Settable and Non-Interfering Signal Functions for FRP

How a First-Order Switch is More Than Enough

Daniel Winograd-Cort
Yale University
dwc@cs.yale.edu

Paul Hudak
Yale University
paul.hudak@yale.edu

Abstract

Functional Reactive Programming (FRP) provides a method for programming continuous, reactive systems by utilizing *signal functions* that, abstractly, transform continuous input signals into continuous output signals. These signals may also be *streams of events*, and indeed, by allowing signal functions themselves to be the values carried by these events (in essence, signals of signal functions), one can conveniently make discrete changes in program behavior by "switching" into and out of these signal functions. This higher-order notion of switching is common among many FRP systems, in particular those based on arrows, such as Yampa.

Although convenient, the power of switching is often an overkill and can pose problems for certain types of program optimization (such as *causal commutative arrows* [14]), as it causes the structure of the program to change dynamically at run-time. Without a notion of just-in-time compilation or related idea, which itself is beset with problems, such optimizations are not possible at compile time.

This paper introduces two new ideas that obviate, in a predominance of cases, the need for switching. The first is a *non-interference law* for arrows with choice that allows an arrowized FRP program to dynamically alter its own structure (within statically limited bounds) as well as abandon unused streams. The other idea is a notion of a *settable signal function* that allows a signal function to capture its present state and later be restarted from some previous state. With these two features, canonical uses of higher-order switchers can be replaced with a suitable first-order design, thus enabling a broader range of static optimizations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Functional Reactive Programming; Arrows; ArrowChoice; Switch

1. Introduction

Functional Reactive Programming (FRP) is based on the idea of programming with *signals*, or time-varying values. Signals can be continuous, in which case they are defined for every moment in time, or they can be discrete event streams, in which case they are defined at particular moments. The FRP model allows one to essentially define behaviors for these streams, using signal functions that react as the streams change over time.

A problem with classic FRP systems (such as Fran [6]) is their propensity toward space and time leaks [15]. One method for addressing these leaks is by using *arrows* [12, 13] in so called *arrowized* FRP (AFRP), which has been used in *Yampa* [3, 11] (for animation, robotics, GUI design, and more), *Nettle* [18] (for networking), and *Euterpea* [10] (for audio processing and sound synthesis). In AFRP, instead of treating the signal as a first class value, one treats the *signal function* as the core component. The arrow structure then allows the signal functions to be composed quite naturally.

Furthermore, the arrow abstraction lends itself well to aggressive optimizations. An arrow's structure must be defined statically, and once defined, it cannot be altered mid-computation. Therefore, regardless of what data the signals contain, the arrow's overall behavior is fixed. For example, CCA [14] relies on this restriction to optimize an FRP program and often improve its performance in GHC by an order of magnitude.

1.1 Switch

One problem with arrows is that they do not naturally have the full capabilities that classic FRP provides. As mentioned, an arrow's structure must be fixed at compile-time, but classic FRP provides behavior-switching mechanisms. Thus, arrows are typically augmented with a higher-order *switch* operator to recover this ability.

Switching allows a program to accept and utilize a stream of signal functions, thus allowing for higher-order signal function expression in which the program can update its own structure during execution. Additionally, in the realm of signal functions, a higher-order ability like this provides the only means of starting and stopping signals mid-computation, which is often a necessity for good performance. For instance, new signal functions can be provided at runtime and "switched on" to augment the current behavior of a program. Likewise, given an event that a certain signal is no longer needed, the program can "switch off" the portion of itself that is computing values for that signal, thus preventing unneeded computations from being performed. In fact, arrows with switch are as powerful as *ArrowApply* arrows, which are equivalent to monads [12].

Unfortunately, this power comes at a cost: the inherent higher-order nature of switch that allows it to run arbitrary signal functions from a stream makes certain compile-time optimizations and static guarantees much more difficult or even impossible. For example, arrows with switch cannot undergo the CCA optimizations. Likewise, in the realm of embedded systems, where static code is required due to strict time and resource constraints, switch can be an intolerable hole in a static guarantee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2873-9 /14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628140

1.2 An Alternative to Switch

The motivation of this research is to ask whether switch is really necessary. Most FRP programmers would be reluctant to give it up – indeed, some FRP programs would be inexpressible with just first-order arrows – but perhaps there is an operator that is powerful enough to replace switch in most cases while still being weak enough to allow for CCA-like optimizations. In order to consider this, we first must examine more closely exactly what switching provides.

Switch allows one to express two fundamental behaviors that are otherwise impossible with just arrows. First, it provides a way for signal functions to dynamically start and stop mid-computation, which is useful not just for expressing certain programs but also for obtaining high performance. Second, it allows for higher-order signal expression, essentially providing a way to flatten a stream of streams into a single stream or insert a dynamic signal function into the arrow structure itself.

The first of these effects is similar to what is provided by *arrow choice*, which allows an arrow to choose between statically defined branches based on a dynamic argument. However, although the streaming argument will only be processed by one branch of an arrow choice conditional, every effect from the arrows from every branch will be executed. This means that arrow choice cannot be used to entirely suspend a branch in the way that switch can suspend a “switched out” signal function.

To address this, we can modify arrow choice by adding a new law in order to make it *non-interfering*. Non-interfering choice asserts that effects from only one branch of the choice will happen, and so if one branch is taken, it is as if the other does not exist.

Technically, non-interfering choice allows us only to pause signal functions and not actually start or stop them. For this reason, we additionally provide a method for making an arrow *settable*: a settable arrow’s state can be saved, reloaded, and even reset.

Combining settability with non-interfering choice gives us the full power of the first effect of switch. That is, we can “start” a signal function by using choice and then resetting its state, and we can “stop” a signal function by indefinitely pausing it.

Interestingly, non-interfering choice allows for another unforeseen benefit: arrowized recursion. Because only one branch’s effects can take place, we can do a form of recursion that allows behaviors that were previously only possible with switch. Combining this with settability allows for some surprising power.

1.3 Contributions

In this paper, we aim to show that switch is not essential to AFRP and that many powerful FRP programs that were previously believed to require switch may not actually need it. Indeed, we will take a number of example programs that utilize various forms of switchers in standard ways and show that our system is just as expressive. With this conclusion, we hope that we can open the doors to new and improved optimization techniques for arrows; we begin this process by demonstrating an extension to the CCA optimization that takes non-interfering choice and settability into account.

In the next section we will discuss arrows in general along with some details about the switch operators that we will be comparing our work against. Following that, we will make cases for both non-interfering choice and settability in Sections 3 and 4, in which we will show leading examples and present our first-order solutions. In Section 5, we will culminate our examples with a parallel choice example in the music domain that will bring together all of the topics so far discussed. From there, we will move into some implementation details, first describing our implementation of settability in Section 6 and then detailing an optimization for non-interfering choice in Section 7. Finally, we will present a brief, concluding discussion of the differences between our work here and switch in general as well as a comparison of this work to related work in Sections 8 and 9.

$$
\begin{array}{rl}
\mathit{arr} & :: (\alpha \to \beta) \to (\alpha \rightsquigarrow \beta) \\
\mathit{first} & :: (\alpha \rightsquigarrow \beta) \to ((\alpha,\gamma) \rightsquigarrow (\beta,\gamma)) \\
(\ggg) & :: (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma) \\
(|||) & :: (\alpha \rightsquigarrow \gamma) \to (\beta \rightsquigarrow \gamma) \to ((\alpha + \beta) \rightsquigarrow \gamma) \\
\mathit{loop} & :: ((\gamma,\alpha) \rightsquigarrow (\gamma,\beta)) \to (\alpha \rightsquigarrow \beta) \\
\mathit{delay} & :: \beta \to (\beta \rightsquigarrow \beta)
\end{array}
$$

Figure 1. The types of the arrow operators.

2. Arrows

2.1 Signal Processing

Programming with AFRP is a lot like expressing signal processing diagrams. Where signal processing diagrams have lines, AFRP has *signals*, and where diagrams have boxes that act on those lines, AFRP has *signal functions*. These signals can represent either continuously-defined time-varying values or streams of discrete events.

Because AFRP is based on arrows, we can use Paterson’s *arrow syntax* [17] to make programming with it easier. For example, we can turn this simple signal processing diagram:

into just as simple a code snippet:

$$y \leftarrow \mathit{sigfun} \prec x$$

In this example, *sigfun* is a signal function that takes the input stream *x* and produces the output stream *y*.

For this paper, we will use Haskell’s arrow syntax and operators to express code examples. Thus, the above code fragment cannot appear alone, but instead must be part of a **proc** construct. The expression in the middle must be a signal function, whose type we write as $\alpha \rightsquigarrow \beta$ for some types α and β. The expression on the right may be any well-typed expression with type α, and the expression on the left must be a variable or pattern of type β.

The purpose of the arrow notation is to allow the programmer to manipulate the instantaneous values of the signals. For example, the following is a definition for *sigfun* that integrates a signal and adds one to the output:

$$
\begin{array}{l}
\mathit{sigfun} = \textbf{proc}\ x \to \textbf{do} \\
\quad y \leftarrow \mathit{integral} \prec x \\
\quad \mathit{returnA} \prec y + 1
\end{array}
$$

The notation “**proc** $x \to$ **do** ...” introduces a signal function, binding the name *x* to the instantaneous values of the input. The second line sends the input signal into an integrator, whose output is named *y*. Finally, we add one to the value and feed it into the signal function *returnA*, that returns the result. The last line of this notation has no binding component – instead, whatever value is produced in the last line is returned in total.

Of course, one can use arrows without Haskell’s arrow syntax. Arrows are made up of three basic operators: construction (*arr*), partial application (*first*), and composition ($\ggg$). Furthermore, we extend our arrows with choice ($|||$) [12] to allow dynamic control flow, looping (*loop*) [17] to allow value-level recursion, and delay (*delay*). The types of these operators are shown in Figure 1.

For example, the signal function *sigfun* defined earlier can be written without arrow syntax as follows:

$$sigfun = integral \ggg arr\ (\lambda y.\ y + 1)$$

Note that *returnA* is defined simply as *arr id*, which is why it is used for clarity to return values in the last line of arrow syntax but is omitted from the above definition of *sigfun*. We will also use the function $constA :: \beta \to (\alpha \rightsquigarrow \beta)$ in this paper, which takes one static argument and returns a signal function that ignores its input stream and returns a constant stream of the given value.

Events and Event Streams

The classical interpretation of a signal of type α is that it is a function from time to α defined for all points in time. We call this a *continuous* signal. However, we frequently require the ability to define a signal that has values at only discrete points in time and is undefined elsewhere. These so-called *event streams* are represented by encapsulating the signal's type with an option type. For this paper, we will use the following:

$$\textbf{data}\ Event\ \alpha = Event\ \alpha \mid NoEvent$$

Note that we are overloading the name *Event* such that it is both the general type as well as the constructor for an event. Thus, if a signal has type *Event* α, then we know that it is defined when it provides an *Event* and undefined when it provides *NoEvent*.

In this paper, we will make use of the fact that *Event* is a functor in the obvious way and freely *fmap* functions over *Event* values.

2.2 State via loop and delay

A key component of FRP systems (AFRP included) is the ability to perform stateful computation. For example, Yampa includes the *integral* function that integrates its input signal, a process impossible without some form of internal state.

Although stateful signal functions can be achieved in a variety of ways, we follow Liu et al. [14] in the use of a *delay*[1] operator along with *loop*. In this model, we use the loop as a feedback mechanism, allowing an auxiliary output containing the state to be fed back as an input, and we use the delay to prevent an infinite feedback loop. Indeed, Liu et al. [14] even demonstrate that *integral* can be defined using this method:

$$\begin{aligned}&integral = \textbf{proc}\ x \to \textbf{do}\\ &\quad \textbf{rec}\ v \leftarrow delay\ 0 \prec v + dt * x\\ &\quad returnA \prec v\end{aligned}$$

Note here that the **rec** keyword in arrow syntax invokes the *loop* operator and that we assume *dt* is a global time step.

2.3 Switch

As discussed in the introduction, the ability to dynamically *switch* one signal function for another during the execution of a program is a staple of most FRP systems. Considering that one of our primary goals is to show an alternative to switching, we will here describe switch's capabilities.

The idea of switching was introduced along with the earliest models of FRP [6]. These non-arrowized FRP implementations had the ability to sequence periods of signal function execution, a process that is inherently monadic in nature. However, the move to the arrow abstraction would not allow this behavior, and to prevent any loss in expressiveness, Hudak et al. [11] introduced the *switch* function in Yampa.

Actually, Yampa includes some 14 different variations on the switch function ranging from the simplest switch to the recursive, parallel, batch-input, delayed switch. We will briefly examine three of these switchers.

Switch

The most basic switch function has the following type:

$$\begin{aligned}switch :: &(\alpha \rightsquigarrow (\beta, Event\ \gamma))\\ &\to (\gamma \to (\alpha \rightsquigarrow \beta))\\ &\to (\alpha \rightsquigarrow \beta)\end{aligned}$$

The first argument is the initial signal function that the result will behave as. When that signal function produces an event, the switch will use the data from that event along with its second argument to produce a new signal function. From then on, it will behave as that new signal function.

Recursive Switch

A slightly more advanced version of switching allows for the signal function to be switched out more than once:

$$\begin{aligned}rSwitch :: &(\alpha \rightsquigarrow \beta)\\ &\to ((\alpha, Event\ (\alpha \rightsquigarrow \beta)) \rightsquigarrow \beta)\end{aligned}$$

Here, the resulting signal function takes an event stream of signal functions along with the stream of input α values. When the event stream contains an event, it switches into the signal function contained in the event.

Parallel Switch

The parallel version of switch is significantly more intimidating from its type signature but also quite powerful:

$$\begin{aligned}pSwitch :: &\ Functor\ col\\ &\Rightarrow\ col\ (\alpha \rightsquigarrow \beta)\\ &\to ((\alpha, col\ \beta) \rightsquigarrow Event\ \gamma)\\ &\to (col\ (\alpha \rightsquigarrow \beta) \to \gamma \to (\alpha \rightsquigarrow col\ \beta))\\ &\to (\alpha \rightsquigarrow col\ \beta)\end{aligned}$$

The parallel switcher works on *collections* of signal functions, where a collection must be a *Functor*. First, it is given an initial collection of signal functions to run and a signal function that produces update events. The third argument takes the current collection of signal functions and the value from an event in order to produce a new collection of signal functions. In total, *pSwitch* will run every signal function in its collection and produce as output a collection of their results.

Note that any one of these versions of switch is strong enough to implement the others. The reason for Yampa's many varieties of switch is not due to power differences, but rather due to ease of use. That is, for example, using *switch* to do an operation that requires *rSwitch* is tedious, so both varieties are provided.

3. A Case for Non-Interfering Choice

We will begin this section by exploring one of the main uses of switchers: as a method to allow the dynamic starting and stopping of signal functions. We will present our first-order alternative and then demonstrate it in a few practical settings.

3.1 Pausable Signal Functions

At a basic level, switch is often used to improve performance of an AFRP program. Without switch, signal functions will last forever, and this typically means that they will compute future values indefinitely. Using switch, one can "turn off" signal functions that are not currently necessary and even turn them back on if they are required again in the future.

For example, consider the scenario where we would like to integrate a stream only when a certain condition holds. Naïvely, we can write the following program:

[1] Note that in [14], this operator is referred to as *init*.

$$
\begin{array}{l}
\mathit{integralWhen}_{\mathit{Naive}} :: (\mathit{Double}, \mathit{Bool}) \rightsquigarrow \mathit{Double} \\
\mathit{integralWhen}_{\mathit{Naive}} = \mathbf{proc}\ (i, b) \rightarrow \mathbf{do} \\
\quad v \leftarrow \mathit{integral} \prec i \\
\quad v_{\mathit{prev}} \leftarrow \mathit{delay}\ 0 \prec v \\
\quad \mathbf{let}\ v_{\Delta} = v - v_{\mathit{prev}} \\
\quad \mathbf{rec}\ \mathit{result} \leftarrow \mathit{delay}\ 0 \prec \mathbf{if}\ b\ \mathbf{then}\ \mathit{result} + v_{\Delta}\ \mathbf{else}\ \mathit{result} \\
\quad \mathit{returnA} \prec \mathit{result}
\end{array}
$$

This program will only update the result when the boolean is *True*, but it is still unsatisfying that the integral is being computed at all when it is not being used. If integral were instead a costly signal function and the boolean were usually *False*, this could be seriously problematic to performance.

In cases like this, switch can be employed to prevent the integral from running when it is not needed:

$$
\begin{array}{l}
\mathit{integralWhen}_{\mathit{Switch}} :: (\mathit{Double}, \mathit{Event\ Bool}) \rightsquigarrow \mathit{Double} \\
\mathit{integralWhen}_{\mathit{Switch}} = \mathbf{proc}\ (i, e_b) \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ v \leftarrow \mathit{rSwitch}\ (\mathit{constA}\ 0) \prec (i, \\
\qquad\qquad \mathit{fmap}\ (\lambda b \rightarrow \mathbf{if}\ b \\
\qquad\qquad\qquad\qquad \mathbf{then}\ (\mathit{integral} \ggg \mathit{arr}\ (+v)) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \ (\mathit{constA}\ v))\ e_b) \\
\quad \mathit{returnA} \prec v
\end{array}
$$

For this version, we modified the type to make it more amenable to switching by converting the streaming boolean value to an event stream that will send events only when the stream would change from *True* to *False* or back. Internally, we use the *rSwitch* function that we introduced in Section 2.3 to switch between *integral* and a constant function. Each time we switch into *integral*, it is fresh and has no history from the last time we were using *integral*, so we additionally compose it with $\mathit{arr}\ (+v)$ so it can maintain its history.

3.2 Non-interfering Choice

Although the above example is a fairly common use for switch, careful examination of the problem reveals that switch is far more powerful that necessary. That is, while switch allows us to dynamically incorporate new signal functions into the running computation, here, we are simply making a *choice* of whether to run a component signal function based on a dynamic value. Our solution to this problem will thus be built around arrow choice, so we will begin by examining it more closely.

The general choice operator we use ($|||$ in Figure 1) can actually be built from a simpler component:

$$\mathit{left} :: (\alpha \rightsquigarrow \beta) \rightarrow ((\alpha + \gamma) \rightsquigarrow (\beta + \gamma))$$

where $\mathit{left}\ f$ calls f when the input signal contains *Left* values and acts as the identity function otherwise. With the *left* function, we can also define an analogous *right* function and then use the two together to define $|||$.

Choice also comes with a set of laws that we show in Figure 2. For us, the most notable law is the *exchange* law, which acts as a weak form of commutativity between *left* functions and *right* functions. One may ask why choice does not demand full commutativity (i.e. $\mathit{left}\ f \ggg \mathit{right}\ g = \mathit{right}\ g \ggg \mathit{left}\ f$), and in the context of signal processing, this question is very sensible. After all, it seems intuitively obvious that either the *left* function or the *right* function will run, but in no case will both run. However, because arrows can have effects regardless of their dynamic inputs, and the compositional order of these effects can alter the program itself, choice is weakened. It is precisely this leniency that makes switching necessary in cases such as the above example.

$$
\begin{array}{lrcl}
\text{Extension} & \mathit{left}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (\mathit{left}\ f) \\
\text{Functor} & \mathit{left}\ (f \ggg g) & = & \mathit{left}\ f \ggg \mathit{left}\ g \\
\text{Exchange} & \mathit{left}\ f \ggg \mathit{arr}\ (\mathit{right}\ g) & = & \mathit{arr}\ (\mathit{right}\ g) \ggg \mathit{left}\ f \\
\text{Unit} & f \ggg \mathit{arr}\ \mathit{Left} & = & \mathit{arr}\ \mathit{Left} \ggg \mathit{left}\ f \\
\text{Assoc.} & \mathit{left}\ (\mathit{left}\ f) \ggg \mathit{arr}\ \mathit{assoc}_{+} & = & \mathit{arr}\ \mathit{assoc}_{+} \ggg \mathit{left}\ f
\end{array}
$$

$$
\begin{array}{ll}
\mathit{assoc}_{+}\ (\mathit{Left}\ (\mathit{Left}\ x)) & = \mathit{Left}\ x \\
\mathit{assoc}_{+}\ (\mathit{Left}\ (\mathit{Right}\ y)) & = \mathit{Right}\ (\mathit{Left}\ y) \\
\mathit{assoc}_{+}\ (\mathit{Right}\ z) & = \mathit{Right}\ (\mathit{Right}\ z)
\end{array}
$$

$$\text{Non-interference} \quad \mathit{arr}\ \mathit{Right} \ggg \mathit{left}\ f \ = \ \mathit{arr}\ \mathit{Right}$$

Figure 2. The standard laws for arrow choice with our new non-interference law below.

In order to give choice the extra power it needs to be an adequate replacement for switch, we strengthen the *exchange* law into the more powerful:

$$\text{Non-interference} \qquad \mathit{arr}\ \mathit{Right} \ggg \mathit{left}\ f \ = \ \mathit{arr}\ \mathit{Right}$$

Indeed, *non-interference* implies exchange and even commutativity as it is stronger than either (see Appendix A for details). It states that once the streaming value is tagged as a *Right* value, then it will not be applicable to $\mathit{left}\ f$, and so it should behave as if the $\mathit{left}\ f$ is not even there. Thus, by including the non-interference law for choice, we assert that either signal functions cannot have static effects or that the choice operation has the power to dynamically choose which effects to perform.

3.3 Pausable Signal Functions Revisited

With non-interfering choice in our arsenal, we can define a new version of *integralWhen* in an even more intuitive and straightforward way:

$$
\begin{array}{l}
\mathit{integralWhen}_{\mathit{Choice}} :: (\mathit{Double}, \mathit{Bool}) \rightsquigarrow \mathit{Double} \\
\mathit{integralWhen}_{\mathit{Choice}} = \mathbf{proc}\ (i, b) \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ v \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \mathit{integral} \prec i \\
\qquad\qquad\qquad \mathbf{else}\ \ \mathit{returnA} \prec v \\
\quad \mathit{returnA} \prec v
\end{array}
$$

Because we are not actually switching out of the *integral* signal function, it will retain its state internally. When it is executed, it will calculate and add the latest delta of integral, and otherwise, it will simply wait.

3.4 A Single First-Order Switch

The most basic switching operation is to non-recursively switch out one signal function for another dynamically. For example, we could write a simple guessing game that accepted an event stream of guesses, and when the correct answer was provided, it would switch into a signal function that ignored its input and declared that the game was over:

$$
\begin{array}{l}
\mathit{guess} :: \mathit{Event\ Int} \rightsquigarrow () \\
\mathit{guess} = \mathit{switch}\ (\mathit{arr}\ f)\ (\lambda\ t \rightarrow \mathit{label}\ t) \\
\quad \mathbf{where}\ \ f\ (\mathit{Event}\ i) | (i == 3) = ((),\ \mathit{Event}\ \text{“You Win!”}) \\
\qquad\qquad\ \ f\ \_ \qquad\qquad\qquad\ \ = ((),\ \mathit{NoEvent})
\end{array}
$$

where *label* is a signal function widget that ignores its streaming input and displays the text it was given as its static argument. Note that we are using the plain, non-recursive, non-parallel version of switch that we presented in Section 2.3. In *guess*, when the event containing 3 is processed, the string “You win!” is given to the label, and the guessing is switched out for that label.

For this example again, switch is too strong. Notice that the argument given to the switched-in signal function is not itself a

```
runNTimes :: Int → (α ⇝ β) → ([α] ⇝ [β])
runNTimes 0 _   = constA [ ]
runNTimes n sf = proc (b : bs) → do
  c ← sf −≺ b
  cs ← runNTimes (n − 1) sf −≺ bs
  returnA −≺ (c : cs)
```

Figure 3. The implementation of *runNTimes* using structural recursion.

signal function. In fact, it's just a constant! We can rewrite this with non-interfering choice:

```
guess_choice :: Int ⇝ ()
guess_choice = proc i → do
  rec haveWon ← delay False −≺ haveWon || (i == 3)
  if haveWon then label "You Win!" −≺ ()
             else  returnA −≺ ()
```

Note that we changed the input stream to a continuous stream as opposed to an event stream simply to make the example clearer.

Reacting to dynamic events

The above versions of *guess* are quite primitive, and although we use switching in the first one, we are far from using its full power. We can make the example slightly more complex by adding an additional component to the input such that the program is actually reactive:

```
guess′ :: Event (Int, String) ⇝ ()
guess′ = switch (arr f) (λ t → label t)
  where f (Event (i, s))|(i == 3) = ((), Event s)
        f _                       = ((), NoEvent)
```

In *guess*$'$, the text to put in the label is no longer static and instead is part of the guess event, and in its current form, switching is a necessity as it is the only way to provide the dynamically streaming string to the static *label* function. However, we could once again lift the need for switching if we could redesign the label to instead take an *impulse*. An impulse is a one time event that initializes a signal function, so in this case, the type for *label* would change from $String \to (\alpha \rightsquigarrow ())$ to $(Event\ String) \rightsquigarrow ()$.

With an impulse driven label widget, we can once again convert the *guess*$'$ function to a switch-free alternative:

```
guess′_choice :: (Int, String) ⇝ ()
guess′_choice = proc (i, s) → do
  rec haveWon ← delay False −≺ haveWon || (i == 3)
  let imp = if not haveWon && i == 3
            then Event s else NoEvent
  if haveWon then label −≺ imp
             else  returnA −≺ ()
```

3.5 Arrowized Recursion

As we have shown in the previous two examples, there is a direct usage for non-interfering choice, but the non-interference law also gives us a less obvious benefit. By restricting the arrow effects to only one branch, we open the door to the possibility of a new kind of recursion.

Typically, arrows can perform recursive behaviors in one of two ways. First, arrows can use the *loop* functionality to perform a value level recursion, or a sort of fix point recursion. After all, one of the laws for *loop* is:

$$loop\ (arr\ f) = arr\ (\lambda\ b \to fst\ (fix\ (\lambda(c,d) \to f\ (b,d))))$$

```
runDynamic :: (α ⇝ β) → ([α] ⇝ [β])
runDynamic sf = proc lst → do
  case lst of
    [ ]      → returnA −≺ [ ]
    (b : bs) → do c ← sf −≺ b
                  cs ← runDynamic sf −≺ bs
                  returnA −≺ (c : cs)
```

Figure 4. The implementation of the choice-based *runDynamic* function using arrowized recursion.

Second, there is *structural* recursion. Structural recursion happens when the host language's recursion is used to create an arrow in a recursive way. For instance, we might have a function like:

$$runNTimes :: Int \to (\alpha \rightsquigarrow \beta) \to ([\alpha] \rightsquigarrow [\beta])$$

When defining this function, we use Haskell's conditional syntax to recur on the value of the first argument: while it is greater than zero, we run the signal function and recur, and when it is equal to zero, we return a constant stream of the empty list. We show a definition of *runNTimes* using this form of recursion in Figure 3.

A key frustration with structural recursion is that the recursive argument is static as opposed to streaming. Thus, structural recursion is often performed in tandem with higher-order switching to allow a streaming value to be used in place of the static argument.

With, non-interfering choice, we extend arrows with a new kind of recursion that we call *arrowized* recursion. Arrowized recursion is very similar to structural recursion except that instead of using the host language's conditional, we use arrow choice. Ordinarily, this would be impossible: because all branches of an arrow choice must be executed for their effects, if one were recursive, then it would cause an infinite loop. However, non-interference gets around this by restricting arrow effects dynamically.

Thus, with arrowized recursion, we can write a function similar to the above *runNTimes* but that needs no static argument to perform its recursion. In fact, we can make the input stream of lists the recursive argument and eliminate the need for an "N" altogether. We call this function *runDynamic* and show it in Figure 4.

3.6 Dynamic GUI

One power of switch, showcased particularly in *Fruit* [2], is the ability to allow a dynamic number of signal functions to execute. That is, by default, arrows have a fixed structure, and the streaming values moving through an AFRP program cannot affect that structure. However, switch allows one to dynamically alter the arrow at runtime based on the streaming values.

For example, one may desire a GUI that gathers the names of an unknown group of people. If the size of the group were fixed or at least known at compile time, then this is achievable trivially with arrows, but if the size is a parameter that is filled in by the user of the GUI, then standard arrows are stymied. One approach is to use a switching mechanism.

For this example, we will assume a few GUI widgets:

```
label       :: String → (() ⇝ ())
getInteger  :: () ⇝ Int
getIntegerE :: () ⇝ Event Int
getName     :: () ⇝ String
```

Note that we have both a regular and event-based version of *getInteger*: the event-based one, which produces an event each time the value changes, is useful for our example with switch, and we will use the regular one with choice.

We can use these widgets in combination with the *rSwitch* function to make our GUI:

```
getNames :: () ⇝ [String]
getNames = proc () → do
  _ ← label "How many people?" ⤙ ()
  e_n ← getIntegerE ⤙ ()
  rSwitch (constA [ ]) ⤙ (repeat (),
      fmap (λ n → runNTimes n getName) e_n)
```

where the *runNTimes* function is the one we discussed in the previous subsection (that uses structural recursion to run the given signal function the given number of times, as shown in Figure 3).

The above definition of *getNames*, although correct, is using the higher order nature of switch when it is not truly necessary. Switching gives the power to substitute in any new signal function for the currently running one, but here, the nature of the new signal function is already known: it will be some number of *getName* widgets. Because this fact is known at compile time, we can use arrowized recursion instead to create a simpler, switch-free GUI.

```
getNames :: () ⇝ [String]
getNames = proc () → do
  _ ← label "How many people?" ⤙ ()
  n ← getInteger ⤙ ()
  runDynamic getName ⤙ replicate n ()
```

Because *runDynamic* uses arrow choice to do arrowized recursion, we do not need to use any switching.

4. A Case for Settability

In this section, we will explore a second main use of switchers: the ability to start a signal function mid-computation with no prior state. Once again, we will begin with a simple yet canonical example before describing our first-order alternative and some further usage examples.

4.1 Restartable Computation

Although pausing signal functions is useful (as in the *integralWhen* example of Sections 3.1 and 3.3), there are times when we really do want to restart a signal function, resetting its state to its initial defaults. In fact, with switching, this is even easier than pausing considering that switch naturally starts its new signal function from the beginning.

For instance, let us consider the scenario where we would like to take the integral of a stream, but at any moment, we may be given an event that indicates that we should reset the integral's accumulation to its initial default. With switch, this is actually trivial: we simply lift the *integral* function into the resetting event, and send everything into a recursive switcher:

```
integralReset_Switch :: (Double, Event ()) ⇝ Double
integralReset_Switch = proc (i, e) → do
  rSwitch integral ⤙ (i, fmap (const integral) e)
```

Without switch, this seems like a tough problem, and nothing about non-interfering choice lends any help.

One idea is to try to simulate the behavior of a restart without actually touching *integral* itself. That is, because the function we are lifting is just an integral, we could take a snapshot of its output at the restarting moment and then continuously subtract that value from future outputs:

```
integralReset_Basic :: (Double, Event ()) ⇝ Double
integralReset_Basic = proc (i, e) → do
  o ← integral ⤙ i
  rec k ← delay 0 ⤙ k′
      let k′ = if isEvent e then o else k
  returnA ⤙ o − k
```

Although this is a valid solution to this particular situation, it is a technique that does not scale well to more complicated problems.

4.2 Settability

At this point, the idea of lifting a signal function into the event stream, as we did in *integralReset*$_{Switch}$ above, should seem unnecessary. Indeed, we are not even switching into some dynamically given new signal function but rather just using a new instance of the same signal function again. Rather than switching, our first-order approach is to develop a notion of signal function *settability*, or a way to change the internal state of a signal function at arbitrary points.

Because we are dealing with state, we will begin with an even more primitive example and examine the *delay* operator directly. At first glance, it seems to suffer from the same problem as *integral* – the *delay* will always output old values, so what can we do to reset it? However, modifying it to be resettable requires only the addition of a single input event stream:

```
resettableDelay :: β → ((β, Event ()) ⇝ β)
resettableDelay i = proc (b, e) → do
  out ← delay i ⤙ b
  returnA ⤙ case e of
    NoEvent → out
    Event () → i
```

Whenever *resettableDelay* is given an event, it will immediately output its initial value again, essentially behaving as if it has only just started. In fact, we can take this one step further and construct a version of *delay* that can be set to any value of our choosing:

```
settableDelay :: β → ((β, Event (Maybe β)) ⇝ β)
settableDelay i = proc (b, e) → do
  out ← delay i ⤙ b
  returnA ⤙ case e of
    NoEvent         → out
    Event Nothing   → i
    Event (Just s)  → s
```

With *settableDelay*, the event stream can potentially carry a new value to set the internal state, and if there is no value, we perform a reset. It may seem superfluous to have an event of an option, but adding the ability to set the state does not make resetting the state obsolete.

A fortuitous bonus to this function is that, in addition to being able to set the state, we can also capture the current state. That is, because the input stream is necessarily setting the new current state, it can also be made to provide it directly. Thus, we can use *settableDelay* to both "store" and "load" state.

General Settability

Although a settable version of *delay* may be useful on its own, it would be much more useful to have any arbitrary signal function be settable. However, this would require manually changing every internal *delay* operator to its settable alternative and then properly routing the state-setting events to the appropriate places. Additionally, if capturing the state at a given moment were important, all of the inputs to the *delay* functions would also need to be grouped and appropriately routed to the output. This would be exceptionally cumbersome and not at all feasible. What we want is a function like:

```
settable :: (α ⇝ β) → ((α, Event State) ⇝ (β, State))
```

that will automatically take a signal function and allow us to both pass in an optional new state as well as save its current state.

This *settable* function should hold to certain principles of behavior. For example, if it is never provided with a state, then it

$$settable :: (\alpha \rightsquigarrow \beta) \to ((\alpha, Event\ State) \rightsquigarrow (\beta, State))$$

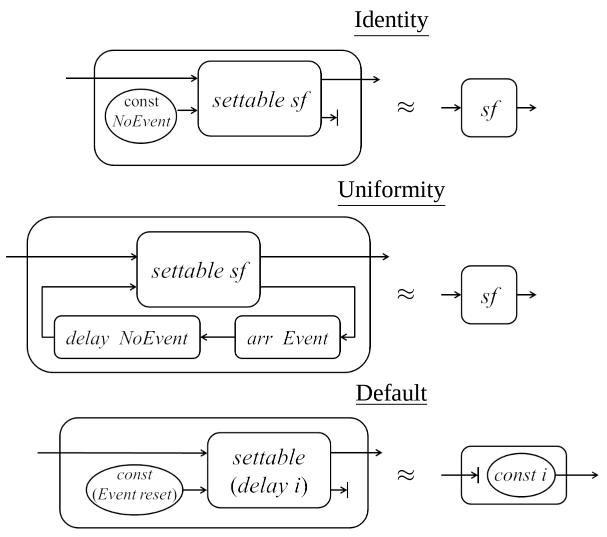

Figure 5. The *settable* function and its laws.

should do nothing. Similarly, if the state it produces is used to set it, then there should be no observable difference in behavior. Additionally, there should be a particular value of *State* that acts as a *reset* (in our *settableDelay* function from earlier, this was *Event Nothing*). Thus, if one were to feed a constant stream of reset states, the output would always use the default values. We declare these principles as laws of behavior for *settable* and show them diagrammatically in Figure 5.

In fact, with an appropriate code transformation, any arrow can be extended with a *settable* function. We will explore the details of this transformation in Section 6, but for now, it suffices to state that it is possible and available in our examples.

4.3 Restartable Computation Revisited

With the *settable* function, defining *integralReset* is just as trivial as with switch:

```
integralReset :: (Double, Event ()) ⇝ Double
integralReset = proc (i, e) → do
  (v, s) ← settable integral ─≺ (i, fmap (const reset) e)
  returnA ─≺ v
```

Rather than lifting a dynamic signal function to the signal level just to be activated by switch as we did previously, we lift only a reset signal. The difference in the amount of code between this function and $integralReset_{switch}$ is negligible (it basically comes down to ignoring the state output of the settable signal function), but the conceptual difference is quite important: rather than needing to stop a currently running signal function to replace it with a new, fresh instance of itself, it is possible to refresh it while leaving it active.

4.4 Freezing and Duplicating

This *settable* function has applications beyond just resetting arbitrary, stateful signal functions. By separating the state from the signal function, we are essentially separating the current behavior from the structure. That is, the *settable* function gives us the power to *freeze* signal functions.

Typically freezing a signal function is thought of as a higher-order operation achievable only with a switch operator. Specifically, freezing is the process of stopping a running signal function mid-execution and providing it as a piece of data to reuse. Later, it can be resumed by using a switcher to reintegrate it into the structure of the program.

```
gui :: () ⇝ ()
gui = proc () → do
  e_dup ← button "Duplicate pane?" ─≺ ()
  e_del ← button "Delete pane?" ─≺ ()
  i ← choosePane ─≺ ()
  rec stateLst ← delay [reset] ─≺ stateLst_new
      ((), state_new) ← settable drawing ─≺ ((), stateLst !! i)
      let stateLst_new = case (e_dup, e_del) of
              (Event (), _) →  set i stateLst state_new ++ [state_new]
              (NoEvent, Event ()) →  delete i stateLst
              _ →  set i stateLst state_new
  returnA ─≺ ()
```

Figure 6. The implementation of the GUI from Section 4.4.

Rather than providing a copy of itself, a function made settable will provide a stream of its *essence* (i.e. its current state), which can then be reinserted at any time later. Thus, we gain the ability to freeze and resume without actually resorting to switch.

Example

For this example, we will construct a GUI for drawing. The main window will feature a drawing pane, but the user will be able to create new panes and switch between them. When a new pane is created, it is automatically populated with a copy of whatever is currently on the current pane.

For this example, we will assume a few widgets:

```
drawing    :: () ⇝ ()
choosePane :: () ⇝ Int
button     :: String → (() ⇝ Event ())
```

The *drawing* widget is a stateful, effectful widget that provides a canvas and allows the user to draw; the *choosePane* widget returns an *Int* stream that represents the currently selected pane; and the *button* widget takes a static label and produces an event stream that indicates when the button is depressed.

With these widgets, we can create the GUI we described (shown in Figure 6). The state for the GUI is kept as a list of drawing states, initialized in the sixth line as a one element list containing a *reset* state. This initial list describes a GUI with a single pane that has a blank drawing canvas. When a user wishes to duplicate the current pane, the current state is added to the list allowing the GUI to "save" the original pane while providing a duplicate state for the new one. The key here is that instead of keeping track of different instances of the signal function, each with its own state, we keep track of multiple states themselves and use them with a single signal function.

We omit a version of this GUI that utilizes switching because it is surprisingly complicated, and it is not particularly necessary to contrast it with the GUI we present in Figure 6.

5. An Alternative to *pSwitch*

Here, we will pull together the ideas of both settability and non-interfering choice that we have highlighted in the previous sections to present a high power yet first-order version of a parallel switcher.

As we mentioned in Section 2.3, parallel switchers allow for whole collections of signal functions to be managed and switched in or out at once. One example of the usefulness of this kind of switcher can be seen in the musical realm where one might have a program that plays music with software "instruments" that are actually themselves signal functions. The music is given as a sequence of "On" and "Off" events, where the "On" events provide the instrument to play and some initializing data about what note to play, and the "Off" events tell which instrument to stop:

$$\begin{aligned}&\textbf{data}\ \mathit{NoteEvt} = \mathit{NoteOn}\ \mathit{UID}\ \mathit{Instr}\ \mathit{InitData}\\ &\qquad\qquad\qquad \mid\ \mathit{NoteOff}\ \mathit{UID}\ \mathit{Instr}\\ &\textbf{type}\ \mathit{Instr} = \mathit{InitData} \to (() \rightsquigarrow \mathit{Sound})\\ &\mathit{sumSound} :: [\mathit{Sound}] \rightsquigarrow \mathit{Sound}\end{aligned}$$

Note that the *UID* type is a unique identifier that is used to connect a given *NoteOn* event with its *NoteOff* counterpart, and the *Sound* data type represents the sound that an instrument produces. The *sumSound* signal function is for summing dynamic lists of sounds together.

Although we will use the same *pSwitch* that we introduced in Section 2.3, for clarity, we will show its type signature again, this time with a few of the type variables instantiated for our example.

$$\begin{aligned}\mathit{pSwitch} :: {} & [\mathit{UID}, () \rightsquigarrow \beta]\\ & \to (() \rightsquigarrow \mathit{Event}\ \gamma)\\ & \to ([\mathit{UID}, () \rightsquigarrow \beta] \to \gamma \to [\mathit{UID}, () \rightsquigarrow \beta])\\ & \to (() \rightsquigarrow [\beta])\end{aligned}$$

For our collection, we use a mapping of *UID* to signal function (which we implement as a list for simplicity), and we set α to $()$.

For this musical example, the initial list of signal functions will be empty, the events to change that list will be *NoteEvt*s, and the function will use the *NoteEvt* data to add or remove signal functions from the list as necessary:

$$\begin{aligned}&\mathit{maestro} :: (() \rightsquigarrow \mathit{Event}\ [\mathit{NoteEvt}]) \to (() \rightsquigarrow \mathit{Sound})\\ &\mathit{maestro}\ \mathit{music} = \mathit{pSwitch}\ [\,]\ \mathit{music}\ \ f \ggg \mathit{sumSound}\\ &\quad \textbf{where}\ f\ \mathit{lst}\ [\,] = \mathit{lst}\\ &\qquad\quad f\ \mathit{lst}\ (\mathit{NoteOn}\ \ u\ i\ \mathit{imp} : \mathit{rst}) = f\ ((u, i\ \mathit{imp}) : \mathit{lst})\ \mathit{rst}\\ &\qquad\quad f\ \mathit{lst}\ (\mathit{NoteOff}\ u\ i : \mathit{rst}) = f\ (\mathit{filter}\ ((\neq u)\ .\ \mathit{fst})\ \mathit{lst})\ \mathit{rst}\end{aligned}$$

In order to remove our reliance on switch, we need to make a few small changes to the layout of the problem. First, as we did in Section 3.4, we will need to change the instruments from functions that take a "static" initializing argument to functions that take that argument as an impulse. Second, we need to know statically what the different signal functions are, so we make use of a finite data type and add one layer of indirection:

$$\begin{aligned}&\textbf{data}\ \mathit{Instr} = \mathit{Trumpet} \mid \mathit{FHorn} \mid \mathit{Trombone} \mid \mathit{Tuba}\\ &\textbf{type}\ \mathit{Instrument} = \mathit{Event}\ \mathit{InitData} \rightsquigarrow \mathit{Sound}\\ &\mathit{toInstrument} :: \mathit{Instr} \to \mathit{Instrument}\end{aligned}$$

Because the *Instr* type is finite, we know exactly which *Instrument* signal functions can possibly be called. This is critical because choice is not actually higher order. Fortunately, in most situations where parallel switching is used, the possibilities of signal functions are known statically, so a transformation like this one is not difficult.

With these changes made, we can utilize the *pChoice* function. The idea behind *pChoice* is that as long as we know the possible signal functions that we may use, we can run each one a dynamic number of times. So, rather than keep a dynamic list of signal functions, we keep a static list of signal functions and a dynamic list of signal function *states*. We then use a combination of structural and arrowized recursion: structural recursion to provide access to each possible signal function and arrowized recursion to allow a dynamic number of runs per possibility.

The type of *pChoice* is:

$$\begin{aligned}\mathit{pChoice} :: \mathit{Eq}\ \mathit{key} \Rightarrow [(\mathit{key}, \mathit{Event}\ \alpha \rightsquigarrow \beta)] \to\\ ([(\mathit{key}, (\mathit{UID}, \mathit{Event}\ \alpha))] \rightsquigarrow [\beta])\end{aligned}$$

and as it is somewhat complicated, we leave its implementation and a more detailed description of its inner-functioning to Appendix B.

We can use *pChoice* to reimplement our music program without switch:

$$\begin{aligned}&\mathit{maestro} :: [\mathit{NoteEvt}] \rightsquigarrow \mathit{Sound}\\ &\mathit{maestro} = \mathit{arr}\ (\mathit{map}\ \ f) \ggg \mathit{pChoice}\ \mathit{lst} \ggg \mathit{sumSound}\\ &\quad \textbf{where}\ \mathit{lst} = \mathit{map}\ (\lambda\ i \to (i, \mathit{toInstrument}\ i))\ \mathit{allInstrs}\\ &\qquad\quad f\ (\mathit{NoteOn}\ \ u\ i\ \mathit{imp}) = (i, (u, \mathit{Event}\ \mathit{imp}))\\ &\qquad\quad f\ (\mathit{NoteOff}\ u\ i) \qquad = (i, (u, \mathit{NoEvent}))\end{aligned}$$

where *allInstrs* is a complete list of all of the *Instr*s that might be played. In fact, one notable difference between this version of *maestro* and the switch-based alternative from earlier is this *allInstrs* list: the reason that we can write this program at all is because *allInstrs* can be defined statically.

6. Implementing Settability

As we mentioned in Section 4.2, we can achieve settability of any arrow with a code transformation. Here, we will provide a detailed description of the transformation process before presenting Haskell code that implements it.

6.1 Design

In essence, the idea of settability is the idea of having access to the internal state of an arrow. Thus, as we discussed previously, it is encapsulated by a function like:

$$\mathit{settable} :: (\alpha \rightsquigarrow \beta) \to ((\alpha, \mathit{Event}\ \mathit{State}) \rightsquigarrow (\beta, \mathit{State}))$$

that will automatically take a signal function and allow us to both pass in an optional new state as well as save its current state. However, in order to achieve this, we will need to rewrite the underlying arrow to support this behavior. Therefore, we will describe a recursive transformation that will provide settable capabilities to ordinary arrows.

Intuitively, this settability transformation is a simple process of routing state update information in through the various arrow combinators so that it can be easily accessed by any internal delay operators and then routing current state data back out through the combinators to the level of the *settable* call. For each combinator, there is a transformation that achieves exactly this goal; we show circuit diagrams for these transformations in Figure 7 and describe them in detail below. Note that we use the notation $\overline{sf}$ to denote the signal function *sf* after having been transformed, and we assume that the *Event State* input stream and *State* output stream are always the lower input and output.

- We will begin at the lowest level by examining the *delay* operator itself. In Section 4.2, we showed a design for a settable version of delay, but we need to modify it just slightly in order for it to be general enough for our *settable* transformation: in addition to taking in an *Event State* stream, it also needs to emit its current *State* as a stream. This is rather trivial as its current state is identical to its own input stream, but this is important to the transformation as a whole. Thus, our circuit diagram shows the input stream both being sent to the embedded *delay* operator as well as being duplicated to the *State* output, and the output is determined by a case analysis of the *Event State* input with data from the *delay*'s output.
- The simplest transformation is that of the *arr* operator, which has no state and should essentially remain unaffected. In this

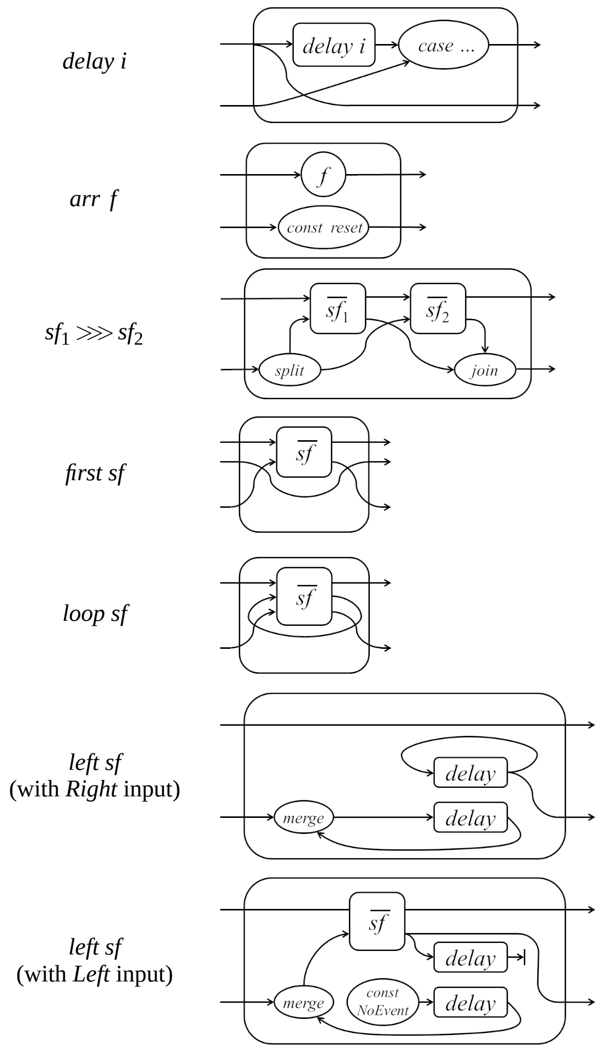

Figure 7. The circuit diagrams showing the settability transformations for the various arrow combinators.

case, we ignore the input *Event State* and return a constant stream of the null, or *reset*, state.

- The composition of two functions is a little more interesting. Each of the two composed signal functions may have state, so we need to split the incoming *Event State* into two pieces and pass the first to the first signal function and the second to the second. We gather the resulting states together and join them into a single output state.
- Applying a partial application (*first*) is a simple matter of rerouting the state data and the unused input stream properly.
- Looping is handled similarly to partial application with a simple rerouting of streams.
- The most complicated transformation is for our non-interfering choice's *left* operator. This is because there are two difficult questions that we must address in designing this transformation. First, in the case of an input *Right* value, the embedded signal function is not executed, so where can we get a *State* value for the output *State* stream? And second, again in the case of an input *Right* value, if we are given an *Event State* that requires updating the embedded signal function, how can we get that event where it needs to go? The way to address both of these questions is to allow the transformed choice operator to contain some internal state, which we achieve with *loop* and *delay*.

  Furthermore, in an effort to clarify the behavior of the transformed choice, we provide two diagrams to describe its behavior: one that shows how it behaves when given a *Right* value and the other for when it is given a *Left* value. The *delay*s are shared between both diagrams: the upper *delay* should be assumed to be initialized with a *NoEvent* value and the lower with a null, or *reset*, state value. The *merge* function is a standard overwriting event merge that favors the left (newly incoming) event in the case of two events.

  When given a *Right* input, the input stream is identical to the output stream. The *Event State* input is merged with the stored *Event State* and stored once again, thus updating the store with any new setting events. The output *State* is the stored one.

  When given a *Left* input, we will execute the embedded signal function. We still merge the *Event State* input with the stored one, but the result goes directly into the embedded signal function, and the store is instead updated with a *NoEvent*, indicating that there are no past *Event State*s waiting to be delivered. The output of the transformed, embedded signal function, both the streaming *Left* value as well as the output *State*, become the output of the overall transformed signal function, but the output *State* is also stored for potential future use. The stored *State* value is discarded outright as it is now obsolete.

6.2 Haskell Implementation

Rather than relying on Haskell's rewrite rules or Template Haskell, we can perform the entire transformation with only type classes. Our method involves creating a wrapper for a generic arrow that itself instantiates the arrow classes. Then, any code that is an arbitrary arrow could just as well be this wrapper.

Thus, our goal will be to concretely define our types and then instantiate the arrow classes using them.

Data Types

The first type we must choose a concrete representation for is the *State* data type. Although we could use Haskell's type families and other features to make a type-safe solution, its complexity would detract from the point. Therefore, to keep the types simple, we will make use of Haskell's *Dynamic* data type to store arbitrary state information from individual *delay* functions.[2] Also, rather than use an auxiliary option type to represent a default state or an absence of state (as we did in the *settableDelay* function in Section 4.2), we will build this directly into the type.

We show the definition of the *State* data type along with the few helper functions we need in Figure 8. Note that because *NoState* represents an absence of state information, trying to split it returns a similar lack of information.

With the *State* type defined, we next build our wrapper for a general arrow:

$$\textbf{data}\ SA\ (\leadsto)\ \alpha\ \beta = SA\ ((\alpha, \mathit{Event\ State}) \leadsto (\beta, \mathit{State}))$$

Already, we can see that this *SA* data type is merely hiding the extra piping that will be required to store and load the state.

[2] Technically, using *Dynamic* in this way enforces a *Typeable* restriction to the types of the individual state components, but this is of little consequence.

```
data State = NoState
           | DState Dynamic
           | PairState State State

reset = NoState

split :: Event State → (Event State, Event State)
split NoEvent                 = (NoEvent, NoEvent)
split (Event NoState)         = (Event NoState, Event NoState)
split (Event (PairState l r)) = (Event l, Event r)

join :: State → State → State
join l r = PairState l r

merge :: Event → Event → Event
merge NoEvent e = e
merge e       _ = e
```

Figure 8. The *State* data type and its two accessor functions.

Instantiating Arrow

Next, we show how $SA\ (\rightsquigarrow)$ can instantiate the arrow operators themselves. If it can, then any program written using the arrow operators could just as well be written for the generic arrow $(\rightsquigarrow)$ as for $SA\ (\rightsquigarrow)$. Thus, this instantiation will essentially provide a method to perform a code transformation to allow any arrow to behave as if it could be made settable. In fact, it will not even matter if this instantiation actually obeys the arrow laws; because the arrow it is built atop does, we can always strip off the wrapper and be left with an arrow that does satisfy the laws. The implementations are shown in Figure 9.

The implementations follow directly from the circuit diagrams from Figure 7, and thus we will omit any further description of how they function.

Settable

It feels like we could make an $SA\ (\rightsquigarrow)$ settable merely by removing the *SA* wrapper – after all, the underlying arrow will be of the appropriate type. However, this approach limits modularity by forcing the input and output arrows of the *settable* function to be different. Therefore, we instead write a *settable* function for *SA* directly:

```
settable (SA f) = SA $ proc ((b, e_s), e'_s) → do
  (c, s) ← f ≺ (b, merge e_s e'_s)
  returnA ≺ ((c, s), s)
```

This *settable* function is straightforward with one exception. If there is already a state-update event that is propagating a new state (shown here as e'_s), and the settable signal function is also given a state-update event (e_s), which one takes precedence? In fact, the new one must take precedence in order to guarantee the laws we set out in Figure 5.

7. Optimizations

Providing such an expressive, first-order alternative to the higher-order switch function is a boon for optimizations as it allows the arrow structure to be fully determinable at compile time. For instance, Causal Commutative Arrows (CCAs) are a particular subclass of arrows that have been shown to be highly optimizable [14], but they are restricted to be only first-order. As a demonstration of the optimization capabilities of our work, we extend the Haskell CCA transformation to include non-interfering choice and show the promising results. We begin with a brief overview of CCAs.

```
arr f = SA $ arr (λ (b, _) → (f b, NoState))

first (SA f) = SA $ proc ((b, d), e_s) → do
  (c, s) ← f ≺ (b, e_s)
  returnA ≺ ((c, d), s)

(SA f) ⋙ (SA g) = SA $ proc (b, e_s) → do
  let (e_l, e_r) = split e_s
  (c, s_l) ← f ≺ (b, e_l)
  (d, s_r) ← g ≺ (c, e_r)
  returnA ≺ (d, join s_l s_r)

loop (SA f) = SA $ proc (b, e_s) → do
  rec ((c, d), s) ← f ≺ ((b, d), e_s)
  returnA ≺ (c, s)

delay i = SA $ proc (s_new, e_s) → do
  s_old ← delay i ≺ s_new
  returnA ≺ (f s_old e_s, DState (toDyn s_new))
 where f s NoEvent = s
       f _ (Event NoState) = i
       f _ (Event (DState d)) = fromDyn d

left ~(SA f) = SA $ proc (bd, e_s) → do
  rec (s_old, e_old) ← delay (NoState, NoEvent) ≺ (s_now, e_next)
      let e_now = merge e_s e_old
      (s_now, e_next, cd) ← case bd of
         Left b → do
              (c, s) ← f ≺ (b, e_now)
              returnA ≺ (s, NoEvent, Left c)
         Right d → returnA ≺ (s_old, e_now, Right d)
  returnA ≺ (cd, s_now)
```

Figure 9. *SA* implementations of the Arrow class functions.

7.1 Causal Commutative Arrows

Causal Commutative Arrows are arrows that have two additional laws: a commutativity law that essentially states that signal function effects can be reordered at will, and a product law that governs the behavior of the causal operator (the *init* or *delay* operator). With these two laws at their disposal, Liu et al. [14] describe a transformation that allows an arrow to be reduced to a normal form, which they call the Causal Commutative Normal Form (CCNF), and then even stream fused into a standard function. The authors demonstrate that GHC can then aggressively optimize this, yielding performance increases of orders of magnitude.

The CCA transformation is of particular interest to us as it is what we will be extending to add support for non-interfering choice, but first, we must describe the CCNF. The CCNF of an arrow is either of the form:

$$arr\ f$$

or

$$loop\ (arr\ f \ggg second\ (delay\ i))$$

where f is a pure function and i is a state. We can express these more simply by calling them *Arr f* and *LoopD i f*. The transformation, then, is the process of reducing an arrow built with the arrow operators into one of these two forms. It is a recursive transformation that applies a set of reduction rules until the normal form is produced.

For instance, if the transformation comes across an arrow of the form *first sf*, then it will recursively reduce *sf* and then choose one of the following two rules based on the result:

$$\begin{aligned} \mathit{first}\ (\mathit{Arr}\ f) &\mapsto \mathit{Arr}\ (f \times \mathit{id}) \\ \mathit{first}\ (\mathit{LoopD}\ i\ f) &\mapsto \mathit{LoopD}\ i\ (\mathit{juggle}\ .\ (f \times \mathit{id})\ .\ \mathit{juggle}) \end{aligned}$$

where *juggle* is a pure helper function to reorder the inputs and outputs as necessary.

7.2 Extending CCA

CCAs already have a mechanism for dealing with choice, and at first glance, it appears to work with non-interfering choice too. However, it is the arrowized recursion that non-interfering choice allows, and not the choice operator directly, that actually poses a problem for the CCA transformation.

As is, the CCA transformation does not support recursion. Of course, as we mentioned when we introduced arrowized recursion in Section 3.5, arrows themselves are not guaranteed to support it, so its absense is perfectly sensible as it serves no purpose without non-interfering choice. However, the absense of recursion support is *not* due to inability – indeed, we can add that functionality in a straightforward manner.

Intuitively, the presense of arrowized recursion will present us with the following two scenarios:

$$\mathit{Arr}\ f = \mathit{Arr}\ (g\ f)$$

$$\mathit{LoopD}\ i\ f = \mathit{LoopD}\ (j\ i)\ (g\ f)$$

In the first case, we find that a signal function of the form $\mathit{Arr}\ f$ is defined based on that same function f, and the second is the same except for both f and its state i. However, because f and g (and j) are pure functions, this is a trivial relation to solve: indeed the solution to the first form is as simple as applying a fix point operator:

$$f = \mathit{fix}\ g$$

The second form is slightly more complicated as it requires the use of a coinductive data type for i. We would need a data type such as:

$$\textbf{data}\ \mathit{StateCCA}\ k = S\ (k\ (\mathit{StateCCA}\ k))$$

and with it, we can solve for i and f:

$$\begin{aligned} i &= s \quad \textbf{where}\ s = S\ (j\ s) \\ f &= \mathit{fix}\ g \end{aligned}$$

7.3 Haskell Implementation

We model the Haskell implementation off of the original CCA transformation design. We use Template Haskell along with a clever use of the Arrow type classes to perform a preprocessing step on only the arrowized components. Thus, rather than try to interfere with Haskell's native recursion support, we introduce a new type class to capture it only where we need it:

$$\begin{aligned} &\textbf{class}\ \mathit{ArrowFix}\ (\rightsquigarrow)\ \textbf{where} \\ &\quad \mathit{afix} :: (b \rightsquigarrow c \to b \rightsquigarrow c) \to b \rightsquigarrow c \end{aligned}$$

The *ArrowFix* type class introduces the *afix* function that acts as a fix point function particularly for arrowized recursion. In practice, we could merely define *afix* to be equivalent to the regular fix point operator, but we will make better use of it for the transformation.

Specifically, when the recursive transformation encounters an arrow of the form $\mathit{afix}\ f$, the first thing it will do is to produce a fresh, unique "hole". The hole (which we represent with $\bullet$) is a special internal data structure that acts like *Arr* or *LoopD* except that instead of holding the function f and state i, it keeps track of the modifying functions g and j. That is, if the hole is an *Arr* form, then we know that we will eventually come to a scenario such as

$$\mathit{Arr}\ f = \mathit{Arr}\ (g\ f)$$

Name	GHC	arrowp	CCNF	Stream
Dynamic Counters	1.0	1.66	10.91	12.73
Chained Adder	1.0	1.91	4.06	4.29
Chained Integral	1.0	2.17	13.27	15.40

Figure 10. Performance Ratio (higher is better)

and since f is unknown and will be deduced via the fix point operation, the hole instead keeps track of g. Applying this hole as the argument to f and then recursively running the transformation will reduce the result to one of the two forms we identified in the previous subsection, which we have already shown can be solved easily.

To facilitate this, we create a second set of transformation rules that are nearly identical to the original except that they expect an additional argument. For instance, if the transformation comes across a partial application of a hole, then it will follow one of the following two rules:

$$\begin{aligned} \mathit{first}\ (\bullet_{\mathit{Arr}}\ g) &\mapsto \bullet_{\mathit{Arr}}\ (\lambda\ f \to (g\ f \times \mathit{id})) \\ \mathit{first}\ (\bullet_{\mathit{LoopD}}\ j\ g) &\mapsto \bullet_{\mathit{LoopD}}\ j\ (\lambda\ f \to \\ &\qquad (\mathit{juggle}\ .\ (g\ f \times \mathit{id})\ .\ \mathit{juggle})) \end{aligned}$$

Note the similarities between this and the description for the non-hole version at the end of Section 7.1. They are almost identical except for the fact that the hole's arguments are functions of functions.

Coinductive State

We mentioned in the previous subsection that taking a fix point of the state i would require a coinductive data type, but we make no mention of that. Indeed, because Haskell makes it difficult to dynamically create new types and adding that behavior to the existing CCA Template Haskell transformation would involve completely rewriting it, we instead utilize Haskell's *Dynamic* data type as an all-purpose coinductive wrapper. Although not the most elegant solution, it gets the job done.

7.4 Performance Results

We followed the same procedure for performance testing that Liu et al. [14] use. That is, for each program, we:

1. Compiled with GHC, which has a built-in translator for arrow syntax.
2. Translated the arrow syntax to arrow combinators using Paterson's *arrowp* pre-processor [17] and then compiled with GHC.
3. Normalized into CCNF combinators and compiled with GHC.
4. Normalized into CCNF combinators, rewrote in terms of streams, and compiled with GHC using stream fusion.

The three benchmark programs we used are based on the examples from this paper but are simplified. The first uses the *runDynamic* function to run multiple stateful counters at the same time. The second and third use a function similar to *runDynamic* that runs a signal function multiple times but chains the output from one run to the input of the next, essentially linking them together. For the second, we link together a basic, stateless adder, and for the third, we link an integral function.

The programs were compiled and run on an Intel Core i7 machine with GHC version 7.6.3, using the −O2 optimization. The results are shown in Figure 10, where the numbers represent normalized speedup ratios.

In general, the results show a similarly dramatic performance improvement compared with standard CCA. Notably, the performance of the chained adder, although improved in CCNF, does not show nearly the speedup that the others show. We believe this is because the chained adder has no internal state whatsoever, making the pre-processed performance better.

8. Other effects of switching from switch

As stated earlier, arrows with switch are fundamentally more powerful than those without. Thus, it was never our goal to demonstrate that non-interfering choice and state settability could provide the tools to replace switch outright, but rather that switch's power is often underutilized, and in those cases, switch can be replaced.

8.1 First order

The primary and most important difference between switch and non-interfering choice is that switch is truly higher order while choice is not. This means that while programs with switch can accept streams of signal functions and then run those signal functions, programs with only choice cannot.

8.2 Memory Use

One of the main reasons to use switch in a program is to improve performance. Rather than run a signal function when its results are not being used, we can switch it off, reducing unneeded computation. Signal functions that have been switched out will never be restarted and so can be garbage collected to free memory.

With non-interfering choice, we can similarly stop a signal function, but because it might be restarted, it cannot be garbage collected. Rather, once started, it will remain in memory forever. This is a fundamental reason for demonstrating state settability of signal functions: a signal function that is waiting in memory can have its state re-set so that it can behave as a fresh instance of itself. Thus, with proper management of state, we should never be creating new signal functions while others are left for dead but stranded in memory. Therefore, though our system will always use at least as much memory as a version with switch and often times more, it should be capped by the maximum amount of memory that a comparable switch-based version would use at any one time.

9. Related Work

The idea of using continuous modeling for dynamic, reactive behavior (now usually referred to as "functional reactive programming," or FRP) is due to Elliott, beginning with early work on TBAG, a C++ based model for animation [7]. Subsequent work on Fran ("functional reactive animation") embedded the ideas in Haskell [6, 9], and other embeddings were explored in [5].

After the Arrow framework was proposed by Hughes [12], it was quickly adopted for use in FRP in the GUI language Fruit [1, 2], which also introduced the first arrowized switch function (before then, higher-order signals were dealt with by a function typically called *until*). The design of Yampa [3, 11] built off of this and expanded the idea of switching into fourteen distinct switch operators.

As mentioned, Liu et al. [14] proposed causal commutative arrows, which provide an optimization strategy for first-order arrowized FRP, but which cannot handle higher-order switching. Additionally, some work has minimally explored a restricted form of switch [19], although there is no evidence that this provides any actionable benefits. Patai [16] presents an alternative approach of embracing the higher-order mentality and shows a method for dealing with higher-order streams directly and efficiently using a monadic interface.

Other attempts at optimizing FRP (such as Reactive [8] and Elm [4]) have focused on avoiding recomputation of values when unnecessary. Reactive additionally uses deterministic concurrency for even better performance.

Acknowledgments

This research was supported by a gift from Microsoft Research and a grant from the National Science Foundation (CCF-1302327). We would also like to thank our referees for their comments and suggestions.

References

[1] A. Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, May 2004.

[2] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

[3] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop*, Haskell '03, pages 7–18. ACM, August 2003.

[4] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, 2013.

[5] C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273. ACM, June 1997.

[7] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *21st Conference on Computer Graphics and Interactive Techniques*, pages 421–434. ACM, July 1994.

[8] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, September 2009. ACM.

[9] P. Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, 2000.

[10] P. Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.6), January 2014.

[11] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, August 2003.

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

[13] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, January 2010.

[14] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *Journal of Functional Programming*, 21(4–5):467–496, September 2011.

[15] P. Liu and P. Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193(1):29–45, November 2007.

[16] G. Patai. Efficient and compositional higher-order streams. In *Proceedings of the 19th International Conference on Functional and Constraint Logic Programming*, WFLP'10, pages 137–154, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] R. Paterson. A new notation for arrows. In *Sixth International Conference on Functional Programming*, pages 229–240. ACM, September 2001.

[18] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2011.

[19] D. Winograd-Cort and P. Hudak. Wormholes: Introducing Effects to FRP. In *Haskell Symposium*, pages 91–103. ACM, September 2012.

A. Proof That Non-Interference Implies Commutativity (and Exchange)

Theorem (Commutativity)**.**

$$\mathit{right}\ f \ggg \mathit{left}\ g = \mathit{left}\ g \ggg \mathit{right}\ f$$

Proof. This proof is straightforward. We will begin by assuming *Right* inputs only, and thus we can modify our assertion to:

$$\mathit{arr\ Right} \ggg \mathit{right}\ f \ggg \mathit{left}\ g = \mathit{arr\ Right} \ggg \mathit{left}\ g \ggg \mathit{right}\ f$$

Starting with the left hand side,

$$\begin{aligned}
& \mathit{arr\ Right} \ggg \mathit{right}\ f \ggg \mathit{left}\ g \\
= \ & \{ \text{ Unit backwards } \} \\
& f \ggg \mathit{arr\ Right} \ggg \mathit{left}\ g \\
= \ & \{ \text{ Non-Interference } \} \\
& f \ggg \mathit{arr\ Right} \\
= \ & \{ \text{ Unit } \} \\
& \mathit{arr\ Right} \ggg \mathit{right}\ f \\
= \ & \{ \text{ Non-Interference backwards } \} \\
& \mathit{arr\ Right} \ggg \mathit{left}\ g \ggg \mathit{right}\ f
\end{aligned}$$

For *Left* input values, the proof works in exactly the same way except that we must use non-interference's mirror:

$$\mathit{arr\ Left} \ggg \mathit{right}\ f = \mathit{arr\ Left}$$

which follows directly from non-interference and the definition of *right*. □

B. Choice-Based Implementations of First-Order Switch

Although using non-interfering choice and settability allows for a different paradigm for designing FRP programs, we can also use these tools to implement operators that are similar to the classic switchers. We show two such implementations in this appendix.

B.1 Standard Switch

The standard *switch* function can be implemented with non-interfering choice in a straightforward manner:

```
switch_choice :: (α ⇝ (β, Event γ)) → ((Event γ, α) ⇝ β)
        → (α ⇝ β)
switch_choice sf1 sf2 = proc a → do
  onOne ← delay True ⤙ not onTwo
  (b, et) ← if onOne then sf1 ⤙ a
                     else returnA ⤙ (undefined, NoEvent)
  let onTwo = (isEvent et) || (not onOne)
  if onTwo then sf2 ⤙ (et, a)
           else returnA ⤙ b
```

Here, we keep track of two internal state variables called *onOne* and *onTwo* that indicate whether we should be running the first or the second signal function. When the first produces an event, we set *onOne* to *False* so that we stop running it, and we set *onTwo* to *True*. Then, we pass the impulse generated from the first signal function to the second one, and for the future, the impulse stream contains only *NoEvent* values.

```
pChoice :: Eq key ⇒ [(key, Event α ⇝ β)] →
        ([(key, (UID, Event α))] ⇝ [β])
pChoice [ ] = constA [ ]
pChoice ((key, sf) : rst) = proc es → do
  rec states ← delay [ ] ⤙ states_new
      let es_this = map snd $ filter ((== key) . fst) es
          states_inp = update states es_this
      output ← runDynamic (first (settable sf)) ⤙ states_inp
      let states_new = map (λ ((_, s), uid) →
                          ((NoEvent, Event s), uid)) output
  rs ← pChoice rst ⤙ es
  returnA ⤙ (map (fst . fst) output) ++ rs

 where update :: [((Event α, Event State), UID)]
              → [(UID, Event α)]
              → [((Event α, Event State), UID)]
       update s [ ] = s
       update s ((uid, NoEvent) : rst) =
            update (filter ((≠ uid) . snd) s) rst
       update s ((uid, i) : rst) =
            update (((i, Event reset), uid) : s) rst
```

Figure 11. The implementation of *pChoice*.

B.2 Parallel Switch

The *pChoice* function is somewhat more complicated and is shown in Figure 11. *pChoice* takes a mapping of keys to signal functions (implemented here as a list for simplicity) as its static argument. For each element of this static list, we keep a dynamic list of states (the *states* variable in the figure). We check the input events for any that are keyed to the signal function we are currently processing and update the state list accordingly (by either adding or removing elements), and then we run the signal function for each state and recur. Note that the static signal functions are all impulse driven; thus, when new states are first added to the state list (which is done in the *update* helper function), they are given an impulse event, but otherwise, they are given *NoEvent* (i.e. in the definition of states_{new}). This restriction to strictly impulse driven signal functions is not fundamental – indeed, we could write a version of *pChoice* that accepts signal functions that also take a streaming input – but making it more generic would needlessly complicate this already dense definition.

It is also worth noting that there is a subtle difference in performance between *pChoice* and *pSwitch*. When the finite data type is large but rarely used, *pSwitch* may outperform *pChoice* because *pChoice* still has to iterate through its entire static list on each step while *pSwitch*'s dynamic list will be just the relevant signal functions. That said, their performance should be comparable when the finite data type is small compared to the number of currently running signal functions.