Denotational Semantics of SimpleSwiftFRP

This document attempts to define the denotational semantics of SimpleSwiftFRP in a manner that is implementation independent.

Anyone reading this who is familiar with FRP will see that it has borrowed numerous concepts from other FRP systems. To pause at every idea that originated with someone else would be impossible, since every idea originated with someone else. Additionally, it turns out that some ideas are simply inherent in the concept of Functional Reactive Programming, and pretty much common between them all. Oftentimes while working on it I would discover that I had discovered what had already been discovered.

But in particular, I drew most of the ideas from the work of Conal Elliott and Paul Hudak, starting with their paper on FRAN. *Behavior, Event, lift* and *at,* are all concepts drawn directly from their work.

Additionally, I want to thank Stephen Blackheath from the Sodium project. SimpleSwiftFRP started out as an attempt to port Sodium to Swift, but faltered when I realized I didn't understand enough Java to really understand how it worked, so I took some of the ideas I had learned and worked up my own implementation. Before I gave up on Sodium, however, Mr. Blackheath was kind enough to answer some of the questions that I had, and his book Functional Reactive Programming, available through Manning Publications, is a good introduction to FRP.

For those who are formally trained in math and/or logic, I apologize in advance. I have no formal training in those fields. I have done my best to express the ideas in a logical, mathematically sound form, but at times I had to use my own symbols and/or terminology because I lack the formal training. At times I resort to just saying what I mean in natural language.

This whole process has been a learning exercise for me. Comments, criticism, advice, and suggestions are not only welcome - they are solicited. I can be reached at letvargo@gmail.com.

Time: *Time* is the total order consisting of all real numbers R where

$$-\infty < R < \infty$$

The continuous set of all valid *Time* values is denoted by the capital letter T.

A lower case t is a value in T.

To express relationships between times a lower case t followed by a numeral is used, *e.g.*

$$t_0 < t_1 < t_2 < \ldots < t_n$$

A unit of time is represented by *s* for seconds, *m* for minutes, or *h* for hours. For example, *10s* denotes 10 seconds.

Values: A value is a function of *Time*.

F(T) denotes the set of all valid functions on *Time*.

f(exp_T) is a particular function on *Time*, where exp_T is an expression that evaluates to a value for all *t* in *T*. exp_T is a function of type *t -> a*.

Some examples:

f(t)	the identify function on *t*
f(t + 2s)	2 seconds after *t*
f(t / 2)	*t* reduced by a factor of 2

A constant is a function *f(exp_T)* where exp_T does not reference *t*. Without any reference to *t,* exp_T will evaluate to the same value for all *t* in *T*.

Some examples:

f(2)	the constant number 2
f("Hello")	the constant string "Hello"

Events: An *Event* is a *(T_a, F(exp_{Tb}))* pair where the subscripts *a* and *b* indicate that the *t* used for the first element of the pair need not be the same as the *t* that is used to evaluate the exp_T in the second element of the pair.

Given an *Event e*, we can denote the first element of the pair, T_a, as *e.t* and the second element as *e.f*.

The first element, *e.t*, is an ordering value. It represents some *t* in *T* when an *Event* is said to occur.

The ordering value allows us to state that one *Event* occurred before another. Given two events, e_0 and e_1, e_0 is before e_1 if $e_0.t < e_1.t$. This relationship creates a partial order between *Events*, allowing us to write $e_0 < e_1$.

Furthermore, we state as a rule that for any two *Events* e_n and e_m where $n > 0$ and $m > 0$, and $n \neq m$, $e_n.t \neq e_m.t$. Under this rule no two *Events* other than e_0 can occur at the same time. *Events* are, by definition, serial in nature and cannot co-occur. e_0 is a special case discussed below.

The second element, *e.f*, is the *Event's* value as a *F(T)*. We set as a condition that *e.f* return the same type of value for all *t* in *T,* and we attribute that type to the *Event*. For

example, an *Event* of the Integer type will have an *e.f* element that returns an Integer value for all *t* in *T*.

When it is necessary to specify the particular type of an *Event* the type is placed in angle brackets, *e.g*. e_0<*Int*> specifies an Integer type *Event*.

With the exception of *e0, Events* occur serially, and no two events > e0

Behaviors: A *Behavior* is a higher-order representation of values that change over time represented as a series of one or more *Events,*

$$e_0, e_1, e_2 \ldots e_n$$

where all of the *Events* are of the same type. As with *Events*, we specify the type of a *Behavior* using angle brackets.

To evaluate the value of a *Behavior* at a particular *Time t,* we apply the following algorithm:

1. If the *Behavior* consists of only one *Event*, e_0, we apply $e_0.f$ to *t* and return the result as the value.
2. If the *Behavior* consists of *n Events*, we start with e_n. If $e_n.t \leq t$, we apply $e_n.f$ to *t* and return the result as the value.
3. Otherwise, discard the last *Event* and repeat from step 1 using *Events* $e_0 \ldots e_{n-1}$.

The primitive operation *at* is a function with the following definition,

at :: Beh a -> t -> a

which applies the above algorithm to the *Behavior a* at *Time t* to return a value of type *A*.

As an example, we begin with a *Behavior a* that consists of a single *Event:*

$$Beh\ a = \{ e_0 \}\ where\ e_0 = (t_0, F(7))$$

b represents the number 7. Because it consists of only one *Event*, applying *a `at` t* will evaluate to 7 for all *t* in *T*.

Now we assume a second *Event* $e_1 = (t_1, F(12))$, such that $Beh\ a = \{ e_0, e_1 \}$. Applying *at* to *a* now will produce the following results depending on the value supplied for *t:*

$$Beh\ a\ `at`\ t = F(12) = 12\ for\ all\ t\ in\ T\ where\ t_1 \leq t < \infty$$

$$Beh\ a\ `at`\ t = F(7) = 7\ for\ all\ t\ in\ T\ where\ -\infty < t < t_1$$

The value of *Beh a* will be 12 for all *t* in *T* greater than or equal to t_1 and 7 for all t in T less than t_1. In natural language we would describe *Beh a* as, "7 until t_1, then 12."

Assuming *Beh a* had a third *Event*, $e_2 = (t_2, F(-4))$, *Beh a* would evaluate as, "7 until t_1, then 12 until t_2, then -4."

The above example uses a succession of constant *F(T)*, but continuous *F(T)* work in the same way. For example, we can model the change from Daylight Standard Time to Daylight Savings Time as a *Behavior* like this:

$$Beh\ a = \{ e_0, e_1 \}\ where\ e_0 = (t_0, F(t)),\ e_1 = (t_1, F(t + 3600s))$$

In this example, e_0 is the identity function on *Time* and e_1 is the identity function plus 3,600 seconds, or 1 hour.

Synchronization of Time values: From the above examples it should be apparent that e_0 has a special quality. It does not "begin" at a particular point in *Time*. $e_0.f$ will be the *F(T)* for all *t* in *T* where $t > -\infty$ up until some new *Event* e_1 and therefore the actual value of $e_0.t$ does not matter.

Because the actual value of $e_0.t$ does not matter, we assign it the arbitrary time value of 0 and all other *Time* values are measured against it. This allows us to synchronize *Time* values between *Behaviors* using a single, absolute *Time* scale.

Lift: *Behaviors* can be first-order or higher-order. A first-order *Behavior* is created by *lifting* a *F(T)* into a *Behavior.*

We define $lift_0$ as the primitive,

$$lift_0 :: (t -> a) -> Beh\ a$$

where the result of applying *at Beh a* is *t -> a*.

Using the identity function on *Time* as an example,

$$lift_0 (t -> t) = Beh\ t$$

where *Beh t `at` t = t* for all *t* in *T*.

Behaviors themselves can be *lifted*. Given a function *a -> b* and a *Beh a* we can create a new *Beh b* like so,

$$lift_1 :: (a -> b) -> Beh\ a -> Beh\ b$$

where lift in this case is defined as applying the function *a -> b* to *at Beh a t,*

$$(a -> b)\ (at\ Beh\ a\ t)$$

which, we may see, is a function that takes a *t* and returns a *b*, *i.e.* it is a function of the form *t -> b*. In other words,

$$Beh\ b = lift_1 (a -> b)\ Beh\ a$$

is the equivalent of

Beh b = lift₀ ((a -> b) (at Beh a t))

which the equivalent of *lifting* a function *t -> b* to obtain a new *Behavior* of type *Beh b*.

Using this same method we can combine *Behaviors* to create a new *Behaviors* by supplying a combination function and *lifting* it. We lift two *Behaviors* into a third, like this:

lift₂ :: (a -> b -> c) -> Beh a -> Beh b -> Beh c

We apply the same process, using *at Beh a t* and *at Beh b t* to form a new function of the form *t -> c*, which is *lifted* to provide *Beh c*.

Beh c = lift₀ ((a -> b -> c) (at Beh a t) (at Beh b t))

Thus, we can generalize *lift* into a general proposition,

$lift_n$ :: (b_1 -> b_2 ->…-> b_n) -> Beh b_1 -> Beh b_2 ->…-> Beh b_n

In this way *Behaviors* can be combined to form new *Behaviors* of theoretically infinite complexity.

When a higher-order *Behavior* is lifted from other *Behaviors* we say that it *listens* to those *Behaviors*, in the sense that the value of the higher-order *Behavior* is just a function of the lower-order *Behaviors* at any given point in *Time*.

Merge: SimpleSwiftFRP defines the primitive operation *merge* whereby a multiple *Behaviors* are merged and lifted into a new *Behavior* such that a change in any one of them will induce a change in the lifted *Behavior. Merge* takes the form,

merge :: (a -> b) -> [Beh a] -> Beh b

and is defined as the *Behavior* that results from applying *at max([Beh a) t* to the transformation function *(a -> b)*, and is equivalent to the following *lift* operation:

Beh b = lift₀ ((a -> b) max([Beh a]))

Accumulate: *Accumulate* is a primitive operation whereby a *Behavior* is lifted into itself and takes the form,

accumulate :: (a -> a) -> Beh a -> Beh a′

An example of an accumulator is a counter that counts events of a certain type, such as mouse-clicks, keypresses, network requests, etc.

Snapshot: *Snapshot* is a primitive operation where a *Behavior* is lifted from multiple lower order *Behaviors*, but only listens for changes to only one of the lower-order *Behaviors*. It takes the form of a regular *lift* operation:

snapshot :: (a -> b -> c) -> Beh a_e *-> Beh b -> Beh c*

The subscript a_e indicates that *Beh c* will only change when *Beh a* changes. *Beh b* may change multiple times, but those changes will be ignored until *Beh a* is receives a new *Event* and becomes *Beh a′*.

Filter: The *filter* primitive enables a *Behavior* to ignore *Events* that do not meet a certain criteria. It takes the form,

filter :: (a -> Bool) -> Beh a -> Beh a′

In this formulation, when a *Behavior* is updated, the predicate function *a -> Bool* is applied to *at Beh a t* and a new *Behavior, Beh a′* is returned that either includes the additional value or is identical to *Beh a*, depending on whether the predicate evaluates to true or false.

Send: *Send* is the operation by which we add an *Event* to a *Behavior* to create a new *Behavior.*

send :: Ev a -> Beh a -> Beh a′

The effect of the *send* operation is to create a new *Behavior* that is identical to the original *Behavior* with respect to all $t < e_n.t$.

The use of *a′* denotes the fact that while a new *Behavior* is produced, it should take the place of the original *Behavior* such that all higher-order *Behaviors* that have been *lifted* from the original *Behavior* will continue to listen to the new *Behavior.* Throughout this document, a ′ indicates that a new *Behavior* is to take the place of an old *Behavior* will preserving all relationships to other *Behaviors*. The mechanism for accomplishing this is left up to the implementation.

Send is the only mechanism by which new values are introduced into the FRP system. As noted earlier, *Events* after *e0* are serial, and no two subsequent *Events* can occur at the same time. Thus, *send* must be serial and may only introduce a single value at a time.

This does not, however, preclude the categorization of tuples as single values, *i.e.* you can have a *Behavior* of the form *Beh (a, b)*, where *(a, b)* is a tuple with a value that changes over time. In such a situation, *send* will send an *Ev (a, b)* to *Beh (a, b)*, but this will be a single *Event* and the tuple *(a, b)* is considered a single value.

Send represents input from the external environment. The source of the input is left up to the implementation. In SimpleSwiftFRP we create a type called *Source* which

implements the *send* operation. A *Source* is connected to some part of the external environment that is capable of generating input - a keypress, a GUI button, or some other program generated input such as reading from the file system.

Because a *send* operation introduces a single value into the system, *send* is the equivalent of lifting a single value into a *Behavior* and therefore only first-order *Behaviors* can be used as an argument in a *send* operation. Conceptually, higher-order *Behaviors* are simply functions of first-order *Behaviors* and changing a first-order *Behavior* is the equivalent of changing a single argument to the transformation function of a higher-order *Behavior*, and all higher-order *Behaviors* will reflect the changes to any first-order *Behavior* that it listens to.

Output: Side effects are modeled by the *output* operation. An *output* operation is attached to a *Behavior* and represents some change in the external environment that occurs as the result of a change in the value of the *Behavior*.

As with *send*, the manner in which side effects are handled is left up to the implementation. SimpleSwiftFRP uses a type called an *Outlet* which *listens* for changes to a *Behavior* and performs a side effect whenever a change occurs.

The purpose of the *output* operation is to clearly define where and when the FRP system may generate side-effects. Just as *send* is the only mechanism for introducing new values into the FRP system, *output* is the only mechanism for effecting a change in the external environment. *Behaviors* are expected to be pure functions that do not change any aspect of the external environment.

Moreover, an *output* operation cannot change any value *within* the FRP system except by invoking *send*.

The intention of these rules is to create a domain of referential transparency inside a larger environment that may or may not be referentially transparent. Thus, the role that the FRP system plays in the computing system as a whole is left up to the programmer.

Summary: That summarizes the formal properties necessary to implement SimpleSwiftFRP. The code for the current implementation in Swift is available on Github at https://github.com/letvargo/SimpleSwiftFRP. The author is available for correspondence and comments at letvargo@gmail.com. Communication is welcome.