Chapter 22

Signals, Not Generators!

Wolfgang Jeltsch[1]
Category: Research

Abstract: Functional Reactive Programming (FRP) uses signals to describe temporal behavior. Push-based FRP implementations avoid recomputation of signal values in certain cases by taking data dependencies into account. However, they typically do not provide signals directly. Instead, signals are produced by signal generators. Using the same generator multiple times leads to repeated computation and dependence on generation time. This reduces scalability and complicates semantics. This paper presents a push-based implementation approach which does not have these problems. Its keys to success are lazy evaluation, rank-2 polymorphism, and impredicative polymorphism. Our work results in a scalable FRP system which gives the user direct access to the key concept of FRP: the signal.

22.1 INTRODUCTION

22.1.1 The Problem

Functional Reactive Programming (FRP) is based on discrete and continuous signals. A discrete signal is a sequence of occurrences, each consisting of a time and an associated value. A continuous signal is a time-varying value. Dealing with continuous signals in full generality is out of the scope of this paper. We restrict our discussion to continuous signals whose values only change at discrete times. We call them segmented signals.

As an example, let us use signals to describe network traffic. We can see network traffic as a sequence of packets, each being sent or received at a specific time. So we have an example of a discrete signal. We can use this signal to calculate for each time the amount of data transmitted so far. This mapping from times to data volumes can be described by a segmented signal.

[1]Brandenburgische Technische Universität Cottbus, Lehrstuhl Programmiersprachen und Compilerbau, Konrad-Wachsmann-Allee 1, 03046 Cottbus, Germany; `jeltsch@informatik.tu-cottbus.de`

Now, say we want to create a GUI label that always shows the current amount of transmitted data. We use the segmented signal of data volumes to specify the label texts over time. The label should only be updated when a packet is sent or received. Push-based implementations of FRP provide such demand-driven behavior.

In a typical push-based implementation, the label stores the current data volume internally. It initializes it with zero when it starts to process the signal of data volumes. Each time a packet is sent or received, the label is notified. It reacts by updating its stored data volume and its visual appearance. All widgets that visualize time-varying values work in this way. This causes the following problems:

- If several components consume the same signal, each of them calculates the values of this signal itself. So the same calculations are done multiple times.

- If a component switches between different signals, computation of signal values starts afresh at every switch. Particularly, accumulating computations start (again) with their initial value. Say we want to visualize only incoming or outgoing traffic depending on some user selection. Each time the selection changes, the signal value is reset to zero, and accumulation starts anew.

The second point is especially problematic since it means that the values of a signal may depend on the time the signal started to be used. This means that the same signal can have different values at the same time.

For both of the above problems there is a single explanation: What we called signals so far are actually signal generators. Components consume signal generators, and signals switch to signal generators. Whenever a generator is consumed or switched to, it generates a new signal. The same generator might produce different signals at different times.

22.1.2 Contents of This Paper

In this paper, we show how a push-based FRP implementation can provide first-class signals instead of signal generators. In Sect. 22.2, we introduce a simple push-based implementation and show why it suffers from the abovementioned problems. Afterwards, we make the following contributions:

- In Sect. 22.3, we introduce the concept of a vista. A vista is a tree that contains potential future signal values. The actual future values correspond to a path through the vista. We use lazyness to restrict evaluation to such paths. We implement signals based on vistas, which gives us memoization of signal values for free. This eliminates the problem of duplicated computation. We compare the performance of the vista-based solution with the performance of a conventional implementation.

- In Sect. 22.4, we show how to encode signal lifetimes as phantom type parameters. We use this to tie signals to specific start times. Thus, we avoid the problem of start time dependence. Our approach is similar to the technique that makes Haskell's *ST* monad safe.

- In Sect. 22.5, we introduce a switching combinator that works with signal functions. This combinator trims argument signals so that their start times match switching times.[2] We use impredicative polymorphism to ensure that only properly trimmed signals can be used for forming a signal that is switched to.

Finally we discuss related work, give a conclusion, and suggest topics for further work.

All code in this paper is written in Haskell. Our ideas have been implemented in the Grapefruit library [4].

22.2 A TRADITIONAL PUSH-BASED IMPLEMENTATION

22.2.1 The Implementation

Let us look at a traditional way to implement FRP in a push-based fashion. The approach we present here was used by Grapefruit in its early stages. It has also much in common with other push-based FRP implementations like, for example, FranTk [9].

A consumer of a discrete signal registers a handler, which gets called at every occurrence. The handler receives the occurring value as an argument. We want a function *consume* that takes a discrete signal and returns a corresponding handler registration action:

```
consume :: DSignal val → Registration val
```

A value of type *Registration val* takes a handler and turns it into an I/O action that registers this handler. Since we want to be able to unregister the handler later, the registration action returns another I/O action that undoes the registration. So we end up with the following definition:

```
type Registration val = (val → IO ()) → IO (IO ())
```

We simply represent each discrete signal by its corresponding registration action, so that the implementation of *consume* becomes trivial:

```
newtype DSignal val = DSignal (Registration val)
consume (DSignal reg) = reg
```

A segmented signal is represented by an initial value and a discrete signal called the update signal. An occurrence in the update signal means that at the time of this occurrence, the segmented signal changes its value to the value of this occurrence. The type declaration is straightforward:

[2]This trimming is often called aging.

```
scan :: accu → (accu → val → accu) → DSignal val → SSignal accu
scan init next (DSignal reg) = SSignal init (DSignal reg′) where
  reg′ hdlr = do
                accuRef ← newIORef init
                reg (λval → do
                            accu ← readIORef accuRef
                            let
                              accu′ = next accu val
                            writeIORef accuRef accu′
                            hdlr accu′)
```

FIGURE 22.1. Traditional implementation of *scan*

data *SSignal val* = *SSignal val* (*DSignal val*)

Based on these type declarations, a wide variety of signal functions can be implemented.[3] As an example, Fig. 22.1 shows the implementation of the function *scan*. A segmented signal *scan init next sig* starts with value *init*. Every time a value *val* occurs in *sig*, the segmented signal changes from its current value *accu* to the value *next accu val*.

We can use *scan* to compute the segmented signal of data volumes that we mentioned in Subsect. 22.1.1. If *packets* denotes the sequence of network packets and *size* is a function which maps packets to their sizes, the signal of data volumes is *scan* 0 (λ*vol packet* → *vol* + *size packet*) *packets*.

22.2.2 Generators Instead of Signals

To make use of the volume signal, we have to consume its update signal. This involves creating a mutable variable that always holds the current signal value, and registering a handler which uses this variable. The calculation of new signal values is entirely done inside this handler. So every consumer calculates the signal values itself using its own mutable variable.

Switching from one signal to another can easily be implemented by unregistering any handlers of the old signal and consuming the new signal afterwards. So if we switch to a signal which is constructed by *scan*, we create a new mutable variable and initialize it with the initial value that is passed as an argument to *scan*. Therefore, the calculation of signal values starts anew with this initial value at the time when the switch occurs.

[3]However, it is impossible to implement discrete signal union as described in Subsect. 22.3.2. The reason is that the above implementation of *DSignal* makes it impossible to detect whether two occurrences in different signals happen at the same time. There is an alternative implementation [5] that solves this problem while still providing signal generators instead of signals. However, we stick to the above implementation for simplicity.

22.3 SIGNAL MEMOIZATION

22.3.1 Utilizing Native Memoization

Virtually every Haskell implementation performs memoization for expressions that are bound to a variable, although this is not required by the standard [8]. It would be natural if binding signal expressions to variables would lead to signal value memoization. Thus, duplicate computation of signal values could be prevented the same way duplicate computation of, say, list elements can be avoided.

Signal values are only memoized if they are part of the internal data structure that represents the signal. This is not the case for the traditional implementation sketched in Subsect. 22.2.1, except for initial values of segmented signals. The problem is that registration actions only provide a way to receive signal values but do not contain the signal values themselves. We will keep the definition of *SSignal* but introduce a completely different implementation of discrete signals.

The data structure behind a discrete signal has to contain all future signal values. However, a data structure is normally determined when it is created, although lazyness can defer its evaluation. To make a data structure dependent on future information, we have to resort to Haskell's "unsafe I/O" functions [6, 7]. The least unsafe of these functions is *unsafeInterleaveIO* of type $IO\ val \to IO\ val$. For any I/O action *io*, *unsafeInterleaveIO io* does nothing but return a thunk. Evaluation of this thunk triggers the execution of *io*. The output of *io* is evaluated, and the result is taken as the result of the thunk evaluation.

We do not use *unsafeInterleaveIO* directly. Instead, we use lazy channel reading, which is implemented using *unsafeInterleaveIO*. Haskell's channels [7] are FIFO queues that were introduced for communication between concurrent threads.[4] The function *getChanContents* can be used to get a lazy list of all elements in the channel, including those that have not yet been put into it. Values are taken out of the channel as the evaluation of this list progresses.

22.3.2 Using Occurrence Lists

We will now look at a straightforward approach for memoizing signal values. In this approach, each discrete signal contains its list of occurrences as part of its internal representation. For each signal that directly mirrors a sequence of external events, we create a channel of occurrences. Whenever a respective event occurs, its time and associated data are combined to form an occurrence. This occurrence is then put into the channel. The list of occurrences is produced by lazily reading from the channel.

Sadly, the use of occurrence lists makes it difficult to calculate discrete signal unions. The union of two signals sig_1 and sig_2 generally contains all occurrences of sig_1 and sig_2. However, if sig_1 and sig_2 both have an occurrence at the same

[4]Note that our signal implementation does not rely on concurrency. We just need an implementation of FIFO queues that supports lazy reading.

time, only the one from sig_1 is included in the union.[5] A function *union* that computes signal unions has to merge the occurrence lists of its argument signals. In order to do so, it may have to analyze times that have not yet been reached.

Say we want to merge two non-empty occurrence lists $occ_1 : occs_1$ and $occ_2 : occs_2$. The first occurrence of the resulting list is either occ_2 or occ_1, depending on whether occ_2 happens before occ_1 or not. The problem is that we have to know the first occurrence of the result when it actually occurs. Say occ_1 and occ_2 happen at different times. Then we have to successfully compare the times of occ_1 and occ_2 when the earlier one of them happens. At this point, the time of the later occurrence is not yet known.

Elliott [3] proposes a solution to this problem that uses *unsafePerformIO* and concurrency in clever but tricky ways. Implementing his idea correctly has turned out to be hard. At the time of writing, his implementation still contains bugs which seem to be difficult to fix. In the next subsection, we show a much simpler way to make signal unions work. Elliott's approach is further discussed in Subsect. 22.6.2.

22.3.3 Using Vistas

So the key problem with occurrence lists is that *union* cannot easily determine the order of occurrences. Instead of solving this problem, we simply work around it. We do not force *union* to yield an exact sequence of occurrences. A result of *union* may denote a whole collection of possible such sequences instead. It is the responsibility of signal consumers to decide which of these sequences is the correct one. To make implementation of this idea possible, the representation of discrete signals must be able to express such uncertainty introduced by *union*.

We base our implementation of discrete signals on discrete sources [5]. A discrete source provides a class of external events. Examples of discrete sources are the source of all key press events and the source of all incoming network packets. We make the following restrictions regarding discrete sources and events:

- For each discrete source, there is a mechanism to get notified about events provided by this source. Usually, this mechanism is registration of event handlers.

- Each event is provided by exactly one discrete source.

- Different events cannot happen at the same time. The order of event handler calls determines the order of events.

Now, say we have two discrete sources *inPacketSrc* and *outPacketSrc* of incoming and outgoing network packets. We want to form two discrete signals *inPackets* and *outPackets* that mirror the event sequences of these sources. We

[5]Another solution would be to combine the values of both occurrences using some user-specified function. The implementation presented in Subsect. 22.3.3 also allows for such an approach.

can use channels again to get lazy lists of incoming and outgoing packets, respectively.

These lists are not enough to describe the *inPackets* and *outPackets* signals since they do not tell us anything about occurrence times. However, we do not need to include times in our signal data structure. It is sufficient to know what discrete sources the signals stem from. A consumer of *inPackets* or *outPackets* can then be notified about events of the respective source. On each event, it can fetch the next packet from the list and react accordingly. So the occurrence times are given implicitly as the times when the discrete source notifies the consumer.

Since the occurrences of *inPackets* and *outPackets* are induced by different sources, they happen at different times. Therefore, no occurrences are dropped when forming the union of both signals. The occurrence sequence of the union is an interleaving of the occurrence sequences of *inPackets* and *outPackets*. The choice of the correct interleaving depends solely on the order in which *inPacketSrc* and *outPacketSrc* fire events. Since we do not know this order in advance, the representation of the union must cover all possible such orders.

Either *inPacketSrc* or *outPacketSrc* will fire first. For both cases, the representation of the union gives the first occurrence value and the remainder of the signal. The representation of a remainder is structured like the representation of the complete signal. So it distinguishes two cases according to the source that will fire second. It provides an occurrence value and a remainder for each case. And so on.

We call such a data structure a vista and represent discrete signals by vistas:

newtype *DSignal val* = *DSignal* (*Vista val*)
type *Vista val* = *Map DSource* (*Variant val*)
data *Variant val* = *Variant val* (*Vista val*)

Note that we can also represent the primitive signals *inPackets* and *outPackets* using these definitions. We just have to use vistas with only one source.

A vista corresponds to a kind of Mealy machine whose underlying graph is a tree. Figure 22.2 shows the machine of the signal *union inPackets outPackets*. Thereby, a label $\frac{src}{val}$ means that the respective transition is used if *src* fires next, and that there is an occurrence of *val* in this case. Compared to a true Mealy machine, our “vista machines” differ in the following ways:

- The number of states is potentially infinite (albeit the number of outgoing edges per state is finite).
- There is not necessarily a transition for each pair of a state and a discrete source. If a source fires an event, and there is no transition for this source from the current state, the machine simply does nothing.

The above definition of vistas does not allow for filtering of discrete signals. Filtering removes all occurrences from a discrete signal whose values do not have a given property. This is very much like the standard *filter* function dropping elements from a list. We cannot drop occurrences by removing transitions from the

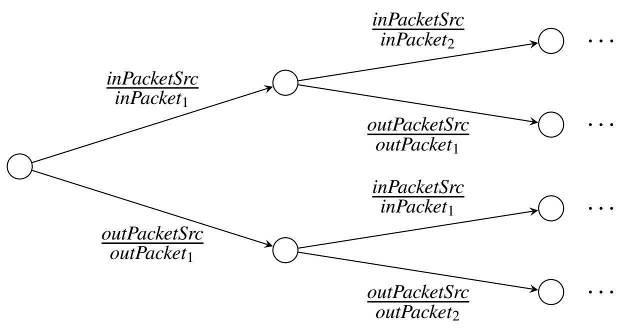
$\frac{\underline{inPacketSrc}}{inPacket_1}$
$\frac{\underline{outPacketSrc}}{outPacket_1}$
$\frac{\underline{inPacketSrc}}{inPacket_2}$
$\frac{\underline{outPacketSrc}}{outPacket_1}$
$\frac{\underline{inPacketSrc}}{inPacket_1}$
$\frac{\underline{outPacketSrc}}{outPacket_2}$
...

FIGURE 22.2. **Vista of** *union inPackets outPackets*

underlying machine. While this would prevent values from being output, it would also inhibit necessary state changes. Therefore, we change the vista concept so that outputs become optional. We only have to modify the definition of *Variant*:

data *Variant val* = *Variant* (*Maybe val*) (*Vista val*)

The definitions of *DSignal* and *Vista* are left unchanged.

22.3.4 Implementation of Signal Combinators

To filter a signal, we just recurse into the vista and replace terms of the form *Just val* by *Nothing* if *val* does not fulfill the given predicate. The *scan* function works in a similar way. We accumulate values during recursion and replace occurrence values accordingly. Note that all these transformations are done lazily.

For implementing the *union* function, we have to consider that its arguments may already contain uncertainty introduced by other applications of *union*. The vista of the union of two signals must be a machine that simulates the machines of the argument signals in parallel. So calculating the union essentially means calculating a kind of product automaton, which is easy because of the tree structure of vistas. Note that such a calculation combines simultaneous transitions. Thereby, values of simultaneous occurrences can be composed in arbitrary ways. Dropping the value from the second signal is just a special case.

22.3.5 Signal Consumption

The actual occurrence sequence of a signal corresponds to a path through its vista. This path is determined by the consumers of the signal. Each consumer only evaluates those parts of the vista that correspond to this path. All variants that describe possibilities that do not actually happen are garbage-collected without having been evaluated.

A consumer of a discrete signal registers an event handler for every source that can trigger a first occurrence. If one of these sources fires an event, the corresponding handler fetches the variant that corresponds to this source. If the variant contains an occurrence value, the handler triggers the corresponding reaction of the consumer. Afterwards, the handler unregisters itself and all other handlers of the consumer. Finally, the whole procedure starts again, this time using the vista that describes the remainder of the signal.

22.3.6 Performance Comparison

We compare the performance of the current version of Grapefruit (which uses vistas) with the performance of the version that directly preceded the introduction of vistas[6]. Performance is especially crucial when events occur very frequently. An example of such a situation is real-time sound synthesis. Here, we generate a signal that represents a sequence of timer ticks. From this signal, we calculate audio signals. Each audio signal assigns a block of multiple consecutive samples to each tick.

For our performance measurements, we use FM synthesis to generate a sound that is faded out afterwards by multiplying it with an exponential function. The resulting signal is processed by a number of consumers. Each consumer iterates through all samples and evaluates those that are still unevaluated. We variegate the number of consumers between 1 and 10.

To ease measurements, we do not produce actual timer ticks. Instead, we generate a tick at the beginning of the simulation and every time the reaction to a previous tick has been finished. That way, the program never becomes idle. Since a sampling rate of 96 kHz and a tick frequency of about 1 kHz are typical for audio processing, we fix the block size to 100 samples. In each simulation run, we process 10 000 blocks.

We perform our measurements on an Intel Pentium M processor with a clock frequency of 600 MHz. We compile the code using version 6.10.4 of the Glasgow Haskell Compiler with optimizations turned on. For each of the two Grapefruit versions and each number of consumers, we run the simulation five times and form the mean of the total CPU times.

The results of our measurements are shown in Fig. 22.3. We can see that the new implementation already pays off in the case of only two consumers. Note that it is not unusual for a signal to be used more than two times. For example, a sound synthesizer might compose a signal with a variation of itself to achieve a phaser effect and feed the resulting signal into four different audio channels. This would result in the original signal being used eight times.

22.4 STATIC ERA CHECKS

We want to ensure that the production time of a signal equals all consumption times of this signal and of all signals derived from it. We employ the type system

[6]The implementation ideas of this pre-vista version have already been described elsewhere [5].

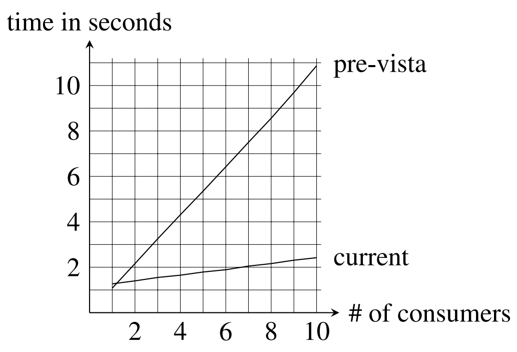

FIGURE 22.3. Performance comparison between current and pre-vista Grapefruit

to make this property statically checkable. Our solution is heavily inspired by the technique that makes Haskell's *ST* monad [6] safe.

We introduce a phantom type parameter *era* for *DSignal* and *SSignal*. This parameter denotes the time interval during which the respective signal is alive. The beginning of this interval is what we are actually interested in since this is the time when the signal is consumed or switched to. Signal functions like *scan* enforce through their types that argument and result signals have the same era. The type of *scan*, for example, becomes

$$accu \to (accu \to val \to accu) \to DSignal\ era\ val \to SSignal\ era\ accu \ .$$

Instead of using the *IO* monad for producing and consuming signals, we use a wrapper around *IO* which has an *era* parameter:

newtype *Reactive era val* = *Reactive* (*IO val*)

Signals that are produced or consumed are forced to have the same era as the producing or consuming action. This is done by using types that use the same type variable for the era of a signal and of an associated reactive action. For example, the *consume* function now has the type

$$DSignal\ era\ val \to (val \to IO\ ()) \to Reactive\ era\ (IO\ ()) \ .$$

To actually run a reactive action, we have to turn it into an I/O action:

toIO :: ($\forall$*era*.*Reactive era val*) $\to$ *IO val*
toIO (*Reactive io*) = *io*

The universal quantification of the era prevents signals from being shared between different *Reactive* actions. This is necessary because different actions may be executed at different times. If a reactive action would consume a signal that is derived from a signal produced by a different reactive action, both actions would be forced to have the same era. However, this would contradict the universal quantification which states that both eras must be independent.

22.5 SWITCHING

22.5.1 Safety through Impredicativity

We introduce a combinator *switch* that constructs signals whose behavior switches between the behavior of other signals. If we would not use eras, *switch* would have the type

$$(\mathit{Signal}\ \mathit{sig}) \Rightarrow \mathit{SSignal}\ (\mathit{sig}\ \mathit{val}) \rightarrow \mathit{sig}\ \mathit{val}\ \ .$$

Here, we assume the existence of a class *Signal* which *DSignal* and *SSignal* are instances of. A signal constructed by *switch* would first behave like the initial value of the argument signal. Every time the argument signal is updated, the result signal would start to behave like the signal the argument is updated to. We call the values of the argument signal inner signals.

It is clear that the argument and result of *switch* have the same era. However, the eras of the inner signals are generally proper subintervals of, and therefore not equal to, this era. Worse, they are not even mutually equal. This conflicts with the fact that all inner signals have to have the same type and therefore have to share a single era parameter.

We can solve this problem by using universal quantification again. If the inner signals are compatible with any era, they are compatible with the era that is actually used. We give *switch* the type

$$(\mathit{Signal}\ \mathit{sig}) \Rightarrow \mathit{SSignal}\ \mathit{era}\ (\forall \mathit{era}'.\mathit{sig}\ \mathit{era}'\ \mathit{val}) \rightarrow \mathit{sig}\ \mathit{era}\ \mathit{val}\ \ .$$

The use of universal quantification plays a similar role as in the type of *toIO* from Sect. 22.4. However, we do not use rank-2 polymorphism here since the universally quantified type is not the domain of a function type but a parameter of a non-function type. What we use here, is impredicative polymorphism [11].

22.5.2 Safe, but Useless

A *switch* function of the above type is safe but essentially useless. The only discrete signal with arbitrary era is the empty signal, a signal without occurrences. The only segmented signals with no era constraints are constant signals.

The reason is that signal combinators do not "invent" occurrences. They only aggregate and drop occurrences and transform occurrence values. So every non-empty discrete signal has to be derived from a signal that mirrors external events. Such a signal is produced by a reactive action. The non-empty signal inherits the era of the reactive action. Therefore, its era is constrained. Segmented signals inherit the era of their update signals. So a segmented signal is only era-independent if its update signal is empty.

22.5.3 Signal Functions to the Rescue

Our solution is to switch between signal functions instead of signals. Thereby, a signal function must use a single era for all its arguments and its result. Because signal functions can have different arities, there is no most general signal function type. Therefore, we cannot directly come up with a type for *switch*. We will show a solution to this problem in Subsect. 22.5.4. For now, we pretend that *switch* has every type of the following structure:

$$\begin{aligned}&(\mathit{Signal}\ \mathit{sig}_1, \ldots, \mathit{Signal}\ \mathit{sig}_n, \mathit{Signal}\ \mathit{sig}') \Rightarrow \\ &\mathit{SSignal}\ \mathit{era}\ (\forall \mathit{era}'.\mathit{sig}_1\ \mathit{era}'\ \mathit{val}_1 \to \ldots \to \mathit{sig}_n\ \mathit{era}'\ \mathit{val}_n \to \mathit{sig}'\ \mathit{era}'\ \mathit{val}') \to \\ &(\mathit{sig}_1\ \mathit{era}\ \mathit{val}_1 \to \ldots \to \mathit{sig}_n\ \mathit{era}\ \mathit{val}_n \to \mathit{sig}'\ \mathit{era}\ \mathit{val}')\end{aligned}$$

To form the result of an expression *switch ctrl* $\mathit{sig}_1 \ldots \mathit{sig}_n$, we split sig_1 to sig_n at the update times of *ctrl*. The eras of the resulting slices correspond to intervals during which *ctrl* remains constant. For each of these eras, we compose all n slices of this era by applying the corresponding value of *ctrl*, which is a signal function. We glue the resulting pieces together to get the final result.

So the result of the above *switch* application can depend on the argument signals sig_1 to sig_n. This is in contrast to the useless approach of Subsect. 22.5.1. On the other hand, impredicative polymorphism still forbids the use of other signals in forming the result. The signal functions of *ctrl* can only access sig_1 to sig_n, and they even cannot access them directly but only the correct slices of them. So era consistency is still guaranteed.

We can now implement the switching example from the introduction correctly. Remember that the goal was to display the amount of either incoming or outgoing network traffic. We construct a signal *ctrl* that changes between *curry fst* and *curry snd*[7] according to selection changes triggered by the user. We define a helper function *volumes* as follows:

$$\mathit{volumes} = \mathit{scan}\ 0\ (\lambda \mathit{vol}\ \mathit{packet} \to \mathit{vol} + \mathit{size}\ \mathit{packet})$$

The signal *switch ctrl* (*volumes inPackets*) (*volumes outPackets*) has the amount of either incoming or outgoing traffic as its value.

Note that we are not restricted to putting the value of either argument signal into the result signal. For example, we could also offer to display the total amount of network traffic. In the *ctrl* signal, we would use a lifting of the (+) function, that is, a signal function that forms the pointwise sum of two signals.

The traditional implementation resets traffic volumes at every switch. Note that we could also get this effect with our new implementation if we wanted to. The trick is to split *inPackets* and *outPackets* and perform accumulation on the resulting slices. This is done by the expression *switch ctrl′ inPackets outPackets* where the signal functions in *ctrl′* first apply *volumes* to their arguments before actually composing them. In contrast, the correct solution accumulates first and then splits the resulting signals.

So we can choose at which "stage" we want to split. In contrast, the traditional implementation always splits the signals that are directly produced by a reactive action. These signals might not even be mentioned directly in the *switch* application but buried deep into the definition of complex signals.

[7] Since *fst* and *snd* select the first and second component from a pair, their curried versions return their first and second argument, respectively.

22.5.4 A Generic Signal Function Type

So far, we have assumed that *switch* can work with signal functions of arbitrary arity. This is not directly possible with Haskell's type system. We overcome this problem by defining a Generalized Algebraic Data Type (GADT) that covers all signal functions, independently of arity:

```
data SignalFun era shape where
  OSF :: (Signal sig) ⇒ sig era val → SignalFun era (sig ‘Of‘ val)
  SSF :: (Signal sig) ⇒
           (sig era val → SignalFun era shape′) →
           SignalFun era (sig ‘Of‘ val ↦ shape′)
data shape ↦ shape′
data (sig :: ∗ → ∗ → ∗) ‘Of‘ (val :: ∗)
```

The *shape* parameter of *SignalFun* is a phantom parameter. It should have the form sig_1 ‘*Of*‘ $val_1 \mapsto \ldots \mapsto sig_n$ ‘*Of*‘ $val_n \mapsto sig'$ ‘*Of*‘ val'. In contrast to ordinary signal function types, shape types leave era parameters out. *SignalFun* adds its *era* parameter consistently to all signal types of the shape. The constructors *OSF* and *SSF* construct nullary and non-nullary functions, respectively.[8]

Using *SignalFun*, the type of *switch* becomes

$$SSignal\ era\ (\forall era'.SignalFun\ era'\ shape) \rightarrow SignalFun\ era\ shape \quad .$$

Compared to the non-solution of Subsect. 22.5.1, we have just replaced the signal type *sig* by *SignalFun* (and removed its type class constraint).

22.5.5 Implementation

The implementation of *switch* has to deal with different signal types (*DSignal* and *SSignal*), different arities of signal functions, and technical challenges arising from the use of universal quantification. Therefore, we do not show the complete implementation here. Instead, we only present the key idea for implementing signal splitting. The full implementation of switching is part of the source code of Grapefruit [4].

To cut a slice out of a signal, we have to remove the part of the signal that lies before the start of the slice, and the part that lies after the end of the slice. Removing the latter one is easy. When the end time of the slice is reached, we just drop the remainder of the signal. Removing the part before the slice is more difficult. This removal is called aging.

The good thing about our switching combinator is that it knows from the beginning which signals it has to age, namely, the arguments of the resulting signal function. So *switch* can age these signals successively as time passes. This is in contrast to implementations like the one by Elliott [3], where the need for aging might be realized only later so that aging has to catch up.

[8] *O* and *S* stand for "zero" and "successor". *SF* means "signal function".

```
remains :: DSignal era val → SSignal era (Vista val)
remains (DSignal vista) = SSignal vista (DSignal (vistaRemains vista))
vistaRemains :: Vista val → Vista (Vista val)
vistaRemains = fmap variantConv where
  variantConv (Variant _ vista) = Variant (Just vista) (vistaRemains vista)
```

FIGURE 22.4. Implementation of *remains*

For every time, we track what the current aged signals are. To do this, we use a helper function *remains*, whose implementation is shown in Fig. 22.4. For a discrete signal *sig*, the expression *remains sig* is a segmented signal. For each time, this signal gives the vista that represents the part of the discrete signal that lies after this time.

22.6 RELATED WORK

22.6.1 Implementations in Impure Languages

In the traditional implementation shown in Subsect. 22.2.1, building a signal generator involves building an I/O action. Running this I/O action means generating a signal. However, execution of this action has to be deferred until the generator is consumed. The reason is that side effects cannot happen during expression evaluation since Haskell is a pure language. So signal generation can only happen during consumption.

The situation is different for impure languages. FRP implementations like FrTime [1] (Scheme) and Frappé [2] (Java) are similar to the traditional implementation. However, they execute the necessary I/O actions immediately. So signal generation happens during expression evaluation, and the result of the expression is already a signal, not a generator.

However, this causes a different problem. As we have seen, the same signal generator can create different signals at different times. In the case of the traditional implementation, this makes signals dependent on consumption time. In the case of FrTime, Frappé, and similar libraries, it makes signals dependent on evaluation time. So the same expression may evaluate to different signals. Therefore, these libraries are not referentially transparent.

22.6.2 Implementations Based on Occurrence Lists

Sage [10] presents an implementation of discrete signals which is based on occurrence lists. He uses imperative programming to overcome the problem with calculating signal unions. His implementation creates a new discrete source for each union. This source raises an event for each occurrence in either union argument. Since creating a source involves I/O, the union operator has to use *unsafePerformIO*. The problem with Sage's solution is that simultaneity of occurrences cannot be detected. Two simultaneous occurrences result in two occurrences in the union signal, and the order of these occurrences is undefined.

Elliott [3] uses a representation of discrete signals which is equivalent to a list of occurrences. His implementation of times permits successful comparison of two times at both of these times. When two future times $time_1$ and $time_2$ need to be compared, two threads are run concurrently. One thread waits for $time_1$ and compares both times then, the other one does the same for $time_2$. The thread that finishes earlier yields the result, and the other thread is killed. So the result of the comparison is available at the earlier one of both times. This solves the problem with union calculation. Unfortunately, Elliott's idea has not been implemented correctly until now. In addition, it is not yet known whether the massive creation of short-lived threads causes unacceptable performance decrease [3, section 14].

In both Sage's and Elliott's approach, it is safe to switch to a signal after it has started. However, it is generally not possible to know in advance that such a switch will occur. If the signal has not been used before the switch, evaluation has to catch up. The only way to avoid this is to age the signal explicitly by using some aging operator. This problem should be solvable with our ideas for static era handling from Sect. 22.4 and Sect. 22.5. Signals that are switched to later would be known in advance so that they could be aged right from the start.

22.7 CONCLUSIONS AND FURTHER WORK

22.7.1 Conclusions

We have shown how a push-based FRP implementation can provide first-class signals instead of signal generators. We have represented signals using a data structure that describes possible future behavior. This enables memoization of signal values. In addition, we have encoded signal lifetimes as phantom type parameters. We have used rank-2 and impredicative polymorphism to ensure that signal consumption cannot be deferred. This makes signal values unambiguous.

22.7.2 Further Work

In this paper, we have not shown how to support arbitrary continuous signals. However, we have already developed a basic solution, which we want to expand further. Its key idea is to represent continuous signals by segmented signals of I/O actions that yield the current signal value when called.

An open problem is how to deal with continuous signals that are specified by systems of recursive equations. So far, we do not support such specifications appearing directly in the source code. However, the solutions of such equation systems can often be expressed as convolutions. We want to explore how support for signal convolution can be implemented and whether such a feature can make recursive signal definitions obsolete.

Our performance measurements only provide a first estimation of the usefulness of the vista approach. More measurements, especially in real-world situations, are necessary to gain more precise information. Another important goal are performance comparisons between Grapefruit and other FRP libraries.

Acknowledgments

I wish to thank Roman Cheplyaka, Petra Hofstedt, Janis Voigtländer, and the anonymous reviewers for their helpful remarks on earlier versions of this paper.

REFERENCES

[1] G. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, page 294–308. Springer, Berlin/Heidelberg, 2006.

[2] A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, page 29–44. Springer, Berlin/Heidelberg, 2001.

[3] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, page 25–36, New York, NY, 2009. ACM.

[4] W. Jeltsch. The Grapefruit homepage. `http://haskell.org/haskellwiki/Grapefruit`.

[5] W. Jeltsch. Improving push-based FRP. In P. Achten, P. Koopman, and M. Morazán, editors, *Draft Proceedings of the 9th Symposium on Trends in Functional Programming (TFP '08)*, number ICIS-R08007, Nijmegen, 2008. Radboud Universiteit Nijmegen.

[6] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[7] S. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, number 180 in NATO Science Series: Computer and Systems Sciences, page 47–96. IOS Press, Amsterdam, 2001.

[8] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, May 2003.

[9] M. Sage. FranTk – a declarative GUI language for Haskell. *ACM SIGPLAN Notices*, 35(9):106–117, Sept. 2000.

[10] M. Sage. *Declarative Support for Prototyping Interactive Systems*. PhD thesis, Department of Computing Science, University of Glasgow, Mar. 2001.

[11] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. *ACM SIGPLAN Notices*, 41(9):251–262, Sept. 2006.