Scripted Signal Functions

David A. Stuart
alex@das.li

Abstract

Programming time-dependent signals like animations involves expressing both continuous and discrete changes in signal values. The method of functional reactive programming (FRP) represents this simply and effectively as discrete modes of an otherwise continuous signal. In variants of FRP based on arrows, programs are composed not of signals but rather functions on signals. Accordingly, these signal functions can switch between discrete modes of operation. However, the literature emphasizes expressions of mode switching that are unnecessarily limited. An analysis of their limitations indicates the need for two new, primitive transformations of signal functions. These transformations help to define a monad that represents signal function modes, and this allows programmers to express switching in an imperative, script-like style. This scripting interface gains flexibility and power from several novel operations, including a general-purpose mapping between modes and a combination that mixes two concurrent modes into one. We demonstrate its usefulness with several examples.

CCS Concepts: • **Software and its engineering → Control structures**; **Functional languages**; Concurrent programming structures; Scripting languages.

Keywords: functional reactive programming, Haskell

ACM Reference Format:
David A. Stuart. 2020. Scripted Signal Functions. In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3406088.3409016

1 Introduction

Several languages for functional reactive programming (FRP) are embedded in Haskell as arrow combinators that represent *signal functions* [1, 17, 21]. These are relations on time-dependent quantities, and a relation itself can vary over time. They elegantly model the behavior of hybrid dynamical systems ranging from robots to interactive animations [5, 11, 20]. Languages based on signal functions use Haskell to directly and consistently discretize a model's complex behavior into simple state transitions over short time steps.

Users of these languages compose signal functions to represent common signal-processing relationships. Each of these compositions yields another signal function, making it convenient to construct complex relationships out of simpler ones. They include *mode switching*, where the composite behaves first as one signal function and then switches to behave as another. This in particular allows a modeler unique flexibility when inventing system models, because several models of nearly arbitrary diversity can comprise the history of one signal function. It is not necessary to devise a single, continuous model that covers every moment [16].

However, the common ways to express mode switching are limited. Yampa for instance provides a combinator for simple expressions that makes more involved expressions hard to change or even impossible to write. It also provides a way to manage a set of concurrent signal functions, but this depends on one set of parameters that constrain the whole course of every signal function in the set. These issues limit the flexiblity that mode switching can offer.

As it turns out, there are ways for a modeler to quite simply express sophisticated mode switching in a signal function as just a schedule of activities for the signal function to engage in, much like a stage actor's script. This can be a very abstract expression in which each line of the script denotes a long interval of complex behavior punctuated by a dramatic state transition. Yet because we give the expression meaning in terms of signal functions, it is still directly and consistently defined as short intervals of primitive behaviors.

Prior research on Haskell programs for robot control developed this concept to some extent for the purpose of robot task planning [13, 20]. This paper goes further by showing how signal functions can implement plans of new flexibility and power for any component of a behavior while maintaining their simple semantics of continuous time. Its main technical contributions are

- a novel monadic representation of discrete modes in signal functions (section 3.2),
- a general way of transforming one mode into another (section 3.6),
- an operator that combines two modes into one multi-threaded mode (section 4.3), and
- two new primitive operators on signal functions that enable all of this (sections 3.1 and 4.2).

Haskell '20, August 27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409016

It also presents examples of how to use signal function scripts effectively. In particular, it shows how they can serve as a simple, unified language for expressing animation, character behaviors, scene changes, and other features of interactive graphics applications such as games.

2 Background

2.1 Continuous Behaviors

It is common to model the behavior of a system as a quantity that varies over time: a *signal*. We can think of a signal as a function $f : \mathbb{R} \to V$ from the continuum of time to some set of possible quantities. Thus at any time $t \in \mathbb{R}$ a signal has a particular value $f(t) \in V$, and this value can change continuously as time passes continuously. This simple idea underpins our intuition about our physical environments.

FRP aims to help programmers recruit that intuition to directly express system models as processes in a computer without introducing undue complexity and error [8]. Users of the arrow languages for FRP express functions on signals, which take one signal as input and transform it to produce another signal as output. We can think of a signal function as a function on functions $\mathcal{F} : S_1 \to S_2$, where $S_1 = \{f \mid f : \mathbb{R} \to V_1\}$ is a set of input signals and $S_2 = \{g \mid g : \mathbb{R} \to V_2\}$ is a set of output signals. To evaluate the output signal at a given time, we evaluate the history of the input signal up to that time and compute its transformation under $\mathcal{F}$.

In Yampa, a signal may carry values of an arbitrary type. A signal function has the type *SF a b*, where its input signal carries values of type *a* and its output signal carries values of type *b*. The type *SF* is an arrow: an instance of the *Arrow* type class [14]. The primitive arrow combinators in this instance amount to primitive ways of routing signals between signal functions, and arrow notation amounts to a block diagram for a network of signal chains [19]. Importantly, *SF* is an instance of *ArrowLoop*; the *loop* combinator (and hence *rec* in arrow notation) defines feedback loops in signal chains. Signal functions are closed under these combinations, allowing us to make new signal functions by composing existing ones.

2.2 Discrete Behaviors

Of course, not every quantity in every system changes continuously all the time. Hybrid dynamical systems, such as models of urban traffic or switched electrical circuits, express relations between both continuous and discrete changes [7, 12]. Reactive computer systems like graphical user interfaces and industrial process controllers receive both continuous signals and reports of discrete events as input [2]. To model discrete behaviors, signal functions transform *event streams*. These are signals whose values have an option type, such as

```
data Event a = NoEvent | Event a
```

where, at any time, the signal value *Event a* denotes an event occurrence and the value *NoEvent* denotes non-occurrence. Signal functions can treat event streams just as they treat any other signal. But event streams can also drive mode switching: discontinuous changes in how signal functions map their input to their output. These changes can be arbitrarily dramatic; only the input and output types must remain the same throughout. We can express a broad range of systems as signal functions that switch between different modes at the moments of certain events.

2.3 Expressions of Mode Switching

Yampa represents mode switching with the combinator

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
```

that combines a signal function with a continuation to produce a new signal function. The first signal function must have a second output channel that signifies the termination of the first mode and gives the argument to the continuation. Applying the continuation results in a second signal function, and altogether the combination denotes one signal function with two modes: one active before it detects a termination event, and the other active thereafter.

To combine more than two modes we must combine them recursively: we combine the last two modes, and then we combine the result with the third-to-last mode, and so on:

```
m_{n-3} `switch` (λc_3 → ⋯
  m_{n-2} `switch` (λc_2 → ⋯
    m_{n-1} `switch` (λc_1 → ⋯
      m_n)))
```

Each of these modes except the last one must be represented by a signal function with an auxiliary event-stream channel.

Combining modes with *switch* is analogous to combining monadic actions with the bind operator (≫=). In each, the first parameter produces a value, and the second parameter uses that value to generate a new producer like the first parameter. A monadic interface to mode switching would provide many conveniences. The most immediate of these would be the freedom to write modes not as a single, nested continuation, but rather as a sequence of activities, each expressively and meaningfully separate from the continuation:

```
runMode (do c_3 ← m_{n-3}
            c_2 ← ⋯ m_{n-2}
            ⋯ m_{n-1})
        (λc_1 → ⋯ m_n)
```

Inserting an extra mode between two others would be more straightforward with such an interface. We could define new modes as finite sequences of other modes, rather than continuations of them. We could define an infinite, looped sequence by recursively punctuating a mode with itself.

Normally this is not possible because *switch* lacks the associative property of (≫=), where it is always true that

$$
\begin{aligned}
&(m_1 \gg\!= \lambda a_1 \to \cdots m_2) \gg\!= \lambda a_2 \to \cdots m_3 \\
&= m_1 \gg\!= (\lambda a_1 \to \cdots m_2 \gg\!= \lambda a_2 \to \cdots m_3)
\end{aligned}
$$

The analogous associations of terms combined with *switch* imply two different types for the same signal function. On one hand,

$$
\begin{aligned}
&((m_1\ \text{`}switch\text{`}\ \lambda c_1 \to m_2) \\
&\qquad \text{`}switch\text{`}\ \lambda c_2 \to m_3) &&:: SF\ a\ b \\
&\therefore (m_1\ \text{`}switch\text{`}\ \lambda c_1 \to m_2) &&:: SF\ a\ (b, Event\ c) \\
&\therefore m_1 &&:: SF\ (a, ((b, Event\ c), \\
&&&\qquad Event\ d))
\end{aligned}
$$

but on the other hand,

$$
\begin{aligned}
&(m_1\ \text{`}switch\text{`}\ (\lambda c_1 \to \\
&\quad m_2\ \text{`}switch\text{`}\ \lambda c_2 \to m_3) &&:: SF\ a\ b \\
&\therefore m_1 &&:: SF\ a\ (b, Event\ c)
\end{aligned}
$$

There is no finite type b that matches $(b, Event\ c)$.

So, functional reactive programming with arrows is, overall, an elegant way to model systems with both continuous and discrete behaviors. However, its expressions of mode switching can be cumbersome, and their similarity to monad expressions suggests how they can be more flexible.

3 Sequential Modes

A monad to represent modes clearly requires a different way of concatenating them than *switch*. To determine what that is, we must observe the asymmetry between the type of *switch*'s first operand and the type of its result. The output type of the former includes a switching event, and the output type of the latter does not. This, by design, controls the complexity of signal functions under the mode-switching combination. The presence of modes in a signal function is immaterial to further combinations involving that signal function.

3.1 Linking Signal Functions Associatively

By relaxing this design goal we can define a mode-switching combinator *link* that retains the denotation of a mode and is thereby associative. It could have the type

$$
\begin{aligned}
&SF\ a\ (b, Event\ c) \to (c \to SF\ a\ (b, Event\ d)) \\
&\qquad \to SF\ a\ (b, Event\ d)
\end{aligned}
$$

and its implementation would be nearly identical to that of *switch*. The only difference is that the result puts out values that include a second component, just as the first operand does. The value of that component denotes the absence of a termination event throughout the first mode and up to the end of the second mode. In this way, modes represented by the type $SF\ a\ (b, Event\ c)$ would be closed under *link*.

Furthermore, we can do the same for modes represented by the type $SF\ a\ (Either\ b\ c)$. Neither *link* nor *switch* ever computes these b and c values at the same time, so a sum type can stand in for both. As we will see, this is a more convenient representation.

Despite its similarity to *switch*, there is no way to define *link* in Yampa. We must instead add it as a new primitive, defining it in terms of an underlying implementation: the embedding of signal functions in Haskell. Perez and colleagues expressed a particularly simple and extensible implementation using monadic stream functions, or MSFs [21], so we will base our definition on that. They define MSFs as

$$
\mathbf{data}\ MSF\ m\ a\ b = MSF\ \{\, unMSF :: a \to m\ (b, MSF\ m\ a\ b)\,\}
$$

where the function *unMSF* maps an input sample to both an output sample and another MSF, in some monad. The new MSF will process the next input sample, and thus it defines how the result of *link* continues given its output value.

```
link :: SF a (Either b c) → (c → SF a (Either b c))
                          → SF a (Either b d)
link sf k = MSF (λa → do
                  (bc, sf') ← unMSF sf a
                  case bc of
                    Right c → unMSF (k c) a
                    Left b  → return (Left b, link sf' k))
```

This samples the output of the first signal function. If its second component matches *Right c*, the second signal function takes over. Otherwise the sample stands. See Appendix A for a proof of associativity.

3.2 Representing Modes

Normally it is simple to represent modes with signal functions of type $SF\ a\ (b, Event\ c)$. However, some modes are what we may term *recessive modes*. They have no duration, and their hypothetical output is always superceded by the remainder of the signal function. Ultimately these modes just record, instantaneously in simulated time, a parameter for other modes. To represent this accurately, our representative signal functions instead have output of type $Either\ b\ c$:

$$
\mathbf{newtype}\ Mode\ a\ b\ c = Mode\ (SF\ a\ (Either\ b\ c))
$$

At the moment of switching, the output value will have the *Right* constructor. At all other times the output value will have the *Left* constructor.

The type $Mode\ a\ b$ is a monad in the type c of the termination parameter:

```
instance Monad (Mode a b) where
  return c       = Mode (constant (Right c))
  Mode sf >>= f  = Mode (link sf g)
    where g c = let Mode sf' = f c in sf'
```

The expression $m \gg\!= k$ denotes a mode that itself has two modes. $(\gg\!=)$ is identical to *link* aside from wrapping and unwrapping the *Mode* constructor. The expression *return c* denotes a recessive mode that records the value c.

3.3 Signal Functions from Modes

One operation on values of type *Mode a b c* is indispensible, namely producing well defined, mode-switching signal functions from them. A natural candidate for this operation is just *switch*, though we must translate the output type of our representative signal function to be compatible with it.

```
runMode :: Mode a b c → (c → SF a b) → SF a b
runMode (Mode sf) k = switch (sf >>> arr toEvent) k
  where toEvent x = case x of
                      Left b  → (b, NoEvent)
                      Right c → (⊥, Event c)
```

As before, we produce a signal function by combining a mode with a continuation that gives the final mode. In fact any continuation that works with *switch* will work here as well. Note that the term $\bot$ in the *Left b* case of *toEvent* will never be evaluated, because it is superceded by the output of the final signal function. *switch* ignores the output of the first signal function at the moment of switching.

Often a sequence of modes should not just run once but instead recur indefinitely in a loop. We can define such a loop recursively as we do for any monad. We could apply *runMode* to the loop, but it would never apply its final continuation. For this case we define a special version of *runMode* that does not require a final continuation as an argument:

```
loopMode :: Mode a b c → SF a b
loopMode m = runMode m (const (loopMode m))
```

This will of course diverge when applied to a recessive mode, since it will never stop switching out of it and thence back into it. This divergence is an extant pitfall of switching in Yampa and its MSF-based descendent, Bear River [21]; we do not propose to resolve it here.

3.4 Modes from Signal Functions

Conversely it is useful to define some basic modes as intervals of arbitrary signal functions. While we can do so with the *Mode* constructor, we may want to hide it, and anyway the desired behavior in a mode may not be readily available as a value of type *SF a* (*Either b c*).

We produce the most basic of these modes with *always*. Its application to a signal function denotes a mode that never terminates, always behaving as that signal function does:

```
always :: SF a b → Mode a b c
always sf = Mode (sf >>> arr Left)
```

Note that (>>>) is the composition combinator for arrows and *arr* is the function that lifts a generic function to an arrow command. Here the combination $sf_1 >>> sf_2$ denotes a signal function that feeds the output signal of sf_1 to the input of sf_2, and the expression *arr f* denotes a signal function whose output signal is just the pointwise application of f to its input signal.

A variation of that is *over*, which takes as arguments a length of time and a signal function and gives a mode that terminates after the length of time has passed. During that length of time it behaves as the signal function does:

```
over :: DTime → SF a b → Mode a b ()
over interval sf = Mode (sf &&& time >>> arr check)
  where check (b, t) | t < interval = Left b
                     | otherwise    = Right ()
```

Note that (&&&) is the fan-out combinator for arrows. Here the combination sf_1 &&& sf_2 denotes a signal function with two output channels, transforming an input signal with both sf_1 and sf_2 simultaneously and pairing their output signals.

We can define an interval of activity for a signal function more generally than as a predetermined length of time. We can combine an arbitrary signal function with an event source, producing a mode that runs until the first event.

```
before :: SF a (Event c) → SF a b → Mode a b c
before interrupt sf = Mode (sf &&& interrupt >>> arr check)
  where check (_, Event c)  = Right c
        check (b, NoEvent) = Left b
```

3.5 Other Primitive Modes

Some useful, basic modes are not based on a signal function parameter. The most useful of these modes is *sample*, a recessive mode that binds the current value of the input signal for use in defining the next mode.

```
sample :: Mode a b a
sample = Mode (arr (λa → Right a))
```

This allows a script to directly express, among other things, a branching decision based on the input signal. For example, given terms *fight* and *flight* that model long-interval activities of an animal, and some function *danger* of the animal's senses, we can concisely script a crude stress response:

```
response :: Mode Senses Actions c
response = do senses ← sample
              if danger senses < d_max then fight
                                       else flight
```

Often it is necessary to put out a signal value for as short an interval of time as possible, especially if the value denotes a current event. By applying *moment* to a function f which produces that value we obtain a very short-lived mode with constant output equal to that value.

```
moment :: (a → b) → Mode a b ()
moment f = Mode (proc a → do
                    done ← iPre False -< True
                    returnA -< if done then Right ()
                                       else Left (f a))
```

The expression *iPre False* denotes a signal function that translates a Boolean signal forward in time by one step, delaying the end of the mode by the smallest possible interval.

3.6 Transforming Modes

Thus far we have developed the means to construct a variety of composite modes. But we still lack a way to make slight modifications to an existing mode, other than modifying its existing expression. We can instead define a function that maps one mode to another based on a transformation of its signal relation, its termination condition, or both.

Any transformation that we apply to a whole signal function we may reasonably apply to only one mode of that signal function. For instance, we may want to amplify a signal function's output during one mode only. Or we may want to filter the input in one mode so that we can compose it with modes of a different input type. We may also wish to alter the duration of a mode but retain the same relation on signals. For example, we may want an otherwise lengthy mode to last no longer than one second.

Given our representation of modes, we can represent all of these transformations as functions of type

```
SF a (Either b c) → SF d (Either e f)
```

and apply them trivially with the function

```
mapMode :: (SF a (Either b c) → SF d (Either e f))
        → Mode a b c
        → Mode d e f
mapMode f (Mode sf) = Mode (f sf)
```

One such mapping terminates a mode early based on an extra event source. Whenever the mode's representative signal function indicates a normal output with the *Left* constructor, we can also run the event source to conditionally construct a terminating output with *Right*. The *ArrowChoice* instance for signal functions allows a case analysis of the output to determine whether the event source runs. Hence the event source can observe both the input and output signals.

```
onlyUntil :: SF (a, b) (Event c) → Mode a b c → Mode a b c
onlyUntil source = mapMode clip
  where clip sf = proc a → do
          ebc ← sf ≺ a
          case ebc of
            Left b → do
              stop ← source ≺ (a, b)
              returnA ≺ case stop of
                          Event c  → Right c
                          NoEvent → Left b
            Right c → do
              returnA ≺ Right c
```

Applied to an event source, *onlyUntil* gives us a simple way to add behavior to a script under certain conditions. Given an existing script that runs unconditionally, we can summarily clip it to an interval preceding a certain event and insert it between two other modes in a larger script.

By the same mechanism, we can transform a mode to attach a sample of its output signal to its termination parameter. This can be particularly useful for handing off the value of a shared state variable between modes:

```
lastOut :: b → Mode a b c → Mode a b (c, b)
lastOut b0 = mapMode (record b0)
  where record b0 sf = proc a → do
          rec bR ← iPre b0 ≺ b
              ebc ← sf     ≺ a
              b   ← case ebc of
                      Left b1 → returnA ≺ b1
                      Right c → returnA ≺ bR
          returnA ≺ fmap (, b) ebc
```

At this point our interface to sequential mode switching is fairly robust. It lets us create modes from signal functions, create signal functions from modes, and create modes by combining and transforming other modes.

4 Concurrent Modes

Representing modes with a monad allows us to do more than express mode switching in a single signal function. In fact, we can go further and use modes to express concurrent mode switching among multiple signal functions.

4.1 Parallel Switching in Yampa

For this same purpose Yampa provides the function

```
pSwitch :: Functor col ⇒
           (forall sf.a → col sf → col (b, sf))
           → col (SF b c)
           → SF (a, col c) (Event d)
           → (col (SF b c) → d → SF a (col c))
           → SF a (col c)
```

that produces one signal function with the combined output of multiple signal functions in an evolving collection. It takes four values as arguments, respectively defining a *routing function* to distribute input to the collection, the *initial collection* before any changes have occurred, an *event source* to detect when the changes should occur, and a *continuation* to effect the changes themselves.

Individually these concepts are reasonably easy to understand, and together can represent a very general kind of concurrent mode switching. However, the complexity of *pSwitch* is a departure from the simple and isolated operations on signal functions that constitute the rest of the language. Using it at all immediately requires us to write expressions whose types are unique. For that reason we must usually work them out from scratch rather than building them from existing definitions, and we usually cannot reuse them in

other definitions. Moreover they must be valid throughout the entire course of the signal function. We cannot construct them from simpler definitions that are valid only during certain modes. Since that capability is one of the virtues of mode switching itself, we should consider a different interface.

4.2 Access to Remainders of Signal Functions

Before we can do that we must make more generally available one of the central provisions of *pSwitch*: access to the remainders, at the moment of switching, of the signal functions that it manages. These are their relations on signals as they have evolved up to that moment, also called their "continuations" [21]. We will refer to them as "remainders" to avoid confusion with the functions that produce them, which are called "continuations" as well. *pSwitch* passes these remainders in a collection of type *col* (*SF b c*) to its continuation, which transforms the collection based on the switching event parameter and then encapsulates it once again as a single signal function. From the continuation's type

$$Functor\ col \Rightarrow col\ (SF\ b\ c) \rightarrow d \rightarrow SF\ a\ (col\ c)$$

we can infer that the collection of remainders cannot escape the signal function as such (see Appendix B.2).

This is an appropriate constraint on a switching function, because just as *switch* controls complexity by hiding the presence of modes in a signal function, *pSwitch* controls complexity by hiding the presence of concurrent processes. No further combinations involving the result of *pSwitch* depend on how it develops its collection of remainders.

At the same time, there is no reason to make the continuation of *pSwitch* the exclusive custodian of signal function remainders. We can provide a way to produce as an output signal the remainder, at any moment, of an arbitrary signal function. This can be an operator

$$vain :: SF\ a\ b \rightarrow SF\ a\ (b, SF\ a\ b)$$

that transforms a signal function into one with an auxiliary output channel denoting its current remainder.

While it may seem unusual for a continuous signal to have type *SF a b*, it poses no semantic problems. All signal functions are defined not on the entirety of time $(-\infty, \infty)$ but on the interval of time $[t_0, \infty)$ greater than or equal to some particular moment t_0. We can always define a signal function for which t_0 is the present, so we can define a signal denoting the evolution of that signal function as the present changes. Specifically, let us consider a signal function to be a second-order function $\mathcal{F} : S_A \rightarrow S_B$, where $S_A = \{f \mid f : \mathbb{R}^+ \rightarrow A\}$ and $S_B = \{g \mid g : \mathbb{R}^+ \rightarrow B\}$. Let us say that

$$\mathcal{F}(a) = t \mapsto b(t, a(t))$$

for some function $b : \mathbb{R}^+ \times A \rightarrow B$ of both time t and the value of $a(t)$. Then the signal function's remainder after a quantity $\tau \in \mathbb{R}^+$ of time has passed is

$$\mathcal{G}(a) = t \mapsto b(t + \tau, a(t + \tau)).$$

Abstracted over τ, this is a function $h : \mathbb{R}^+ \rightarrow G$, where G is the set of functions $\{\mathcal{G} \mid \mathcal{G} : S_A \rightarrow S_B\}$. It is a function of time, just like any signal.

This operator, like *link*, is impossible to implement in Yampa, and we must add it as a new primitive. Doing so is straightforward if the language, like Yampa, already implements signal functions in terms of their remainders. Bear River implements signal functions with monadic stream functions, which make this particularly easy. In fact, we can implement *vain* not just as a function on signal functions but as a function on MSFs in general:

```
vain :: Monad m ⇒
        MSF m a b → MSF m a (b, MSF m a b)
vain msf₀ = MSF { unMSF = f }
  where f a = do (b, msf₁) ← unMSF msf₀ a
                 return ((b, msf₁), vain msf₁)
```

Here the function that underlies *vain* msf_0 applies the function underlying msf_0 to an input sample a, producing an output sample b and a remainder msf_1. It pairs these as its own output sample, and it applies *vain* to the remainder to produce its own remainder.

4.3 Combining Modes in Parallel

The importance of this for mode switching is that we can sample the remainder signal upon a switching event and use the sample to define the next mode—or indeed the next set of modes. We can define a combinator *mix* that combines two modes of the same type into one mode with multiplex output, terminating once both components have terminated. That specification assumes that we can perform some operation that combines two output signals. It also assumes that such a combination is well defined even after one component stops producing output—that we can substitute an identity value for a missing output in the combining operation. Both of these assumptions are captured at once by assuming that the output type is a monoid. Thus a good type for *mix* is

```
mix :: Monoid b ⇒
       Mode a b c → Mode a b d → Mode a b (c, d)
```

Many types that represent multiple values, such as lists and tables, have natural monoid instances. Our specification also assumes that we can end one mode but keep the remainder of the other mode: this is what *vain* allows us to do.

The composite mode itself has two possible modes: a primary mode when both component modes are running, and a secondary mode when one component outlasts the other:

```
mix (Mode sf₁) (Mode sf₂)
  = do result ← Mode both
       case result of
         Win1 (c, r) → Mode r ≫= λd → return (c, d)
         Win2 (d, r) → Mode r ≫= λc → return (c, d)
         Tie (c, d)  → return (c, d)
```

Here we represent the relative timing of the two modes with a sum type whose constructors indicate, respectively, the first mode terminating first (*Win1*), the second mode terminating first (*Win2*), or both terminating simultaneously (*Tie*).

The signal function representing the primary composite mode applies *vain* to each of the signal functions representing its components:

$$\begin{aligned} &both = \textbf{proc}\ a \to \textbf{do} \\ &\quad (bc_1, r_1) \leftarrow vain\ sf_1 \prec a \\ &\quad (bc_2, r_2) \leftarrow vain\ sf_2 \prec a \\ &\quad returnA \prec watch\ (bc_1, r_1)\ (bc_2, r_2) \end{aligned}$$

On the first terminating *Right* value we attach the other remainder (r_2 or r_1) to its parameter and use it to report the primary composite mode's termination as well.

$$\begin{aligned} watch\ (Left\ b_1, \_)\ (Left\ b_2, \_) &= Left\ (b_1 \diamond b_2) \\ watch\ (Right\ c, \_)\ (Left\ \_, r_2) &= Right\ (Win1\ (c, r_2)) \\ watch\ (Left\ \_, r_1)\ (Right\ d, \_) &= Right\ (Win2\ (d, r_1)) \\ watch\ (Right\ c, \_)\ (Right\ d, \_) &= Right\ (Tie\ (c, d)) \end{aligned}$$

Note that ($\diamond$) is the monoid operator for the output type b. Afterward the attached remainder represents the secondary composite mode, and we say that its output is combined with the monoid identity. If both values bc_1 and bc_2 indicate termination we do not attach either remainder and instead report the simultaneity.

As its type and effect of parallelism indicate, *mix* is closely related to applicative functors. In fact, we could define an instance of the *Applicative* type class for modes such that

$$mix = liftA2\ (,)$$

It can be shown that this would satisfy the applicative functor laws. However, there are several wrinkles in this relationship. First, the parallel effect of the application operator ($\circledast$) would be different than the sequential effect of ($\gg\!=$), meaning that

$$m_1 \circledast m_2 \neq m_1 \gg\!= (\lambda f \to m_2 \gg\!= (\lambda x \to return\ (f\ x)))$$

Also, since *Applicative* is a superclass of *Monad*, the monoid constraint on the output signal of concurrent modes would unnecessarily constrain those modes that we only compose sequentially. To avoid these issues, we could wrap our type *Mode a b c* in a new type and define the *Applicative* instance for the wrapper type instead. Even then, there is a case for different kinds of signal routing between concurrent modes and hence different wrappers. Because of these complexities, we will consider *mix* independently of applicative functors.

4.4 Multithreaded Scripts

Mix allows us to plainly express the concurrency of activities that otherwise requires *pSwitch*. While we can apply *mix* directly, it also allows us to extend the scripting interface with expressions of concurrent threads. For instance, a script can run parallel to others if we define a thread for it with

$$voice :: Monoid\ b \Rightarrow Mode\ a\ b\ c \to Threads\ a\ b\ c$$

Let us assume that some operation can combine multiple thread values into one representing all of them. Then we can determine its place in the rest of a sequential script by using

$$chorus :: Monoid\ b \Rightarrow Threads\ a\ b\ c \to Mode\ a\ b\ c$$

to map a thread combination to a single mode that runs as long as the longest thread runs.

We can represent threads in such a way that the syntax of our thread expressions blends well with our syntax for sequential modes. Ideally *voice* would express a new thread branching off at a specific moment, parallel to the main script. That is simple if we add continuation passing to our *Mode a b* monad using the *ContT* monad transformer:

$$\textbf{type}\ Threads\ a\ b\ c = ContT\ c\ (Mode\ a\ b)\ ()$$

This way, when *voice* creates a new thread from a mode, it has access to the current continuation of the thread combination. That could include not only more *voice* expressions but also lifted mode sequences: we need only apply *lift* to any mode to include it in the main thread.

Specifically, *voice* uses *callCC* to obtain the current continuation. From that it produces a mode representing the main thread, and it uses *mix* to combine it with the new mode:

$$\begin{aligned} &voice\ m = callCC\ (\lambda cc \to \\ &\qquad ContT\ (\lambda k \to mix\ m\ (runContT\ (cc\ ())\ k))) \end{aligned}$$

Given this representation, we can infer from the type of *chorus* the definition

$$chorus\ t = runContT\ t\ return$$

This is a mode that runs the concurrent modes combined in t and produces their combined result.

Compared to using *pSwitch*, this kind of multithreaded scripting is more like our means of composing ordinary sequential modes. Expressions of routing and switching, if any, are specific to certain modes of individual threads, whereas *pSwitch* requires that they are the same for all of its threads at all times. And as with sequential mode switching, there is no need to define a continuation for every switching event.

5 Examples

Scripted signal functions, as we derive them in this paper, first transpired during the production of a game using Yampa. They repeatedly emerged as the most satisfactory solution to a number of superficially disparate problems that other productions solve with substantially disparate mechanisms. While this is in large part thanks to Yampa itself, the value of the scripting interface is evident in several examples.

One of these examples relies on the extended types of signal functions that monadic stream functions can implement [21]. In Bear River the type of signal functions is *SF m a b*, polymorphic in the monad parameter m of the underlying MSF. Therefore, in these examples our mode type

has this parameter too: *Mode m a b c*. Previous definitions do not rely on this polymorphism.

5.1 Animation Playback

Perhaps the most obvious of these examples is a signal function that plays back pre-rendered frames of an animation. This has type *SF m a Image*, where *Image* is some representation of an image that can change. It could just be a command of type *IO* () that writes image data to a framebuffer.

We can express a particular, four-frame walking animation (fig. 1) directly as a script:

```
walk :: Monad m ⇒ Mode m a Image
walk = do over 0.1 forward
          over 0.1 stepRight
          over 0.1 forward
          over 0.1 stepLeft
```

Each mode could be just *constant x* for some image *x*, or it could be sensitive to input such as a position on the screen.

We can generalize this to an arbitrary number of frames and an externally controlled playback speed:

```
play :: Monad m ⇒
        [SF m a Image] → Mode m a Image ()
play frames = sequence (map (before timeUp) frames)
  where timeUp = proc a → do
                   t ← time ≺ ()
                   edge ≺ t < pace a
```

Here *edge* is a primitive signal function that puts out an event when its input changes from *False* to *True*, and *pace* is some function of the input signal value giving the current duration of a frame. We can implement *play* with *switch*, though it is more complicated:

```
play' :: Monad m ⇒ [SF m a Image] → SF m a Image
                                  → SF m a Image
play' frames k = chain sfs
  where sfs           = map mode frames
        mode frame    = proc a → do
                          image ← frame ≺ a
                          t     ← time  ≺ ()
                          done  ← edge  ≺ t < pace a
                          returnA ≺ (image, done)
        chain [ ]     = k
        chain (x : xs) = switch x (chain xs)
```

Critically, in the first case we can just as easily play the animation once, or twice, or right before a different one, or in a loop:

```
once      = play x
twice     = play x ≫ play x
different = play x ≫ play y
looping   = forever (play x)
```

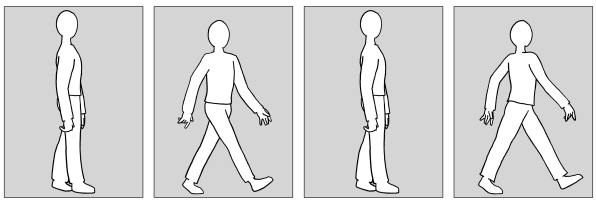

Figure 1. Four frames of a walking animation

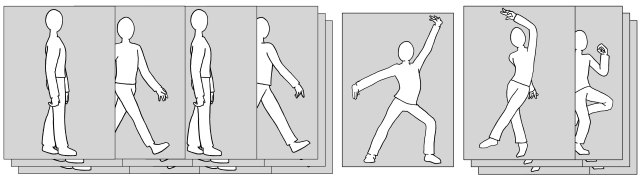

Figure 2. A sequence of animations

In the second case these choices are determined by the continuation *k*. If we want our signal function to switch from *play' x k* into a different mode, we must come up with a new event source and a new continuation.

5.2 Character Behaviors

The ease of constructing animation sequences is significant, because a walking animation is usually only part of a character's total animation (fig. 2). A character may play *walk* in a loop for some number of seconds *n*, but then strike a pose and start dancing:

```
disco :: Monad m ⇒ Double → Mode m a Image c
disco n = do onlyUntil (after n ()) (forever walk)
             pose
             forever dance
```

Some animations terminate early and thence lead to a different animation sequence. For example, if something strikes a character, the previous animation should stop immediately and switch to a staggering animation.

```
mosh :: Monad m ⇒ Mode m a Image c
mosh = do onlyUntil struck disco
          stagger
          headbang
```

This of course depends on *struck* receiving enough information from the input signal to produce the early-termination event stream.

We can furthermore associate other behaviors with these animation modes. One of the most important is a changing position within a frame as the walking animation plays—otherwise the character walks in place, as if on a treadmill. Let us assume that our frames in section 5.1 indeed receive a position as input. Then a particularly convenient way to work this out is by making position part of the output as

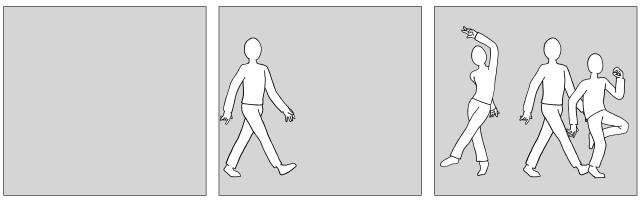

Figure 3. Concurrent dancers entering the dancefloor

well, feeding it back to the input, and hiding the whole loop with a state-handling monad transformation. That way we can just use *mapMode* to include, in the output of *walk*, an extra state-recording action that updates the position:

```
trek :: Monad m ⇒ Mode (StateT m) a Image c
trek = mapMode move (forever walk)
  where move sf = (sf &&& time) ⋙ go
        go      = arrM (λ(b, t) →
                    put (speed ∗ t) ≫ return b)
```

Thus we can substitute *trek* for *forever walk* in the definition of *disco*, and we can define a transformation of it that closes the state loop:

```
dancer :: Monad m ⇒ Double → Mode m a Image c
dancer n = mapMode (runStateSF 0) (disco n)
```

Here *runStateSF* 0 is a function that maps a signal function with monadic state handling to one with an initialized feedback loop. Only the walking animation will move across the frame, and the other animations will remain stationary. Because of the state handling, the other animations will nevertheless be stationary at the last position that *trek* recorded.

Concurrent dancers are simple to express using *voice* and *chorus*. Several characters can follow one script together:

```
club :: Monad m ⇒ Mode m a Image c
club = chorus (do onlyUntil musicBegins
                    (always (constant mempty)) & lift
                  strobeLight                  & lift
                  voice (dancer 5)
                  onlyUntil musicCrescendo
                    (always (constant mempty)) & lift
                  strobeLight                  & lift
                  voice (dancer 3)
                  voice (dancer 7))
```

Note that (&) is the reverse application operator such that $x \,\&\, f = f\ x$. This script says that three characters begin dancing in two stages. Nothing happens until *club* (the main thread) senses that music begins. At that time a strobe light begins flashing for some interval, and then the first dancer *dancer* 5 (a new thread) begins. Then, in the second stage, the light flashes again, and both of the remaining dancers begin simultaneously. All three of the dancers end at arbitrary times, and *club* ends when the last dancer ends (fig. 3).

As illustrated here, a script for one of many characters can be the same as the script for a lone character. The only additional work we may need to do is to express how its input signal relates to that of *club* and, if its output type is not a monoid, how it is mapped to one. *mapMode* does both.

5.3 Scene Changes

Such a scene as *club* may be the correct scene only until a user enters a command to leave, at which point an outdoor scene must entirely replace it. This changing of scenes is a critical feature of many games, and some implement it with weighty protocols or ad-hoc mechanisms for specific scenes. For us this is just a reiteration of mode switching, at yet a higher level:

```
story :: Monad m ⇒ Mode m a Image c
story = onlyUntil leave club ≫ outdoors
```

That much is simplistic, but without much more effort we can implement an automaton that loops in and out of four different scenes. If we bundle a scene with an event source that provides the next scene, we can define a programmable scene-changing system:

```
data Scene m a
  = Scene (SF m a Image,
           SF m (a, Image) (Event (Scene m a)))
world :: Monad m ⇒
         Scene m a → Mode m a Image (Scene m a)
world (Scene (here, next)) = onlyUntil next here ⋙= world
```

The program is given by defining functions for each scene that, by referring to each other, define the adjacencies of a state-transistion graph. Here we use four scenes—*club*, *street*, *alley*, and *cellar*—to define the possible transitions between them. From the club we can reach the street, alley, or cellar:

```
clubNext :: Monad m ⇒ (a, Image) → Event (Scene m a)
clubNext (A0, _) = Event (Scene (street, arr streetNext))
clubNext (A1, _) = Event (Scene (alley, arr alleyNext))
clubNext (A2, _) = Event (Scene (cellar, arr cellarNext))
clubNext _       = NoEvent
```

The street and alley are connected:

```
streetNext :: Monad m ⇒ (a, Image) → Event (Scene m a)
streetNext (A0, _) = Event (Scene (club, arr clubNext))
streetNext (A1, _) = Event (Scene (alley, arr alleyNext))
streetNext _       = NoEvent
alleyNext :: Monad m ⇒ (a, Image) → Event (Scene m a)
alleyNext (A0, _) = Event (Scene (club, arr clubNext))
alleyNext (A1, _) = Event (Scene (street, arr streetNext))
alleyNext _       = NoEvent
```

And the cellar is a dead end:

```
cellarNext :: Monad m ⇒ (a, Image) → Event (Scene m a)
cellarNext (A0, _) = Event (Scene (club, arr clubNext))
cellarNext _       = NoEvent
```

Here A_0, A_1, etc. are arbitrary patterns of user commands carried by the input signal, which cause the scene to change as each equation specifies. The expression for this automaton beginning from the club is *world* (*Scene* (*club*, *arr clubNext*)).

5.4 Graphical User Interfaces

The signal that drives the transitions between scenes is a stream of user commands coming from either the top-level input signal or another signal function. Let us consider the latter case. The signal function transmitting these commands can also display a window on the screen containing button icons. It can receive "feedback" from the user, who actuates the buttons using a mouse. In response, it can issue a command. The overall, combined signal function might be

```
game :: Monad m ⇒ SF m Mouse Image
game = proc mouse → do
         (imageG, cmd) ← gui   ≺ mouse
         imageS        ← scene ≺ cmd
         returnA ≺ imageG ◊ imageS
  where scene = loopMode (world (club, arr clubNext))
```

The signal function *gui* has two output channels carrying, respectively, an image of the window and a command event. It depends on an input signal carrying the state of the mouse.

Expressing the simultaneity of the command event and a button press is a standard matter of using a generic edge detector to generate a stream of click events, which we transform to include a command. We do not need to design additional mode switching for that. However, we may want to move the window around the screen by clicking on it and moving the mouse while we hold the button down. This progresses in three modes specific to the problem:

1. The cursor lies outside the window.
2. The cursor lies inside, and the mouse button is lifted.
3. The cursor lies inside, and the mouse button is pressed.

Some of these modes have different behaviors; most obviously, the window only moves during the third mode. Indeed, each mode might have unique behavior, such as a special window animation when the cursor is hovering over the window. We can script this as

```
gui :: Monad m ⇒ SF m Mouse (Image, a)
gui = loopMode window

window :: Monad m ⇒ Mode m Mouse Image c
window = do onlyUntil enter idle
            onlyUntil exit
              (forever (do onlyUntil press hover
                           onlyUntil loose move))
```

Here, *idle*, *hover*, and *move* define each mode's output. We express their switching conditions with *enter*, *exit*, *press*, and *loose*, which map the signal carrying the state of the mouse to event streams that report the cursor crossing the window boundary and the mouse button clicking down and up.

6 Related Work

Peterson and colleagues first presented a monadic interface to mode-switching robot behaviors in Frob, a system for controlling robots with functional reactive programming [23]. Their initial work represents task plans with a monad that wraps a continuation-passing-style (CPS) function, and it defines operators that add error handling and time limits to tasks. Later treatments develop operators for adding extra termination conditions and composing tasks in parallel [13, 22]. That work emphasizes tasks as high-level plans for robots, not a general-purpose interface to mode switching.

In later research on Frob, Pembeci and colleagues develop the task abstraction in terms of signal functions [20]. For intuition they propose to represent tasks with signal functions of type *SF a* (*b*, *Event c*), though they do not discuss how to implement this. They present an operation, analogous to *runMode*, for mapping tasks to signal functions. It handles tasks that terminate immediately (like our recessive modes) by producing only signal functions whose output signal has the sum type *Either b c*. Our method, on the other hand, expresses modes of signal functions with any output type.

Although it goes unmentioned in their first publications about Yampa, Courtney and Nilsson included a module for these signal function tasks in their source code as early as 2003 [4]. There they represent a task with a CPS function of type (*c* → *SF a* (*Either b d*)) → *SF a* (*Either b d*), in contrast to our direct-style type, *SF a* (*Either b c*). They define primitive tasks, early-termination transformations, and operations for mapping between tasks and signal functions. The CPS representation allows them to define (⋙=) without *link*, but adds complexity to functions that transform tasks.

More recently, Perez and colleagues cast mode switching as exception handling in monadic stream functions [21]. They show that, given a monad parameter with exception handling, stream processors can apply a handler to compute an alternative output sample and continuation. If the monad also includes a *Reader*-like environment to provide an implicit time step, the stream processors can represent mode-switching signal functions.

Bärenz and Perez went on to present a monadic interface for these exception handlers [1]. To represent recessive modes they directly lift an exception action to a stream processor. Applying this general scheme to processors with a *Reader* environment is nontrivial because both exceptions and handlers presume to cover the same time step. Our approach avoids this issue by not representing mode switching and time steps separately.

7 Discussion

7.1 Advantages over Imperative Scripts

Signal function scripts are high-level expressions that denote only infrequent transitions between discrete states, but they have a direct relation to the actual, high-frequency transitions between machine states that approximate continuous time as a rapid succession of time steps. In contrast, many imperative programming systems call their programs scripts, but they do not support such high-level expressions [6, 10, 18]. They express behavior over long intervals merely as long sequences of low-level state transitions over short intervals, each one barely perceptible. A line in such a script runs and terminates as fast as possible, and any observable output occurs only after the line terminates. We must infer from its short-interval behavior whatever long-interval behavior the script might express.

Many scripting architectures group those short-interval behaviors into callbacks or handlers for named events [3, 9, 15]. This gives us some certainty about when state transitions happen on a longer time scale. However, the scope of those transitions is seldom clearly defined, and the extent of their consequences is seldom obvious from a single handler expression [24]. For example, it is often natural for one object to have two event handlers that read from and overwrite the same set of variables. This allows one handler to permanently alter the effect of the other without any clear expression of a causal relationship. The sets of future transitions caused by these two event handlers are therefore difficult to distinguish from each other. They are totally ambiguous when the events occur simultaneously—to cover that case we must further define, somewhere, the relative timing of the handlers.

In comparison, the consequences of mode switching in signal functions are always restricted to the same output signal, which is explicit in the expression of the signal function, its type, and the other expressions we combine it with. The input signal is unchanged except in cases where we feed the output back to it, and these cases too are notated explicitly. The causes and resolutions of concurrent sequences are explicit in each instance of $(\gg\!=)$, *voice*, and *chorus*.

7.2 Limitations and Future Work

Our representation of modes does not support Yampa's delayed switching, where the output sample at the moment of switching comes from the first component rather than the second. In this sense the observable effect of the switch is delayed by one time step. In contrast, our representation assumes that this is not the case. The representative signal functions have output signals of type $Either\ b\ c$, so the output sample of the mode preceding the moment of switching is undefined at that moment. We could instead use the type $(b, Event\ c)$, though we run into ambiguity when defining this kind of output for recessive modes. This may allow one to express delayed switching by marking delayed modes with a different constructor.

The necessity of lifting ordinary modes in a multithreaded script is annoying. Ideally these scripts would more closely resemble standard thread code. There may be a way to omit the *ContT* monad transformer and handle continuations directly in our monad without impairing its other features.

Our approach performs as well in our examples as the ordinary use of Yampa does, but we have not fully analyzed its efficiency. Introducing *vain* probably does not increase the maximum memory use of Yampa programs, because they can in principle retain signal function remainders with *pSwitch* at the same rate as they can with *vain* (see Appendix B.2). Nonetheless, it is worth determining whether performance may degrade unexpectedly in certain cases.

7.3 Conclusion

Switching between different modes is a critical capacity of signal functions. We have developed a programming interface that provides straightforward and flexible access to this capacity, and we have demonstrated how to apply it. The variety of signal functions that we find cause to control with scripts, and the simplicity of writing and executing them, throw further light on the significance of functional reactive programming. It is remarkable how simple it is to express so many different behaviors with such consistency of meaning.

Acknowledgements

The author thanks the anonymous reviewers for their helpful suggestions, in particular the suggestion that the type $Either\ b\ c$, rather than $(b, Event\ c)$, could serve as the output type in the mode representation. The author also thanks Ivan Perez and Henrik Nilsson for their correspondence and conversation, which helped develop this paper.

A Associative Property of *link*

We want to show that, given the definition in section 3.1, it is always the case that

$$(m_1\ `link`\ \lambda c_1 \to \cdots m_2)\ `link`\ \lambda c_2 \to \cdots m_3 = m_1\ `link`\ (\lambda c_1 \to \cdots m_2\ `link`\ \lambda c_2 \to \cdots m_3)$$

Proof. There are two possibilities: either the output of m_1 is forever equal to $Left\ b$ for some b, or after a finite number of time steps it is equal to $Right\ c$ for some c. In the former case *link* never applies its continuation, so it is clear that

$$\forall x\ \ m_1\ `link`\ x = m_1$$
$$\therefore (m_1\ `link`\ \lambda c_1 \to \cdots m_2)\ `link`\ \lambda c_2 \to \cdots m_3 = m_1$$
$$\therefore m_1\ `link`\ (\lambda c_1 \to \cdots m_2\ `link`\ \lambda c_2 \to \cdots m_3) = m_1$$

Hence we need only consider the latter case, where after a finite number of time steps *link* applies its continuation. By substituting the body of *link* for applications of it, we know

$$
\begin{aligned}
&(m_1\ `link`\ \lambda c_1 \to m_2)\ `link`\ \lambda c_2 \to m_3\\
&= MSF\ (\lambda a \to \mathbf{do}\\
&\quad (bc_2, m_2') \leftarrow (\mathbf{do}\\
&\quad\quad (bc_1, m_1') \leftarrow unMSF\ m_1\ a\\
&\quad\quad \mathbf{case}\ bc_1\ \mathbf{of}\\
&\quad\quad\quad Right\ c_1 \to unMSF\ ((\lambda c_1 \to m_2)\ c_1)\ a\\
&\quad\quad\quad Left\ b_1 \to return\ (Left\ b_1, link\ m_1'\ (\lambda c_1 \to m_2)))\\
&\quad \mathbf{case}\ bc_2\ \mathbf{of}\\
&\quad\quad Right\ c_2 \to unMSF\ ((\lambda c_2 \to m_3)\ c)\ a\\
&\quad\quad Left\ b_2 \to return\ (Left\ b_2, link\ m_2'\ (\lambda c_2 \to m_3)))
\end{aligned}
$$

The **do** notation in the body of *link* expresses an action in the parameter monad of the monadic stream function. For signal functions we assume that this monad satisfies the law of associativity. Therefore, the above expression is equal to

$$
\begin{aligned}
&MSF\ (\lambda a \to \mathbf{do}\\
&\quad (bc_1, m_1') \leftarrow unMSF\ m_1\ a\\
&\quad \mathbf{case}\ bc_1\ \mathbf{of}\\
&\quad\quad Right\ c_1 \to \mathbf{do}\\
&\quad\quad\quad (bc_2, m_2') \leftarrow unMSF\ ((\lambda c_1 \to m_2)\ c_1)\ a\\
&\quad\quad\quad \mathbf{case}\ bc_2\ \mathbf{of}\\
&\quad\quad\quad\quad Right\ c_2 \to unMSF\ ((\lambda c_2 \to m_3)\ c_2)\ a\\
&\quad\quad\quad\quad Left\ b_2 \to return\ (Left\ b_2, link\ m_2'\ (\lambda c_2 \to m_3))\\
&\quad\quad Left\ b_2 \to return\ (Left\ b,\\
&\quad\quad\quad\quad\quad (m_1'\ `link`\ (\lambda c_1 \to m_2))\\
&\quad\quad\quad\quad\quad\quad `link`\ (\lambda c_2 \to m_3)))
\end{aligned}
$$

Finally, if we substitute an application of *link* for its body in the first clause of the first **case** expression, abstract the application over the argument value a, and wrap the abstraction in the *MSF* constructor, we have

$$
\begin{aligned}
&MSF\ (\lambda a \to \mathbf{do}\\
&\quad (bc_1, m_1') \leftarrow unMSF\ m_1\ a\\
&\quad \mathbf{case}\ bc_1\ \mathbf{of}\\
&\quad\quad Right\ c_1 \to unMSF\ ((\lambda c_1 \to\\
&\quad\quad\quad\quad\quad m_2\ `link`\ (\lambda c_2 \to m_3))\ c_1)\ a\\
&\quad\quad Left\ b_1 \to return\ (Left\ b_1,\\
&\quad\quad\quad\quad\quad (m_1'\ `link`\ (\lambda c_1 \to m_2))\\
&\quad\quad\quad\quad\quad\quad `link`\ (\lambda c_2 \to m_3)))
\end{aligned}
$$

Except for the association of terms in the second clause of the **case** expression, where the output is equal to $Left\ b_1$, this would clearly be equal to the body of *link*, and we could substitute the application

$$m_1\ `link`\ (\lambda c_1 \to m_2\ `link`\ \lambda c_2 \to m_3)$$

for it. This is what we wanted to show, but it is not obvious that it is true.

To see why we can make this substitution anyway, note that the second clause is not evaluated for every remainder m_1' of m_1—we are considering only the case where after a finite number of time steps the auxiliary output is equal to $Right\ c_1$ for some c_1. Therefore we may repeat the above derivation for each remainder m_1' in those preceding time steps. At that point our desired equality will be clearly true because it will not depend on the second **case** clause. Therefore we can substitute the other association in the second **case** clause of every preceding expression. Therefore, in both the case where m_1 terminates and the case where it does not,

$$
\begin{aligned}
&(m_1\ `link`\ \lambda c_1 \to \cdots m_2)\ `link`\ \lambda c_2 \to \cdots m_3\\
&\quad = m_1\ `link`\ (\lambda c_1 \to \cdots m_2\ `link`\ \lambda c_2 \to \cdots m_3)
\end{aligned}
$$

□

B Alternative Implementations

It is possible to implement in Yampa, without new primitives, some facilities like those described in this paper. These alternatives contrast with and further motivate our approach.

B.1 Naive CPS Representation

Partially applying *switch* to its first argument yields a function in continuation-passing style:

$$switch\ sf :: (c \to SF\ a\ b) \to SF\ a\ b$$

We can wrap this with the CPS monad:

$$cont\ (switch\ sf) :: Cont\ (SF\ a\ b)\ c$$

This allows us to separate modes from their continuations in a way that the raw use of *switch* does not, and it requires no languages changes. However, it introduces a new limitation as well, namely that there is no way to transform modes. That is, there is no function g of these CPS functions for which there exists a function f of signal functions such that, for all signal functions sf,

$$g\ (switch\ sf) = switch\ (f\ sf)$$

The function g could apply $(switch\ sf)$ to a continuation, which does not change the mode. It could subsequently apply some function to the result, which changes the first mode only inasmuch as it changes both modes together. Working around such a limitation has proved awkward.

B.2 Remainders from Switching Functions

Several switching functions have access to signal function remainders, and we can use them to export as a signal something like a remainder. For a signal function of type $SF\ a\ b$ this is a value of type

$$\mathbf{newtype}\ KeepSF\ a\ b = KeepSF\ (SF\ a\ (b, KeepSF\ a\ b))$$

which has an extra output signal. That extra signal has the same type as its source: $KeepSF\ a\ b$.

We can express this signal using *kSwitch*, which upon an input event captures the current remainder of the original signal function:

$$
\begin{aligned}
kSwitch :: &SF\ a\ b \to SF\ (a, b)\ (Event\ c)\\
&\to (SF\ a\ b \to c \to SF\ a\ b)\\
&\to SF\ a\ b
\end{aligned}
$$

To make the interface simple we can define this event inside a function *keep*. We say that this event "occurs" at every moment (after the first), so the remainder is always up to date (within one time step). *keep* maps a signal function to its corresponding *KeepSF* value that exports its own remainder:

```
keep :: SF a b → KeepSF a b
keep sf = KeepSF (kSwitch kf d k)
  where kf      = sf &&& constant (keep sf)
        d       = constant (Event ()) ⋙ notYet
        k sf' _ = n
          where n   = kSwitch kf' d k
                kf' = sf' ⋙
                      second (constant (KeepSF n))
```

The last parameter *k* of *kSwitch* is a continuation that defines the value of the signal function after the switching event. It is applied to the captured remainder, and the result is almost identical except that it replaces the old output remainder with the new one.

This requires no language changes, and the result is almost as good as that of *vain*. It is however an awkward caricature of the intended meaning of events and switching. To use it we must incessantly wrap and unwrap the *KeepSF* constructor, and we must accept that the output remainder is always delayed by one time step from the signal function it comes from. On all of these points it is better to include *vain* as a new primitive in the language.

Note that we cannot use *kSwitch*, or any of Yampa's switching functions, to export a signal function's remainder as a signal without defining that signal's type recursively, as we defined *KeepSF a b*. The result of *kSwitch* provides its own remainder to the continuation *k*, which just produces a remainder of the same type. Therefore that type must account for all output signals of the result of *kSwitch*. Since *k*'s purpose is to turn the remainder into a signal, one of those output signals must be the remainder itself. Therefore the type must be recursive.

References

[1] Manuel Bärenz and Ivan Perez. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 145–157, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[3] Gregory Bollella and James Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[4] Antony Courtney and Henrik Nilsson. Yampa: library for programming hybrid systems [source code]. https://hackage.haskell.org/package/Yampa-0.9.1.1, 2003. Retrieved March 2020.

[5] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery.

[6] Thijs Jeffry de Haas, Tim Laue, and Thomas Röfer. A scripting-based approach to robot behavior engineering using hierarchical generators. In *2012 IEEE International Conference on Robotics and Automation*, pages 4736–4741. IEEE, 2012.

[7] Angela Di Febbraro, Davide Giglio, and Nicola Sacco. Urban traffic control structure based on hybrid Petri nets. *IEEE Transactions on Intelligent Transportation Systems*, 5(4):224–237, 2004.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, page 263–273, New York, NY, USA, 1997. Association for Computing Machinery.

[9] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[10] Diego Garcés. Scripting language survey. *Game Programming Gems*, 6(2006):323–340, 2006.

[11] George Giorgidze and Henrik Nilsson. Switched-on Yampa. In *International Symposium on Practical Aspects of Declarative Languages*, pages 282–298. Springer, 2008.

[12] Rafal Goebel, Ricardo G Sanfelice, and Andrew R Teel. Hybrid dynamical systems. *IEEE Control Systems Magazine*, 29(2):28–93, 2009.

[13] Gregory D Hager and John Peterson. Frob: A transformational approach to the design of robot software. In *Robotics Research*, pages 257–264. Springer, 2000.

[14] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67 – 111, 2000.

[15] Mike McShaffry and David Graham. *Game Coding Complete*. Course Technology, Boston, 2013.

[16] Pieter J Mosterman and Gautam Biswas. A theory of discontinuities in physical system models. *Journal of the Franklin Institute*, 335(3):401–439, 1998.

[17] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 51–64, New York, NY, USA, 2002. Association for Computing Machinery.

[18] John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[19] Ross Paterson. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, page 229–240, New York, NY, USA, 2001. Association for Computing Machinery.

[20] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, page 168–179, New York, NY, USA, 2002. Association for Computing Machinery.

[21] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA, 2016. Association for Computing Machinery.

[22] John Peterson and Greg Hager. Monadic robotics. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 95–108, 1999.

[23] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, page 91–105, Berlin, Heidelberg, 1999. Springer-Verlag.

[24] Miro Samek. Who moved my state? *Dr. Dobb's Journal*, April 2003.