A review of "Push-Pull Functional Reactive Programming"

J.P. Pizani Flor[1]

[1] Department of Information and Computing Sciences, Utrecht University - The Netherlands
e-mail: j.p.pizaniflor@students.uu.nl

1 Introduction

The paper "Push-Pull Functional Reactive Programming", by Conal Elliott, discusses Functional Reactive Programming (FRP), a technique developed around 1996 and which can be applied to the design of Graphical User Interfaces in a purely functional fashion, and is highly expressive and elegant. However, the FRP paradigm had not found (until then) an efficient implementation.

On the way to implement FRP more efficiently, the author first redefines and organizes FRP semantics in terms of very well-known type classes (*Functor*, *Applicative* and *Monad*). Then the central FRP concepts of *Behavior* and *Event* are given new definitions and are made instances of these type classes. Finally, using these new definitions a more efficient, push-pull (hybrid between data-driven and demand driven) implementation for FRP is derived.

2 FRP Fundamentals

Functional Programming Programming (FRP) is in short the functional embodiment of Reactive Programming, a paradigm of programming that sees computation through the establishment of dataflows and the propagation of changes in values through these dataflows. One good example of a reactive programming setting is a spreadsheet program, where a change in the value of one cell might cause *reactions*, triggering the values of other cells and so forth, in a cascade of change propagation. One application realm which is said to fit FRP very well is the programming of Graphical User Interfaces (GUIs).

FRP derives its expressiveness from 2 fundamental concepts, which can be embodied in types: a type with dynamic values, i.e, *values over time*, and a type representing an "event source", which is just a (infinite) stream of values, paired with the point in time when they occur. Elements of the first datatype (dynamic values) are traditionally called *behaviors* and elements of the second datatype are called *events*. One example of a behaviour would be the position of an object in an animation. When in the context of a GUI, a simple example of an event source would be the user's mouse clicks.

3 Packaging FRP in standard typeclasses

In the original formulation of FRP there was a family of functions ($lift_n$) that "lifts" a function over values to a function over behaviors. We can define $lift_1$ with an instance of the Functor typeclass, which – considering the semantic domain of behavior to be a function of time – has the following definition:

```
instance Functor ((->) t) where
    fmap f g = f . g
```

The denotational semantics of fmap (and of every other operation involving behaviors) can be defined with the help of the function "at". The "semantic instance" for Functor bellow is not a real instance, but establishes a law which any implementation must comply with, and is:

```
instance Functor Behavior where  -- semantic instance
    at (fmap f b) = fmap f (at b)
```

With the Functor instance for behaviors, we can write $lift_1$. With an additional instance for Applicative, it is also possible to define any of the functions in the family $lift_n$. As the semantic domain of behaviors is a function of time, the Applicative instance of functions is helpful when defining the semantics of behaviors.

```
instance Applicative ((->) t) where
    pure    = const
    f <*> g = \t -> (f t)(g t)

instance Applicative Behavior where  -- semantic instance
    at (pure a)    = pure a
    at (bf <*> bx) = at bf <*> at bx
```

In this way the denotational semantics of a behavior and operations over behaviors are nicely composed using instances of standard Haskell typeclasses, and the semantics is precisely specified *without being mixed with implementation concerns*. This is an important point made by the author, as this freedom allows the more efficient implementation presented on this paper.

In the same way done with behaviors, the concept of *Event* (or event source) can also be package in instances of standard typeclasses. The *monoid* instance for Event is analogous to the list monoid, and the monoid functions “mempty” and “mappend” replace classic FRP’s “neverE” and “(.|.)”, which are the never-occurring event and an operation to merge two event sources. In the semantic monoid instance for Event it is also made clear that time-ordering is preserved in the merging operation. The functor instance for Event is also straightforward, and the semantic instance expresses that mapping a function over an event changes only the values, leaving occurrence times intact:

```
instance Functor Event where  -- semantic instance
    occs (fmap f e) = map (\\(t, v) -> (t, f v)) (occs e)
```

The monad instance for Event is more interesting, and we can come to the need of a monad instance by thinking about applications where there is a Event-valued event source, Event (Event a). Such an event source can be used to model scenarios where some events might cause a new event source to “come alive”. In the paper the example is given of an Asteroids game, where hitting an asteroid might cause it to break in multiple pieces, which collisions must be tracked – thus, one event is able to “spawn” an event source.

Usually monad instances are defined by writing the functions *return* and *bind*, but in this case a different approach is taken: a monad instance can equivalently be defined by the functions *fmap* and *join*. The Functor instance already takes care of fmap, and the definition of join below “flattens” a nested Event, making sure that the inner events do not occur before the outer Event brought that source to life. The denotational semantics of join is again defined in terms of the occs function:

```
join :: Event (Event a) -> Event a
occs (join ee) = foldr merge [] . map delayOccs . occs ee
```

4 Problems in current implementations

The main goal of the paper is to provide a more efficient implementation for FRP. The redefinition of FRP concepts in terms of standard typeclasses, along with the precise definition of its denotational semantics was only a means to achieve this goal. Now, with this precisely-defined semantics in hand, we can attack the following problems:

– Merging two event sources involves comparing the occurrence times in both of them (to maintain time-ordering). This can cause the handling of two events to happen only when the **later** of the two arrives. This can be solved by embedding *partial information* in the values used as time, and is achieved in this paper with the abstraction called "future values"
– The semantics of the *switcher* function involve searching an event source for events which occurrence time match certain criteria. This search becomes costlier as time goes on. In most use cases of FRP, time evolves in a monotonically increasing way, and exploring this fact can improve implementation efficiency.
– The definition of behaviors as functions impose an inefficient implementation. In the paper, a new representation is introduced for behaviors, making explicit the non-reactive and constant phases.

5 Future values and future times

To simplify the semantics of events and improve its implementation, as well as provide a better representation for time, the concept of "future values" is introduced. A future value is just a value with an associated time (in which it is bound to be "available". By taking the semantics of a future value to be a pair, the functor instance is just a partial applied pairing and the semantic function is a functor morphism (a function that *preserves the structure* of the functor).

```
type F a = (T, a)  -- T must satisfy some pre-requisites defined later
force :: Future a -> F a

instance Functor ((,) t) where
    fmap f (t, v) = (t, f v)  -- straightforward functor instance

instance Functor Future where  -- semantic instance
    force (fmap f u) = fmap f (force u)
```

The Applicative instance for Future is also somewhat straightforward, but has an additional requirement: the time type, T, must be a monoid, so that we can use *mempty* as the time of (pure a), and *mappend* to combine the times of 2 future values. Furthermore, the definition of mempty must be a lower bound for T (T must be Bounded), and mappend must choose the maximum between two times. Besides that, the definition holds no surprise, and the semantics is defined so that *force* is an Applicative morphism.

Further instances of Monad and Monoid are given for future values in the paper, along with corresponding semantics. It is interesting to note that the semantics given might be directly used as implementation if the type of future times chosen satisfies the following properties:

– Is ordered and bounded, i.e, a member of the typeclasses Ord and Bounded
– Is a monoid in which *mempty* is minBound and *mappend* is max.
– The structure of the type must reveal partial information on times, so that comparisons can be performed even when the times are not yet fully known.

These three requirements can be embodied in three newtypes, with Max and AddBounds having pretty straightforward Monoid, Ord and Bounded instances, respectively. The innermost newtype in this onion-like structure (Improving) is describe in more detail later, and provides partial information in time representation.

```
type FTime = Max (AddBounds (Improving Time))
```

6 Separating the reactive and non-reactive parts of behaviors

One of the biggest problems posed by the author FRP implementations is the demand-driven model, caused by the implementation of (reactive) behaviors simply as a function of time. However, this is not the only possible implementation: as noticed by the author, a behavior has several non-reactive *phases*, punctuated by events. Then, if we can view behaviors through a datatype that makes this distinction explicit, we can exploit this in an implementation (for example, only one sample is needed to represent a constant phase). The new representation proposed for (reactive) behaviors is built on top of two concepts, reactive values and time functions, each embodied in a Haskell datatype.

Reactive values are discrete "step" functions, with time as their domain, and are defined by an initial value and some discrete changes, represented exactly by an FRP Event.

```
data Reactive a = a ‘Stepper‘ Event a
```

Reactive values have pretty straightforward instances for Functor, Applicative and Monad, and its semantic function (*rat*, which translates a reactive value into a reactive behavior) is a morphism on the Functor, Applicative and Monad instances.

Now, in-between the moments of discrete change, the value of a behavior can change according to a non-reactive (and possibly continuous) function of time. Even though we might represent the concept of a function directly as a Haskell function, this is not very useful. The authors chose to create a datatype of time functions, in which two cases are distinguished: the case in which the function is constant and the case in which it's not (this allows for optimization in the common use case of constant functions):

```
data Fun t a = K a | Fun (t -> a)
```

Off course there are also neat instances of Functor, Applicative and Monad for the Fun datatype, and its semantic function (*apply*, which just corresponds to usual function application) is a morphism over the instance of all these type classes.

Having these two component in hand (reactive values and time functions), (reactive) behaviors can be given a new definition:

```
type Behavior = Reactive . Fun Time  -- the dot is type-level composition, as below
newtype (h . g) a = O (h (g a))
```

This means that behaviors are, in essence, reactive values in which the values themselves are time functions. In the common case of step functions, for example, each time function is simply a constant. Instances of Functor and Applicative are given for the type-level composition operator, which (together with the instances for Reactive and Fun) give us Functor and Applicative functionality for free for this new behavior representation.

7 Events redefined and monotonic time sampling

After the definitions of future values (a value and a time when it will occur) and reactive values (an initial value and a sequence of discrete changes), we can define Event quite elegantly by using these two concepts:

```
newtype Event a = Ev (Future (Reactive a))
```

That means that an Event is a time and a value (the first occurrence) and a sequence of discrete changes. This corresponds intuitively with the original definition of Events, in terms of lists of occurrences, but solves a subtle problem which impacts performance: By using a list of occurrences as a model for Event, we had to test the list for emptiness when merging two Events. This test results in a Bool, which can provide no partial information and thus must wait before all occurrences arrive before yielding a result.

After this redefinition of Event, the paper goes on to discuss a very important optimization for implement FRP efficiently: *monotonic sampling*. The key observation is that, even though the semantics of FRP don't restrict behaviors – and they can be applied to time values in any order (they are just a function of time) – in *real-world situations* behaviors are evaluated with monotonically increasing times as arguments. Thus, making this usage pattern explicit and exploiting it might result in bug efficiency gains.

The paper goes on to discuss how to render behaviors forward in time. For this we first need functions that take "sinks" (functions that have `IO ()` as counter-domain) and consume Reactive values and Events:

```
sinkR :: Sink a -> Reactive a -> IO b
sinkE :: Sink a -> Event a -> IO b
```

Their implementation is simple, and they are mutually recursive: sinkR immediately renders the initial value and then proceeds to render the nested Event (the "tail"). The function sinkE waits (blocks) until the first occurrence time t_0 and then proceeds to render the nested Reactive value.

The whole "puzzle" on how to render a behavior forward in time is almost complete, there being only a single missing piece – how to render a time function. With this piece in place, an implementation of how to render a behavior forward in time finally comes together:

```
sinkN :: Sink a -> Behavior a -> IO b
sinkN snk (O rf) = newTFunSink snk >>= flip sinkR rf
```

We first create a sink for time functions, then use sinkR to render each of the time functions in the sequence (the Reactive sequence). And how does this sink for time functions behave? Well, if the function is constant, its value is rendered just once (using the snk parameter). If the function is truly time-dependent, a thread is spawned which samples and renders its value periodically.

8 Partial information on occurrence times

If we refer back to the end of section 5, a mention is made to a type constructor `Improving`, in the definition of the type of future times. Applying this constructor to a time type yields a type containing partial information, and thus allowing for comparisons between future times to evaluate before either of them has fully arrived.

The name of the type constructor mentioned in section 5 (`Improving`) is not a coincidence, as the abstraction used by the author to solve this problem is that of *Improving values*, invented by Warren Burton in 1989. An improving value can be seen as a sequence of lower bounds ending up in an exact value (in our case a time).

The author further improves on improving values (no pun intended) by establishing a method with which to compare two future times. It is assumed that we have a function that can compare an exact value with an improving value. Then faced with the comparison between two future times $t_a \leq t_b$ we can do it in two ways:

1. Extract the exact time from t_a and compare it with t_b
2. Extract the exact time from t_b and compare it with t_a

Now, these methods will result in the same value, but not at the same time. The problem is that if $t_a \leq t_b$ then the first method will be quicker, otherwise the second. How to solve this? Well, both approaches are attempted in parallel and whichever one finishes first is returned. This parallel implementation is done by using Concurrent Haskell and a `MVar`.