Bridging the GUI Gap with Reactive Values and Relations

Ivan Perez Henrik Nilsson

School of Computer Science
University of Nottingham
United Kingdom
{ixp,nhn}@cs.nottingham.ac.uk

Abstract

There are at present two ways to write GUIs for functional code. One is to use standard GUI toolkits, with all the benefits they bring in terms of feature completeness, choice of platform, conformance to platform-specific look-and-feel, long-term viability, etc. However, such GUI APIs mandate an imperative programming style for the GUI and related parts of the application. Alternatively, we can use a functional GUI toolkit. The GUI can then be written in a functional style, but at the cost of foregoing many advantages of standard toolkits that often will be of critical importance. This paper introduces a light-weight framework structured around the notions of *reactive values* and *reactive relations*. It allows standard toolkits to be used from functional code written in a functional style. We thus bridge the gap between the two worlds, bringing the advantages of both to the developer. Our framework is available on Hackage and has been been validated through the development of non-trivial applications in a commercial context, and with different standard GUI toolkits.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Functional Languages; H.5.2 [*Information Interfaces and Presentation*]: User Interfaces – Graphical user interfaces (GUIs)

Keywords GUI, pure functional programming, reactive values

1. Introduction

Modern interactive applications are often large and complex with many interrelated elements and sophisticated Graphical User Interfaces (GUIs). The complexity stems from many sources, including the nature of specific application domains, existing software infrastructure, and orthogonal usability features like undo-redo and support for cancellation of long-running operations (requiring concurrency) [33]. Consequently, structuring such programs well is difficult, as is reasoning about them [17, 20, 29]. Yet the prevalence of this class of programs, which we refer to as *GUI applications*, makes it important to look for ways to ameliorate these difficulties.

Let us consider some of the obstacles in more detail. GUI applications are inherently stateful and effectful, raising difficulties in its own right [39]. Current standard GUI toolkits associate computations to *widgets* (interactive visual elements) through call-back mechanisms which results in an event-oriented programming style that inverts control [12, pp. 36–37] and is hard to reason about [32].

Functional programming in itself offers potential advantages for programming GUI applications. These include abstraction facilities, which help managing some of the complexities, referential transparency, which facilitates reasoning and program transformations, and transparent parallelisation, which obviates concerns about deadlocks or rolling back unfinished transactions [18].

There are dozens of implementations of and proposals for functional GUI frameworks. Some are little more than low-level bindings to existing GUI toolkits. The result is visually appealing applications with good GUI performance. However, the price is an imperative programming style that suffers from the problems discussed above. Others seek to integrate with and capitalise from the functional setting by employing functional structuring principles. This can facilitate reasoning and, in a statically typed setting, provide good static correctness guarantees. However, there are generally caveats such as practically prohibitive maintenance costs, failure to conform to platform-specific GUI standards, and issues with modularity, scalability, and efficiency. We elaborate in section 2.

The central idea of this paper is to provide light-weight abstractions that on the one hand enable easy, uniform access to arbitrary existing GUI toolkits and other external resources, and on the other seamlessly blend with the functional programming paradigm, scale well, and support modularity. This is what we refer to as "bridging the GUI gap". The specific contributions of this paper are:

- *Reactive Values* (RVs): an abstraction that can represent widget properties, model fields, files on disk, network sockets and other changing entities, and further be combined and adapted using a series of combinators as well as general lenses [13].
- *Reactive Relations* (RRs): declarative, uni- or bi-directional rules used to relate existing RVs, and which can be grouped and factored out into reusable libraries of *choreographies*
- A framework implementing these ideas, available off Hackage, and validated on a series of non-trivial examples, including some commercial examples developed by Keera Studios[1], and with different GUI backends.

The rest of this paper is structured as follows. We first present, in more detail, the problems we seek to address. We then introduce reactive values and relations, illustrating with concrete examples and explaining how our proposal addresses the identified problems. We then explain how we have used our approach in real-world applications and the impact this had on the architecture of these applications. Finally, we review related work and draw conclusions.

[1] Startup founded by the first author. See `http://keera.co.uk`.

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3808-0/15/09...$15.00
http://dx.doi.org/10.1145/2804302.2804316

2. Background

Current solutions for functional GUI programming address two concerns: 1) description of the GUI itself, and 2) definition of patterns for interactive and stateful programs that conform to functional principles. This section gives a brief overview of the field, mainly from a Haskell perspective, and identifies three problems of current solutions:

- GUIs can be written using low-level imperative APIs, but the code exhibits the usual problems associated with imperative programming (section 2.1).
- Purely Functional GUIs are a much better fit for functional programming, but have high maintenance costs, especially if one seeks a "natural" appearance and behaviour across platforms (section 2.2).
- Existing GUI toolkits and programming paradigms impose architectural constraints that limit the scalability of applications (section 2.3).

In section 3 we introduce Reactive Values and Relations to address these problems.

2.1 Imperative GUIs in Functional Languages

I/O in functional languages is often performed using libraries written in C/C++ and imported as effectful computations via a Foreign Function Interface. Many such libraries work at a very low-level, such as OpenGL[2]. Functional bindings usually do minimal work, resulting in APIs that resemble the underlying C interface.

In functional languages, effectful computations such as monads and applicative functors are first-class entities [28, 40]. Together with their associated laws, this enables some forms of equational reasoning [15, 22]. However, without higher-level abstractions, large programs that do input/output still tend to look imperative [39, p. 11–12]. The strict execution order of the effectful computations imposes sequential thinking and a need to mentally track the implicit program state. Reasoning thus remains hard [4].

To abstract over visual elements and provide standardised appearance and behaviour, developers use libraries that implement interactive elements or *widgets*. Examples include Gtk+[3], wxWidgets[4] and Qt[5]. Widgets have associated properties and events. Developers can modify the properties, and install event handlers to execute computations when, for instance, a button is depressed. As an example, the following code[6], builds an application with a button that prints a text when the button is clicked:

```
1  import Graphics.UI.Gtk

   main :: IO ()
   main = do
5    initGUI
     window <- windowNew
     set window [windowTitle := "Pix",
                 containerBorderWidth := 10]
     button <- buttonNewWithLabel "Click␣me"
10   onClicked button (putStrLn "clicked!")
     containerAdd window button
     widgetShowAll window
     onDestroy window mainQuit
     mainGUI
```

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

[3] `http://www.gtk.org`

[4] `http://www.wxwidgets.org`

[5] `http://qtproject.org`

[6] Based on `http://code.haskell.org/gtk2hs/docs/tutorial/Tutorial_Port/chap4-1.xhtml`

Code that uses these GUI libraries "feels" imperative [37, p. 522–527]. Furthermore, event-driven architectures result in inversion of control [12, p. 36–37], here exemplified by the event handler on line 10, making reasoning about programs hard [32]. Further, commonly used design patterns for imperative interactive applications, such as the family of Model-View-Controller [26] patterns, move too much logic into the controller, causing quadratic growth of the size of its codebase with new features [39, p. 8] and leading to what informally is known as "callback hells" [9, p. 2].

Reasoning about the behaviour of GUI toolkits is also hard. Widget properties are not plain mutable variables [39, p. 13]. The semantics of GUI toolkits is often poorly defined. Furthermore, many of these libraries (including Gtk and wxWidgets) are *not thread-safe*: GUI functions must be called from the main (UI) thread, controlled by the toolkit itself[7]. Applications that need to control the execution loop or do background work must thus handle concurrency explicitly, 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 [19]. Moreover, the performance can be comparable to that obtained using imperative languages [27], making low-level bindings a customary choice for CPU-demanding multimedia.

2.2 Functional GUI Toolkits

Functional Programming can address the concerns discussed above by defining a pure API modelling 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 impure work, projecting the changes onto the screen. Objects I/O [1], implemented in Clean, is an example where GUIs are pure values and event handlers apply transformations to the world 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 widgets themselves. Fudgets [5] 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. A number of combinators connects fudgets and determines their visual layout. However, and just for that reason, there is no way to connect visually non-adjacent widgets. Gadgets [35], 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.

To cover the whole GUI domain, one would need to define a type, an implementation, and a set of operations for each kind of widget supported. As GUI toolkits are notoriously large[8], this results in *very large codebases with high maintenance costs*[9], rendering some projects unrealistic in the long term. Furthermore, platforms differ slightly, and creating a GUI abstraction that provides all the features of each platform under a common, clean interface has proved challenging. The opposite, maintaining several (similar) sets of code, only exacerbates the maintenance costs.

A different school of thought seeks to generate GUIs automatically based on the types of expressions. GuiTV [11], a Haskell implementation for type-based GUI generation, can construct widget compositions for functions, to provide the arguments and show

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

[8] As an example, GTK2hs, the Haskell bindings to GTK+, currently exports 6185 symbols.

[9] This extra cost is not exclusive to functional languages, but the result of defining a new API for a large domain. In languages allowing effects everywhere such an expensive mediating GUI abstraction would hardly be justifiable.

```
import Fudgets

main = fudlogue
         (shellF "Up␣Counter" counterF)

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

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

Figure 1. A sample Fudgets program. `intDispF` is an integer text fudget that uses a text box for interaction, `mapstateF` keeps a counter, and `buttonF` is a button fudget. `>==<` chains fudgets from right to left, placing them next to one another in the GUI.

the result, eliminating one level of indirection between models and visualisations (see Fig. 2).

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

Figure 2. A Tangible Value and the GUI generated based on its type, with text boxes being used to interact with `String`s.

A similar idea is used in iTask [31], a client-server, task-oriented, state-transforming web programming framework for Clean. iTasks seeks to address large-scale architectural GUI concerns, generating user interfaces automatically from types [2] and then rendering these to a browser.

Mapping a type to exactly one kind of widget is arguably a bit inflexible: there may be more than one right way to interact with values of a specific type. To circumvent this, type wrappers (e.g. Haskell's `newtype`) can be used, but only at the expense of additional code to handle artificial type distinctions.

2.3 Functional Reactive Programming

Functional Reactive Programming (FRP) [6, 10, 34] is a paradigm for reactive applications focusing on data dependencies with state handled in a referentially transparent manner. Key points include:

- *Referential transparency*: Values that change with time are defined as *signals*. Conceptually they are functions from time to a value ($Signal\ \alpha \approx Time \to \alpha$).
- *Reactivity*: Signals may depend on past and present values of other signals.
- *Interactivity*: Designated signals represent user input. Other designated signals represent system output.

Time is often taken as continuous and represented as a non-negative real number. The parameter α specifies the type of values carried by the signal. For example, the type of an audio signal might be $Signal\ Sample$, while that of the mouse position could be $Signal(Int, Int)$.

FRP is not a GUI toolkit, but rather a particular way to implement stateful reactive processes. It has been used in conjunction with libraries for graphics, multimedia, and GUIs.

The core idea to grasp is that FRP signals are defined *by their values over time*: there is no separation between a signal's definition and its value. The limitations of this approach become apparent in GUI applications, where circular dependencies between unrelated elements are commonplace.

Consider the program Xournal[10] (Fig. 3). There are four different ways to move from one page to the next: with the toolbar buttons (top), by dragging the central area with the mouse (centre left), by scrolling down the page (centre right), and with the bottom toolbar controls. Each of these acts *both as an input and an output*: no matter which method we use, the central area will show different contents, the scroll bar will be at a different position, the toolbar buttons will be enabled or disabled depending on whether there are more pages before or after, and so on.

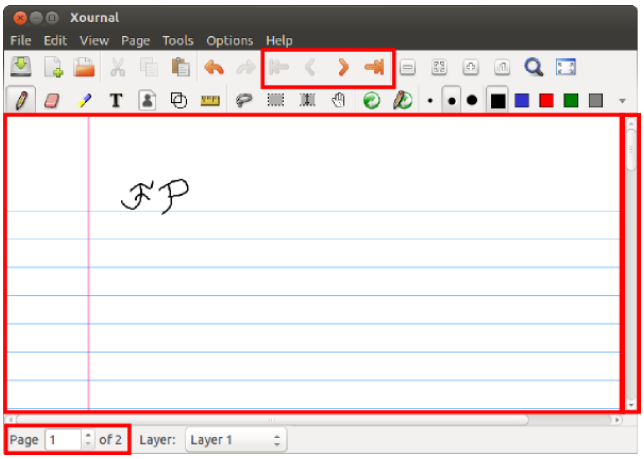

Figure 3. A screenshot of Xournal, showing different ways to change the page number.

The following pseudo-FRP code illustrates these mutual dependencies[11]:

```
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`
    ...
  ]
```

[10] `http://xournal.sourceforge.net`

[11] We use the symbol `<-` to define signals: the left-hand side being the signal and the right-hand side being its definition. We lift functions onto signals using `liftA2`. This is not based on any specific FRP implementation, but illustrates that signals are defined *in terms* of other signals.

We need `currentPage` to define `toolbarButtonRight` (line 1), `pageSelectionEntry` (line 6) and `pageArea` (line 9), but we need all three (and probably many others) to define the value of `currentPage` (line 11). These mutually dependent elements have to be defined together, thus impairing modularity and separation of concerns. This only gets worse as the codebase grows.

Some FRP implementations and languages offer mechanisms to work around this problem. Elm [7], for instance, offers *handles* to push specific changes onto widgets, thus helping to break cycles involving interactive visual elements. Reactive Banana[12] offers *sinks* for each WX widget property, to which a signal can be attached. These are, 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. Reactive Values and Relations

Our proposal for addressing the issues discussed in section 2 is based on a concept we call *Reactive Values* (RVs). A Reactive Value is a *typed mutable value with access rights and subscribable change notification.*

RVs provide a *uniform interface* to GUI widgets, other external components such as files and network devices, and application models (the conceptual problem representation in MVC [26]). Each entity is represented as a collection of RVs, each of which encloses an individual property. RVs can be transformed and combined using a range of combinators including (n-ary) function lifting and lens application [13].

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

RRs are defined separately from RVs. Indeed, relations involving the same RVs can even be defined in separate modules. This is in contrast to FRP signals, which are defined by their dependencies on other signals. Allowing RRs to be defined separately is a *key strength* of our approach as this promotes separation of concerns and modularity (Sec. 2.3). The work presented here addresses static RRs and provides no way of removing installed relations. So far we have not found this to be a major limitation.

MVC controllers [26] can thus be seen as sets of Reactive Relations. Because the model is reactive and notifies of changes to it, the controller no longer needs to know how changes propagate within the model. This allows us to move more of the problem's logic into the model (whenever it conceptually belongs there), while minimising data propagation from the model to the view.

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

3.1 Reactive Values

A *Reactive Value* (RV) is characterised by a *type* and an *access property*:

- The type is the type of the element that it stores or represents. In our implementation, this can be any Haskell data type.
- The access property states whether the reactive value is read-only, write-only or read-write.

We use the following types to represent reactive values, parameterised over the type of values they contain:

```
data ReadOnly  a = ...
data WriteOnly a = ...
data ReadWrite a = ...
```

In our implementation a monad is used to trigger notifications and manipulate mutable properties. For flexibility, types and classes are parameterised over this monad, enabling easier integration with different backends. We simplify the exposition by always using the IO monad, and removing it from type signatures.

It is useful to classify RVs based on whether we can read from or write to them. We capture that with the following type classes:

```
class Readable r a where
  read      :: r -> IO a
  onCanRead :: r -> IO () -> IO ()

class Writable r a where
  write     :: r -> a -> IO ()

class (Readable r a, Writable r a) =>
      ReadableWritable r a
```

`ReadOnly` is an instance of `Readable`, `WriteOnly` is an instance of `Writable`, and `ReadWrite` instantiates all three.

Example (Reactive Values) We borrow the following examples from a posture monitoring application (Sec.4.1.2) that uses a webcam to monitor the user's sitting posture and trigger a warning when it differs too much from a reference posture.

Users can customise how much time needs to pass until a warning is triggered. In the GUI (Fig. 4) this is configured using a spin button (text entry for numbers).

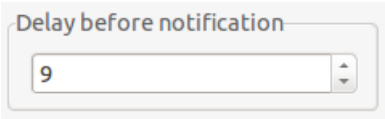

Figure 4. Entry used to configure the warning delay.

Following the MVC architectural pattern we would expect to have definitions like the following[13]:

```
-- UI (Gtk) module
delaySpinButton :: SpinButton
delaySpinButton = ...

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

In our approach, a reactive application would typically *also* include the following two definitions:

```
delayEntry :: ReadWrite Int -- UI: Entry value
delayEntry = ...            -- (to be implemented)

notificationDelayField :: ReadWrite Int -- Model
notificationDelayField = ... -- (to be implemented)
```

The first of the two reactive values represents the numeric value held in `delaySpinButton`; the second one represents the field `notificationDelay` of the model.

[12] `https://wiki.haskell.org/Reactive-banana`

[13] We omit the implementation because the details of how the spin button and the model are created are irrelevant to the discussion at this point.

In Section 3.4 we will connect Reactive Values to keep them in sync during program execution. The two reactive values above are *both read/write* and *have the same type*, which will make connecting them bi-directionally straightforward. But it may not always be so easy. For example, in our application, a text entry gives users the possibility of customising the sound played when the posture is incorrect. In our program we need to connect the following two RVs:

```
soundEntryText :: ReadWrite String       -- UI
soundEntryText = ...

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

which have different types. In sections 3.3 and 3.4 we will see how to adapt the types when they do not match and how to connect different kinds of RVs so that they stay in sync during execution.

3.2 Creating Reactive Values

Reactive Values are created from and linked to an underlying entity. These "backing" entities can be external (widgets, files, etc.) or internal (pure values and application models).

In this section we limit our discussion of GUIs to Gtk+, but our approach can be used with other toolkits such as wxWidgets or Qt. Our implementation [24] includes examples of reactive applications written with different toolkits.

3.2.1 Externally-backed Reactive Values

Some reactive values represent mutable entities that exist outside our functional code. These could be, for instance, GUI widgets and their properties, files, network connections or hardware devices.

Graphical User Interfaces In Gtk+ terminology, widgets (graphical components) have attributes (properties) with access rights (read/write). Widgets may trigger *signals* to execute event handlers when attributes change or when other events take place (clicks, key presses, etc.).

Checkboxes, for instance, have attributes such as the state (checked/unchecked) and whether users can interact with them (enabled/disabled). Clicking on an enabled checkbox toggles its state and fires an event that can be handled programmatically (Fig. 5).

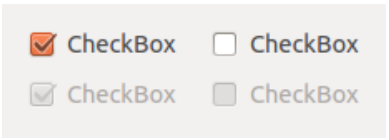

Figure 5. Checkboxes can be checked/unchecked (left/right columns) and enabled/disabled (top/bottom rows).

There is a one-way correspondence between Gtk+'s signals and attributes and our reactive values. In most cases, an attribute defines a reactive value, possibly accompanied by the signal (event) triggered when the attribute changes (Fig. 6).

Our API covers most of the essential widget properties in gtk2hs [21]. We also provide a generic signal/attribute-based interface, suitable for widget properties not specifically supported. Additionally, we have published (experimental) reactive layers for wxWidgets and Qt [24].

Example (GUIs) To access a text entry's text, we provide:

```
entryTextReactive :: (EditableClass e,
                      EntryClass e)
                  => e -> ReadWrite String
```

Gtk+	Reactive equivalent
Read-only attribute (+ associated event)	ReadOnly Reactive value
Write-only attribute	WriteOnly Reactive value
Read-write attribute (+ associated event)	ReadWrite Reactive value
Signal or event	ReadOnly Reactive value

Figure 6. Correspondence between Gtk+ and our Reactive Values

which, for a given Gtk+ text entry, returns an RV representing the text in the entry. The text can be accessed and modified via the Reactive Value, which fires notifications when it changes.

We may also be interested in detecting events that do not correspond to any property or carry data (for instance, a button being clicked). Events can always be seen as read-only RVs carrying no information, eg.:

```
buttonActivateField :: Button -> ReadOnly ()
```

which, for a given button, defines an RV that fires a notification every time the button is clicked.

Files Reactive Values can be used to interact with files and other sources/sinks of information. The predefined function `fileReactive` creates an RV enclosing a file:

```
fileReactive :: FilePath -> ReadWrite String
```

We use a file monitoring library[14] that relies on OS facilities to monitor files for changes without polling. This results in an RV that will notify dependent RVs when the file changes on disk.

Example (Files) The following RV is connected to the file `myFile.txt`. When the RV is written to, so is the file. When the file changes, RV subscribers are notified:

```
myFile :: ReadWrite String
myFile = fileReactive "myFile.txt"
```

Network Similarly, an experimental network reactive layer allows sockets to be seen as RVs. For instance, the function:

```
udpSink :: HostName -> String
        -> IO (WriteOnly String)
```

creates a writable reactive value that sends any text written to it to the specified host and port using User Datagram Protocol (UDP).

3.2.2 Internally-backed Reactive Values

Library users have access to the value constructors of different RVs, and can thus define RVs that enclose pure values. In most applications we want to be able to detect when values change, update other RVs accordingly and guarantee thread-safe access to the Reactive Value.

We provide a library with a default implementation of "very light" RVs that fulfils all of these requirements. The library offers several RV constructors, of which the default one compares the value of an RV with the previous one before setting it, to break unnecessary propagation loops.

This solution works well for simple programs, but it is suboptimal for very large applications: a change to only one part of a value (for instance, the first component of a tuple) will provoke forward propagations to RVs that depend only on other parts that did not change (for instance, the second component of the same tuple).

[14] `https://hackage.haskell.org/package/fsnotify`

Protected Models To address the aforementioned scalability concerns we define an abstraction that encloses an application's model, called Protected Model, implemented as a polymorphic, thread-safe mutable variable with change detection and an event dispatching thread, parameterised over two types[15].

The first type argument of `ProtectedModel` represents the type of values stored in it, that is, the pure model it encloses. The second argument acts as an identifiable reference to a part of the model and is used to identify what has changed[16]:

```
data (Ord e, Eq e) => ProtectedModel a e =
 ProtectedModel
  { reactiveModel :: TVar (ReactiveModel a e)
  ...
  }

data (Ord e, Eq e) => ReactiveModel a e =
  ReactiveModel
    { basicModel      :: a
    , eventHandlers   :: M.Map e [IO ()]
    , pendingHandlers :: [IO ()]
    ...
    }
```

We make use of STM TVars [18] to guarantee exclusive access and atomic modifications.

Protected Models can be created with the function `startProtectedModel`. This function also starts a dispatcher thread that executes pending handlers:

```
startProtectedModel :: (Ord e, Eq e)
                    => a -> IO (ProtectedModel a e)
```

In the following we will see how to define RVs that give access to only parts of a Protected Model.

Projecting Protected Models to Reactive Values The main difference between a plain RV and a Protected Model is that the latter is intended to be a collection of RVs. Thus, one should define RVs that project specific parts of the Protected Model.

To make that extra layer as simple as possible for the users of our library, we provide a high-level API that uses Template Haskell to define RVs that represent projections of fields of the model. For instance, given:

```
data Model = Model {
        language :: Maybe Language
        ...
}

data ModelChange = LanguageChanged
                 | ...
 deriving (Eq, Ord)
```

the following call to Template Haskell in a module (all the referred types, and additional RV libraries, must be in scope):

```
protectedField "Language" [t|Maybe Language|]
               ''Model ''ModelChange
```

generates a definition with signature:

```
languageField :: ProtectedModel Model ModelChange
              -> ReadWrite (Maybe Language)
```

[15] Our signature uses the type class `Event` for identifiable changes, which is an instance of `Ord` and `Eq`. Events have additional, orthogonal uses in our framework. To facilitate understanding, we present a simpler version here.

[16] In some of our programs, we overload the type `e` with semantic information about the nature of the change itself. See [39, p. 33] for more details.

Of course, a lower level API to Protected Models and Reactive Values is also available, which can be used in case the given Template Haskell is not adequate for the user's needs.

Protected Models can incorporate more machinery than simply change detection and event dispatching. For instance, in the SoOSim UI[17] and Gale IDE (Sec. 4.1.1), the Protected Model incorporates a change-aware undo/redo queue. The model is extended with three operations to control the queue, which can be used by the controller. The RVs generated using `protectedField` are the same.

Protected Models allow us to hide other design decisions, such as having a global event dispatcher vs executing events in new threads as they come in. We believe that this ability to introduce orthogonal features without affecting the rest of the codebase is another key strength of our framework.

3.3 Transforming and combining RVs

Reactive values can only be connected if they have compatible types. We can transform and combine reactive values by lifting (n-ary) functions, by applying lenses, and by letting one control the other (*governance*).

Unary lifting A function of type `a → b` can be applied to a reactive value in one of two ways:

- To write values of type `a` into an writeable RV of type `b` (converting the `a` into a `b` *before* writing).
- To read values of type `b` from a readable RV of type `a` (converting the values *after* reading).

This implies that:

1. Lifting *one* function onto a read-write reactive value will render a read-only or write-only reactive value.
2. To produce a read-write reactive value, we need to lift *two* functions, one for each direction of the transformation (Fig. 7).

We thus define three unary lifting combinators:

```
liftR :: Readable r
      => (a -> b) -> r a -> ReadOnly b
liftW :: Writable r
      => (b -> a) -> r a -> WriteOnly b
liftB :: (a -> b, b -> a)
      -> ReadWrite a -> ReadWrite b
```

Read-only RVs are covariant in the read end, Write-only RVs are contravariant in the write end.

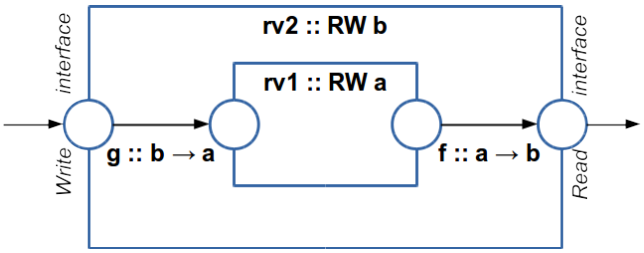

Figure 7. `rv2 = liftB (f, g) rv1`

Example (lifting) Continuing with our previous example, we might want to render the language selection in a label, for which we need to transform the `Maybe Language` from our model into a `String`. We might do so as follows:

[17] `https://github.com/ivanperez-keera/SoOSiM-ui`

```
showLangSelection :: Maybe Language -> String
showLangSelection (Just English) = "EN"
showLangSelection (Just Spanish) = "ES"
showLangSelection Nothing        = "--"

langText :: ReadWrite (Maybe Language)
         -> ReadOnly String
langText lang = liftR showLangSelection lang
```

If we are given a function to parse the language selection, then we can easily make the reactive value writable as well:

```
readLangSelection :: String -> Maybe Language
readLangSelection "EN" = Just English
readLangSelection "ES" = Just Spanish
readLangSelection _    = Nothing

langTextRW :: ReadWrite (Maybe Language)
           -> ReadWrite String
langTextRW lang =
  liftB (showLangSelection, readLangSelection) lang
```

Read-only Reactive Values are `Functors` and `Applicatives`. Write-only Reactive Values are `Contravariant` functors. Using the Applicative and Contravariant infix lifting operators, we can write clearer, less verbose code.

```
langText :: ReadOnly (Maybe Language)
         -> ReadOnly String
langText lang = showLangSelection <$> lang
```

Read-write RVs are instances of our own `GFunctor` (an abstraction for types that are covariant in one direction and contravariant in the other), for which we have defined the operator `<$$>`, analogous to `<$>` for applicatives. For example, the following is a more succinct definition of `langTextRW`:

```
langTextRW :: ReadWrite (Maybe Language)
           -> ReadWrite String
langTextRW lang =
  (showLangSelection, readLangSelection) <$$> lang
```

When lifting functions onto read-write reactive values, it is often desirable that the transformation be an *isomorphism* (in which case we would lift the function by the functor and the inverse by the contrafunctor). Given the limitations of Haskell, we cannot but trust users in this respect, providing only a small facility for *involutions*:

```
reversedText :: ReadWrite String
reversedText = (involution reverse) <$$> textValue
 where textValue :: ReadWrite String
       textValue = ...
```

Not using real isomorphisms may impact performance. Our default setters compare the new values to the old ones (if they are instances of `Eq`). This stops unnecessary data propagation and breaks loops. However, if the inverse provided is not the true inverse, the value that propagates in the inverse direction after a change may cause a new propagation. It is therefore necessary to provide inverses that will lead to a fixed point. This will be discussed further in section 6.

N-ary lifting Similarly, we can lift n-ary functions into RVs using analogous functions (eg. `liftR2`, `liftW2`, `liftB2`, etc). The signatures of `liftR2` and `liftW2`, for instance, are:

```
liftR2 :: (Readable r1 a, Readable r2 b)
       => (a -> b -> c) -> r1 -> r2 -> ReadOnly c

liftW2 :: (Writable r1 b, Writable r2 c)
       => (a -> (b,c)) -> r1 -> r2 -> WriteOnly a
```

N-ary lifting onto read-only values can also be achieved using applicative syntax [28].

Example (n-ary lifting) We could, for instance, render several configuration parameter in a tuple, to later show them in a label, as follows:

```
notificationDelay :: ReadWrite Int
notificationDelay = ...

correctionFactor :: ReadWrite Int
correctionFactor = ...

configurationPair :: ReadOnly String
configurationPair =
  liftR2 (\d f -> show (d,f))
         notificationDelay
         correctionFactor
```

Lenses Lenses [13] provide a way to *focus* on subparts of data structures by means of a getter and a setter that read from and inject values into an existing structure. Lenses are compositional (they can be combined using a notation similar to function composition), and can be derived automatically for some type definitions.

Lens application onto RVs is a specific form of lifting bijections. We provide a specific lens lifting combinator:

```
(<$$$>) :: Lens' a b
        -> ReadWrite a -> ReadWrite b
```

Example (Lenses) Given the lens `(_1) :: Lens' (a,b) a`, which focuses on the first component of a pair, one can write:

```
window1Top      :: ReadWrite Int
window1Top      = _1 <$$$> window1Position
window1Position :: ReadWrite (Int, Int)
window1Position = ...
```

Governance Another possible way of combining RVs is by letting one control another. Consider, for instance, the case in which one wants changes to a text box to be "reported" *only* when the button is depressed. If we use `liftR2` to combine them, both clicks on the button and text entry changes will trigger notifications. To address these situations, we provide the function:

```
governing :: Readable r a
          => r -> ReadWrite b -> ReadWrite b
```

which defines a new Reactive Value that encloses the value in the second argument, and notifies of changes only when the first RV changes. An analogous function is provided for read-only RVs.

Examples (governance) Following the case described above, we often want the text of an entry not to be synchronised or passed around, except when a button is clicked. We can use governing to create a RV that encloses the entry's text, but whose changes are only propagated when the user clicks the button:

```
buttonAndEntry :: ReadWrite String
buttonAndEntry =
  button1Clicks `governing` textEntry1Text
button1Clicks  :: ReadOnly ()
button1Clicks  = ...
textEntry1Text :: ReadWrite String
textEntry1Text = ...
```

3.4 Reactive Relations

So far we have given ways to *create* reactive values, but we have not given any way to *relate* readable and writable RVs to allow changes to be propagated correctly to achieve overall consistency (for instance, to synchronise two text boxes, or an RV that represents a Protected Model field with one that encloses a widget attribute).

We introduce rule-building functions to capture the idea that two reactive values must be "in sync" for all time. The functions $<:=$ and $=:>$ (depending on the direction of change propagation) build directional synchronisation relations. The source value (the origin of the change) must be *readable*, the destination must be *writable*, and they must contain values of the same type. To simplify code further we provide the function $=:=$, syntactic sugar for two directional relations. Their types are as follows:

```
(<:=) :: (Writable r1 a, Readable r2 a)
      => r1 -> r2 -> IO ()

(=:>) :: (Readable r1 a, Writable r2 a)
      => r1 -> r2 -> IO ()

(=:=) :: (ReadableWritable r1 a,
          ReadableWritable r2 a)
          => r1 -> r2 -> IO ()
```

Example (reactive relations) In our posture monitoring application we need a GUI to manipulate the delay until a notification is presented. We have already seen how to define each reactive value (for the GUI and the model). To keep them in sync we write:

```
delayEntry =:= notificationDelayField
```

This combinator installs the necessary change listeners on each RV, so that the other RV is updated when either changes. For another example, based on `configurationLabel` defined earlier, we can show the correction factor and the warning delay in a label:

```
configurationPair :: ReadOnly String      -- Model
confLabel         :: GtkLabel             -- UI
confLabelString   :: ReadWrite String     -- UI
confLabelString   = labelString confLabel

rule = confLabelString <:= configurationLabel
```

where `configurationLabel` was defined in the last example that accompanies Section 3.3. Note that the rule is directional: there is no need to update the model from the view in this case because labels are not interactive.

To answer a question posed at the end of section 3.1, we can use lifting and a relation to synchronise two RVs of different types:

```
soundEntryText :: ReadWrite String       -- UI
soundEntryText = ...
soundField :: ReadWrite (Maybe FilePath) -- Model
soundField = ...

stringToMaybe :: String -> Maybe FilePath
stringToMaybe "" = Nothing   -- Default sound
stringToMaybe fp = Just fp   -- Sound in file fp

rule =
  soundEntryText =:=
    (fromMaybe "", stringToMaybe) <$$> soundField
```

3.5 Choreographies

GUI programs often contain common, re-occurring patterns. By parameterising Reactive Relations over Protected Models and Views containing certain Reactive Values, we can describe sets of rules in separate libraries that can be reused within the same application and across multiple applications. We refer to these abstract patterns as *choreographies*.

We present a very simple one that we have found both illustrative and useful: showing the file name in the title bar of the main window. Type classes capture the requirements of model and view.

```
class ModelWithFilename m e where
  filenameField :: ProtectedModel m e
                -> ReadWrite FilePath

class ViewWithMainWindow v where
  mainWindow :: v -> Window

composeTitleBar :: (ModelWithFilename model event,
                    ViewWithMainWindow view)
                => String
                -> model -> view -> IO ()
composeTitleBar programName model view =
  (composeTitle `liftR` filenameField model)
    =:> (windowTitleReactive (mainWindow view))
 where composeTitle fp = fp ++ " - " ++ programName
```

Choreographies are usually more complex and not limited to one relation. They can contain internal models and views, and spawn threads. For example, the choreography that offers to save files when a program is closed contains two rules (one to present the confirmation dialog, one to save the file), introduces one additional model type class and contains a view of its own (the dialog).

4. Experience

An implementation of Reactive Values is available online as a collection of libraries, as part of the Haskell GUI Programming toolkit Keera Hails [24]. They provide definitions of Reactive Values and Reactive Rules, and bindings for a series of backends, including Protected Models, Gtk+ widgets and properties, files, network sockets, FRP signal functions (using Yampa [34]) and Nintendo Wii Controllers. It also includes libraries to simplify common architectural patterns (MVC) as well as choreographies often needed in different kinds of applications. At the time of this writing, the libraries comprise over 7K lines of code.

We have used our approach to develop several real-world applications, currently amounting to slightly over 25K lines of Haskell (not counting comments, empty lines or code generated automatically by our library, using Template Haskell or the Keera Hails project generator, which generates an initial project skeleton).

Examples of the software created include an interactive tool to visualise Supercomputer Operating System node simulations (Fig. 8) [3], a webcam-based posture monitor (Sect.4.1.2), a OCR-based PDF renamer and a Graphic Adventure IDE (Sect. 4.1.1). We have also published several demonstration applications and small examples, such as an SMS sender[18] and a replacement for WMGUI[19], the Nintendo Wii Controller debugging GUI available on most Linux distributions (Fig. 9).

4.1 Evaluation

4.1.1 Gale IDE

Keera Gale is a graphic adventure development IDE written entirely in Haskell[20]. The IDE uses Gtk+ for the user interface, and al-

[18] `https://github.com/ivanperez-keera/keera-diamondcard-sms-trayicon`

[19] `https://github.com/keera-studios/hails-reactive-wiimote-demo`

[20] `http://keera.co.uk/blog/products/gale-studio/`

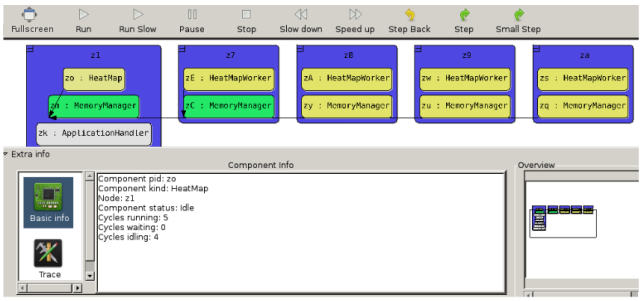

Figure 8. SoOSim UI, an interactive visualisation tool for Supercomputer Operating System node simulations.

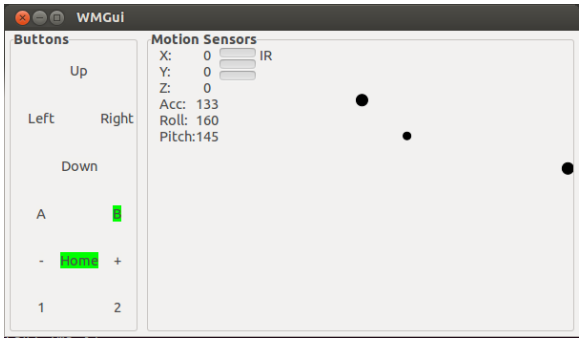

Figure 9. A demo that shows the state of a Nintendo "Wiimote".

lows users to create graphic adventure games without prior knowledge of programming. Users can define screens, characters, actions and reactions, enable conversations and customise the game interface. Other stages of the game design process, such as storyboarding and state transition diagrams, are also supported. The IDE is accompanied by a graphic adventure engine written in Haskell using SDL2, that has been tested on Windows, Linux and Android. The final distributable file can be generated directly from the IDE using only the GUI. Games created using Gale IDE are currently being beta-tested on Google Play.

The program uses MVC as its main architecture. The IDE features, at the current time, 385 modules of Haskell code, without including the engine or other auxiliary components. 228 of those modules conform the application's controller and contain 50 per cent of the code. Template Haskell is used to generate the View (from glade files) and the reactive layers of the model, decreasing the number of lines of code further.

A separate thread is used to handle a responsive progress dialog when the distributable files for the game are being generated. The controller starts that thread, but further communication occurs only indirectly through the protected model (Fig. 12).

The controller currently contains 75 reactive rules. We have ported imperative MVC Haskell code to this new reactive interface, and using Reactive Values and Rules makes the controller's modules between 50 and 66 per cent smaller (in lines of code, without comments or empty lines) compared to code that had already been optimised to avoid code duplication due to bi-directional synchronisation[21].

[21] In bi-directional synchronisation one needs to obtain the values on both sides, compare them and possibly update one side. Our original code already received the direction of the update as a parameter, so that the code that polled the view and the model could be shared for both directions.

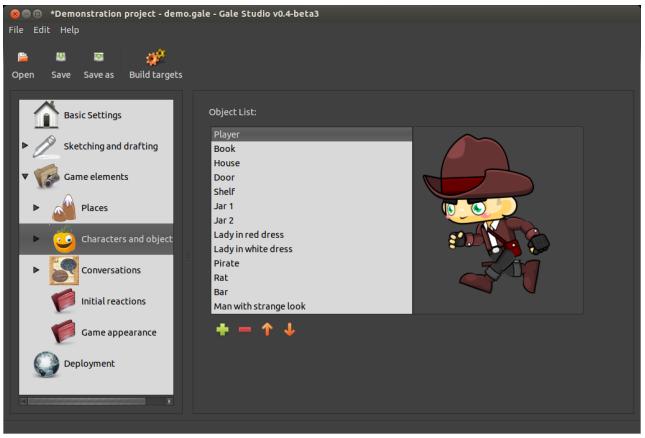

Figure 10. GALE IDE's object preview screen, showing an animation with each character's default state. Double clicking on any object opens the object details screen, where users can modify the object properties by selecting states, animations, actions applicable to them and reactions to those actions.

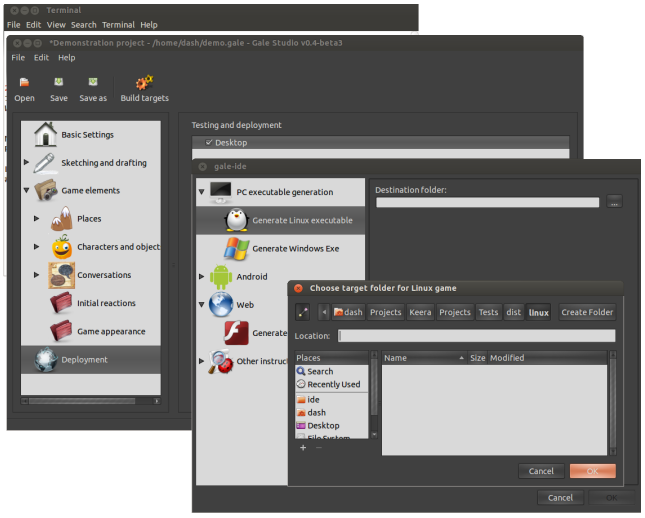

Figure 11. GALE IDE can target Windows, Linux, Android and Web. This screenshot of the running application shows three nested windows: the main application, the target/distributable selection window, and the target directory selection dialog.

Compared to the whole application's codebase, we estimate this approach to have saved us between 25 and 35 per cent of code. Combined with being able to generate UIs and Reactive Fields using Template Haskell gives us a combined estimate of 35 to 45 percent of lines of code saved.

The controller makes heavy use of choreographies to eliminate boilerplate code. Re-occurring patterns include synchronising the selection on a tree view and on a tab page group (using the tree view to change the tab page), and efficient interaction with dynamic lists of elements (scenes, objects, etc.).

Furthermore, because Reactive Values encapsulate both bi-directional access and the relevant notification subscription mechanisms in one unique entity, we have observed that we are less likely to make errors such as installing handlers on the wrong events.

We believe that Keera Gale IDE clearly shows that our approach addresses all three problems introduced in section 2: it uses a

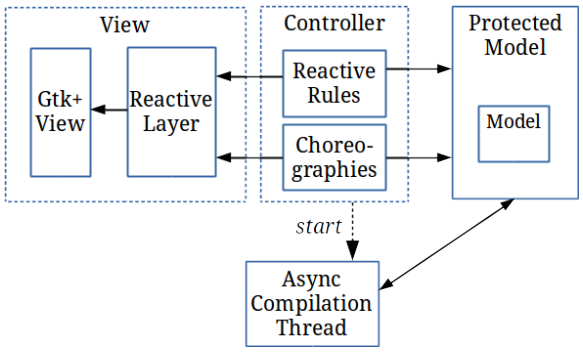

Figure 12. GALE IDE's architecture. The compilation thread is started by the Controller, but further communication takes place only indirectly, through the Protected Reactive Model.

standard GUI toolkit (Gtk+), it enables functional style through the use of reactive relations, and it is large and complex enough to prove that our approach scales well in terms of code modularity, and even enhances it.

4.1.2 Keera Posture

Keera Posture is a posture monitor written in Haskell using OpenCV for image recognition and Gtk+ for the GUI [22].

The program works by comparing the current position of the user (estimating the distance based on the size of the head) with a reference position given during program calibration. When both differ "too much", a warning is shown.

Users can customise the sensitivity, the form of the warning (popup message, message bubble and/or sound), the sound being played, the language and the webcam being used. The initial calibration uses a step-by-step assistant. Users can pause the program by clicking on its icon, located in the system tray bar.

The program has been implemented for end-users and thus care has been placed on providing common usability features and an intuitive user interface. Both Windows and Linux are supported.

Like Gale IDE, Keera Posture runs several threads to keep the GUI responsive while doing image recognition. Changes in the posture are communicated to the GUI only indirectly, through the protected, reactive model.

Of the 53 modules included in the program, the Model contains 13 (plus 4 which are generated automatically). The Model constitutes 30% of the code (measured in lines, without comments or empty lines) and exposes 16 Protected Model fields (projections of model parts onto Reactive Values). The Controller contains 30 modules, which constitute 50% of the code and comprise 29 Reactive Relations. The image recognition module contains 10% of the code, and the View (generated during compile time from a Gtk+ Glade file, using Template Haskell) contains only 4%.

Keera Posture is a clear demonstration of how, using the right abstraction, one can write software that addresses real problems, in a purely functional way, with minimal boilerplate code. Also, through the use of Reactive Values and Relations, it exemplifies how one can limit the side effects of using imperative bindings mainly to the GUI, without sacrificing any of the features that standard GUI toolkits offer.

[22] `http://github.com/keera-studios/keera-posture`

4.2 Summary

Using our solution for GUI programming we have observed benefits in terms of separation of concerns, modularity, reduction of code size, and dealing with concurrency.

Our MVC controllers [25] no longer know about the internals of models, nor how change propagates within them. Most view-model synchronisation is now done using separate, abstract, easily readable rules. Also, as callbacks are no longer explicit and many relations are bi-directional, code duplication in the controllers has been eliminated, reducing their size to less than half.

The ability to state synchronisation constraints separately from reactive values through reactive relations allows the constraints to be grouped by the feature they implement rather than by the UI or model elements involved. This promotes separation of concerns and allows orthogonal features, like saving or printing, to be added or removed locally. Individual constraints can easily be disabled, which is a great debugging aid. They can also be factored out in choreographies (Sec. 3.5) that can be reused across applications.

Finally, thanks to thread-safe models, our applications accommodate concurrent threads easily. This was exploited in the PDF Renamer and the Game IDE to asynchronously transfer files while showing a responsive, cancellable progress dialog. Similarly, in the posture monitor, one thread records images from the webcam while another shows warnings as popup messages when the posture is incorrect. The threads do not communicate explicitly with each other, but rather modify the application's model. Any configuration change through the GUI's preference panel is applied to the model and then immediately used by the posture detection thread.

5. Related work

5.1 Comparison to Functional GUIs and FRP

Fudgets [5] is a functional GUI framework structured around the notion of *fudgets*: visual, interactive data transformers with one input and one output. Fudgets was reviewed in Sec. 2.2. Limitations of Fudgets include not supporting connection of visually non-adjacent widgets and that mutually interconnected fudgets must be defined together. Our approach overcomes such issues by using one Reactive Value (RV) per widget property and by allowing separately defined RVs definitions to be related through Reactive Relations (RR). Gadgets [35] is similar to Fudgets, but tries to overcome some of its limitations. However, as discussed in Sec. 2.2, by their nature, both Fudgets and Gadgets need to provide a fudget/gadget definition for every single GUI widget of interest, meaning that such libraries necessarily become very large. This leads to high maintenance costs, which is one reason Fudgets is only available on a selection of Unix-like platforms and has not seen any major update since the late 1990's. In contrast, RVs and RRs have a much smaller footprint and are designed to work in conjunction with existing GUI toolkits on any platform, thus side-stepping this issue.

A key difference between Functional Reactive Programming (FRP) [6, 10] and our approach is that ours allows separately defined reactive entities to be related, while an FRP signal is defined in terms of the signals it depends on once and for all. As discussed in Sec. 2.3, this aspect of FRP often leads to scalability issues in large applications, in particular for mutually recursive signals. Unlike FRP, our approach is agnostic about time, thus not lending itself to reasoning about temporal properties. Soft real-time guarantees have been studied for at least some FRP variants [23]. It may be possible to give a semantics for RVs in terms of FRP. This would provide one way to reason about RVs, which certainly would be interesting. Nevertheless, so far, we have not experienced any issues with timeliness of responses intrinsic to our framework.

Some FRP implementations, like Elerea [38], take special precautions to break change propagation loops. We use equality tests

in setters to minimise change propagation. For loops, this means that propagation stops when reaching a fixed point. It is thus crucial that the functions provided for transforming read-write RVs are each others inverses, or propagation could go on indefinitely. This problem also exists in other frameworks such as Yampa [34] or object-oriented GUI toolkits like Qt. Our approach does not provide further guarantees, but specifying both directions of the transformation in a single place may facilitate discovering bugs quickly.

There are some similarities between RVs and iTask's [31] *Uniform Data Sources* (UDS) [30], but UDS has no support for subscription to change notification. Further, a central aspect of iTask is automatic generation of GUIs from types with a particular focus on Web applications, whereas RVs and RRs provide generic, re-usable infrastructure for GUI programming and more.

There are also similarities to Lenses [41]. However, RVs are not lenses as they in general do not satisfy any lens-like laws. Nevertheless, RVs can beneficially be used together with lenses and we view them as complementary. Recent developments in monadic lenses applied to User Interfaces[23] and lenses with notifications [8] could help simplify our formalisation to its true core objective: a data-binding language between typed reactive elements.

Parametric Views [8] are based on the same basic operations (get, put, subscription) as RVs. Further, like Parametric Views, our Protected Models make *change* a first-class entity to minimise data propagation and screen refreshes. One difference is that our setters compare previous and new values when possible to minimise change propagation and break propagation loops. Parametric Views provides a versatile notion of invalidation function to that end. Our goal, however, is to make change detection as transparent as possible for which we are experimenting with automatically deriving change definitions for Haskell datatypes [39].

5.2 Comparison with OO and Reactive Programming

From an Imperative or Object-Oriented perspective, our work is closest in spirit to Reactive Programming[24], and then in particular to *change subscription* facilities and *data binding* languages.

Reactive Values are similar to widget properties in Qt[5], which are typed, mutable attributes paired with a change event. Qt's signals and slots can be seen as read-only and write-only RVs and are versatile enough to accommodate files, sockets and other external entities. Qt further provides data binding facilities to connect signals to slots, but unlike in our approach, these are *uni-directional*, and there is no mechanism for breaking propagation loops. When using Qt as a GUI backend from our framework, we provide an intermediate library that takes care of change detection and thread safety, shielding users from such details.

Our notification system is similar to the observer design pattern [14] frequently encountered in object-oriented programming. This pattern has specific support in recent versions of Javascript in the form of `Object.observe()` [36]. The observer pattern enables detecting changes to objects, but it is necessary to install change handlers. This leads to issues of inversion of control common in event-driven programming [12, p. 36–37], and the scheme is further inherently uni-directional, unlike our bi-directional relations.

Facebook's React[25] has similar goals to the observer pattern but is more declarative. Unlike our approach, React only provides uni-directional data-binding. Like our approach, React uses change detection mechanisms to minimise data propagation, which in the case of Web sites produces minimal DOM migrations. React gathers change propagation responsibilities to a central dispatcher in an attempt to maximise throughput. In contrast, our solution opts for a middle ground, using a global dispatcher per Protected Model, but allowing different Protected Models to co-exist and even coordinate during execution.

Our reactive rules constitute a data dependency language not unlike the data-binding facilities of frameworks like AngularJS[26] and EmberJS[27]. There are, however, structural differences. AngularJS, for instance, merges data-binding, function lifting, and view declaration into a single, annotated XML tree. We believe our approach results in a more modular and abstract design, partly because it maximises separation of concerns, and partly because it allows factoring choreographies out into libraries. As we have discussed, our framework uses equality tests to minimise change propagation and break loops. This approach is typically more efficient than the *dirty-checking* used in AngularJS, but further research is needed to determine how our solution compares to the aforementioned frameworks in terms of performance.

6. Summary and Future work

In this paper we have described a functional, compositional, reactive framework that provides a uniform interface to widget properties, files, sockets, application models and other external entities. We have demonstrated how reactive values can be defined, transformed and connected. Our solution works well with different GUI toolkits, and we have implemented several non-trivial applications.

Our work has been guided by industrial experience. We have not yet undertaken formal analysis of temporal properties, but we plan to do this in the future; for example, through a semantics based on FRP. We expect to be able to reason about delays, change propagation and temporal inconsistencies.

Our solution sacrifices consistency across a network of possibly duplicated values in favour of responsiveness and scalability [16]. We rely on always being able to break circular dependencies to achieve *eventual consistency* [42]. We believe that, so long as the only circular dependencies are due to direct bi-directional liftings and relations, it is sufficient if there is a limited number of compositions of the function in one direction and the one in the other that converges to a fixed point.

We have observed constant memory consumption while profiling some applications. However, we expect the introduction of dynamic reactive relations to impact garbage collection, which we will need to take into account to avoid memory leaks.

In this paper we have not described all the tools and libraries in our framework. This includes choreographies to update dynamic lists efficiently, support undo/redo, check for updates and log errors to a visual console. Our framework also includes tools to help with internationalization and to generate application skeletons.

Eventually, out framework could evolve towards a compositional application toolkit structured around the concepts of model, view, controller, threads and relations, and a set of well-defined combinators. In such a setting choreographies could have a more precise meaning.

We would like to give more importance to the change type associated to Protected Models, similarly to what is done in Parametric Lenses. We are experimenting with automatic instance derivation solutions, to abstract users from the details of defining custom types for change/focus. We have sometimes overloaded this type to carry information about the nature of the change; a high-level abstraction over value differences might help us address these concerns [39].

[23] `http://people.inf.elte.hu/divip/LGtk/LGtk.html`

[24] `https://github.com/Netflix/RxJava`

[25] `http://facebook.github.io/react/`

[26] `http://angularjs.org/`

[27] `http://emberjs.com`

Acknowledgements

The authors would like to thank Paolo Capriotti, Robert Mitchelmore, Ambrus Kaposi, Graham Hutton, David McGillicuddy, Florent Balestrieri, Philip Hölzenspies, Jennifer Hackett for their input during multiple discussions. We thank anonymous reviewers for helpful comments on earlier drafts of this manuscript.

References

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

[2] Peter Achten, Marko Van Eekelen, and Rinus Plasmeijer. Generic Graphical User Interfaces. In *Implementation of Functional Languages*, pages 152–167. Springer, 2005.

[3] C. P. R. Baaij, J. Kuper, and L. Schubert. Soosim: Operating system and programming language exploration. In G. Lipari and T. Cucinotta, editors, *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time System (WATERS 2012), Pisa, Italy*, pages 63–68, Italy, 2012. Giuseppe Lipari.

[4] John Backus. Can programming be liberated from the von neumann style? *Commun. ACM*, 21(8):613–641, 1978.

[5] Magnus Carlsson and Thomas Hallgren. FUDGETS: A graphical user interface in a lazy functional language. (Section 6):321–330, 1993.

[6] Antony Courtney and Conal Elliott. Genuinely Functional User Interfaces. In *Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[7] Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs*. PhD thesis, 2012.

[8] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. Parametric lenses: change notification for bidirectional lenses. In *Proceedings of the Symposium on Trends in Functional Programming*, Soesterberg, The Netherlands, May 2014. Accepted for publication.

[9] 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.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[11] Conal M Elliott. Tangible functional programming. *ACM SIGPLAN Notices*, 42(9):59–70, 2007.

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

[13] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[14] 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.

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

[16] 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.

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

[18] 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, 2005.

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

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

[21] Kenneth Hoste. An Introduction to Gtk2Hs, a Haskell GUI Library. In Shae Erisson, editor, *The Monad.Reader Issue 1*. 2005.

[22] 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.

[23] 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. ACM, 2007.

[24] Keera Studios. Keera Hails - Haskell on Rails. `https://github.com/keera-studios/keera-hails`.

[25] 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.

[26] Glenn E Krasner, Stephen T Pope, et al. A Description of the Model-View-Controller User Interface paradigm in the Smalltalk-80 System. 1988.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] Brad A. Myers. Why Are Human-Computer Interfaces Difficult to Design and Implement? Technical report, 1993.

[34] 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.

[35] 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.

[36] Addy Osmani. Data-binding revolutions with Object.observe().

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

[38] Gergely Patai. Eventless reactivity from scratch. *Draft Proceedings of Implementation and Application of Functional Languages (IFL'09)*, pages 126–140, 2009.

[39] Ivan Perez. 1st Year PhD Report. `http://www.cs.nott.ac.uk/~ixp/`, December 2014.

[40] 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.

[41] 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*.

[42] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.