

Behavioral Amnesia: Gradual Accumulation of Memory for Temporal Values *3rd DRAFT*

Noam Lewis
lenoam → gmail.com

March 8, 2010

Abstract

It's impossible to tell the future. Furthermore, in our physical reality, there is no way to access arbitrary events (or phenomena) in the past. We must store information in real-time if we want to use it later. Functional Reactive Programming (FRP) aims to supply a semantically simple and precise model for programming temporally reactive systems. This short report is about an attempt to form a semantic model for FRP that includes a restriction on arbitrary access in time. We explore the idea of realistic accumulation of memory by considering different classes of time dependent values. Implementation issues (which may turn out to be critical) are purposely ignored.

1 Introduction

As a programmer experienced mainly in imperative languages, I am still regularly amazed by the simplicity and elegance that functional programming offers in general, and that FRP (Functional Reactive Programming) seems to promise in particular. However, awe and amazement are no replacement for comprehension. The question “What is FRP?” still has no clear answer in my mind. Having read a little about FRP ((Elliott and Hudak 1997), (Nilsson et al. 2002), (Courtney et al. 2003), (Hudak et al. 2003), (Elliott 2009b)) and several blog posts, and after discussing FRP with different people from various different angles, I am still confused. FRP seems to mean several different things, and I like when definitions are precise. So how should we define FRP? Lately, following insights shared by Conal Elliott and others, I've realized that the essence of FRP should be summarized by at least the two following notions:

- *Semantic precision* and clarity, as manifest by the denotational approach to design (see (Elliott 2009a)).
- *Temporality*, or more specifically a functionally pure, referentially transparent and composable approach to temporally changing values. Time, as in real life, is continuous.¹ For a starting point about the reasons for working with continuous time, see (Elliott 2010b).

Elliott suggested an alternative name for FRP: “Functional *Temporal* Programming”.² For a paradigm that is centered on the above two concepts, I suggest also “*Denotational Temporal* Programming”. I'm not sure that these concepts are what people mean when they talk about FRP. If yes, then “denotational-temporal” can be used to clarify what FRP is about. If it *isn't* what people mean, then I don't know what they do mean. In any case, in this report FRP means an approach that concentrates on temporally changing values and denotational design. With this point cleared up we may proceed to the issue at hand.

2 Remembering and forgetting

In some incarnations of FRP, temporally changing values are divided into two classes, each with its own denotation:

¹In our normal life it is. I'm disregarding the possibility of amazingly short time quanta that quantum physics introduces.

²With an unfortunate acronym - FTP.

- Values that depend on continuous time. They are called *behaviors*. The denotation is a function $\text{Time} \rightarrow \mathbf{a}$ where \mathbf{a} is the value type. Time , for our purposes, can be just \mathbb{R} .
- Values that “occur” at specific time points. These are known as *events*. One denotation is a list of occurrences, $[(\text{Time}, \mathbf{a})]$, with monotonically non-decreasing times.

Both denotations offer the possibility of “querying” the value at arbitrary times (see (Elliott 2010a)). For behaviors we can invoke the function with any time as an argument, and with events we can traverse the list to read the value at any time. However, arbitrary access in time is not realistic, it violates the design principle of “What would reality do?” (WWRD). Reality suggests that systems should not only be forbidden from accessing future values (breaking causality), but they should also not have access to arbitrary *past* values. In reality, after all, memory is lossy and finite.³ Just as it is impossible to know the future, there is no way to reach back in time and examine a temporal value as it was in the past. The best we can do is store values, and memory is finite. We are allowed to remember, but we must also forget. This limitation leads to the conclusion that an appropriate denotation for temporal values must not allow arbitrary access in time.

If we can’t access the past nor the future, if all we can access is the present, then how can we compute anything (beside point-wise computations) from a temporal value? How do we compute things that require memory? For example, it seems impossible to integrate a numerical temporal value. After all, integration requires knowing the history of a function until the current time. Similarly, we may ask how to do memory-full computations on event streams. As an example consider a value that is being “edited” as time goes by. The editing is done by discrete actions, so at discrete points in time the value suddenly changes. If the editing computation depends on the previous result, then don’t we need to access the past? More generally, how powerful does the temporal model have to be, what operations should be possible while retaining the limitation on arbitrary time access? I propose the following operations as test cases for a temporal model:

- Integration and differentiation
- Time delay
- Maximum (or minimum)

Note that integration will allow us to implement (for vectorial values) the general class of linear time-invariant causal systems (via convolution with a known impulse response). This is just a test of the minimum “power” that we want from our denotational framework. Integration requires memory up to the current time. Lastly, maximum is an example of an operation that also requires such memory but can’t be defined (as far as I know) using integration.⁴

We may ask the same question about reality. How does reality perform memory-full operations? Integration, in reality, can be achieved by a capacitor (integrates current, stores result in charge). A capacitor doesn’t need to access arbitrary past currents to determine the present charge. A capacitor integrates by adding up a small amount of charge at real-time, as the current flows through it. This explanation is perhaps crude, but it captures the essence of memory in reality: saving a little information about the present, and combining that information with previously saved information.

With this idea in mind, we may try to define a semantic interface. The interface must be powerful enough to allow the above test cases. First, let us use “*Temporal a*” as the general semantic type of temporal values (both behaviors and events). Then, what we want is a sort of a “temporal scanl”, maybe:⁵

$$\text{scanl}T :: \text{ScanFunc } a \ b \rightarrow b \rightarrow \text{Temporal } a \rightarrow \text{Temporal } b$$

Remember that we’re discussing denotations, and the type in an actual implementation may differ. The argument of type *ScanFunc a b* is a function that is specific to the kind of transformation we want to perform on the temporal value argument. *Temporal a* may be semantically equivalent to $\text{Time} \rightarrow a$, but I suggest we postpone the discussion of the denotation of *Temporal*. Instead, let’s try to figure out the *interface*, as manifest in the precise type and meaning of *scanlT*. Meanwhile, for *Temporal a* we’ll use the functional denotation $\text{Time} \rightarrow a$, under the assumption that the final denotation is more limited, not more permissive.

³I am deliberately ignoring any of the exotic properties of nature suggested by modern physics. The model I want to explore should be based on our normal experience with nature.

⁴Another possible test case to consider is time scaling. Is that something we really want to allow? Time “squeezing” is not causal, and “stretching” seems to require infinite memory.

⁵Reactive’s (see (Elliott 2009b)) function, *scanlE*, has a similar type; so do functions from other existing FRP frameworks.

My initial intuition was that the solution lies in “infinitesimal time delays”, that if we allow access to the “most recent” past, we’ll be able to perform “memory-full” calculations. You can probably imagine my hands waving wildly as I wrote that last sentence. A first attempt to formalize that statement was:

$$\text{“Infinitesimal past” of } f \text{ at } t = \lim_{dt \rightarrow 0} f(t - dt)$$

A somewhat obvious fault with this definition is that for continuous valued functions, the infinitesimal past is always equal to the present. In fact that is exactly a common definition for continuous functions:

$$f \text{ is continuous} \Leftrightarrow \forall x : f(x) = \lim_{dx \rightarrow 0} f(x - dx)$$

As such, the above attempt is useless.⁶ On the other hand, for non-continuous functions things are different. These thoughts lead to my next attempt at defining a proper notion of operations with finite memory.

3 Divide and Conquer

Temporal values don’t *have* to change whenever time changes. Nor are they limited to the range \mathbb{R} of real numbers, or even to types that are infinite at all. A temporal value may even be binary, switching between two values. To facilitate the discussion, consider the following definitions:

Definition 1 (Continuous time) *If a function f is defined over an interval $[a, b] \subset \mathbb{R}$, it is called a function of continuous time for that interval.*⁷

Definition 2 (Discrete time) *If for every finite interval $[a, b]$ a function f is defined at a finite number of points $t \in [a, b]$ and otherwise undefined, it is called a function of discrete time.*

The two classes of functions above are mutually exclusive, but not complementary. There are functions that don’t fit in any of the two definitions. For example, a function of rational time ($t \in \mathbb{Q}$). I’d like to limit the discussion to functions that belong in either the continuous-time or discrete-time classes. We *could* have defined discrete-time to mean a countable domain, but I’m pretty sure that the more limited definition presented above is (a) sufficiently general, and (b) easier to work with.

The above definitions categorize functions according to their domain, time. An orthogonal categorization concerns the range - the type of the function’s result. The criterion I want to focus on is whether or not we can say that the temporal value is a “step function”. Stated more precisely:

Definition 3 (Discrete value) *A function $\text{Time} \rightarrow \mathbf{a}$ that is piecewise constant (a step function) is called a function of discrete value. Piecewise constancy means that for every interval $\mathbf{t} \in [a, b]$ the function changes its value a finite number of times.*

Similarly we may talk about having discrete value in an interval (piecewise constant in that interval), and of functions that are piecewise discrete and non-discrete valued (piecewise constant in some intervals but not in others).⁸

Now we can divide the world of temporal values, $\text{Time} \rightarrow \mathbf{a}$, into four territories:

1. Discrete time, discrete value
2. Discrete time, non-discrete value
3. Continuous time, discrete value
4. Continuous time, non-discrete value

This division may help us tackle the issue of how to represent finite memory systems with no arbitrary access in time. Our first step deals with functions of discrete-time, corresponding to the union of classes 1 and 2 above.

⁶For the less mathematically inclined (such as myself), continuity also has a more general topological definition, which is probably more appropriate here in the context of arbitrary types \mathbf{a} in $\text{Time} \rightarrow \mathbf{a}$.

⁷It is possible that a more general type than \mathbb{R} can be used here.

⁸It may be possible to find types \mathbf{a} such that *every* function of type $\text{Time} \rightarrow \mathbf{a}$ is of discrete value. The characterization of such types has something to do with topology and the notion of a totally disconnected space.

3.1 Discrete time

Any function $\text{Time} \rightarrow \mathbf{a}$ of discrete time (Definition 2) is equivalent to a countable set of pairs of the type $(\text{Time}, \mathbf{a})$, such that for every interval $[t_0, t_1] \subset \text{Time}$ there are a finite number of pairs $(t, x) : t \in [t_0, t_1], x :: \mathbf{a}$. The discreteness of the range does not alter this representation, and therefore does not matter for our discussion. It is due to this independence on the value's type that we unify the 1st and 2nd of the four classes above. In this case $\text{scanl}T$ (the “temporal scanl”) can be simply scanl on a time-ordered sequence of the pairs. Recall the type of scanl :

```

scanl :: (a -> b -> b) -> b -> [a] -> [b]
-- so the suggestion is, using the fact that our “Temporal a” is discrete-timed,
-- and therefore Temporal a = [(Time, a)]:
newtype TP a = (Time, a)
scanlT :: (TP a -> TP b -> TP b) -> TP b -> [TP a] -> [TP b]
-- a semantic implementation:
scanlT = scanl

```

$\text{scanl}T$ is a primitive of our semantic framework. It is used by passing a function, an initial result and a time-discrete temporal value. The function that we pass never has access to more than one time point at once. Thus, our interface for transforming temporal values of discrete time does not allow arbitrary access in time: it only allows access to the “current” time, plus access to the operation's result from the last defined time point, which serves as the memory. There is no access to the temporal value at any time but the present.

3.2 Continuous time, discrete value

Once we enter the domain of continuous time, we can no longer meaningfully discuss “the last defined time point”. We have shown how our attempted definition of infinitesimal time delay becomes problematic, but the problem was based on the continuity property. We are now discussing temporal values that have discrete value (Definition 3), and therefore continuity is impossible. Our temporal values are step functions, where the value suddenly changes at some points in time. We can then construct a list of those steps, and pair each new value with the time of the step. In this fashion we again end up with an ordered list of pairs $(\text{Time}, \mathbf{a})$. Therefore the definition of $\text{scanl}T$ used in the discrete-time case above can be used here too, except that here we work on the list of steps rather than the list of defined time points.

3.3 Continuous time, non-discrete value

Finally, we have to deal with temporal values of continuous time and non-discrete value. This means that in a finite time interval the temporal value may change at an infinite number of time points. How can we define $\text{scanl}T$ in a way that makes sense in the case of non-discrete changes in value? So far we've managed by choosing $\text{scanl}T$ to take a function of type $(\text{Time}, \mathbf{a}) \rightarrow (\text{Time}, \mathbf{b}) \rightarrow (\text{Time}, \mathbf{b})$, and run that function through a list of pairs, $[(\text{Time}, \mathbf{a})]$. With non-discrete value, however, between every two time points there is another in which the function may change its value. It may be possible to define a list of step values as before, but it will be “infinitely dense” and the time value of one pair will be the same as the time value of the next pair.

How can we deal with this infinitely confusing sequence?

4 The Generalized Memory Accumulator

To answer the question, let's take a look at the usual tool used to tackle infinitely small values: the limit. Consider a sequence of $(\text{Time}, \mathbf{a})$ pairs where the time interval between successive pairs goes, in the limit, to zero. Using the denotation $\text{Temporal } a = \text{Time} \rightarrow \mathbf{a}$, the value of a temporal value u at time t is simply $u\ t$ (or $u(t)$ in the usual mathematical notation). The sequence is then:

$$\begin{aligned}
\{u_n\}_{\Delta t} &= \{\dots, (t_n, u\ t_n), (t_{n+1}, u\ t_{n+1}), (t_{n+2}, u\ t_{n+2}), \dots\} \\
\text{where } &\dots \leq t_n \leq t_{n+1} \leq t_{n+2} \leq \dots \\
&\text{and } \Delta t \geq \max_n (t_{n+1} - t_n)
\end{aligned}$$

For convenience, we shall denote the sequence $\{u_n\}_{\Delta t}$ as a function of Δt and a temporal value:

$$\text{sampleList} :: dt \rightarrow \text{Temporal } a \rightarrow [(Time, a)]$$

Now, for non-discrete temporal values of continuous time, the *scanlT* function can be defined as before except that we use the limit $\Delta t \rightarrow 0$ on the result of applying *scanlT* to the sequence (rather than trying to find the “stepping points”). The difference between this and the previous cases is that we end up evaluating the temporal value at *all* time points, regardless of whether the value actually changes in their vicinity at all or not. Since this case demands special treatment, we’ll use the name *scanlT'* for the continuous-time, non-discrete value scanning function:

$$\text{scanlT}' f r ta = \lim_{\Delta t \rightarrow 0} \text{scanlT } f r (\text{sampleList } \Delta t ta)$$

Where *scanlT* is as defined before, basically a specialization of *scanl* for lists of pairs $(Time, a)$. The limit we have just defined must exist and converge for the meaning of *scanlT'* to make any sense. The existence and convergence of the limit depends on the given function *f*, but also on the nature of the temporal value *ta*. To make things simpler, let us assume that the value changes “smoothly”, so that at sufficient “magnification” of the time axis the value doesn’t change much (this issue will be made a little more precise in Section 5.2). This assumption ensures that as Δt approaches zero, the sequence *sampleList* $\Delta t ta$ becomes “close” to the continuous function (for example with a squared-mean error measure).

Now we have a candidate for an operator that “scans over” continuous time, but we still can’t be sure about the type of functions it can scan (the type of the first argument to *scanlT*). Let us check our test cases, starting with differentiation. Differentiation should be easiest to implement, because it’s local - it does not require memory of the past (except the immediate past). To begin, recall one common definition of the derivative:

$$\frac{df}{dt}(t) = \lim_{\Delta t \rightarrow 0} \frac{f(t) - f(t - \Delta t)}{\Delta t}$$

Notice that the function inside the limit expression requires knowledge of the time delta. The limit expression also makes use of *two* values of the function *f*: one at the current time, *t*, and a second at *t - Δt*. The usage of these values by the differentiation operator suggests that we too should allow access to them in our scanning function. The values are Δt , $f(t)$ and $f(t - \Delta t)$. Otherwise we will have no way to implement differentiation and similar operations.

An issue we have ignored when previously defining *scanlT* is that by allowing *f* to return the type *TP b* we essentially allow writing to arbitrary times in the output temporal value.⁹ Allowing arbitrary access in time goes directly against our goals, so let’s change also this aspect of *f*’s type. Instead of taking and returning *TP b* we’ll change it to *b*. That way, the scanning operation can use the input to calculate the output, but it can’t control the time of each output: the time will be “real time” (i.e. the same time of the current instantaneous input).

Consequently, a better type for *scanlT* is (we only change the type of the first argument):

$$\text{scanlT} :: (TP a \rightarrow TP a \rightarrow b \rightarrow b) \rightarrow TP b \rightarrow [TP a] \rightarrow [TP b]$$

Where as before, $TP a = (Time, a)$. The function *f* given to *scanlT* takes the following arguments:

1. (t, x_t) ,
2. $(t - \Delta t, x_{t-\Delta t})$, and
3. the result of applying *f* to the previous two values.

The function returns the result value that matches the time *t*, the value that the output of *scanlT* will have at *t*. With the new type it’s still not hard to “implement” *scanlT*:

$$\begin{aligned} \text{uncurry2} &:: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow ((a, b) \rightarrow c \rightarrow d) \\ &-- \\ \text{scanlT} &:: (TP a \rightarrow TP a \rightarrow b \rightarrow b) \rightarrow TP b \rightarrow [TP a] \rightarrow [TP b] \end{aligned}$$

⁹Consider also a case where the function tries to “rewrite history” or the future by outputting two different values for the same time.

$scanlT f b xs = scanl f' b xs'$

where

$f' (t2, y2) = ((,) t2 \circ uncurry2 f) (t2, y2)$

$xs' = zip (tail xs) xs$

$scanlT'$ should also have its type updated. The type is identical to $scanlT$: the only difference between $scanlT'$ and $scanlT$ is that the former is a limit on the latter, when applied to the *sampleList* of the temporal value.

As a side note, I'd like to mention that passing explicit time values (e.g. in the pair (t, x_t)) is also not realistic. There is no way to measure absolute time in reality. The concept of absolute time is itself questionable. Instead, we should perhaps pass only Δt and a pair of values, in place of two time-value pairs. Nevertheless, for now we'll leave the definition as it is.

Now, for differentiation we can write:

$differentiate :: Fractional a \Rightarrow Temporal a \rightarrow Temporal a$

$differentiate = scanlT' pdiff 0$

where $pdiff (t2, y2) (t1, y1) = (y2 - y1) / (t2 - t1)$

and for integration:

$integrate :: Fractional a \Rightarrow Temporal a \rightarrow Temporal a$

$integrate = scanlT' psum 0$

where $psum (t2, y2) (t1, y1) s = s + (y2 + y1) / 2 * (t2 - t1)$

Proof that $differentiate$ is inverse of $integrate$, by substituting $pdiff$'s output into $psum$'s input:

$$\left[\left(s + \frac{y(t + \Delta t) + y(t)}{2} \Delta t \right) - s \right] \frac{1}{\Delta t} = \frac{y(t + \Delta t) + y(t)}{2} \xrightarrow{\Delta t \rightarrow 0} y(t)$$

Another test case - maximum:

$maximum :: Ord a \Rightarrow Temporal a \rightarrow Temporal a$

$maximum = scanlT' pmax MinBound$

where $pmax (t2, y2) m = max m y2$

5 Food for thought

5.1 Time delay

Time delay is problematic. We could allow the function passed into $scanlT$ to output the time value of each output it produces (output a time-value pair instead of just a value), but that opens the door to setting values at arbitrary times - exactly the problem we avoided by disallowing outputting the time. Also, time delay seems to require infinite memory for the continuous-time case. Despite these problems, time delay seems to be a valid operation - after all, it is both linear and time-invariant. Time delay can be performed via convolution with an impulse (dirac delta) placed at a non-zero time, and we have already shown that convolution is possible (because intergration is possible). However, the impulse is not really a function and thus the convolution integral in this case is no more (or less) than mathematical trickery. The best we can do is model the impulse as a limit on a constant energy signal, one which in the limit is concentrated at one time point. We end up using two limits - the integration limit and another for the convolving impulse function - something our model does not support.

Thus, because of its requirement for infinite memory and for the ability to write to arbitrary times, and despite being a linear, time-invariant operation, time delay is not necessarily something we want to allow within our model. One consequence of this conclusion is that not all operations that are both linear and time-invariant are allowed in our model, despite belonging to a very limited class of operations.

Yet, in some intuitive sense, time delay seems like a realistic operation, one that we experience. We must once again turn to reality for ideas. How does time delay work in the real world? Apparently delays are due to reality having not only time, but also *space*. In reality, we may delay information by transmitting

it to a different location, an operation that must take time.¹⁰ FRP in general, and our proposed model in particular has no equivalent concept. This difference between our model and reality hints that we may be missing something. At the time of writing, this was an open question.

5.2 Band-limited signals

We made an effort to deal with the continuous-time, non-discrete-value case. A point to consider is whether for some temporal values, sampling is sufficient. Recall Nyquist’s theorem, which states that every signal (a function of time, specifically $\mathbb{R} \rightarrow \mathbb{R}$) whose Fourier transform is zero outside some finite interval, can be sampled and later reconstructed exactly. The condition is that the sampling rate be greater than twice the highest frequency in the signal. Signals satisfying the aforementioned condition are said to be *band-limited*, and all the signals (temporal values) we normally deal with are band-limited. Therefore, apparently we can simply sample our continuous-time temporal values at some sufficiently high sampling frequency, and the samples will contain (and indeed do contain) all the necessary information about the signal, in a discrete-time equivalent form. However, ideal reconstruction¹¹ from samples to continuous-time signals is *not* causal! This means that we can’t meaningfully say that Nyquist sampling represents exactly the original signal, if future samples are not all known.

5.3 Computability and continuous functions

As explained in (Bauer 2006), computable functions are all continuous. Continuous functions can be approximated to arbitrary precision, as the precision of the function’s argument approaches infinite precision. On the other hand, non-continuous functions do not have this property. Consider a step function such as $\text{sign}(x)$, which is -1 for $x < 0$ and 1 at $x \geq 0$. To know the value of the function in the neighborhood of 0 even at the “rough” precision of just knowing whether the result is closer to -1 or to 1 , we need to know the argument at infinite precision. For example, if the argument’s digits are $-0.000000\dots$, we need to know “till the end” whether the number is really just *zero*, or whether there’s a 1 somewhere that makes it a negative number. Thus, to know anything at all about the function’s value around zero we may need infinite information about the exact location. How, if at all, should this fact alter our model? Do we need to assume that all temporal values are continuous (eliminating the class of “step functions” of continuous-time completely)?

6 Conclusion

We have discussed the following points:

- What should (or does) FRP mean? The point is apparently denotational design of a framework for temporal systems.
- The idea that FRP should follow the realistic limit on arbitrary access in time.
- An attempt to define an operation that makes it possible to compute various temporal functions with memory, without using access to values at arbitrary time points.

Finally we have mentioned a few open questions (for me at least) relating to the main discussion. It is my hope that this short report will inspire more rigorous, deeper insights into FRP by the readers.

References

Andrej Bauer. Sometimes all functions are continuous. Blog post, March 2006. URL <http://math.andrej.com/2006/03/27/sometimes-all-functions-are-continuous/>.

Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell’03)*, page 718, Uppsala, Sweden, August 2003. ACM Press.

¹⁰Due to the relativistic limitation of finite velocity, but also on the simpler level of us never experiencing delay-less transmission on a day-to-day basis.

¹¹Shannon interpolation, if sampling is uniformly spaced

- Conal Elliott. Garbage collecting the semantics of frp. Blog post, January 2010a. URL <http://conal.net/blog/posts/garbage-collecting-the-semantics-of-frp/>.
- Conal Elliott. Denotational design with type class morphisms (extended version). Technical Report 2009-01, LambdaPix, March 2009a. URL <http://conal.net/papers/type-class-morphisms>.
- Conal Elliott. Why program with continuous time? Blog post, January 2010b. URL <http://conal.net/blog/posts/why-program-with-continuous-time/>.
- Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
- Conal M. Elliott. Push-pull functional reactive programming. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-508-6. doi: <http://doi.acm.org/10.1145/1596638.1596643>. URL <http://conal.net/papers/push-pull-frp/>.
- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, page 159187. Springer-Verlag, 2003.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, page 5164, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.