Slim: functional reactive user interface programming

Jaap van der Plas

August 26, 2016

M.Sc. thesis ICA-3998312

Supervisors:
dr. Wouter Swierstra
dr. Atze Dijkstra

Utrecht University

Department of Information and Computing Sciences

Abstract

Functional programming has a number of important benefits over imperative programming. However, most GUI libraries for Haskell provide an imperative API. *Functional Reactive Programming* provides a way for programming reactive systems in a purely functional style. This report details the design and implementation of *Slim*[1], an embedded DSL for Haskell that applies FRP concepts for programming user interfaces. Its strong support for implementing complex and dynamic user interfaces is demonstrated by a comparison to existing FRP libraries.

[1] `https://hackage.haskell.org/package/slim`

In dedication to my father.

Contents

1. Introduction

1.1. Imperative user interface programming

Most graphical user interfaces (or *GUIs*) are created using imperative programming languages. Programs written in such languages consist of commands and statements that affect state. Imperative programs react to events by running blocks of code called 'event handlers' (also 'callbacks' or 'observers'.)

Consider a simple Haskell[9] program that, using a hypothetical imperative UI toolkit, creates a button that shows the number of times it's been clicked:

```
counter :: IO Widget
counter = do
  let count = 0
  countRef <- newIORef count
  countButton <- mkButton "0"
  onClick countButton $ do
    count <- readIORef countRef
    let count' = count + 1
    writeIORef countRef count'
    setText countButton (show count')
  return countButton
```

Listing 1.1.: Imperative counting button

The click count is initialized to `0` and stored in a mutable reference created with `newIORef`. A button is created with the current count as its text. Then, using `onClick`, an event handler (the `do`-block) is registered to be run whenever the button is clicked. This event handler reads the value of the count variable, increases it by one and stores it before updating the button text.

The imperative model makes user interfaces easy to create: the programmer starts with a blank slate and imagines the steps the computer has to take to produce the desired result, and which steps have to be taken to handle user input events. The problems come later on when existing programs have to be changed. It becomes hard to reason about what the state of the system will be when some sequence of events has resulted in a complex series of commands and statements to be run.

Consider extending the program from listing 1.1 with a button that resets the counter:

```
resettableCounter :: IO Widget
resettableCounter = do
  let count = 0
  countRef <- newIORef count
  countButton <- mkButton "0"
  onClick countButton $ do
    count <- readIORef countRef
    let count' = count + 1
    writeIORef countRef count'
    setText countButton (show count')

  resetButton <- mkButton "reset"
  onClick resetButton $ do
    writeIORef countRef 0
    setText countButton "0"

  mkContainer [countButton, resetButton]
```

Listing 1.2.: Imperative counting button with reset

There is now an extra `resetButton` element that, when clicked, resets `countRef` back to `0`. In addition, the label of `countButton` also has to be reset to `0`. There is no longer a single, simple definition of `countButton`s label. Instead, it now depends on multiple separate events and their handlers:

- Initially the label is `0`.
- When `countButton` is clicked `countRef` is increased and used as label.
- When `resetButton` is clicked both `countRef` and `countButton`s label are reset to `0`.

This makes it hard for the programmer to keep track of all the various factors that can (or should) affect a user interface element and makes it easy to introduce bugs. In this example, for instance, the programmer might forget to update `countButton`s label and only be concerned with resetting the mutable variable. This mistake would be easy to spot in such a simple program but is easy to miss in a larger and more complex application spread out over hundreds of source files.

1.2. Purely functional user interfaces

A solution for the extensibility problem is provided by the *functional programming* paradigm. Commands and statements are replaced by *equations* that define how programs input relates to its outputs. These equations can be reasoned about more easily than command sequences and provide better modularity and composition as a result[12].

But how can equations be used to define a system that reacts to events? A promising way to do this is through *Functional Reactive Programming* or FRP for short. It provides a way for defining a reactive system declaratively without resorting to imperative commands and statements.

The program from listing 1.1 rewritten using a hypothetical FRP library would look as follows:

```
program :: IO Widget
program = mdo
  let
    count = accumB 0 update
    update = (+1) <$ onClick countButton

  countButton <- mkButton (show <$> count)
  return countButton
```

Listing 1.3.: FRP counting button

The number of clicks is tracked by `count` which is defined as a *behavior*: a time-varying value. Through the accumulating combinator `accumB` creates a behavior from an initial value of `0` and an event to update it: `update`. This update event is based on the button's `onClick` event but with the value replaced by the function `(+1)`. The button is constructed using the behavior as its label. (Note that this `mdo`-block does not imply any specific creation order; the reason for this is explained in chapter 3.)

This formulation has the advantage that the state, encapsulated by the `count` behavior, is only defined in one place. The example does not define event handlers that depend on the state; instead, the dependencies are reversed and the state depends on the events. This dependency is more clear when we extend this example with a reset button as before:

```
resettableCounter :: IO Widget
resettableCounter = mdo
  let
    count = accumB 0 (merge update reset)
    update = (+1) <$ onClick countButton
    reset = const 0 <$ onClick resetButton

  countButton <- mkButton (show <$> count)
  resetButton <- mkButton (pure "reset")

  mkContainer [countButton, resetButton]
```

Listing 1.4.: FRP counting button with reset

The `count` behavior now depends on the reset button as well as the count button: the `update` event is merged with the `reset` event. Compared to the extended imperative example from the previous section, the dependencies are clear, the code is concise and there is little room for mistakes.

1.3. Contributions

This report describes an approach for applying FRP for graphical user interface programming.

Chapter 2 will provide background information about functional reactive programming and describe a couple of implementations and their unique properties. Building on this foundation, chapter 3 describes the main contribution of this report: *Slim*, an FRP-based domain-specific language for creating user interfaces.

Emphasis is placed on interfaces that with a dynamic structure based on the data they control and are created using composable elements. A series of examples will highlight the design choices involved to provide expressiveness for creating dynamic and complex user interfaces. Programs are constructed along the principles of the purely functional programming language Haskell; using equations without resorting to imperative commands and statements. Chapter 4 describes an implementation of this DSL.

There are already a number of existing FRP libraries that can be used to create user interfaces. Chapter 5 compares the DSL to some of the most popular ones and highlights their shortcomings in comparison to the library from chapter 3 when it comes to dynamicity, bidirectionality and composability.

Conclusions and proposals for future work are outlined in chapter 6.

2. Functional Reactive Programming

2.1. Origins

The concept of *Functional Reactive Programming* originates from the seminal paper *Functional Reactive Animation*[11] It describes a language called *Fran* which is a domain-specific language (embedded in Haskell) designed for modeling interactive animations. The key idea behind *Fran* is to provide a way of declaring values that vary over time in a continuous way, similar to how vector graphics are defined over continuous space. Such time-varying value are called *behaviors*, each of which can be thought of as a (pure) function from timestamp to value. To support reactivity, *Fran* defines *events* as values available after a certain point in time. An event can be thought of as a pair of a timestamp and a value.

Behaviors and events can depend on each other in various interesting ways (as described in the following section) to facilitate the modeling of reactive systems. For example, behaviors can switch to another value after an event occurrence, and events can sample the value of a behavior at the time they occur. By combining events and behaviors, a *network configuration* is created. Values flow through this network from various *source* events (such as button clicks) to *sink* behaviors (such as dynamically changing text labels.)

Through *higher-order FRP*, behaviors can themselves carry behaviors, to be switched between (or "flattened") so that it acts as a regular behavior. This means that the network configuration is dynamic and that it can be reconfigured based on values that flow through it.

Over the past two decades, FRP has been developed in several different directions. On the theoretical side, the formal model has been extended and redefined[10] in terms of Haskell's `Functor`, `Applicative`[14] and `Monad`[17] type classes. On the practical side, a few serious implementations problems have lead to multiple flavors of FRP with an altered API and functionality. higher-order FRP as originally formulated sometimes requires the unbounded recording of past values (leading to a space leak) and then later processing these past values (leading to a time leak.)

One class of alternative FRP formulations uses the `Arrow` type class[15] and without the combinators that provide higher-order behaviors. In place of events and behaviors, *arrowized FRP* provides 'signals' which are effectively a stream of values at discrete points in time. These signals are not exposed directly but are instead transformed via *signal functions* (functions of signal to signal), which are composed through functions provided by the `Arrow` type class and a set of combinators. The `Arrow` type class ensures a static configuration of the signal functions, which allows for an efficient implementation free of space and time leaks.

In addition, several other changes to the original FRP language have been proposed to prevent space and time leaks while preserving higher-order capabilities. *Elerea*[16] and *FRPNow!*[3] use an extra (monadic) type to 'trim' the start times of events and behaviors

and to allow past events to be forgotten. *Grapefruit*[13] and *Reactive Banana*[1] use a phantom type parameter to distinguish between start times and force the programmer to manually trim start times, again, to allow past events to be forgotten.

Finally, it should be remarked that FRP — at least in its original formulation — is a very general concept capable of constructing various sorts of reactive models, from animations to simulations and signal processors. Conal Elliott's original formulation of FRP focuses on events and behaviors defined over continuous time, with a capability to speed up and slow down behaviors as you would expect from an animation toolkit. The benefit of this is that, when consuming the values produced by a reactive model, behaviors can be sampled at arbitrary times: because events and behaviors are defined over continuous time, any sampling rate can be applied without affecting the model itself. This is similar to how vector graphics can be sampled at any resolution.

In contrast, this report looks at FRP for the purposes of user interface programming. A notion of continuous time (such a floating point value of seconds since the application started) is not very important for user interfaces: they should react to user events at discrete points in time. Other properties of FRP implementations are more important to focus on such as dynamic network configurations and declarative updating of interface elements based on events and behaviors. In addition, composability is vital when creating large and complex user interfaces.

2.2. Concepts

The cornerstone of a reactive system is the *event*: the occurrence of some message at some point in time. Reactive systems in imperative language often implement the *observer pattern*: a procedure (or *event handler*) is registered to run when a certain event occurs. This procedure can then perform side effects based on the event message.

Events in FRP are very similar but with the key difference that instead of registering handlers, events are used to derive behaviors and other events. They can be thought of as a list of tuples with (monotonically increasing) timestamps and values: `type Event a = [(Time, a)]`. For the purposes of user interface programming, the `Time` values are only interesting for determining which events have already occurred and which are in the future. For example, recall listing 1.3 from chapter 1 where `onClick countButton` produces an event carrying unit values.

In some formulations of FRP, such as *Fran*[11] and *FRP Now!*[3], events have (at most) one occurrence. This means that single-fire *events* are distinguished from *event streams*, where multiple message may be produced. In most other formulations, such as *Reactive*[10] and *Sodium*, events may have multiple occurrences. For user interface programming, where most events (such as button clicks) may occur multiple times, this distinction is not very useful. Most FRP implementations don't make this distinction, and in this thesis the multiple-occurrence variant will be used.

[1] Prior to version 0.9; starting from version 1.0 Reactive Banana uses the monadic type class `MonadMoment` instead of phantom type parameters.

While events only produce values for certain discrete points in time, behaviors can produce values for every point in time. They can be thought of as a function of time to value: `type Behavior a = Time -> a`. Alternatively, and especially in the context of user interface programming where time is discrete, behaviors can be thought of as a stream of values. This can be expressed as a pair of an initial value and an event that produces new values: `type Behavior a = (a, Event a)`. This concept is called the *stepper*[2] and is one of the core ways of constructing behaviors in FRP.

The *stepper* combinator introduces state that is only constructed from the value of the latest event occurrence. The *accumulator*[3] concept is more powerful in that it allows its state to be constructed from last event occurrence as well as the previous state. For example, `count` from listing 1.3 is a behavior that counts button clicks and is created from an initial value of 0 and an event that increases this value by one.

Accumulators express one of the core concepts of reactivity: feedback loops. These are essential in user interface programming, where user input is used to update a state, and this state is used to update the user interface. For example, consider the program from listing 1.3 with a button which has the number of times its been clicked as its label. A behavior is used to express the click counter: initialized with zero, and the click event as the event that updates the counter.

In order to use events and behaviors with the various combinators it is often necessary to manipulate the underlying values by applying a function to them. This *lifting* of functions into events and behavior is a core part of functional reactive programming. Events have a `Functor` instance allows functions to be applied to event values. In the case of the counter button this can be used to replace the event value with a function to increase the count.

Behaviors have a `Functor` and `Applicative` instance to allow lifting of functions with multiple parameters over multiple behaviors using combinators such as `liftA2`. This is a simple and intuitive way of constructing new behaviors because they always contain a value for every point in time. While in the original formulation events also have an applicative instance this is not as intuitive or useful: only when there is an event occurrence for all involved events at the same time is it possible to lift a function over them.

The combinators mentioned above describe how to create behaviors that observe events but it is also possible for events to observe behaviors. With the `snapshot` combinator each occurrence of an event has its value combined with the value of a behavior at that time: `snapshot :: Event a -> Behavior b -> Event (a, b)`.

Behaviors can also have a `Monad` instance. Higher-order FRP introduces the notion of switching behaviors and events, where, for instance, a behavior may carry a behavior within it. This inner behavior can be exposed through a combinator such as `switchB :: Behavior (Behavior a) -> Behavior a`, which flattens the nested behavior 'stream'. This combinator resembles the monadic combinator `join`, which can be used to implement a monad instance.

[2] As a combinator: `stepper :: a -> Event a -> Behavior a`

[3] One such a combinator produces a behavior: `accumB :: a -> Event (a -> a) -> Behavior a`

Higher-order FRP is where things become less intuitive as the network configuration is dynamically changing. In the next chapter it will be demonstrated that higher-order FRP leads to increased complexity and code structures that are reminiscent of imperative code.

2.3. Classical FRP: *Reactive*

Conal Elliott's *Reactive* library for Haskell is the successor to his *Fran* library and implements the idea's from his paper "Push-pull functional reactive programming"[10]. It implements largely the same language as *Fran* but it leverages the Haskell type-classes `Functor`, `Applicative`, `Monad` and `Monoid` extensively instead of implementing its own lifting combinators. *Reactive* implements a higher-order FRP language that is prone to space and time leaks: it contains combinators that may prevent the history of some events and behaviors from being garbage collected.

The paper provides denotational semantics of the FRP system which is also used to guide the implementation. Its API is purely functional with no hints of `IO` actions being used. However, `unsafePerformIO` is used for caching and spawning threads.

Behaviors are defined in terms of events while events are defined in terms of *future values*. Upon sampling these future values will block the current thread until they become available. When combining future values multiple threads are created so that they can be received as they become available. Actual `Time` values are used to determine the exact order in which events occur.

The following example will demonstrate *Reactive*'s API and its leak problem. It implements the behavior `numClicks` that counts the number of times a button is clicked, and a behavior `numClicksToggled` that either shows `Just` the amount of clicks or `Nothing`, based on an event that toggles this.

```haskell
-- provided by Reactive: this switches in new behaviors as they are produced by an event
switcher :: Behavior a -> Event (Behavior a) -> Behavior a

-- event that fires whenever the button is clicked
clicks :: Event ()

-- event that decides whether or not the amount of clicks should be shown
toggle :: Event Bool

-- behavior that accumulates the number of clicks
numClicks :: Behavior Int
numClicks = accumB 0 ((+1) <$ clicks)

-- behavior that shows Nothing, or Just the number of clicks
numClicksToggled :: Behavior (Maybe Int)
numClicksToggled = switcher (pure Nothing) (showClicks <$> toggle)
  where
    showClicks True = Just <$> numClicks
    showClicks False = pure Nothing
```

Listing 2.1.: Toggled counter button in `Reactive`

This example is problematic from an implementation standpoint because of the state that is kept by the `numClicks` behavior. Consider the following chain of events: `toggle` produces a `False` message to signal that clicks should not be shown, followed by a number of `clicks` messages. `numClicks` is not currently switched in, but may be switched in at some later point. The implementation doesn't know this however: `numClicks` is not currently reachable from the observed behavior `numClicksToggled`. This means that in order to properly count all `clicks` messages, they have to be retained so that they can later be used when `numClicks` is evaluated. In *Reactive*, this means that the past messages of `clicks` cannot be garbage collected because `numClicks` still holds references to it.

2.4. Network configuration as monadic actions: *Sodium*

Sodium[4] is a Haskell library that implements a higher-order FRP language in Haskell using the monadic `Reactive` type to specify when events and behaviors are created. Many functions involving events and behaviors are `Reactive` actions. This always makes it clear when, for example, event messages should start to be observed: the combinator to accumulate event values results in an action to create a behavior, instead of a behavior directly. Only when this action is run will the resulting behavior start observing the event. This means that it is only possible to keep track of history by explicitly accumulating it.

Sodium is implemented using `IO` and mutable references: events are essentially a form of the observer pattern, where an `IO` action can be subscribed to event messages.

[4] `https://hackage.haskell.org/package/sodium`

Behaviors consist of a mutable reference to the current value and an event that fires when this value is updated.

The following example demonstrates how *Sodium* prevents the space and time leak that is possible in Reactive:

```
-- provided by Sodium: this switches in new behaviors as they are produced by a behavior
switch :: Behavior (Behavior a) -> Reactive (Behavior a)

-- provided by Sodium: because accumulation involves state, this combinator produces an
action
accum :: a -> Event (a -> a) -> Reactive (Behavior a)

-- event that fires whenever the button is clicked
clicks :: Event ()

-- event that decides whether or not the amount of clicks should be shown
toggle :: Event Bool

-- action to create a behavior that accumulates the number of clicks
mkNumClicks :: Reactive (Behavior Int)
mkNumClicks = accum 0 ((+1) <$ clicks)

-- action to create a behavior that shows Nothing, or Just the number of clicks
mkNumClicksToggled :: Reactive (Behavior (Maybe Int))
mkNumClicksToggled = do
  toggled <- accum False (const <$> toggle)
  numClicks <- mkNumClicks
  let
    showClicks True = Just <$> numClicks
    showClicks False = pure Nothing
  switch (showClicks <$> toggled)
```

Listing 2.2.: Toggled counter button in `Sodium`

This example demonstrates how reactive network configurations are created from actions using monadic composition. These actions tell *Sodium* exactly what is going on: because `numClicks` is created only once it is clear that this behavior will be sampled in the future and that it should start its observation of `clicks`. Note that the name `numClicks` refers to that specific instantiation of `mkNumClicks` within the context of `mkNumClicksToggled`. In *Sodium* it is always clear from context when a stateful event or behavior is initialized and so it is safe to use dynamically switched events and behaviors. The downside of Sodiums `Reactive` API is that, because every stateful combinator is an action, even programs that do not involve higher-order FRP have to use monadic constructs extensively.

2.5. Start times as phantom types: *Reactive Banana*

Reactive Banana is a Haskell library that uses extra types and type parameters to keep track of when events and behaviors start. Events and behaviors have an extra

(phantom) type parameter that indicates their start time. This type parameter is bound to the context in which they are first defined, which means that it is not possible to initialize stateful events and behaviors within switching combinators. These contexts are introduced by the monadic `Moment`-type; the phantom type parameters are bound by functions that involve `Moment` actions.

In practice this means that some parts of a network configuration will be explicitly initialized by actions similar to how it's done in Sodium. An important difference is that only the parts involved in dynamic switching and side effects involve actions. Stateful combinators that do not involve dynamic switching (such as `accumB`) are pure functions. The following example will demonstrate this by implementing the same toggled click behavior as before:

```
-- provided by Reactive Banana: this explicitly initializes a behavior at a certain
moment
trimB :: Behavior t a -> Moment t (AnyMoment Behavior a)

-- provided by Reactive Banana: this switches in new behaviors that are not bound to a
context
switchB :: Behavior t a -> Event t (AnyMoment Behavior a) -> Behavior t a

-- an action to create a network configuration, based on events
networkDescription :: forall t. Event t () -> Event t Bool -> Moment t ()
networkDescription clicks toggle = do
  let
    -- a behavior that counts the number of clicks, bound to the same start time as the
    events and the network itself
    numClicks :: Behavior t Int
    numClicks = accumB 0 ((+1) <$ clicks)

  -- create a new behavior that can be used in other contexts
  numClicks' <- trimB numClicks

  let
    -- behavior that shows Nothing, or Just the number of clicks
    numClicksToggled :: Behavior t (Maybe Int)
    numClicksToggled = switchB (pure Nothing) (showClicks <$> toggle)
      where
        showClicks True = Just <$> numClicks'
        showClicks False = pure Nothing

  -- in a real program there would be output handlers here
  return ()
```

Listing 2.3.: Toggled counter button in *Reactive Banana*

The example demonstrates how the `switch` combinator's augmented type enforces explicit initialization through a `Moment` action (`trimB`.) The types of the combinators `trimB` and `switchB` are essential to *Reactive Banana*'s type system:

`trimB` converts a behavior tied to a certain context (the `t` in `Behavior t a`) into a `Moment` action to create a behavior that is not: `AnyMoment Behavior a`. The type

constructor `AnyMoment` is used in place of a quantifier for an impredicative type which, according to the author[1], should be read as `forall s. Moment s (Behavior s a)`, a type GHC could not handle well at the time. The resulting behavior has its history "trimmed" and will thus not hold on to it because it no longer depends on it.

The `switchB` can use an event that produces such trimmed behaviors to switch them in and out at any point in time; these behaviors can be freely sampled or simply discarded without worrying about past-dependence. Note that regular stateful combinators, such as `accumB`, are not actions. Non-dynamic parts of the network are composed using pure combinators.

The benefit of *Reactive Banana*'s approach is that first-order stateful combinators are pure functions, just like in *Fran* and *Reactive*, while space and time leaks are ruled out by the type system. The drawback is in the type system itself: even a simple first-order program requires care for the phantom type parameter and higher-order FRP requires conversion back and forth between `AnyMoment`s and plain `Event`s and `Behavior`s, involving monadic `Moment` actions. In fact, starting from version 1.0, its author has decided to replace the phantom types with a more monadic API similar to that of Sodium[2].

2.6. Arrowized FRP: *Yampa*

Yampa is a Haskell library that implements "Arrowized FRP", which unifies events and behaviors into a single type—*signals*—and involves constructing *signal functions* instead of reactive networks. Signals are a stream of values that change at discrete points in time. Events can be modeled as a stream of `Maybe` values, with occurrences represent as `Just` values. It does not have combinators or lifting functions for working with signals directly: signals are manipulated using *signal functions* of type `SF a b`, which can be thought of as a function of `Signal a` to `Signal b`. Signal functions are composed using the methods provided by its `Arrow` type-class instance and Haskell's `proc` syntax for arrows. Although they can create stateful signals, signal functions themselves are stateless.

In Yampa, reactive networks are modeled as a `signal function`, transforming an input signal to an output signal. The signals themselves are never explicitly initialized or trimmed, only composed as part of other signals. As a result, implementing the same example application as in the previous chapter confronts us with a choice: when the toggled click count (`NumClicksToggled`) is toggled off, should it continue observing clicks?[5] In *Reactive*, the most straight-forward implementation automatically continues observing clicks (which turned out to involve a space and time leak.) In *Sodium* and *Reactive Banana*, emulating this behavior was accomplished using explicit initialization.

Yampa, however, does not provide a switching combinator that supports this same behavior. This limitation stems from the fact that there is no way to work with signals directly and there is thus no combinator to switch signals either. Although signal

[5] Evan Czaplicki (author of Elm[7]) used a very similar example to highlight this design choice when working with higher-order FRP[6].

functions can be switched dynamically[6], because they are stateless this functionality is strictly less powerful than the switching combinators of classic higher-order FRP.

Instead, the same behavior as in the previous examples must be implemented by combining both the click counting signal and the toggled signal into one:

```
-- provided by Yampa, this function creates a stepper signal function
hold :: a -> SF (Event a) a

-- provided by Yampa, this function creates an accumulator signal function
accumHold :: a -> SF (Event (a -> a)) a

-- signal function that counts the number of occurrences of the input event (ie. clicks)
numClicksSF :: SF (Event ()) Int
numClicksSF = proc clicks -> do
  accumHold 0 -< (+1) <$ clicks

-- signal function with two input events: clicks and toggle, the output is the switched
amount of clicks
numClicksToggledSF :: SF (Event (), Event Bool) (Maybe Int)
numClicksToggledSF = proc (clicks, toggle) -> do
  numClicks <- numClicksSF -< clicks
  showClicks <- hold False -< toggle
  returnA -<
    if showClicks
      then Just numClicks
      else Nothing
```

Listing 2.4.: Toggled counter button in *Yampa*

`numClickSF` is a signal function that takes a signal of click events and produces a signal that counts the clicks using the `accumHold` signal function, which creates an `accumulating` signal. `numClicksToggledSF` combines it with a `hold` signal function, which creates a `stepper` signal, to produce a signal that can choose its value between either two. Essentially, `numClicksToggledSF` creates a signal with as state a tuple of the amount of clicks and a boolean that decides whether or not to show it.

Through its `Arrow`-based API, Yampa makes it hard to create space and time leaks. However, because of its focus on signal functions instead of signals, the developer is forced to think about interaction between signals in a different, more indirect way compared with classical FRP.

In addition, because signal functions are limited to just one input and output type, various different input events and output actions must be combined. This leads to increased complexity in larger applications.

[6] Through combinators such as `rSwitch :: SF a b -> SF (a, Event (SF a b)) b`, which switches to a signal function provided by an event produced from another signal function.

2.7. First-order signals: *Elm*

Elm is a Haskell-inspired language designed for creating user interfaces using FRP[7]. Its compiler targets Javascript and HTML. Similar to Arrowized FRP, events and behaviors are unified as signals. However, signals can be worked with directly in a manner similar to "classic" FRP. Such signals are the only way to provide interactivity: Elm does not have anything similar to Haskell's `IO` type for controlling side effects.

Elm's FRP system is limited to first-order signals and so there are no switching combinators of any kind. This limitation is by-design: no complicated phantom types or monadic constructs are necessary for programming Elm and its programs are straightforward. In fact, Elm encourages developers to embrace this limitation: there is a standard architecture to be used for Elm applications[8]. It involves a single accumulator that processes an input signal carrying commands that describe how the single piece of state must be updated. This state is then used to produce an interface element structure which contains event sources.

The following example describes how a toggled button can be expressed in Elm using its first-order FRP combinators.

```
-- signal that pulses whenever the button is clicked
clicks : Signal ()

-- signal that decides whether or not the amount of clicks should be shown
toggle : Signal Bool

-- signal that accumulates the number of clicks
numClicks : Signal Int
numClicks =
  let
    plus1 () x = x + 1
  in
    foldp plus1 0 clicks

-- behavior that shows Nothing, or Just the number of clicks
numClicksToggled : Signal (Maybe Int)
numClicksToggled =
  let
    showClicks b x = if b then Just x else Nothing
  in
    map2 showClicks toggle numClicks
```

Listing 2.5.: Toggled counter button in *Elm*

Elm uses different names and combinators from the other libraries; `foldp` ("fold past") is a type of accumulator, `map` is `fmap`, and `map2` is its version of Haskell's `Applicative` combinator `liftA2`. Similar to the Arrowized FRP implementation from the previous section no switching combinator is available to implement the desired functionality.

[7] This section describes Elm as of version 0.16. Elm version 0.17 removed the `Signal` type and FRP combinators and now provides only an accumulating combinator to be used at the top level (`main`.)

Instead, the `numClicksToggled` signal chooses its value by lifting a function over the `numClicks` and `toggle` signals. Note that all signals in Elm must be defined as top-level bindings; Elm's strict semantics takes care of efficient and leak-free run-time behavior.

Although Elm programs are straight-forward and involve no scary types or exotic syntactical constructs, the limitation to first-order FRP is very apparent when developing more complex user interfaces. The Elm Architecture provides guidelines for working around this limitation, with a few caveats. Chapter 5 provides a more in-depth analysis of this issue.

3. Designing *Slim*

3.1. Introduction

This chapter describes a series of domain-specific languages for creating user interfaces in Haskell. The languages are of increasing expressiveness and are described alongside example user interfaces of increasing dynamicity, ranging from a static label to a dynamic list of buttons. The final language, *Slim*, provides a safe and purely functional way to develop user interfaces in Haskell. Its implementation is described in chapter 4.

3.2. *Hello world*: static components

```
span :: String -> Component
staticRoot :: Component -> IO ()
```

Listing 3.1.: Functions used to implement the listing 3.2

The `Component` data type is used to define static structures of user interface elements. Due to its ubiquity and ease of use, the user interfaces in this chapter (and in the rest of this thesis) will be described in terms of HTML. Accordingly, each value of `Component` defines an HTML structure, consisting of an HTML element and zero or more child elements.

This component structure is sufficient for the most basic example of a user interface: a `<span>` element with the text "Hello, World!" (listing 3.2.) This component is rendered as the root component of the user interface by the `staticRoot` action.

```
runHelloWorld :: IO ()
runHelloWorld = staticRoot helloWorld

helloWorld :: Component
helloWorld = span "Hello, World!"
```

Listing 3.2.: Example: Hello world

3.3. Counting clicks: values and feedback loops

```
button :: String -> Component ()
statefulRoot :: a -> (a -> Component a) -> IO ()
instance Functor Component
```

Listing 3.3.: Functions used to implement listing 3.4

The next step up in dynamicity is a single feedback loop: a button with as its label the number of times it has been clicked. The definition of its component structure relies

on recursion: every time it is clicked the button produces an event. This event is used to accumulate the amount of clicks and the button is rendered based on this state.

The type of values of this event are expressed by the type parameter of `Component`, which in the case of `button`'s click event is the unit type as it carries no information. `Component`'s instance of the `Functor` typeclass allows the event values to be updated or replaced.

This accumulator structure can be defined using a function that somewhat resembles a fix-point combinator: `statefulRoot`. This combinator accumulates a value based on an initial state, and a function from a state to a component. The resulting component produces an event with an updated state value. This component is then rendered as the root component of the user interface.

In the following example (listing 3.4) the count is initialized to 0 and the component is a button which shows the current count. Using `Component`'s `Functor` instance, more specifically the `<$` combinator, the unit value produced by the button's event is replaced by the amount of clicks.

```
runCounter :: IO ()
runCounter = statefulRoot 0 counter

counter :: Int -> Component Int
counter x = x + 1 <$ button (show x)
```

Listing 3.4.: Example: counting clicks

At first glance it would seem that this program completely replaces the button on every click. The component renderer[1], however, keeps track of any previously rendered components and only updates the elements and attributes that have changed. This is done by first rendering components as an intermediate HTML tree structure, the *Virtual DOM*[2]. This intermediate structure is then compared to the version from the previous render, and only the differences are applied to the actual DOM. This technique was pioneered by Facebook's *React* library for Javascript[4].

In this case, when the button is clicked, only the body text of the button is replaced. This allows changes in state to be reflected by the user interface without explicitly defining when and how these changes should be made. This is a key property of the `Component` type that, when combined with FRP, allows for dynamic component structures to be defined. This will be demonstrated in a later example.

3.4. Resettable counter: basic FRP and initialization

```
div :: [Component a] -> Component a
```

Listing 3.5.: Functions used to implement listing 3.6

[1] For the precise mechanism, please refer to the implementation chapter.

[2] DOM stands for *Document Object Model*, the programmatic model for working with HTML structures in web browsers.

The following example involves two buttons: each counts the number of clicks but button #2 is reset to zero whenever button #1 is clicked. This can be implemented in at least two ways: using a single piece of state for both values and using separate states. The version with the single piece of state can be implemented using (almost) entirely the same functions as those used in the previous example.

Storing the counts of both buttons in the same state also means that there is a single feedback loop: clicks on either button are merged into a single event stream. This merging is done by the `div` function, which produces an HTML `<div>` element and serves as a container element for the two buttons. Each button produces an event with counts of both buttons: button #1 increases its own count while resetting the other to zero, while button #2 only increases its own value and leaves the other the same.

```
runCountPair :: IO ()
runCountPair = statefulRoot (0, 0) countPair

countPair :: (Int, Int) -> Component (Int, Int)
countPair (x,y) = div [button1, button2]
  where
    button1 = (x + 1, 0) <$ button (show x)
    button2 = (x, y + 1) <$ button (show y)
```

Listing 3.6.: Example: resettable counter using one feedback loop

Although this example looks simple and demonstrates that the functions used in listing 3.4 are powerful enough for this use case, it also illustrates a limitation of the single feedback loop: the counts of each button (bound to `x` and `y`) are not defined in a single place. Both counts are defined by both buttons: button #1 is concerned not only with its own count but is also with resetting the count of button #2, and button #2 is concerned with the count of button #1 even though it has no business changing it.

```
getEvent :: Component a -> Event a
merge :: Event a -> Event a -> Event a
stepper :: a -> Event a -> Behavior a
mount :: Behavior (Component a) -> Component a
instance Functor Event
instance Functor Behavior
instance Applicative Behavior
```

Listing 3.7.: Functions used to implement listing 3.8

Instead, the following example uses two separate feedback loops, where button #1 is defined based only on itself and button #2 is defined based on both. For this, several new concepts will be introduced:

- The event stream produced by button #1 will be accessed so that it can be used for defining button #2.
- Button #2's count depends on multiple event streams: its own and that of button #1. These event streams will be combined and used for accumulating state.

- Button #1 will be referenced in multiple places: in defining button #2 and in the root component structure (the container element.)

The following example will show how basic FRP constructs can be used to define the two feedback loops and their state. The `getEvent` function is used to produce the event streams from two counter components which are then used to define stateful behaviors using `stepper`. The resulting behaviors are used to define the counter components as type `Behavior (Component Int)` (time-varying components, with an event stream of `Int` values). These counter components are then `mount`ed in the container `div` element.

```
runTwoCounters :: IO ()
runTwoCounters = staticRoot twoCounters

twoCounters :: Component Int
twoCounters =
  let
    bCount1 = stepper 0 (getEvent button1)
    bCount2 = stepper 0 (merge (0 <$ getEvent button1) (getEvent button2))
    button1 = mount (counter <$> bCount1)
    button2 = mount (counter <$> bCount2)
  in
    div [button1, button2]

counter :: Int -> Component Int
counter x = x + 1 <$ button (show x)
```

Listing 3.8.: Example: resettable counter using two feedback loops

This example contains a serious problem, however: `button1` and `button2` are defined as pure values, while used in multiple places. This causes a problem for the calls to the `getEvent` and `mount` functions: both functions should refer to the same actual component as it is rendered on the screen, while this component does not have any real identity. Referential transparency dictates that `button1` must be equal to its body expression, so what should happen if this expression is used to add it to the container element? The root of the problem is that the `let`-bindings provide a name for expressions but not an *identity*.

```
runStartRoot :: Start (Component Dynamic void) -> IO ()
startC :: Behavior (Component Static a) -> Start (Component Dynamic a)
getEvent :: Component Dynamic a -> Event a
mount :: Component Dynamic a -> Component Static a
instance Functor Start
instance Applicative Start
instance Monad Start
instance MonadFix Start
```

Listing 3.9.: Functions used to implement listing 3.10

The `Start` monad provides a way to create an identity for components as they are added to the user interface. This type only deals with initialization of components

and behaviors: it marks the point in time at which components and behaviors are 'booted up' in order to be used by subsequent `getEvent` and `mount` calls. By using the `startC` action to produce initialized components, the following example specifies exactly how each button should be shared between the various behaviors and the container component. The `Start` monad is ultimately run when the root component is rendered by the `runStartRoot` function.

With the introduction of initialized components it also becomes important to tell them apart from regular, static components. Components that have not been initialized yet, produced by functions such as `button` and `div` are now marked `Static` while initialized component, produced from a behavior of static components, are marked `Dynamic`.

```
runResettableCounter :: IO ()
runResettableCounter = runStartRoot resettableCounter

resettableCounter :: Start (Component Dynamic Int)
resettableCounter = mdo
  let
    bCount1 = stepper 0 (getEvent button1)
    bCount2 = stepper 0 (merge (0 <$ getEvent button1) (getEvent button2))
  button1 <- startC (counter <$> bCount1)
  button2 <- startC (counter <$> bCount2)
  startC (pure (div [mount button1, mount button2]))

counter :: Int -> Component Static Int
counter x = x + 1 <$ button (show x)
```

Listing 3.10.: Example: resettable counter using two feedback loops and sharing

The `Start` monad is used to explicitly initialize each button, after which they can be used as an event source. Button #1 is referred to three times: in defining its own behavior, button #2's behavior, and in producing the root component. If instead this definition would be based on an (uninitialized) `Component` value, its value would not contain any uniquely identifying information and its meaning in the context of the user interface would be unclear. By using the monadic bind for initializing the button components, they can then be used as an event source and child component without ambiguity.

Recursive-do notation (based on the `MonadFix` instance) is used to define the buttons based on their own events. Recursive monadic expressions can easily lead to "black holes", where a computation depends (possibly indirectly) on its own result and becomes stuck. The `getEvent` function, however, is implemented in such a way that it is always safe to use in computations that rely on its result. By making sure event subscription is delayed until after the component is fully defined it is safe to use `getEvent` on initialized components that are defined later. (This does not hold for the `mount` function but use of this function in a recursive way would mean mounting a component inside itself which raises philosophical issues.)

The functions introduced for the last example can be used to implement a couple of the functions used previously:

```
staticRoot :: Component Static a -> IO ()
staticRoot c = runStartRoot (startC (pure c))

statefulRoot :: a -> (a -> Component Static a) -> IO ()
statefulRoot init mkComponent = runStartRoot $ mdo
  dynComponent <- startC $ mkComponent <$> stepper init (getEvent dynComponent)
  return dynComponent
```

Listing 3.11.: Implementations of `staticRoot` and `statefulRoot`

3.5. Dynamic list of counters: initialization on demand

```
silence :: Component Static a -> Component Static void
track :: Eq k => Behavior [k] -> (k -> Start (Component Dynamic a)) -> Behavior
[Component Dynamic a]
```

Listing 3.12.: Functions used to implement listing 3.13

The previous example shows how to implement an interface with multiple, separate feedback loops, each with their own piece of state. The following example consists of a list of counter buttons, a button to add a counter to the end of the list and a button to remove the last counter from the list. While this could be implemented using the same constructs as the previous example, this would lead to a data flow that is probably less than ideal. The states of the all counter buttons in the list must be combined in a single value and feedback loop because, until now, feedback loops could not be added and removed at run-time.

First off, the `silence` function can be used to suppress the events from a component that you are not interesting in. This is useful when the events from a component are already observed separately, and shouldn't interfere with the events produced by the structure it is mounted in.

The `track` function can be used to implement a dynamic list of initialized components. Based on a behavior with a list of keys and a function to create a new component based on such a key a behavior is created that starts components for every key currently in the list. Component are only initialized once for any given key, and then stored for when they may reappear later.

This limited form of dynamic switching is powerful enough for user interfaces but limited enough to not require initialization itself. Tracking may start at any point after the behavior of keys has been initialized; the result will be the same because it can "catch up" by initializing components for all keys right away.

```
runDynamicCounters :: IO ()
runDynamicCounters = runStartRoot dynamicCounters

dynamicCounters :: Start (Component Dynamic Int)
dynamicCounters = mdo
  let
    bAmount = stepper 0 (getEvent controls)
    bKeys = (\n -> [1..n]) <$> bAmount
  controls <- startC (mkControls <$> bAmount)
  buttons <- startC (div . map mount <$> track bKeys mkCounter)
  startC (pure (div [silence (mount controls), silence (mount buttons)]))

mkCounter :: Int -> Start (Component Dynamic Int)
mkCounter k = mdo
  button <- startC (counter <$> stepper 0 (getEvent button))
  return button

counter :: Int -> Component Static Int
counter x = x + 1 <$ button (show x)

mkControls :: Int -> Component Static Int
mkControls x =
  div
    [ min 10 (x + 1) <$ button "+"
    , max 0 (x - 1) <$ button "-"
    ]
```

Listing 3.13.: Example: initialization on demand

3.6. Dynamic list of counters with shared state: initialized behaviors

```
accumB :: a -> Event (a -> a) -> Behavior a
startB :: Behavior a -> Start (Behavior a)
```

Listing 3.14.: Functions used to implement listing 3.15

The example in this section is much like the previous one, with one extension: the counter buttons in the list also track the clicks on all buttons, in addition to just their own, and show both counts separately. There is one important detail here that requires a new function for the implementation to work as expected: explicit initialization of behaviors.

In the previous examples, all behaviors were initialized together with the component that uses them. In this example, several initialized components (the counter buttons) should share a behavior: the total amount of clicks. Behaviors are defined as pure values in `let`-bindings; if care is not taken to initialize this shared behavior, a problem similar to the one of example 5 will appear: when referencing the same behavior from multiple places, how should this behavior be shared?

This behavior, `bTotal`, should start counting before the first counter button appears, and be used to define all counter buttons. In the following example, this behavior is therefore explicitly initialized right before the root component, using the `startB` function. This initialized behavior is then shared by all the counter buttons, regardless of when they are initialized themselves.

```
runDynamicCounters' :: IO ()
runDynamicCounters' = runStartRoot dynamicCounters'

dynamicCounters' :: Start (Component Dynamic Int)
dynamicCounters' = mdo
  let
    bAmount = stepper 0 (getEvent controls)
    bKeys = (\n -> [1..n]) <$> bAmount
  controls <- startC (mkControls <$> bAmount)
  bTotal <- startB (accumB 0 ((+1) <$ getEvent buttons))
  buttons <- startC (div . map mount <$> track bKeys (mkCounter' bTotal))
  startC (pure (div [silence (mount controls), silence (mount buttons)]))

mkCounter' :: Behavior Int -> Int -> Start (Component Dynamic Int)
mkCounter' bTotal k = mdo
  let bCount = stepper 0 (getEvent comp)
  comp <- startC (counter' <$> bCount <*> bTotal)
  return comp

counter' :: Int -> Int -> Component Static Int
counter' x y = x + 1 <$ button (show (x,y))
```

Listing 3.15.: Example: initialization on demand with a shared behavior

Consider what would happen if this behavior was not initialized at the start but instead left uninitialized. On initializing each counter button it would then initialize its own version of the behavior instead of sharing it. As a result, buttons added after one or more clicks are made would start from zero as they will have missed the click events that occurred before their initialization. The `startB` functions mark the start time of behaviors, and the `Start` monad is used to keep track when each behavior is initialized.

Start times are an import aspect of FRP networks because they are necessary to prevent ambiguity and space/time leaks. Consider a behavior that keeps track of an event stream defined by a parent component, initialized earlier. Should the behavior observe events from before its initialization? If so, this requires keeping track of all event occurrences, should they ever be required by some future component. If not, this requires some specification of when exactly this event stream will start to be observed.

3.7. Enforcing initialization with types

```
runStartRoot :: (forall s. Start s (Component (Dynamic s) a)) -> IO ()
track
  :: Eq k
  => Behavior (Local t) [k]
  -> (k -> (forall s. Start s (Component (Dynamic s) a)))
  -> Behavior (Local t) [Component (Dynamic t) a]
startC
  :: Behavior (Local t) (Component Static a)
  -> Start t (Component (Dynamic t) a)
startB :: Behavior (Local t) a -> Start t (Behavior Shared a)
getEvent :: Component (Dynamic t) a -> Event (Local t) a
useB :: Behavior Shared a -> Behavior (Local t) a
```

Listing 3.16.: Functions used to implement listing 3.17

The previous section demonstrated explicit initialization of behaviors to allow shared state between component 'scopes' (calls to `startC`.) The important difference between such scopes is that of their start times: in listing 3.15, the root component scope (`component`) starts when the application first runs, while its child component scopes (`mkCounter`) start based on events. The example in this section shows how Haskell's type system can be used to prevent events and behaviors from 'leaking' into scopes where they would produce unpredictable results. More precisely, the extended types used by the example enforce the following invariants:

- Initialization done by `track` may involve only *uninitialized* events and behaviors, which will be initialized at the start time specified by `track`, and previously *initialized* events and behaviors that can be shared. In other words: events and behaviors that are not explicitly initialized may not escape the scope in which they are defined.
- Dynamic components that are initialized by another scope may not be used as event source or sub-component. In other words: initialized (dynamic) components may not escape the scope in which they are initialized.

This is achieved by introducing a new type variable t on `Component`, `Event`, `Behavior`, and `Start`, which indicates their scope:

- Static components are not limited in scope: `Component Static a`
- Dynamic components are limited in scope t: `Component (Dynamic t) a`
- Local behaviors (and events) are limited in scope t: `Behavior (Local t) a`
- Shared behaviors (and events) are not limited in scope: `Behavior Shared a`

This type variable is bound by the universal quantifier in the types of `runStartRoot` and `track`. Essentially, this means that the type variable cannot leak between scopes: a

value tagged with type variable t cannot be part of two different function calls to either `runStartRoot` or `track`. This is very similar to (and inspired by) the `ST` type of the Haskell standard library, where a type variable is used to ensure references to mutable variables do not escape the context in which they are defined.

listing 3.17 implements a simple counter button using the `track` combinator. The behavior that is tracked is a list with always just one (unit) element, so there is always just one button, but it is initialized by `track` and is thus in its own scope.

```
runTrackedCounter :: IO ()
runTrackedCounter = runStartRoot trackedCounter

trackedCounter :: (forall s. Start s (Component (Dynamic s) Int))
trackedCounter = mdo
  let
    bKeys = pure [()]
    bButton = track bKeys (mkCounter bCount)
  bCount <- startB (stepper 0 (getEvent button))
  button <- startC (div . map mount <$> bButton)
  return button

mkCounter :: Behavior Shared Int -> () -> Start t (Component (Dynamic t) Int)
mkCounter bCount () = startC (counter <$> useB bCount)

counter :: Int -> Component Static Int
counter x = x + 1 <$ button (show x)
```

Listing 3.17.: Example: explicit initialization enforced by the type system

Note the type of `mkCounter`: its first argument is a shared behavior. listing 3.18 shows what would happen if, instead, this would be a local (uninitialized) behavior. This produces a type error in the definition of `bButton`: the t parameter from the local behavior `bCount` is bound to the initialization action, as you can see from the type of `mkCounter`, while it should be unbound. The initialization scope must be independent: this is enforced by the universal quantifier on the second argument of `track`.

```
trackedCounter' :: (forall s. Start s (Component (Dynamic s) Int))
trackedCounter' = mdo
  let
    bKeys = pure [()]
    bButton = track bKeys (mkCounter' bCount)
    bCount = stepper 0 (getEvent button)
  button <- startC (div . map mount <$> bButton)
  return button

mkCounter' :: Behavior (Local t) Int -> () -> Start t (Component (Dynamic t) Int)
mkCounter' bCount () = startC (counter <$> bCount)
```

Listing 3.18.: Example: `track` used with a local behavior, which causes a type error

4. Implementing *Slim*

4.1. Overview

The implementation of *Slim* consists of the following parts:

- The `Component` type with its smart constructors and rendering functions. This part of the system involves the definition of components, element definitions, and event subscriptions.
- The `Start` data type and associated functions that are responsible for running components, combining the FRP system and Virtual DOM.
- An FRP system with events and behaviors with a distinction between *local* and *shared* events and behaviors (as described in chapter 3.) This system does not expose IO actions but does allow for recursive definitions with no restrictions. In addition, it provides a limited form of higher-order FRP through the `track` combinator.
- An adapter that binds to a traditional, imperative graphical user interface library. This system provides handles input from the user interface by firing the *master DOM event* and handling the actions produced in response to that event.

The code in this chapter makes use of the following GHC extensions:

- `GADTs` for distinguishing between `Component` types and supporting child events of a different type as the parent component.
- `EmptyDataDecls` for the "tag" types `Local`, `Shared`, `Static` and `Dynamic`, which serve only as type parameter but don't have members.
- `FlexibleInstances` for defining class instances for `Event`, `Behavior` and `Component` in combination with specific tag types.
- `RecursiveDo` for recursive `do`-syntax (`mdo`.)
- `RankNTypes` for defining the `StartComponent` wrapper type.
- `GeneralizedNewtypeDeriving` for deriving instances for the wrapper type `Start`.
- `RecordWildCards` for wild card pattern match and update syntax for record fields. For example, when matching a component as `StaticComponent { .. }` all record fields are in scope as values. As an expression, `StaticComponent { .. }` constructs a component with all appropriately named bindings in scope used as arguments to the constructor.

The implementation relies on only a few basic dependences, namely *base* (for the prelude), *containers* (for `Map` and `Set`), *mtl* (for *reader-writer-state* monads) and *pretty* for pretty-printing HTML documents from the simulator adapter.

Some of the implementation code is omitted from this chapter and combination instead be found in appendix A. The full implementation can also be found on Hackage[1] and GitHub[2].

4.2. Components

The `Component` type represent an interface element, optionally with child components. It is defined as a GADT with records to distinguish between the different constructors based on the type parameter `t`. There are three variants:

- Static components represent the state of interface elements at a single point in time. They have an `ElementDefinition` describing a DOM node, an `EventRouter` that describes how they should produce events and a list of (static) child components.
- Dynamic components represent interface elements that vary over time. (Recall from chapter 3 that they are created from a behavior of static components.) They have a unique `ComponentId` (used in the `ComponentRegistry` described later on) and an event. This event, constructed using an `EventRouter`, is used to produce values based on the component's interface elements or subcomponents.
- Mounted components represent dynamic components that are part of another component. They are constructed using the smart constructor `mount` and simply wrap a dynamic component so that it can be used as a static component.

`Component` has a `Functor` instance which maps a function over the events produced by the component. This is done using the `Functor` instances of `EventRouter` and `Event`, both of which are described in section 4.4.

[1] `https://hackage.haskell.org/package/slim`
[2] `https://github.com/jvdp/slim`

```
data Dynamic t
data Static

data Component t a where
  StaticComponent ::
    { c_elementDefinition :: ElementDefinition
    , c_eventRouter :: EventRouter b a
    , c_children :: [Component Static b]
    } -> Component Static a
  DynamicComponent ::
    { c_id :: ComponentId
    , c_event :: Event Shared a
    } -> Component (Dynamic t) a
  MountedComponent ::
    { c_component :: Component (Dynamic t) a
    } -> Component Static a

instance Functor (Component t)

mount :: Component (Dynamic t) a -> Component Static a
mount = MountedComponent
```

Listing 4.1.: Component type definition

The `ElementDefinition` type represents the "virtual DOM" and describes an HTML element: it has a (optional) namespace[3], tag name, list of attributes and an (optional) text body. In addition it defines a list of events that the element should produce. Note that there is no parameter for event types: event data is always of type `EventData` (which is an alias for `String`.)

```
data ElementDefinition = ElementDefinition
  { ed_namespace :: Namespace
  , ed_tagName :: TagName
  , ed_attributes :: [(AttributeName, AttributeValue)]
  , ed_text :: Maybe String
  , ed_eventSources :: [EventName]
  }

type Namespace = Maybe String
type TagName = String
type AttributeName = String
type AttributeValue = String
type EventName = String
```

Listing 4.2.: Component type definition

A component's event depends on two things: the events defined on the DOM element it represents (such as the "click" event for a button component) and events of child components (such as those of a `div` container element.) This configuration is defined using the `EventRouter` type which consists of a set of functions to produce events based

[3]Namespaces are necessary in HTML when embedding XML fragments such as SVG.

on the DOM event and child events. It has a `Functor` instance and support multiple event subscriptions for the same element, in addition to child events.

Child components' events are mapped to the result type and then merged together with the DOM events. This is done by the `mkEvent` function.

```
type EventData = String

data EventRouter a b = EventRouter
  { er_dom :: ElementId -> Event Shared b
  , er_sub :: [Event Shared a] -> Event Shared b
  }

instance Functor (EventRouter a)

mkEvent :: EventRouter a b -> ElementId -> [Event Shared a] -> Event Shared b
```

Listing 4.3.: Event routing

`EventRouter`s are constructed using one of two smart constructors: `nullRouter` discards child events and does not produce a DOM event, while `childRouter` merges child events. Subscription to DOM events is done using the `addEvent` function which updates the `er_dom` field by merging in an event that subscribes to specified `EventName`, and adding the event name to the the `ed_eventSources` field of the element definition.

Component events can be suppressed using the `silence` function. This is useful when mounting an initialized component in a container element or when discarding the events a static component was initialized with, such as the `click` event for a `button` component.

```
nullRouter :: EventRouter a b
childRouter :: EventRouter a a

addEvent :: EventName -> (EventData -> a) -> Component Static a -> Component Static a

silence :: Component Static void -> Component Static a
```

Listing 4.4.: Creating and updating event routers

Static components are constructed with one of the following three smart constructors:

- `containerComponent` constructs a component with child components, routing their events using `childRouter`. Used for container elements such as `<div>`.
- `textComponent` constructs a component with a text body instead of child components. Used for text elements such as `<span>`
- `emptyComponent` constructs a component with no text body or child components. Used for empty elements such as `<input>`.

```
containerComponent
  :: Maybe String -> String -> [(String, String)] -> [Component Static a]
  -> Component Static a

textComponent
  :: Maybe String -> String -> [(String, String)] -> String
  -> Component Static Void

emptyComponent
  :: Maybe String -> String -> [(String, String)]
  -> Component Static Void
```

Listing 4.5.: Component smart constructors

Whenever a component is first initialized a `RenderedComponent` is created. This type has fields for the component that was rendered, its unique element identifier and any rendered child components. Next, when the component is updated (in response to events), this rendered component is compared with the updated component structure. This process, called *reconciliation*, is performed using the monadic `Reconciliation` type which is based on `RWS` and consists of:

- A *reader* of `ComponentId -> ElementId` to lookup element identifiers for components. This is necessary when adding a mounted component as child element.
- A *writer* of `ElementAction` to instruct the GUI library adapter layer on which actions to perform. These actions cover all possible HTML element mutations necessary to create and update a user interface. Since the reconciliation system only refers to elements by their identifiers the adapter layer is tasked with keeping track of the actual elements.
- A *state* of `ElementId` to produce fresh element identifiers for when new elements are created.

```
data RenderedComponent where
  RenderedComponent ::
    { rc_component :: Component t a
    , rc_id :: ElementId
    , rc_children :: [RenderedComponent]
    } -> RenderedComponent

type ElementId = Int

data ElementAction
  = Create ElementId Namespace TagName
  | Replace ElementId ElementId
  | Destroy ElementId
  | SetAttribute ElementId AttributeName AttributeValue
  | UnsetAttribute ElementId AttributeName
  | SetText ElementId (Maybe String)
  | AddChildren ElementId [ElementId]
  | Subscribe ElementId EventName
  | Unsubscribe ElementId EventName

type ComponentId = Int

type Reconciliation a = RWS (ComponentId -> ElementId) [ElementAction] ElementId a
```

Listing 4.6.: Rendered components and reconciliation types

The initial rendering of a component is done by the `firstRender` function. It produces a `RenderedComponent`, used for the next reconciliation, and an `Event`, created using the event router described earlier. Only rendering a static component requires work to be done; mounted components and the dynamic components within them have already been rendered previously and can be referred to by their element identifier.

```
firstRender :: Component t a -> Reconciliation (RenderedComponent, Event Shared a)

reconcile
  :: RenderedComponent -> Component t a
  -> Reconciliation (RenderedComponent, Event Shared a)

reconcileChildren
  :: ElementId -> [RenderedComponent] -> [Component t a]
  -> Reconciliation [(RenderedComponent, Event Shared a)]

updateElement :: RenderedComponent -> ElementDefinition -> Reconciliation ()

terminate :: RenderedComponent -> Reconciliation ()
```

Listing 4.7.: Rendering and reconciliation

The `reconcile` function is responsible for updating existing interface structures. It compares a previously rendered component with the next iteration. When both are determined to be compatible (based on namespace and tag name) the attributes are added,

updated and removed where necessary and child elements are compared recursively. If they are incompatible the element is removed and recreated using `firstRender`.

Reconciliation of a component's child elements is done by pairwise comparison of the previously rendered list of child components and the new list of child components. If they were previously not rendered they are created. If they are missing from the new list they are removed. If they they are both present they are passed to the `reconcile` function.

Finally, `updateElement` works by checking changes in the element definition's fields and writing the appropriate element actions where they differ. The `terminate` function removes a rendered component by writing the destroy action for the element and all of its child components recursively.

4.3. The `Start` type

The monadic `Start` type is used to initialize components and provide them with an identity (concretely: `ComponentId`.) It is defined in terms of the `Execution` type which provides an environment in which components are initialized, updated and rendered, using an `RWST` monad transformer stack:

- A *reader* of `MasterDomEvent` to provide access to the DOM events that various elements are subscribed to.
- A *writer* of `ElementAction` to instruct the GUI library adapter layer on which actions to perform. These actions are produced by `Reconciliation` actions described in the previous section.
- A *state* of `(ComponentRegistry, ComponentId, ElementId)` to keep track of active components and produce fresh element and component identifiers.
- A base type of `IO` which is used for mutable references.

```
type ComponentRegistry = Map ComponentId ElementId
type MasterDomEvent = Event Shared DomEventInfo
type DomEventInfo = (ElementId, EventName, EventData)
type Execution =
  RWST MasterDomEvent [ElementAction] (ComponentRegistry, ComponentId, ElementId) IO
```

Listing 4.8.: `Execution` type definition

In order to support recursive definitions execution should in some cases be deferred when initializing a component. For this reason `Start` uses the `Deferred` monad transformer which provides delayed execution by using a writer monad that gathers up actions to be run later. The `runDeferred` function is used to run `Deferred` actions; it first runs the specified action and then recursively runs any actions that were deferred.

```haskell
newtype Deferred m a = Deferred { unDeferred :: WriterT [Deferred m ()] m a }
  deriving (Functor, Applicative, Monad, MonadFix, MonadIO)

instance MonadTrans Deferred where
  lift x = Deferred (lift x)

defer :: Monad m => Deferred m () -> Deferred m ()
defer m = Deferred (tell [m])

runDeferred :: Monad m => Deferred m a -> m a
runDeferred m = do
  (x, ms) <- runWriterT (unDeferred m)
  unless (null ms) (runDeferred (sequence_ ms))
  return x
```

Listing 4.9.: Deferred type definition

The `Start` type is a wrapper around `Deferred Execution` actions, without the `MonadIO` instance to prevent library users from running arbitrary `IO` actions. As described in section 3.7 it has a phantom type parameter `t` to distinguish between scopes of different `startC` and `track` actions.

The `StartComponent` type provides a shorthand for a `Start` action that produces a dynamic component. This type can be used to prevent having to specify the universal quantifier for its scope.

```haskell
newtype Start t a = Start { unStart :: Deferred Execution a }
  deriving (Functor, Applicative, Monad, MonadFix)

newtype StartComponent a = StartComponent (forall s. Start s (Component (Dynamic s) a))

executeStart :: Start t a -> Execution a
executeStart = runDeferred . unStart

executeReconciliation :: Reconciliation a -> Execution a
```

Listing 4.10.: Start type definition

Component rendering and reconciliation (`Reconciliation` actions) are run as an `Execution` action using the fuction `executeReconciliation`. This function keeps track of fresh element identifiers and element actions and uses the component registry to provide the function that looks up element identifiers by component identifier.

Components, reconciliation and deferred execution come together in the `startC` function. It uses an event proxy (explained in the next section) to allow the component to be used immediately while further setup is deferred. Effectively, initialization of the FRP network is done in multiple phases:

- First, events are initialized, ready for subscription by behaviors.
- Next, behaviors are initialized by setting up `IORef`s and subscribing to events.

- Finally, the FRP network is subscribed to by the DSL layer and used to render components.

Similar to `startC`, the functions `startE` and `startB` can be used to initialize events and behaviors respectively. These are defined using `executeE` and `executeB` which are described in the following section. Last but not least is `track` which unwraps `StartComponent` and sets up a new behavior and subscribes to the behavior passed to it. As that behavior produces new keys, new components are started and tracked based on these keys.

```
startC :: Behavior (Local t) (Component Static a) -> Start t (Component (Dynamic t) a)

startB :: Behavior (Local t) a -> Start t (Behavior Shared a)

startE :: Event (Local t) a -> Start t (Event Shared a)

track
  :: Eq k => Behavior (Local t) [k] -> (k -> StartComponent a)
  -> Behavior (Local t) [Component (Dynamic t) a]
```

Listing 4.11.: Starting components, events and behaviors

4.4. Events and behaviors in IO: the FRP system

The FRP system is built up around the `Event` type. There are two constructors:

- Local events are actions that create shared events.
- Shared events implement the "observer pattern": a handler action can be subscribed which will receive the event value when the event fires. The action of subscribing to this event produces an action that unsubscribes the handler.

The `Functor` instance for events applies the mapping function to the value received by the handler action.

```
data Local t
data Shared

data Event t a where
  LocalE ::
    { e_run :: Execution (Event Shared a)
    } -> Event (Local t) a
  SharedE ::
    { e_subscribe :: (a -> Execution ()) -> Execution (Execution ())
    } -> Event Shared a

instance Functor (Event (Local t))
instance Functor (Event Shared)
```

Listing 4.12.: Event type definition

There is also a `Monoid` instance which merges events by subscribing handler actions to all underlying events. The unit value (`mempty`) is an event which never fires; the handler action is simply discarded. For convenience and consistency with other FRP libraries, these functions are also available with the names `never` and `merge`. Merging a list of events can be done with the derived monoid combinator `mconcat`, also available as `mergeAll`.

```
instance Monoid (Event (Local t) a)
instance Monoid (Event Shared a)

never :: Event (Local t) a
never = mempty

merge :: Event (Local t) a -> Event (Local t) a -> Event (Local t) a
merge = mappend

mergeAll :: [Event (Local t) a] -> Event (Local t) a
mergeAll = mconcat
```

Listing 4.13.: Event monoid

Events are created by the `newEvent` action which produces a (shared) event and an action to fire the event, pushing a value to its handlers. This is implemented by keeping a registry of event handlers by a unique identifier. Firing the event is done by simpling calling all the handlers in the registry with the event value.

Subscription is done by producing a fresh identifier and adding the handler to the registry at that identifier. The handler can be unsubscribed by deleting the handler for the identifier from the registry.

```
newEvent :: IO (Event Shared a, a -> Execution ())
proxyEvent :: IO (Event Shared a, Event Shared a -> Execution ())
whenJust :: Event t (Maybe a) -> Event t a
```

Listing 4.14.: Event creation

Another way to create an event is by creating a proxy for another event that can later be swapped using the `proxyEvent` function. This function returns a new event and an action to set that event as proxy for a given event. This is done by first unsubscribing to the previously proxied event and then subscribing to the new event.

Events can also be filtered using the `whenJust` function, which only runs the handler actions when the `Maybe`-wrapped value is not empty.

Behaviors are built on top of `Event` and there are again two constructors:

- *Local* behaviors are not yet initialized: they are implemented as an action that produces an initialized behavior.
- *Shared* behaviors are initialized: they consist of an action to read their current value, and an event that fires whenever the value has refreshed.

The `Functor` instance applies the mapping function to the action that reads the current value. There is also an `Applicative` instance: `pure` is defined as a behavior that always returns a given value and never changes, and `<*>` is defined by reading the current value of the given behaviors.

```
data Behavior t a where
  LocalB :: Execution (Behavior Shared a) -> Behavior (Local t) a
  SharedB :: Execution a -> Event Shared () -> Behavior Shared a

instance Functor (Behavior (Local t))
instance Functor (Behavior Shared)

instance Applicative (Behavior (Local t))
instance Applicative (Behavior Shared)
```

Listing 4.15.: Behavior type definition

Behaviors can be created from events using the stateful combinators `stepper` and `accumB`, the former being implemented in terms of the latter. `accumB` is implemented by initializing a mutable reference and subscribing to the event. When the event fires its value is used to update the reference. The same event is then also used to indicate when the behavior is changed.

```
accumB :: a -> Event (Local t) (a -> a) -> Behavior (Local t) a
accumB x e = LocalB $ do
  ref <- liftIO (newIORef x)
  e' <- e_run e
  e_subscribe e' $ \f -> do
    x <- liftIO $ readIORef ref
    let x' = f x
    liftIO $ writeIORef ref x'
  return SharedB
    { b_sample = liftIO $ readIORef ref
    , b_pulses = () <$ e'
    }

stepper :: a -> Event (Local t) a -> Behavior (Local t) a
trackM :: Eq k => Behavior (Local t) [k] -> (k -> Execution a) -> Behavior (Local t) [a]
```

Listing 4.16.: Creating and tracking behaviors

The `track` combinator is implemented using the `trackM` combinator which implements a form of higher-order FRP: a behavior of key values is observed, and when new keys are added an action is run to produce a value. These values are tracked in a list of key-value pairs. When a key is removed from the list the corresponding value is dropped.

As described in chapter 3, events and behaviors can be shared using `startB` respectively `startE`. This is implemented using `executeB` respectively `executeE`. The `executeB` function uses deferred execution to defer initialization to allow for recursive definitions. Upon initial execution only an (uninitialized) reference to actual behavior

value is created, while execution of the behavior itself (involving event subscription in the case of `accumB`) is deferred.

Shared events and behaviors can be converted back to local ones using `useE` respectively `useB`, which simply wrap them in the corresponding constructors. Finally, events can be extracted from dynamic components using the function `getEvent` which simply accesses the `c_event` field and localizes the event using `useE`.

```
executeB :: Behavior (Local t) a -> Deferred Execution (Behavior Shared a)
executeB lb = do
  ref <- liftIO (newIORef (error "behavior not yet initialized"))
  defer . lift $ do
    sb <- b_run lb
    liftIO (writeIORef ref sb)
  return SharedB
    { b_sample = b_sample =<< liftIO (readIORef ref)
    , b_pulses = SharedE $ \h -> do
        SharedB { .. } <- liftIO (readIORef ref)
        e_subscribe b_pulses h
    }

executeE :: Event (Local t) a -> Deferred Execution (Event Shared a)
executeE le = lift (e_run le)

useB :: Behavior Shared a -> Behavior (Local t) a
useB b = LocalB (return b)

useE :: Event Shared a -> Event (Local t) a
useE e = LocalE (return e)

getEvent :: Component (Dynamic t) a -> Event (Local t) a
getEvent DynamicComponent { .. } = useE c_event
```

Listing 4.17.: Initialization and conversion

4.5. Simulation adapter

The `runStartRoot` function is used for starting a component structure. Given a root component it runs an `IO` action that returns a list of element actions for the initial render, the element identifier of the root element and an action to fire the master DOM event.

```
runStartRoot
  :: StartComponent a -> IO ([ElementAction], ElementId, DomEventInfo
  -> IO [ElementAction])
runStartRoot (StartComponent sc) = do
  (mde, fire) <- newEvent
  (comp, s@(reg, _, _), as) <- runRWST (executeStart sc) mde (Map.empty, 0, 0)
  ref <- newIORef s
  let
    update dei = do
      s <- readIORef ref
      ((), s', as) <- runRWST (fire dei) mde s
      writeIORef ref s'
      return as
  return (as, reg ! c_id comp, update)
```

Listing 4.18.: Running the root component

The GUI adapter is responsible for processing event actions and rendering the actual user interface elements to the screen while keeping track of elements by their given identifier. This section will describe an adapter that simulates an HTML DOM by applying element actions on a tree structure.

The `Document` type holds a map of node elements by their identifier and the identifier of the root element. The `Node` type represents an HTML element and is very similar to the `ElementDefinition` type. Child nodes are expressed using a list of element identifiers.

```
type Document = (Map ElementId Node, ElementId)
data Node = Node
  { n_elementId :: ElementId
  , n_namespace :: Namespace
  , n_tagName :: TagName
  , n_attributes :: Map AttributeName AttributeValue
  , n_text :: Maybe String
  , n_eventSources :: Set EventName
  , n_children :: [ElementId]
  } deriving Show

newNode :: ElementId -> Namespace -> TagName -> Node
newNode n_elementId n_namespace n_tagName = Node
  { n_attributes = Map.empty
  , n_text = Nothing
  , n_eventSources = Set.empty
  , n_children = []
  , ..
  }

applyAction :: Document -> ElementAction -> Document
```

Listing 4.19.: Document and node type definition

Applying actions to the simulated node structure is done using the `applyAction` function. This involves straight-forward inserts, updates and deletes from the node map

and node fields.

The document can be pretty-printed using the `ppDocument` function. This function renders the element structure as a string and show all elements identifiers, attributes and event sources.

```
ppDocument :: Document -> String

startSim
  :: StartComponent void -> IO (Document, Node -> EventName -> EventData -> IO Document)

findNode :: Document -> (Node -> Bool) -> Node
findNodeN :: Int -> Document -> (Node -> Bool) -> Node
findNodes :: Document -> (Node -> Bool) -> [Node]
```

Listing 4.20.: Simulating user interaction

The `startSim` function is based on `runStartRoot` and takes care of applying element actions to the document structure. Given a root component it returns the initially rendered document and an action to fire DOM events.

The simulator adapter is useful during development of user interfaces to test components and inspect the elements they create. The functions `findNode`, `findNodeN` and `findNodes` can be used to find nodes within the document structure based on a predicate.

The example of listing 4.21 demonstrates how the document simulator can be used to start a component and click on a button. The output of running this example is as follows:

```
initial render:
<button#0 onclick>
    0
</button#0>

after button click:
<button#0 onclick>
    1
</button#0>
```

```
sim :: IO ()
sim = do
  (doc, fire) <- startSim example
  putStrLn "initial render:"
  putStrLn (ppDocument doc)
  putStrLn ""
  putStrLn "after button click:"
  doc' <- fire (findNode doc (\Node { .. } -> n_tagName == "button")) "click" ""
  putStrLn (ppDocument doc)
  return ()

example :: StartComponent Int
example = StartComponent $ mdo
  comp <- startC $ counterButton <$> stepper 0 (getEvent comp)
  return comp
  where
    counterButton :: Int -> Component Static Int
    counterButton x = x + 1 <$ Html.onClick (Html.button (show x))
```

Listing 4.21.: Example: simulation of the counter button component

5. Related work

5.1. Key properties

This chapter will compare *Slim*, the main contribution from this thesis, to a couple of modern (FRP) libraries for creating user interfaces, ranging from a first-order FRP with imperative supplements to higher-order FRP. The strengths and drawbacks of each library will be highlighted, focusing on the following properties:

Dynamicity
: Can interface elements be created and updated in a declarative manner, without resorting to imperative code? Some applications contain interface elements which must to be added, changed or removed based on application state.

Composability
: Can interface elements be defined as isolated components, to be seamlessly combined with other components? Large applications are composed of many different interface elements. By grouping these interface elements together in small units they can be developed in isolation from the rest of the system. Applications can then be built by combining these components, much the same way those components were built by combining the basic interface elements.

Bidirectionality
: Can components display the changes that they capture, using feedback loops and local state? Some components may need to process user input and keep small amounts of local state. Consider an interface element that supports drag-and-drop functionality where mouse positions are stored so that mouse move offsets can be computed and applied to element and its data.

5.2. First-order FRP: *Threepenny.Reactive*

5.2.1. Overview

Threepenny is a user interface library for Haskell that ships with a first-order FRP system that integrates well with the imperative parts. Although *Threepenny* applications run in Haskell, the user interface is HTML-based and viewed via a browser.

Interface elements are created using side-effecting actions of type `UI` but they can be updated using the `sink` action which uses the output of a behavior to keep an element attribute up to date. This function connects the output of a behavior with the attribute of an element and keeps that attribute up to date with the value of the behavior.

Creating elements using `sink` is declarative: the element is created once and updated based on the specified behavior. In simple cases, where not much dynamicity is needed, no further actions need to be run to update interface elements after they are created.

For example, a simple counting button implemented in *Threepenny* looks as follows:

```
-- creates a button element; UI is not much more than a wrapper around IO
button :: UI Element

-- creates an accumulating behavior
accumB :: a -> Event (a -> a) -> UI (Behavior a)

-- returns an event that fires when the element is clicked
click :: Element -> Event ()

-- reverse function application, this is used to update attributes on an element after
creation
(#) :: a -> (a -> b) -> b

-- the type of an attribute for element type x, input type i and output type o
data ReadWriteAttr x i o

-- attribute representing the text contents of an element
text :: ReadWriteAttr Element String ()

-- use a behavior to continually update an attribute on an element
sink :: ReadWriteAttr x i o -> Behavior i -> UI x -> UI x
```

Listing 5.1.: *Threepenny* library functions used by listing 5.2

```
counter :: UI Element
counter = do
  countingButton <- button
  count <- accumB 0 ((+1) <$ click countingButton)

  return countingButton
    # sink text (show <$> count)
```

Listing 5.2.: Simple counting button implementation in *Threepenny*

Note the careful order of setup steps: first the button element is created, then a behavior is initialized based on the behavior, and then the button is reconfigured to use the behavior for its text attribute.

5.2.2. Dynamicity

While *Threepenny*'s `sink` combinator is very useful for updating attributes of a static set of interface elements, things become more complicated when the structure of interface elements is dynamic. This is illustrated by the following example which implements a list of buttons, each of which keeps track of their own click count, and they can be added and removed by buttons labeled "+" and "-".

```
-- runs an IO action from UI
liftIO :: UI a -> IO a

-- creates a new event and an action to fire it
newEvent :: IO (Event a, a -> IO ())

-- register an event handler for an element's event
on :: (element -> Event a) -> element -> (a -> UI void) -> UI ()

-- merge two events with the function applied when they fire simultaneously
unionWith :: (a -> a -> a) -> Event a -> Event a -> Event a

-- attribute representing the child elements of an element
children :: ReadWriteAttr Element [Element] ()

-- aligns elements in a row
row :: [UI Element] -> UI Element

-- aligns elements in a column
column :: [UI Element] -> UI Element
```

Listing 5.3.: *Threepenny* library functions used by listing 5.4

```
listOfCounters :: UI Element
listOfCounters = do
  (addElement, fireAddElement) <- liftIO newEvent

  plusButton <- button
    # set text "+"
  on click plusButton $ \() -> do
    newCounter <- counter
    let
      update es = es ++ [newCounter]
    liftIO (fireAddElement update)

  minusButton <- button
    # set text "-"
  let
    remove [] = []
    remove es = init es
    removeElement = remove <$ click minusButton

  elements <- accumB [] (unionWith (.) addElement removeElement)
  elementsDiv <- div
    # sink children elements

  column
    [ row [return plusButton, return minusButton]
    , return elementsDiv
    ]
```

Listing 5.4.: Dynamic list of counting buttons in *Threepenny*

Listing 5.4 consists of a considerable amount of imperative code: the click event from `plusButton` has an event handlers with side effects attached. This is because creating a new `counter` element (see listing 5.2) is an action with a side effect: a new interface element is created and a behavior is initialized. The event handler uses a specially created event (`addElement`) that it can fire manually with the newly created element. From this event (and the click event from `minusButton` which simply drops the last element from the list), a behavior is created that contains the counter elements to be shown.

5.2.3. Composability

Creation of elements in *Threepenny* always requires side-effecting actions so components must also be defined as actions. These actions are of type `UI`, which is a wrapper around `IO`. Consequently, *Threepenny* components consist of actions with unconstrained side-effects, so composability is not controlled by Haskell type-system. In addition, the order in which components are run may have affect the outcome.

5.2.4. Bidirectionality

Bidirectionality in a *Threepenny* application may involve recursive definitions. This can sometimes be problematic because many combinators involve actions for which the precise order is important. The following example illustrates this by implement a counting button using a recursive `do`-block.

```
counter :: UI Element
counter = mdo
  countingButton <- button
    # sink text (show <$> count)

  count <- accumB 0 ((+1) <$ click countingButton)

  return countingButton
```

Listing 5.5.: Recursive implementation of a counting button in *Threepenny*

In listing 5.5 the `sink` combinator is applied before the `count` behavior is defined. This is possible because the `sink`'s implementation[1] explicitly defers observation of the behavior until the `UI` block has run. If it it wasn't this code would produce a program that gets stuck in an endless loop due to two actions that are mutually dependent on each other's result. This situation can be triggered by simply swapping the lines for `countingButton` and `count`.

In practice it is not always clear how to avoid this issue in *Threepenny* when using just FRP combinators and recursive `do`-notation. This means that bidirectionality in

[1] `UI` includes a writer monad with `IO` actions to be run at the end of a `UI` block. (The `Deferred` monad transformer from Chapter 4 is inspired by this method.)

Threepenny sometimes involves regular `do`-blocks and imperative actions; for instance using split element definitions such as in listing 5.2 or using extra events as in listing 5.4.

5.3. Higher-order FRP: *Sodium*

5.3.1. Overview

To recap from section 2.4, *Sodium* is a Haskell library for general purpose higher-order FRP. The `Reactive` type is used to initialize stateful events and behaviors and can also contain side-effecting (IO) actions.

Although it does not include any functionality for creating user interfaces it is straightforward to use *Sodium* together with Threepenny. For instance, interface updates can be performed by producing actions from events using `execute :: Event (Reactive a) -> Event a`. As many of its combinators involve the `Reactive` monad most *Sodium* code lives in `do`-blocks and consists of side-effecting actions.

5.3.2. Dynamicity

Sodium's higher-order FRP functionality goes some way towards making dynamic interface element structures possible to define declaratively. An application that consists of a dynamic list of counting buttons tests an FRP library in the following ways:

1. Can interface elements be added and removed based on events in a declarative way?
2. Can events produced from these interface elements be merged into a single event stream?

Sodium's `switch` combinator takes care of the latter, but the first still requires `IO` actions being run in response to events. Chapter 3 shows this can be solved by tight integration between the FRP combinators and user interface library.

5.3.3. Composability

Just as with Threepenny, user interface components written using *Sodium* will consist of actions that may involve `IO`, providing no guarantees about composability. Composition involves managing side-effects and careful ordering of initialization.

5.3.4. Bidirectionality

Unlike most of Threepenny's `UI` actions, *Sodium*'s `Reactive` actions are capable of recursive definitions. The FRP system has a notion of *transactions*: behaviors are only updated at the end of transactions so that actions will observe values set by the previous transaction. As a result, it is easier to recursively define events and behaviors based on themselves without the program becoming stuck.

5.4. Pure, first-order FRP: *Elm*

5.4.1. Overview

To recap from section 2.7, *Elm* is a programming language specialized for creating user interfaces using first-order FRP. Instead of distinguishing between events and behaviors, *Elm* works with *signals*[2] which can be thought of as behaviors defined over discrete time. Because FRP in *Elm* is first-order, network configurations are static and signals of signals cannot be flattened.

Elm's interface library is based on a virtual DOM, similar to Facebook's React, and targets HTML. The `main` function is the only "sink" applications can utilize and it must be a Signal representing the structure of the whole user interface. Whenever this signal is updated the new structure is compared to the previous one and the changes are applied to the actual HTML DOM. The user interface structure may contain event sources that specify which events should be subscribed to and to which signal (via an *address*) these events should be sent.

5.4.2. Dynamicity

The combination of FRP and a virtual DOM interface is sufficient for expressing how a user interface structure depends on events over time. An application with a dynamic list of counting buttons can be implemented using an `Html` div element with a varying number of child buttons. However, signals cannot be added and removed dynamically in *Elm* so each button must posts its events to the same signal.

5.4.3. Composability

Although the `Html` type can provide dynamic user interfaces, it is limited when it comes to composability. `Html` values cannot introduce any state or signals; they must be provided separately. As a result, components in *Elm* consist of several different parts which need to be integrated into different places in the application::

- A *model* type that defines all the state the component needs. This type needs to be added in the application model type as its own field so that it can be stored, updated and used for the view function.
- An *action* type and *update* function to apply actions to the state to update it. This type needs to be added to the application action type under its own constructor so that they can be fired and handled.
- A *view* function that creates an `Html` structure from the state. This view function needs to be added to the application view function. In addition, it must be passed its own part of the application state and its actions need to be wrapped with the new constructor.

[2] Just like chapter 2, this section describes *Elm* as of version 0.16 since version 0.17 removed the FRP system.

In short, the issue with composability in *Elm* is caused by the inability to define components as single, isolated units and instead the need to integrate them in multiple places the application. In addition to the inconvenience, this limitation introduces boilerplate code (such as the introduction of extra type constructors) and increased complexity.

5.4.4. Bidirectionality

Elm enforces a strict unidirectional data flow: user input is sent to a statically defined set of signals which are then used to update the state and generate an interface element structure. This element structure defines the user input events and how they should be channeled. Data always flows in this circle and state is only kept at one place. As a result, components cannot work with their own signals directly and cannot have local state: bidirectionality within components is simply not possible.

5.5. Conclusions

Threepenny.Reactive provides a simple set of FRP combinators that integrates well with the (imperative) Threepenny user interface toolkit.

+ Easy to use FRP combinators that integrate well with imperative code. This makes it easy for programmers experienced with imperative GUI programming to get started with FRP and allows seamless embedding of FRP in imperative applications.
+ Properties and non-interactive interface elements such as text labels and colors are easy to hook up to behaviors using `sink`.
- Because it is limited to first-order FRP it is frequently necessary to use imperative actions to effect changes in the network configuration and user interface.
- Stateful FRP networks are defined by side-effecting `UI` actions.
- While it is possible to specify recursive events and behaviors using `UI`'s `MonadFix` instance, it easy to introduce recursive actions that cause the program to become stuck.

Sodium features a powerful FRP system that is well-equipped for the dynamicity of the network editor application.

+ Dynamic reactive networks can be expressed using its higher-order FRP constructs.
+ Recursive events and behaviors are easy to define.
- While the higher-order FRP constructs facilitate dynamic data flow, imperative actions are still required for the dynamic interface elements.

Of the related work described in this chapter, *Elm* allows for the simplest, smallest and purest example applications (in the 'purely functional' sense.)

+ Applying the patterns described in the Elm Architecture tutorial resulted in an application consisting of modular components, communicating via localized channels.
+ The immediate-mode interface provided by the *elm-html* library allows for seamless interface updates without writing code that keeps track of changes.
+ Elms first-order, side-effect free FRP constructs results in code that is simple and easy to reason about.
- *Elm* applications consist of a single, one-directional data flow, and each component has to be hooked up to it in several different places (view, updates, actions, and state.)
- When components rely on channels (as prescribed by the Elm Architecture) instead of signals they cannot have local state.

Slim combines the strengths of all of the 3 examined FRP libraries:

+ It provides an immediate-mode interface for simpler interface updates.
+ It provides a way to define components so that they can be seamlessly integrated, without hooking them up in multiple places.
+ It allows components to define local state without compromising referential transparency.
+ It allows recursive definitions with no risk of the program becoming stuck.

6. Conclusion

6.1. Summary

Creating interactive user interfaces traditionally involves an imperative programming style. Commands and statements react to input events to update the system state and interface elements. An unanticipated sequence of input events or subtle bugs in the event handlers may lead to an invalid system state.

Functional programming provides an alternative to the imperative style by relying on (pure) functions and by providing equational reasoning. In turn, *Functional Reactive Programming (FRP)* defines the essentials for modeling reactive systems using pure functions and equations.

This report introduces *Slim*, a DSL for Haskell that allows for pure functional user interface programming using FRP. Its language constructs are substantiated and demonstrated by a set of example user interfaces of increasing dynamicity.

Slim combines pure FRP constructs with an immediate mode interface element rendering library based on a Virtual DOM. By taking advantage Haskell's type system the programmer can define stateful FRP networks without using order-dependent monadic structures. Components can be defined in a monadic `do`-block to indicate dependencies between components. These dependencies may be cyclical using (safe) recursive definitions.

A proof-of-concept implementation is provided that, while being relying on `IO` actions, provides a pure API. In addition, a GUI simulator is provided that can be used to (unit) test components and programs without running an actual GUI library.

Slim is compared to a number of established FRP libraries in programming interfaces to demonstrate how it is able to seamlessly implement classes of user interfaces that the existing libraries struggle with.

6.2. Future work

The simulator adapter makes it possible to write unit tests and *QuickCheck*[5] properties for interactive programs. A set of helpers and properties should make it easier to test components and check if expected properties hold.

Running a program in debug mode could write a log of element actions to facilitate debugging. Such a log could be played back in simulation mode so that the exact state of the interface elements can be checked at every step to identify problems.

Slim is not as expressive as other higher-order FRP languages because `mount` only instantiates components, not behaviors directly. This is not as expressive as, for example, `switchB` as component's events has to originate from the DOM. The consequences of this limitation deserves further experimentation in larger applications.

The implementation relies heavily on `IO`. An implementation that relies less on mutable references and more on pure updates could be more efficient and easier to debug.

The `Event` type described in chapter 4 misses a few possibly useful combinators such as accumulation and snapshotting. Adding new ways to create `Event`s may need further testing to ensure recursive definitions are still safe.

There is currently no defined way of interacting the "outside world" such as the file system. The implementation could be extended so that the value produced by the root component is returned when a DOM event is fired. Alternatively, a new `ElementAction` constructor could be introduced for this purpose.

The implementation listed in chapter 4 is not tuned for performance. There may be cases where strictness annotations limit memory usage to provide speedups. Events and behaviors may also benefit from memoization to improve sharing when they are used in multiple places. In addition, a single input event may lead to multiple events triggering rerenders so it may be more efficient to delay reconciliation in those cases.

The virtual DOM algorithm could be improved by taking advantage of element and component identifiers to track subtrees throughout the component structure instead of focusing only on the position they are in. For example, when reordering a list of components it would be more efficient to reorder the elements as well instead of recreating the elements. This would require an extra `ElementAction` constructor for moving elements from one place of the DOM structure to another.

Not much care has been taken to prevent memory leaks; components that are removed from the program are not cleaned up. This could be accomplished using weak references or through a more pure implementation.

Appendices

A. Full implementations

A.1. Code omitted from section 4.2

```
instance Functor (Component t) where
  fmap f component = case component of
    StaticComponent { .. } ->
      StaticComponent { c_eventRouter = fmap f c_eventRouter, .. }
    DynamicComponent { .. } ->
      DynamicComponent { c_event = fmap f c_event, .. }
    MountedComponent { .. } ->
      MountedComponent { c_component = fmap f c_component }


instance Functor (EventRouter a) where
  fmap f EventRouter { .. } = EventRouter
    { er_dom = fmap f . er_dom
    , er_sub = fmap f . er_sub
    }

mkEvent :: EventRouter a b -> ElementId -> [Event Shared a] -> Event Shared b
mkEvent EventRouter { .. } ei subEvents = er_dom ei <> er_sub subEvents


nullRouter :: EventRouter a b
nullRouter = EventRouter
  { er_dom = const mempty
  , er_sub = const mempty
  }

childRouter :: EventRouter a a
childRouter = EventRouter
  { er_dom = const mempty
  , er_sub = mconcat
  }

addEvent :: EventName -> (EventData -> a) -> Component Static a -> Component Static a
addEvent en f c@StaticComponent { .. } = StaticComponent
  { c_eventRouter = c_eventRouter
    { er_dom = \ei -> fmap f (getDomEvent en ei) <> er_dom ei
    }
  , c_elementDefinition = c_elementDefinition
    { ed_eventSources = en : ed_eventSources
    }
  , ..
  }
  where
    ElementDefinition { .. } = c_elementDefinition
    EventRouter { .. } = c_eventRouter

silence :: Component Static void -> Component Static a
silence c =
```

```
  case c of
    StaticComponent { .. } -> StaticComponent
      { c_eventRouter = nullRouter
      , c_elementDefinition = c_elementDefinition { ed_eventSources = [] }
      , ..
      }
    MountedComponent { .. } -> MountedComponent
      { c_component = c_component { c_event = never }
      , ..
      }


containerComponent
  :: Maybe String -> String -> [(String, String)] -> [Component Static a]
  -> Component Static a
containerComponent ed_namespace ed_tagName ed_attributes c_children = StaticComponent {
.. }
  where
    c_eventRouter = childRouter
    c_elementDefinition = ElementDefinition { .. }
    ed_text = Nothing
    ed_eventSources = []

textComponent
  :: Maybe String -> String -> [(String, String)] -> String
  -> Component Static Void
textComponent ed_namespace ed_tagName ed_attributes text = StaticComponent { .. }
  where
    c_eventRouter = nullRouter
    c_children = []
    c_elementDefinition = ElementDefinition { .. }
    ed_text = Just text
    ed_eventSources = []

emptyComponent
  :: Maybe String -> String -> [(String, String)]
  -> Component Static Void
emptyComponent ed_namespace ed_tagName ed_attributes = StaticComponent { .. }
  where
    c_eventRouter = nullRouter
    c_children = []
    c_elementDefinition = ElementDefinition { .. }
    ed_text = Nothing
    ed_eventSources = []


freshElementId :: Reconciliation ElementId
freshElementId = do
  i <- get
  put (i + 1)
  return i

firstRender :: Component t a -> Reconciliation (RenderedComponent, Event Shared a)
firstRender rc_component = case rc_component of
```

```
  StaticComponent { c_elementDefinition = ElementDefinition { .. }, .. } -> do
    rc_id <- freshElementId
    tell $
      [ Create rc_id ed_namespace ed_tagName
      , SetText rc_id ed_text
      ] ++
      [ Subscribe rc_id en | en <- ed_eventSources
      ] ++
      [ SetAttribute rc_id an v | (an,v) <- ed_attributes
      ]
    (rc_children, subEvents) <- unzip <$> mapM firstRender c_children
    tell
      [ AddChildren rc_id
        [ childId
        | RenderedComponent { rc_id = childId } <- rc_children
        ]
      ]
    return (RenderedComponent { .. }, mkEvent c_eventRouter rc_id subEvents)

  DynamicComponent { .. } -> do
    rc_id <- ($ c_id) <$> ask
    return (RenderedComponent { rc_children = [], .. }, c_event)

  MountedComponent { .. } ->
    firstRender c_component

reconcile
  :: RenderedComponent -> Component t a
  -> Reconciliation (RenderedComponent, Event Shared a)
reconcile
  rc@RenderedComponent
    { rc_component = StaticComponent { c_elementDefinition = prevElementDefinition }
    , ..
    }
  rc_component@StaticComponent { .. }
  | isCompatible prevElementDefinition c_elementDefinition = do
    updateElement rc c_elementDefinition
    (rc_children, subEvents) <- unzip <$> reconcileChildren rc_id rc_children c_children
    return (RenderedComponent { .. }, mkEvent c_eventRouter rc_id subEvents)

reconcile
  renderedComponent@RenderedComponent
    { rc_component = DynamicComponent { c_id = prevId }
    }
  DynamicComponent { .. }
  | prevId == c_id
  = return (renderedComponent, c_event)

reconcile renderedComponent MountedComponent { .. } =
  reconcile renderedComponent c_component

reconcile renderedComponent component = do
  terminate renderedComponent
  (renderedComponent', event) <- firstRender component
```

```
  tell [Replace (rc_id renderedComponent) (rc_id renderedComponent')]
  return (renderedComponent', event)

isCompatible :: ElementDefinition -> ElementDefinition -> Bool
isCompatible node node' =
  ed_namespace node == ed_namespace node' && ed_tagName node == ed_tagName node'

reconcileChildren
  :: ElementId -> [RenderedComponent] -> [Component t a]
  -> Reconciliation [(RenderedComponent, Event Shared a)]
reconcileChildren parentId renderedComponents components =
  catMaybes <$> mapM reconcileChild (zipMaybe renderedComponents components)
  where
    reconcileChild m =
      case m of
        (Just renderedComponent, Just component) -> do
          (renderedComponent', event) <- reconcile renderedComponent component
          return (Just (renderedComponent', event))
        (Just renderedComponent, Nothing) -> do
          terminate renderedComponent
          return Nothing
        (Nothing, Just component) -> do
          (renderedComponent, event) <- firstRender component
          tell [AddChildren parentId [rc_id renderedComponent]]
          return (Just (renderedComponent, event))

zipMaybe :: [a] -> [b] -> [(Maybe a, Maybe b)]
zipMaybe (x:xs) (y:ys) = (Just x, Just y) : zipMaybe xs ys
zipMaybe [] ys = [(Nothing, Just y) | y <- ys]
zipMaybe xs [] = [(Just x, Nothing) | x <- xs]


updateElement :: RenderedComponent -> ElementDefinition -> Reconciliation ()
updateElement
  RenderedComponent { rc_component = StaticComponent { c_elementDefinition = old }, .. }
  new = do
    tell $ updateText ++ updateAttributes ++ updateEventSources
    where
      updateText
        | ed_text old /= ed_text new = [SetText rc_id (ed_text new)]
        | otherwise = []

      updateAttributes =
        [ UnsetAttribute rc_id an | an <- Map.keys $
            Map.difference oldAttributes newAttributes
        ] ++
        [ SetAttribute rc_id an v | (an,v) <- Map.toList $
            Map.differenceWith updateAttr newAttributes oldAttributes
        ]
        where
          oldAttributes = Map.fromList (ed_attributes old)
          newAttributes = Map.fromList (ed_attributes new)
          updateAttr new old = if new == old then Nothing else Just new
```

```
      updateEventSources =
        [ Unsubscribe rc_id en | en <- Set.toList $
            Set.difference oldEventSources newEventSources
        ] ++
        [ Subscribe rc_id en | en <- Set.toList $
            Set.difference newEventSources oldEventSources
        ]
        where
          oldEventSources = Set.fromList (ed_eventSources old)
          newEventSources = Set.fromList (ed_eventSources new)


terminate :: RenderedComponent -> Reconciliation ()
terminate RenderedComponent { .. } = do
  tell [Destroy rc_id]
  mapM_ terminate rc_children
```

A.2. Code omitted from section 4.3

```
startB :: Behavior (Local t) a -> Start t (Behavior Shared a)
startB b = Start (executeB b)

startE :: Event (Local t) a -> Start t (Event Shared a)
startE e = Start (executeE e)

track
  :: Eq k => Behavior (Local t) [k] -> (k -> StartComponent a)
  -> Behavior (Local t) [Component (Dynamic t) a]
track b f = trackM b (\k -> case f k of StartComponent c -> executeStart c)

executeReconciliation :: Reconciliation a -> Execution a
executeReconciliation r = do
  (reg, cid, eid) <- get
  let (x, eid', as) = runRWS r (reg !) eid
  tell as
  put (reg, cid, eid')
  return x

startC :: Behavior (Local t) (Component Static a) -> Start t (Component (Dynamic t) a)
startC b = Start $ do
  c_id <- lift freshComponentId
  (c_event, redirect) <- liftIO proxyEvent
  b' <- executeB b
  defer . defer . lift $ do
    comp <- b_sample b'
    (rc, ev) <- executeReconciliation (firstRender comp)
    setComponentElementId c_id (rc_id rc)
    redirect ev
    ref <- liftIO $ newIORef rc
    e_subscribe (changes b') $ \comp' -> do
      rc <- liftIO $ readIORef ref
      (rc', ev) <- executeReconciliation (reconcile rc comp')
```

```
      setComponentElementId c_id (rc_id rc')
      liftIO $ writeIORef ref rc'
      redirect ev
    return ()
  return DynamicComponent { .. }

freshComponentId :: Execution ComponentId
freshComponentId = do
  (reg, cid, eid) <- get
  put (reg, cid + 1, eid)
  return cid

setComponentElementId :: ComponentId -> ElementId -> Execution ()
setComponentElementId cid eid = do
  (reg, cid', eid') <- get
  put (Map.insert cid eid reg, cid', eid')
```

A.3. Code omitted from section 4.4

```
instance Functor (Event (Local t)) where
  fmap f (LocalE me) = LocalE (fmap f <$> me)

instance Functor (Event Shared) where
  fmap f (SharedE g) = SharedE $ \h -> g (h . f)

newEvent :: IO (Event Shared a, a -> Execution ())
newEvent = do
  counter <- newIORef 0
  registry <- newIORef Map.empty
  value <- newIORef Nothing

  let
    subscribe handler = liftIO $ do
      i <- readIORef counter
      writeIORef counter (i + 1)
      modifyIORef registry (Map.insert i handler)
      return . liftIO $ do
        modifyIORef registry (Map.delete i)

    fire x = do
      liftIO $ writeIORef value (Just x)
      handlers <- Map.elems <$> liftIO (readIORef registry)
      mapM_ ($ x) handlers

  return (SharedE subscribe, fire)

proxyEvent :: IO (Event Shared a, Event Shared a -> Execution ())
proxyEvent = do
  (ev, fire) <- newEvent
  ref <- newIORef (return ())
  let
    redirect ev' = do
```

```
      join (liftIO $ readIORef ref)
      unsubscribe <- onEvent ev' fire
      liftIO $ writeIORef ref unsubscribe
  return (ev, redirect)

whenJust :: Event t (Maybe a) -> Event t a
whenJust e = case e of
  LocalE me -> LocalE (whenJust <$> me)
  SharedE f -> SharedE $ \g -> f (maybe (return ()) g)

instance Functor (Behavior (Local t)) where
  fmap f (LocalB mb) = LocalB (fmap f <$> mb)

instance Functor (Behavior Shared) where
  fmap f (SharedB mx e) = SharedB (f <$> mx) e

instance Applicative (Behavior (Local t)) where
  pure x = LocalB (return (pure x))
  LocalB mbf <*> LocalB mbx = LocalB ((<*>) <$> mbf <*> mbx)

instance Applicative (Behavior Shared) where
  pure x = SharedB (return x) never
  SharedB mf e1 <*> SharedB mx e2 = SharedB (mf <*> mx) (e1 <> e2)


stepper :: a -> Event (Local t) a -> Behavior (Local t) a
stepper x e = accumB x (const <$> e)

trackM :: Eq k => Behavior (Local t) [k] -> (k -> Execution a) -> Behavior (Local t) [a]
trackM lbks f = LocalB $ do
  let
    update xs k = do
      case lookup k xs of
        Just x -> return (k, x)
        Nothing -> do
          x <- f k
          return (k, x)
  bks <- b_run lbks
  ks <- b_sample bks
  xs <- mapM (update []) ks
  ref <- liftIO $ newIORef xs
  e_subscribe (changes bks) $ \ks' -> do
    xs <- liftIO $ readIORef ref
    xs' <- mapM (update xs) ks'
    liftIO $ writeIORef ref xs'
  return SharedB
    { b_sample = map snd <$> liftIO (readIORef ref)
    , b_pulses = b_pulses bks
    }
```

A.4. Code omitted from section 4.5

```
applyAction :: Document -> ElementAction -> Document
applyAction (nodes, rootId) action =
  case action of
    Create ei ns tn ->
      (Map.insert ei (newNode ei ns tn) nodes, rootId)

    Replace ei1 ei2 ->
      let f n = n { n_children = [if ei1 == ei then ei2 else ei | ei <- n_children n] }
      in (Map.map f nodes, if ei1 == rootId then ei2 else rootId)

    Destroy ei ->
      let f n = n { n_children = filter (ei /=) (n_children n) }
      in (Map.map f (Map.delete ei nodes), rootId)

    SetAttribute ei an av ->
      let f n = n { n_attributes = Map.insert an av (n_attributes n) }
      in (Map.adjust f ei nodes, rootId)

    UnsetAttribute ei an ->
      let f n = n { n_attributes = Map.delete an (n_attributes n) }
      in (Map.adjust f ei nodes, rootId)

    SetText ei t ->
      let f n = n { n_text = t }
      in (Map.adjust f ei nodes, rootId)

    AddChildren ei eis ->
      let f n = n { n_children = (n_children n) ++ eis }
      in (Map.adjust f ei nodes, rootId)

    Subscribe ei en ->
      let f n = n { n_eventSources = Set.insert en (n_eventSources n) }
      in (Map.adjust f ei nodes, rootId)

    Unsubscribe ei en ->
      let f n = n { n_eventSources = Set.delete en (n_eventSources n) }
      in (Map.adjust f ei nodes, rootId)

ppDocument :: Document -> String
ppDocument (nodes, rootId) = renderStyle style (ppNode $ nodes ! rootId)
  where
    ppAttribute (k, v) = text k <> text "=" <> text v
    ppEventSource k = text ("on" ++ k)
    ppChildren childIds = vcat [ppNode n | Just n <- map (`Map.lookup` nodes) childIds]
    ppNode Node { .. } =
      text "<" <> ppElementName <+> ppAttributes <+> ppEventSources <> text ">" $$
        nest 4 (maybe empty text n_text $$ ppChildren n_children) $$
        text "</" <> text n_tagName <> text "#" <> int n_elementId <> text ">"
      where
        ppElementName = text n_tagName <> text "#" <> int n_elementId
        ppAttributes = hsep (ppAttribute <$> Map.toList n_attributes)
        ppEventSources = hsep (ppEventSource <$> Set.toList n_eventSources)
```

```
startSim
  :: StartComponent void -> IO (Document, Node -> EventName -> EventData -> IO Document)
startSim s = do
  (as, rootId, fire) <- runStartRoot s
  let doc = foldl' applyAction (Map.empty, rootId) as
  ref <- newIORef doc
  let
    fire' n en ed = do
      doc <- readIORef ref
      as <- fire (n_elementId n, en, ed)
      let doc' = foldl' applyAction doc as
      writeIORef ref doc'
      return doc'
  return (doc, fire')

findNode :: Document -> (Node -> Bool) -> Node
findNode = findNodeN 0

findNodeN :: Int -> Document -> (Node -> Bool) -> Node
findNodeN x doc f =
  case drop x (findNodes doc f) of
    (n:_) -> n
    [] -> newNode (-1) Nothing "not found"

findNodes :: Document -> (Node -> Bool) -> [Node]
findNodes (nodes, rootId) f = filter f (bfs [nodes ! rootId])
  where
    bfs [] = []
    bfs ns = ns ++ bfs [nodes ! i | n <- ns, i <- n_children n]
```

B. List of Listings

Acknowledgements

I want to thank my supervisors Wouter Swierstra and Atze Dijkstra who helped me by guiding this project in the right direction. In particular I am grateful to Wouter for his patient and helpful supervision throughout the project, and to Atze for introducing me to FRP and helping me identify some open problems to research.

My thanks to the Haskell community, for all their mind-bending inventions and discoveries, and to Heinrich Apfelmus in particular for providing valuable feedback to my ideas (and also for creating *Reactive Banana* and *Threepenny GUI*.)

I would not have gotten this far with my studies without my father's support. He inspired me and kept me motivated. I will forever remember him by everything he helped me achieve.

My special thanks to Naomi for her support, especially when I needed it the most.

Bibliography

[1] Heinrich Apfelmus. *FRP - Dynamic Event Switching in reactive-banana-0.7*. Sept. 2012. URL: http://apfelmus.nfshost.com/blog/2012/09/03-frp-dynamic-event-switching-0-7.html.

[2] Heinrich Apfelmus. *FRP - Release of reactive-banana version 1.0*. Oct. 2015. URL: http://apfelmus.nfshost.com/blog/2015/10/29-frp-banana-1-0.html.

[3] Koen Claessen Atze van der Ploeg. "Princlipled Practical FRP: Forget the past, change the future, FRPNow!" In: *ICFP*. 2015.

[4] Facebook Christopher Chedeau. *Reconciliation | React*. 2013-2016. URL: https://facebook.github.io/react/docs/reconciliation.html.

[5] 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. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[6] Evan Czaplicki. "Controlling Time and Space: understanding the many formulations of FRP". Strange Loop. 2014. URL: http://www.thestrangeloop.com/2014/controlling-time-and-space-understanding-the-many-formulations-of-frp.html.

[7] Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs*. 2012. URL: http://www.testblogpleaseignore.com/wp-content/uploads/2012/03/thesis.pdf.

[8] Evan Czaplicki. "The Elm Architecture". 2015-2016. URL: http://guide.elm-lang.org/architecture/index.html.

[9] Simon Marlow (editor). *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[10] Conal Elliott. "Push-pull functional reactive programming". In: *Haskell Symposium*. 2009. URL: http://conal.net/papers/push-pull-frp.

[11] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/.

[12] J. Hughes. "Why Functional Programming Matters". In: *Comput. J.* 32.2 (Apr. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. URL: http://dx.doi.org/10.1093/comjnl/32.2.98.

[13] Wolfgang Jeltsch. "Signals, Not Generators!" In: *Trends in Functional Programming*. 2009, pp. 145–160.

[14] Conor Mcbride and Ross Paterson. "Applicative Programming with Effects". In: *J. Funct. Program.* 18.1 (Jan. 2008), pp. 1–13. ISSN: 0956-7968. DOI: 10.1017/S0956796807006326. URL: http://dx.doi.org/10.1017/S0956796807006326.

[15] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional Reactive Programming, Continued". In: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*. Pittsburgh, Pennsylvania, USA: ACM Press, Oct. 2002, pp. 51–64.

[16] Gergely Patai. "Efficient and Compositional Higher-order Streams". In: *Proceedings of the 19th International Conference on Functional and Constraint Logic Programming*. WFLP'10. Madrid, Spain: Springer-Verlag, 2011, pp. 137–154. ISBN: 978-3-642-20774-7. URL: http://dl.acm.org/citation.cfm?id=2008270.2008280.

[17] Philip Wadler. "Monads for Functional Programming". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 24–52. ISBN: 3-540-59451-5. URL: http://dl.acm.org/citation.cfm?id=647698.734146.