First Year Report

Ivan Perez

University of Nottingham

ixp@cs.nott.ac.uk

Abstract

Functional Programming brings with it the promise of highly-declarative code, efficient, parallelisable execution, modularity and reusability. This promise has already materialised in multiple fields of application. Yet when it comes to programming with effects —common in interactive software— sticking to a purely functional style, and obtaining all the benefits of doing so, is arguably much harder.

Writing interactive software comprises a wide range of subproblems, such as describing both static and dynamic user interfaces, connecting them to the rest of the program, or keeping both in synchrony while minimising data propagation and keeping response time low.

These and other related concerns have been explored by the functional programming community, and proposals to address them 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 text I explore the area of interactive application programming in functional languages, presenting what I consider to be the most relevant open problems, together with a review of the existing literature. A narrower selection of open problems is then identified, and I propose a possible way to address them. Finally, I describe what I have done already, what remains to be completed, and a potential plan for my PhD studies that includes a draft outline of my thesis.

Contents

Chapter 1

Introduction

This document explores programming software with Graphical User Interfaces (GUIs) in purely functional languages. Programming software with Graphical User Interfaces (GUIs) is known to be hard [Mye93], irrespectively of the programming paradigm.

Ideally, one would like to write GUI applications that are well-structured and 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 achieving all three.

Limitations of imperative GUI programming

GUI applications are often large and intrinsically complex, with many inter-related elements [Mye93]. This complexity lies both in the application domain and in common usability features, such as being able to cancel long operations (which requires *concurrency*, critical regions, 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 [God07, Bur92, Hei05, MH06, Dij71, KWLM09, FLSR10] and of particular importance.

GUI programs are inherently stateful. This is true as applications interact with the outside world (*input/output*) and because they keep internal state, reacting 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. 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. Reasoning about the application requires that programmers keep mental track of how the state changes along execution, for which a deep understanding of the potential side effects of each function, GUI-related or not, is necessary.

These GUI toolkits also present one further problem: reacting to user input is done by associating certain computations to specific widgets and events. This results in programs broken up in multiple inter-related parts, an event-oriented programming style [NS79] that inverts control [FS97, p. 36-37] and is hard to reason about [Mye91].

To address modularity and structural concerns, multiple GUI application architectures have been suggested. The widely-used[1,2] Model-View-Controller [KP+88] 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[3], 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 (SoC), an error-prone style and codebases that grow quadratically with every new feature or UI element[4].

To address the problems of MVC and similar architectural patterns, languages have incorporated the idea of listening to changes in data structures [GHJV95, 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 more callbacks[5].

Reactive Programming [CR96, BDS96] takes this approach one step further by ubiquitously using objects with change listening capabilities. 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 objects that encapsulate such values.

The Functional approach to 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 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 [HMPJH05].

There are dozens of Functional libraries to implement GUIs. As will be seen in the following, low-level bindings to existing GUI toolkits provide competitive efficiency and a visually appealing appearance, but impose an imperative programming style. High-level, pure, compositional widget toolkits 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.

Functional Languages make state explicit and require that we pass it around as we modify it, always in the right order. Abstractions like monads [Mog91, Wad92, PJW93] 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 [FLM+09], to parallelise [RA09, p. 309], to

[1]"After reading this guide, you will know: [...] The basic principles of MVC (Model, View, Controller)", `http://guides.rubyonrails.org/getting_started.html` (on October 23, 2014).

[2]"Django is a MTV framework", `https://docs.djangoproject.com/en/dev/faq/general/` (on October 23, 2014).

[3]Full screen refreshes produce noticeable *flickers*.

[4] GUI applictions 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.

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

optimise (both manually and by the compiler), and suffers from the kinds of modularity concerns described earlier.

An alternative approach is to define a functional abstraction for reactive processes that facilitates reasoning and specifying data dependencies, and leverages the responsibility of linearising state. Functional Reactive Programming addresses reactive application concerns by defining reactive, dynamic programs as collections of inter-dependent time-varying values or *signals*. Data dependencies and relations among signals are specified upon signal creation. As cyclic dependencies are extremely common in GUI programs [Sym06], this results in loops that pose signal initialisation problems [FX07] and modularity concerns (section 3.4). Efficient implementation of FRP has traditionally been a concern, and is still an active area of research.

As will be detailed in chapter 2, there exist proposals to implement GUI toolkits directly in functional languages, with pure, declarative APIs and well-defined semantics. This approach can be extremely expensive, it does not adapt well to today's requirements, and is not flexible enough, due to the following reasons.

GUI toolkits are notoriously large[6] 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. To obtain a natural appearance, implementing several such toolkits, one per platform, would only exacerbate such costs.

One of the most challenging aspects of GUI software is designing the graphical interfaces themselves [Mye93]. Domain experts tend to prefer visual tools to create GUIs [Vic11], which can then be loaded by the program during runtime[7] —with minimal static guarantees[8]– or converted into code and compiled with the rest of the application[9].

When GUIs are constructed programmatically, adapting to changing conditions requires extra code that removes, adjusts and introduces interactive elements, connecting them to the rest of the program and manually handling callbacks. Analysing how new elements interact with the existing ones can be extremely hard. On the other hand, mobile platforms, for instance, can automatically load and apply a new GUI specifications from XML files during runtime to adapt to new conditions, such as when users switch from portrait mode to landscape mode. This results in programs that are easier to adapt to new devices and environments, and in code that is less error-prone.

Outline

The rest of the report is structured as follows. Chapters 2 and 3 introduce functional solutions for GUI programming and FRP respectively, covering the current state of the art, related research and outlining existing problems. Chapter 4 describes a proposal to address the aforementioned concerns, lists open problems, drafts a thesis outline and lists contributions to date. Appendix A contains an article co-authored with Henrik Nilsson and presented at TFP 2014.

[6]See footnote 6 on page 15.

[7]See `http://developer.android.com/guide/topics/ui/declaring-layout.html`.

[8]Tools such as the Android SDK generate a table of unique identifiers, one per GUI widget, so that basic mistakes —misspellings— can be avoided. Accessing widgets that have been removed, however, is not detected in compile time.

[9]See, for instance, `https://launchpad.net/gladex`.

Chapter 2

GUI programming in Functional Languages

This chapter examines existing solutions for GUI and multimedia programming from a functional perspective, from the imperative and low-level implementations to the abstract and high-level. While much of the discussion revolves around Graphical User Interfaces, the most common way of bidirectional human-computer interaction, many of the ideas are applicable to other forms of communication, including networking and other forms of multimedia.

There are dozens of papers and implementations of Functional GUI programming, only a subset of which are presented here. The text aims at brevity, hopefully introducing just enough detail to help understand existing open problems and the motivation to focus on the issues proposed in chapter 4. Functional Reactive Programming has a broader scope than just Graphical User Interfaces and will be explored separately in chapter 3.

2.1 Imperative Multimedia in Functional Languages

Input/Output in functional languages is often performed using libraries written in C/C++, accessible via a Foreign Function Interface [RM92, Sex87] as effectful computations. Many of those underlying libraries work at a very low-level, such as Hcwiid [P14a], used to access Wiimote devices, and OpenGL [GL], 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 one can store and pass around effectful computations as first-class values. Custom control structures can therefore be defined using higher-order functions. This can be exploited to provide further abstractions to write declarative, succinct effectful code, using monads, functors and applicative functors [PJW93, MP08], which together with their associated laws, enable some forms of equational reasoning [SA07, GH11, HF08].

However, without further abstractions, large programs that do Input/Output tend to look imperative. The strict execution order of the effectful computations imposes sequential thinking and a need to mentally track the implicit program state, which makes reasoning hard [Bac78]. 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 abtractions (`mapM_` on line 40), yet there is no clear, easy way to write it more declaratively, even with applicative functors.

```
25  gameDraw :: IORef WorldState -> IO ()
    gameDraw worldStateRef = do
      worldState <- readIORef worldStateRef

      Nxt.Graphics.begin
30
      let (cameraX, cameraY) = MainPanel.cameraPos
↪                                (mainPanel worldState)

      Nxt.Graphics.worldTransform 0.0 0.0

35    -- draw background
      Nxt.Graphics.drawTexture 0.0 0.0
↪       (MainPanel.backgroundTexture
↪          (mainPanel worldState))
↪       (1.0::GLdouble)

      Nxt.Graphics.worldTransform cameraX cameraY
      -- draw foreground
40    mapM_
↪       (\((x, y), tex) ->
↪           Nxt.Graphics.drawTexture x y tex
↪             (1.0::GLdouble))
↪       (levelBackgrounds $ levelData $
↪          curLevel worldState)
      ...
```

The need for GUIs

Graphical User Interfaces let users discover applications progressively, instead of having to read lengthy manuals before running the program for the first time. This is the result of two factors: designing friendly applications and having standard interfaces.

To provide such standard behaviour, developers use libraries that implement sets of interactive elements, which can be composed into larger and more complex ones. Examples of such libraries are Gtk+ [GTK], wxWidgets [WX] and Qt [QT]. Interactive GUI elements have properties that govern their behaviour and appearance, and associated events. Developers can modify the properties and install event handlers to execute custom computations when, for instance, a button is depressed or the user focuses on it. As an example, the following code builds a simple one-window application with a button that has an icon and a label[2], and installs an event handler (line 10) to print a text when the button is clicked:

[1] `https://github.com/styx/Raincat/blob/master/src/Game/GameGraphics.hs`
[2] `http://code.haskell.org/gtk2hs/docs/tutorial/Tutorial_Port/chap4-1.xhtml`

```haskell
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  set window [windowTitle := "Pix",
              containerBorderWidth := 10]
  button <- buttonNew
  onClicked button (putStrLn "button␣clicked")
  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 [OGS08, p. 522-527] as the one presented earlier. Event-driven architectures [NS79] result in inversion of control [FS97, p. 36-37], leading to difficulty reasoning about programs [Mye91]. Design patterns such as Model-View-Controller [KP$^+$88] move too much logic into the controller, whose codebase grows quadratically[3] with new features and leads to what is informally known as "callback hells" [Edw09, p. 2].

Reasoning about the behaviour of GUI toolkits is equally hard. Widget properties are not plain mutable variables: modifying a property does not guarantee that reading from it at any point afterwards will return the last value written to it, even in the absence of user interaction[4]. The semantics of GUI toolkits are often poorly defined, mainly consisting of their own implementations. Furthermore, many of these libraries (including Gtk and Wx) are *not thread-safe*. GUI functions must be called from the main (UI) thread, controlled by the toolkit itself [GML]. Applications that need to remain in control of the execution loop or do background work must explicitly handle *concurrency*, making the code even more complex.

On the bright side, the resulting functional code is *not substantially worse* than its C/C++ equivalent, and sometimes can be better, even when part of it is imperative [Hea08]. The *performance can be comparable* to that obtained using imperative languages [LGPA13, Swe99, Pet10], making low-level bindings the preferred choice for CPU-demanding multimedia.

Some functional GUI implementations try to define an intermediate layer that wraps the underlying effects. We can distinguish between the purely functional solutions, which render functional code

[3]See footnote 4 on page 8.

[4]For one, the admissible ranges of property values may be smaller than the type can accomodate. For another, animated components may change in size and position, regardless of user interaction.

and will be explored in the next section, and the monadic ones. Examples of the latter include TkGofer [CVM97, VSS96] and Haggis [FJ95]. 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 toolkits, such as Fudgets [CH93]. Haggis is substantially simpler, but makes use of concurrent access to shared memory to implement asynchronous UIs. In both cases, the GUI code is monadic.

Summary

Using bindings to existing GUI toolkits one can applications that adhere to a standard look and feel, without a substantial performance loss. However:

- **Problem 1.1**: Interaction with GUI toolkits imposes an imperative style, an event-oriented structure and concurrency, making reasoning and abstracting harder.
- **Problem 1.2**: GUI toolkits keep part of the application's state in a separate layer, out of the programmer's control. The UIs' operational semantics are not formally defined.

2.2 Purely functional GUI solutions

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. Multiple libraries follow this approach, such as Diagrams [Yor08], Chart [Doc06] and Gloss [Lip10]. Contrary to the imperative case, this kind of code looks declarative, compositional and abstract. The following illustrative example of purely functional Gloss code [Lip12] 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) ]
...
```

Functional GUI toolkits can use a similar approach: widgets and containers are pure values that programmers can adapt and combine. An effectful interpretation function then converts these pure values into GUI-producing computations, or applies changes onto existing GUIs.

Such is the case in Objects I/O [AP98], implemented in Clean [BvEVLP87], in which GUIs are pure values and event handlers perform transformations over the world state. Clean's uniqueness types ensure that the world is never duplicated or discarded, leading to pure transformational code over an explicit state.

Since interactive widgets must handle user actions, produce a visualisation and notify of changes, some purely-functional solutions adopt a function-like view of GUI elements themselves. Fudgets [CH93] 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*[5]. A series of combinators allow fudgets to be connected to one another and, depending on the one being used, determines how the elements are laid out on the screen. A particular limitation of Fudgets is that, because combinators determine both connections and layouts, there is no way to connect visually distant widgets. Gadgets [NR95], a similar approach, tries to work around Fudgets' limitation of one input and output channel per process. In Gadgets and Fudgets, code is more declarative than in imperative toolkits, but both are limited in terms of feature coverage.

```
import Fudgets

main = fudlogue (shellF "Up Counter"
                        counterF)

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

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

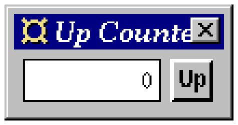


Figure 2.1: *A small Fudgets program and a screenshot of the GUI it generates.* `intDispF` *is an integer display text fudget,* `mapstateF` *keeps a counter, and* `buttonF` *is a button fudget.* `>==<` *chains fudgets from right to left and places them next to one another in the GUI.*

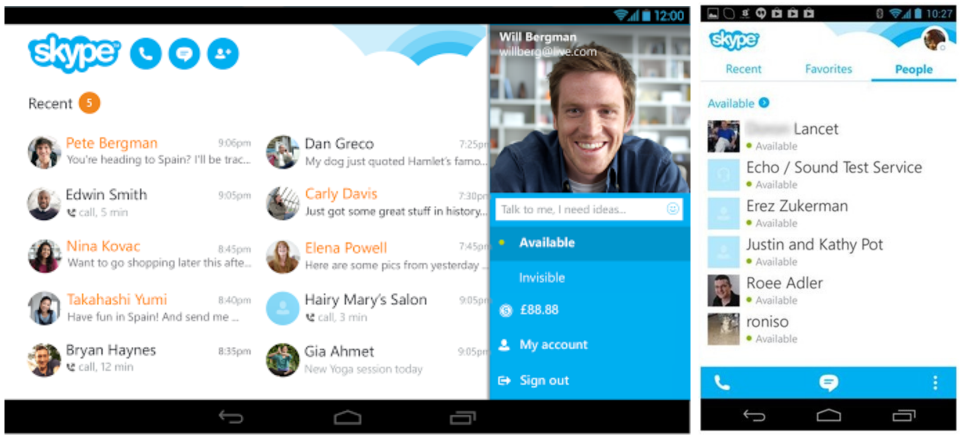
The freedom to define a purely functional abstraction for the domain is both a blessing and a curse: to cover the whole domain, one needs to implement all possible types and operations that may be needed. For GUIs, this implies a type, an implementation and set of operations for each kind of widget supported. GUI toolkits are notoriously large[6], and this results in *very large codebases with high maintenance costs*[7], 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) sets of code, only exhacerbates the maintenance costs.

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

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

[7]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.

Another problem, as explained in the introduction (page 9), is that some of these solutions require that the GUI be coded manually, which is suboptimal and expensive in large projects (expert UI designers tend to prefer to use visual tools to design GUIs [Vic11]). That also affects the adaptability of the interface to new OSs like Android and iOS, in different devices use different user interfaces and changes to the environment (e.g. device orientation) [AnSa] may require switching to a completely different UI without closing the application.



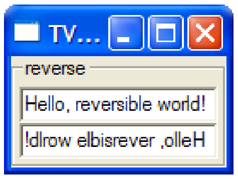
Pete Bergman 9:06pm You're heading to Spain? I'll be trac...
Dan Greco 7:25pm My dog just quoted Hamlet's famo...
Edwin Smith 9:05pm call, 5 min
Carly Davis 7:30pm Just got some great stuff in history...
Nina Kovac 8:45pm Want to go shopping later this afte...
Elena Powell 7:45pm Here are some pics from yesterday
Takahashi Yumi 8:40pm Have fun in Spain! And send me ...
Hairy Mary's Salon 9:05pm call, 3 min
Bryan Haynes 8:35pm call, 12 min
Gia Ahmet 9:05pm New Yoga session today
Talk to me, I need ideas...

Lancet Available
Echo / Sound Test Service Available
Erez Zukerman Available
Justin and Kathy Pot Available
Roee Adler Available
roniso Available

Figure 2.2: *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.*

A limited form of abstraction over User Interface programming is pursued by implmentations that generate GUIs automatically based on the types of expressions. GuiTV [Ell07], a Haskell implementation for type-based GUI generator, can construct widget compositions for functions, to provide the arguments and show the result, eliminating one level of indirection between models and visualisations (see Fig. 2.3).

```
reverseT :: CTV (String -> String)
reverseT = tv (oTitle "reverse"
   defaultOut)
↪                 reverse
```



reverse
Hello, reversible world!
!dlrow elbisrever ,olleH

Figure 2.3: *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* `Strings`*.*

A similar idea is used in iTask [MPA11], a client-server, task-oriented [WJ], (monadic) state-transforming web programming framework for the Clean [BvEVLP87] 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 [AVEP05], 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* [RoRa]. A well-known limitation of this approach was that views cannot be fully adapted[8]. Dynamic (runtime) scaffolding was substituted in new versions by generators that create UI code in different files, which developers can then adapt and modify [RoRb].

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[9].

Summary

- **Problem 2.1**: GUI toolkits are large and complex, and reimplementing them is costly. To get standard behaviour, it must be possible to interoperate with existing toolkits, including all widgets, their properties and events, and full layout control.
- **Problem 2.2**: GUIs must be realizable using visual tools, with full control over the representation. Hard-coded GUIs are not flexible enough, and there may be several valid visual representations and interaction mechanisms for values of the same type.

[8]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 [MPA11].

[9]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.

Chapter 3

Functional Reactive Programming

Reactive Programming [CR96, BDS96] tries to address “callback hells” and the poor modularity of Model-View-Controller [KP$^+$88] by focusing on data-dependencies between view and model fields, updating them when either changes. The functional take on reactivity needs to, additionally, handle state in a referentially transparent way, which Functional Reactive Programming [EH97, CE01, NCP02] addresses as follows:

- *Referential transparency*: Values that change with time are defined as *signals*, which are, conceptually, functions from time to a value ($Signal\ a \approx Time \rightarrow a$).
- *Reactivity*: Signals may depend on present or past values of other signals.
- *Interactivity*: Some distinguishable signals represent user input and signals that depend on them will change when the input does. FRP systems also have identifiable output signals.

The previous definition is deliberately abstract. There are multiple variants of FRP, some built around different basic constructs, but all carrying a sense of time-variance and reactivity. The main purpose is to enable thinking about *what things are over time* (declarative programming), instead of how to convert what things were into what they have become (imperative programming).

Publications and discussions on FRP usually focus on one of the following (non-exclusive) topics:

- Language, semantics and reasoning.
- Evaluation strategy and efficiency.
- Effects, visualisation and/or connections to other systems.
- Modularity and programming style.

This section presents each topic separately, exploring *design decisions* made by different FRP variants, *open and solved problems* and the relations between them. The order of presentation does not mean to reflect importance; it has been chosen so to facilitate the exposition.

Chapter 4 and appendix A present a proposal for Functional Application Programming based on the reactive paradigm. Since I expect future research on that approach to make use of FRP, this chapter gives a general view of the field, with more detail than in previous chapters. However, it does not constitute an exhaustive review (for such an analysis see [BCVC$^+$12]) nor an introduction to FRP (see [CNP03]).

3.1 Language, semantics, reasoning and guarantees

3.1.1 The nature of signals

Time Multimedia applications often portray environments whose time dimension appears continuous[1], which can be represented mathematically by the real numbers. The most common and efficient machine approximation are floating point numbers (e.g. `float` and `double`).

However, not all applications need such continuity. A view of time as discrete unit steps is sometimes sufficient and renders simpler solutions [200b]. In those cases the time dimension can be the natural numbers, giving a correspondence between *streams of values* and signals.

More generally, the requirement is that *time be ordered* [EH97, Ell09], with evaluation happening at strictly increasing times. Because simulations always have a beginning, most FRP implementations represent time as $\mathbb{R}^+$ and $\mathbb{N}$. Some variants add infinity and negative infinity to the time domain, at least at a conceptual level, to capture *what has always been* and *what will never be* respectively. This allows some equations to be simplified [EH97, p. 4].

The original formulation of FRP [EH97] worked with continuous time, as do AFRP/Yampa [NCP02] and Uniti [Rov11]. Discrete-time variants of FRP include Elerea [Pat11] and Ordrea [ORD].

Kinds of signals Just as not all applications have the same requirements over time, the same can be said about signals: some change continuously, others change only sporadically. In general terms, we can distinguish between three kinds of signals:

- Continuous-time continuously-changing signals.
- Continuous-time sparsely-changing signals (analogous to step-functions).
- Discrete-time signals, defined only at discrete points in time.

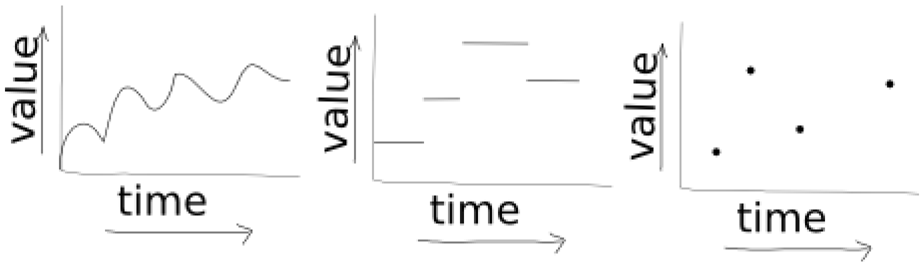


Figure 3.1: *From left to right: a continuously (or frequently) changing signal, a sparsely-changing signal, and an event-stream.*

Not all FRP variants use all three categories, which motivates a distinction between *Single-kinded* and *Multi-kinded* FRP. Fran [EH97], SOE FRP [Hud00] and Reactive-banana [HA1] are multi-kinded, while Yampa [NCP02], Elerea [Pat11], Elm [CC13a] and Flapjax [MGB$^+$09] are uni-kinded.

Continuous-time signals have traditionally been called *behaviours*, step-function-like signals are sometimes called *step-signals*, and signals defined only at discrete points in time have been called *events* (or, more properly, *event streams*). Event-based GUI toolkits have a closer correspondence

[1]Whether time *really* is continuous is out of the scope of this work.

with FRP variants with a notion of events [MGB+09, CK04, Cou01, Sag00][2]. Games are most often implemented in variants with continuous time [CNP03, NP14, P14b, Che05].

Further variations are possible. N-ary FRP [SN10a] 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.

Structural dynamism and higher-order Interactive applications, especially games, change dynamically during execution, with new elements being introduced and others removed. In FRP, this is accomplished by allowing signals to change or "become" other signals, captured in Fran with the `untilB` combinator:

```
redBlue u = buttonMonitor u `over` withColor c circle
  where
   c = constantly red `untilB` lbp u -=> constantly blue
```

The signal *c* defined in this example (adapted from [EH97]) should behave like `constantly red`, called the *subordinate* signal, until a certain property is fulfilled (`lbp u`, a mouse left-button press), in which case it should become `constantly blue`, known as the *residual* signal [SN10a, p. 6].

More general forms of dynamism exist in FRP. Yampa and others introduce dynamic collections [CNP03], e.g. collections of signals that new elements can be added to or removed from during execution. Higher-order signals, which carry other signals, can sometimes be "flattened" with given combinators [Pat11], turning on the internal signals as the output at specific times.

This kind of structural changes involve important concerns, such as network optimisation and determining the value of the signal at the time of change/switching. This question was explored by Nilsson et al. [NCP02, 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 [CNP03, p. 8]. Some FRP implementations like Yampa [NCP02] provide delays and initialisation combinators to break looped switching[3], while others like Elerea [Pat11] introduce these delays automatically in the presence of *circular dependencies*.

The task-model of signals described in O-FRP [200a] and Monadic FRP [vdP13] also allows signals to be turned off (in both cases, using monads), giving them temporary lifetimes. This could be simulated using just plain, infinite signals, but the conceptual separation between tasks and ordinary signals may facilitate thinking and a substantially more efficient implementation.

3.1.2 Semantics and guarantees

Work on FRP has been frequently applied to multimedia, game programming and user interfaces. Precise definitions of the language and its meaning have not been a major concern in those domains. On the other hand, hardware simulation and real-time systems are some of the use cases that can be aided by having formal semantics.

[2]This similarity is progressively disappearing, since in new GUIs, animations –conceptually continuous– play a central role.

[3]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.

FRP was given denotational semantics by Elliott and Hudak [EH97], to "facilitate and guide implementations". A similar approach was used by Courtney [Cou04]. However, these semantics can only be approximated, and there are discrepancies between implementations and the ideal meaning. As an example, take a counter that increases in one unit every time it is sampled[4]. This construction is completely dependent on the sampling rate, diverging in the limit (as the sampling frequency increases), yet it is a common and useful FRP construct. In practice, some FRP implementations sample based on CPU load, which even makes programs exhibit non-deterministic behaviour.

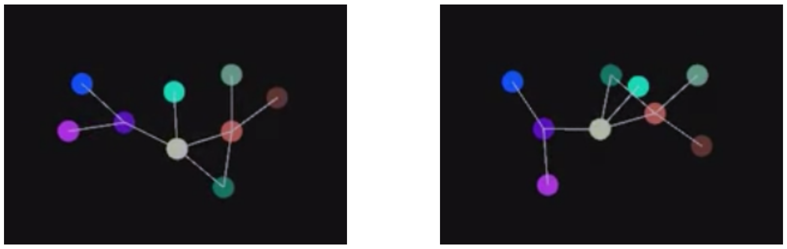


Figure 3.2: *Because sampling times depend on CPU load, some FRP libraries exhibit non-determinism. For example, two runs of the same force-based graph layout algorithm, with identical initial conditions and force coefficients, may converge onto different stable configurations.*

Conversely, think of an event that should conceptually fire at π time. Such an event would clearly be missed. Adding a margin of error could make the event fire twice or more in a time period, and correcting the second event would not prevent a third (fictitious) one from firing. Implementations like reactive-banana [HA1] address this problem by not allowing to convert an arbitrary behaviour into an event. For a discussion on how to avoid duplicated and missing input events, see [Ell09].

A branch of research has been focused on giving different FRP variants more precise (operational) semantics, that enable reasoning about the implementation, the output values and resources. RT-FRP [WTH01] and the later E-FRP [WTH02] followed this path by reducing the language to a minimum core, adding static types and minimal extensions. Priority-based FRP [KTZ07] built on this work by adding priorities, enabling near-backwards compatibility while facilitating a suitable implementation for hardware analysis and real-time systems. Krishnaswami [KB11, KBH12, Kri13] defines a stream-based FRP language with higher-order constructs and temporal operations, constrained so as to reason about necessary resources and providing proofs of space leak avoidance[5].

On the opposite end we find implementations with no formal semantics, such as reactive-banana [HA1]. Concurrent variants, especially GUI-oriented ones [CK04, CC13a], pose difficulty reasoning about programs and exhibit non-determinism, relying on *eventual consistency* [ANVdB13] and going to great lengths to break infinite loops caused by circular dependencies [Pat09, p. 10].

Testing Because some signals represent user input, the only way to test some signals that depend on them is to provide input for testing purposes. In the following example, to test `signalText`, we need to provide credible `mouseInput` (manually or automatically).

[4]Counters rely on state keeping, which will be presented later on.

[5][KB11] gives semantics of FRP based on ultrametric spaces, [KBH12] gives a language based on linear types, and [KBH12] gives one without linear types, motivated by trying to write more abstract code.

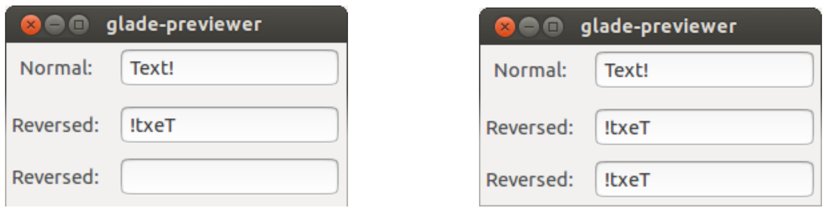


Figure 3.3: *Some asynchronous implementations rely on* eventual consistency*, resulting in* glitches *(short-lived inconsistent states). In this application the text in the first text box should be immediately shown reversed twice. There is a short delay (left) until the state becomes consistent (right).*

```haskell
signalText :: Signal String
signalText = show <$> mouseInput
```

Manual tests are not always reproducible, limiting the conclusions we can draw from them. FRP variants with signals as first-class citizens hide data dependencies in implementations (the type `Signal String` does not reflect a dependency on user input). To test these variants automatically, a testing layer would need to *overload all input signals* (e.g. `mouseInput`) with fake-data-producing definitions, merely for testing purposes. This approach is used in the Android SDK [AnSb].

Making the inputs explicit, as Arrowized FRP [CE01, NCP02] does, eliminates this problem altogether, and allows us to use existing tools (e.g. QuickCheck [CH00]) to test our implementations[6].

3.2 Efficiency and evaluation

Efficiency has been a major concern in FRP implementations since the very beginning [LH07, Ell09], and it is strongly tied to the semantics of the language. The major approaches to improve the efficiency of FRP applications have been:

- To restrict the language to guarantee well-behaved expressions.
- To add structure, in order to optimise specific cases.
- To change the evaluation model, avoiding unnecessary computations, propagating changes more quickly and executing subnetworks in parallel.

3.2.1 Looking back and state

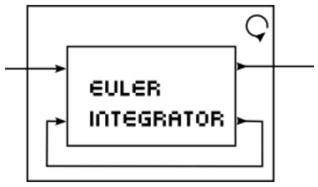
Signals can depend both on present and past values of other signals. This ability to inspect the past requires that the whole history of other signal be saved indefinitely, leading to memory leaks [CE01]). If a great part of that history is evaluated at once it may take a long time to compute, causing a time leak [CE01, LH07]. Both are common symptoms in implementations that use functions [EH97] and streams [Hud00] to model signals.

One way to tackle this situation is to let signals inspect other signals only at *the present time*, but to let them keep samples (state) for later. This is *less expressive than full time access*, since knowledge of previous values of signals is restricted to the times at which they were sampled.

[6]Of course, not every input generated by QuickCheck will be credible for the specific application. For more details, see the cited article.

Keeping state can be done explicitly, or be hidden using monads [LJ95] or continuations [Rey93, Wad97]. The latter is used in UniTi [Rov11] and Yampa [NCP02]. To guarantee that continuations are passed in the right order, without discarding states or branching [KPVES01, p. 99], combinators are used to hide the plumbing.

Arrowized FRP [CE01, NCP02] goes one step further than signal-based FRP variants, by not giving developers access to arbitrary signals, but rather, to *signal functions*. Signal functions are connected using *arrow combinators* [Hug00], which results in declarative constructs that resemble circuit design. Signal functions are not allowed to sample signals at arbitrary times, and Arrowized FRP has been shown to avoid different kinds of space leaks [CE01, LH07]. Further benefits of Arrowized FRP are described in [Scu11, sec. 3.5].



State in these variants can be kept either with stateful signal (function) generators, or by creating feedback loops (see figure on the left). In these loops, part of the last known output becomes part of the current input. The implementation of these loops can be done using the least-fixed point combinator, in which case users need to take special care so as to avoid undesired infinite loops or undefined values [FX07].

3.2.2 Evaluation model

FRP signals may change continuously, but their realisation on the screen takes place by sampling at discrete points in time. Higher-fidelity may be achieved by sampling more frequently, but if a value has not changed, recomputation may be unnecessary.

This gives rise to two ways of processing the network: by sampling (output) signals as frequently as possible, known as *pull* evaluation, or by propagating changes forward, known as *push* evaluation.

In pull evaluation, the output signals are sampled again and again, until the application is closed. This may result in unnecessary work, since only a small subset of the network, or nothing at all, may need to be recomputed [Jel08]. Pull evaluation is used by Yampa [NCP02], and results adequate for games [CNP03, Che05, NP14] and physics simulations.

In push evaluation the output signals do not change unless there is a change to one of the input signals. This may result in minimal recomputations and fast time-to-screen, but it comes at a cost:

- The effect of time-driven signals that should change continuously is not perceived until an input event provokes network recomputation. A common workaround is to force such recomputation by triggering events at regular intervals [HA1].
- Depending on the implementation, if several events happen very close in time, their effect may be propagated independently, instead of calculating the output taking both changes into consideration, resulting in unnecessary computations.

Push evaluation is used by most modern implementations, including reactive-banana [HA1] and Elm [CC13b]. Most event-based variants are push-based, and connections to GUI toolkits are more natural if events and push-evaluation are used [Cou01, 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 [Ell09, Ams12] uses this approach to only recompute continuous signals, but not events, as frequently as possible, while reactive values are calculated only when there are actions to react to. A similar approach is used by Frappé [Cou01], in which time-dependent signals can be distinguished from time-independent, reactive ones. Although, strictly speaking, Frappé is a push-based implementation; only in the presence of time-dependent signals is the network polled at regular intervals.

Concurrency and Parallelism Traditional implementations of FRP have been synchronous and one-threaded. This has often been cited as a difference between FRP-backed GUIs and other functional toolkits such as Fudgets [Cou03, p. 15]. Synchronicity is the simplest way to implement deterministic, well-defined, invariant-keeping systems.

As computational requirements grow and the demand for precision not so high in many multimedia applications, these requirements have been commonly relaxed. This topic is of particular importance at the present time, as the number of cores per machine is growing [OH05] and Functional Programming is often advertised as capable to taking advantage of such capacity [Jon89].

In this context, we should distinguish between deterministic implementations that use parallelism and concurrency and implementations that rely on *eventual consistency* [Vog09, ANVdB13]. The former kind has been used in Parallel FRP [PTS00] and Reactive [Ell09]. Implementations such as Elm [CC13b] and Netwire [Net] relax such restriction, and rely on the fact that applications will eventually come to a balance. This results in *glitches* [BCVC^{+}12, p. 6], temporal incoherences between FRP invariants and the network state (Fig. 3.3).

Breaking loops In all cases, including synchronous implementations, circular dependencies may lead to never-ending data propagation. Variants like Parallel FRP [PTS00] do not allow circularity, while others like Elerea [Pat11] break loops by identifying each signal, detecting loops and introducing delays.

Loops may be difficult to detect, both in compile time and in runtime. In the presence of asynchronous evaluation, it may even be difficult to observe by users themselves, as data propagation can make programs progress and respond while never reaching a standing point.

3.2.3 Adding structure

Optimisations to FRP networks can also be achieved by distinguishing between different kinds of signals and/or transformations. This lets implementations exploit known laws and equivalences, reduce the size of data structures and process them more efficiently.

Nilsson et al. [NCP02] distinguish between identity, constant, pure and other kinds of signal transformations, exploiting arrow laws to reduce expressions. Generilised Algebraic Datatypes (GADTs) helped avoid certain runtime checks, resulting in a more efficient execution [Nil05].

Without having to add such distinction, works on Causal Commutative Arrows [LCH09] also exploits the arrow laws for, in specific conditions (arrow commutativity, no circular dependencies, etc.) reduce any expression to a minimal normal form that accepts more efficient execution.

Work on Push-Pull FRP [Ell09, Ams12] took advantage of the event/behaviour dichotomy to provide efficient implementations. Similarly, Frappé [Cou01] detects time-dependent signal functions and uses pull evaluation for them and push evaluation for all other signals.

Work on N-ary FRP [Scu11, chapter 8] explored optimisation opportunities in a form of dependently typed, multi-kinded FRP extended with Signal Vectors.

3.3 Visualisation and connection to external systems

Interactive and visual applications have been the major domain of application of FRP. Similarly to the way we explored GUI toolkits in chapter 2, FRP implementations have either enabled connections to existing graphics and GUI toolkits, or used functional abstractions. Most modern FRP variants are not tied to a particular backend or GUI toolkit, and very few maintain several independent backends up to date.

Connections to GUI toolkits are easier in push-based, event-aware FRP, like Frappé (Java Beans, [Cou01]), reactive-banana (HTML, wx, [HA1]), grapefruit (wx, [Jel09]), Elm (HTML DOM and canvas, [CC13a]), KSWorld [OFLK12]. Pull-variants have also been connected to existing toolkits, such as in wxFruit [Rob04], although this connection results more complex and artificial.

When it comes to low-level visualisation, both variants have been equally successful. Yampa has several known backends (SDL [SDLa, SDLb], Glut [YGL], Fruit [CE01], HGL [HGL] and OpenGL [GL]), Fran uses Win32 [W32], Reactive has been connected to FieldTrip [RFT], and Elerea to OpenGL and LambdaCube3D [L3D].

FRP has been used in multiple other domains, including computer vision [RPHH99], robotics programming [HCNP03] and web applications [Prz, MGB$^+$09].

3.4 Style and code modularity

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.

One of the specific 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 [Rus03]. 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[7].

A second problem is that, at least in user interfaces, visually-related elements may not be conceptually related. Consider, for instance, the screenshot from Xournal [Xou] shown in figure 3.4. 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 (center left), by scrolling down the page (center 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

[7]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.

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:

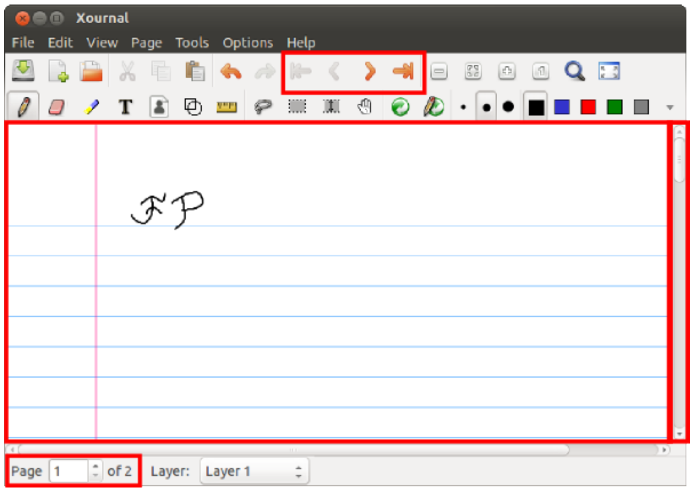


Figure 3.4: *Figure : a Screenshot of Xournal, showing four different ways to change the page number shown by the program.*

```
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`
    ...
  ]
```

As can be seen, we need `currentPage` to define `toolbarButtonRight`, `pageSelectionEntry` and `pageArea`, but we need all three (and probably many others) to define the value of `currentPage`. These mutually dependent elements all have to be defined together in the same part of our code, an obstacle for concern separation and modularity that only gets worse as the codebase grows.

Directly bi-directional data dependencies are omnipresent in software with user interfaces, as some view elements are mere interactive visualisations of parts of the model. In FRP, data dependencies are uni-directional[8], which results in (error prone) code duplication, as every relation between the model and the view needs to be written in both directions.

[8]In shallow embeddings, using the host language's $=$ (or $<-$) definition symbol, which is uni-directional, imposes such restriction.

3.5 Summary

In this chapter we have examined the field of Functional Reactive Programming, with emphasis on the design decisions made by different variants. With respect to GUI programming, we can highlight the following aspects:

- **Problem 3.1**: In order to provide a standard GUI, we need to be able to connect FRP implementations to existing GUI toolkits. Signal-based FRP variants have difficulty adapting to such toolkits; event-based variants do not accommodate continuous animation as well as signal-based ones.
- **Problem 3.2**: Both synchronous and asynchronous implementations may exhibit non-deterministic behaviour, and rely instead on eventual consistency. In both cases, reasoning about the state of the program is hard.
- **Problem 3.3**: Implementations must be efficient, avoid space leaks and guarantee short change-propagation time. Both the signal kindedness and the evaluation strategy impact the performance of the implementation.
- **Problem 3.4**: Circular dependencies, ubiquitous in GUI applications, may lead to infinite loops and, without the proper abstractions, to poor modularity and code duplication.

Chapter 4

Thesis proposal and work to date

In the previous chapter I identified several problems with existing solutions to write GUI software in functional languages. The objective during my research is to focus on large-scale concerns, mainly:

- Visual UI design, UI adaptability (plasticity) and connecting to existing GUI toolkits.
- Application architecture, concern separation, and declarative programming style.
- Dynamic structure.
- Efficiency.

The initial details of a proposed solution were included in the paper "Bridging the GUI gap with reactive values and relations", a revision of which is included in appendix A. In this chapter I introduce the approach described by that paper and list problems that remain to be addressed. I also give a tentative outline for my thesis and describe the contributions made to this date.

4.1 Bridging the GUI gap

Key elements of the proposal The key idea is to extend reactive time-varying entities, like FRP's signals, with a set operation, making them *typed, time-varying, settable, gettable values whose changes we can listen to.* In this chapter we will refer to them as *Reactive Values*, and they are represented by the parametric type constructor ∀`a.ReactiveRW a`. Additional constructors `ReactiveRO` and `ReactiveWO` can be used for read-only and write-only reactive values respectively. Following the example in section 3.4, we could have:

Conceptual page number	`currentPage           :: ReactiveRW Int`
Text entry in bottom toolbar	`curPageTextEntryText :: ReactiveRW String`

We define primitives and combinators to create and compose reactive values. Combinators let us, among other things, apply point-wise transformations to existing values, by *lifting* pure functions, and focus on specific parts of existing values, by using *lenses* [FGM+07]:

```
   Number in the text   curPageEntryAsInt :: ReactiveRW Int
entry in bottom toolbar curPageEntryAsInt =
                            liftRW (read, show) curPageTextEntryText
```

`liftRW` needs a function to transform values being read and one to transform values being written. To lift only in one direction, the functions `liftR` and `liftW` render read-only and write-only reactive values respectively. A family of n-ary lifting functions `liftR2`, `liftR3`, . . . is also given.

Uni- and bi-directional dependencies between values can then be defined using the rule-building combinators `<:=` and `=:=` respectively. The constraint on page 27, which then required knowledge of several UI elements, could now be expressed *separately* for the bottom toolbar textbox as:

```
   Update the other   curPageEntryAsInt =:= currentPage
 when either changes
```

This gives a general *uniform reactive interface* suitable to connect models and views (and possibly other systems such as network and disk I/O) using a declarative, compositional style.

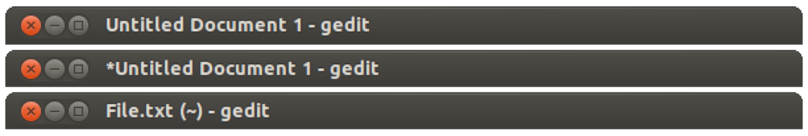
Separation of Concerns The proposed approach lets us write rules for the text box based only on a conceptual page number, regardless of any other UI elements that might influence either of them. It also lets us *separate the definition of the reactive element from the rules that affect them.*

Constraints can be expressed in separate modules, they do not need to be defined where either the text entry or the page number are defined. Rules can be organized and *grouped by the features they try to capture*, resulting in better concern separation[1].

Constraints may also be parametric, as depicted by the following rule-building function, which implements the common behaviour seen in figure 4.1 below:

```
  View: Window caption   titleBar :: ReactiveRW String
   Model: FilePath, if set         -> ReactiveRW (Maybe FilePath)
   Model: File modified?           -> ReactiveRW Bool
          Program name             -> String
                                   -> ReactiveRule
                         titleBar captionRW curFileRW modifiedRW programName =
      Directional rule     captionRW <:= liftR2 titleShould curFileRW modifiedRW

    Point-wise title      where titleShould f m = modified m ++ fileAsTitle f
            assembler                             ++ "␣-␣" ++ programName
 Modified and not saved         modified m      = if m then "*" else ""
   Name if never saved          fileAsTitle     = fromMaybe "Untitled␣document"
```



Untitled Document 1 - gedit

*Untitled Document 1 - gedit

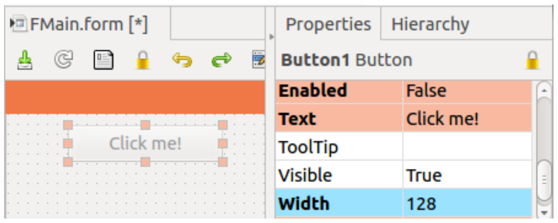
File.txt (~) - gedit

Figure 4.1: *Common behaviour for programs' title bars, showing the current file name or location (if available) and the program name. A * symbolises modified, unsaved files.*

[1]Based on my own experiments with mid-sized (~10K lines of code) programs [KGI, S12].

This makes it possible to separate rules in libraries that implement common application behaviour, letting developers focus solely on the core of the problem they want to solve. For further abstraction, type classes could capture whole models or views with specific characteristics, giving our choreographic functions, like the previous one, shorter and more declarative signatures like (`ModelWithFiles m, ViewWithTitleBar v => m -> v -> ReactiveRule`).

Connecting to existing GUI toolkits Reactive values have a clear correspondence with existing GUI toolkit properties and events. This makes it trivial to connect to existing imperative GUI libraries, by just giving wrappers to see any widget property as a reactive value. Creating such wrappers can be simplified and automatized, making it feasible to maintain over time.



Enabled	False
Text	Click me!
ToolTip	
Visible	True
Width	128

```
-- Library functions
buttonEnabled :: Button -> ReactiveRW Bool
buttonText :: Button -> ReactiveRW String

-- Our definitions
button1Enabled :: ReactiveRW Bool
button1Enabled = buttonEnabled button1
button1Text :: ReactiveRW String
button1Text = buttonText button1
button1 :: GtkButton -- Defined elsewhere
```

Figure 4.2: *Unlike in libraries such as Fudgets, in this proposal each widget property is seen as a separate reactive value, giving fine-grained control.*

The given implementation works with GTK+ GUIs created with visual design tools such as Glade [Gla], using Glade library functions to load GUI specification files and specific reactive GUI libraries to turn widget properties into reactive values (Fig. 4.2). The connection library to GTK+ is small, suggesting it would be feasible to write backends for wxWidgets [WX] or Qt [QT].

4.2 Open problems

This section describes aspects of the proposed solution that remain to be studied. It is comprised of three blocks: 1) dynamism, 2) semantics and evaluation model, and 3) extensions to the representation of reactive values. They are introduced in order of priority, with dynamism and semantics forming part of the foundations upon which more novel features (sync-async evaluation models, first-order changes and timeline integration) can be explored.

Dynamism The previous approach works well for static interfaces. Ideally, one would also want a way to “uninstall” or disable rules, that is, make them no longer valid.

The elimination of a reactive value (for instance, because a widget is removed) currently “works” only because data propagation does not find an end point, or simply because the widget never sends any data. For instance, given a reactive value `textEntry1Text :: ReactiveRW String` representing the text of a Gtk+ text entry `textEntry1`, and a reactive value `modelText` of the same type, the following reactive rule:

```
modelText <:= textEntry1Text
```

is translated into code equivalent to:

```
onEditableChanged textEntry1 $ do
  text <- entryGetText textEntry1
  set modelText text
```

If `textEntry1` is removed, the event handler is also deleted by Gtk, and so the rule is disabled without any special treatment. This, however, relies on Gtk+ doing the right thing. The workings of other GUI toolkits could be different, and so a backend-agnostic interface should be provided. Also, it only works for GUI elements and it is not a solution for the general case.

Dynamism is non-trivial and must be handled with special care. Consider, for instance, the following rule, which connects the same reactive string to two status bars:

```
liftW2 (,) statusBar1String statusBar2String <:= liftR (\x -> (x,x)) appStatus
```

This rule hides two independent dependencies inside. If we remove one of the status bars, we could then consider that the end-point of the rule is not fully defined, and therefore the whole rule should be disabled. But we could also consider that only part of the rule (propagation to the eliminated status bar) should be disabled[2]. We could manually solve this situation by writing two independent rules, but maybe the reactivity implementation could do some work behind the scenes for us by removing "dead" parts of reactive networks. This problem was also enunciated in [SN10b, p. 27].

Semantics The defined reactive approach does not include any formal, rigorous definition of the semantics of getting, setting, combinining and syncing reactive values. Reasoning about complex data propagation networks can be, as examples in other data-flow frameworks show, substantially hard.

Even though FRP does not allow us to separate signal definitions from value declarations, I believe FRP could be a suitable layer to formally state that meaning, possibly by extending signals, or by making reactive values signal functions.

One requirement of doing so would be to clearly state the *time dimension* of reactive programs. In particular, multiple data dependencies for the same reactive value can lead to ambiguous, non-deterministic or ill-defined reactive values at specific times. This may happen if several rules exist for the same reactive value (implicit in the presence of cyclic dependencies and bi-directional rules. Circular dependencies can, as seen in previous chapters, lead to endless data propagation[3], and stating the requirements for efficient convergence would be necessary. Currently, because real (user) time is used, the time that it takes for changes to propagate is theoretically unbounded.

Related literature which could serve as a starting point to study this problem includes [Cou04, Chapter 4][BG92, PHP87, Vog09, ANVdB13, LL91].

Evaluation model The current implementation is fully asynchronous, mainly driven by the same concurrency necessities seen in chapter 1. More sophisticated implementations, combining asynchronicity for GUIs and deterministic, synchronous propagation for models, could be designed. The evaluation model and the semantics influence each other, and we should expect several stages between one and the other until converging on a specific design.

[2]For this particular reason, the current library implementation does not provide `liftW2`.

[3]The current implementation does analyse whether the value has changed, halting propagation if, and only if, a fix-point is reached.

Potential optimisations, such as parallelism, could be introduced at this reactive level. Preliminary tests using the FRP implementation Yampa showed that multiple changes to its API would be required, since Yampa's parallel combinators act on (individible) Functors, not (divisible) collections [CNP03]. If our reactive library were re-implemented on top of FRP, a more specific API should be implemented, based on divisible structures that can be processed parallely. However, since the net benefit of parallelism depends on the size of data structures and the number of cores available [SSOG93], some workload subdivisions may only be feasible at application level.

The nature of change When new values are propagated along, the change itself is lost. Consider, for instance, a list widget and a list of strings, synchronized as follows:

```
let listItems :: ReactiveRW [String] -- Widget property holding the elements
    listItems = ...
    items :: ReactiveRW [String] -- Model value holding the elements
    items = ...
in items =:= listItems
```

Given this implementation, we cannot distinguish between adding a new item to `items`, and clearing the list and subsequently adding all the elements one by one, plus the additional element. We also cannot distinguish between moving one element and loading a brand new re-sorted list. One alternative is computationally more expensive than the other, and information is being lost, resulting in a different final widget state with potentially undesirable visual effects (fig. 4.3).

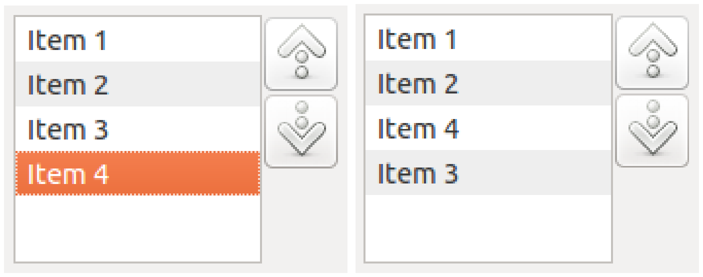


Figure 4.3: *With an element of the list selected (left), changing the order by pressing the up-arrow button will actually load a new list, clearing the selection (right). In this example, Item 4 should have remained selected.*

One possible solution is to pass along change information together with the new value, that is, to include, for every change, the difference between the old value and the new one (and possibly the reason behind that change). In our example, the implementation of reactive data propagation for lists could know how to synchronize two lists when the only change has been the addition of a new element at a specific position.

Additional related literature includes [Aca05, LW10, Car02, AMP]

Timeline integration If time was made explicit, at least conceptually, a further question would be if several reactive systems with different notions of time (continuous and discrete) could be synchronised using reactive rules (equations). Consider, for instance, a game UI with a continuous notion of moving pieces on a board, and a discrete, conceptual notion of the same board. Synchronizing these two structures would require having a clear time-mapping, from the discrete to the continuous-dimension, and back.

Together with adding first-class changes, this would help address a long-standing issue of data duplication across different non-inclusive models of the same data, common in GUIs and games [HF05, p. 6].

4.3 PhD Plan

A tentive plan for my thesis follows:

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. More general solutions, such as Functional Reactive Programming, provide a pure functional view of interactive, reactive processes, which can then be connected to effectful backends for specific domains such as Graphics, UI, sound, etc. FRP has resisted an efficient implementation, and existing FRP libraries sacrifice determinism and abstraction in the name of performance.

FRP requires that signals be defined by their values over time. This facilitates reasoning about the execution, but at the cost of modularity and concern separation. Also, FRP's uni-directionality of data dependencies results in code duplication in the presence of circular dependencies, ubiquitous in User Interfaces and multimedia.

This work presents an abstraction for reactive, time-varying values in which declarations and data dependencies are separated. A relational language is given to transform these *reactive values*, to observe them, and to define bi-directional and uni-directional dependencies between them. This results in increased modularity, to the point that common model-view coordinations can be abstracted, separated from the application completely and reused across different programs. A definition of Model and View is provided, and a combinator language for applications with GUIs facilitates the definition of larger applications from smaller ones.

To achieve efficiency without sacrificing model invariants, we give an implementation strategy that uses synchronous propagation (with well-defined, deterministic semantics) for intra-model dependencies and asynchronous, non-deterministic propagation for extra-model dependencies.

An implementation, connected to existing GUI and multimedia libraries, is provided, together with comprehensive examples. This illustrates the resulting benefits in terms of clarity and modularity, feature coverage of the supported GUI toolkits, and the low maintenance costs of this solution.

Table of contents

1. Introduction
2. Technical background. This chapter should be an extension of chapters 2 and 3, with further examples to facilitate the exposition, a short introduction to FRP, and to a review of other (imperative and OO) reactive approaches.

3. Reactive values. This chapter should include a definition and introduction to reactive values, together with the common operations that can be applied on them, including lifting, lens application, and reactive rules.
4. Synchronous reactivity. This chapter should explore how to “reactivate” application models and give a formal denotational and operational interpretation of reactive values over time.
5. Asynchronous reactivity. This chapter should explore reactive rules that are asynchronously propagated. This will be necessary to connect to existing GUI toolkits, to other concurrent applications, and for network communication.
6. Connections to the outside world. This chapter should provide implementation details of the connection to specific existing I/O backends, including GUI toolkits, OS files and network.
7. Application combinators. This chapter should define applications, composed of GUIs, models and rules, and a combinator language to define complex applications out of simpler ones. It should also include choreographies, that is, template applications that can be instantiated onto specific models and views to create additional connections on them, providing general-purpose functionality.
8. Case studies. This chapter should introduce three non-trivial examples: one of synchronous reactive models, another of asynchronous model-view communication, and a final example that uses choreographies and application composition.

4.4 Work to date

Over the course of my first year of PhD, I have carried out the following activities:

- **Papers**
  - I have written a paper, together with my supervisor, outlining a preliminary solution to the aforementioned problem based on mutable, thread-safe variables with rule-building combinators. This paper was accepted for presentation in Trends of Functional Programming 2014, but was rejected for publication in the proceedings. Part of the reviewer’s comments have been included in a version of the paper attached as annex I.
  - I have written a draft paper, together with my supervisor, on using QuickCheck to specify and test temporal properties of FRP applications. Due to the correspondence between FRP and Temporal Logic [JN11, Jel12, Jel13, SN10b, p. 27-34], the tests themselves have declarative specifications. I have written multiple tests both of Yampa properties and for a game, to prove tunneling effects and detect bugs in the physics subsystem.
- **Software**
  - I have included extensions to the existing reactive library on which these ideas are based. In particular, I have extended the library with lenses.
  - I have written a game using the Functional Reactive Programming implementation Yampa, in order to explore the possible difficulties of using FRP in the real-world. This game has been made available online [P14b].
  - I have used that game to explore the introduction of parallelism in Yampa. So far, this has been done at the game level, since Yampa’s combinators act on Functors, not just

any form of collections [CNP03]. Parallelism, often based on “divide an conquer”, could for instance be introduced in Yampa if we could split collections in smaller subcollections. That approach could also have its drawbacks: the net benefit of introducing parallelism often depends on the size of data structures and the number of cores available [SSOG93]. Knowledge of the problem at hand, which is mainly present outside Yampa, might enable smarter subdivisions, such as space partitioning [Gre09, p. 467].

  – The aforementioned game was the culmination of a series of increasingly-complex FRP examples, which could evolve in a published tutorial on FRP and game programming. I have received messages from members of the FP community expressing interest in such a publication.

  – I have taken over the maintenance of Yampa, contributing a considerable amount of documentation and communicating actively with other users of FRP, in order to understand other problems that the community may see in FRP or in this particular variant that may not be apparent to me or my supervisor.

- **Talks**
  - The aforementioned Yampa game was used by my supervisor, Henrik Nilsson, to present a tutorial at PPDP 14 on Declarative Game Programming [NP14].
  - I gave a talk on Reactive Values and FRP at the London Haskell Meetup, in June 2014. The aforementioned game was presented at that talk.
  - I gave two talks on the same topic during internal activities of the Functional Programming Lab (FPLunch, FPLab Away Day).
- **Additional research**
  - I have identified and classified over 150 papers specific to the area, compiled in an online database.
  - I am in the process of compiling a database with details on (so far) 31 FRP implementations, 9 reactive programming implementations, 50 related frameworks and libraries, 10 Functional GUI libraries, 9 imperative GUI bindings, 4 imperative graphics bindings, and 5 pure graphics manipulation libraries. This database contains details on each implementation, including some of the framework’s features, the relations between different implementations, details on how maintained they are to this date and their domain of application.
- **Courses**
  - At the Midlands Graduate School 2014, I attended courses on Category Theory, Functional Reactive Programming and Dependently-typed Programming.
  - Since the beginning of October 2014, I have formed a regular study group of Category Theory together with first-year PhD students Jan Bracker and Jon Fowler, meeting weekly to discuss problems taken from study books.

I believe that lacks in other areas, such as Temporal Logics and Parallel Programming, can be addressed more easily by myself, since those are topics that I studied in the past.

Nevertheless, some expert knowledge could be acquired by doing an internship during my PhD. In particular, I have located and spoken to employees of several companies that already are applying or are interested in applying Functional Programming and may consider the possibility of accepting me as an intern. This, however, would be limited to a short period of time (possibly between two and four months), just enough to acquire the necessary knowledge about one particular problem, and always strictly related to my PhD studies.

Appendix A

Paper: Bridging the GUI Gap with Reactive Values and Relations

Bridging the GUI Gap with Reactive Values and Relations

Ivan Perez and Henrik Nilsson

Functional Programming Laboratory
School of Computer Science
University of Nottingham
United Kingdom
{ixp,nhn}@cs.nott.ac.uk

Abstract. Despite a rapidly growing number of successful, real-world applications implemented in functional languages, there is still arguably a lack of good frameworks for GUI programming in a functional setting. Of course, one of the many imperative frameworks can be used. However, for a number of reasons, they are usually a less than ideal fit, in particular if we are concerned with *pure* functional languages. There are GUI frameworks that have been designed ground up for functional settings. However, they have drawbacks of their own, such as difficulties in accommodating platform-specific look-and-feel, or lack of maturity. More generally, but for different reasons, both current imperative and current functional approaches tend to impact negatively on the modularity of the application code. This paper presents a novel approach for structuring GUIs based on *reactive values*, which can be seen as a form of communication channels, and a declarative specification of the *relationships* among such values. We argue that this approach is a good fit for a functional language, while at the same time allowing easy interfacing to the event-based, imperative GUI frameworks that are standard on most platforms, and that it further addresses some of the structural problems of present approaches. We demonstrate by presenting an implementation in Haskell that has been used to realise a range of non-trivial applications with GUI interfaces.

Keywords: GUI, pure functional programming, reactive values

1 Introduction

Functional programming is today in many ways a mature and successful programming paradigm, as witnessed by the development of languages such as F#, Erlang, OCaml, and Haskell, along with associated tools and libraries, and the growth of events such as CUFP (Commercial Users of Functional Programming). However, when it comes to Graphical User Interface (GUI) programming, the state of the art is arguably not very satisfying.

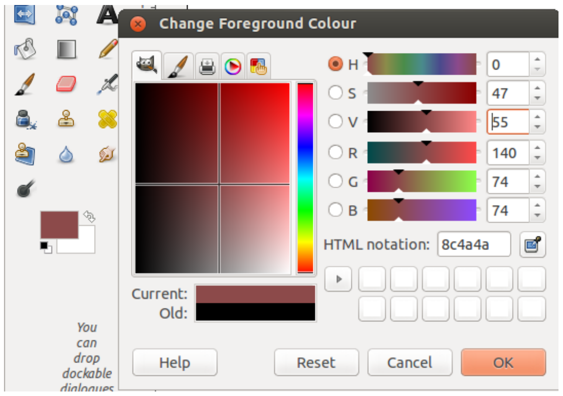
Of course, using standard imperative or object-oriented GUI frameworks, such as GTK+ or wxWidgets, is, at least in principle, straightforward [12], and indeed what typically is done. The look and feel of the resulting applications can thus be as professional as for applications written in any other language. A fundamental problem, though, is that the imperative nature of these frameworks is deeply ingrained in their Application Programming Interfaces (API). In a mostly functional language, such an API can be used without further ado as side-effects are permitted everywhere. However, the API will mandate that an overall imperative style of coding is adopted for the GUI part of an application, meaning that many of the benefits of working in a functional language can no longer be fully realised [2, 21]. Exacerbating matters, to meet the needs of the GUI part, the structure of other parts of an application may be impacted as well, particularly in the setting of a pure functional language like Haskell where all effects have to be accounted for explicitly, even if abstractions such as monads can hide much of the "plumbing".

There are many different architectures for structuring applications with GUIs. Model-View-Controller (MVC) [15] is a classic, widely-used UI programming pattern that remains one of the predominant choices also in large functional software. However, MVC, as traditionally realised, is notoriously non-scalable and suffers from poor separation of concerns as controllers tend to know and do "too much". Analysis of this and other problems, together with proposed solutions, are available in the literature of design patterns [9], data-binding languages [6] and application architectures [23].

In view of concerns such as those above, a number of attempts have been made to design GUI frameworks that are inherently functional. In the context of Haskell, an early and comprehensive example is Fudgets [3], and there have been a number of attempts since based around Functional Reactive Programming (FRP) [8, 4] or FRP-like ideas. However, these efforts come with their own sets of problems. Some, such as lack of maturity, are in essence "just" a question of engineering and resources. But others are more fundamental. For example, depending on the nature of the specific abstractions, interfacing with standard GUI toolkits, which in many cases is required for long-term sustainability and to get acceptable look-and-feel, can be very difficult indeed. For another, these approaches have their own implications regarding the structure of application code, some of which do not sit well with notions such as modularity, reuse, and separation of concerns.

To illustrate this last point, let us present a case inspired by the real-world drawing software GIMP. In GIMP users can select the foreground drawing color using an advanced dialog; see Fig. 1. This dialog enables viewing and selecting the color in multiple forms, including a luminosity panel, HSV, RGB and hexadecimal (HTML) notations. Interaction using any of these widgets is immediately reflected on all of them, including the application's main window.

Due to the interactive nature of all of these widgets, which essentially manipulate the same conceptual value, there is a circular dependency between each and every one of them as well as the application's internal representation. If we tried to represent this problem in FRP, because *signal definitions and data dependencies are merged* in this paradigm, the result would be a monolithic block



H 0
S 47
V 55
R 140
G 74
B 74
HTML notation: 8c4a4a

Fig. 1. GIMP's color selection dialog showing *seventeen interdependent ways* of representing and selecting the foreground color.

of interdependent signals that would be difficult to divide. Even in Arrowized FRP [18], where signal routing is more manifest, which arguably makes for more modular system descriptions, this multiple inter-relation must be resolved by making the circular dependencies explicit, consequently making it more difficult to structure the code based on conceptual units and not on the specific relations between different interacting elements.

In this paper, we present a novel approach for structuring GUIs that we believe is a good fit for the functional paradigm while simultaneously allowing standard GUI toolkits to be used as the "backend" for a look and feel appropriate to any specific platform. Additionally, it addresses some of the issues related to the impact on the structure of applications that affect most current GUI programming approaches, be they imperative or functional. The idea is based on the notion of *reactive values*, acting as channels of communication, and specifying *relationships* among such values in a declarative manner. We demonstrate by presenting an implementation in Haskell that has been used to realise a range of non-trivial applications with GUI interfaces.

The rest of this paper is structured as follows. We first present, in more detail, the background for the problem we are trying to address. We then state key objectives that we believe make for a better approach to GUI programming in a functional setting. Then we introduce the idea of reactive values, illustrating it with concrete examples and explaining how this approach meets the stated objectives. Finally, we review related work and draw conclusions.

2 Background

FRP remains the most prominent contestant when it comes to developing interactive applications in functional languages. We introduce only the basic concepts necessary to understand the problem that we are trying to address. For further details, see earlier papers on FRP and Arrowized FRP (AFRP) as embodied by Yampa [8, 18, 5]. This presentation draws heavily from the summary in [5]. We center our discussion on AFRP, and later summarize how the problems exposed are also present in other variants.

2.1 Fundamental Concepts

FRP is based on the concepts of *signals* and *signal functions*. A signal is a function from time to values of some type:

$$Signal\ \alpha \cong Time \rightarrow \alpha$$

$Time$ is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an audio signal, i.e., a representation of sound, would be *Signal Sample* if we take *Sample* to be the type of the varying quantity.

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

$$SF\ \alpha\ \beta \cong Signal\ \alpha \rightarrow Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first class entities* both in (classic) and in AFRP, while signals are first class only in the former, and exist indirectly through the notion of signal function in the latter.

2.2 Composing Signal Functions

Programming in AFRP consists of defining signal functions compositionally using primitive signal functions and a set of combinators. In the Haskell implementation Yampa, signal functions are an instance of the arrow framework proposed by Hughes [13]. Some central arrow combinators are *arr* that lifts an ordinary function to a stateless signal function, composition >>>, parallel composition &&&, and the fixed point combinator *loop*. In Yampa, they have the following types:

```
arr    ::  (a -> b) -> SF a b
(>>>)  ::  SF a b -> SF b c -> SF a c
(&&&)  ::  SF a b -> SF a c -> SF a (b,c)
loop   ::  SF (a,c) (b,c) -> SF a b
```

2.3 Describing cyclic dependencies in AFRP

One of the main motivations to propose an alternative to FRP for GUI programming is that, in presence of complex cyclic hybrid networks, AFRP code becomes hard to modularize. Consider the description depicted in Fig. 2.
which we could implement as a hybrid system in Yampa as follows:

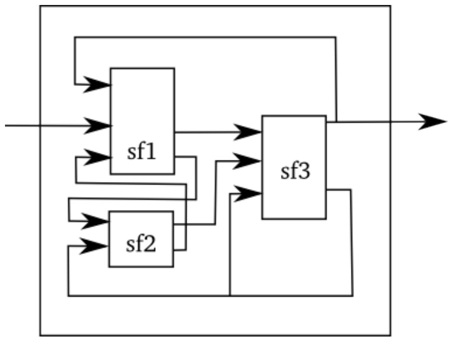


Fig. 2. A sample cyclic hybrid system.

```
sf = proc(sig) -> do
  rec (s11, s12) <- sf1 -< (sig, s22, s31)
      (s21, s22) <- sf2 -< (s12, s32)
      (s31, s32) <- sf3 -< (s11, s21, s32)
      returnA -< s31
```

Even though the notation is short and declarative, it is not easy to divide such a network of relations in separate blocks that can be specified independently of one another. In a much larger program, in which many different components may be interrelated, the difficulty in dividing these complex networks into modular, conceptual units will be even more pronounced. In particular, the previous an example would be very hard to realize without Patterson's arrow notation or without restructuring the network to make the definitions simpler. However, restructuring this network appropriately would necessitate grouping signal functions that might not be conceptually related.

As applications grows in complexity, these problems become more frequent, especially given that circular dependencies are extremely common in interactive applications, suggesting that a different approach might be more suitable for the bridge between the conceptual and presentation layers.

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

3 Key objectives

In our opinion, to constitute a successful GUI framework for a pure functional setting, the following objectives need to be met.

Architectural objectives:

- Keeping the code clean as the code base grows in size, allowing *declarative, functional programs.*
- Ensuring that controllers have minimal knowledge of the internals of both the view and the model, and that different features affecting the same elements can be separated into logical units, thus achieving *separation of concerns.* In particular, controllers should not need to know how change propagates internally within the model.
- Achieving *reusability* across applications, not only for UI and models, but also for parts of the controller.
- Allowing programs to easily be broken into cooperating *subcomponents*, possibly having more than one controller, view, model and *concurrent* access from separate threads.

UI objectives:

- *GUI-agnostic* models and controllers. Applications can potentially work with different GUI toolkits, and be compiled for different platforms with minimal changes.
- Near-complete coverage of the UI toolkit, and a way to *use unsupported features* without having to extend the framework or sacrifice purity.
- The model and the view should have no knowledge of the concurrency requirements of any GUI toolkit (such as communicating with the UI only from a special UI thread).

4 Our approach

In order to decouple different components, we provide a *uniform interface* to the model, the view, and any other external component of our program. In our framework we describe them as collections of read-write/read-only/write-only typed values whose changes we can listen to, which we call *reactive values.*

In order to manipulate these values, we give a way to *lift pure n-ary functions*, and specific rules stating the kind of component that results from such lifting.

Controllers [15] can be described as *sets of synchronization rules*, which can be *directional or bi-directional.* This allows us to break controllers into subsets of rules specific to a feature, and to move more of the problem's logic into the conceptual problem definition, which is kept in the model as an ADT.

Because we cannot hope to cover every possible use case and widget property, we provide a *low-level interface* to reactive values. Equations involving UI elements are transformed into GUI-aware code using this interface, and we allow users to write low-level code for those widgets or operations that are not specifically supported by our framework and interact with the rest of their application.

4.1 Reactive values

We tackle the problem at hand by first providing a uniform interface to both models and views, turning each field, attribute or property into a value that can be changed and observed, and whose changes we are aware of.

For that purpose we define reactive values, which are characterized by their *type* and an *access property*:

- The type is the type of the element that they store or represent. In our implementation, this can be any Haskell data type.
- The access property states whether it the reactive value is read-only (can be observed, but not modified), write-only (can be modified, but not observed) or read-write.

We say that a reactive variable is *readable* if it is read-only or read-write, and *writeable* if it is write-only or read-write. We use analogous Haskell type classes for these access properties.

Interaction with UI libraries, which often happens in terms of procedure calls, can now translated into write-only typed reactive values, attributes have a one-to-one mapping to our reactive values, and event handling functions that are not associated with any attribute can be encoded as read-only values.

Example A hypothetical text entry, which we will use to show SSH usernames, could have two read-write reactive values, of types String and Boolean, which correspond to the entry's text and whether it's enabled or disabled. In a hypothetical SFTP client application, we could let users optionally provide a username and store it in a `Maybe String` in the model.

```
usernameEntryText    :: ReadWrite String         -- UI
usernameEntryEnabled :: ReadWrite Bool           -- UI
username             :: ReadWrite (Maybe String) -- Model
```

4.2 Transforming and combining reactive values

There is often not a one-to-one correspondence between models and views. This, it is necessary to transform or combine values before they are "transfered" to other reactive values.

Applying a transformation to a readable reactive value is straightforward, and it always yields a *read-only* reactive value. Transformations to writeable values are applied before storing the new, transformed value, and they always yield a *write-only* reactive value.

In fact, read-only values are *functors* and they would fulfill the functor laws [16], and we have done the same for write-only values and *contravariant functors*. We provide `liftR` and `liftW` respectively to lift functions onto reactive variables, which have the types:

```
liftR :: Readable r => (a -> b) -> r a -> ReadOnly b
liftW :: Writeable r => (b -> a) -> r a -> WriteOnly b
```

Continuing with our previous examples, we can write:

```
usernameCustomized :: ReadOnly Bool
usernameCustomized = liftR isJust username
plainUsername :: ReadOnly String
plainUsername = liftR (fromMaybe "") username
justUsername :: WriteOnly String
justUsername = liftW Just username
```

To map a read-write reactive value, we need to provide the transformation functions in both directions. We often want the transformation to be an *isomorphism* (in which case we would lift the function by the functor and the inverse function by the contrafunctor). We cannot but trust users in this respect, providing only a small facility for *involutions*. In the extended article we briefly detail the consequences of not using actual isomorphisms, and the cases in which it may be justified.

The type of the function that lifts mappings onto read-write reactive values is:

```
liftB :: (a -> b, b -> a) -> ReadWrite a -> ReadWrite b
```

which can be used as follows:

```
plainStringUsername :: ReadWrite String
plainStringUsername = liftB (fromMaybe "", Just) username
```

Similar constraints apply to n-ary functions, depending on the variables' read/write access properties. We have defined these using analogous names for the lifting combinators, indicating the artity of the function being lifted.

Example In a hypothetical SSH application, such as the one depicted in figure 3, we give users a checkbox to decide whether the username must be customized and, if so, a text field in which to write it. The possibly customized username could be defined as:

```
customUsername :: ReadOnly (Maybe String)
customUsername = liftR2 (\c v -> if c then Just v else Nothing)
                      customUsernameCheckboxValue usernameEntryText
```

4.3 Reactive equations

Now that we can define and combine reactive values, it is time to describe dependencies between them. For this purpose, we use constraint-building combinators that capture the idea that the value of a reactive value x must be updated to the value of a reactive value y. Therefore, whenever y changes, x must be updated accordingly. We use the symbols <:= and =:> for such combinators, depending on the direction of change propagation. One of the values (the origin of the change) must be **readable**, the other must be **writeable**, and they must both contain values of the same type.

To simplify our rules further, we also provide the combinator =:=, which is syntactic sugar for two directional rules. For obvious reasons, it can only be used with **read-write** variables.

Example Imagine that we have a checkbox and a text entry in our SSH GUI. If the user wants to customize the username, she must tick the checkbox and write a username. If not, the checkbox must be unticked, and the text entry must be empty and disabled. In the model, we use a `Maybe String` to encode all possibilities.

```
-- If username changes, update view
customUsernameCheckboxValue <:= liftR isJust          username
usernameEntryText           <:= liftR (fromMaybe "") username

-- Coherence between checkbox and text view
usernameEntryEnabled =:= customUsernameCheckboxValue

-- Update model upon changes
liftR2 (\c v -> if c then Just v else Nothing)
  customUsernameCheckboxValue usernameEntryText =:> username
```

Using these combinators we can specify circular data dependencies (and therefore create a potentially-infinite data-transfer loop). Implicitly, every =:= rule specifies one.

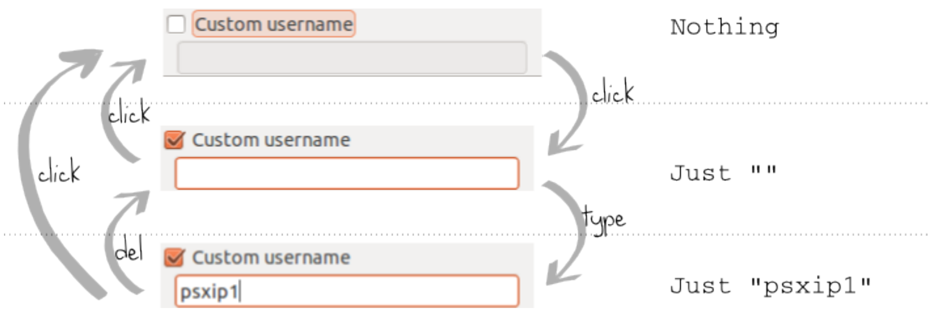


Fig. 3. Transitions in the application, showing the two widgtets' states in synchrony (left), and the contents of the model in each state (right).

5 Lower-level interface and implementation details

Reactive values are polymorphic on the type of the values they contain. Their access properties are captured using type classes, each of which defines corresponding operations.[1] A readable reactive value must let us read its value and listen to changes to it:

```
class Readable a b m where
  read     :: a -> m b
  onChange :: a -> m () -> m ()
```

Writable reactive values, on the other hand, must be able to consume values:

```
class Writable a b m where
  write :: a -> b -> m ()
```

A read-write value needs just be an instance of both classes:

```
class (Readable a b m, Writable a b m) => ReadableWritable a b m
```

We also provide instances of this class that can be used to create reactive values, although developers need not use those specific instances, and can supply one of their own. One example for which the appropriate instance has been defined is:

```
data RW m a = RW (a -> m ()) (m a) (m () -> m ())
```

The interface provided for all reactive values is the same, regardless of whether we are using widget properties or purely functional values. The implementation of reactive rules is now straightforward, and we illustrate it with the left-to-right example:

```
(=:>) :: (Monad m, Readable a b m, Writable c b m)
      => a -> c -> m ()
(=:>) v1 v2 = onChange v1 (read v1 >>= write v2)
```

5.1 Reactive widget attributes

Most UI widgets, at least in GTK+ and WX, can be controlled by means of a collection of attributes. Except in very specific cases, these attributes can be both set and viewed, and they have a corresponding event handler installer that we can use to listen to changes. One example of how we can turn a GTK+ text entry's text into a reactive value is:

[1] Our implementation signature is slightly more convoluted than the one presented in the previous section, parameterizing also over a monad. In practice, and for GUI applications, this monad will always be the IO monad. Nevertheless, we want our signature to be general enough to account for other possibilities.

```
entryTextReactive :: Entry -> RW IO String
entryTextReactive e = RW setter getter notifier
 where getter     = get e entryText
       setter v   = set e [entryText := v]
       notifier p = void (on e editableChanged p)
```

Note that the functions `get`, `set` and `on` are GTK+ functions, that `entryText` is an attribute of entries that refers to its text, and that `editableChanged` is a GTK+ Signal (read "event") that we use to install an event listener. We could now use the previous function two synchronize two (given) text entries:

```
entryTextReactive entryName =:>
  liftW reverse (entryTextReactive entryReversedName)
```

5.2 Reactive models

The same can be done for pure Haskell values, which we can embed into a reactive value using something we called "reactive models" and "protected models". A reactive model is just a polymorphic, purely functional holder of reactive values. These reactive models hold not only the actual value, but also a queue of event pending handlers to be called. A protected model wraps a reactive model in a Software Transactional Memory [11] MVar, and uses one thread to dispatch all the pending change handlers in order upon changes.

This is just one of many possible implementations, and we do not impose this specific construct upon users. Furthermore, we have sometimes changed our default implementation in order to adapt it to specific applications, for instance adding knowledge of past states in order to implement a transparent, reactive undo/redo queue. Provided that the type class interfaces are fulfilled, reactive values may take many forms.

To make our applications lighter, we also provide several facilities in our library to focus only on specific parts of these protected models, either by means of lifting pure functions and lenses [22], or using record field accessors. For this last case, we provide several Template Haskell functions that generate the necessary code to observe, and change, only one subpart of a larger model.

6 Real-world experience and impact on software architecture

We have used the approach described in this paper in the development of several real-world applications, amounting to over 20,000 lines of Haskell code. This includes small (yet useful) applications that showcase our framework, and fully-fledged programs, originally written in an imperative fashion, which we have partially and progressively transformed into reactive programs.

The first and most important benefit is that our controller [15] no longer knows about the internals of the model and the view. Our applications are now structured as a *view*, a *model* and a *set of constraints*. Occasionally we have added IO code, either in write-only variables or using the low-level interface to reactive values. This also allowed us to progressively add support for more and more widgets and properties, while being able to benefit from reactivity from the very start. These IO blocks do not pollute our codebase, and become cleaner as a result of moving most of the code to separate, declarative reactive constraints.

A second benefit is palpable when we use threads that interact with our models. This was done in Keera Posture [20] to record images from a webcam and detect when users sit incorrectly (one thread does visual recognition and communicates with the model, which is separately syncronized with the view). It was also used in Keera Renamer (a content-based PDF renamer) and in Keera Gale Studio (a Graphical Adventure Game IDE), to asynchronously move files in the hard disk from a thread that communicates with the model while showing a progress dialog.

In all of these cases, the controller is *not aware of any concurrency requirements* of the model or the view. All UI functions are transparently executed from the UI thread.

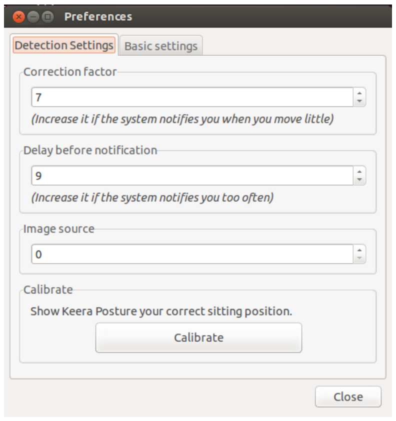

Preferences
Detection Settings | Basic settings
Correction factor
7
(Increase it if the system notifies you when you move little)
Delay before notification
9
(Increase it if the system notifies you too often)
Image source
0
Calibrate
Show Keera Posture your correct sitting position.

Fig. 4. Keera Posture's preference dialog, whose updates are immediately used by the image recognition thread without restarting or explicit message passing.

A consequence of having separate conditions involving the same reactive values is that we can group features into modules, resulting in greater *separation of concerns*, compared to structuring the application based on the UI layout or the model structure. It also allows us to completely add or remove a feature to a program without affecting the rest of our codebase, making *debugging* easier.

A final outcome of our architecture is that we can *factorize code* out into libraries that have minimal knowledge of the view and the model, yet implement useful coordinations between them. A simple, yet frequent feature we factorized was to show the program name in the title bar of our program. The independent library only needs access to a writeable String-typed value for the title bar, a readable `Maybe String` containing the current file name, and a readable boolean variable indicating if the file has been modified but not saved. This rule can now be imported into any program, cleanly and concisely.

7 Related work

From an imperative or object-oriented perspective, our work is closest in spirit to reactive programming[2], and then in particular to *change subscription* facilities and *data binding* languages.

Our notification system is similar to the observer design pattern [9] frequently encountered in object-oriented programming. Indeed, this pattern has specific support in recent versions of Javascript in the form of `Object.observe()` [19]. The observer pattern does provides the possibility of detecting changes to objects, which is one of the pillars of our proposal, but it does not constitute a reactive DSL on its own.

Our reactive rules specify a data dependency language, with similar motivations and aims as the data-binding facilities of frameworks like AngularJS and EmberJS. There are, however, structural differences. AngularJS, for instance, merges data-binding, function lifting, and view declaration into a single, annotated XML tree. For that reason, we believe our approach results in a more modular and abstract design, partly because it maximizes separation of concerns, and partly because it allows factoring sets of rules out of specific applications for independent reuse.

Nonetheless, and as one would expect, our approach poses challenges similar to those of other frameworks that provide data-binding in terms of minimizing data propagation, performance and consistency. To minimize data propagation we use equality tests (where permitted) to avoid unnecessary updates: the "setters" of our reactive values only update values and notify subscribers when the values have actually changed. This approach is typically[3] more efficient than the *dirty-checking* used in AngularJS, but it does present some challenges both in the design of reactive values and in the implementation of our library.

Guaranteeing consistency, especially in the presence of asynchronous propagation, is hard to express formally. Our current implementation sacrifices consistency across a network of possibly-duplicated values in favour of responsiveness and scalability [10]. We rely on always being able to break circular dependencies to achieve *eventual consistency* [24]. We return in the next section to how we intend to tackle this aspect more formally and provide minimal sufficient conditions that guarantee the desired convergence.

As to purely functional user interfaces, there are several existing approaches most notably Fudgets [3] and ones based on Functional Reactive Programming (FRP) [8, 4]. Compared to Fudgets, we believe that we achieve greater separation of concerns and controller modularization. Further, our framework is not limited to one reactive value per widget. Nonetheless, we consider the idea of defining a compositional UI framework worth exploring, and it is the direction we will follow with our polymorphic controller rule sets.

[2] https://github.com/Netflix/RxJava

[3] Should equality testing be undesirable, e.g. because prohibitively expensive, the user can always take control by defining a new type.

In contrast to FRP, our approach disregards time, both at the conceptual and implementation level. While, for example, analysis of soft real-time guarantees has been carried out in the FRP setting [14], the lack of time-based semantics prevents us from reasoning about temporal properties. That said, so far, we have not needed to reason about time guarantees, and change propagation in Gtk applications has always been fast enough. Nevertheless, we believe that giving precise semantics to Reactive Values in terms of FRP signals is entirely possible, and would provide the necessary formal foundations that would allow implementors to reason about Reactive Values.

However, the central difference between Reactive Values and FRP signals is the idea of separating value declaration from data dependencies: our reactive values are declared independently of their connections to other elements, while in FRP each signal is defined in one place by a single expression. As we showed before, the FRP approach often leads to scalability issues in larger applications in the presence of circular dependencies. In the context of user interfaces, we argue that our framework achieves greater modularity than FRP and is easier to use and understand. The ability to easily use any IO code to make up for unsupported features is only rivalled, to the extent of our knowledge, by reactive-banana [1].

In this area of DSLs for UI programming in Functional Languages, Gtk+ lenses [7] also has similarities to the low-level interface of our reactive values. Although lenses and change propagation are, in principle, orthogonal, a lens-based interface would help describe the semantics of reactive values, facilitate reusing existing libraries, and simplify definitions. Reactive values have their own laws, weaker than the lens laws, and there are practical cases in which they are purposely not fulfilled. We have also combined reactive values with lenses in order to allow focusing on specific parts of a reactive value.

There are further clear similarities between our approach and *Uniform Data Sources* [17]. Our reactive values constitute a constrained form of Data Sources in which the input and the output type are the same, augmented with an event subscription operation.

Finally, both Reactive Values and Uniform Data Sources have elements in common with Lenses [22]. Although there is an obvious terminology overlap, reactive values are neither lenses nor fulfil the lens laws. We expect future work on formalizing RVs with FRP to help us express a temporal version of similar laws that may be applicable to our case. Nevertheless, RVs can be combined with lenses to focus on specific parts of reactive values, hence our vision that lenses and reactive values are complementary and address different problems. Recent developments on monadic lenses [7] and lenses with notifications could help us simplify our formalisation in the future, allowing us to reduce our language to its true core objective which is to serve as a data-binding language between typed reactive elements, a bridge for the gap in GUI application architecture.

8 Future work

In this paper we have described a reactive framework suitable to implement applications with GUIs in Haskell. We have demonstrated how reactive values can be defined and how to establish dependencies between them.

Our work has been guided by industrial experience, and we have successfully used this framework to implement several applications. Due to the nature of these programs, formal analysis of temporal properties of change propagation and consistency guarantees have not been explored. Our approach ignores time at a conceptual level, which makes it less expressive than Qt's QML and FRP. In the future, we would like to provide a well-founded FRP-based definition to reason about delays, change propagation and temporal inconsistencies.

We have observed constant memory consumption while profiling some applications. Nevertheless, for the reasons stated above, it is currently difficult in our framework to reason about efficiency, memory footprints and garbage collection.

In particular, it has been suggested that making our reactive rules first-class citizens would facilitate the specification of dynamic data-bindings, which would increase the importance of garbage collection in our framework.

Recent developments combining applicative syntax and lenses have allowed us to obtain less verbose rule sets. We expect this combination to have a sizeable impact on the codebase of large applications. Future work will measure and compare the complexity of software specified in each style.

In this paper we have not described all the tools and libraries in our framework's ecosystem, which contains several prepackaged controller rule sets. Examples include adding a visual loggin console, quitting programs (optionally saving modified files), undoing/redoing, and checking for updates. We have identified other common features that could be also be factorized. Eventually, this framework could evolve towards an algebra of applications, structured around the concepts of model, view, controller, threads and conditions, and a set of well-defined combinators, in which orchestrations would have a more precise meaning.

We have also developed tools and libraries that have not been described in this paper, and which help with internationalization, accessing UI widgets, generating application and reactive model skeletons, etc. Some of these tools and libraries are GTK+ centric, and we heavily rely on *convention over configuration.* Nonetheless, configuration is possible and the division in multiple libraries makes supporting other UI toolkits straightforward.

References

1. Heinrich Apfelmus. Reactive-banana. http://www.haskell.org/haskellwiki/Reactive-banana, 2011.
2. J.-P. Banâtre, S. B. Jones, and D. Le Métayer. *Prospects for Functional Programming in Software Engineering.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.
3. Magnus Carlsson and Thomas Hallgren. FUDGETS: A graphical user interface in a lazy functional language. (Section 6):321–330, 1993.

4. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.
5. Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.
6. Peter Bacon Darwin and Pawel Kozlowski. *AngularJS web application development*. Packt Publishing, 2013.
7. Péter Diviánszky. LGtk: Lens-based Gtk Interface. http://people.inf.elte.hu/divip/LGtk/LGtk.html, 2013.
8. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
10. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
11. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.
12. Kenneth Hoste. An Introduction to Gtk2Hs, a Haskell GUI Library. In Shae Erisson, editor, *The Monad.Reader Issue 1*. 2005.
13. John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.
14. Roumen Kaiabachev, Walid Taha, and Angela Zhu. E-FRP with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 221–230, New York, NY, USA, 2007. ACM.
15. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.
16. Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.
17. Steffen Michels and Rinus Plasmeijer. Uniform data sources in a functional language. In *Submitted for presentation at Symposium on Trends in Functional Programming, TFP*, volume 12, 2012.
18. Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.
19. Addy Osmani. Data-binding revolutions with Object.observe().
20. Ivan Perez. Keera Posture. https://github.com/keera-studios/keera-posture, 2012.
21. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
22. Benjamin C. Pierce. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem, October 2004. Invited talk at *New England Programming Languages Symposium*.
23. Mike Potel. MVP: Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc*, 1996.
24. Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

Bibliography

[200a] Can gui programming be liberated from the io monad.

[200b] Introduction to discrete-event simulation and the simpy language.

[Aca05] Umut Acar. *Self-adjusting computation.* PhD thesis, Carnegie Mellon University, 2005.

[AMP] Martın Abadi, Frank McSherry, and Gordon D Plotkin. Foundations of differential dataflow.

[Ams12] Edward Amsden. Push-pull signal-function functional reactive programming. In *Draft Proceedings of the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, page 128, 2012.

[AnSa] Supporting Different Screens. `http://developer.android.com/training/basics/supporting-devices/screens.html`.

[AnSb] Android SDK. `https://developer.android.com/sdk`. Accessed: 2012-10-20.

[ANVdB13] Tom J Ameloot, Frank Neven, and Jan Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):15, 2013.

[AP98] Peter Achten and Rinus Plasmeijer. Interactive functional objects in clean. In *Implementation of Functional Languages*, pages 304–321. Springer, 1998.

[AVEP05] Peter Achten, Marko Van Eekelen, and Rinus Plasmeijer. Generic graphical user interfaces. In *Implementation of Functional Languages*, pages 152–167. Springer, 2005.

[Bac78] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[BCVC$^+$12] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. In *ACM Computing Surveys*. Citeseer, 2012.

[BDS96] Frédéric Boussinot, Guillaume Doumenc, and Jean-Bernard Stefani. Reactive objects. In *Annales des télécommunications*, volume 51, pages 459–473. Springer, 1996.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[Bur92] Steve Burbeck. Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2*, 5, 1992.

[BvEVLP87] TH Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. CleanâĂŤa language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, pages 364–384. Springer, 1987.

[Car02] Magnus Carlsson. Monads for incremental computing. *SIGPLAN Not.*, 37(9):26–35, September 2002.

[CC13a] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[CC13b] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330. ACM, 1993.

[CH00] Koen Claessen and John 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, pages 268–279, New York, NY, USA, 2000. ACM.

[Che05] Mun Hon Cheong. Functional programming and 3d games. *BEng thesis, University of New South Wales, Sydney, Australia*, 2005.

[CK04] Gregory H Cooper and Shriram Krishnamurthi. Frtime: Functional reactive programming in plt scheme. *Computer science technical report. Brown University. CS-03-20*, 2004.

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[Cou01] Antony Courtney. Frappé: Functional reactive programming in Java. In *Third International Symposium on Pratical Aspects of Declarative Languages (PADL)*, March 2001.

[Cou03] Antony Courtney. Functionally modeled user interfaces. In *Interactive Systems. Design, Specification, and Verification*, pages 107–123. Springer, 2003.

[Cou04] Antony Alexander Courtney. *Modeling User Interfaces in a Functional Language.* PhD thesis, New Haven, CT, USA, 2004. AAI3125177.

[CR96] Denis Caromel and Yves Roudier. Reactive programming in eiffel. In *Object-Based Parallel and Distributed Computation*, pages 125–147. Springer, 1996.

[CVM97] Koen Claessen, Ton Vullinghs, and Erik Meijer. Structuring graphical paradigms in tkgofer. In *ACM SIGPLAN Notices*, volume 32, pages 251–262. ACM, 1997.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[Doc06] Tim Docker. Chart - a library for generating 2d charts and plots. `https://github.com/timbod7/haskell-chart`, 2006. Accessed: 2014-10-24.

[Edw09] Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 925–932. ACM, 2009.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[Ell07] Conal M Elliott. Tangible functional programming. In *ACM SIGPLAN Notices*, volume 42, pages 59–70. ACM, 2007.

[Ell09] Conal M Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[FGM$^+$07] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17, 2007.

[FJ95] Sigbjørn Finne and Simon Peyton Jones. *Composing haggis*. Springer, 1995.

[FLM$^+$09] Dirk Fahland, Daniel LÃijbke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 353–366. Springer Berlin Heidelberg, 2009.

[FLSR10] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 221–230. IEEE, 2010.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997.

[FX07] Manuel Fahndrich and Songtao Xia. Establishing object invariants with delayed types. In *ACM SIGPLAN Notices*, volume 42, pages 337–350. ACM, 2007.

[GH11] Jeremy Gibbons and Ralf Hinze. Just do it: Simple monadic equational reasoning. In *ICFP*, September 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GL] OpenGL Haskell bindings. `http://hackage.haskell.org/package/OpenGL`. Accessed: 2014-10-20.

[Gla] Glade - A User Interface Designer. `http://glade.gnome.org`. Accessed: 2014-10-20.

[GML] GTK+ 3 Reference Manual: Main loop and Events. `https://developer.gnome.org/gtk3/stable/gtk3-General.html`. Accessed: 2014-11-22.

[God07] S. Goderis. *On the separation of user interface concerns: A Programmer's Perspective on the Modularisation of User Interface Code*. Asp / Vubpress / Upa, 2007.

[Gre09] Jason Gregory. *Game engine architecture*. CRC Press, 2009.

[GTK] GTK+. `http://www.gtk.org`. Accessed: 2014-10-20.

[HA1] Reactive-banana. `https://www.haskell.org/haskellwiki/Reactive-banana`. Accessed: 2014-10-20.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[Hea08] J. R. Heard. Beautiful code, compelling evidence. Technical report, 2008.

[Hei05] George T Heineman. An instance-oriented approach to constructing product lines from layers. *Technical Report, WPI CS Tech Report 05-06*, 2005.

[HF05] Stuart Hansen and Timothy V Fossum. Refactoring model-view-controller. *Journal of Computing Sciences in Colleges*, 21(1):120–129, 2005.

[HF08] Graham Hutton and Diana 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.

[HGL] A simple graphics library based on X11 or Win32. `https://hackage.haskell.org/package/HGL`. Accessed: 2014-10-20.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[Hud00] Paul Hudak. The Haskell school of expression: learning functional programming through multimedia. January 2000.

[Hug00] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[Jel08] Wolfgang Jeltsch. Improving push-based frp. *Draft Proceedings of Trends in Functional Programming (TFP'08)*, 2008.

[Jel09] Wolfgang Jeltsch. Signals, not generators! *Trends in Functional Programming*, 10:145–160, 2009.

[Jel12] Wolfgang Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, 2012.

[Jel13] Wolfgang 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*, pages 69–78. ACM, 2013.

[JN11] Wolfgang Jeltsch and Teooriapäevad Nelijärvel. The curry-howard correspondence between temporal logic and functional reactive programming. *Slides of a talk given at the Estonian Computer Science Theory Days at Nelijärve*, 2011.

[Jon89] SL Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[KB11] Neelakantan R Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 257–266. IEEE, 2011.

[KBH12] Neelakantan R Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. *ACM SIGPLAN Notices*, 47(1):45–58, 2012.

[KGI] Keera Studios' GALE IDE. `http://keera.co.uk/blog/products/gale-studio/`. Accessed: 2014-10-20.

[KP+88] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. 1988.

[KPVES01] PIETER KoopmaN, RINUS PLASMEIJER, MARKO VAN EEKELEN, and SJAAK SMETSERS. Functional programming in clean, 2001.

[Kri13] Neelakantan R Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 221–232. ACM, 2013.

[KTZ07] Roumen Kaiabachev, Walid Taha, and Angela Zhu. E-frp with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 221–230, New York, NY, USA, 2007. ACM.

[KWLM09] Terence Kelly, Yin Wang, Stéphane Lafortune, and Scott Mahlke. Eliminating concurrency bugs with control engineering. *IEEE Computer*, 42(11):52–60, 2009.

[L3D] LambdaCube 3D. `http://lambdacube3d.wordpress.com/`. Accessed: 2014-10-20.

[LCH09] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *ACM Sigplan Notices*, volume 44, pages 35–46. ACM, 2009.

[LGPA13] Hai Liu, Neal Glew, Leaf Petersen, and Todd A Anderson. The intel labs haskell research compiler. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 105–116. ACM, 2013.

[LH07] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[Lip10] Ben Lippmeier. Gloss - painless 2d vector graphics, animations and simulations. `http://gloss.ouroborus.net/`, 2010. Accessed: 2014-10-24.

[Lip12] Ben Lippmeier. Gloss machina. `https://github.com/benl23x5/gloss/blob/master/gloss-examples/picture/Machina/Main.hs`, 2012. Accessed: 2014-10-24.

[LJ95] John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[LL91] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In *Handbook of theoretical computer science (vol. B)*, pages 1157–1199. MIT Press, 1991.

[LW10] Ruy Ley-Wild. *Programmable self-adjusting computation*. PhD thesis, Carnegie Mellon University, 2010.

[MGB$^+$09] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[MH06] Sean McDirmid and Wilson C Hsieh. Superglue: Component programming with object-oriented signals. In *ECOOP 2006–Object-Oriented Programming*, pages 206–229. Springer, 2006.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2008.

[MPA11] Steffen Michels, Rinus Plasmeijer, and Peter Achten. itask as a new paradigm for building gui applications. In *Implementation and Application of Functional Languages*, pages 153–168. Springer, 2011.

[Mye91] Brad 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, pages 211–220, New York, NY, USA, 1991. ACM.

[Mye93] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical report, Pittsburgh, PA, USA, 1993.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.

[Net] Netwire. `http://hub.darcs.net/ertes/netwire`. Accessed: 2014-10-20.

[Nil05] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP*, volume 5, pages 54–65, 2005.

[NP14] Henrik Nilsson and Ivan Perez. Declarative Game Programming. In O. Danvy, editor, *International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, Canterbury, UK, September 2014. ACM Press.

[NR95] Rob Noble and Colin Runciman. Gadgets: Lazy functional components for graphical user interfaces. In *Programming Languages: Implementations, Logics and Programs*, pages 321–340. Springer, 1995.

[NS79] William M Newman and Robert F Sproull. *Principles of interactive computer graphics.* McGraw-Hill, Inc., 1979.

[OFLK12] Yoshiki Ohshima, Bert Freudenberg, Aran Lunzer, and Ted Kaehler. A report on kscript and ksworld. *VPRI Research Note-2012-008*, 2012.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell.* O'Reilly Media, Inc., 1st edition, 2008.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, September 2005.

[ORD] Ordrea: Push-pull implementation of discrete-time FRP. `https://hackage.haskell.org/package/ordrea`. Accessed: 2014-11-22.

[P14a] Hcwiid: Haskell bindings to Cwiid. `https://github.com/ivanperez-keera/hcwiid`. Accessed: 2014-10-20.

[P14b] Haskanoid on Github. `http://github.com/ivanperez-keera/haskanoid`. Accessed: 2014-10-20.

[Pat09] Gergely Patai. Eventless reactivity from scratch. *Draft Proceedings of Implementation and Application of Functional Languages (IFLâĂŹ09)*, pages 126–140, 2009.

[Pat11] Gergely Patai. Efficient and compositional higher-order streams. In *Functional and Constraint Logic Programming*, pages 137–154. Springer, 2011.

[Pet10] 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.".

[PHP87] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[PJW93] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[Prz] Elm at Prezi. `http://engineering.prezi.com/blog/2013/05/21/elm-at-prezi/`. Accessed: 2014-11-22.

[PTS00] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. In *Practical Aspects of Declarative Languages, Volume 1753 of LNCS*, pages 16–31. Springer, 2000.

[QT] QT Project. `http://qtproject.org`. Accessed: 2014-10-20.

[RA09] Vignesh T Ravi and Gagan 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*, pages 308–315. IEEE Computer Society, 2009.

[Rey93] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[RFT] Connect Reactive and FieldTrip. `http://hackage.haskell.org/package/reactive-fieldtrip`. Accessed: 2014-10-20.

[RM92] John R. Rose and Hans Muller. Integrating the scheme and c languages. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 247–259, New York, NY, USA, 1992. ACM.

[Rob04] Bart Robinson. wxfruit: A practical gui toolkit for functional reactive programming, 2004.

[RoRa] Getting Up and Running Quickly with Scaffolding. `http://guides.rubyonrails.org/v3.2.13/getting_started.html#getting-up-and-running-quickly-with-scaffolding`.

[RoRb] Deprecating Dynamic Scaffolding. `https://groups.google.com/d/topic/rubyonrails-core/fSkbnrdw5PM/discussion`.

[Rov11] Kenneth Christian Rovers. *Functional model-based design of embedded systems with UniTi*. University of Twente, 2011.

[RPHH99] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in dsl design. In *Proceedings of the 21st international conference on Software engineering*, pages 484–493. ACM, 1999.

[Rus03] George Russel. Fruit & co. Message posted on the Haskell GUI mailing list, available at `https://www.haskell.org/pipermail/gui/2003-February/000140.html`, 2003.

[S12] SoOSiM UI. `https://github.com/ivanperez-keera/SoOSiM-ui`. Accessed: 2014-10-20.

[SA07] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 25–36, New York, NY, USA, 2007. ACM.

[Sag00] Meurig Sage. Frantk-a declarative gui language for haskell. *ACM SIGPLAN Notices*, 35(9):106–117, 2000.

[Scu11] Neil Sculthorpe. *Towards safe and efficient functional reactive programming*. PhD thesis, University of Nottingham, 2011.

[SDLa] SDL: binding to libSDL. `https://hackage.haskell.org/package/SDL`. Accessed: 2014-10-20.

[SDLb] SDL2: bindings to libSDL >= 2.0.0. `https://github.com/Lemmih/hsSDL2`. Accessed: 2014-10-20.

[Sex87] Harlan Sexton. Foreign functions and common lisp. *ACM SIGPLAN Lisp Pointers*, 1(5):11–23, 1987.

[SN10a] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change. *Higher-Order and Symbolic Computation*, 23(2):227–271, 2010.

[SN10b] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change. *Higher-Order and Symbolic Computation*, 23(2):227–271, 2010.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[Swe99] Sweeney, Tim. [gclist] Games and Realtime Garbage Collection. `http://lists.tunes.org/archives/gclist/1999-July/001632.html`, 1999. Quote: "I need to manage thousands objects in realtime and 60 frames per second".

[Sym06] Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computer Science*, 148(2):3 – 25, 2006. Proceedings of the ACM-SIGPLAN Workshop on {ML} (ML 2005) ACM-SIGPLAN Workshop on {ML} 2005.

[vdP13] Atze van der Ploeg. Monadic functional reactive programming. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM, 2013.

[Vic11] Bret Victor. Inventing on principle. `http://vimeo.com/36579366`, 2011. Accessed: 2014-10-23.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VSS96] Ton Vullinghs, Wolfram Schulte, and Thilo Schwinn. *An introduction to TkGofer*. 1996.

[W32] A binding to part of the Win32 library. `http://hackage.haskell.org/package/Win32`. Accessed: 2014-10-20.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys (CSUR)*, 29(3):240–263, 1997.

[WJ] Stephanie Wilson and Peter Johnson. Bridging the generation gap: From work tasks to user interface designs.

[WTH01] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time frp. In *ACM SIGPLAN Notices*, volume 36, pages 146–156. ACM, 2001.

[WTH02] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven frp. In *Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

[WX] wxWidgets. `http://www.wxwidgets.org`. Accessed: 2014-10-20.

[Xou] Xournal. `http://xournal.sourceforge.net`. Accessed: 2014-10-20.

[YGL] Yampa-glut: an adapter that connects OpenGL/GLUT to the FRP library "Yampa". `https://hackage.haskell.org/package/yampa-glut`. Accessed: 2014-10-20.

[Yor08] Brent Yorgey. Diagrams - a domain-specific language for creating vector graphic. `http://projects.haskell.org/diagrams/`, 2008. Accessed: 2014-10-24.