Monadic Functional Reactive Programming

Atze van der Ploeg
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
ploeg@cwi.nl

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language Classifications—Applicative (functional) languages

Keywords Functional Reactive Programming; Monads

Abstract

Functional Reactive Programming (FRP) is a way to program reactive systems in functional style, eliminating many of the problems that arise from imperative techniques. In this paper, we present an alternative FRP formulation that is based on the notion of a *reactive computation*: a monadic computation which may require the occurrence of external events to continue. A *signal computation* is a reactive computation that may also emit values. In contrast to signals in other FRP formulations, signal computations can end, leading to a monadic interface for sequencing signal phases. This interface has several advantages: routing is implicit, sequencing signal phases is easier and more intuitive than when using the switching combinators found in other FRP approaches, and dynamic lists require much less boilerplate code. In other FRP approaches, either the entire FRP expression is re-evaluated on each external stimulus, or impure techniques are used to prevent redundant re-computations. We show how Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing redundant re-computations.

1. Introduction

Many computer programs are *reactive*: they engage in a dialogue with their environment, responding to events as they arrive. Examples of such programs are computer games, control systems, servers, and GUI applications. Imperative techniques to create reactive systems, such as the observer pattern, lead to plethora of problems: inversion of control, non-modularity and side effects [12].

Functional Reactive Programming (FRP) [7] is a programming paradigm to define reactive systems in functional style, eliminating many of the problems of imperative techniques. FRP has been successfully applied in many domains, such as robotics [8, 17, 18], computer vision [19], gaming [4], web programming [14] and graphical user interfaces [3].

The primary abstraction in FRP is a *signal* [15]: a value that changes over time. Traditionally, signals are modeled as mappings from points in time to values. For example, the position of the mouse can be modeled by a function that takes a number of seconds since the program started and returns the coordinates of the pointer at that time. Such signals can then be composed directly [7] or by composing *signal functions* [3], functions from signal to signal.

In this paper, we present a novel approach to FRP called *Monadic Functional Reactive Programming* that does not model signals as mappings from points in time to values. Instead, Monadic FRP is based on the notion of a *reactive computation*: a monadic computation which may require the occurrence of external events to continue. The Monadic FRP variant of a signal is a *signal computation*: a reactive computation that may also *emit* values during the computation.

This novel formulation has two main differences with other FRP approaches:

- In contrast to signals in other FRP formulations, signal computations can *end*. This leads to a simple, monadic interface for sequencing signal phases.
- In other FRP approaches, either the entire FRP expression is re-evaluated on each external stimulus, or impure techniques are used to prevent redundant re-computations: re-computing the current value of signal while the input it depends on has not changed. Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing such redundant re-computations.

Our contributions are summarized as follows:

- A novel monadic FRP programmer interface. We demonstrate this programming model by composing a drawing program from simple components (Section 2).
- A comparison of the Monadic FRP programmer interface with the programmer interface of other FRP formulations (Section 3).
- The first purely functional FRP evaluation model which prevents redundant re-computations (Section 4).
- The implementation of the composition functions from the programmer interface on top of this evaluation model (Section 5).
- A comparison of the Monadic FRP evaluation model with other FRP evaluation models (Section 6).

In Section 7 we conclude and discuss future work. A library based on the ideas in this paper is available as hackage package $DrClickOn$.

2. Programming with Monadic FRP

2.1 The drawing program

In this section, we demonstrate the Monadic FRP programming interface by composing a simple drawing program from small parts. The drawing program allows the user to draw boxes, change their color and delete boxes. The lifetime of each box consists of three phases:

Haskell '13, September 23–24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2383-3/13/09...$15.00.
http://dx.doi.org/10.1145/2503778.2503783

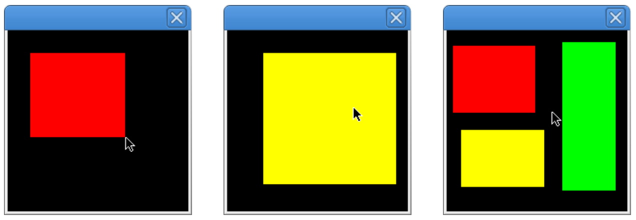
(a) (b) (c)

Figure 1: Screenshots of the simple drawing program.

1. *Define:* The user can define a box by holding down the left mouse button. The left-upper point of the rectangle is the mouse position when the user presses the left mouse button, the right-lower point is the mouse position when the user releases the left mouse button. While the user holds down the left mouse button, the preliminary rectangle is shown like in Figure 1(a).
2. *Choose color:* The user can cycle through possible colors for the box by pressing the middle mouse button, which changes the color of the box as shown in Figure 1(b). During this phase the box is animated so that is slowly wiggles from left to right to indicate that the color is not fixed yet. This phase ends when the user presses the right mouse button.
3. *Wait for delete:* The color and size of the box are now fixed. The user can delete the box by right double-clicking on it.

As soon as Phase 1 of a box ends, a new box can be defined. In this way there may be multiple boxes on screen, as shown in Figure 1(c). We develop an expression for each phase of the box, the lifetime of a box is then described by sequentially composing these phases. Finally, a combination of sequential and parallel composition is used to allow multiple boxes to be active at the same time. The entire code for this example can be obtained at `http://github.com/cwi-swat/monadic-frp`.

2.2 Reactive computations

The basic concept in Monadic FRP is a *reactive computation*: a monadic computation of a *single* value, which may require the occurrence of external events to continue. The type of a reactive computation is $React_g\ a$, where a is the type of the result of the reactive computation. The drawing program is created by composing the following basic reactive computations[1]:

```
mouseDown :: React_g {MouseBtn}
mouseUp   :: React_g {MouseBtn}
mouseMove :: React_g Point
deltaTime :: React_g Time
sleep     :: Time → React_g ()

type Point    = (Double, Double)    -- in pixels
data MouseBtn = MLeft | MMiddle | MRight
type Time     = Double              -- in seconds
```

Here, $mouseDown$ is a reactive computation that completes on the next mouse press by the user, and then returns the mouse buttons that are pressed. Typically this will be a single mouse button, but it may be that the user presses multiple buttons simultaneously, and hence the result is a *set* of buttons. Similarly, $mouseUp$ returns the mouse buttons that are *released* next. The reactive computation $mouseMove$ completes on the next move of the mouse, and gives the new mouse position on screen. The reactive computation $deltaTime$ reports a *change in time*: the elapsed time in seconds since the last update. How fast $deltaTime$ completes depends on the processing power available, as we will see later. Finally, $sleep$ is the reactive computation that completes after waiting the given number of seconds. The subscript $_g$ in the type of reactive computation $React_g$ indicates the set of events that the reactive computation may deal with, and will be explained in Section 4.

[1] In this paper we use $\{a\}$ to denote $Set\ a$.

Our drawing program is an expression where the above basic reactive computations are the leaves of the expression. The functions that are used to form this expression by converting, transforming and composing other expressions are shown in Figure 2. In the rest of this section, we discuss these functions and show how they are used to compose the drawing program from small components.

Reactive computations can be composed *sequentially*, yielding a new reactive computation that acts as the first reactive computation until it completes, then passes its result to a function which returns a second reactive computation, and finally acts as this second reactive computation until it completes. The function to compose reactive computations sequentially is the bind (≫=) function from the $Monad$ type class. As an example, the following defines a reactive computation that decides if the user has pressed the same mouse button(s) in succession, using do notation:

```
sameClick :: React_g Bool
sameClick = do pressed  ← mouseDown
               pressed2 ← mouseDown
               return (pressed ≡ pressed2)
```

Here the function $return$, also from the $Monad$ type class, converts a value into a reactive computation which immediately completes and returns the given value.

Another example of sequential composition is the following reactive computation, which completes when a given mouse button is pressed:

```
clickOn :: MouseBtn → React_g ()
clickOn b =
  do bs ← mouseDown
     if b `member` bs then return () else clickOn b

leftClick   = clickOn MLeft
middleClick = clickOn MMiddle
rightClick  = clickOn MRight
```

The basic function to compose reactive computations in *parallel* is $first$, whose type is listed in Figure 2. This function gives the reactive computation that runs both argument reactive computations in parallel, and completes as soon as either one of the arguments completes. The result is then the pair of the new states of both reactive computations, one of which has completed (or both when they complete simultaneously). We can use this function, for example, to create a reactive computation that given two reactive computations decides if the first completes before the second:

```
before :: React_g a → React_g b → React_g Bool
before a b = do (a′, b′) ← first a b
                case (done a′, done b′) of
                  (Just _, Nothing) → return True
                  _                 → return False
```

Where $done$ is a function that given the state of a reactive computation, returns the result of this reactive computation wrapped in $Just$ if the reactive computation is done, and $Nothing$ otherwise.

Sequential and parallel composition can be combined to form more complex expressions. For example, the following reactive computation completes when the user has double-clicked the right mouse button, where a double-click is defined as two clicks within 200 milliseconds:

Parallel composition

$first$	::	$React_g\ a$	$\to React_g\ b$	$\to React_g\ (React_g\ a, React_g\ b)$
at	::	$Sig_g\ a\ y$	$\to React_g\ b$	$\to React_g\ (Maybe\ a)$
$until$	::	$Sig_g\ a\ l$	$\to React_g\ b$	$\to Sig_g\ a\ (Sig_g\ a\ l, React_g\ b)$
$<\!\!/\!\!\!\backslash\!\!>$	::	$Sig_g\ (a \to b)\ l$	$\to Sig_g\ a\ r$	$\to Sig_g\ b\ (Sig_g\ (a \to b)\ l, Sig_g\ a\ r)$
$indexBy$	::	$Sig_g\ a\ l$	$\to Sig_g\ b\ r$	$\to Sig_g\ a\ ()$

Sequential composition

$(\gg\!=)$	::	$React_g\ b$	$\to (b \to React_g\ a)$	$\to React_g\ a$
$(\gg\!=)$	::	$Sig_g\ x\ b$	$\to (b \to Sig_g\ x\ a)$	$\to Sig_g\ x\ a$

Repetition

$repeat$	::	$React_g\ a$	$\to Sig_g\ a\ ()$
$spawn$	::	$Sig_g\ a\ x$	$\to Sig_g\ (ISig_g\ a\ x)\ ()$

Conversion

$return$	::	a	$\to React_g\ a$
$return$	::	a	$\to Sig_g\ x\ a$
$done$	::	$React_g\ a$	$\to Maybe\ a$
cur	::	$Sig_g\ a\ x$	$\to Maybe\ a$
$emit$	::	a	$\to Sig_g\ a\ ()$
$always$	::	a	$\to Sig_g\ a\ ()$
$waitFor$	::	$React_g\ a$	$\to Sig_g\ x\ a$

Transformation

map	::	$(a \to b) \to Sig_g\ a\ r \to Sig_g\ b\ r$
$scanl$	::	$(a \to b \to a) \to a \to Sig_g\ b\ r \to Sig_g\ a\ r$
$find$	::	$(a \to Bool) \to Sig_g\ a\ r \to React_g\ (Either\ a\ r)$

Parallel element composition

$dynList$	::	$Sig_g\ (ISig_g\ a\ x)\ ()$	$\to Sig_g\ [a]\ ()$

Figure 2: The types of composition, transformation and conversion functions for reactive and signal computations in Monadic FRP.

```
doubler :: React_g ()
doubler = do rightClick
             r ← rightClick `before` sleep 0.2
             if r then return () else doubler
```

2.3 Signal computations

The second concept in Monadic FRP is a *signal computation*, a reactive computation that may also *emit* values. A signal computation has type $Sig_g\ a\ b$, with two type arguments: the type of the values that it emits, a, and the type of the value that it returns, b. As the name suggests, the analogue to a signal computation in other FRP formulations is a signal. In contrast to a signal in other FRP formulations, a signal computation can *end*, yielding its result. Another way of looking at it is that a signal computation is a *fragment* of a signal.

To understand the usage of signal computations, consider a modal dialog in a GUI application: a pop-up window where the user must type his name before the program continues. We can model this pop-up window as a signal computation in the following way: The values that the signal computation emits are the descriptions of the appearance of the pop-up window. This description can be, for example, the current size of the pop-window and the text in the text field. When signal computation emits new descriptions, for example because the user enters letters, these descriptions should be processed and the resulting image should be drawn on screen. This signal computation completes when the user has finished entering his name, after which the pop-up disappears and the signal computation returns the name of the user.

A signal computation describes the lifetime of some object, such as a pop-up window. We call the values that a signal computation emits, such as the descriptions of the appearance of the pop-up window, the *form* of the signal computation, i.e. what can be observed from the outside. Each emission is an update to the form of the object. The *current form* is the last emitted value, and if a signal computation did not emit a value yet we say that it is *uninitialized*. When a signal computation ends, the object that it describes ends, and the result is the information to the rest of the program on how to continue, for example the name of the user. In contrast, a reactive computation cannot emit values, it just computes a value for use in the rest of the program.

The two basic functions to create a signal computation are $waitFor$ and $emit$. The first, $waitFor$ converts a reactive computation into a signal computation, where the resulting signal computation never emits a value (i.e. it has no form) and returns the result of the reactive computation. The second, $emit$ takes a value and gives a signal computation that emits that value and then immediately returns. Like reactive computations, signal computations can be composed sequentially using $\gg\!=$, in much the same way.

As an example, consider the signal computation that models the color of the box during the Phase 2. It emits a color at the start and after each middle mouse click, until the user presses the right mouse button, after which it returns the number of colors it emitted. This signal computation is defined as as follows:

```
cycleColor :: Sig_g Color Int
cycleColor = cc colors 1 where
  cc (h : t) i = do
    emit h
    r ← waitFor (middleClick `before` rightClick)
    if r then cc t (i + 1) else return i
```

Where $colors$ is an infinite list of colors (not shown).

Another way to create a signal computation is to *repeat* a reactive computation. The function to do this is unsurprisingly named *repeat*, and gives the signal computation that indefinitely repeats the given reactive computation, each time emitting the resulting value. This signal computation never ends, and hence its result, $()$, will never be reached. An example is the signal computation that emits the current mouse positions:

```
mousePos :: Sig_g Point ()
mousePos = repeat mouseMove
```

Signal computations can be transformed by functions such as map, $scanl$ and $find$ that are familiar from list programming. As an example, the following signal computation emits the preliminary rectangles in Phase 1 of a box, given the left-upper point of the rectangle.

```
curRect :: Point → Sig_g Rect ()
curRect p1 = map (Rect p1) mousePos
data Rect = Rect {leftup :: Point, rightdown :: Point}
```

The list function *scanl* is similar to *foldl*, but it returns a list of successive reduced values instead of a single value. The signal transformation function *scanl* works analogously, it emits a new reduced value each time the given signal emits. Using *scanl*, we define a signal that on each update, emits the number of seconds since it started:

```
elapsed :: Sig_g Time ()
elapsed = scanl (+) 0 (repeat deltaTime)
```

Using *elapsed*, we implement *animation* by transforming each point in time to the frame of the animation at that time. As an example, the following signal emits the rectangle animation in Phase 2:

```
wiggleRect :: Rect → Sig_g Rect ()
wiggleRect (Rect lu rd) = map rectAtTime elapsed
   where rectAtTime t = Rect (lu +. dx) (rd +. dx)
                          where dx = (sin (t ∗ 5) ∗ 15, 0)
```

Where $+.$ (not shown) is the vector addition operator for points.

The last list-like function that we use in our example, *find*, gives a reactive computation that completes as soon as the given signal computation emits a value on which the given predicate holds. As an example, the following function gives a reactive computation which completes as soon as the argument signal computation emits a point inside a given rectangle:

```
posInside   :: Rect → Sig_g Point y
            → React_g (Either Point y)
posInside r = find (‘inside‘r)
inside :: Point → Rect → Bool
```

Signal computations and reactive computations can be composed in parallel by two functions: *at* and *until*. The first, *at*, takes a signal computation and a reactive computation, and returns the current form of the signal computation at the time the reactive computation completes. For example, the mouse position at the next left mouse click is defined as follows:

```
firstPoint :: React_g (Maybe Point)
firstPoint = mousePos ‘at‘ leftClick
```

The second, *until*, takes a signal computation and a reactive computation, and runs the signal computation until the reactive computation completes. Like *first*, the result of l ‘*until*‘ a is the pair of the new state of l and the new state of a. For example, the following gives the preliminary rectangles in Phase 1 until the user releases the left mouse button.

```
completeRect :: Point → Sig_g Rect (Maybe Rect)
completeRect p1 = do (r, _) ← curRect p1 ‘until‘ leftUp
                     return (cur r)
```

Where *leftUp* (not shown) is defined analogously to *leftDown*. The function *cur* gives the current form of a signal computation, i.e. the last value it emitted.

By composing *firstPoint* and *completeRect* sequentially, we define the signal computation that emits the rectangles in Phase 1:

```
defineRect :: Sig_g Rect Rect
defineRect = do Just p1 ← waitFor firstPoint
                Just r  ← completeRect p1
                return r
```

The function to compose two signal computations in parallel is </\>, which takes a signal computation emitting functions and a signal computation emitting values, and gives the signal computation that emits the results obtained by feeding the values to the functions over time. More precisely, the signal computation f </\> x operates as follows:

- Wait until both input signals have started emitting values.
- On each emission from either the function signal computation or the value signal computation we apply the latest value to the latest function and emit the resulting value.
- Repeat the previous step until either of the signals end.

The result of the signal computation f </\> x is the new state of both input signal computations, one of which has ended.

We can use this operator to compose the signal computation of the rectangle and the signal computation of the color in parallel, to obtain a signal computation which describes Phase 2 of a box:

```
chooseBoxColor :: Rect → Sig_g Box ()
chooseBoxColor r =
   do always Box </\> wiggleRect r </\> cycleColor
      return ()
data Box = Box Rect Color
```

The operator </\> binds less strongly than function application. The function *always* takes a value and gives a signal computation that emits that value and then never emits again and never ends. In this way, the current form of *always* x is always x. The signal computation *chooseBoxColor* r ends when the user presses the right mouse button, as this causes *cycleColor* to end, which in turns ends the compositions using </\>.

The functions </\> and *always* are inspired by the *Applicative functor* type class [13]: the function </\> corresponds to ⊛ and *always* corresponds to *pure*. The difference is that the *Applicative* type class operates on the last argument of a type constructor, but here we want </\> to operate on the emitted arguments, i.e. the first type argument of the type constructor Sig_g. In this way Monads are used for sequential composition, and an Applicative functor-like interface is used for parallel composition.

Another interesting way to compose signal computations in parallel it to use one as a *time index* for the other. This means that we sample the form of the first signal computation each time the second signal computation emits. For instance, *mousePos* ‘*indexBy*‘ *repeat doubler* is the signal that emits the mouse positions at the times when the user right-double clicks. We can use this operator to define a reactive computation that completes as soon as the user double right clicks on a given rectangle:

```
drClickOn :: Rect → React_g (Maybe Point)
drClickOn r =
  posInside r (mousePos ‘indexBy‘ repeat doubler)
```

We now have all the ingredients to define the behavior of a single box, as we have defined each phase of the box, so we only have to compose them sequentially:

```
box :: Sig_g Box ()
box = do r ← map setColor defineRect
         chooseBoxColor r
         waitFor (drClickOn r)
         return ()
   where setColor r = Box r (head colors)
```

This signal computation describes the entire lifetime of a box, its form is appearance of the box and the signal computation ends when the user deletes the box.

2.4 Dynamic lists

We now have the signal computation for a single box, but we would like our drawing program to allow the user to draw *multiple* boxes. Luckily, signal computations are just *values*, and hence like reactive computations, they can be *repeated*. For this we introduce the function *spawn* which takes a signal computation and returns a signal computation that emits *initialized signals*: signal computations which are initialized, i.e. the first form of the object it describes is known. In this way, we can define a signal that emits initialized signals of the boxes that the user creates as follows:

$$\begin{aligned}&newBoxes :: Sig_g\ (ISig_g\ Box\ ())\ ()\\&newBoxes = spawn\ box\end{aligned}$$

This signal computation starts a box computation, and as soon as it emits its first value, $newBoxes$ emits the initialized signal corresponding to that box. Afterwards, a new box computation is started and the process repeats.

These initialized signals can then be composed *parallel*, so that there are multiple boxes on the screen, and the user can interact with all of them. For this we introduce the function $dynList$, which takes a signal computation emitting initialized signals, and composes these initialized signals in parallel. The result is a *dynamic list*: a list that changes over time. The signal computation that describes this dynamic list emits the lists of boxes, namely the current forms of all boxes that are active at that time. When a new box is defined it is added to the list and when a box is deleted, i.e. its initialized signal ends, it is removed from the list. In this way, we can define the top-level expression of our drawing program simply as:

$$\begin{aligned}&boxes :: Sig_g\ [Box]\ ()\\&boxes = dynList\ newBoxes\end{aligned}$$

2.5 Time-branching

Monadic FRP has *time-branching semantics*: we can observe the values a signal computation emits when given some event occurrences, and afterwards we can still observe what values the orignal signal computation emits when given other event occurrences. These time-branching semantics are also known as *shallow causality* [10]. They are also supported by Arrowized FRP [3], by “freezing” signal transformers.

We can use these time-branching semantics, for example, to easily implement multiple tabs in our drawing program. The user can then duplicate its current drawing into two tabs, modify the drawing and switch back to the tab holding the original drawing, which can then again be modified. Each of these tabs is described by a signal computation, but only one observes the current event occurrences. Duplication of a tab is then simply duplicating the signal computation in the list of tabs, and switching between tabs controls which tab observes the current event occurrences and is rendered to the screen. The code for this tabbed drawing program is not included in this paper for space reasons, but can be seen online. As we show in Section 6, time-branching semantics are only supported by purely functional evaluation mechanisms.

3. Comparison with other FRP programmer interfaces

In this section, we compare the Monadic FRP programmer interface to other FRP programmer interfaces. We compare mainly with Arrowized FRP [3], more precisely the Yampa [15] framework, and discuss other FRP formulations in passing.

In Arrowized FRP, signals are not first class entities: they cannot be created or manipulated directly. Instead, the basic concept in Arrowized FRP is a *signal function*: a mapping from input signal to output signal. A signal function has type $SF\ a\ b$, where a is the type of the input signal and b is the type of the output signal. Signal functions can then be composed using the $Arrow$ type-class [9]. We assume basic familiarity with this type-class and its notation [16] in the rest of this section. It should be noted that the examples in this section are cherry-picked to show the advantages of Monadic FRP and hence may give a skewed impression.

In contrast to signal computations in Monadic FRP, signals in Arrowized FRP cannot end. Another difference is that signals in Arrowized FRP must emit a value for *each* input value. For this reason, among others, Arrowized FRP has the concept of an *event source*: a signal that emits values of the option type $Event\ \ a$. An event source emits $NoEvent$ when there is no event, and an $Event\ \ a$, where a is the information associated with the event, when there is an event.

Figure 3 shows the implementation of the $cycleColor$ signal (function) in both Monadic and Arrowized FRP. In the Arrowized version, $cycleColor$ is a signal function which takes a signal producing mouse press events, and transforms it into a signal producing a color and an event of type Int. This event occurs when the user is done choosing colors, and then contains the number of different colors the user considered. Notice that when such an event occurs, the signal does not stop as in the Monadic FRP formulation of $cycleColor$, because signals cannot end.

3.1 Advantages of Monadic FRP

3.1.1 Implicit routing

The most obvious difference when considering the code in Figure 3 is the difference between do notation and arrow notation. To compose signal functions in arrow notation, the programmer needs to route the output of component arrows and the input signal into the input of component arrows and the output signal. In other FRP formulations, such as Classic FRP [7], such wiring is also necessary, but by composing functions instead of arrows. In Monadic FRP, this routing is *implicit*, reducing boilerplate code and visual clutter.

3.1.2 Easier sequential composition

Because signals in Arrowized FRP cannot end, a different approach is taken to describe signals which consist of multiple phases. For this a variety of *switching combinators* is used, which allow us to switch from one signal function to another, when a certain event occurs. The most basic switching combinator in Yampa is $switch$, which has the following type:

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

The first argument to this combinator is a signal function transforming a signal of type a into a signal giving a combination of something of type b and an event of type c. The second argument is a continuation function: given a value of type c it will produce a new signal function. The result of the $switch$ combinator is a signal function from a to b, which first behaves as the first argument signal function, except that the $Event\ \ c$ is not visible from the outside. When this first argument signal function generates an event of type c, the continuation function is called. Afterwards, the resulting signal function is *switched* to: the result of the $switch$ combinator will behave as this signal function.

In our example in Figure 3(b), the signal function $cycleColor$ is intended to be switched out when a right mouse event occurs. However, a right mouse click event does not contain the color count. For this reason, we have to set the associated data of the mouse press event to the color count, by means of the $tag :: Event\ \ a \rightarrow b \rightarrow Event\ \ b$ combinator. In Monadic FRP, such explicit transformation of the associated data of events is not necessary.

All Yampa switching combinators come in two flavors:

- *immediate*, in which case the output at the time of switching is determined by the signal function being switched to.
- *decoupled*, in which case the output at the time of switching is determined by the original signal function.

In Monadic FRP, signals can *either* end or emit a value, but not both at the same time. Hence, the distinction between immediate

```
cycleColor :: Sig_g Color Int
cycleColor = cc colors 1 where
  cc (h : t) i = do
    emit h
    r ← waitFor (middleClick `before` rightClick)
    if r then cc t (i + 1) else return i
```

(a) Monadic FRP.

```
cycleColor :: SF MouseDown (Color, Event Int)
cycleColor = cc colors 1 where
  cc (h : t) i = switch (proc md → do
    mc ← notYet ⋘ middleClick ≺ md
    rc  ← rightClick           ≺ md
    returnA ≺ ((h, tag rc i), mc)
    )(λ_ → cc t (i + 1))
```

(b) Arrowized FRP.

Figure 3: Side-by-side comparison of *cycleColor* in Monadic and Arrowized FRP.

and decoupled switching is not present in Monadic FRP, and the associated subtleties disappear.

In our example, the *notYet* combinator is used to delay the switching event by one time-step. Without an invocation of *notYet*, the program will go into an infinite loop when the middle mouse button is pressed. The reason for this is that the new signal function is again an application of *cc*, which will then immediately switch again, since the input signal currently indicates that the middle mouse button is down. In Monadic FRP, such event suppression is not necessary, because an event can only be consumed once by a reactive computation. For example, *rightClick* ≫ *rightClick* will complete after two right clicks, not one.

Another benefit of Monadic FRP is that signal computations decide themselves that they end, whereas with *switching* combinators this is decided by the context. Hence, in Arrowized FRP, if a programmer intends a signal function to be switched out after a certain event occurs, the programmer must still provide the signal function after this event. In Monadic FRP this is not necessary: the programmer can force the context to "switch".

3.1.3 A simpler way of creating dynamic lists

In Yampa, creating dynamic lists requires the following *parallel switching combinator*[2]:

```
pSwitchList :: [SF a b] → SF (a, [b]) (Event c)
               → ([SF a b] → c → SF a [b]) → SF a [b]
```

This switching combinator requires three arguments:

- The initial list of signal functions.
- A signal function that transforms the input and the current list of values to a switching event.
- A continuation function that when given the current list of signal functions and the value associated with a switching event, returns the new signal function.

The code for the dynamic list of boxes in Monadic and Arrowized FRP is shown in Figure 4. In the Arrowized FRP code, we assume that the signal function for a single box produces the current form of the box and an event indicating that the box has ended. The difficulty in creating the dynamic list then lies in wiring the switching events of all boxes and the switching event for creating a new box together, and then picking the resulting switching event information apart again in the continuation function. In Monadic FRP, such wiring is not necessary.

3.2 Disadvantages of Monadic FRP

While Monadic FRP has several advantages over other FRP formulations, it also has some disadvantages. In particular, to share the

```
boxes :: Sig_g [Box] ()
boxes = dynList (spawn box)
```

(a) Monadic FRP.

```
type BoxSF = SF GUIIn (Box, Event ())
boxes :: SF GUIIn [Box]
boxes = boxes′ [] ⋙ arr (map fst) where
  boxes′ i = pSwitchList i
    (newBox *** arr toEv ⋙ arr choose ⋙ notYet)
    (λe l → boxes′ (mutateList e l))
choose (a, b) = merge (fmap Left a) (fmap Right b)
toEv l = let l′ = map (isNoEvent ∘ snd) l
         in if and l′ then NoEvent else Event l′
mutate :: [BoxSF] → Either (BoxSF) [Bool] → [BoxSF]
mutate l (Left b) = b : l
mutate l (Right l′) = map fst (filter snd (zip l l′))
box    :: BoxSF
newBox :: SF GUIIn (Event BoxSF)
```

(b) Arrowized FRP.

Figure 4: Dynamic list in the drawing program.

computation of a signal which occurs more than once in an expression, we have to resort to a manual invocation of a memoization function. This is not necessary in several other FRP formulations, including Arrowized FRP.

A related disadvantage is that it is unclear how to declare mutually dependent signals in Monadic FRP, such as two sliders in a temperature conversion application that influence each other. In Arrowized FRP, such mutually dependent signals can simply be declared by recursive arrow notation.

4. Evaluating Monadic FRP expressions

In this section, we show how reactive and signal computations are evaluated in a simple, straightforward manner.

4.1 Event requests and occurrences

Central to Monadic FRP evaluation is the notion of an event: a stimulus from the environment. Reactive computations *request* the observation of such events, an interpreter then observes such events and passes the event *occurrence* back. We model event requests and occurrences with the following data type:

```
data Event a = Request | Occurred a
```

[2] This switching combinator is the list version of *pSwitch*. We have chosen to use this specialized combinator in the comparison, since *dynList* also deals with lists.

Where the argument to the constructor *Occurred* is the associated data of the occurred event. For simplicity, event requests and occurrences are defined using the same data type in our approach.

To make things more concrete, the following events are used in the program drawing example:

$$\begin{array}{lll}\mathbf{data}\ \mathit{GUIEv} = & \mathit{MouseDown} & (\mathit{Event}\ \{\mathit{MouseBtn}\}) \\ \mid & \mathit{MouseUp} & (\mathit{Event}\ \{\mathit{MouseBtn}\}) \\ \mid & \mathit{MouseMove} & (\mathit{Event}\ \mathit{Point}) \\ \mid & \mathit{DeltaTime} & (\mathit{Event}\ \mathit{Time}) \\ \mid & \mathit{TryWait}\ \mathit{Time} & (\mathit{Event}\ \mathit{Time}) \\ & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Ord}) & \end{array}$$

When a reactive computation, for example, wants to know the next mouse button that is pressed, it passes the event request *MouseDown Request* to the interpreter of the reactive expression. This interpreter, from now on called the *reactive interpreter*, then waits for the next mouse press and returns the event occurrence, for example $\mathit{MouseDown}\ (\mathit{Occurred}\ \{\mathit{MLeft}, \mathit{MMiddle}\})$, which indicates that the user pressed the left and middle mouse buttons simultaneously.

The reactive interpreter can wait for multiple events in parallel, and hence we pass a *set* of event requests to it. As soon as at least one of these events occurred, the reactive interpreter responds by returning the occurred event(s). This response is a set of event occurrences, since multiple events may occur simultaneously. Since event requests and occurrences are modeled by the same datatype, we use the following type aliases to make the distinction clear:

$$\begin{array}{ll}\mathbf{type}\ \mathit{EvReqs}\ e = \{e\} & \text{-- event requests} \\ \mathbf{type}\ \mathit{EvOccs}\ e = \{e\} & \text{-- event occurrences}\end{array}$$

4.2 Reactive computations

Using this basic terminology introduced above, a state of a reactive computation is defined as follows:

$$\begin{array}{l}\mathbf{data}\ \mathit{React}\ e\ a \\ \quad = \mathit{Done}\ a \\ \quad \mid \mathit{Await}\ (\mathit{EvReqs}\ e)\ (\mathit{EvOccs}\ e \to \mathit{React}\ e\ a)\end{array}$$

If a reactive computation is done, it is in state *Done* and carries the resulting value of the computation of type a. Otherwise, it *awaits* at least one event occurrence from its set of event requests. As soon as one of these events occur, or multiple events occur simultaneously, the event occurrences can be passed to the *continuation function*. This continuation function then processes the event occurrences and returns the new state of the reactive computation. The type e is the type of the events that the reactive computation may request and process. In our drawing program in Section 2, the type of events is GUIEv, hence the type React_g that is used throughout Section 2 is defined as follows:

$$\mathbf{type}\ \mathit{React}_g = \mathit{React}\ \mathit{GUIEv}$$

The basic reactive computations $\mathit{mouseDown}$, $\mathit{mouseUp}$, $\mathit{mouseMove}$, $\mathit{deltaTime}$ and $\mathit{tryWait}$ are then defined as follows:

$$\begin{array}{l}\mathit{mouseDown} = \mathit{req}\ (\mathit{MouseDown}\ \mathit{Request}) \gg\!= \mathit{get} \\ \quad \mathbf{where}\ \mathit{get}\ (\mathit{MouseDown}\ (\mathit{Occurred}\ s)) = \mathit{return}\ s \\ \quad \vdots \\ \mathit{tryWait}\ t = \mathit{req}\ (\mathit{TryWait}\ t\ \mathit{Request}) \gg\!= \mathit{get} \\ \quad \mathbf{where}\ \mathit{get}\ (\mathit{TryWait}\ \_\ (\mathit{Occurred}\ t)) = \mathit{return}\ t \\ \mathit{req} :: e \to \mathit{React}\ e\ e \\ \mathit{req}\ a = \mathit{Await}\ (\mathit{singleton}\ a)\ (\mathit{Done} \circ \mathit{head} \circ \mathit{elems})\end{array}$$

Here, *req* is a function that given an event request gives the reactive computation that returns the next event occurrence that satisfies this request. The function *elems* converts a set to a list. Notice that the continuation function of a reactive computation is called with the set of event occurrences *which it awaits*. If there are no event occurrences which the reactive computation awaits, then the continuation function will *not* be called. Since $\mathit{mouseDown}$ awaits only $\mathit{MouseDown}$ events, we can be sure that the pattern match $\mathit{MouseDown}\ (\mathit{Occured}\ s)$ cannot fail. The same reasoning holds for the patterns in the other basic reactive computations.

4.3 Evaluating reactive computations

In essence, our evaluation model is a purely functional way to use *blocking-IO multiplexing*: the program is organized as a main loop that first decides which events should be listened for, then waits for at least one of these events to occur, and finally processes the event(s) that occurred. Waiting for several events in parallel can be done by means of for example the Unix `select` or Linux `epoll` method[3], which take a set of file-descriptors and waits for one of them to become ready for reading or writing. Another example is the `waitEvent` method of the Simple Directmedia Layer[4], which waits for a user input event, such as a mouse-click or keystroke. The main loop in our approach is the reactive interpreter which interprets the top-level reactive or signal computation.

The interpreter for reactive computations is defined as follows:

$$\begin{array}{l}\mathit{interpret} :: \mathit{Monad}\ m \Rightarrow (\mathit{EvReqs}\ e \to m\ (\mathit{EvOccs}\ e)) \\ \qquad\qquad \to \mathit{React}\ e\ a \to m\ a \\ \mathit{interpret}\ p\ (\mathit{Done}\ a) \quad\ \ = \mathit{return}\ a \\ \mathit{interpret}\ p\ (\mathit{Await}\ r\ c) = p\ r \gg\!= \mathit{interpret}\ p \circ c\end{array}$$

Here p is a function that takes a set of event requests and waits for any of these events to occur in the monad m, which is for example the *IO* monad. The drawing program described in Section 2 can be run in an interpreter which uses the `waitEvent` method of the Simple Directmedia Layer to define the function p. After an interesting event occurred, p returns the set of event occurrences, which is then fed back into the reactive computation. This process continues until the reactive computation completes and returns a value.

The reactive computation that is interpreted consists solely of the sequential and parallel composition of basic reactive computations, other composition operators are defined in terms of these two composition operators. As an example, consider the following reactive expression:

$$\begin{array}{l}\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp}) \\ \qquad (\mathit{mouseDown} \gg \mathit{deltaTime})\end{array}$$

Figure 5(a) shows the tree of this expression and which event requests are propagated *upwards* to the reactive interpreter. When composing reactive computations sequentially, using $\gg\!=$, the event requests of the composed expression are just the event requests of the first argument. Hence, the event requests of $\mathit{mouseDown} \gg \mathit{deltaTime}$ are just $\{\mathit{MouseDown}\}$. When composing reactive computations in parallel, using *first*, the event requests of the composed expression are the union of the event requests of both arguments. In this way the reactive interpreter knows exactly which events to wait for.

The reactive interpreter then waits for events from such a set of event requests. When one event occurred, or multiple events occurred simultaneously, the set of event occurrences is passed to the continuation function of the reactive computation. If the reactive computation is a sequential composition, then the event occurrences are simply passed to the first argument. When the reactive computation is a parallel composition, the set of event

[3] For more information see `man select` or `man epoll`.

[4] `http://libsdl.org`

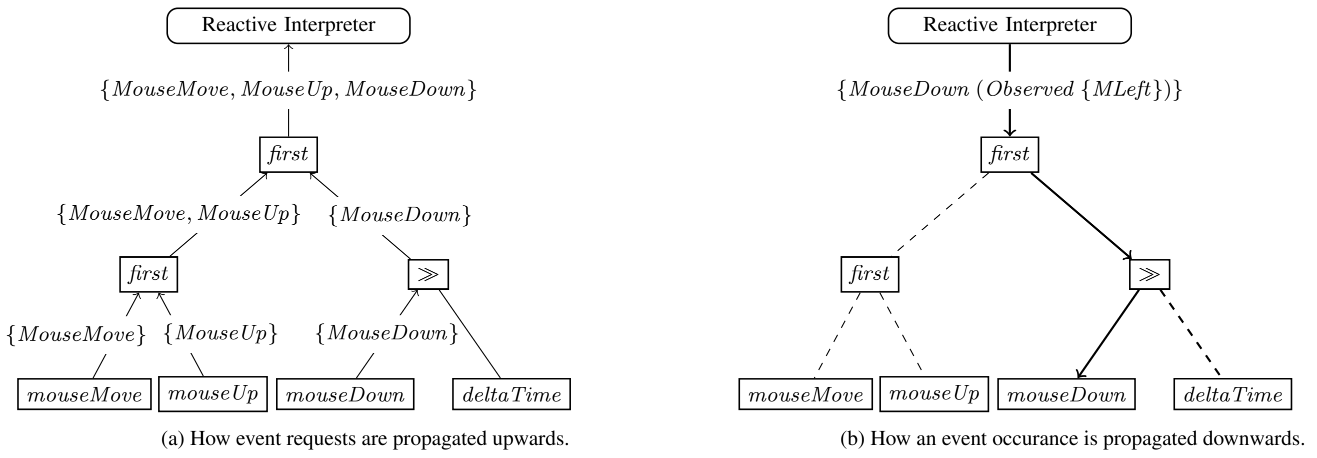

Figure 5: The tree of the expression $\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp})\ (\mathit{mouseDown} \gg \mathit{deltaTime})$.

occurrences is passed to the argument(s) that await any of these events.

Figure 5(b) shows how an event occurrence, stating that the left mouse button was pressed, is propagated *downwards*. Notice that the entire left leg of the tree is not updated in this process, since it did not await this particular event. In this way, the evaluation avoids unnecessary re-computations, by updating only those components that await the occurred events.

After processing this event occurrence, the reactive computation proceeds as the reactive computation:

$$\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp})\ \mathit{deltaTime}$$

Hence, the reactive expression is *dynamic*: each sub-expression may change after each update. This new expression leads to different event requests than the original expression, namely the set $\{MouseMove, MouseUp, DeltaTime\}$. In this way the events in which the reactive computation is interested in can also change over time.

4.4 Time semantics

In our set of GUI events, there are two events that deal with time: $DeltaTime$ and $TryWait$. The first, $DeltaTime$, asks to observe *any* change in time and returns the change in time since the previous update of the reactive interpreter. The second, $TryWait$, works similarly, but takes an argument that indicates the time it wants to wait. The result of such a $TryWait$ request is also the change in time since the last update of the reactive interpreter. The difference between the two lies in how they are handled: $DeltaTime$ tells the reactive interpreter to respond as quickly as possible, whereas $TryWait$ tells the reactive interpreter to try and wait the given time before responding. Hence, when only $TryWait$ requests are given to the reactive interpreter, then the reactive interpreter just waits for time to pass, without wasting CPU cycles needlessly updating the reactive expression.

An event request $TryWait$ asks the reactive interpreter to wait for the given time, but there may be another event request that can be answered earlier. In that case, the interpreter cannot wait the given time and must respond. Hence, the time it takes for the event $TryWait$ to occur might be less than the requested amount of time.

As an example usage of $TryWait$, consider the *sleep* reactive computation, which completes after the given number of seconds:

$$\begin{aligned} \mathit{sleep}\ t = \mathbf{do}\ & t' \leftarrow \mathit{tryWait}\ t \\ & \mathbf{if}\ t' \equiv t\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ \mathit{sleep}\ (t - t') \end{aligned}$$

Notice that testing for equality here is safe, because the result of $TryWait$ request may be less than the requested time, but not more. Hence, we can be sure that $\mathit{sleep}\ 1.1$ never completes earlier or simultaneously to $\mathit{sleep}\ 1$. In other purely functional implementations of FRP, such exact timing is not available: testing for equality on time is unsafe, since the precision of timing depends on how often the signal is sampled.

Such exact timing is achieved by handling the event requests in the reactive interpreter as follows:

- Compute the maximum time to wait, which is the minimum of the times given to $TryWait$ event requests. It is infinity if there are no $TryWait$ requests.
- See if the maximum time to wait, t, is smaller than the time since the last update, t'. If so, we construct only $TryWait$ and $DeltaTime$ occurrences with t as their associated data and return them, the other steps will not be executed on this iteration. The next update will have time difference $t' - t$ plus the new time difference. In this way, the result of a $TryWait$ request will never be more than the requested time.
- Otherwise, wait for an event from the set of event requests, using the maximum time to wait as a *timeout* duration. Blocking I/O multiplexing functions such as `select` and SDL's `waitEvent` usually allow such a timeout duration. If there is a $DeltaTime$ request, then 0 is passed as the time duration, and the events that are currently available will be returned, i.e. the blocking I/O multiplexing function will not block.
- Construct and return the set of event occurrences, including the occurrences of $TryWait$ and $DeltaTime$, which get the time since the last update of the reactive interpreter.

Thanks to these semantics, the drawing program will simply wait for the next mouse click or mouse move when there are no animated boxes currently on screen. If one of the boxes is animated (in Phase 2), then the reactive interpreter updates the animation as quickly as possible so that the animation is as smooth as possible. In this way, the animation is *conceptually continuous*: we describe it as if the animation is continuous, abstracting from how often the animation is actually sampled.

4.5 Evaluating signal computations

Evaluation of a signal computation is very much the same as evaluation of a reactive computation, since signal computations are defined in terms of reactive computations as follows:

```
newtype Sig  e a b = Sig (React e (ISig e a b))
data    ISig e a b = a :| Sig e a b
                   | End b
```

Here the type *Sig* is the type of a signal computation and *ISig* is the type of an initialized signal, i.e. a signal computation of which the first form is known or which has already ended. Signal computations and initialized signals are defined mutually recursively, a signal computation is a reactive computation of the initialized signal, and the tail of an initialized signal is again a signal computation. The argument e is the type of events that can be handled inside the signal computation, a is the type of the values that it emits and b is the type of its result. The signal computation and initialized signals in Section 2 are specialized to *GUIEv*, i.e.:

```
type Sig_g  = Sig GUIEv
type ISig_g = ISig GUIEv
```

The interpreter for signal computations uses the interpreter for reactive computations to evaluate a signal computation to its corresponding initialized signal. Additionally, the values that are emitted by the signal computation are processed. For example, the interpreter of our example in Section 2 draws each emitted list of boxes on screen. The signal computation interpreter is defined as follows:

```
interpretSig :: Monad m ⇒ (EvReqs e → m (EvOccs e))
             → (a → m ()) → Sig e a b → m b
interpretSig p d = interpretSig' where
  interpretSig' (Sig s)    = interpret p s >>= interpretISig
  interpretISig (h :| t)   = d h >> interpretSig' t
  interpretISig (End a)    = return a
```

Here the new argument d is the function which processes each new emission of the signal computation.

4.6 Sharing computation results

If a reactive or signal computation occurs multiple times in an expression, then standard evaluation techniques may lead to a source of inefficiency. The simplest example of this is:

```
first x x
```

When an event occurs that x is interested in, then the evaluation of x to its new state will be performed twice. To solve this problem we introduce a *memoization* function, as is also done in other FRP approaches [5]:

```
memo :: Ord e ⇒ React e a → React e a
```

In this way, we can rewrite our example to eliminate the potential problem:

```
let x' = memo x in first x' x'
```

We also introduce a memoization function for signal computations, that applies memoization on the reactive computation of the initialized signal *and* on the signal computation that is the tail of that initialized list (if any).

```
memoSig :: Ord e ⇒ Sig e a b → Sig e a b
```

The need for invocations of memoization functions is not necessary in some other FRP approaches, such as Arrowized FRP. Hence, this is a disadvantage of Monadic FRP.

5. Implementing Monadic FRP composition functions

In this section, we show how a selection of the composition functions from Figure 2 are implemented. In this way, this section shows the semantics of the programming model explained in Section 2 by building on the basic evaluation mechanism explained in Section 4. The definition of the composition operators is mostly straightforward: the entire Monadic FRP library consists of just 137 lines of code, excluding blank lines (not including the drawing program which consists of 108 lines of code and the interface to SDL which consists of 109 lines of code). The structure of this section reflects the structure of Section 2. We first show the implementation of sequential and parallel composition of *reactive* computations. Afterwards, we show how these can be used to implement composition functions for signal computations, and finally we show how dynamic lists are implemented.

5.1 Basic composition operators

The basic composition operators in Monadic FRP are the *sequential* and *parallel* composition of *reactive computations*, all other composition and transformation operators are defined using these two basic composition operators.

5.1.1 Sequential composition of reactive computations

Sequential composition of reactive computations is defined as an instance of the Monad type class:

```
instance Monad (React e) where
  return            = Done
  (Await e c) >>= f = Await e (λx → c x >>= f)
  (Done v)    >>= f = f v
```

If the first reactive computation awaits some event, then its next state is again sequentially composed with f. This process repeats until the first reactive computation completes, after which the function f will be called with the result of the reactive computation, and the new reactive computation will be executed.

5.1.2 Parallel composition of reactive computations

Recall that parallel composition of reactive computations is achieved using *first*, which runs two reactive computations in parallel until either completes, and then gives the new state of both reactive computations. Its definition is as follows:

```
first l r = case (l, r) of
  (Await el _, Await er _) →
    let e   = el `union` er
        c b = first (update l b) (update r b)
    in Await e c
  _ → Done (l, r)
```

If both reactive computations await some event, then *first* waits for the union of their event requests, as shown in Figure 5(a). Then, on an event occurrence, *first* updates both reactive computations to their next state and calls *first* again, which then checks again if both reactive computations await some event. If this is not the case, then at least one of the reactive computations must have completed, and the state of both reactive computations is returned.

As shown in Figure 5(b), only those reactive computations that await an event that occurred should be updated. This is done by the function *update* that is used in the above definition of *first*. This function returns the new state of a reactive computation given a set of event occurrences. If the reactive computation awaits some of the events that occurred, then *update* obtains the new state of a reactive computation by calling its continuation function. Otherwise, the new state is simply the old state. The definition of *update* is as follows:

```
update :: Ord e ⇒ React e a → EvOccs e → React e a
update (Await r c) oc | oc' ≢ empty = c oc'
  where oc' = oc `filterOccs` r
update r _ = r
```

Here, *filterOccs*(not shown) filters the event occurrences that the reactive computation awaits from the set of event occurrences. If the resulting set of event occurrences is empty, then the reactive computation did not await in any of the events that occurred.

5.2 Signal computation composition functions

Since signal computations are defined in terms of reactive computations, the composition functions dealing with signal computations are implemented by combining the sequential and parallel composition of reactive computations in various ways. Figure 6 shows the definition of a selection of these signal computation composition functions and Figure 7 shows the definition of the conversion functions.

Signal computations and initialized signals are mutually recursively defined data types, so functions dealing with signal computations often alternate between processing a signal computation and processing an initialized signal. In the code this can be seen, for example, in the function *map*, which obtains the initialized signal and then calls *imap*, which is like *map*, but on initialized signals. The function *imap* processes the initialized signal, and calls *map* again to process the tail, which is a signal computation. The same pattern arises in the sequential composition of signal computations, and in the functions *scanl*, *until* and *res*.

The signal computation *l* ‘*until*‘ *a*, splits the signal computation *l* in two: *l* ‘*until*‘ *a* is the part of the signal computation before *a* completes, and the result of *l* ‘*until*‘ *a* is the signal computation after *a* completed. If the signal computation *l* was initialized before *a* occurred, i.e. it had already emitted its first value, then the signal computation after *a* should *not* be an uninitialized signal computation. For instance, the result of *mousePos* ‘*until*‘ *leftClick*, the mouse position *after* the left click, should not be uninitialized, but should start with the emission of the last mouse position before the left click. Hence, the result of *until* differs depending on if the signal computation was initialized before the reactive completes. If this is this case, then *iuntil* ensures that the signal computation after the reactive computation completes starts with the last emission before the reactive computing completed.

To implement the parallel composition operator, <^>, we introduce another function, *pairs*, which takes two *initialized signals* as arguments and gives the initialized signal that emits the pairs of both arguments. The head of *pairs l r* is the pair of the head of *l* and the head of *r*. On each new emission of *l* or *r*, *pair l r* emits the pair of the current form of *l* and the current form of *r*. To achieve this, we first wait for the reactive computation of the tail of one of the initialized signals to complete and then update both initialized signals. This is done using the function *lup*: if the tail has not emitted a value yet, the initialized signal is the head of the old initialized signal followed by the new state of the computation of the tail. If the tail already emitted a value, the initialized signal is simply that tail. The function <^> is then implemented by first waiting for both signal computation to start emitting values, and then applying the second element to the first element of each pair.

5.3 Dynamic lists

The signal functions from the previous section can be used to define *dynList*, which takes a signal computation emitting initialized signals, and composes them in parallel. For this, we first define a dynamic variant of cons (:), that takes an initialized signal that has as form type something of type a (the head), and an initialized signal that emits something of type $[a]$ (the tail) and returns the result of "consing" the head to the lists from the tail over time:

```
cons :: Ord e ⇒ ISig e a l → ISig e [a] r
                → ISig e [a] ()
cons h t = do (h, t) ← imap (uncurry (:)) (pairs h t)
              imap (:[]) h
```

Sequential composition

```
instance Monad (Sig e a) where
  return = emitAll ∘ End
  (Sig l) ≫= f = Sig (l ≫= ib)
    where ib (h :| t)   = return (h :| (t ≫= f))
          ib (End a) = let Sig x = f a in x
instance Monad (ISig e a) where
  return = End
  (End a) ≫= f = f a
  (h :| t)   ≫= f = h :| (t ≫= emitAll ∘ f)
```

Repetition

```
repeat :: React e a → Sig e a ()
repeat x = xs where xs = Sig (liftM (:| xs) x)
spawn :: Sig e a r → Sig e (ISig e a r) ()
spawn (Sig l) = repeat l
```

Transformation

```
map :: (a → b) → Sig e a r → Sig e b r
map  f (Sig l)   = Sig (liftM (imap f) l)
imap f (h :| t)   = f h :| map f t
imap f (End a) = End a
scanl :: (a → b → a) → a → Sig e b r → Sig e a r
scanl  f i l          = emitAll (iscanl f i l)
iscanl f i (Sig l) = i :| (waitFor l ≫= lsl)
  where lsl (h :| t)   = scanl f (f i h) t
        lsl (End a) = return a
```

Parallel composition

```
until :: Ord e ⇒    Sig e a r → React e b →
             Sig e a (Sig e a r,    React e b)
until (Sig l) a = waitFor (first l a) ≫= un where
  un (Done l, a) = do (l, a) ← emitAll (l ‘iuntil‘ a)
                      return (emitAll l, a)
  un (l, Done a) = return (Sig l, Done a)
iuntil :: Ord e ⇒ ISig e a r →        React e b →
           ISig e a (ISig e a r,         React e b)
iuntil (End l)      a = End (End l, a)
iuntil (h :| Sig t)  a = h :| Sig (liftM cont (first t a))
  where cont (Done l, a) = l ‘iuntil‘ a
        cont (t, Done a) = End (h :| Sig t, Done a)
(<^>) :: Ord e ⇒ Sig e (a → b) l → Sig e a r →
           Sig e b ( Sig e (a → b) l,    Sig e a r)
l <^> r = do (l, r) ← waitFor (bothStart l r)
             emitAll (imap (uncurry ($)) (pairs l r))
bothStart ::    Ord e ⇒  Sig   e a l → Sig   e b r →
                React e (  ISig e a l,    ISig e b r)
bothStart l (Sig r) =    do (Sig l, r) ← res (      l ‘until‘ r)
                            (Sig r, l) ← res (Sig r ‘until‘ l)
                            return (done′ l, done′ r)
pairs ::  Ord e ⇒        ISig e a l →      ISig e b r →
           ISig e (a, b) (ISig e a l,        ISig e b r)
pairs (End a)     b           = End (End a, b)
pairs a               (End b) = End (a, End b)
pairs (hl :| Sig tl) (hr :| Sig tr) = (hl, hr) :| tail
  where tail = Sig (liftM cont (first tl tr))
        cont (tl, tr) = pairs (lup hl tl) (lup hr tr)
        lup _ (Done l) = l; lup h t = h :| Sig t
```

Figure 6: Implementation of sequential composition, repetition, transformation and parallel composition functions.

$$
\begin{array}{l}
\mathit{emitAll} \;\; = \mathit{Sig} \circ \mathit{Done}; \mathit{emit}\ a = \mathit{emitAll}\ (a :\mid \mathit{return}\ ())\\
\mathit{always}\ a = \mathit{emit}\ a \gg \mathit{hold}; \mathit{waitFor}\ a = \mathit{Sig}\ (\mathit{liftM}\ \mathit{End}\ a)\\
\mathit{hold} \qquad = \mathit{waitFor}\ \mathit{never}\ \mathbf{where}\ \mathit{never} = \mathit{Await}\ \mathit{empty}\ \bot\\
\mathit{res}\ (\mathit{Sig}\ l)\ = l \ggg\!= \mathit{ires}\\
\mathit{ires}\ (\_ :\mid t) = \mathit{res}\ t; \mathit{ires}\ (\mathit{End}\ a) = \mathit{Done}\ a\\
\mathit{done}\ (\mathit{Done}\ a) \qquad\qquad = \mathit{Just}\ a; \mathit{done}\ \_ = \mathit{Nothing}\\
\mathit{cur}\ (\mathit{Sig}\ (\mathit{Done}\ (h :\mid \_))) = \mathit{Just}\ h; \mathit{cur}\ \_ \;\; = \mathit{Nothing}\\
\mathit{done}' = \mathit{fromJust} \circ \mathit{done}
\end{array}
$$

Figure 7: Implementation of conversion functions.

```
            $t$
            $return\ ()$
```

The initialized signal $\mathit{pairs}\ h\ t$ gives the pairs of the head and tail over time. Hence, if we transform these pairs so that the head is prepended to the tail, we get the list over time. After step $\mathit{pairs}\ h\ t$, either the head or the tail has ended. We then emit the residual values of the head and the tail, one of which is empty. In this way, if we are given two an initialized signals a and b of the same type and an initialized signal emitting lists of that type, c, then $a\ `\mathit{cons}`\ (b\ `\mathit{cons}`\ c)$ will emit lists of the current states of a,b and c. If an an initialized signal ends, it has no current form and it will not be included in the list. For example, if b ends before a and c, then we will continue as $a\ `\mathit{cons}`\ c$.

To define $\mathit{dynList}$, we start with the empty dynamic list, i.e. the initialized signal list that always has as current form the empty list. We then run this initialized signal until the argument of $\mathit{dynList}$ emits a new initialized signal. Then, we prepend this new initialized signal to the current dynamic list to obtain the new dynamic list. Afterwards, we run this dynamic list until the argument emits another initialized signal and the process repeats.

```
$dynList\ x = emitAll\ (idynList\ x)$
$idynList :: Ord\ e \Rightarrow Sig\ e\ (ISig\ e\ a\ l)\ r \to ISig\ e\ [a]\ ()$
$idynList\ l = rl\ ([\,] :\mid hold)\ l \gg return\ ()$ where
  $rl\ t\ (Sig\ es) =$ do $(t, es) \leftarrow t\ `iuntil`\ es$
                     case $es$ of
                       $Done\ (e :\mid es) \to rl\ (cons\ e\ t)\ es$
                       $\_ \qquad\qquad\quad \to t$
```

6. Comparison with other FRP evaluation schemes

To implement reactive systems, one needs a basic mechanism to deal with events that occur over time. We identify four such mechanisms:

- Busy waiting
- Blocking I/O multiplexing
- Concurrency
- Callback networks

For each of these basic mechanisms there exists one or multiple corresponding FRP evaluation mechanisms. Our approach is the only one which uses blocking I/O multiplexing. In the following subsections we will discuss FRP evaluation schemes for each of these other basic mechanisms.

6.1 Busy waiting

The original FRP formulation [7] and Arrowized FRP [3] use an implementation which models signals as functions, which given an amount of time and input values return the pair of their current emission and their continuation function. Since in this approach signals do not communicate *which* events they are interested in, the *entire* signal expression must be evaluated on each update, including the parts for which the input did not change. The reactive interpreter does not know which events to wait for and is hence in a busy waiting loop, constantly calling the signal continuation function with the new time and possibly interesting event occurrences. Since this continuation-based implementation of signals is purely functional, it allow time-branching signals.

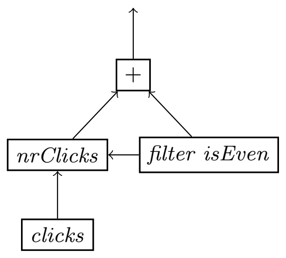

Figure 8: A simple signal dependency network.

6.2 Concurrency

A second basic mechanism is to use *concurrency* in the form of multiple parallel threads or processes. Elliot [6] gives a FRP evaluation scheme which avoids unnecessary re-computations based on the following observation: if we know the order in which the events arrive *in advance*, then we could just use *blocking I/O* to implement FRP. He then introduces the concept of *unambiguous choice*: given two ways to compute the *same* value using blocking I/O, we can start both computations in parallel, see which one completes first, kill the other and use the result. This approach does not allow time-branching semantics, because the intermediate states of signals are simply not accessible as values.

6.3 Callback networks

The typical way to implement FRP using callbacks networks is to organize the system in a directed acyclic graph, where the nodes are signals and there is an edge between two signals if one depends on the other. Signals can then notify other signals if they update their value (i.e. emit). Variants of this basic model are used in many FRP systems, such as Scala.React [12], FrTime for Racket [1], Frappe for Java [2], and Microsoft's Reactive Extensions (Rx)[5].

As an example of such a network consider the following simple (Monadic) FRP expression:

```
let $nrClicks = memo\ (scanl\ (+)\ 0\ clicks)$
in $always\ (+) \mathbin{<\!\!\wedge\!\!>} nrClicks \mathbin{<\!\!\wedge\!\!>} filter\ isEven\ nrClicks$
```

The dependency network of this expression is shown in Figure 8. As an example reduction, suppose that the current value of $\mathit{nrClicks}$ is 3 and the value of $\mathit{filter}\ \mathit{isEven}\ \mathit{nrClicks}$ is 2. Suppose then that the user presses a button, which will cause the signal clicks to update. This signal then calls $\mathit{nrClicks}$, which depends on it. The signal $\mathit{nrClicks}$ then updates its value to 4 and calls the signals that depend on it. If it calls $\mathit{filter}\ \mathit{isEven}$ first, then that also updates its value to 4 and calls $+$, which will then update its value to 8. However, if $\mathit{nrClicks}$ calls $+$ before $\mathit{filter}\ \mathit{isEven}$, then $+$ will use a stale value of $\mathit{filter}\ \mathit{isEven}$, namely 2, and incorrectly update its value to 6.

An incorrect update due to the order of calls in the signal network, such as the update of $+$ to 6, is called a *glitch*. Most FRP systems based on callback networks use a glitch prevention system. The exception is Rx, which does not prevent glitches (according to [12]). The most common way to prevent glitches, is not to let signals call each other directly but instead place their calls in a

[5] https://rx.codeplex.com/

priority queue [1, 12]. This priority queue schedules updates of nodes according to the topological ordering of the directed acyclic graph, which ensures that no glitches will occur.

A complication is then that the topology of the signal network may change dynamically and hence the system needs to maintain a topological ordering of the evolving directed acyclic graph. Another complication is that to prevent needless computations, we would like to prevent scheduling updates to signals that no other signal depends on. A non-solution is to use weak references for dependence links and then rely on the garbage collector to collect the dead signals. It may be a while before dead signals are collected, and during this time needless computations are possible. Hence there needs to be some form of instant garbage collection, for example reference counting. For more information on possible solutions for these complications see for example [1, 12] or [2].

The difference between Monadic FRP and a glitch-free callback network based FRP system is that in Monadic FRP the events come in at the root of the expression and evaluation proceeds *top-down*, whereas in glitch-free callback network based FRP events arrive at the leaves and evaluation proceeds *bottom-up*. In Monadic FRP there is no way to create a glitch, as the expression itself *is* the ordering on signals. Since Monadic FRP traverses the signal network in top-down fashion, signals that no other signal depends on will never be computed, and are collected by ordinary garbage collection.

Another difference is that callback-based FRP systems use the signal network as *mutable data*, whereas in Monadic FRP the signal network is *immutable*, i.e. the next network is a new signal network, not a modification of the old network. This is the reason that time-branching operations are possible in purely functional evaluation models such as that of Monadic FRP and Arrowized FRP, and are impossible in callback-based systems.

7. Conclusion and Future work

In this paper we introduced Monadic Functional Reactive Programming, an alternative programming model and evaluation mechanism for FRP. The basic notion in Monadic FRP is a reactive computation, a monadic computation which may require the occurrence of external events to continue. A signal computation is a reactive computation that may also emit values. In contrast to signals in other FRP formulations, signal computations can end. This leads to a monadic interface for sequencing signal phases, which is arguably more intuitive and flexible than the switching combinators found in other FRP approaches. This also allows us to define dynamic lists, lists that change over time, more easily than in other FRP approaches. In contrast to other FRP approaches, Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing redundant re-computations.

This gives rise to several directions for further research:

- How can mutually depended signals be expressed in Monadic FRP?
- Arrowized FRP does not require manual invocations of memoization functions like Monadic FRP and makes it possible to define mutually dependent signals. We are currently investigating whether it is possible to combine Monadic FRP and Arrowized FRP into a single framework that has the best of both worlds.
- How can Monadic FRP be formulated in a dependently typed setting, allowing us to statically rule out more meaningless and incorrect programs in the style of Sculthorpe and Nilsson [20]?
- How can Monadic FRP be integrated with a declarative graphics library, such as our previous work [11]?

Acknowledgments

We thank Chung-chieh Shan and the anonymous reviewers for their great help on this paper.

References

[1] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. of the '06 European Symposium on Programming*, pages 294–308, 2006.

[2] A. Courtney. Frappé: Functional reactive programming in Java. In *Proc. of the '01 International Symposium of Pratical Aspects of Declarative Languages*, March 2001.

[3] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Proc. of the '01 Haskell Workshop*, September 2001.

[4] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proc. of the '03 Haskell Workshop*, pages 7–18, Aug. 2003.

[5] C. Elliott. Functional implementations of continuous modeled animation. In *Proc. of the 10th International Symposium on Principles of Declarative Programming*, pages 284–299, 1998.

[6] C. Elliott. Push-pull functional reactive programming. In *Proc. of the '09 Haskell Symposium*, 2009.

[7] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of the 1997 International Conference on Functional Programming*, ICFP '1997, pages 163–173, June 1997.

[8] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *'02 Summer School on Advanced Functional Programming*, pages 159–187, 2003.

[9] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000.

[10] A. S. A. Jeffrey. Causality for free!: Parametricity implies causality for functional reactive programs. In *Programming Languages meets Program Verification*, PLPV '13, 2013.

[11] P. Klint and A. van der Ploeg. A library for declarative resolution-independent 2d graphics. In *Proc. of the '13 International Symposium on Practical Aspects of Declarative Languages*, PADL '13, 2013.

[12] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, LAMP, 2012.

[13] C. Mcbride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, Jan. 2008.

[14] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proc. of the '09 Conference on Object oriented programming systems languages and applications*, pages 1–20, 2009.

[15] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. of the '02 Haskell Workshop*, pages 51–64, Oct. 2002.

[16] R. Paterson. A new notation for arrows. In *Proceedings of the '01 international conference on Functional programming*, pages 229–240, 2001. ISBN 1-58113-415-0.

[17] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proc. of the 1999 International Conference on Robotics and Automation*, 1999.

[18] J. Peterson, P. Hudak, and C. Elliott. Lambda in Motion: Controlling robots with Haskell. In *Proc. of the 1th International Workshop on Practical Aspects of' Declarative Languages*, PADL 1999, January 1999.

[19] J. Peterson, P. Hudak, A. Reid, and G. Hager. FVision: A declarative language for visual tracking. In *Proc. of the '01 International Workshop on Practical Aspects of Declarative Languages*, PADL '01, pages 304–321, Jan. 2001.

[20] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *Proc. of the '09 International conference on Functional programming*, ICFP '09, pages 23–34, 2009.