Chapter 14

Improving Push-based FRP

Wolfgang Jeltsch[1]
Category: Research

Abstract: Push-based implementations of Functional Reactive Programming allow for writing reactive programs in a declarative style and execute them efficiently. Previous approaches in this area are not able to combine simultaneously occuring internal events. This may lead to efficiency problems and introduction of inconsistent intermediate states. Most of these implementations also lack declarative means to describe systems with feedbacks. We limit ourselves to the data flow aspects of FRP and present an implementation which does not suffer from these deficiencies. This leads to a foundation for FRP which is more declarative than previous solutions and more efficient at the same time.

14.1 INTRODUCTION

Functional Reactive Programming (FRP) makes it possible to implement reactive systems in a declarative way. It is based on the notions of discrete and continuous signals. A discrete signal is a sequence of discrete points in time with a value attached to each of it. A continuous signal is a time-dependent value. FRP systems provide means to construct new signals from existing ones in purely declarative ways. Especially, they support switching, that is, the creation of signals whose behaviors switch to the behaviors of arbitrary other signals.

There are two principal approaches to implement FRP:

Push-based implementations react to external events by updating their internal state and influencing the environment.

Pull-based implementations regularly poll interesting parts of the environment's state and modify their internal state and the environment accordingly.

---

[1]Brandenburgische Technische Universität Cottbus, Lehrstuhl Programmiersprachen und Compilerbau, Ewald-Haase-Straße 12/13, 03044 Cottbus, Germany; `jeltsch@informatik.tu-cottbus.de`

Typically, push-based implementations recalculate only those parts of the internal state and the environment which need updating. This is an advantage over pull-based approaches which always recalculate everything.

However, previous push-based systems fell short of providing the declarativeness and flexibility pull-based systems offer. A common problem is *missing treatment of simultaneity*. When discrete signals are combined, values belonging to the same time are not identified as simultaneously occuring. This can result in multiple reactions of the system instead of one large reaction which can reduce performance and introduce inconsistent intermediate states. In addition, most push-based FRP systems do not allow *declarative descriptions of feedbacks*. Feedbacks are dependencies of a subsystem's input on its own output.

In this paper we attack these two problems. We discuss an FRP foundation which is essentially a data flow system. This means that, in contrast to a full FRP system, it lacks support for continous signals and switching. We introduce our system by example and give a discrete signal semantics in section 14.2. Afterwards we present a push-based implementation of our approach. We make thus the following contributions:

- In section 14.3, we implement a framework for describing systems of reactive components communicating via signals. Our implementation accumulates parts of a system's initialization action for later execution. This allows declarative descriptions of feedbacks under certain conditions which we identify.

- In section 14.4, we present an implementation of discrete signals that includes a merge operation which combines simultaneously occuring values.

- After showing a way to memoize the values of a discrete signal in section 14.5, we show that this approach destroys the prerequisites for feedbacks. In section 14.6, we introduce lax arrows. Lax arrows provide a way to reorder parts of an arrow value in order to remove divergence for certain arrow feedbacks. We use lax arrows to make feedbacks possible in the presence of memoization.

Finally, we discuss related work in section 14.7, and give a conclusion and an outlook on further work in section 14.8 as well as acknowledgments in section 14.9.

All code presented in this paper is written in Haskell. Our concepts have been implemented in the Grapefruit library [9] and the *lax* package [10].

14.2   A SURVEY OF OUR DATA FLOW SYSTEM

Discrete signals can describe sequences of events, each of them occuring at a specific time and being parametrized by a certain value (for example, a key code in the case of key press events). They can also be composed to form new signals. A discrete signal is a value of type *DSignal val* for some type *val*. The internal structure of discrete signals is designed such that it aids a push-based execution of reactive systems. It cannot be accessed directly by the application programmer. However, we define the *meaning* of a signal to be a Haskell value which directly reflects the intuition behind the signal.

We define times to be non-negative real numbers whereas zero is the time where the reactive system starts. The type *Time* covers all strictly positive reals.[2] If *dSignal* :: *DSignal val* for some type *val* then $\llbracket dSignal \rrbracket :: [(Time, val)]$ is the meaning of *dSignal*. The elements of the meaning are called occurences. We say that the value *val* occurs at time *time* in the signal *dSignal* if $(time, val) \in \llbracket dSignal \rrbracket$. The fact that 0 is not a value of *Time* makes it impossible for a value to occur at the start of the reactive system.

The meaning of each signal *dSignal* fulfilles the condition that for every *time* :: *Time*, the expression

$$all \ (>time) \ (tail \ (dropWhile \ (<time) \ (map \, fst \ \llbracket dSignal \rrbracket)))$$

is equivalent to *True*. Note that this restriction implies the following:

- The times of the occurences are strictly ascending.

- For every time *time* there are only finitely many occurences with a time less than *time*.

- $\perp$ can only occur as part of the value components.

The second and third point hold because otherwise the above expression would be equivalent to $\perp$ instead of *True*.

Interaction with the environment is described by *circuits*. A circuit has an input and an output, both of which are typically tuples of discrete signals.[3] A circuit may produce occurences in its output and change the outside world in response to occurences in its input and external events. A circuit with input type *i* and output type *o* is a value of type *Circuit i o*. *Circuit* is an instance of *Arrow* and *ArrowLoop* [11] which allows complex circuits to be composed out of simpler ones. Arrow syntax [11] is typically used for this.

In the following, we show signals and circuits in action using a simple GUI application. During our walk through this example, we introduce several operations on signals and state their semantics. Our example application displays the current time in the format *HH*:*MM*. Beside the displayed time are two buttons for modifying the time, an hour incrementing button to the left and a minute incrementing button to the right. The source code of the application is shown in figure 14.1.

clockApp is a circuit based on the predefined circuit constructing functions *minutePulse*, *button* and *label* whose type signatures are given in figure 14.2. When *clockApp* is run, an *instance* of this circuit is created. This consists of instances of its subcircuits. A *minutePulse* instance outputs a signal in which the value () occurs every minute. An instance of *button initCap* denotes a button which will show up as part of a GUI. This button initially has the caption *initCap*. Everytime a value *newCap* occurs in the input, the button's caption changes to *newCap*. The value () occurs in the output whenever the user presses the button.

---

[2]We disregard the fact that the type of real numbers is actually not implementable.

[3]Hereby we consider () to be the 0-tuple and identify discrete signals with their corresponding 1-tuples.

```
clockApp :: Circuit () ()
clockApp = proc () → do
  rec let
        hPulse   = filter (λm → m 'mod' 60 ≡ 0) (count mPulse)
        update   = merge (count (hPulse 'merge' hPress))
                         (count (mPulse 'merge' mPress))
        timeUpd = scan nextTime (0,0) update
      mPulse ← minutePulse   ⊣ ()
      hPress ← button "H"     ⊣ empty
      ()       ← label "00:00" ⊣ fmap timeStr timeUpd
      mPress ← button "M"     ⊣ empty
  returnA ⊣ ()

count :: DSignal val → DSignal Int
count = scan (λnum _ → succ num) 0

nextTime :: (Int, Int) → MergeVal Int Int → (Int, Int)
nextTime (_, oldM) (First newH)        = (newH, oldM)
nextTime (oldH, _) (Second newM)       = (oldH, newM)
nextTime (_, _)    (Both newH newM) = (newH, newM)

timeStr :: (Int, Int) → String
timeStr (h, m) = twoDigits (h 'mod' 24) ++ ":" ++ twoDigits (m 'mod' 60)

twoDigits :: Int → String
twoDigits n = reverse (take 2 (reverse ('0' : show n)))
```

FIGURE 14.1.   Source code of the clock application

An instance of *label initText* denotes a text label. Setting the label's text is handled analoguously to setting a button's caption. The label in our example is used to display the time.

The order of button and label circuits in the application circuit determines the order of the buttons and labels on the screen. Since the time shown by the label depends on the minute incrementing button, we have to use feedback as provided by *ArrowLoop*. This is done using **rec**. As a side effect, we are able to put all local variable definitions into a single **let** block at the beginning.

The example code uses several operations on discrete signals whose type declarations and semantics are shown in figure 14.3. Because it is impossible in

```
minutePulse ::               Circuit ()               (DSignal ())
button      :: String → Circuit (DSignal String) (DSignal ())
label       :: String → Circuit (DSignal String) ()
```

FIGURE 14.2.   Type signatures of circuit constructing functions

$empty :: DSignal\ val$

$[\![empty]\!]_{time} = [\,]$

$filter :: (val \rightarrow Bool) \rightarrow DSignal\ val \rightarrow DSignal\ val$

$[\![filter\ prd\ dSignal]\!]_{time} = Prelude.filter\ (prd \circ snd)\ [\![dSignal]\!]_{time}$

$fmap :: (val \rightarrow val') \rightarrow DSignal\ val \rightarrow DSignal\ val'$

$[\![fmap\ fun\ dSignal]\!]_{time} = onVals\ (fmap\ fun)\ [\![dSignal]\!]_{time}$

$scan :: (val' \rightarrow val \rightarrow val') \rightarrow val' \rightarrow DSignal\ val \rightarrow DSignal\ val'$

$[\![scan\ acc\ init\ dSignal]\!]_{time} = onVals\ (tail \circ scanl\ acc\ init)\ [\![dSignal]\!]_{time}$

$onVals :: ([val] \rightarrow [val']) \rightarrow [(Time, val)] \rightarrow [(Time, val')]$

$onVals\ valsFun\ occs = zip\ (map\ fst\ occs)\ (valsFun\ (map\ snd\ occs))$

data $MergeVal\ val_1\ val_2 = First\ val_1 \mid Second\ val_2 \mid Both\ val_1\ val_2$

$merge :: DSignal\ val_1 \rightarrow DSignal\ val_2 \rightarrow DSignal\ (MergeVal\ val_1\ val_2)$

$[\![merge\ dSignal_1\ dSignal_2]\!]_{time} = occsMerge\ [\![dSignal_1]\!]_{time}\ [\![dSignal_2]\!]_{time}$

$occsMerge :: [(Time, val_1)] \rightarrow [(Time, val_2)] \rightarrow [(Time, MergeVal\ val_1\ val_2)]$

$occsMerge\ [\,]\qquad occs_2 = occs_2$

$occsMerge\ occs_1\ [\,]\qquad = occs_1$

$occsMerge\ occs_1\ occs_2 = $ **let**

$$(headTime_1, headVal_1) = head\ occs_1$$
$$(headTime_2, headVal_2) = head\ occs_2$$

in case $compare\ headTime_1\ headTime_2$ **of**

$LT \rightarrow (headTime_1, First\ headVal_1):$
$\qquad occsMerge\ (tail\ occs_1)\ occs_2$
$EQ \rightarrow (headTime_1, Both\ headVal_1\ headVal_2):$
$\qquad occsMerge\ (tail\ occs_1)\ (tail\ occs_2)$
$GT \rightarrow (headTime_2, Second\ headVal_2):$
$\qquad occsMerge\ occs_1\ (tail\ occs_2)$

FIGURE 14.3. Type signatures and semantics of basic discrete signal functions

general to state the meaning of a *filter* result as a Haskell expression, we define meanings indirectly through *bound meanings*. For any $dSignal :: DSignal$ and $time :: Time$, the bound meaning $[\![dSignal]\!]_{time}$ is equivalent to *takeWhile* $((\leqslant time) \circ fst)\ [\![dSignal]\!]$.

empty denotes a signal without any occurences. It is used to state that the button captions never change. *fmap* applies a function to all occuring values. The helper function *count* counts the occurences in a given signal. It uses the *scan* function which accumulates occurence values and has similarities with the *scanl* function from the prelude. *scan* is also used to generate a signal of times from the *update* signal which describes time changes. *filter*[4] drops occurences whose

---

[4]This function is different from the prelude function of the same name and can be distinguished from the prelude function because it belongs to a different module.

values do not fulfill a given predicate. This is used in the definition of *hPulse* to get a signal which has an occurence every hour.

The most important function is *merge*. *merge* aggregates the occurences of two signals. Thereby, it combines occurences with equal time. Different external events are always considered to happen at different times. Therefore, the merges of pulse and press signals in the definition of *update* will not combine a pulse with a press. On the other hand, the outer merge of the *update* definition will combine each hour update with the corresponding minute update. If it would not, there would be two changes of the label on every full hour. One would change the hour digits and one the minute digits. Such behavior means a performance loss which is not critical in this case but becomes significant in situations where arbitrarily many signals are merged. Moreover, there would be an inconsistent intermediate state. Depending the order of updates, there would be transitions like 07:59 → 08:59 → 08:00 or 07:59 → 07:00 → 08:00. Such behavior can become cruical if signal occurences do not just affect a display but control some safety-critical device. Note that our discrete signal semantics do not even allow multiple occurences at the same time in the same signal.

Being a pure data flow system, our system lacks a switch function. Such a function would have type *DSignal val → DSignal (DSignal val) → DSignal val* and produce a signal which initially behaves like the first argument and switches to the behavior of any signal occuring in the second argument whenever a occurence takes place there.

14.3 IMPLEMENTING CIRCUITS

The key idea of our push-based FRP implementation is that the task of actually running the system is left completely to event handlers. Initializing the system, including registering the event handlers, is all which has to be done in the first place. Control is then given to an event loop which lets the event handlers do their job. This leads us to an implementation of circuits where a circuit is just an I/O action which does the necessary initialization work:

> **newtype** *Circuit i o = Circuit* (*Kleisli IO i o*) **deriving** (*Arrow*)

Note that we use GHC's generalized **newtype** deriving mechanism to carry the *Arrow* instance of *Kleisli IO* over to *Circuit*.

Since initialization done in the beginning might give rise to necessary finalization at the end, we allow the initialization action to output a finalization action in addition to the ordinary output. We use the *WriterArrow* transformer [12] for this:

> **newtype** *Circuit i o = Circuit* (*WriterArrow* (*IO* :$ ()) (*Kleisli IO*) *i o*)
> **deriving** (*Arrow*)

The :$ type operator denotes type application and is taken from the *TypeCompose* package [5]. Using *IO* :$ () instead of *IO* () gives us a *Monoid* instance for free

where *mempty* means no finalization and *mappend* means sequencing of finalizations. Thus, the finalization action of a circuit is automatically composed from the finalization actions of its parts.

The above implementation has a problem. It is not possible to provide a sensible *ArrowLoop* instance for *Circuit*. This can be demonstrated with the clock example. The input of the clock depends on the signal *mPress*. Event handler registration for the clock would need to know this signal in order to register a handler for reacting on presses of the *M* button. However, since this button would not have been constructed yet, this signal would not yet be known.

We solve this problem by only doing things like GUI widget creation in the first place and deferring event handler registration to the time when every other initialization has been done. Instead of writing a finalization action, the writer arrow now writes an action which registers the event handlers and returns the corresponding finalization action. This action is called a setup and has the type *IO* :$ *IO* :$ (). Again, we get a sensible *Monoid* instance. *mempty* means no registration and no finalization while *mappend* denotes sequencing of registration and finalization actions. The definition of *Circuit* becomes the following:

newtype *Circuit i o = Circuit* (*WriterArrow Setup* (*Kleisli IO*) *i o*)
   **deriving** (*Arrow*, *ArrowLoop*)

type *Setup = IO* :$ *IO* :$ ()

With this solution, the clock builds a setup which depends on the yet unknown *mPress* signal which is no problem because of lazy evaluation. When the registration is performed, all subcircuits have been created and therefore all subcircuit outputs are known. In general, feedbacks work as long as I/O actions which use a circuit's input are not needed to compute the circuit's output. However, if they are needed, they cannot be put into the setup because the setup cannot contribute to a circuit's output. We will see an example of this situation in section 14.5 and show that the above implementation has a shortcoming here. Section 14.6 will show how to remove this shortcoming.

14.4 IMPLEMENTING DISCRETE SIGNALS

Certain discrete signals describe sequences of external events whereby the occurence values are the event parameters. For example, the output of a *minutePulse* instance describes a sequence of minute change events, and the output of a *button* instance describes the sequence of press events of the respective button. Event parameters are always () in these cases. A discrete signal which directly mirrors an event sequence is provided by a *discrete source*. Each instance of *minutePulse* and *button* covers a discrete source while *label* instances cover none. A signal *depends* on a source if it is the signal provided by this source or if it is formed from a signal which depends on this source by application of a signal function.

A *discrete sink* consumes a discrete signal and reacts on occurences in it. Each *button* and *label* instance contains a discrete sink which changes the button's caption or the label's text, respectively.
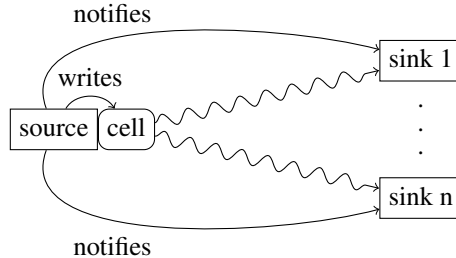
FIGURE 14.4. Event handling

An occurence in a discrete signal always originates from an external event. The reason is that all signal functions create only occurences with times which have been taken from the occurences of existing signals. Signal functions may transform and combine occurence values, and drop occurences but they never "invent" completely new occurences. So a sink has only to react to external events which are made available through sources. The aim of the internal structure of *DSignal* values is to make the communication between sources and sinks possible.

Figure 14.4 shows what happens in response to an event. Each source has a mutable variable attached to it. This is called a *cell* and is created during circuit initialization. Normally, the cell contains ⊥. Everytime an event occurs, the event's parameter is stored in the cell. Afterwards, all sinks which might have to react to the event are notified of the event occurence. They read the event parameter from the cell and determine whether the event occurence leads to an occurence in their input signal. If this is the case, they calculate the occurence's value and react to the occurence. Finally, the cell's content is reset to ⊥.

We introduce a type *DSource* for discrete sources:

data *DSource = DSource Unique Notifier*

type *Notifier = IO* () → *Setup*

Each source consists of a unique identifier and a notifier. Source IDs are generated during circuit initialization via *newUnique*. There are *Eq* and *Ord* instances of *DSource* which define equivalence and ordering of sources in terms of equivalence and ordering of their IDs. These instances are needed since discrete sources are used as keys in maps, as we will see soon. Notifiers are functions which map event handlers to setups. The setup for a given event handler registers this handler at the source so that it gets called on every event of this source.

The central part of a signal's internal structure is the *generator map*. This maps all sources, the signal depends on, to *generators*. A generator is an I/O action which outputs *Nothing* when an event of the corresponding source does not lead to an occurence in the signal, and *Just val* if it leads to an occurence of value *val*. The generator reads the source's cell in order to determine the generator's result. Since the *scan* function allows occurence values to depend on the signal's

history, a generator may receive references to mutable variables as its input. These mutable variables store values accumulated by *scan*. The references pointing to them are called *locals* and are provided to the generator in form of a nested tuple. The type definitions for generator maps and generators are as follows:

type *GenMap locals val = Map DSource* (*Gen locals val*)

type *Gen locals val = locals → IO* (*Maybe val*)

A discrete signal consists of an I/O action which creates any mutable variables for holding accumulated values, and of a generator map:

data *DSignal val = ∀locals.DSignal* (*IO locals*) (*GenMap locals val*)

The action for creating mutable state is run once by each sink which consumes the signal. Its result is used together with the generators to form the event handlers which are registered at the respective sources afterwards.

Using this definition of *DSignal*, all functions introduced in section 14.2 can be implemented. We present the implementations of *filter* and *merge*. They are shown in figure 14.5. We use the fact that *Map* is a functor and that (→) *locals* and *IO* are functors and monads for pushing function applications inside generators and generator maps via *fmap* and *liftM2*. *merge* transforms its arguments' generator maps into *prepared maps*. These already work with the result's locals and extract the part, they are interested in, via *fst* or *snd*, respectively. In addition, they apply *First* or *Second*, respectively, to the occurence values of the argument signals. When forming the result signal's generator map, generators belonging to the same source are combined via *combGen*. This makes sure that simultaneous occurences are combined.

14.5 MEMOIZATION

Consider two different sinks whose signals are derived from a common signal by signal function applications. The locals creation action of the common signal is contained by the locals creation actions of both sinks' signals. The analogue holds for the generators of the common signal. Locals creation of the common signal is therefore done for every sink instead of just once, resulting in an extra time and possibly space cost. A further time penalty arises from the common signal's generators being called for every sink.

To solve these problems, we introducing the *memo* circuit. *memo* has type *Circuit* (*DSignal val*) (*DSignal val*) and is denotationally equivalent to the identity arrow *pure id*. In contrast to *pure id*, the output signal can be used multiple times without imposing the time and space overhead discussed above.

Event handling in the presence of memoization is depicted in figure 14.6. An instance of *memo* for a signal of type *DSignal val* covers a mutable variable of type *Maybe* (*Maybe val*) which is called a *box*. Normally, the content of the box is *Nothing* which means that there is no information concerning occurences in the signal. During event handling for any of the signal's sources, the content is

filter prd (*DSignal newLocals genMap*) = *DSignal newLocals genMap*′ **where**

   genMap′   = (*fmap* ∘ *fmap* ∘ *fmap*) *outputConv genMap*

   outputConv Nothing   = *Nothing*

   outputConv (*Just val*) = **if** *prd val* **then** *Just val* **else** *Nothing*

merge (*DSignal newLocals$_1$ genMap$_1$*)

      (*DSignal newLocals$_2$ genMap$_2$*) = *DSignal newLocals*′ *genMap*′ **where**

   newLocals′   = *liftM2* (,) *newLocals$_1$ newLocals$_2$*

   genMap′   = *unionWith combGen prepGenMap$_1$ prepGenMap$_2$*

   combGen   = (*liftM2* ∘ *liftM2*) *combVal*

   combVal maybe$_1$ maybe$_2$ = **case** *catMaybes* [*maybe$_1$*, *maybe$_2$*] **of**

                            []   → *Nothing*

                            [*mergeVal*]   → *Just mergeVal*

                            [*First val$_1$*, *Second val$_2$*] → *Just* \$

                                                          Both val$_1$ val$_2$

   prepGenMap$_1$ = *fmap* ((∘*fst*) ∘ (*fmap* ∘ *fmap* ∘ *fmap*) *First*)

                                 genMap$_1$

   prepGenMap$_2$ = *fmap* ((∘*snd*) ∘ (*fmap* ∘ *fmap* ∘ *fmap*) *Second*)

                                 genMap$_2$

FIGURE 14.5.   **Implementation of the** *filter* **and the** *merge* **function**

changed to *Just Nothing* if there is no occurence in the signal, or to *Just* (*Just val*) if the value *val* occurs. Every generator of the *memo* instance's output checks the content of the box and changes it if the box still contains *Nothing*. It then uses the content of the box to produce its output.

    After all respective sinks have reacted to an event occurence, the content of the box has to be reset to *Nothing*. The *memo* instance is responsible for doing that. Therefore, it has to be notified after event handling is complete. We extend the *DSource* type with a second notifier. Handlers registered via this notifier are called after all handlers registered with the first notifier have been executed.

    In order to change the content of the box, the generator of the output signal calls the respective generator of the input signal. The locals argument used by this call is created by the *memo* instance during initialization. So in order to produce the output signal, the locals creation action of the input signal has to be executed. This means that this execution cannot be deferred by putting it into the setup but has to be performed during the first intialization stage. So at this time, the locals creation action of the input has to be already known. This is problematic if the memoized signal depends on a source which is situated after the *memo* instance.

    Signals which correspond to a certain source use *return* () for locals creation, independently of any I/O outputs. Therefore, locals creation actions of signals can be calculated purely functionally and could therefore be known early enough. Nevertheless, the output of an I/O action is completely unknown until the action
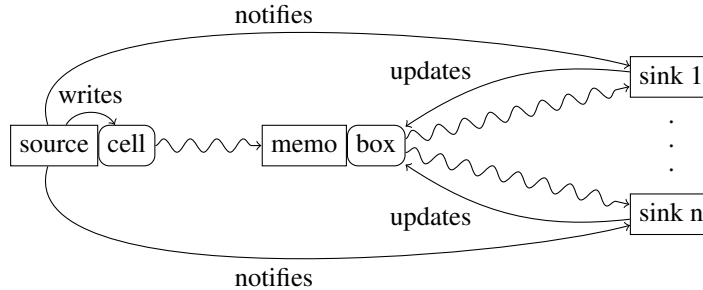
FIGURE 14.6. Event handling with memoization

has been executed completely. To illustrate this, take a look at the I/O Kleisli arrow

$$loop\ (\textit{first get} \ggg pure\ (uncurry\ (\mathbin{+\!+})) \ggg put \ggg pure\ (const\ ((),\texttt{"."})))$$

where $get = Kleisli \, \$ \, const \, getLine$ and $put = Kleisli \, \$ \, putStrLn$. Only after the last *pure* arrow, it is known that the output of *loop*'s argument is $((), \texttt{"."})$. So *put* will work with a feedback of $\bot$, not $\texttt{"."}$, and therefore fail.

So it is impossible to memoize a signal if this signal is derived from a signal which is fed back via *loop*. We will solve this problem in the next section. The idea is to automatically move all pure computations to the beginning of the initialization action. We develop a general arrow transformer to achieve this.

14.6 GETTING LAZY VIA RELAXING

We introduce a type *LaxArrow* of kind $(* \to * \to *) \to (* \to * \to *)$. For every *Arrow* instance *base*, *LaxArrow base* is an instance of *Arrow* and *ArrowLoop*. In addition, there exist the following functions:

$$impure :: (ArrowLoop\ base) \Rightarrow base\ i\ o \qquad\qquad \to LaxArrow\ base\ i\ o$$
$$runLax :: (Arrow\ base) \qquad \Rightarrow LaxArrow\ base\ i\ o \to base\ i\ o$$

Applying *impure* to a base arrow value results in a so-called *impure particle*. The *Arrow* and *ArrowLoop* methods of *LaxArrow base* can be used to build further lax arrow values from impure particles. Note that these methods add only pure computations.

Now, the idea is to perform all these pure computations at the beginning. *runLax* achieves this by generating a base arrow value starting with a pure arrow value, called the *converter*. The converter receives the outputs of all impure particles and the input of the complete arrow value as its input. It produces the inputs of the impure particles as well as the output of the whole arrow value. Following the converter there are the base arrow values which were used to construct
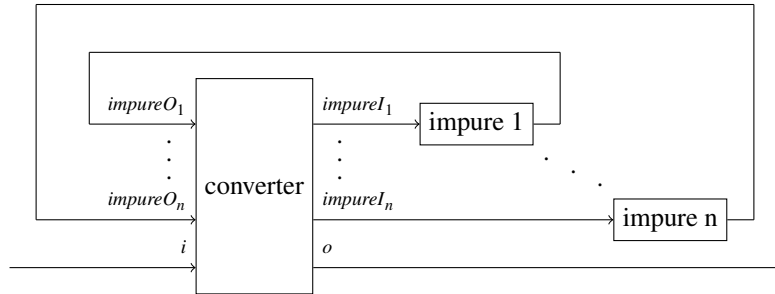
FIGURE 14.7.   **Structure of *runLax* results**

the impure particles. Their results are immediately fed back into the converter. Figure 14.7 shows the structure of *runLax* results graphically.

The feedback example from the last section can now made working by replacing *get* and *put* with *impure get* and *impure put*, respectively, and applying *runLax* to the whole expression. The result is equivalent to

$$loop \, (loop \, (pure \, conv \ggg second \, get) \ggg second \, put)$$

where *conv* is defined as follows:[5]

$$conv \sim (\sim(i, putO), getO) = (((), getO + +\texttt{"."}), i)$$

The $\texttt{"."}$ now appears directly in the input of *put* as generated by the converter. This is because $\texttt{"."}$ is fed back inside the converter. This feedback is not visible in the above definition of *conv* because this definition is already in a simplified form. Further note that *put* does not receive the result of *get* directly but only via feedback and through the converter. It is important that *get*'s output is fed back immediately after *get* because otherwise it would be available too late.

The type definition of *LaxArrow* is given in figure 14.8. Internally, a *LaxArrow* value consist of the converter function and the *base generator*. The base generator is a function which, given the converter as its argument, constructs the complete arrow value yielded by *runLax*. However, the base generator can also be applied to non-pure arrow values and to arrow values which use different types in place of the input and output types. This generalization is necessary for the implementation of ($\ggg$).

Using this definition, the implementation of *runLax* is straightforward. The definitions of the other lax arrow functions are quite complicated in part because of necessary tuple restructuring. We only give the general ideas here, for the exact definitions, the reader is referred to the source code of the *lax* package [10]:

---

[5]Note that *conv* is not exactly the converter function because the input and output tuples of the converter have a different structure than the arguments and results of *conv*. We have chosen this definition of *conv* to make the presentation a bit simpler.

data *LaxArrow base i o* = ∀*impureI impureO*.
$$LaxArrow\ ((impureO, i) \rightarrow (impureI, o))$$
$$(BaseGen\ base\ impureI\ impureO)$$
type *BaseGen base impureI impureO* = ∀*i′ o′*.
$$base\ (impureO, i')\ (impureI, o') \rightarrow$$
$$base\ i'\ o'$$

FIGURE 14.8. **Implementation of the** *LaxArrow* **type**

- Results of *impure* have a base generator which adds a single cycle containing the argument of *impure*. The converter function does only tuple restructuring.

- The converter function of a *pure* result applies *pure*'s argument to the input. The base generator does only tuple restructuring.

- The base generator of a (⋙) result is basically the composition of the arguments' base generators. The converter function is responsible for feeding the output of the first arrow to the second arrow.

- *first* and *loop* do not touch the base generator. They basically apply *first* or *loop*, respectively, to the converter function.

We now obtain an improved implementation of *Circuit* by replacing *Kleisli IO* with *LaxArrow* (*Kleisli IO*). Using this definition, *memo* instances are fully functional also in the presence of feedbacks.

14.7 RELATED WORK

The first push-based FRP implementation was developed for the Haskell GUI library *FranTk* [13, 14]. Alas, FranTk's merge function does not detect simultaneous occurences and therefore produces multiple occurences instead of combined occurences. Instead of a circuit arrow, FranTk uses a *GUI* monad which does not support feedbacks via a *MonadFix* instance. Feedbacks are implemented using so-called *wires* and *listeners* which leads to a less declarative style.

FrTime [3, 2] is a push-based FRP implementation in Scheme. FrTime runs an event loop in a separate thread. This makes it possible to incrementally develop and test reactive programs in the interactive DrScheme environment. There is a merge function for discrete signals which does not combine simultaneously occuring values. On the other hand, continuous signals in FrTime respect simultaneity. Feedbacks can be implemented declaratively using **letrec**.

Frappé [4] is an FRP library written in Java. It implements discrete and continuous signals as JavaBeans. Signals notify dependent signals about value occurences and value changes by calling listener methods. This has the consequence that simultaneity cannot be detected as multiple occurences at the same time are handled by independent method invocations. Feedbacks can be achieved only

imperatively. First, a partially-defined signal is constructed. The missing information is provided later when necessary signals are known.

Fudgets [1] is a Haskell GUI library. Its central concept is the *fudget* which is similar to a circuit with a single discrete input signal and a single discrete output signal. Fudgets are implemented as stream transformers using a continuation-passing style. While feedbacks are possible, simultaneously occuring values are not combined. Handling of events and modifying the environment needs time proportional to the depth of the respective fudget in the fudget hierarchy.

The Haskell library *Yampa* [8] is a popular example of a pull-based approach to FRP. Yampa represents discrete signals as continous signals over *Maybe* types[6] where applications of *Just* represent occurences while *Nothing* stands for the absence of occurences. The downside of this approach is that Yampa cannot enforce that values occur at discrete times. Due to the pull-based implementation, the time for updating the system state grows linearly with the system size. On the other hand, Yampa supports recursive signal definitions which are not possible with our approach. Feedbacks and simultaneity are no problems in Yampa.

The declarative data flow languages *Signal* [6] and *Lustre* [7] are taylored to the concepts we dealt with in this paper. Like Yampa, they allow recursive signal definitions. The compiler even checks that such definitions do not lead to cyclic value dependencies. Furthermore, certain properties of occurence times can be checked statically. Simultaneous occurences are identified and handled correspondingly. In Lustre, time and space leaks are prevented and the adherence to given time and space constraints can be checked. Compilation of Lustre programs generates efficient code by using finite automata as an intermediate representation.

14.8   CONCLUSIONS AND FURTHER WORK

We have presented a push-based implementation of a data flow library. Our system allows for declarative descriptions of reactive systems, including system with feedbacks. To make feedbacks work, we employed a writer arrow for delaying parts of the system initialization. We developed lax arrows to enable feedbacks in combination with memoization. With lax arrows, parts of an arrow value are re-arranged so that pure computations are performed first which makes their outputs available to all impure computations. Our implementation also provides a merge function for discrete signals which combines simultaneously occuring values instead of generating multiple occurences.

Ongoing research deals with extending our signal implementation to support switching. Futhermore, we investigate implementation techniques for continuous signals. In addition, we are working on support for circuits whose structure changes over time, and for incrementally updating continuous signals. *merge* could execute generators and locals creation actions of its argument signals concurrently. It should be analyzed how this can help improving performance.

---

[6]Actually, Yampa uses a type *Event* which is isomorphic to *Maybe*.

14.9 ACKNOWLEDGMENTS

REFERENCES

[1] M. Carlsson and T. Hallgren. *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology/Göteborg University, 1998.

[2] G. Cooper and S. Krishnamurthi. FrTime: Functional reactive programming in PLT Scheme. Technical Report CS–03–20, Brown University, Providence, RI, Apr. 2004.

[3] 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*, pages 294–308. Springer, Berlin/Heidelberg, 2006.

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

[5] C. Elliott. *TypeCompose-0.3* (Haskell package). `http://hackage.haskell. org/cgi-bin/hackage-scripts/package/TypeCompose-0.3`, Dec. 2007.

[6] T. Gautier, P. L. Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, Berlin/Heidelberg, 1987.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[8] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, Berlin/Heidelberg, 2004.

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

[10] W. Jeltsch. *lax-0.0.0.1* (Haskell package). `http://hackage.haskell.org/ cgi-bin/hackage-scripts/package/lax-0.0.0.1`, Mar. 2008.

[11] R. Paterson. A new notation for arrows. *ACM SIGPLAN Notices*, 36(10):229–240, Oct. 2001.

[12] R. Paterson. *arrows-0.4* (Haskell package). `http://hackage.haskell.org/ cgi-bin/hackage-scripts/package/arrows-0.4`, Feb. 2008.

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

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