*identity*) is always equal to lifting (*arr*) the identity function (*id*). Testing more complex laws requires that we generate, again, suitable random data. Consider, for instance, the following attempt at testing that composition with *identity* leaves a signal function unchanged:

```
propCompositionRightIdentity =
  forAll myStream (evalT (alwaysEqual (arr (**2) >>> identity) (arr (**2))))
    where
      myStream :: Gen (SignalSampleStream Double)
      myStream = uniDistStream


> quickCheck propCompositionRightIdentity
+++ OK, passed 100 tests.
```

This test is rather limited, since it only checks that a specific SF, *arr* (∗∗2), has a right identity for (⋙). We can use QuickCheck's function generators to quantify our predicate over functions from *Int* to *Int*:

```
propTestAlwaysEqualIntFunctions =
  forAll uniDistStream $ λs →
    forAll streamFunction $ λf →
      evalT (alwaysEqual (arr (apply f) >>> identity) (arr (apply f))) s
      where
        dataStream :: Gen (SignalSampleStream Double)
        dataStream = uniDistStream
        streamFunction :: Gen (Fun Int Int)
        streamFunction = arbitrary
```

This property holds in Yampa, as we would expect:

```
> quickCheckWith (stdArgs {maxSuccess = 1000, maxSize = 300})
                 propTestAlwaysEqualIntFunctions
+++ OK, passed 1000 tests.
```

8.4 Summary

We began this chapter by identifying a problem we sought to address: making tests for interactive applications fully reproducible. We observed that *IO* computations are hard to reproduce and, in the context of FRP, these drive both system input and clocking information. We identified an Arrowized FRP variant that, due to its complete separation between logic and *IO*, makes reproducing tests trivial.

In order to specify the properties of FRP systems precisely, we introduced a simple form of discrete-time temporal logic with until and strict Priorean operators. We then created a language of Temporal Predicates that followed the same design, built around Arrowized FRP's Signal Functions.

To evaluate these temporal predicates, we introduced a function that checks if they hold for a given stream of input samples and sampling times. We saw, by example, how this evaluation mechanism can be used to specify and check interesting game properties.

To facilitate generating suitable input streams and exploring the input space more effectively, we introduced a series of random data generators for the testing framework QuickCheck. We saw that simple API can be used to constrain the sampling times effectively and detect more bugs, and a stream manipulation API allows us to include pre-recorded data or create more sophisticated tests.

In the next chapter, we will see how this family of Arrowized FRP variants also facilitates debugging, by means of a combination of both a suitable FRP extension and a series of tools. In Chapter 10 we will see how this testing and debugging approach can be used to find subtle bugs in real games.

Chapter 9

Debugging

The previous facilities allow us to treat the program as a closed box and test its behaviour from the outside against a complete execution trace. This section introduces two facilities to debug programs as they run: temporal assertions, checked at runtime, and tools for analysing the progress of an FRP simulation. This will help us determine not only whether programs fail, but also exactly when and where.

9.1 Temporal Assertions

Some of the temporal constructs presented in previous sections require looking back arbitrarily far in history or into the future. Modalities like *always* and *eventually*, which look into the future, only become decidable once we reach the end of the input stream. Until that point, they are semi-decidable: *always* can be falsified, if we find that the condition does not hold at some point, while *eventually* can be verified, if we find out that the condition holds at some point.

Implementing future-time temporal operators would require a multi-valued logic for those cases in which the value is not yet defined. To work with booleans only, we must choose *causal* modalities we can implement efficiently, that is, without keeping in memory arbitrary amounts of past values. Non-causal temporal propositions, which we can express with the temporal language presented in the previous chapter, need to be transformed into causal ones, and asked with respect to a different point in time. For example, if one examines a limited trace with sampling times $[t_0, t_1, \ldots, t_n]$, that if a condition holds for all points greater or equal than t_0, then it also holds for all points earlier or equal to t_n.

9.1.1 Implementation in FRP

We introduce the type *SPred* as a boolean-carrying signal, which represents causal Temporal Predicates that can be defined as signal functions:

$$\textbf{type}\ SPred\ a = SF\ a\ Bool$$

Point-wise operators like *not* or *and* have straightforward implementations by lifting the existing Haskell implementation of those logical operators over booleans to the signal function level:

$$\begin{aligned}
notSF\ \ &sf\ \ &&= sf \ggg arr\ not \\
andSF\ \ &sf1\ sf2\ &&= (sf1\ \&\&\&\ sf2) \ggg arr\ (uncurry\ (\wedge)) \\
orSF\ \ &sf1\ sf2\ &&= (sf1\ \&\&\&\ sf2) \ggg arr\ (uncurry\ (\vee)) \\
implySF\ &sf1\ sf2\ &&= orSF\ sf2\ (notSF\ sf1)
\end{aligned}$$

Temporal modalities that refer to the past can be easily described using signal function combinators. We implement *history*[1], which checks a condition at every point and becomes *False* forever as soon as the internal condition becomes *False*, as follows.

$$\begin{aligned}
&history :: SPred\ a \to SPred\ a \\
&history\ sf = loopPre\ True\ \$\ \textbf{proc}\ (a, last) \to \\
&\quad b \leftarrow sf \multimap a \\
&\quad \textbf{let}\ cur = last \wedge b \\
&\quad returnA \multimap (cur, cur)
\end{aligned}$$

The FRP construct *loopPre* creates a loop over a Signal Function, feeding the second element of the output tuple back as an input in the next simulation step.

Similarly, we can define *ever*, which checks if a condition ever held[2].

$$\begin{aligned}
&ever :: SPred\ a \to SPred\ a \\
&ever\ sf = loopPre\ False\ \$\ \textbf{proc}\ (a, last) \to \\
&\quad b \leftarrow sf \multimap a \\
&\quad \textbf{let}\ cur = last \vee b \\
&\quad returnA \multimap (cur, cur)
\end{aligned}$$

[1]This is sometimes called *always* in Part-time Linear Temporal Logic. Using a different name here helps avoid confusion with the Future-time Linear Temporal Logic operator with the same name used in the previous chapter.

[2]This is sometimes called *eventually* in Part-time Linear Temporal Logic. Using a different name here helps avoid confusion with the Future-time Linear Temporal Logic operator with the same name used in the previous chapter.

We can also insert simple predicates into signal functions, for example:

```
ballAboveFloor :: SF () (Double, Bool)
ballAboveFloor = proc () → do
  ballPos ← bouncingBall ⤙ ()
  let aboveFloor = ballPos ⩾ 0
  returnA ⤙ (ballPos, aboveFloor)
```

In an implementation like Yampa, assertions like the one above produce booleans. In order to report these violations, for instance, to print them or to record them in a log, they need to be passed as output of the signal function, affecting the types of the function that used that signal function, and so on, all the way up to the top-level signal function. This changes all of our types, making this approach cumbersome, as debugging should not require major changes in the source code. Low-level workarounds with *Debug.Trace* are not portable to mobile platforms, and *unsafePerformIO* might hinder referential transparency across executions unless introduced with care.

In implementations that allow monads, like Monadic Stream Functions (Section 4), we could define a debugging monad that logs assertions that are violated and the times when that happens:

```
type AssertionId = String
type DebuggingMonadT = WriterT [(String, DTime)]
assert :: String → SF (DebuggingMonadT m) Bool ()
assert assertionId = proc (val) → do
  t ← localTime ⤙ ()
  let optionallyLog (t1, v1) = when v1 (tell (assertionId, t1))
  () ← liftS optionallyLog ⤙ (t, val)
  returnA ⤙ ()
```

While this also changes the types in our program, we only need to make this change once to introduce the debugging monad regardless of how many assertions we introduce. We could then use assertions as follows:

```
ballAboveFloorM :: SF (DebuggingMonadT m) () Double
ballAboveFloorM = proc () → do
  ballPos ← bouncingBall ⤙ ()
  () ← assert "Ball must always be above the floor" ⤙ (ballPos ⩾ 0)
  returnA ⤙ ballPos
```

9.2 Interactive FRP Debugger

Debugging games using the facilities provided so far is technically possible, but can be cumbersome. In many cases, visual inspection is needed to understand whether a problem has occurred and why. Game developers often rely on alpha- and beta-testers to test games.

With the approach embodied by Yampa, based on pure FRP, users can record the inputs and sampling times while they play and send them to developers indicating the time when a bug manifested. Developers can later replay these traces and move forward to that time, run additional tests, introduce assertions and visualise the problem with total reliability.

To facilitate this task, I have created an FRP debugger for Yampa. The system consists of two components: an extension to Yampa's *reactimate* function that allows controlling and recording simulations, and a Graphical User Interface that connects to running Yampa games via the network and allows controlling the debugger.

9.2.1 Local Debugger

I have extended Yampa's *reactimate* function with a communication channel to send messages to and receive commands from an external debugger. This extension also carries additional state, saving the history of all the inputs and sampling times, as well as simulation preferences.

At every game loop iteration, the program checks for incoming commands. Features supported by the system include saving the input trace to a file, loading or substituting input samples, communicating the contents and time of an input sample, pausing, stopping and playing the simulation, moving or skipping steps forwards and backwards, playing until a certain condition is met and indicating when assertions are violated.

The communication with the remote debugger takes place via two sockets: a synchronous one to receive commands and send responses, and an asynchronous one to notify interesting events to the remote debugger. The remote debugger listens on the asynchronous channel and, when it detects an event, or when instructed by the user, uses the synchronous socket to send commands and obtain results. This lets us implement the remote debugger in a reactive way, with less knowledge of the internals of how reactimation work, as described in [29].

The Yampa simulator runs locally on the device where we want to test the game, which can be a developer's computer, but also can be an iOS or Android device. In the case of Android, applications use the Foreign Function Interface of Haskell, C, C++ and Java, as some features are only available in Java and we have no direct access from Haskell. The Android application contains code to transmit the input trace, stored in a file in the SD card, back to developers when users report a bug.

9.2.2 Remote Debugger

The second component of the proposed system consists of a Graphical User Interface (Figure 9.1) that allows us to connect to, follow and control a game running remotely on an iOS device (i.e. iPhone or iPad), an Android phone or a computer. Apart from executing the commands provided by the extended *reactimate* function, this GUI enables two advanced use cases, described below.

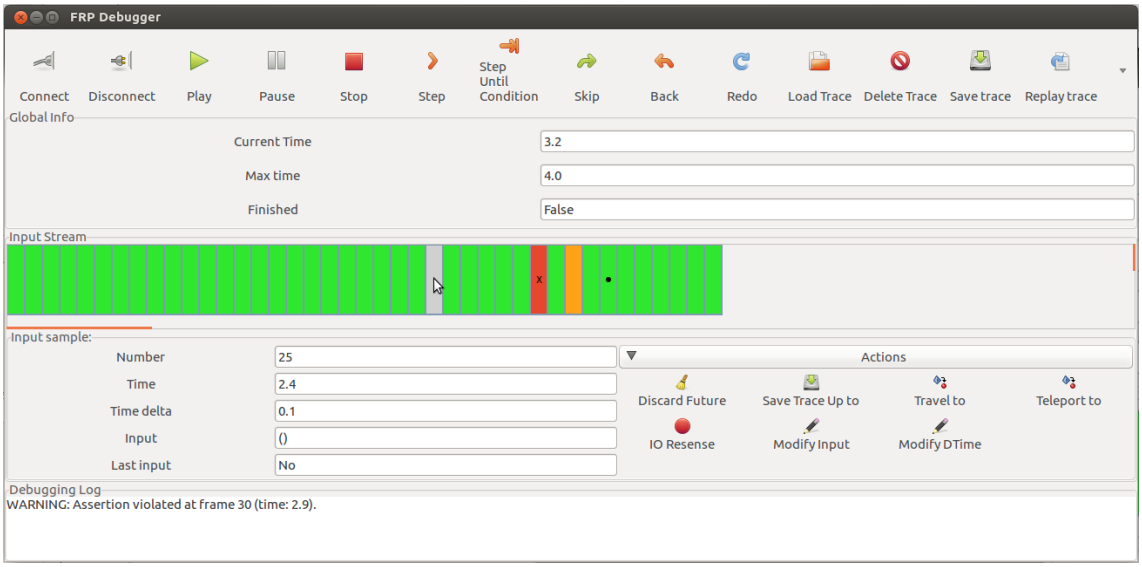


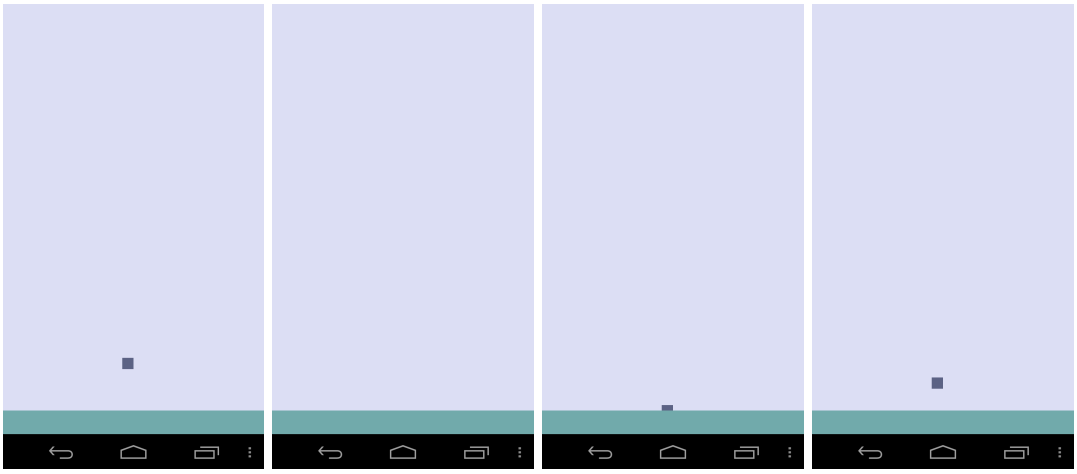
Figure 9.1: Screenshot of the interactive FRP debugger, with the central area showing a loaded input stream being used to execute a game on a mobile phone. The screenshot shows a sample in which a temporal assertion was violated (red), the current sample according to the phone (orange), a breakpoint (dot), and the sample selected by the user (grey).

First, because we can save and reload traces, and they contain all the information actually used by the FRP Signal Functions, developers can run the traces provided by users and visualise, with absolute guarantee, what the user saw, provided that the bug was not in the Input/Output layer of the game. Once a bug is detected, they can go back in time, step by step or as far back as desired, and find the points when assertions were violated. As the phone follows the debugging GUI as it moves along the trace, it will actually show the animation going backwards, forwards and jumping steps as instructed, which is an excellent visual aid for developers. Furthermore, developers can hot-swap the game, that is, make changes to the program, recompile it, restart it on the phone, and take it back to the same point in time to see if the bugs persist, all without having to close the GUI application.

Second, and aided by this first feature, we can take the user traces and feed them to QuickCheck

in order to find bugs. The stream manipulation API presented in Section 8.3 allows us to instruct QuickCheck to take part of an input stream from a real user and continue it randomly to try to find counter-examples for the game properties. As QuickCheck generates new input traces as counter-examples, we can save them in files, load them in the debugger and on the phone, and visualise the issue. So, in effect, we can see QuickCheck play.

As an example, let us show how we can use this approach to debug the input stream provided by QuickCheck invalidating the property *propTestBallOverFloor* (Section 8.3). If we connect an Android phone running this application with the debugger, we can use the GUI to load the counter-example input stream generated by QuickCheck and visualise on the phone the point at which an assertion fails (Figure 9.2).



(a) Frame #1: 0s. (b) Frame #3: 5.5176s. (c) Frame #4: 8.5218s. (d) Frame #5: 9.5218s.

Figure 9.2: Screenshots of an Android demo controlled remotely with the FRP Debugger GUI, executing the counter-example generated by QuickCheck, step-by-step. The ball is under the floor after 2 frames and takes 2 more frames to come back to the screen area. Frames #3 and #4 produce assertion violations while the ball is below the floor.

9.3 Summary

In this chapter we explored the topic of actually finding bugs in FRP applications. To that end, we explored two ideas. First, the specification of FRP assertions as boolean-carrying signal functions. Similarly to the temporal language seen in the previous chapter, this simple language allows us to monitor signal functions and detect when a condition stops holding. The second extension was an interactive debugging tool that allows us to record a game run, reproduce it, and connect to a

running application to move along the execution timeline. We have presented both the debugging simulation function that replaces the standard execution function in the game and an interactive GUI that allows us to control the simulation.

Together with the temporal language and the QuickCheck generators seen in the previous chapter, this presents a mechanism suitable to detect bugs in games, both in compile time and in runtime. In the next chapter we see how these techniques can be applied to find real, non-trivial bugs in an existing FRP game.

The claims in this and the previous chapter about testing and debugging being reproducible in this form of FRP rely on game traces containing all the information required to replicate the game state exactly. This is a consequence of the requirement that Signal Functions be free from any I / O, and is captured in the type signatures of Signal Functions. Although the debugging tools presented here do not currently work for the implementation Dunai, the claims apply to any pure form of Arrowized FRP, including those implemented on top of MSFs in Chapter 5.

The fact that we can rely on Haskell's purity and explicit, strong types to obtain the referential transparency needed to debug deterministically supports the idea that pure functional programming is a very good fit for developing many kinds of games.

Chapter 10

Experience

In previous chapters we have seen that pure Arrowized Functional Reactive Programming facilitates systematic testing and debugging. Both the testing and the debugging mechanisms proposed rely on defining temporal logic languages using FRP terms and giving an evaluation strategy for them.

The verification methodologies proposed have been used to analyse and find bugs in the FRP implementation Yampa. The combination of Temporal Logic, QuickCheck and our debugging GUI application has also been used to debug existing demonstration Yampa games like Haskanoid[1], and is currently in use in production for games written in Haskell for mobile platforms[2].

While the methodologies described in previous and this chapter are applied to FRP as embodied by Yampa, the conclusions apply in general to pure (*IO*-free) Arrowized FRP implementations based on Monadic Stream Functions as well. As Yampa and Yampa games can be implemented on top of Monadic Stream Functions, the tests proposed can also serve to verify that the implementation of Monadic Stream Functions Dunai and the FRP implementation BearRiver both satisfy the expected properties.

10.1 Yampa

A new suite of unit tests has been added to Yampa. These tests are based on Yampa's pre-existing battery of unit tests, which checked that arrow laws and other additional laws held for Signal Functions. Let us examine a few of those properties encoded using the new abstraction based on temporal logic, as well as the limitations that we find.

For reasons of efficiency, Yampa uses GADTs to distinguish between different kinds of Signal

[1] `http://github.com/ivanperez-keera/haskanoid`

[2] The development of the mobile extensions to the framework presented in this dissertation and their use to test these games have been carried out by Keera Studios Ltd., start-up founded by the author.

Functions. For example, to produce a constant value, it is more efficient to use the function $constant :: b \to SF\ a\ b$, but the expressions $arr\ (const\ a)$ and $constant\ a$ should behave the same way for all values b.

We might want to check that these equalities hold for any value, and that our implementation of signal functions using special cases is correct. We can do that with the following specification:

```
propConstEqualsArrowConst =
  forAll arbitrary $ λb →
  forAll myStream $ λstream →
  evalSF stream $
    Always $ Prop $ proc (_) → do
      b1 ← constant b     -≺ ()
      b2 ← arr (const b) -≺ ()
      return (b1 ≡ b2)
```

Thanks to QuickCheck, such properties provide better coverage of the input test space than the pre-existing tests, which had some of their inputs hard-coded in the test. Before, we would find hard-coded input streams in Yampa's tests, whereas now those tests pass random input streams with random deltas.

Nevertheless, there are multiple cases in which properties specified with the limited language we possess do not cover the input space we would desire. For example, one arrow law states, specialised to the case of signal functions, that for any signal function $sf :: SF\ a\ b$, the following holds:

$$sf \ggg arr\ id \equiv sf$$

However, because we have no built-in way of quantifying over all possible signal functions, we can only test this property against specific signal functions. It might be possible to create generators that produce arbitrary kinds of signal functions, which remains as future work.

10.2 Haskanoid

To demonstrate properties that QuickCheck can detect and how it has helped find bugs in games, let us examine examples from the existing game Haskanoid (Figure 10.1), a Haskell implementation of Arkanoid that runs on Windows, Linux, Mac, iOS, Android and can be controlled with devices like Kinects, Nintendo Wii remotes (Wiimotes) and accelerometers.

The player controls a paddle which moves only sideways. A ball is initially attached to the paddle. When the user clicks the mouse, the ball is propelled upwards, bouncing against walls,

Figure 10.1: Screenshot of the Yampa game Haskanoid running on an Android tablet.

blocks, and the paddle itself. Blocks disappear when hit three times. The player wins when all blocks disappear, and loses if the ball hits the floor.

The full game implements more complex features like levels, lives and sound. Here we discuss a simplified version to test the game physics and simple, yet important, game invariants.

10.2.1 Game Definition

Objects in Haskanoid are defined as signal functions that, depending on some input, present their new state and any effects that could affect other objects or their own existence (i.e. requests to create new objects or kill themselves). This idea is described in more detail in [40].

```
type ObjectSF = SF ObjectInput ObjectOutput

data ObjectInput = ObjectInput
  { userInput      :: Controller
  , collisions     :: Collisions
  , knownObjects   :: Objects
  }

data ObjectOutput = ObjectOutput
  { outputObject   :: Object
  , harakiri       :: Event ()
  }
```

Any game object can depend on the user input (*Controller*), previous collisions, and the state of other objects. Game objects produce an internal state with physical properties and indicate whether the object has died. This is used by blocks hit by the ball, to signal that they must be eliminated.

A simplified definition for objects, used for the ball, the paddle, blocks and the walls, follows. All 2D-suffixed types correspond to tuples of *Double*s.

```
data Object = Object
  { objectName      :: String
  , objectKind      :: ObjectKind
  , objectPos       :: Pos2D
  , objectVel       :: Vel2D
  , objectAcc       :: Acc2D
  , objectHit       :: Bool
  , objectDead      :: Bool
  , collisionEnergy :: Double
  }
```

10.2.2 Testing Physics Simulations

In this game, the position of the ball in the game's output can be extracted using a function called *ballPos*. In the following example, the top-level function *mainGame* takes user input (*Controller*) and returns a game state, which includes a collection of *ObjectOutput*s. To verify correctness, we want to know whether the ball can overlap with the wall, or pass through blocks exhibiting an effect called *tunnelling* or *bullet through paper* [99].

```
propBallInScreenAreaW =
  forAll myStream $ evalT $
    Always $ Prop $
      mainGame >>> arr gameObjects >>> arr (λp → isInRange (0, gameWidth) (fst (ballPos p)))
  where myStream :: Gen Controller
        myStream = uniDistStream
> quickCheck propBallInScreenAreaW
*** Failed ! Falsifiable (after 31 tests) :
(Controller { controllerPos = (8.49338899008, 29.07069391684), controllerClick = False },
[(17.68729810519, Controller { controllerPos = (28.12964084927, 12.4975777242)
                             , controllerClick = True })
```

```
, (29.88005602026, Controller { controllerPos = (51.41061779016, 3.0186952000)
                              , controllerClick = False })])
```

In this trace, QuickCheck is telling us that the input controller (e.g. mouse) needs to start at a particular position with the main button released, moved to another position with the button pressed, and moved elsewhere with the button released. However, the delays between one sample and the next are unrealistically large: one new frame every 15 seconds. To make it a bit more realistic we decide to use a stream generator that guarantees shorter delays. Let's assume that our game runs at around 25 FPS (*denseStream* is an auxiliary generator that creates streams with very small time deltas following a normal distribution):

```
propBallInScreenAreaW' = forAll myStream $ evalT $
  Always $ Prop $
    mainGame >>> arr gameObjects >>> arr (λp → isInRange (0, gameWidth) (fst (ballPos p)))
  where myStream :: Gen Controller
        myStream = denseStream 0.04
> quickCheck propBallInScreenAreaW'
+++ OK, passed 100 tests.
```

If we try this property with even higher frame rates (lower deltas), all tests initially pass. However, this apparent correctness is only an illusion. If, instead of increasing the number of deltas, we choose to increase the number of tests, we start seeing that not even 0.04 is low enough:

```
> quickCheckWith (stdArgs {maxSuccess = 100000}) propBallInScreenAreaW'
*** Failed ! Falsifiable (after 3443 tests) : ...
```

There is one artificial way in which we can prevent the ball from going out of the screen: with a speed cap. If we cap the maximum speed and introduce a margin of error in the detection of collisions, then the collision with the ball against any wall will be detected before they overlap.

Nevertheless, this artificial cap can easily be tested by checking the magnitude of the ball's velocity. This test should work regardless of the sampling rate and detecting that the ball moves too fast (one condition) is easier than detecting that the ball is out of the screen, because it was moving too fast (two conditions).

Introducing such a test in this game was particularly useful, as it showed that the velocity was not being capped right after launching it or after collisions, which implied that, if the paddle hit the ball with a quick, sudden movement, it could temporarily "insert" it into the wall. Before

adding the second speed cap, it was still possible to play the game in a way that the velocity would be too fast:

```
propVelInRange = forAll myStream $ evalT $
  Always $ Prop $
    mainGame >>> arr gameObjects >>> arr (λp → isInRange (0, 800) (norm (ballVel p)))
  where myStream :: Gen Controller
        myStream = denseStream 0.004
> quickCheck propVelInRange
*** Failed ! Falsifiable (after 67 tests) :
(Controller { controllerPos = (130.5942684124, 40.2324487584), controllerClick = False },
[(3.9907922435e−3, Controller { controllerPos = (61.5220268505, 108.729668432)
                              , controllerClick = True }),
 (2.1598144494e−3, Controller { controllerPos = (66.5487565898, 50.682566812)
                              , controllerClick = False })])
```

In this trace, the mouse is moved quickly to the left and clicked, propelling the ball with a velocity of thousands of pixels per second. By adding the cap in other places in the code, we guarantee that the velocity is within the expected range, making the ball more likely to remain within the game area even with lower sampling rates. This solution is, unfortunately, ad hoc and not reusable, and it might not behave properly in other situations. A general solution for this problem, called *tunnelling*, is to implement a Continuous Collision Detection system (CCD) [99].

10.3 Summary

In this chapter we have examined the use of the facilities discussed in previous chapters for reliable systematic testing and debugging applied to both an FRP implementation and an FRP game.

We have seen that the predicated building constructs introduced in previous chapters can be used to test general properties of Signal Functions, like the arrow laws, domain specific properties of Signal Functions, like certain equational laws, and to check that optimisations in the implementation conform to their expected definition.

We have seen how, in spite of providing better coverage of the input space, the given building blocks are not enough to provide full coverage to verify our implementation for all possible kinds of signal functions, and further work is needed in this direction.

We have also seen that the mechanisms described have been useful to discover and address bugs in an existing open source FRP game. The testing framework has been useful to capture

interesting and non-trivial game properties, and the combination with QuickCheck helped discover and visualise specific game conditions for which the properties were violated, and modify the game configuration accordingly.

Part IV

Conclusions

Chapter 11

Related Work

In this work we have presented Monadic Stream Functions, an extensible minimal abstraction for Functional Reactive Programming. We have used it to implement fully fledged FRP and Reactive Programming (RP) libraries, and demonstrated that it subsumes and exceeds existing implementations, overcoming many of their limitations. We have used MSFs also to implement Reactive Values and Relations, a reactive programming abstraction tailored towards GUI applications. While catering for additional side effects, MSFs do not limit our ability to formally analyse their behaviour and do equational reasoning, which we have used to prove mathematical properties of MSFs. We have also seen that pure Arrowized FRP, as realised by Monadic Stream Functions and by Yampa, facilitates advanced forms of purely functional testing, which we have demonstrated with a temporal testing abstraction and debugging framework specific for functional reactive applications.

This chapter presents a broader view of the field, first, to identify how the proposals in this thesis compare to existing work, and, second, to understand their impact and meaning in a wider context.

11.1 MSFs

MSFs are a generalisation of Yampa's signal functions [27]. One central difference is that Yampa has a fixed, notionally continuous time domain, with (broadly) all active signal functions sensing the same time flow. This is implemented by passing the time delta since the previous step to each signal function. With MSFs, we can do the same using the *Reader* monad, but the time domain is no longer fixed, and it is easy to arrange for nested subsystems with different time domains and other variations as needed. Our proposal also eliminates some of the bottlenecks of Yampa thanks to the flexibility offered by the monadic parametrisation.

I have tested several Yampa games with the proposed library BearRiver, including games like Haskanoid [103] and the commercial game Magic Cookies for iOS and Android [102, 101]. Tests show that BearRiver runs these games in near constant memory, consuming about ten per cent more memory than Yampa. Yampa is optimised exploiting algebraic identities like the arrow laws [36, 115], suggesting that we could obtain comparative performance gains with similar optimisations.

Yampa and other FRP implementations introduce different mechanisms for dynamism, including switches and dynamic collections [40] (collections of signals that new elements can be added to or removed from during execution). We have demonstrated that, using *Either* or *ListT*, similar behaviour can be implemented. This kind of structural changes may require network optimisations and determining the value of the signal at the time of change/switching. This question was explored by Nilsson et al. [27, p. 54], leading to two forms of switching: instantaneous switching and decoupled switching. Instantaneous switching may result in nested switches and, in the presence of recursion, in infinite loops [40, p. 8]. Implementations like Yampa provide delays and initialisation combinators to break looped switching[1], while others like Elerea [98] introduce these delays automatically in the presence of *circular dependencies*.

Netwire [116], inspired by Yampa, is an FRP framework parametric over the time domain and a monad, in which signals can be inhibited, producing no output, and with a switching mechanism that maintains the global clock. In contrast, Yampa has only local notions of relative time, and signals started due to a switch will perceive their local time to be zero at the time of switching. We can implement in Dunai behaviour equivalent to that of Netwire by using *EitherT* to inhibit signals, and adding the total global time to a *ReaderT* context.

Causal Commutative Arrows (CCA) [87] defines an abstraction of *causal stream transformer*, isomorphic to *MSF*s over the *Identity* monad. Programs written in terms of causal stream transformers, or any CCA, can be optimised by orders of magnitude in speed. MSFs are valid CCAs for commutative monads [117], making it possible, in principle, to apply similar optimisations.

Multiple FRP libraries implement push or push/pull evaluation, such as Elerea [98], Sodium[2] and reactive banana [84], all structured in terms of signals and/or event streams. These libraries are optimised to run efficiently with minimal data propagation, often using weak references and an *IO* monad behind the scenes. Additional measures must be taken to guarantee referential transparency and efficient garbage collection in the presence of non-commutative monads like IO. These frameworks also include combinators to connect signals or event streams to external sources and sinks, which also requires *IO*. This is a pragmatic choice, but the drawback is that the *IO*

[1] In the case of Yampa, the problem manifests itself also in delayed switching, due to the use of *transitions* or *future signal functions* and strict evaluation.

[2] `https://hackage.haskell.org/package/sodium`

connection is not shown explicitly in the type. The manifest reflection of all effects in the types is a strength of the proposed abstraction framework. For example, for pure MSFs without *IO* we can perform reproducible tests using automated testing tools like QuickCheck [118], and obtain more guarantees than in *IO*-based FRP implementations.

Elerea also provides different mechanisms for structural dynamism, based on higher-order signals, which carry other signals, and can be "flattened" with given combinators [98], turning on the internal signals as the output at specific times. While it would be possible to implement a flattening combinator in Dunai, this remains as future work.

Monadic FRP [119] is, despite its name, most similar to Yampa's Tasks[3], which we can model using MSFs using the *EitherT* transformer. Monadic FRP is not parametric over the monad, which makes MSFs more general in that respect.

FRPNow! [79] is a FRP library that allows signals to limit their dependence on past values ("forget the past"). Signals in FRP may depend on external computations, which are scheduled for asynchronous execution and whose value becomes available at some time in the future. FRPNow! enforces a separation between IO and conceptually pure FRP signals, by defining a monad in which all I/O computations occur. However, the dependence or introduction of this monad explicitly, both in the types of behaviours/signals and inside the definition of the *Now* monad, makes the framework, at least in part, less versatile than our *MSFs*, as it cannot capture cases in which the monad is not *IO*. This also makes FRPNow! not support the kind of referential transparency across executions that MSFs and pure Arrowized FRP allow.

O-FRP [120] uses a task-model of signals that allows them to be turned off, giving them temporary lifetimes. Like Monadic FRP, this is similar to Yampa's Tasks and similar behaviour can be achieved in the proposed framework using a *Maybe* or *Either* monad.

UniTi [121] is a hybrid-system simulation framework with local clocks in which signal functions can output debugging information. Monadic Stream Functions can accommodate local clocks using a *Reader* or *State* monad, as well as debugging facilities. Additionally, MSFs enforce causality and can enforce temporal consistency, whereas UniTi can provide inconsistent results for past values of signals.

There are also stream-based programming libraries that share notions with MSFs. Iteratees [122] are stream transformers parametrised over a monad and oriented towards memory-efficient processing of data streams gathered from network and file sources. The application domain is thus rather different from ours, and this is reflected in an asynchronous API with a somewhat imperative feel, centred around reading and writing individual stream elements. Pipes [123] describe interconnected data processors and their coordination in an asynchronous setting. At a

[3]Yampa's Tasks represent signal functions that can terminate with a result.

higher-level, pipes are expressed in terms of constructs like *Producer* and *Pipe*, conceptually similar to MSFs with an *EitherT* monad transformer. Internally, pipes are based on a *Proxy* type which is parametrised dually on the input and the output, making pipes bi-directional. MSFs can be used to describe bi-directional connections with a modified execution function (Section 5.1).

Wormholes [94] pairs white holes (producers) with black holes (consumers) to represent external resources. Reads and writes are sorted to guarantee referential transparency at signal level and commutativity. This could be achieved with custom monads in our setting, as we saw with monadic streams, sinks and Reactive Values. While Wormwholes use a Monadic Stream Function representation to introduce I/O, their types carry resource information, imposing additional constraints that require a customised Arrow type-class.

Hughes uses the same representation as MSFs in a circuit simulator [42]. The approach is stream based, centred around data processors that actively wait for input events. Hughes does not explore the use of different pure monads or how they impact modularity. However, the work shows how to leverage active waits on the monad, something which is also applicable in our setting.

Varying[4] uses a basic construct isomorphic to Monadic Stream Functions. Varying is mainly focused on reactivity and stream processing, as opposed to our focus on time transformations and the combination of multiple monads.

A representation similar to MSFs was mentioned in [124], where the author briefly suggests using different monads to achieve different effects. To the best of my knowledge, that blog post did not spawn further work exploring that possibility.

Formal reasoning and semantics MSFs have strong mathematical foundations, which we used to prove arrow laws of Monadic Stream Functions and other properties. However, we have not demonstrated that the implementation of FRP on top of MSFs conforms to its intended formal definition. Also, giving MSFs precise denotational semantics might allow verifying that our definition of FRP conforms to its denotational semantics, as well as allow us to give Reactive Values precise meaning.

FRP was given denotational semantics by Elliott and Hudak [25], to "facilitate and guide implementations", and Courtney [125]. However, these semantics can only be approximated, and there are discrepancies between implementations and their ideal meaning: some constructions cannot be implemented accurately, and some constructs in the implementation diverge in the limit as the sampling rate increases. Implementations like reactive-banana [84] limit what can be expressed to avoid some of those discrepancies.

A branch of research has focused on giving FRP precise (operational) semantics to reason about

[4] `http://hackage.haskell.org/package/varying`

output values and resources. RT-FRP [126] and E-FRP [127] did so by reducing the language to a minimal core, adding static types and minimal extensions. Priority-based FRP [128] built on this work by adding priorities, providing an implementation suitable for hardware analysis and real-time systems. Krishnaswami [129, 130, 131] defines a stream-based FRP language with higher-order and temporal operations, constrained to reason about resources and to prove freedom from space leaks[5].

Not all implementations have formal semantics. Concurrent variants, especially GUI-oriented ones [132, 133], pose difficulty reasoning about programs and exhibit non-determinism, relying on *eventual consistency* [134], like we did with Reactive Values. This results in *glitches* [77, p. 6], temporal incoherences between FRP invariants and the network state. These implementations go to great lengths to break infinite loops caused by circular dependencies [80, p. 10].

Verification A number of dataflow and stream programming languages have been used to describe synchronous reactive systems, with a specific focus on critical environments. These languages have properties that facilitate static analysis and can provide real-time guarantees.

Lustre [135] is a synchronous data-flow programming language that has been used in aerospace, transportation, nuclear power plants and wind turbines, among many others. Lustre programs are defined in terms of *flows* (streams) and *nodes* (causal stream functions). Lustre's type system includes both information about the amount of history examined of each flow examined by each node, and clocking rates at which each flow is being produced. Together with Lustre's clock calculus, this helps guarantee that all flows are well-formed (always defined). MSFs do not have any notion of clocks or clocking rate, making it, in principle, harder to capture those constraints and ensure the same kinds of guarantees. MSFs are defined and combined using a series of combinators that ensure that all values are well-typed. Nevertheless, our language is embedded in Haskell, which allows representing bottoms, so a program written using MSFs may not be guaranteed to be productive (simply because the programmer may have used non-terminating Haskell expressions).

Lucid Synchrone [136] is a programming language inspired by Lustre, which extends it with high-level concepts from functional programming, a richer type language, and a more versatile and usable clock system. Types in Lucid Synchrone are polymorphic, and the type system supports type inference. This approach is closer to our MSFs, due to the richness of the language, although MSFs do not explicitly support clocks or include any kind of clock calculus. Like in the case of Lustre, the language of MSFs is bigger than that of Lucid Synchrone nodes: constructions that would be rejected by Lucid Synchrone's compiler might be accepted by the Haskell compiler if described using MSF. This makes these languages, in principle, safer than full MSFs, unless the

[5] [129] gives semantics of FRP based on ultrametric spaces, [130] gives a language based on linear types, and [130] gives one without linear types, motivated by trying to write more abstract code.

latter is constrained in some additional way.

Esterel [137] is a synchronous data-flow programming language with support for concurrency and signal inhibition. Esterel rejects programs that give multiple values to the same signal at the same sampling time. Our framework support classic FRP and, in principle, signals are defined once for all time. Signals in Esterel can be broadcasted across the whole program from any point. This form of broadcasting might be implementable with MSFs by means of a *State* monad. However, Esterel provides additional static guarantees, for example, that a broadcasted signal only has one value per cycle, and this could only be detectable during runtime with the approach based on MSF's mentioned above.

Zelus [138] is a programming language for hybrid systems that combines discrete sequential execution with a solver of Ordinary Differential Equations (ODEs), used to detect conditions that depend on continuous signals. Zelus' type system is multi-kinded, and it guarantees that discontinuities only occur on discrete-time signals. Zelus also performs causality analysis to ensure that mutually-dependent definitions are well-founded and do not form instantaneous loops. In contrast, MSFs are inherently discrete. While MSFs can simulate continuous-time FRP and even limited forms of Continuous Collision Detection, the type system does not guarantee evaluation of continuous-time signals at all points of discontinuity. Also, with the optional support for Arrow loop, one can define MSFs that create mutually dependent signals, making it impossible to detect and reject ill-defined signals statically without additional extensions.

Signal kindedness Not all applications have the same requirements over signals: some change continuously, others change only sporadically. In general terms, we can distinguish between three kinds of signals: continuous-time continuously changing (aka behaviours), continuous-time sparsely changing (analogous to step-functions), and discrete-time signals (aka events or event streams, defined only at discrete points in time). Not all FRP variants use all three categories, which motivates a distinction between *Single-kinded* and *Multi-kinded* FRP. Fran [25], SOE FRP [139] and Reactive-banana [84] are multi-kinded, while Yampa [27], Elerea [98] and Flapjax [140] are uni-kinded. Event-based GUI toolkits have a closer correspondence with FRP variants with a notion of events [140, 132, 141, 142][6]. Games are most often implemented in variants with continuous time [40, 143, 103, 144].

Further variations are possible. N-ary FRP [145] extended the signal definition to make signals of tuples equivalent to tuples of signals. To the best of my knowledge, it is the only FRP implementation to have exploited that isomorphism, for which dependent types were used.

[6]This similarity is progressively disappearing, since in new GUIs, animations –conceptually continuous– play a central role.

Like Yampa, MSFs are, in principle, single-kinded, but they are general enough to accommodate FRP variants irrespective of their kindedness.

Evaluation model FRP systems can be evaluated in two main ways: by sampling (output) signals as frequently as possible, known as *pull* evaluation, or by propagating changes forward, known as *push* evaluation. Pull evaluation may result in unnecessary recalculations if signals have not changed [146]. Push evaluation may trigger too many re-calculations when signals change frequently. Pull evaluation is a good fit for games [40, 144, 143] and physics simulations, while push evaluation is frequently used with event-based GUI toolkits [141, p. 2].

Several authors have tried to combine both evaluation strategies by distinguishing signals based on their range of existence and rate of change (behaviours, events, and others). Push-pull FRP variants [147, 148] use this approach to only recompute continuous signals, but not events, as frequently as possible. A similar approach is used by Frappé [141], which is, in principle, push-based, but, in the presence of time-dependent signals, the network polled at regular intervals.

While Monadic Stream Functions are, in principle, synchronous and pull-based, we have shown that they can be used to implement asynchronous pull-based evaluation (Section 5.4) and asynchronous push-based evaluation (Chapter 7).

Visualisation and connection to external systems While the connection to GUI toolkits and multimedia libraries is common in FRP, most libraries are better suited for one kind of interactivity framework or the other depending on their evaluation model. Connections to GUI toolkits are present in FRP variants like Frappé (Java Beans, [141]), reactive-banana (HTML, wxWidgets, [84]), grapefruit (wxWidgets, [149]), KSWorld [150] and wxFruit [151]. Low-level connections to graphics libraries are used in Yampa (SDL [152, 153], Glut [154], Fruit [26], HGL [155] and OpenGL [46]), Fran (Win32 [156]), Reactive (FieldTrip [157]), and Elerea (OpenGL and LambdaCube3D [158]). In contrast, we have shown that MSFs can be used easily with both GUI toolkits and low-level visualisation backends.

Time Transformations Driven by the needs of multimedia applications, many FRP variants are defined over continuous time [25, 27, 121]. However, a view of time as discrete unit steps is sometimes sufficient [159] and renders simpler solutions [98, 160]. Our framework is flexible enough to allow both continuous and discrete clocks, as well as giving MSFs local control over the sampling rate and the sampling step.

The minimal requirement to guarantee signal causality in FRP is that *time be ordered* [25, 147], with evaluation happening at strictly increasing times. Some FRP formalisms add, conceptually, infinity and negative infinity to the time domain, which simplifies some equations [25, p. 4].

The original implementation of Fran [25] allowed time transformations. However, a direct implementation of signals as functions from time to value opens the door to causality violations and memory leaks [26]. Efficiency has been a major concern in FRP implementations since the very beginning [161, 147], and it is strongly tied to the semantics of the language. Our approach, which operates on time deltas instead of absolute time, and uses time-consistent constructs with limited history, avoids leaks and maintains causality, in the same way that Arrowized FRP does.

UniTi [121] signals can be also sampled at any time, even before the conceptual beginning of the simulation. The implementation of signals as functions renders problems similar to the original Fran, while an arrowised implementation based on MSFs provides similar versatility while limiting time and space leaks. Like MSFs, UniTi provides control of the sampling rate and local clocks.

Netwire [116] also uses continuations to implement transformations to input values. Being parameterised over a monad, wires can be used to implement FRP much like Dunai does. However, Netwire does not allow, in principle, running wires backwards in time, and no constructs are provided to guarantee temporal consistency when time flows towards the past.

Implementations like Elerea [98], Sodium[7], reactive-banana [84], as well as the work of Krishnaswami [162], all operate in a step-based fashion with time being a natural number. In these variants the sampling time is the index of a sample in an input stream, and time deltas are of one unit of time. FRPNow! [79] introduces a more complex definition of clock, but progression towards the future occurs in discrete steps and time deltas are of one unit. All these implementations have the limitation that time flows necessarily forward, and there are no ways of progressing towards the past.

Games like Braid [163] implement time travel and time reversal. To guarantee that past states are numerically identical when time is reversed, Braid records the game state at every sample, instead of using reversible signal functions or saving only some samples and interpolating intermediate steps[8].

The synchronisation of multiple systems through an explicit notion of clocks is present in Cλash [164], Lucid Synchrone [136], recent work by Bärenz [165], and other dataflow languages. In contrast, MSFs aim for simplicity for the user, by keeping this notion out of the type level. Furthermore, the use of a completely IO-free framework to synchronise systems running at different rates enables referentially transparent testing, as described in Part III.

[7] `https://hackage.haskell.org/package/sodium`

[8] `http://www.gdcvault.com/play/1012210/`

11.2 Reactive Values

Comparison to Functional GUIs and FRP Fudgets [22] is a functional GUI framework structured around the notion of *fudgets*: visual, interactive data transformers with one input and one output. Fudgets was reviewed in Chapter 3. Limitations of Fudgets include not supporting connection of visually non-adjacent widgets and that mutually interconnected fudgets must be defined together. Reactive Values overcome such issues by using one RV per widget property and by allowing separately defined RVs definitions to be related through Reactive Relations (RRs). Gadgets [23] is similar to Fudgets, but tries to overcome some of its limitations. However, as discussed in Chapter 3, by their nature, both Fudgets and Gadgets need to provide a fudget/gadget definition for every single GUI widget of interest, meaning that such libraries necessarily become very large. This leads to high maintenance costs, which is one reason Fudgets is only available on a selection of Unix-like platforms and has not seen any major update since the late 1990s. In contrast, RVs and RRs have a much smaller footprint and are designed to work in conjunction with existing GUI toolkits on any platform, thus side-stepping this issue.

A key difference between traditional Functional Reactive Programming (FRP) [25, 26] and RVs is that the latter allows separately defined reactive entities to be related, while an FRP signal is defined in terms of the signals it depends on once and for all. As discussed in Chapter 7, this aspect of FRP often leads to scalability issues in large applications, in particular for mutually recursive signals. We have shown that this can be overcome at a lower level in FRP with the introduction of controlled side effects, like we did with Monadic Stream Functions. Unlike FRP, RVs are agnostic about time, thus not lending itself to reasoning about temporal properties. Soft real-time guarantees have been studied for at least some FRP variants [128].

Some FRP implementations, like Elerea [80], take special precautions to break change propagation loops. We use equality tests in setters to minimise change propagation. For loops, this means that propagation stops when reaching a fixed point. It is thus crucial that the functions provided for transforming read-write RVs are each other's inverses, or propagation could go on indefinitely. Reactive Values and Reactive Rules do not provide further guarantees, but specifying both directions of the transformation in a single place may facilitate discovering bugs quickly.

RVs are asynchronous and concurrent. Like Netwire [116], RVs rely on the fact that applications will eventually come to a balance, known as *eventual consistency* [166, 134], This results in *glitches* [77, p. 6], temporal incoherences between FRP invariants and the network state (Figure 11.1). This contrasts with deterministic FRP variants that make use of parallelism, like Parallel FRP [167] and Reactive [147].

There are some similarities between RVs and iTask's [71] *Uniform Data Sources* (UDS) [168],

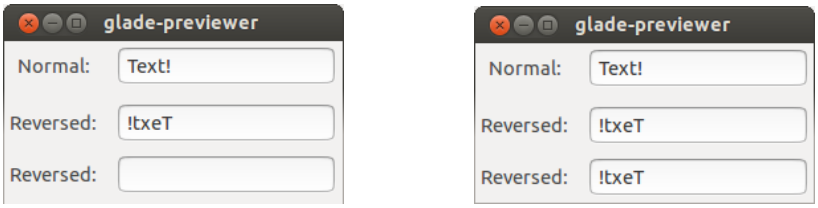


Figure 11.1: Illustration of possible temporal inconsistencies during FRP signal stabilisation. Some asynchronous implementations rely on *eventual consistency*, resulting in *glitches* (short-lived inconsistent states). In this application the text in the first text box should be immediately shown reversed twice. There is a short delay (left) until the state becomes consistent (right).

but UDS has no support for subscription to change notification. Further, a central aspect of iTask is automatic generation of GUIs from types with a particular focus on Web applications, whereas RVs and RRs provide generic, re-usable infrastructure for GUI programming and more.

There are also similarities to Lenses [169]. However, RVs are not lenses as they in general do not satisfy any lens-like laws. RVs satisfy similar laws in the absence of concurrent modification for pure models, but this is not true for GUI-backed RVs or external files. Formalizing RVs using Monadic Stream Functions could help express a temporal version of similar laws. Nevertheless, RVs can beneficially be used together with lenses and we view them as complementary. Monadic lenses applied to User Interfaces[9] and lenses with notifications [170] could help simplify our formalisation to its true core objective: a data-binding language between typed reactive elements.

Parametric Views [170] are based on the same basic operations (get, put, subscription) as RVs. Further, like Parametric Views, Protected Models make *change* a first-class entity to minimise data propagation and screen refreshes. One difference is that RV setters compare previous and new values when possible to minimise change propagation and break propagation loops. Parametric Views provides a versatile notion of invalidation function to that end. My goal, however, is to make change detection as transparent as possible for which I am experimenting with automatically deriving change definitions for Haskell datatypes [24].

Comparison with Reactive Programming From an Imperative or Object-Oriented perspective, RVs are closest in spirit to Reactive Programming[10], and then in particular to *change subscription* facilities and *data binding* languages.

Reactive Values are similar to widget properties in Qt[11], which are typed mutable attributes

[9] `http://people.inf.elte.hu/divip/LGtk/LGtk.html`
[10] `https://github.com/Netflix/RxJava`
[11] `http://qtproject.org`

paired with a change event. Qt's signals and slots can be seen as read-only and write-only RVs and are versatile enough to accommodate files, sockets and other external entities. Qt further provides data binding facilities to connect signals to slots, but unlike with Reactive Values and Reactive Rules, these are *uni-directional*, and there is no mechanism for breaking propagation loops. When using Qt as a GUI backend from our framework, we provide an intermediate library that takes care of change detection and thread safety, shielding users from such details.

RV's notification system is similar to the observer design pattern [171] frequently encountered in object-oriented programming. This pattern has specific support in recent versions of JavaScript in the form of *Object.observe* () [172]. The observer pattern enables detecting changes to objects, but it is necessary to install change handlers. This leads to issues of inversion of control common in event-driven programming [10, p. 36–37], and the scheme is further inherently uni-directional, unlike (bi-directional) Reactive Relations.

Facebook's React[12] has similar goals to the observer pattern but is more declarative. Unlike Reactive Values, React only provides uni-directional data-binding. Like RVs, React uses change detection mechanisms to minimise data propagation, which in the case of Web sites produces minimal DOM migrations. React gathers change propagation responsibilities to a central dispatcher in an attempt to maximise throughput. In contrast, the proposal in this thesis opts for a middle ground, using a global dispatcher per Protected Model, but allowing different Protected Models to co-exist and even coordinate during execution.

Reactive Relations constitute a data dependency language not unlike the data-binding facilities of frameworks like AngularJS[13] and EmberJS[14]. There are, however, structural differences. AngularJS, for instance, merges data-binding, function lifting, and view declaration into a single, annotated XML tree. RVs result in a, possibly, more modular and abstract design, partly because it maximises separation of concerns, and partly because it allows factoring choreographies out into libraries. As we have discussed, RVs use equality tests to minimise change propagation and break loops. This approach is typically more efficient than the *dirty-checking* used in AngularJS, but further research is needed to determine how RVs compare to the aforementioned frameworks in terms of performance.

11.3 Testing

The link between FRP and Temporal Logic has been studied extensively. Jeffrey [173] describes how LTL can be used as a type system for FRP, making any well-typed FRP program a proof

[12] `http://facebook.github.io/react/`
[13] `http://angularjs.org/`
[14] `http://emberjs.com`

of an LTL tautology. Jeffrey [174] also combines LTL with FRP using dependent types to define streams of types, which themselves type reactive programs and form a constructive temporal logic, exploiting the Curry-Howard isomorphism between TL and FRP. The author goes on to define a combination of FRP with past-time LTL, in which combinators like "always" mean "so far", making all modalities executable and causal.

Jeltsch [175] has also described the Curry-Howard isomorphism between Temporal Logic and FRP, in which FRP programs implement Temporal Logic propositions. The author shows that behaviours and events can be generalised in terms of the Until operator from LTL [176]. He then gives categorical semantics for LTL with until and FRP with behaviours and events. This work is used to define Concrete Process Categories (CPCs), which serve as categorical models of FRP with processes. The author defines a new semantics, a new temporal logic for CPCs, which captures causality.

Sculthorpe [96] has described an encoding of Priorean Temporal Logic in a denotational model used to describe FRP signals. In his approach, properties of both the time domain and FRP signals and signal functions can be described as temporal predicates, that may or may not hold, depending on the time domain. Sculthorpe also shows how properties of signal functions, like being temporally decoupled or stateless may be captured using temporal logic. There are several differences between Sculthorpe's approach and the one we explored in this thesis. First, we use a different kind of time, which is bounded because our simulations can actually end. Also, because we are interested in executing or checking these properties live, we cannot depend on the past or the future in the way that Sculthorpe's semantics does. Dependencies on the past lead to leaks, dependencies on the future cannot be determined in the present (which is how Yampa is evaluated).

The link between TL and FRP has also been discussed by Winitzki [177], who explains the meaning of modalities in Temporal Logic with discrete unbounded time in terms of streams and samples. The author also lists desired features to implement Temporal Logic currently unavailable in existing FRP implementations.

The use of Temporal Logic to specify properties of reactive systems is frequent in other domains, and multiple Temporal Logics with different properties have been used in different contexts. Design dimensions that are directly related to our domain of interest are the direction of time (towards the future or towards the past), whether time is discrete or continuous, bound or unbound, and the nature of signals themselves (continuous, step-wise, discrete, and how frequently they can change).

The logic presented earlier in Chapter 8 is based on LTL [112], a logic with forward, discrete, unbounded, non-branching time. While this is intesting due to its simplicity and the relation with FRP that has been discussed previously, other logics might be more suitable for a continuous-time FRP and for the kinds of applications we were discussing.

Havelund and Roşu use a past-time temporal logic, PTLTL, to synthesise monitors efficiently and check properties against finite traces in real time [178].

Variants of LTL for limited traces or bounded time also exist. LTL_f is an extension of LTL for finite traces [179, 180] that has been used in the context of Artificial Intelligence. LTL_f introduces a variant of the operator next, *weak next*, with different meaning for the last time instant in the trace, and adapts the axioms of the logic accordingly.

The study of real-time systems had lead to the development of metric temporal logic (MTL) [181], which might be closer to what FRP demads, in principle, do to its assumption that time be continuous. Alur and Henzinger introduce Metric Interval Temporal Logic (MITL) [182], which is interpreted over time intervals as opposed to time instants. Maler and Nickovic extend this logic for dense-time, real-valued signals, under the assumption of certain desirable properties of continuous signals within time intervals [183].

Hughes et al. [184] use Temporal Logic to specify parts of an asynchronous messaging server, and use QuickCheck to prove properties under timing uncertainty. Tan [185] provides a metric to understand how well a property specified using Temporal Logic has been tested by a test suite.

Giannakopoulou [186] presents an approach at verifying a program's behaviour against LTL specifications, by translating an LTL formula into a finite-state automata that monitors the program's behaviour. This approach is based on the next-free variant of Linear Temporal Logic with Until, as it is guaranteed to be unaffected by *stuttering* (receiving the same input several times in succession).

Havelund [187] has also used Linear Temporal Logic to monitor programs by connecting to them running live via the network. In this work the authors use finite-trace LTL, a variant of infinite-trace LTL in which always means at every point *in the trace*, and next defaults to true at the end of the trace. Some formulae that always hold in limited-trace LTL do not hold in infinite-trace LTL and vice-versa, an aspect explored in also in some detail by Sculthorpe [96]. The use of this logic also results in counter-intuitive properties, for example, at the end of the trace, both *next* X and *next* (*neg* X) hold for any X.

Nejati [188] discuss the construction of a model that approximates a program of interest, and use CTL to verify properties of the model. The authors use a three-valued logic in order to indicate when a model needs further refinements and determine whether a CTL property holds.

In the context of game programming, the idea of recording and replaying programs deterministically was proposed, among others, by John Carmack [189, p. 55], who implemented a single entry point for all input events to a game, and set up a journaling system that recorded everything the game received, including the time. Carmack also pointed out that reproducing bugs is key to fixing them, and that it is important to be able to roll back time in order to find when a bug is

first introduced, even if its effect is only visible later in the execution.

Execution-replay systems have also been studied in the context of general imperative programming and especially parallel and distributed systems [190, 191]. Much of the focus in those areas has been towards dealing with low-level issues to guarantee determinism of replays. In contrast, we rely on Haskell's strong type system to guarantee the freedom from side effects in FRP programs and hence absolute determinism, letting us focus on how to exploit such guarantees for declarative testing and debugging. The debugger presented in this thesis currently records complete input traces, but the idea could be used to log input to only a subpart of the system, making it more suitable for higher-level debugging, as opposed to logging low-level system calls.

Mozilla's tool *rr*[15] monitors a group of Linux processes, capturing the results of kernel calls and non-deterministic CPU effects. In future replays, memory and register contents are the same, and all system calls return the same values. This tool has been designed for general purpose applications, so it works at a very low level. As a result, it is tied to the Linux kernel, it emulates a single-core machine, and it is difficult to port to other architectures (ARM and Android support, for example, is still an open issue[16]). In contrast, the proposal in this thesis works on all architectures for which there is a Haskell compiler with a corresponding back-end (currently iOS, Android, web, Linux, Windows and MacOSX). To control a simulation with the debugging GUI, the only platform-specific adaptation required is a way for the Yampa debugger to open two network sockets for communication.

GDB has *Process Record and Replay*[17], capabilities, which record system calls and the CPU and memory state at each point. Like Mozilla's rr, GDB's replay feature requires adaptations specific for each architecture and system call. An advantage of GDB's implementation is that actions are undoable, which means that we can move backwards along the trace. In contrast, FRP signals only move forward, although the debugging facilities presented in earlier chapters record intermediate Signal Functions to provide random access to any time point in constant time, and they can always be reproduced deterministically if only the inputs are recorded.

[15] http://rr-project.org/
[16] https://github.com/mozilla/rr/issues/1373
[17] https://sourceware.org/gdb/wiki/ProcessRecord

Chapter 12

Summary and Directions for Future Work

We started this thesis by identifying a series of limitations in the current approaches to programming Functional Interactive Applications and, more specifically Functional Reactive Programming. In particular, we observed how abstractions make trade-offs between extensibility, performance, facilities for equational reasoning, the possibility of systematically and consistently testing applications, and code modularity.

To address these concerns, this thesis proposed to refactor FRP into a minimal core, Monadic Stream Functions, capturing what arguably is the essence of FRP, but parametrised over a monad to make it open-ended. Realisation of domain- and application-specific features is then just a matter of picking an appropriate monad. A key benefit of this approach is that the ability to extend MSFs with side effects does not limit our ability to reason about them, which allows us to prove that they satisfy expected arrow laws, preserve monad commutativity and interoperate with monad morphisms.

We demonstrated that MSFs subsume and exceed existing abstractions for Reactive Programming and Functional Reactive Programming, by implementing multiple forms of RP and FRP. Among others, we implemented a reactive abstraction, called Reactive Values, on top of Monadic Stream Functions, which allowed us to describe applications with Graphical User Interfaces in a declarative way, overcoming limitations related to modularity, code duplication, separation of concerns and debugging present in other FRP variants. We also reimplemented an existing FRP system, Yampa, using our minimal core, demonstrating good performance over a selection of medium-sized open-source and commercial games, both on standard mobile and desktop platforms. This FRP variant accommodates additional controlled side effects, different time domains,

clocking frequencies and even sampling policies, making this framework more expressive than other FRP variants.

An advantage of making effects manifest, as when structuring code using arrows or monads, is that it becomes easier to re-execute code. This in turn enabled sophisticated approaches to automated testing in Arrowized FRP variants that separate I/O from logic. We have used an encoding of Linear Temporal Logic over FRP to describe temporal properties of FRP programs and evaluate them against input streams. We have also shown that this simple approach makes FRP programs easily testable with existing tools like QuickCheck [118], and we can add temporal assertions to programs in a similar fashion.

We have extended an FRP implementation with recording and remote control capabilities, and implemented a Graphical User Interface to communicate with games running remotely on PCs, iOS devices and Android phones. The referential transparency available in pure FRP implementations like Yampa helps developers reproduce the exact same situation witnessed by beta-testers, and move back and forth along the trace adding new assertions or visualizing the simulation along on a phone to pinpoint possible bugs. These traces can be fed to QuickCheck for additional help in finding possible property violations, and the counter-examples found by QuickCheck can be loaded back into a phone for future study and to visualise them. So, in practice, we can truly see QuickCheck play our games.

Throughout the thesis, we identified a series of open questions and directions for future work. The following summarises the most prominent directions:

- MSFs could be combined with new monads to implement additional FRP variants. For example, using the same monads used by Elerea [98] or Netwire [116] we might obtain similar expressive power.

- So far, our implementation of MSFs is unoptimised. We would expect optimization techniques like those used in Yampa [26, 27] to carry over, bringing similar performance gains. Similarly, to Push-Pull FRP [147, 148], Frappé [141] and N-ary FRP [96, chapter 8], a notion of different kinds of signals might enable additional optimisations. Guarantees about monad commutativity could also be applicable, under certain circumstances, to normalise and optimise MSF networks, as it was done with Causal Commutative Arrows [87].

  The techniques used to test Arrowized FRP systems in a referentially transparent manner could also be used to benchmark interactive applications reliably, enabling a systematic analysis of the performance gains provided by different optimisation techniques.

- In many interactive applications, different changes to a value can lead to the same result. Because our reactive networks describe time-varying values based on other time-varying values,

which requires additional data to be manually passed along in certain circumstances.

As an example, a list of elements which, after one simulation step, has one element less, could be the result of removing an element from the list, or of loading a similar, but different list. In the former case we might expect a visual animation that eliminates the element, while possibly maintaining the list selection, if any. In the latter case, we might animate a new list appearing in place, and the selection being lost.

A first-class notion of change would enable this kind of adaptation to be expressed in terms of the kind of change being applied to the original value, leading to simpler declarative descriptions of interactive systems.

A notion of change could also help avoid redundant computation when signals are unchanging, with a performance potentially on par with push-based FRP implementations.

Preliminary attempts suggest that Arrows might be too permissive for incremental FRP. Generalized Arrows [192] might facilitate the definition of arrows that only run for changes to the input or due to the progress of time.

- While we explored the use of different time domains, sampling frequencies and limited forms of asynchronous FRP, we have not specified a formal semantics for the time manipulation constructs we provided. A denotational semantics for some time transformations was already defined by Courtney [125], but never introduced in Yampa.

  We also saw that time reversal is possible with MSFs but, unless a notion of branching time is introduced, it remains an implementation concern. Currently, the operational semantics of Yampa is not time-consistent when time flows backwards.

- We have provided a definition of Reactive Values built on top of Monadic Stream Functions, but we have not defined their meaning denotationally or undertaken formal analysis of their temporal properties. Research in this area would help identify minimal conditions of well-defined RV networks and for *eventual consistency* [193] in the presence of circular dependencies.

- It would be interesting to find applications and games for which the temporal language provided is insufficient to express desired properties declaratively and concisely. Our encoding of temporal properties is based on Linear Temporal Logic, but we can answer questions pertaining to multiple possible futures by means of the quantification provided by QuickCheck with *forAll* and the stream generators we provide. Nevertheless, the use of a different formalism like Computation Tree Logic and a comparison with the proposal in this thesis remain as future work.

- A question we have not answered is how well the input space is explored. In this respect, further improvements could be made by adding Signal Function generators, increasing the confidence on abstract properties about Signal Functions, like the Arrow laws that they are expected to fulfil. We have also yet to explore further shrinking strategies for input traces, in order to help QuickCheck find minimal counter-examples.

- The FRP debugger presented in this dissertation has been implemented for Yampa, but the same approach could be used with Arrowized FRP variants for MSFs like the implementation Dunai. An advantage of using Dunai over Yampa would be that the former only has one type for Monadic Stream Functions, which can be parameterised over the time domain [32], making it more versatile while reducing code duplication in the debugger. Also, Dunai introduces monads, which can be used to implement safe debugging facilities, as shown in Chapter 9. Additionally, a monad would allow us to introduce the recording system inside a Signal Function, instead of at the top-level like we do in Yampa. This would let us record high-level input, like declarative game actions, instead of low-level input, like user clicks and mouse movement, which would be more useful as small variations to the user interface are more likely to invalidate the latter.

- We have also not studied the performance cost of using the proposed solutions, or how much they could affect the behaviour of a program and the detection of bugs. In order for a game to be fully reproducible, users need to record the complete input trace. It is in principle possible, for instance, for sampling times to never be fast enough when debugging is enabled for certain bugs to appear, what is known as a *heisenbug* [194] (a bug that is only detectable when debugging tools are not enabled). In some cases we have reduced the debugging facilities introduced in a binary to only saving the inputs to a file, to minimise the overhead and hence the possibility of introducing heisenbugs. If a bug is detected with this minimal debugger and the input is logged accordingly, then it will be completely reproducible with the full debugger.

- A additional feature facilitated by the proposed framework is that we could stream input traces over to another machine to visualise a game as it is being played, or use them to replay a game in a machine with higher visual specifications or to record a video. We could also replay these games later on in higher quality or record videos of the output for publication online, even if they were produced in devices with lower specifications.

Appendix A

MSFs, Arrows and Arrow Laws

Arrow laws

The arrow laws and other properties hold for Monad Stream Functions. To prove them, we model Haskell types as *complete partial orders* (CPOs), functions as continuous maps, and use *fixpoint induction* [195]. Each fundamental category and arrow combinator like *id*, $\ggg$, *arr* and *first* is a fixed point of a corresponding *recursion function*. The following includes a shortened proof of one arrow law. This proof is based on a proof by Jennifer Hackett, and mainly due to Manuel Bärenz. For a detailed account of proofs of other arrow laws for MSFs, including a proof that MSFs are Commutative Arrows for commutative monads, see [117].

Definitions

For conciseness, we will leave out the value constructors, so the type of MSF is simply $MSF\ m\ a\ b = a \to m\ (b, MSF\ m\ a\ b)$.

$$
\begin{aligned}
&arr \quad = fix\ arrRec \\
&arrRec\ rec\ f\ a = return\ (f\ a, rec\ f) \\
&(\ggg) = fix\ compRec \\
&compRec\ rec\ f\ g\ a = \mathbf{do}\ (b, f') \leftarrow f\ a \\
&\qquad\qquad\qquad\qquad\quad (c, g') \leftarrow g\ b \\
&\qquad\qquad\qquad\qquad\quad return\ (c, rec\ f'\ g')
\end{aligned}
$$

Proof: $arr\ (f \ggg g) \equiv arr\ f \ggg arr\ g$

The application of fixpoint induction means that if P is a predicate on values of some type a, and $f : a \to a$, and P holds for $\bot$, and for every value x we can infer $P\ x \Rightarrow P\ (f\ x)$, then P holds for

$fix\ f\ =_{\text{def}}\ f\ (fix\ f)$. We define the predicate P as:

$$P\ (x, y)\ =_{\text{def}}\ \forall\ f\ g\ .\ (x\ f)\ `y`\ (x\ g) \equiv x\ (g \circ f)$$

A requirement to apply fixpoint induction is that P be chain complete. But because it is an equation involving universally quantified free variables and function application, we know it is chain complete [35].
We proceed as follows:

$$\begin{aligned}
& P\ (\bot, \bot) \\
\iff & \{\ \textit{Definition of } P\ \} \\
& \forall\ f\ g\ .\ (\bot\ f)\ `\bot`\ (\bot\ g) \equiv \bot\ (g \circ f) \\
\iff & \{\ \textit{Application to } \bot\ \} \\
& \bot \equiv \bot \\
\iff & \textit{true} \\
& P\ (arrRec\ x, compRec\ y) \\
\iff & \{\ \textit{Definition of } P\ \} \\
& \forall\ f\ g\ .\ (arrRec\ x\ f)\ `compRec\ y`\ (arrRec\ x\ g) \\
& \qquad \equiv arrRec\ x\ (g \circ f) \\
\iff & \{\ \textit{Definition of compRec and arrRec and } \beta\textit{-Reduction}\ \} \\
& \forall\ f\ g\ .\ lhs \equiv \lambda a \to return\ ((g \circ f)\ a, x\ (g \circ f))\ \textbf{where} \\
& \quad lhs = \lambda a \to \textbf{do} \\
& \qquad (b, msf1') \leftarrow return\ (f\ a, x\ f) \\
& \qquad (c, msf2') \leftarrow return\ (g\ b, x\ g) \\
& \qquad return\ (c, msf1'\ `y`\ msf2') \\
\iff & \{\ \textit{Monad laws, function composition}\ \} \\
& \forall\ f\ g\ .\ \lambda a \to return\ (g\ (f\ a), x\ f\ `y`\ x\ g) \\
& \qquad \equiv \lambda a \to return\ (g\ (f\ a), x\ (g \circ f)) \\
\Leftarrow & \quad \forall\ f\ g\ .\ x\ f\ `y`\ x\ g \equiv x\ (g \circ f) \\
\iff & \ P\ (x, y)
\end{aligned}$$

Monadic Streams

In Chapter 5, we illustrated, by expanding the definition of a Monadic Stream on the *Identity* monad, that they are coinductive infinite streams. Let us now provide a more convincing argument, with a proof that operates only at the type level.

Let us define streams by the following type:

$$\textbf{type}\ S\ a = (a, S\ a)$$

And let us define Monadic Streams as:

$$\textbf{type}\ \mathit{MStream}\ m\ a = m\ (a, \mathit{MStream}\ m\ a)$$

The claim is that $\mathit{MStream}\ \mathit{Identity}\ a \equiv S\ a$, for all a.

Let us introduce a new type constructor, T, defined as $\textbf{type}\ T\ a = \mathit{MStream}\ \mathit{Identity}\ a$, and reason as follows:

$$\begin{aligned}
\mathit{MStream}\ \mathit{Identity}\ a &\cong \{\, \mathit{definition\ of\ MStream} \,\} \\
&\quad \mathit{Identity}\ (a, \mathit{MStream}\ \mathit{Identity}\ a) \\
&\cong \{\, \mathit{definition\ of\ Identity} \,\} \\
&\quad (a, \mathit{MStream}\ \mathit{Identity}\ a) \\
&\cong \{\, \mathit{definition\ of}\ T \,\} \\
&\quad (a, T\ a) \\
&\cong \{\, \mathit{definition\ of}\ T \,\} \\
&\quad T\ a
\end{aligned}$$

Therefore, we know that:

$$T\ a \cong (a, T\ a)$$

And, by definition:

$$S\ a \cong (a, S\ a)$$

So both types are defined by the same expression (or, in this case, fulfill the same equation), making them trivially equivalent. As $S\ a \cong T\ a \cong \mathit{MStream}\ \mathit{Identity}\ a$, we conclude that Monadic Streams are just coinductive infinite streams when the monad is the *Identity* monad.

Bibliography

[1] B. A. Myers, "Why Are Human-Computer Interfaces Difficult to Design and Implement?," *Technical Report, No. CMU-CS-93-183*, 1993.

[2] S. Goderis, *On the separation of user interface concerns: A Programmer's Perspective on the Modularisation of User Interface Code.* Asp / Vubpress / Upa, 2007.

[3] S. Burbeck, "Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)," *Smalltalk-80 v2*, vol. 5, 1992.

[4] G. T. Heineman, "An instance-oriented approach to constructing product lines from layers," *Technical Report, WPI CS Tech Report 05-06*, 2005.

[5] S. McDirmid and W. C. Hsieh, "Superglue: Component programming with object-oriented signals," in *ECOOP 2006–Object-Oriented Programming*, pp. 206–229, Springer, 2006.

[6] E. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

[7] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke, "Eliminating concurrency bugs with control engineering," *IEEE Computer*, vol. 42, no. 11, pp. 52–60, 2009.

[8] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pp. 221–230, IEEE, 2010.

[9] W. M. Newman and R. F. Sproull, *Principles of interactive computer graphics.* McGraw-Hill, Inc., 1979.

[10] M. Fayad and D. C. Schmidt, "Object-oriented Application Frameworks," *Commun. ACM*, vol. 40, pp. 32–38, Oct. 1997.

[11] B. A. Myers, "Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs," in *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, (New York, NY, USA), pp. 211–220, ACM, 1991.

[12] G. E. Krasner, S. T. Pope, *et al.*, "A Description of the Model-View-Controller User Interface paradigm in the Smalltalk-80 System," 1988.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[14] D. Caromel and Y. Roudier, "Reactive programming in Eiffel," in *Object-Based Parallel and Distributed Computation*, pp. 125–147, Springer, 1996.

[15] F. Boussinot, G. Doumenc, and J.-B. Stefani, "Reactive objects," in *Annales des télécommunications*, vol. 51, pp. 459–473, Springer, 1996.

[16] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pp. 48–60, 2005.

[17] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[18] P. Wadler, "Comprehending monads," *Mathematical Structures in Computer Science*, vol. 2, no. 04, pp. 461–493, 1992.

[19] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 71–84, ACM, 1993.

[20] D. Fahland, D. Lbke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus imperative process modeling languages: The issue of understandability," in *Enterprise, Business-Process and Information Systems Modeling* (T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, eds.), vol. 29 of *Lecture Notes in Business Information Processing*, pp. 353–366, Springer Berlin Heidelberg, 2009.

[21] V. T. Ravi and G. Agrawal, "Performance issues in parallelizing data-intensive applications on a multi-core cluster," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 308–315, IEEE Computer Society, 2009.

[22] M. Carlsson and T. Hallgren, "FUDGETS: A graphical user interface in a lazy functional language," no. Section 6, pp. 321–330, 1993.

[23] R. Noble and C. Runciman, "Gadgets: Lazy functional components for graphical user interfaces," in *Programming Languages: Implementations, Logics and Programs*, pp. 321–340, Springer, 1995.

[24] I. Perez, "1st Year PhD Report." `http://www.cs.nott.ac.uk/~ixp/`, Dec. 2014.

[25] C. Elliott and P. Hudak, "Functional Reactive Animation," in *ACM SIGPLAN Notices*, vol. 32(8), pp. 263–273, 1997.

[26] A. Courtney and C. Elliott, "Genuinely Functional User Interfaces," in *Haskell Workshop*, pp. 41–69, Sept. 2001.

[27] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Haskell Workshop*, pp. 51–64, 2002.

[28] M. Fahndrich and S. Xia, "Establishing object invariants with delayed types," in *ACM SIGPLAN Notices*, vol. 42, pp. 337–350, ACM, 2007.

[29] I. Perez and H. Nilsson, "Bridging the GUI Gap with Reactive Values and Relations," in *Haskell Symposium*, pp. 47–58, 2015.

[30] D. Syme, "Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 2, pp. 3 – 25, 2006. Proceedings of the ACM-SIGPLAN Workshop on {ML} (ML 2005) ACM-SIGPLAN Workshop on {ML} 2005.

[31] I. Perez, M. Bärenz, and H. Nilsson, "Functional Reactive Programming, Refactored," in *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, (New York, NY, USA), pp. 33–44, ACM, 2016.

[32] I. Perez, "Back to the Future: Time Travel in FRP," in *Proceedings of the 10th ACM SIGPLAN International Haskell Symposium*, Haskell 2017, (New York, NY, USA), ACM, 2017.

[33] I. Perez and H. Nilsson, "Testing and Debugging Functional Reactive Programming," *Proc. ACM Program. Lang.*, vol. 1, pp. 2:1–2:27, Aug. 2017.

[34] G. Hutton, *Programming in Haskell*. New York, NY, USA: Cambridge University Press, 2nd ed., 2016.

[35] R. Bird, *Thinking functionally with Haskell*. Cambridge University Press, 2014.

[36] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, no. 1, pp. 67 – 111, 2000.

[37] R. Paterson, "A New Notation for Arrows," in *International Conference on Functional Programming*, pp. 229–240, Sept. 2001.

[38] B. C. Pierce, *Basic category theory for computer scientists*. MIT press, 1991.

[39] P. Aluffi, *Algebra: chapter 0*, vol. 104. American Mathematical Soc., 2009.

[40] A. Courtney, H. Nilsson, and J. Peterson, "The Yampa arcade," in *Haskell Workshop*, pp. 7–18, 2003.

[41] C. McBride and R. Paterson, "Applicative programming with effects," *Journal of functional programming*, vol. 18, no. 01, pp. 1–13, 2008.

[42] J. Hughes, "Programming with Arrows," in *5th International School on Advanced Functional Programming* (V. Vene and T. Uustalu, eds.), vol. 3622, pp. 73–129, 2005.

[43] J. R. Rose and H. Muller, "Integrating the Scheme and C Languages," in *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, (New York, NY, USA), pp. 247–259, ACM, 1992.

[44] H. Sexton, "Foreign functions and common Lisp," *ACM SIGPLAN Lisp Pointers*, vol. 1, no. 5, pp. 11–23, 1987.

[45] "Hcwiid: Haskell bindings to Cwiid." `https://github.com/ivanperez-keera/hcwiid`. Accessed: 2014-10-20.

[46] "OpenGL Haskell bindings." `http://hackage.haskell.org/package/OpenGL`. Accessed: 2014-10-20.

[47] J. Gibbons and R. Hinze, "Just do it: Simple monadic equational reasoning," in *ICFP*, September 2011.

[48] G. Hutton and D. Fulger, "Reasoning About Effects: Seeing the Wood Through the Trees," in *Proceedings of the Symposium on Trends in Functional Programming*, (Nijmegen, The Netherlands), May 2008.

[49] W. Swierstra and T. Altenkirch, "Beauty in the Beast," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, (New York, NY, USA), pp. 25–36, ACM, 2007.

[50] J. Backus, "Can programming be liberated from the von Neumann style?," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[51] "GTK+." `http://www.gtk.org`. Accessed: 2014-10-20.

[52] "wxWidgets." `http://www.wxwidgets.org`. Accessed: 2014-10-20.

[53] "QT Project." `http://qtproject.org`. Accessed: 2014-10-20.

[54] "Gtk2Hs Tutorial - The Button Widget." `http://code.haskell.org/gtk2hs/docs/tutorial/Tutorial_Port/chap4-1.xhtml`. Accessed: 2017-11-05.

[55] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. O'Reilly Media, Inc., 2008.

[56] J. Edwards, "Coherent reaction," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 925–932, ACM, 2009.

[57] J. R. Heard, "Beautiful code, compelling evidence," tech. rep., 2008.

[58] H. Liu, N. Glew, L. Petersen, and T. A. Anderson, "The Intel labs Haskell research compiler," in *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pp. 105–116, ACM, 2013.

[59] Sweeney, Tim, "[gclist] Games and Realtime Garbage Collection." `http://lists.tunes.org/archives/gclist/1999-July/001632.html`, 1999.

[60] Peter Verswyleven, "New OpenGL package: efficient way to convert datatypes?." `https://www.haskell.org/pipermail/haskell-cafe/2010-March/074100.html`, 2010. Quote: "I just converted an old HOpenGL application of mine to [...] OpenGL[...] [u]sing unsafeCoerce I got 800 FPS again.".

[61] K. Claessen, T. Vullinghs, and E. Meijer, "Structuring graphical paradigms in TkGofer," in *ACM SIGPLAN Notices*, vol. 32, pp. 251–262, ACM, 1997.

[62] T. Vullinghs, W. Schulte, and T. Schwinn, *An introduction to TkGofer*. 1996.

[63] S. Finne and S. Peyton Jones, "Composing Haggis," in *Programming Paradigms in Graphics* (R. C. Veltkamp and E. H. Blake, eds.), pp. 85–101, Springer-Verlag, 1995.

[64] B. Yorgey, "Diagrams - a domain-specific language for creating vector graphic." `http://projects.haskell.org/diagrams/`, 2008. Accessed: 2014-10-24.

[65] T. Docker, "Chart - A library for generating 2D Charts and Plots." `https://github.com/timbod7/haskell-chart`, 2006. Accessed: 2014-10-24.

[66] B. Lippmeier, "Gloss - Painless 2D vector graphics, animations and simulations.." http://gloss.ouroborus.net/, 2010. Accessed: 2014-10-24.

[67] B. Lippmeier, "Gloss Machina." https://github.com/benl23x5/gloss/blob/master/gloss-examples/picture/Machina/Main.hs, 2012. Accessed: 2014-10-24.

[68] B. Victor, "Inventing on Principle." http://vimeo.com/36579366, 2011. Accessed: 2014-10-23.

[69] "Supporting Different Screens." http://developer.android.com/training/basics/supporting-devices/screens.html.

[70] C. M. Elliott, "Tangible functional programming," *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 59–70, 2007.

[71] S. Michels, R. Plasmeijer, and P. Achten, "iTask as a new paradigm for building GUI applications," in *Implementation and Application of Functional Languages*, pp. 153–168, Springer, 2011.

[72] S. Wilson and P. Johnson, "Bridging the generation gap: From work tasks to user interface designs," in *In Computer-Aided Design of User Interfaces*, pp. 77–94, Namur University Press, 1996.

[73] T. Brus, M. C. van Eekelen, M. Van Leer, and M. J. Plasmeijer, "Clean a language for functional graph rewriting," in *Functional Programming Languages and Computer Architecture*, pp. 364–384, Springer, 1987.

[74] P. Achten, M. Van Eekelen, and R. Plasmeijer, "Generic Graphical User Interfaces," in *Implementation of Functional Languages*, pp. 152–167, Springer, 2005.

[75] "Getting Up and Running Quickly with Scaffolding." http://guides.rubyonrails.org/v3.2.13/getting_started.html#getting-up-and-running-quickly-with-scaffolding.

[76] "Deprecating Dynamic Scaffolding." https://groups.google.com/d/topic/rubyonrails-core/fSkbnrdw5PM/discussion.

[77] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 52, 2013.

[78] G. Patai, "Efficient and compositional higher-order streams," in *Functional and Constraint Logic Programming*, pp. 137–154, 2010.

[79] A. v. d. Ploeg and K. Claessen, "Practical Principled FRP: Forget the Past, Change the Future, FRPNow!," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), pp. 302–314, ACM, 2015.

[80] G. Patai, "Eventless reactivity from scratch," *Draft Proceedings of Implementation and Application of Functional Languages (IFL'09)*, pp. 126–140, 2009.

[81] G. Russel, "Fruit & co.." Message posted on the Haskell GUI mailing list, available at `https://www.haskell.org/pipermail/gui/2003-February/000140.html`, 2003.

[82] "Xournal." `http://xournal.sourceforge.net`. Accessed: 2014-10-20.

[83] E. Czaplicki, *Elm: Concurrent FRP for Functional GUIs*. PhD thesis, 2012.

[84] H. Apfelmus, "Reactive-banana." `http://www.haskell.org/haskellwiki/Reactive-banana`, 2011. Accessed: 2014-10-20.

[85] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 52, 2013.

[86] S. Marlow, S. Peyton Jones, E. Kmett, and A. Mokhov, "Desugaring Haskell's do-notation into applicative operations," in *Proceedings of the 9th International Symposium on Haskell*, pp. 92–104, ACM, 2016.

[87] H. Liu, E. Cheng, and P. Hudak, "Causal commutative arrows and their optimization," in *ACM Sigplan Notices*, vol. 44, pp. 35–46, 2009.

[88] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Symposium on Principles of programming languages*, pp. 333–343, 1995.

[89] C. J. Taylor, *Formalising and reasoning about Fudgets*. PhD thesis, 1998.

[90] V. Capretta and J. Fowler, "The continuity of monadic stream functions," in *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'17*, ACM, 2017.

[91] D. Harel and A. Pnueli, "Logics and Models of Concurrent Systems," ch. On the Development of Reactive Systems, pp. 477–498, New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[92] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive Noninterference," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 79–90, ACM, 2009.

[93] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[94] D. Winograd-Cort and P. Hudak, "Wormholes: Introducing Effects to FRP," in *Haskell Symposium*, 2012.

[95] Z. Wan and P. Hudak, "Functional Reactive Programming from First Principles," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 242–252, 2000.

[96] N. Sculthorpe, *Towards Safe and Efficient Functional Reactive Programming*. PhD thesis, School of Computer Science, University of Nottingham, July 2011.

[97] L. Euler, *Institutiones calculi integralis.-Volumen primum [-tertium]*. 1768.

[98] G. Patai, "Efficient and compositional higher-order streams," in *Functional and Constraint Logic Programming*, pp. 137–154, Springer, 2011.

[99] J. Gregory, *Game Engine Architecture, Second Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2nd ed., 2014.

[100] H. Nilsson, "Functional Reactivity: Eschewing the Imperative." Invited talk, Department of Computer and Information Science, Linkpings Universitet, 2003.

[101] "Magic Cookies." `https://play.google.com/store/apps/details?id=uk.co.keera.games.magiccookies`.

[102] "Magic Cookies." `https://itunes.apple.com/us/app/magic-cookies/id1244709871`.

[103] "Haskanoid." `github.com/ivanperez-keera/haskanoid`.

[104] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 233–246, 2005.

[105] Keera Studios, "Keera Hails - Haskell on Rails." `https://github.com/keera-studios/keera-hails`.

[106] K. Hoste, "An Introduction to Gtk2Hs, a Haskell GUI Library," in *The Monad.Reader Issue 1* (S. Erisson, ed.), 2005.

[107] C. P. R. Baaij, J. Kuper, and L. Schubert, "Soosim: Operating system and programming language exploration," in *Proceedings of the 3rd International Workshop on Analysis Tools and*

Methodologies for Embedded and Real-time System (WATERS 2012), Pisa, Italy (G. Lipari and T. Cucinotta, eds.), (Italy), pp. 63–68, Giuseppe Lipari, 2012.

[108] I. Perez, "GALE: A Functional Graphic Adventure Library and Engine," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*, FARM 2017, (New York, NY, USA), pp. 28–35, ACM, 2017.

[109] J. A. Whittaker, "What is software testing? And why is it so hard?," *IEEE Software*, vol. 17, pp. 70–79, Jan 2000.

[110] C. Lewis, J. Whitehead, and N. Wardrip-Fruin, "What went wrong: A taxonomy of video game bugs," in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, (New York, NY, USA), pp. 108–115, ACM, 2010.

[111] E. A. Emerson, "Handbook of Theoretical Computer Science (Vol. B)," ch. Temporal and Modal Logic, pp. 995–1072, Cambridge, MA, USA: MIT Press, 1990.

[112] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[113] A. N. Prior, *Past, present and future*, vol. 154. Oxford University Press, 1967.

[114] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, (New York, NY, USA), pp. 268–279, ACM, 2000.

[115] H. Nilsson, "Dynamic Optimization for FRP using Generalized Algebraic Data Types," *International Conference on Functional Programming*, pp. 54–65, 2005.

[116] "Netwire." `http://hub.darcs.net/ertes/netwire`. Accessed: 2014-10-20.

[117] M. Bärenz, I. Perez, and H. Nilsson, "Mathematical Properties of Monadic Stream Functions." `http://cs.nott.ac.uk/~ixp/papers/msfmathprops.pdf`, 2016.

[118] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *ACM Sigplan Notices*, vol. 46, pp. 53–64, 2011.

[119] A. van der Ploeg, "Monadic Functional Reactive Programming," in *Haskell Symposium*, pp. 117–128, 2013.

[120] J. Peterson, A. Courtney, and B. Robinson, "Can GUI programming be liberated from the IO monad," in *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pp. 22–22, Citeseer, 2005.

[121] K. C. Rovers, *Functional model-based design of embedded systems with UniTi*. PhD thesis, 2011.

[122] O. Kiselyov, "Iteratees," in *Functional and Logic Programming: 11th International Symposium, FLOPS 2012*, pp. 166–181, 2012.

[123] "Pipes." `https://hackage.haskell.org/package/pipes`.

[124] "Reification of time in FRP." `http://pchiusano.blogspot.co.uk/2010/07/reification-of-time-in-frp-is.html`.

[125] A. A. Courtney, *Modeling User Interfaces in a Functional Language*. PhD thesis, New Haven, CT, USA, 2004. AAI3125177.

[126] Z. Wan, W. Taha, and P. Hudak, "Real-time FRP," in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pp. 146–156, ACM, 2001.

[127] Z. Wan, W. Taha, and P. Hudak, "Event-driven FRP," in *Practical Aspects of Declarative Languages*, pp. 155–172, Springer, 2002.

[128] R. Kaiabachev, W. Taha, and A. Zhu, "E-FRP with priorities," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pp. 221–230, ACM, 2007.

[129] N. R. Krishnaswami and N. Benton, "Ultrametric semantics of reactive programs," in *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pp. 257–266, IEEE, 2011.

[130] N. R. Krishnaswami, N. Benton, and J. Hoffmann, "Higher-order functional reactive programming in bounded space," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 45–58, 2012.

[131] N. R. Krishnaswami, "Higher-order functional reactive programming without spacetime leaks," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pp. 221–232, ACM, 2013.

[132] G. H. Cooper and S. Krishnamurthi, "FrTime: Functional reactive programming in PLT Scheme," *Computer science technical report. Brown University. CS-03-20*, 2004.

[133] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *ACM SIGPLAN Notices*, vol. 48, pp. 411–422, ACM, 2013.

[134] T. J. Ameloot, F. Neven, and J. Van den Bussche, "Relational transducers for declarative networking," *Journal of the ACM (JACM)*, vol. 60, no. 2, p. 15, 2013.

[135] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[136] M. Pouzet, "Lucid Synchrone, version 2. Tutorial and reference manual," 2001.

[137] F. Boussinot and R. De Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.

[138] T. Bourke and M. Pouzet, "Zélus: A synchronous language with ODEs," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pp. 113–118, ACM, 2013.

[139] P. Hudak, "The Haskell School of Expression: Learning Functional Programming through Multimedia," Jan. 2000.

[140] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for Ajax applications," in *ACM SIGPLAN Notices*, vol. 44, pp. 1–20, ACM, 2009.

[141] A. Courtney, "Frappé: Functional reactive programming in Java," in *Third International Symposium on Pratical Aspects of Declarative Languages (PADL)*, March 2001.

[142] M. Sage, "FranTk-a declarative GUI language for Haskell," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 106–117, 2000.

[143] H. Nilsson and I. Perez, "Declarative Game Programming," in *International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)* (O. Danvy, ed.), (Canterbury, UK), ACM Press, September 2014.

[144] M. H. Cheong, "Functional programming and 3D games," *BEng thesis, University of New South Wales, Sydney, Australia*, 2005.

[145] N. Sculthorpe and H. Nilsson, "Keeping calm in the face of change," *Higher-Order and Symbolic Computation*, vol. 23, no. 2, pp. 227–271, 2010.

[146] W. Jeltsch, "Improving Push-based FRP," *Draft Proceedings of Trends in Functional Programming (TFP'08)*, 2008.

[147] C. Elliott, "Push-Pull Functional Reactive Programming," in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pp. 25–36, ACM, 2009.

[148] E. Amsden, "Push-Pull Signal-Function Functional Reactive Programming," in *Draft Proceedings of the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, p. 128, 2012.

[149] W. Jeltsch, "Signals, Not Generators!," *Trends in Functional Programming*, vol. 10, pp. 145–160, 2009.

[150] Y. Ohshima, B. Freudenberg, A. Lunzer, and T. Kaehler, "A report on KScript and KSWorld," *VPRI Research Note-2012-008*, 2012.

[151] B. Robinson, "wxFruit: A practical GUI toolkit for functional reactive programming," 2004.

[152] "SDL: binding to libSDL." `https://hackage.haskell.org/package/SDL`. Accessed: 2014-10-20.

[153] "SDL2: bindings to libSDL ¿= 2.0.0." `https://github.com/Lemmih/hsSDL2`. Accessed: 2014-10-20.

[154] "Yampa-glut: an adapter that connects OpenGL/GLUT to the FRP library "Yampa"." `https://hackage.haskell.org/package/yampa-glut`. Accessed: 2014-10-20.

[155] "A simple graphics library based on X11 or Win32." `https://hackage.haskell.org/package/HGL`. Accessed: 2014-10-20.

[156] "A binding to part of the Win32 library." `http://hackage.haskell.org/package/Win32`. Accessed: 2014-10-20.

[157] "Connect Reactive and FieldTrip." `http://hackage.haskell.org/package/reactive-fieldtrip`. Accessed: 2014-10-20.

[158] "LambdaCube 3D." `http://lambdacube3d.wordpress.com/`. Accessed: 2014-10-20.

[159] "Introduction to discrete-event simulation and the simpy language,"

[160] "Ordrea: Push-pull implementation of discrete-time FRP." `https://hackage.haskell.org/package/ordrea`. Accessed: 2014-11-22.

[161] H. Liu and P. Hudak, "Plugging a space leak with an arrow," *Electronic Notes in Theoretical Computer Science*, vol. 193, pp. 29–45, 2007.

[162] N. R. Krishnaswami, "Higher-order reactive programming without spacetime leaks," in *International Conference on Functional Programming (ICFP)*, Sept. 2013.

[163] Jonathan Blow, "Braid." `http://braid-game.com/`, 2008.

[164] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "CLaSH: Structural descriptions of synchronous hardware using Haskell," in *EUROMICRO*, 2010.

[165] M. Bärenz, "Rhine - frp with type-level clocks," 2016. (Submitted).

[166] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[167] J. Peterson, V. Trifonov, and A. Serjantov, "Parallel Functional Reactive Programming," in *Practical Aspects of Declarative Languages, Volume 1753 of LNCS*, pp. 16–31, Springer, 2000.

[168] S. Michels and R. Plasmeijer, "Uniform data sources in a functional language," in *Submitted for presentation at Symposium on Trends in Functional Programming, TFP*, vol. 12, 2012.

[169] B. C. Pierce, "Combinators for bi-directional tree transformations: A linguistic approach to the view update problem," Oct. 2004. Invited talk at *New England Programming Languages Symposium*.

[170] L. Domoszlai, B. Lijnse, and R. Plasmeijer, "Parametric lenses: change notification for bidirectional lenses," in *Proceedings of the Symposium on Trends in Functional Programming*, (Soesterberg, The Netherlands), May 2014. Accepted for publication.

[171] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[172] A. Osmani, "Data-binding revolutions with Object.observe()."

[173] A. Jeffrey, "LTL Types FRP: Linear-time Temporal Logic Propositions As Types, Proofs As Functional Reactive Programs," in *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, (New York, NY, USA), pp. 49–60, ACM, 2012.

[174] A. Jeffrey, "Functional Reactive Types," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, (New York, NY, USA), pp. 54:1–54:9, ACM, 2014.

[175] W. Jeltsch, "Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming," *Electron. Notes Theor. Comput. Sci.*, vol. 286, pp. 229–242, Sept. 2012.

[176] W. Jeltsch, "Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories," in *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, (New York, NY, USA), pp. 69–78, ACM, 2013.

[177] S. Winitzki, "Temporal logic and functional reactive programming." `https://github.com/winitzki/talks/tree/master/frp`, 2014.

[178] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 342–356, Springer, 2002.

[179] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pp. 854–860, AAAI Press, 2013.

[180] G. Roşu, "Finite-Trace Linear Temporal Logic: Coinductive Completeness," in *Runtime Verification - 16th International Conference, RV 2016 Madrid, Spain, September 23-30, 2016, Proceedings*, vol. 10012 of *Lecture Notes in Computer Science*, pp. 333–350, Springer, September 2016.

[181] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[182] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM (JACM)*, vol. 43, no. 1, pp. 116–146, 1996.

[183] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 152–166, Springer, 2004.

[184] J. Hughes, U. Norell, and J. Sautret, "Using temporal relations to specify and test an instant messaging server," in *The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa*, pp. 95–102, 2010.

[185] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pp. 493–498, IEEE, 2004.

[186] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 412–416, Nov 2001.

[187] K. Havelund and G. Rosu, "Monitoring programs using rewriting," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 135–143, IEEE, 2001.

[188] S. Nejati, A. Gurfinkel, and M. Chechik, "Stuttering abstraction for model checking," in *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*, pp. 311–320, IEEE, 2005.

[189] J. Carmack, "John carmack archive - .plan." `http://fd.fabiensanglard.net/doom3/pdfs/johnc-plan_1998.pdf`, 1998.

[190] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. De Bosschere, "A taxonomy of execution replay systems," *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[191] M. Ronsse, K. De Bosschere, and J. C. De Kergommeaux, "Execution replay and debugging," *arXiv preprint cs/0011006*, 2000.

[192] A. M. Joseph, *Generalized Arrows*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[193] W. Vogels, "Eventually Consistent," *Commun. ACM*, vol. 52, pp. 40–44, Jan. 2009.

[194] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," in *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, 1986.

[195] J. Gibbons and G. Hutton, "Proof Methods for Corecursive Programs," *Fundamenta Informaticae Special Issue on Program Transformation*, vol. 66, pp. 353–366, April-May 2005.