Simply efficient functional reactivity

Conal Elliott
LambdaPix
conal@conal.net

Abstract

Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change, and reaction latency can be as high as the sampling period.

This paper presents a way to implement FRP that combines data- and demand-driven evaluation, in which values are recomputed only when necessary, and reactions are nearly instantaneous. The implementation is rooted in a new simple formulation of FRP and its semantics and so is easy to understand and reason about.

On the road to efficiency and simplicity, we'll meet some old friends (monoids, functors, applicative functors, monads, morphisms, and improving values) and make some new friends (functional future values, reactive normal form, and concurrent "unambiguous choice").

1. Introduction

Functional reactive programming (FRP) supports elegant programming of dynamic and reactive systems by providing first-class, composable abstractions for *behaviors* (time-varying values) and *events* (streams of timed values) (Elliott and Hudak 1997; Nilsson et al. 2002).[1] Behaviors can change *continuously* (not just frequently), with discretization introduced automatically during rendering. The choice of continuous time makes programs simpler and more composable than the customary (for computer programming) choice of discrete time, just as is the case with continuous space for modeled imagery. For instance, vector and 3D graphics representations are inherently scalable (resolution-independent), as compared to bitmaps (which are spatially discrete). Similarly, temporally or spatially infinite representations are more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typically used for reactive systems, such as GUIs. There are at least two strong reasons for choosing pull over push for FRP.

- Behaviors may change continuously, so the usual tactic of idling until the next input change (and then computing consequences) doesn't apply.
- Pull-based evaluation fits well with the common functional programming style of recursive traversal with parameters (time, in this case). Push-based evaluation appears at first to be an inherently imperative technique.

[1] See `http://haskell.org/haskellwiki/FRP` for more references.

Although some values change continuously, others change only at discrete moments (say in response to a button click or an object collision), while still others have periods of continuous change alternating with constancy. In all but the purely continuous case, pull-based implementations waste considerable resources, recomputing values even when they don't change. In those situations, push-based implementations can operate much more efficiently, focusing computation on updating values that actually change.

Another serious problem with the pull approach is that it imposes significant latency. The delay between the occurrence of an event and the visible result of its reaction, can be as much as the polling period (and is on average half that period). In contrast, since push-based implementations are driven by event occurrences, reactions are visible nearly instantaneously.

Is it possible to combine the benefits of push-based evaluation—efficiency and minimal latency—with those of pull-based evaluation—simplicity of functional implementation and applicability to temporal continuity? This paper demonstrates that it is indeed possible to get the best of both worlds, combining data- and demand-driven evaluation in a simple and natural way, with values being recomputed only, and immediately, when their discrete or continuous inputs change. The implementation is rooted in a new simple formulation of FRP and its semantics and so is easy to understand and reason about.

In pursuing the goal of combined simplicity and efficiency, this paper describes the following contributions:

- A new notion of *reactive values*, which is a purely discrete simplification of FRP's reactive behaviors (no continuous change). Reactive values have simple and precise denotational semantics (given) and an efficient, data-driven implementation.
- Decomposing the notion of reactive behaviors into independent discrete and continuous components, namely reactive values and (non-reactive) time functions. Recomposing these two notions and their implementations results in FRP's reactive behaviors, but now with an implementation that simply and efficiently combines push-based and pull-based evaluation. This composite representation captures a new *reactive normal form* for FRP.
- Modernizing the FRP interface, by restructuring of much of its functionality and semantic definitions around standard type classes, as monoids, functors, applicative functors, and monads. This restructuring makes the interface more familiar, reduces the new interfaces to learn, and provides new expressive power. In most cases, the semantics are defined simply by choosing the semantic functions to be type class morphisms.
- A notion of composable *future values*, which embody pure values that (in many cases) cannot yet be known, and is at the heart of this new formulation of reactivity. Nearly all the functionality of future values is provided via standard type classes, with semantics defined as class morphisms.

- Use of Warren Burton's "improving values" as a richly structured (non-flat) type for time. Events, reactive values, reactive behaviors, and future values can all be parameterized with respect to time, which can be any ordered type at all. Using improving values (over an arbitrary ordered type) for time suffices to make the semantics of future values be a practical implementation.
- A new technique for semantically determinate concurrency via an "unambiguous choice" operator, and use of this technique to provide a new implementation of improving values.

2. Functional reactive programming

FRP revolves around two composable abstractions: events and behaviors (Elliott and Hudak 1997). Because FRP is a functional paradigm, events and behaviors describe things that exist, rather than actions that have happened or are to happen (i.e., what *is*, not what *does*). Semantically, a behavior (sometimes called a "reactive behavior") is just a function of time, while an event (sometimes called an "event source") is a list of time/value pairs ("occurrences").

$$\mathbf{type}\ B_a = T \to a$$
$$\mathbf{type}\ E_a = [(\widehat{T}, a)] \quad \text{-- for non-decreasing times}$$

Historically in FRP, $T = \mathbb{R}$. As we'll see, however, the semantics of behaviors assumes only that T is totally ordered. The type $\widehat{T}$ of occurrence times is T extended with $-\infty$ and ∞.

The orginal FRP (Elliott and Hudak 1997) had a notion of events as a single value with time, which led to a somewhat awkward programming style with explicit temporal loops (tail recursions). The sequence-of-pairs formulation above, described in, e.g., (Elliott 1998a; Peterson et al. 1999) and assumed throughout this paper, hides discrete time iteration, just as behaviors hide continuous "iteration", resulting in simpler, more declarative specifications.

The semantic domains B_a and E_a correspond to the behavior and event data types, via semantic functions:

$$\begin{aligned} &at \ \ :: Behavior\ a \to B_a \\ &occs :: Event\ a \ \ \ \to E_a \end{aligned}$$

This section focuses on the semantic models underlying FRP, which are intended for ease of understanding and formal reasoning. The insights gained are used in later sections to derive new correct and efficient representations.

FRP's $Behavior$ and $Event$ types came with a collection of combinators, many of which are instances of standard type classes. To dress FRP in modern attire, this paper uses standard classes and methods wherever possible in place of names from "Classic FRP" (CFRP) as in, e.g., (Elliott 1998a; Peterson et al. 1999).

2.1 Behaviors

Perhaps the simplest behavior is $time$, corresponding to the identity function.

$$\begin{aligned} &time :: Behavior\ Time \\ &at\ time = id \end{aligned}$$

2.1.1 Functor

Functions can be "lifted" to apply to behaviors. CFRP had a family of lifting combinators:

$$\begin{aligned} lift_n &:: (a_1 \to \ldots \to a_n \to b) \\ &\quad \to (Behavior\ a_1 \to \ldots \to Behavior\ a_n \to Behavior\ b) \end{aligned}$$

Lifting is pointwise and synchronous:[2]

$$at\ (lift_n\ f\ b_1 \ldots b_n) = \lambda t \to f\ (b_1\ \text{`}at\text{`}\ t) \ldots (b_n\ \text{`}at\text{`}\ t)$$

The $Functor$ instance for behaviors captures unary lifting, with $fmap$ replacing FRP's $lift_1$.

$$fmap :: (a \to b) \to Behavior\ a \to Behavior\ b$$

The semantic domain, functions, also form a functor:

$$\begin{aligned} &\mathbf{instance}\ Functor\ ((\to)\ t)\ \mathbf{where} \\ &\quad fmap\ f\ g = f \circ g \end{aligned}$$

The meaning of $fmap$ on behaviors mimics $fmap$ on the *meaning* of behaviors:[3]

$$\begin{aligned} &\mathbf{instance_{sem}}\ Functor\ Behavior\ \mathbf{where} \\ &\quad at\ (fmap\ f\ b) = fmap\ f\ (at\ b) \\ &\qquad\qquad\qquad\quad = f \circ at\ b \end{aligned}$$

In other words, at is a *natural transformation*, or "functor morphism" (for consistency with related terminology), from $Behavior$ to B (Mac Lane 1998).

The *semantic instances* in this paper ("$\mathbf{instance_{sem}}$ ...") define and clarifies the meaning of type class instances.

2.1.2 Applicative functor

Applicative functors (AFs) are a recently explored notion (McBride and Paterson 2008). The AF interface has two methods, $pure$ and ($\circledast$) (left-associative), which correspond to the monadic operations $return$ and ap. Applicative functors are more structured (less populated) than functors and less structured (more populated) than monads.

$$\begin{aligned} &\mathbf{infixl}\ 4\ \circledast \\ &\mathbf{class}\ Functor\ f \Rightarrow Applicative\ f\ \mathbf{where} \\ &\quad pure\ \ :: a \to f\ a \\ &\quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b \end{aligned}$$

These two combinators suffice to define $liftA_2$, $liftA_3$, etc.

$$\begin{aligned} &\mathbf{infixl}\ 4\ \langle\$\rangle \\ &(\langle\$\rangle) \ \ :: Functor\ f \Rightarrow (a \to b) \to f\ a \to f\ b \\ &f \langle\$\rangle a = fmap\ f\ a \\ &liftA_2 \ :: \ Applicative\ f \Rightarrow (a \to b \to c) \\ &\qquad\quad \to f\ a \to f\ b \to f\ c \\ &liftA_2\ f\ a\ b = f \langle\$\rangle a \circledast b \\ &liftA_3 \ :: \ Applicative\ f \Rightarrow (a \to b \to c \to d) \\ &\qquad\quad \to f\ a \to f\ b \to f\ c \to f\ d \\ &liftA_3\ f\ a\ b\ c = liftA_2\ f\ a\ b \circledast c \\ &\ldots \end{aligned}$$

The left-associative ($\langle\$\rangle$) is just a synonym for $fmap$—a stylistic preference—while $liftA_2$, $liftA_3$, etc. are generalizations of the monadic combinators $liftM_2$, $liftM_3$, etc.

CFRP's $lift_0$ corresponds to $pure$, while $lift_2$, $lift_3$, etc correspond to $liftA_2$, $liftA_3$, etc., so the $Applicative$ instance replaces all of the $lift_n$.[4]

Functions, and hence B, form an applicative functor, where $pure$ and ($\circledast$) correspond to the classic K and S combinators:

[2] Haskellism: The at function here is being used in both prefix form (on the left) and infix form (on the right).

[3] Haskellism: Function application has higher (stronger) precedence than infix operators, so, e.g., $f \circ at\ b \equiv f \circ (at\ b)$.

[4] The formulation of the $lift_n$ in terms of operators corresponding to $pure$ and ($\circledast$) was noted in (Elliott 1998a, Section 2.1).

$$\textbf{instance } \mathit{Applicative}\ ((\to)\ t)\ \textbf{where}$$
$$\mathit{pure} \quad = \mathit{const}$$
$$f \circledast g = \lambda t \to (f\ t)\ (g\ t)$$

The $\mathit{Applicative}$ instance for functions leads to the semantics of the $\mathit{Behavior}$ instance of $\mathit{Applicative}$. As with $\mathit{Functor}$ above, the semantic function distributes over the class methods, i.e., at is an applicative functor morphism:

$$\textbf{instance}_{\textbf{sem}}\ \mathit{Applicative}\ \mathit{Behavior}\ \textbf{where}$$
$$\begin{aligned}\mathit{at}\ (\mathit{pure}\ a) &= \mathit{pure}\ a\\ &= \mathit{const}\ a\\ \mathit{at}\ (b_f \circledast b_x) &= \mathit{at}\ b_f \circledast \mathit{at}\ b_x\\ &= \lambda t \to (b_f\ \text{`}\mathit{at}\text{`}\ t)\ (b_x\ \text{`}\mathit{at}\text{`}\ t)\end{aligned}$$

So, given a function-valued behavior b_f and an argument-valued behavior b_x, to sample $b_f \circledast b_x$ at time t, sample b_f and b_x at t and apply one result to the other.

This $(\circledast)$ operator is the heart of FRP's concurrency model, which is determinate, synchronous, and continuous.

2.2 Events

Like behaviors, much of the event functionality can be packaged via standard type classes.

2.2.1 Monoid

Classic FRP had a never-occurring event and an operator to merge two events. Together, these combinators form a monoid, so $\emptyset$ and $(\oplus)$ (Haskell's `mempty` and `mappend`) replace the CFRP names neverE and $(.|.)$.

The event monoid differs from the list monoid in that $(\oplus)$ must preserve temporal monotonicity.

$$\textbf{instance}_{\textbf{sem}}\ \mathit{Monoid}\ \mathit{Event}\ \textbf{where}$$
$$\begin{aligned}\mathit{occs}\ \emptyset &= [\,]\\ \mathit{occs}\ (e \oplus e') &= \mathit{occs}\ e\ \text{`}\mathit{merge}\text{`}\ \mathit{occs}\ e'\end{aligned}$$

Temporal merging ensures a time-ordered result and has a left-bias in the case of simultaneity:

$$\mathit{merge} :: E_a \to E_a \to E_a$$
$$\begin{aligned}&[\,]\ \text{`}\mathit{merge}\text{`}\ \mathit{vs} = \mathit{vs}\\ &\mathit{us}\ \text{`}\mathit{merge}\text{`}\ [\,] = \mathit{us}\\ &((\hat{t}_a, a) : \mathit{ps})\ \text{`}\mathit{merge}\text{`}\ ((\hat{t}_b, b) : \mathit{qs})\\ &\quad \mid \hat{t}_a \leqslant \hat{t}_b \quad = (\hat{t}_a, a) : (\mathit{ps}\ \text{`}\mathit{merge}\text{`}\ ((\hat{t}_b, b) : \mathit{qs}))\\ &\quad \mid \mathit{otherwise} = (\hat{t}_b, b) : (((\hat{t}_a, a) : \mathit{ps})\ \text{`}\mathit{merge}\text{`}\ \mathit{qs})\end{aligned}$$

Note that occurrence lists may be infinitely long.

2.2.2 Functor

Mapping a function over an event affects just the occurrence values, leaving the times unchanged.

$$\textbf{instance}_{\textbf{sem}}\ \mathit{Functor}\ \mathit{Event}\ \textbf{where}$$
$$\mathit{occs}\ (\mathit{fmap}\ f\ e) = \mathit{map}\ (\lambda(\hat{t}_a, a) \to (\hat{t}_a, f\ a))\ (\mathit{occs}\ e)$$

2.2.3 Monad

Previous FRP definitions and implementations did not have a monad instance for events. Such an instance, however, is very useful for dynamically-generated events. For example, consider playing Asteroids and tracking collisions. Each collision can break an asteroid into more of them (or none), each of which has to be tracked for more collisions. Another example is a chat room having an *enter* event whose occurrences contain new events like *speak* (for the newly entered user).

A unit event has one occurrence, which is always available:

$$\mathit{occs}\ (\mathit{return}\ a) = [(-\infty, a)]$$

The join operation collapses an event-valued event ee:

$$\mathit{join}_E :: \mathit{Event}\ (\mathit{Event}\ a) \to \mathit{Event}\ a$$

Each occurrence of ee delivers a new event, all of which get merged together into a single event.

$$\begin{aligned}&\mathit{occs}\ (\mathit{join}_E\ \mathit{ee}) =\\ &\quad \mathit{foldr}\ \mathit{merge}\ [\,] \circ \mathit{map}\ \mathit{delayOccs} \circ \mathit{occs}\ \mathit{ee}\end{aligned}$$

$$\begin{aligned}&\mathit{delayOccs} :: (\widehat{T}, \mathit{Event}\ a) \to E_a\\ &\mathit{delayOccs}\ (\hat{t}_e, e) = [(\hat{t}_e\ \text{`}\mathit{max}\text{`}\ \hat{t}_a, a) \mid (\hat{t}_a, a) \leftarrow \mathit{occs}\ e]\end{aligned}$$

Here, $\mathit{delayOccs}$ ensures that inner events cannot occur before they are generated.

This definition of occs hides a subtle problem. If ee has infinitely many non-empty occurrences, then the foldr, if taken as an implementation, would have to compare infinitely the first occurrences of infinitely many events to see which is the earliest. However, none of the occurrences in $\mathit{delayOccs}\ (\hat{t}_e, e)$ can occur before time $\hat{t}_e$, and the $\mathit{delayOccs}$ applications are given monotonically non-decreasing times. So, only a finite prefix of the events generated from ee need be compared at a time.

2.2.4 Applicative functor

Any monad can be made into an applicative functor, by defining $\mathit{pure} = \mathit{return}$ and $(\circledast) = \mathit{ap}$. However, this $\mathit{Applicative}$ instance is unlikely to be very useful for Event. Consider function- and argument-valued events e_f and e_x. The event $e_f \circledast e_x$ would be equivalent to $e_f\ \text{`}\mathit{ap}\text{`}\ e_x$ and hence to

$$e_f \ggg\!= \lambda f \to e_x \ggg\!= \lambda x \to \mathit{return}\ (f\ x)$$

or more simply

$$e_f \ggg\!= \lambda f \to \mathit{fmap}\ f\ e_x$$

The resulting event contains occurrences for every *pair* of occurrences of e_f and e_x, i.e., $(\hat{t}_f\ \text{`}\mathit{max}\text{`}\ \hat{t}_x, f\ x)$ for each $(\hat{t}_f, f) \in \mathit{occs}\ e_f$ and $(\hat{t}_x, x) \in \mathit{occs}\ e_x$. If there are m occurrences of e_f and n occurrences of e_x, then there will $m \times n$ occurrences of $e_f \circledast e_x$. Since the maximum of two values is one value or the other, there are at most $m + n$ distinct values of $\hat{t}_f\ \text{`}\mathit{max}\text{`}\ \hat{t}_x$. Hence the $m \times n$ occurrences must all occur in at most $m + n$ temporally distinct clusters.

2.3 Combining behaviors and events

FRP's basic tool for introducing reactivity combines a behavior and and an event.

$$\begin{aligned}\mathit{switcher} ::\ &\mathit{Behavior}\ a \to \mathit{Event}\ (\mathit{Behavior}\ a)\\ &\to \mathit{Behavior}\ a\end{aligned}$$

The behavior $b_0\ \text{`}\mathit{switcher}\text{`}\ e$ acts like b_0 initially. Each occurrence of the behavior-valued event e provides a new phase of behavior to switch to. Because the phases themselves (such as b_0) may be reactive, each transition may cause the $\mathit{switcher}$ behavior to lose interest in some events and start reacting to others.

The semantics of $b_0\ \text{`}\mathit{switcher}\text{`}\ e$ chooses and samples either b_0 or the last behavior from e before a given sample time t:

$$(b_0\ \text{`}\mathit{switcher}\text{`}\ e)\ \text{`}\mathit{at}\text{`}\ t = \mathit{last}\ (b_0 : \mathit{before}\ (\mathit{occs}\ e)\ t)\ \text{`}\mathit{at}\text{`}\ t$$

$$\begin{aligned}&\mathit{before} :: E_a \to T \to [a]\\ &\mathit{before}\ \mathit{os}\ t = [a \mid (\hat{t}_a, a) \leftarrow \mathit{os}, \hat{t}_a < t]\end{aligned}$$

As a simple and common specialization, $\mathit{stepper}$ produces piecewise-constant behaviors (step functions, semantically):

$$\begin{aligned}&\mathit{stepper} :: a \to \mathit{Event}\ a \to \mathit{Behavior}\ a\\ &a_0\ \text{`}\mathit{stepper}\text{`}\ e = \mathit{pure}\ a_0\ \text{`}\mathit{switcher}\text{`}\ (\mathit{pure} \mathbin{<\$>} e)\end{aligned}$$

Hence

$$at\ (a_0\ `stepper`\ e) = \lambda t \to last\ (a_0 : before\ (occs\ e)\ t)$$

There is a subtle point in the semantics of $switcher$. Consider b_0 '$stepper$' $(e \oplus e')$. If each of e and e' has one or more occurrences at the same time, then the ones from e' will get reacted to last, and so will appear in the $switcher$ behavior.

3. From semantics to implementation

Now we have a simple and precise semantics for FRP. Refining it into an efficient implementation requires addressing the following obstacles.

- Event merging compares the two occurrence times in order to choose the earlier one: $\hat{t}_a \leqslant \hat{t}_b$. If time is a flat domain (e.g., $Double$), this comparison could not take place until *both* $\hat{t}_a$ and $\hat{t}_b$ are known. Since occurrence times are not generally known until they actually arrive, this comparison would hold up event reaction until the *later* of the two occurrences, at which time the *earlier* one would be responded to. For timely response, the comparison must complete when the earlier occurrence happens.[5] Section 4 isolates this problem in an abstraction called "future values", clarifying exactly what properties are required for a type of future times. Section 9 presents a more sophisticated representation of time that satisfies these properties and solves the comparison problem. This representation adds an expense of its own, which is removed in Sections 10 and 11.
- For each sample time t, the semantics of $switcher$ involves searching through an event for the last occurrence before t. This search becomes costlier as t increases, wasting time as well as space. While the semantics allow random time sampling, in practice, behaviors are sampled with monotonically increasing times. Section 8 introduces and exploits monotonic time for efficient sampling.
- The semantics of behaviors as functions leads to an obvious, but inefficient, demand-driven evaluation strategy, as in past FRP implementations. Section 5 introduces a *reactive normal form* for behaviors that reveals the reactive structure as a sequence of simple non-reactive phases. Wherever phases are constant (a common case), sampling happens only once per phase, driven by occurrences of relevant events, as shown in Section 8.

4. Future values

A FRP event occurrence is a "future value", or simply "future", i.e., a value and an associated time. In order to simplify the semantics and implementation of events, and to provide an abstraction that may have uses outside of FRP, let's now focus on futures. Semantically,

$$F_a = (\widehat{T}, a)$$
$$force :: Future\ a \to F_a$$

Like events and behaviors, much of the interface for future values is packaged as instances of standard type classes. Moreover, as with behaviors, the semantics of these instances are defined as type class morphisms. The process of exploring these morphisms reveals requirements for the algebraic structure of $\widehat{T}$.

4.1 Functor

The semantic domain for futures, partially applied pairing, is a functor:

$$\begin{array}{l}\textbf{instance}\ Functor\ ((,)\ t)\\ \quad \textbf{where}\ fmap\ h\ (t,a) = (t, h\ a)\end{array}$$

The semantic function, $force$, is a functor morphism:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Functor\ Future\ \textbf{where}\\ \quad force\ (fmap\ h\ u) = fmap\ h\ (force\ u)\\ \qquad\qquad\qquad\qquad = (t, h\ a)\ \textbf{where}\ (t,a) = force\ u\end{array}$$

Thus, mapping a function over a future gives a future with the same time but a transformed value.

4.2 Applicative functor

For applicative functors, the semantic instance (pairing) requires an additional constraint:

$$\begin{array}{l}\textbf{instance}\ Monoid\ t \Rightarrow Applicative\ ((,)\ t)\ \textbf{where}\\ \quad pure\ a \qquad\quad = (\emptyset, a)\\ \quad (t,f) \circledast (t',x) = (t \oplus t', f\ x)\end{array}$$

When t is a future time, what meanings do we want for $\emptyset$ and $(\oplus)$? Two future values can be combined only when both are known, so $(\oplus) = max$. Since $\emptyset$ is an identity for $(\oplus)$, it follows that $\emptyset = minBound$, and so $\widehat{T}$ must have a least element.

The $Applicative$ semantics for futures follow from these considerations, and choosing $force$ to be an applicative functor morphism:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Applicative\ Future\ \textbf{where}\\ \quad force\ (pure\ a) \qquad = pure\ a\\ \qquad\qquad\qquad\qquad\quad = (\emptyset, a)\\ \qquad\qquad\qquad\qquad\quad = (minBound, a)\\ \quad force\ (u_f \circledast u_x) = force\ u_f \circledast force\ u_x\\ \qquad\qquad\qquad\qquad\quad = (\hat{t}_f, f) \circledast (\hat{t}_x, x)\\ \qquad\qquad\qquad\qquad\quad = (\hat{t}_f \oplus \hat{t}_x, f\ x)\\ \qquad\qquad\qquad\qquad\quad = (\hat{t}_f\ `max`\ \hat{t}_x, f\ x)\\ \qquad \textbf{where}\\ \qquad\quad (\hat{t}_f, f) = force\ u_f\\ \qquad\quad (\hat{t}_x, x) = force\ u_x\end{array}$$

Now, of course these definitions of $(\oplus)$ and $\emptyset$ do not hold for arbitrary t, even for ordered types, so the pairing instance of $Applicative$ provides helpful clues about the algebraic structure of future times.

4.3 Monad

Given the $Monoid$ constraint on t, the type constructor $((,)\ t)$ is equivalent to the more familiar writer monad.

$$\begin{array}{l}\textbf{instance}\ Monoid\ t \Rightarrow Monad\ ((,)\ t)\ \textbf{where}\\ \quad return\ a \quad = (\emptyset, a)\\ \quad (\hat{t}_a, a) \gg\!= h = (\hat{t}_a \oplus \hat{t}_b, b)\\ \qquad \textbf{where}\ (\hat{t}_b, b) = h\ a\end{array}$$

Taking $force$ to be a monad morphism (Wadler 1990),

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Monad\ Future\ \textbf{where}\\ \quad force\ (return\ a) = return\ a\\ \qquad\qquad\qquad\qquad = (minBound, a)\\ \quad force\ (u \gg\!= k) = force\ u \gg\!= force \circ k\\ \qquad\qquad\qquad\qquad = (\hat{t}_a\ `max`\ \hat{t}_b, b)\\ \qquad \textbf{where}\ (\hat{t}_a, a) = force\ u\\ \qquad\qquad\quad\ (\hat{t}_b, b) = force\ (k\ a)\end{array}$$

Similarly, the $join$ operation collapses a future future into a future.

$$\begin{array}{l}join_F :: Future\ (Future\ a) \to Future\ a\\ force\ (join_F\ uu) = join\ (fmap\ force\ (force\ uu))\end{array}$$

[5] Mike Sperber noted this issue and addressed it as well Sperber (2001).

$$= (\hat{t}_u \text{ `}max\text{` } \hat{t}_a, a)$$
$$\textbf{where } (\hat{t}_u, u) = force\ uu$$
$$(\hat{t}_a, a) = force\ u$$

So, the value of the *join* is the value of the of the inner future, and the time matches the later of the outer and inner futures.

4.4 Monoid

A useful $(\oplus)$ for futures simply chooses the earlier one. Then, as an identity for $(\oplus)$, $\emptyset$ must be the future that never arrives. (So $\widehat{T}$ must have an upper bound.)

$$\begin{array}{l}
\textbf{instance}_{\textbf{sem}}\ Monoid\ (Future\ a)\ \textbf{where} \\
\quad force\ \emptyset = (maxBound, \bot) \\
\quad force\ (u_a \oplus u_b) = \textbf{if } \hat{t}_a \leqslant \hat{t}_b \textbf{ then } u_a \textbf{ else } u_b \\
\qquad \textbf{where} \\
\qquad\quad (\hat{t}_a, _) = force\ u_a \\
\qquad\quad (\hat{t}_b, _) = force\ u_b
\end{array}$$

(This definition does not correspond to the standard monoid instance on pairs, so *force is not* a monoid morphism.)

Note that this *Monoid* instance uses *maxBound* and *min*, while the *Monoid* instance on future times uses *minBound* and *max*.

4.5 Implementing futures

The semantics of futures can also be used as an implementation, if the type of future times, *FTime* (with meaning $\widehat{T}$), satisfies the properties encountered above:

- Ordered and bounded with lower and upper bounds of $-\infty$ and ∞ (i.e., before and after all sample times), respectively.
- A monoid, in which $\emptyset = -\infty$ and $(\oplus) = max$.
- To be useful, the representation must exploit *partial information* about times, so that time comparisons can complete even when one of the two times is not yet fully known.

Assuming these three properties for *FTime*, the implementation of futures is easy, with most of the functionality derived from the pairing instances above.

$$\begin{array}{l}
\textbf{newtype}\ Future\ a = Fut\ (FTime, a) \\
\quad \textbf{deriving}\ (Functor, Applicative, Monad)
\end{array}$$

A *Monoid* instance also follows directly from the semantics in Section 4.4:

$$\begin{array}{l}
\textbf{instance}\ Monoid\ (Future\ a)\ \textbf{where} \\
\quad \emptyset = Fut\ (maxBound, \bot) \\
\qquad \text{-- problematic:} \\
\quad u_a@(Fut\ (\hat{t}_a, _)) \oplus u_b@(Fut\ (\hat{t}_b, _)) = \\
\qquad \textbf{if } \hat{t}_a \leqslant \hat{t}_b \textbf{ then } u_a \textbf{ else } u_b
\end{array}$$

This definition of $(\oplus)$ has a subtle, but important, problem. Consider computing the earliest of *three* futures, $(u_a \oplus u_b) \oplus u_c$, and suppose that u_c is earliest, so that $\hat{t}_c < \hat{t}_a$ ‘*min*‘ $\hat{t}_b$. No matter what the representation of *FTime* is, the definition of $(\oplus)$ above cannot produce any information about the time of $u_a \oplus u_b$ until $\hat{t}_a \leqslant \hat{t}_b$ is determined. That test will usually be impossible until the earlier of those times arrives, i.e., until $\hat{t}_a$ ‘*min*‘ $\hat{t}_b$, which (as we've supposed) is after $\hat{t}_c$.

To solve this problem, change the definition of $(\oplus)$ on futures to immediately yield a time as the (lazily evaluated) *min* of the two future times. Because *min* yields an *FTime* instead of a boolean, it can produce partial information about its answer from partial information about its inputs.

$$\begin{array}{l}
\textbf{instance}\ Ord\ a \Rightarrow Ord\ (AddBounds\ a)\ \textbf{where} \\
\quad MinBound\ \text{`}min\text{`}\ _ = MinBound \\
\quad _\ \text{`}min\text{`}\ MinBound = MinBound \\
\quad NoBound\ a\ \text{`}min\text{`}\ NoBound\ b = NoBound\ (a\ \text{`}min\text{`}\ b) \\
\quad u\ \text{`}min\text{`}\ MaxBound = u \\
\quad MaxBound\ \text{`}min\text{`}\ v = v \\
\qquad \text{-- similarly for } (\leqslant) \text{ and } max
\end{array}$$

Figure 1. *Ord* instance for the *AddBounds* type

$$\begin{array}{l}
\quad \text{-- working definition:} \\
Fut\ (\hat{t}_a, a) \oplus Fut\ (\hat{t}_b, b) = \\
\quad Fut\ (\hat{t}_a\ \text{`}min\text{`}\ \hat{t}_b, \textbf{if } \hat{t}_a \leqslant \hat{t}_b \textbf{ then } a \textbf{ else } b)
\end{array}$$

This new definition requires two comparison-like operations instead of one. It can be further improved by adding a single operation on future times that efficiently combines *min* and $(\leqslant)$.

4.6 Future times

Each of the three required properties of *FTime* (listed in Section 4.5) can be layered onto an existing type:

$$\textbf{type}\ FTime = Max\ (AddBounds\ (Improving\ Time))$$

The *Max* wrapper adds the required monoid instance while inheriting *Ord* and *Bounded*.

$$\begin{array}{l}
\textbf{newtype}\ Max\ a = Max\ a\ \textbf{deriving}\ (Eq, Ord, Bounded) \\
\textbf{instance}\ (Ord\ a, Bounded\ a) \Rightarrow Monoid\ (Max\ a)\ \textbf{where} \\
\quad \emptyset = Max\ minBound \\
\quad Max\ a \oplus Max\ b = Max\ (a\ \text{`}max\text{`}\ b)
\end{array}$$

The *AddBounds* wrapper adds new least and greatest elements, preserving the existing ordering.

$$\begin{array}{l}
\textbf{data}\ AddBounds\ a = \\
\quad MinBound \mid NoBound\ a \mid MaxBound\ \textbf{deriving}\ Eq \\
\textbf{instance}\ Bounded\ (AddBounds\ a)\ \textbf{where} \\
\quad minBound = MinBound \\
\quad maxBound = MaxBound
\end{array}$$

For an unfortunate technical reason, *AddBounds* does *not* derive *Ord*. The semantics of Haskell's **deriving** clause does not guarantee that *min* is defined in terms of *min* on the component types. If *min* is instead defined via $(\leqslant)$ (as currently in GHC), then partial information in the type parameter a cannot get passed through *min*. For this reason, *AddBounds* has an explicit *Ord* instance, given in part in Figure 1.

The final wrapper, *Improving*, is described in Section 9. It adds partial information to times and has *min* and $(\leqslant)$ that work with partially known values.

5. Reactive normal form

FRP's behavior and event combinators are very flexible. For instance, in b_0 ‘*switcher*‘ e, the phases (b_0, ...) themselves may be reactive, either as made by *switcher*, or by *fmap* or $(<\!*\!>)$ applied to reactive behaviors. This flexibility is no trouble at all for the function-based semantics in Section 2, but how can we find our way to an efficient, data-driven implementation?

Observed over time, a reactive behavior consists of a sequence of non-reactive phases, punctuated by events. Suppose behaviors can be viewed or represented in a form that reveals this phase structure explicitly. Then monotonic behavior sampling could be implemented efficiently by stepping forward through this sequence, sampling each phase until the next one is ready. For constant phases (a common case), sampling would then be driven entirely by relevant event occurrences.

Definition: A behavior-valued expression is in *reactive normal form* (RNF) if it has the form b `switcher` e, where the lead behavior b is *non-reactive*, i.e., has no embedded *switcher* (or combinators defined via *switcher*), and the behaviors in e are also in RNF.

For instance, b can be built up from *pure*, *time*, *fmap*, and ($\circledast$). To convert arbitrary behavior expressions into RNF, one could provide equational rewrite rules that move *switcher*s out of *switcher* heads, out of *fmap*, ($\circledast$), etc, and prove the correctness of these equations from the denotational semantics in Section 2. For example,

$$\mathit{fmap}\ f\ (b\ \text{`}\mathit{switcher}\text{`}\ e) \equiv \mathit{fmap}\ f\ b\ \text{`}\mathit{switcher}\text{`}\ \mathit{fmap}\ f\ e$$

The rest of this paper follows a somewhat different path, inspired by this rewriting idea, defining an RNF-based representation.

5.1 Decoupling discrete and continuous change

FRP makes a fundamental, type-level distinction between events and behaviors, i.e., between discrete and continuous. Well, not quite. Although (reactive) behaviors are defined over continuous time, they are not necessarily continuous. For instance, a behavior that counts key-presses changes only discretely. Let's further tease apart the discrete and continuous aspects of behaviors into two separate types. Call the purely discrete part a "reactive value" and the continuous part a "time function". FRP's notion of reactive behavior decomposes neatly into these two simpler notions.

Recall from Section 1 that continuous time is one of the reasons for choosing pull-based evaluation, despite the typical inefficiency relative to push-based. As we will see, reactive values can be evaluated in push style, leaving pull for time functions. Recomposing reactive values and time functions yields an RNF representation for reactive behaviors that reveals their phase structure. The two separate evaluation strategies combine to produce an efficient and simple hybrid strategy.

5.2 Reactive values

A reactive value is like a reactive behavior but is restricted to changing discretely. Its meaning is a step function, which is fully defined by its initial value and discrete changes, with each change defined by a time and a value. Together, these changes correspond exactly to a FRP event, suggesting a simple representation:

$$\textbf{data}\ \mathit{Reactive}\ a = a\ \text{`}\mathit{Stepper}\text{`}\ \mathit{Event}\ a$$

The meaning of a reactive value is given by translation into a reactive behavior, using *stepper*:

$$\begin{aligned}
&\mathit{rat} :: \mathit{Reactive}\ a \to B_a\\
&\mathit{rat}\ (a_0\ \text{`}\mathit{Stepper}\text{`}\ e) = \mathit{at}\ (a_0\ \text{`}\mathit{stepper}\text{`}\ e)\\
&\qquad = \lambda t \to \mathit{last}\ (a_0 : \mathit{before}\ (\mathit{occs}\ e)\ t)
\end{aligned}$$

where *before* is as defined in Section 2.3.

With the exception of *time*, all behavior operations in Section 2 (as well as others not mentioned there) produce discretely-changing behaviors when given discretely-changing behaviors. Therefore, all of these operations (excluding *time*) have direct counterparts for reactive values. In addition, reactive values form a monad.

$$\begin{aligned}
&\mathit{stepper}_R :: a \to \mathit{Event}\ a \to \mathit{Reactive}\ a\\
&\mathit{switcher}_R :: \mathit{Reactive}\ a \to \mathit{Event}\ (\mathit{Reactive}\ a)\\
&\qquad \to \mathit{Reactive}\ a
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{Reactive}\\
&\textbf{instance}\ \mathit{Applicative}\ \mathit{Reactive}\\
&\textbf{instance}\ \mathit{Monad}\ \mathit{Reactive}
\end{aligned}$$

The semantic function, *rat*, is a morphism on *Functor*, *Applicative*, and *Monad*:

$$\begin{aligned}
&\textbf{instance}_{\textbf{sem}}\ \mathit{Functor}\ \mathit{Reactive}\ \textbf{where}\\
&\quad \mathit{rat}\ (\mathit{fmap}\ f\ b) = \mathit{fmap}\ f\ (\mathit{rat}\ b)\\
&\qquad = f \circ \mathit{rat}\ b\\
&\textbf{instance}_{\textbf{sem}}\ \mathit{Applicative}\ \mathit{Reactive}\ \textbf{where}\\
&\quad \mathit{rat}\ (\mathit{pure}\ a) = \mathit{pure}\ a\\
&\qquad = \mathit{const}\ a\\
&\quad \mathit{rat}\ (r_f \circledast r_x) = \mathit{rat}\ r_f \circledast \mathit{rat}\ r_x\\
&\qquad = \lambda t \to (r_f\ \text{`}\mathit{rat}\text{`}\ t)\ (r_x\ \text{`}\mathit{rat}\text{`}\ t)\\
&\textbf{instance}_{\textbf{sem}}\ \mathit{Monad}\ \mathit{Reactive}\ \textbf{where}\\
&\quad \mathit{rat}\ (\mathit{return}\ a) = \mathit{return}\ a\\
&\qquad = \mathit{const}\ a\\
&\quad \mathit{rat}\ (r \gg\!= k) = \mathit{rat}\ r \gg\!= \mathit{rat} \circ k\\
&\qquad = \lambda t \to (\mathit{rat} \circ k)\ (\mathit{rat}\ r\ t)\ t\\
&\qquad = \lambda t \to \mathit{rat}\ (k\ (\mathit{rat}\ r\ t))\ t
\end{aligned}$$

The *join* operation may be a bit easier to follow.

$$\begin{aligned}
\mathit{rat}\ (\mathit{join}_R\ \mathit{rr}) &= \mathit{join}\ (\mathit{fmap}\ \mathit{rat}\ (\mathit{rat}\ r))\\
&= \mathit{join}\ (\mathit{rat} \circ \mathit{rat}\ \mathit{rr})\\
&= \lambda t \to \mathit{rat}\ (\mathit{rat}\ \mathit{rr}\ t)\ t
\end{aligned}$$

Sampling $\mathit{join}_R\ \mathit{rr}$ at time t then amounts to sampling rr at t to get a reactive value r, which is itself sampled at t.

5.3 Time functions

Between event occurrences, a reactive behavior follows a non-reactive function of time. Such a time function is most directly and simply represented literally as a function. However, functions are opaque at run-time, preventing optimizations. Constant functions are particularly helpful to recognize, in order to perform dynamic constant propagation, as in (Elliott 1998a; Nilsson 2005). A simple data type suffices for recognizing constants.

$$\textbf{data}\ \mathit{Fun}\ t\ a = K\ a \mid \mathit{Fun}\ (t \to a)$$

The semantics is given by a function that applies a *Fun* to an argument. All other functionality can be neatly packaged, again, in instances of standard type classes, as shown in Figure 2. The semantic function, *apply*, is a morphism with respect to each of these classes.

Other optimizations could be enabled by in a similar way. For instance, generalize the K constructor to polynomials (adding a *Num* constraint for t). Such a representation could support precise and efficient differentiation and integration and prediction of some synthetic events based on root-finding (e.g., some object collisions). The opacity of the function arguments used with *fmap* and *arr* would, however, limit analysis.

5.4 Composing

Reactive values capture the purely discrete aspect of reactive behaviors, while time functions capture the purely continuous. Combining them yields a representation for reactive behaviors.

$$\textbf{type}\ \mathit{Behavior} = \mathit{Reactive} \circ \mathit{Fun}\ \mathit{Time}$$

Type composition can be defined as follows:

$$\textbf{newtype}\ (h \circ g)\ a = O\ (h\ (g\ a))$$

Functors compose into functors, and applicative functors into applicative functors (McBride and Paterson 2008).

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Functor}\ h, \mathit{Functor}\ g)\\
&\qquad \Rightarrow \mathit{Functor}\ (h \circ g)\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (O\ \mathit{hga}) = O\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ \mathit{hga})
\end{aligned}$$

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Fun}\ t\ a = K\ a \mid \mathit{Fun}\ (t \to a) \\
\mathit{apply} :: \mathit{Fun}\ t\ a \to (t \to a) \\
\mathit{apply}\ (K\ \ \ a) = \mathit{const}\ a \\
\mathit{apply}\ (\mathit{Fun}\ f) = f \\
\mathbf{instance}\ \mathit{Functor}\ (\mathit{Fun}\ t)\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ (K\ a) \quad = K\ \ (f\ a) \\
\quad \mathit{fmap}\ f\ (\mathit{Fun}\ g) = \mathit{Fun}\ (f \circ g) \\
\mathbf{instance}\ \mathit{Applicative}\ (\mathit{Fun}\ t)\ \mathbf{where} \\
\quad \mathit{pure} \qquad\qquad = K \\
\quad K\ f \circledast K\ x \quad = K\ \ (f\ x) \\
\quad \mathit{cf}\ \ \circledast \mathit{cx} \qquad = \mathit{Fun}\ (\mathit{apply}\ \mathit{cf} \circledast \mathit{apply}\ \mathit{cx}) \\
\mathbf{instance}\ \mathit{Monad}\ (\mathit{Fun}\ t)\ \mathbf{where} \\
\quad \mathit{return} \qquad\quad = \mathit{pure} \\
\quad K\ a\ \ \gg\!= h \qquad = h\ a \\
\quad \mathit{Fun}\ f \gg\!= h \quad = \mathit{Fun}\ (f \gg\!= \mathit{apply} \circ h) \\
\mathbf{instance}\ \mathit{Arrow}\ \mathit{Fun}\ \mathbf{where} \\
\quad \mathit{arr} \qquad\qquad\quad = \mathit{Fun} \\
\quad \_ \qquad\ \ggg K\ b \quad = K\ b \\
\quad K\ a\ \ \ \ggg \mathit{Fun}\ g = K\ (g\ a) \\
\quad \mathit{Fun}\ g \ggg \mathit{Fun}\ f = \mathit{Fun}\ (g \ggg f) \\
\quad \mathit{first} \qquad\qquad = \mathit{Fun} \circ \mathit{first} \quad\ \circ \mathit{apply} \\
\quad \mathit{second} \qquad\ \ = \mathit{Fun} \circ \mathit{second} \circ \mathit{apply} \\
\quad K\ a' \ast\!\!\ast\!\!\ast K\ b' \quad = K\ (a', b') \\
\quad f \qquad \ast\!\!\ast\!\!\ast\ g \qquad = \mathit{first}\ f \ggg \mathit{second}\ g
\end{array}
$$

Figure 2. Constant-optimized functions

$$
\begin{array}{l}
\mathbf{instance}\ (\mathit{Applicative}\ h, \mathit{Applicative}\ g) \\
\qquad\qquad \Rightarrow \mathit{Applicative}\ (h \circ g)\ \mathbf{where} \\
\quad \mathit{pure}\ a \qquad\qquad\ \ = O\ (\mathit{pure}\ (\mathit{pure}\ a)) \\
\quad O\ \mathit{hgf} \circledast O\ \mathit{hgx} = O\ (\mathit{liftA}_2\ (\circledast)\ \mathit{hgf}\ \mathit{hgx})
\end{array}
$$

The semantics of behaviors combines the semantics of its two components.

$$
\begin{array}{l}
\mathit{at} :: \mathit{Behavior}\ a \to B_a \\
\mathit{at}\ (O\ r_f) = \mathit{join}\ (\mathit{fmap}\ \mathit{apply}\ (\mathit{rat}\ r_f)) \\
\qquad\qquad = \lambda t \to \mathit{apply}\ (\mathit{rat}\ r_f\ t)\ t
\end{array}
$$

More explicitly,

$$O\ (f\ \text{`}\mathit{Stepper}\text{`}\ e)\ \text{`}\mathit{at}\text{`}\ t = \mathit{last}\ (f : \mathit{before}\ (\mathit{occs}\ e)\ t)\ t$$

This last form is almost identical to the semantics of $\mathit{switcher}$ in Section 2.3.

This representation of behaviors encodes reactive normal form, but how expressive is it? Are all of the $\mathit{Behavior}$ combinators covered, or do some stray outside of RNF?

The time combinator is non-reactive, i.e., purely a function of time:

$$\mathit{time} = O\ (\mathit{pure}\ (\mathit{Fun}\ \mathit{id}))$$

The $\mathit{Functor}$ and $\mathit{Applicative}$ instances are provided automatically from the instances for type composition (above), given the instances for $\mathit{Reactive}$ and Fun (specified in Section 5 and to be defined in Section 7). Straightforward but tedious calculations show that time and the $\mathit{Functor}$ and $\mathit{Applicative}$ instances have the semantics specified in Section 2.

I doubt that there is a Monad instance. While the semantic domain B is a monad, I think its join surpasses the meanings that can be represented as reactive time functions. For purely discrete applications, however, reactive behaviors can be replaced by reactive values, including the Monad functionality.

6. Another angle on events

The model of events we've been working with so far is time-ordered lists of future values, where a future value is a time/value pair: $[(t_0, a_0), (t_1, a_1), ...]$. If such an occurrence list is nonempty, another view on it is as a time t_0, together with a *reactive value* having initial value a_0 and event with occurrences $[(t_1, a_1), ...]$. If the occurrence list *is* empty, then we could consider it to have initial time ∞ ($\mathit{maxBound}$), and reactive value of $\bot$. Since a future value is a time and value, it follows that an event (empty or nonempty) has the same content as a *future reactive value*. This insight leads to a new representation of functional events:

$$
\begin{array}{l}
\quad \text{-- for non-decreasing times} \\
\mathbf{newtype}\ \mathit{Event}\ a = \mathit{Ev}\ (\mathit{Future}\ (\mathit{Reactive}\ a))
\end{array}
$$

With this representation, the semantic function on events peels off one time and value at a time.

$$
\begin{array}{l}
\mathit{occs} :: \mathit{Event}\ a \to E_a \\
\mathit{occs}\ (\mathit{Ev}\ (\mathit{Fut}\ (\infty, \_ \qquad\qquad\quad))) = [\,] \\
\mathit{occs}\ (\mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_a\ , a\ \text{`}\mathit{Stepper}\text{`}\ e'))) = (\hat{t}_a, a) : \mathit{occs}\ e'
\end{array}
$$

Why use this representation of events instead directly mimicking the semantic model E? The future-reactive representation will be convenient in defined $\mathit{Applicative}$ and Monad instances below. It also avoids a subtle problem similar to the issue of comparing future times using $(\leqslant)$, discussed in Section 4.5. The definition of merge in Section 2.2.1 determines that an event has no more occurrences by testing the list for emptiness. Consider filtering out some occurrences of an event e. Because the emptiness test yields a boolean value, it cannot yield partial information, and will have to block until the prefiltered occurrences are known and tested. These issues are also noted in Sperber (2001).

7. Implementing operations on reactive values and events

The representations of reactive values and events are now tightly interrelated:

$$
\begin{array}{l}
\mathbf{data} \qquad\ \mathit{Reactive}\ a = a\ \text{`}\mathit{Stepper}\text{`}\ \mathit{Event}\ a \\
\mathbf{newtype}\ \mathit{Event} \quad\ \ a = \mathit{Ev}\ (\mathit{Future}\ (\mathit{Reactive}\ a))
\end{array}
$$

These definitions, together with Section 5, make a convenient basis of implementing FRP.

7.1 Reactive values

7.1.1 Functor

As usual, $\mathit{fmap}\ f$ applies a function f to a reactive value pointwise, which is equivalent to applying f to the initial value and to each occurrence value.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Functor}\ \mathit{Reactive}\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ (a\ \text{`}\mathit{Stepper}\text{`}\ e) = f\ \ a\ \text{`}\mathit{Stepper}\text{`}\ \mathit{fmap}\ f\ \ e
\end{array}
$$

7.1.2 Applicative

The $\mathit{Functor}$ definition was straightforward, because the $\mathit{Stepper}$ structure is easily preserved. $\mathit{Applicative}$ is more challenging.

$$\mathbf{instance}\ \mathit{Applicative}\ \mathit{Reactive}\ \mathbf{where}\ ...$$

First the easy part. A pure value becomes reactive by using it as the initial value and $\emptyset$ as the (never-occuring) change event:

$$\mathit{pure}\ a = a\ \text{`}\mathit{Stepper}\text{`}\ \emptyset$$

Consider next applying a reactive function to a reactive argument:

$$r_f@(f \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u_f) \circledast r_x@(x \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u_x) = \\ \quad f\ x \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u \\ \qquad \mathbf{where}\ u = ...$$

The initial value is $f\ \ x$, and the change event occurs each time *either* the function or the argument changes. If the function changes first, then (at that future time) apply a new reactive function to an old reactive argument:

$$\mathit{fmap}\ (\lambda r_{f'} \to r_{f'} \circledast r_x)\ u_f$$

Similarly, if the argument changes first, apply an old reactive function and a new reactive argument:

$$\mathit{fmap}\ (\lambda r_{x'} \to r_f \circledast r_{x'})\ u_x$$

Combining these two futures as alternatives:[6]

$$u = \mathit{fmap}\ (\lambda r_{f'} \to r_{f'} \circledast r_x)\ u_f \oplus \\ \quad \mathit{fmap}\ (\lambda r_{x'} \to r_f \circledast r_{x'})\ u_x$$

More succinctly,

$$u = ((\circledast r_x) \lessdollar u_f) \oplus ((r_f \circledast) \lessdollar u_x)$$

A wonderful thing about this $(\circledast)$ definition for $\mathit{Reactive}$ is that it automatically reuses the previous value of the function or argument when the argument or function changes. This caching property is especially handy in nested applications of $(\circledast)$, which can arise either explicitly or through liftA_2, liftA_3, etc. Consider $u = \mathit{liftA}_2\ f\ r\ s$ or, equivalently, $u \equiv (f \lessdollar r) \circledast s$, where r and s are reactive values, with initial values r_0 and s_0, respectively. The initial value u_0 of u is $f\ r_0\ s_0$. If r changes from r_0 to r_1, then the new value of $f \lessdollar r$ will be $f\ r_1$, which then gets applied to s_0, i.e., $u_1 \equiv f\ r_1\ s_0$. If instead s changes from s_0 to s_1, then $u_1 \equiv f\ r_0\ s_1$. In this latter case, the old value $f\ r_0$ of $f \lessdollar r$ is passed on without having to be recomputed. The savings is significant for functions that do some work based on partial applications.

7.1.3 Monad

The Monad instance is perhaps more easily understood via its join.

$$\mathit{join}_R :: \mathit{Reactive}\ (\mathit{Reactive}\ a) \to \mathit{Reactive}\ a$$

The definition of join_R is similar to $(\circledast)$ above:

$$\mathit{join}_R\ ((a \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u_r) \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u_{rr}) = \\ \quad a \text{ `}\mathit{Stepper}\text{` } \mathit{Ev}\ u \\ \mathbf{where}\ u = ...$$

Either the inner future (u_r) or the outer future (u_{rr}) will arrive first. If the inner arrives first, switch and continue waiting for the outer:

$$(\text{`}\mathit{switcher}\text{`}\mathit{Ev}\ u_{rr}) \lessdollar u_r$$

The $(\lessdollar)$ here is over futures. If instead the outer future arrives first, abandon the inner and get new reactive values from the outer:

$$\mathit{join} \lessdollar u_{rr}$$

Choose whichever comes first:

$$u = ((\text{`}\mathit{switcher}\text{`}\mathit{Ev}\ u_{rr}) \lessdollar u_r) \oplus (\mathit{join} \lessdollar u_{rr})$$

Then plug this join into a standard Monad instance:

$$\mathbf{instance}\ \mathit{Monad}\ \mathit{Reactive}\ \mathbf{where} \\ \quad \mathit{return}\ = \mathit{pure} \\ \quad r \gg\!= h = \mathit{join}_R\ (\mathit{fmap}\ h\ r)$$

[6] Recall from Section 4.1 that $\mathit{fmap}\ f\ \ u$ arrives exactly when the future u arrives, so the $(\oplus)$'s choice in this case depends only on the relative timing of u_f and u_x.

7.1.4 Reactivity

In Section 2.3, $\mathit{stepper}$ (on behaviors) is defined via $\mathit{switcher}$. For reactive values, $\mathit{stepper}_R$ corresponds directly to the $\mathit{Stepper}$ constructor:

$$\mathit{stepper}_R :: a \to \mathit{Event}\ a \to \mathit{Reactive}\ a \\ \mathit{stepper}_R = \mathit{Stepper}$$

The more general switching form can be expressed in terms of $\mathit{stepper}_R$ and monadic join:

$$\mathit{switcher}_R\ ::\ \mathit{Reactive}\ a \to \mathit{Event}\ (\mathit{Reactive}\ a) \\ \qquad\qquad \to \mathit{Reactive}\ a \\ r \text{ `}\mathit{switcher}_R\text{` } e_r = \mathit{join}_R\ (r \text{ `}\mathit{stepper}_R\text{` } e_r)$$

7.2 Events

7.2.1 Functor

The Event functor is also easily defined. Since an event is a future reactive value, combine fmap on Future with fmap on $\mathit{Reactive}$.

$$\mathbf{instance}\ \mathit{Functor}\ \mathit{Event}\ \mathbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Ev}\ u) = \mathit{Ev}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ u)$$

7.2.2 Monad

Assuming a suitable join for events, the Monad instance is simple:

$$\mathbf{instance}\ \mathit{Monad}\ \mathit{Event}\ \mathbf{where} \\ \quad \mathit{return}\ a = \mathit{Ev}\ (\mathit{return}\ (\mathit{return}\ a)) \\ \quad r \gg\!= h\ \ = \mathit{join}_E\ (\mathit{fmap}\ h\ r)$$

This definition of return makes a regular value into an event by making a constant reactive value (return) and wrapping it up as an always-available future value (return).

The join operation collapses an event-valued event ee into an event. Each occurrence of ee delivers a new event, all of which get adjusted to insure temporal monotonicity and merged together into a single event. The event ee can have infinitely many occurrences, each of which (being an event) can also have an infinite number of occurrences. Thus join_E has the tricky task of merging (a representation of) a sorted infinite stream of sorted infinite streams into a single sorted infinite stream.

$$\mathit{join}_E :: \mathit{Event}\ (\mathit{Event}\ a) \to \mathit{Event}\ a \\ \mathit{join}_E\ (\mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_0, e \text{ `}\mathit{Stepper}\text{` } ee'))) = \\ \quad \mathit{adjustTop}_E\ \hat{t}_0\ (\mathit{adjust}_E\ \hat{t}_0\ e \oplus \mathit{join}_E\ ee')$$

$$\mathit{adjust}_E :: \mathit{FTime} \to \mathit{Event}\ a \to \mathit{Event}\ a \\ \mathit{adjust}_E\ \_\ e@(\mathit{Ev}\ (\mathit{Fut}\ (\infty, \_))) = e \\ \mathit{adjust}_E\ \hat{t}_0\ (\mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_a, a \text{ `}\mathit{Stepper}\text{` } e))) = \\ \quad \mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_1, a \text{ `}\mathit{Stepper}\text{` } \mathit{adjust}_E\ \hat{t}_1\ e)) \\ \quad \mathbf{where} \\ \qquad \hat{t}_1 = \hat{t}_0 \text{ `}\mathit{max}\text{` } \hat{t}_a$$

$$\mathit{adjustTop}_E :: \mathit{FTime} \to \mathit{Event}\ a \to \mathit{Event}\ a \\ \mathit{adjustTop}_E\ \hat{t}_0\ (\mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_a, r))) = \\ \quad \mathit{Ev}\ (\mathit{Fut}\ (\hat{t}_0 \text{ `}\mathit{max}\text{` } \hat{t}_a, r))$$

This definition of $\mathit{join}_E\ \ ee$ reaches into the future to get the first occurrence e of ee and a remainder ee' with the rest of the occurrences of ee. It then merges an adjusted e with the recursive result of join-ing ee'. The adjustment made to e ensures that its occurrences are at least as late as the occurrence containing e. (The use of max instead of a boolean comparison ensures that the adjusted time ($\hat{t}_1$) can immediately yield partial information, as discussed in Section 4.5. Unfortunately, it does interfere with the optimization of recursively adjusting only when $\hat{t}_a < \hat{t}_0$.) The entire join_E result also makes a top-level adjustment, which has no semantic effect (since those occurrences are already adjusted for times $\hat{t}_0$ and later), but enables the infinitely cascaded $(\oplus)$s to compare only finitely many first elements at a time. For instance, the outermost $(\oplus)$ is $e_0 \oplus adjustTop_E\ \hat{t}_1\ (e_1 \oplus (...))$, so occurrences of e_0 earlier than (or at) $\hat{t}_1$ do not need be compared with occurrences of e_1.

7.2.3 Monoid

The $Monoid$ instance relies on operations on futures:

$$\begin{aligned}&\textbf{instance}\ Ord\ t \Rightarrow Monoid\ (Event\ a)\ \textbf{where}\\ &\quad \emptyset \qquad\qquad\ \ = Ev\ \emptyset\\ &\quad Ev\ u \oplus Ev\ v = Ev\ (u\ \text{`}merge_u\text{`}\ v)\end{aligned}$$

The never-occuring event happens in the never-arriving future.

To merge two future reactive values u and v, there are again two possibilities. If u arrives first, with value a_0 and next future u', then a_0 will be the initial value and $u'\ \text{`}merge_u\text{`}\ v$ will be the next future. If v arrives first, with value b_0 and next future v', then b_0 will be the initial value and $u\ \text{`}merge_u\text{`}\ v'$ will be the next future.

$$\begin{aligned}&merge_u :: Future\ (Reactive\ a) \to Future\ (Reactive\ a)\\ &\qquad\quad \to Future\ (Reactive\ a)\\ &u\ \text{`}merge_u\text{`}\ v = (inFutR\ (\text{`}merge\text{`}v) \mathbin{<\$>} u) \oplus\\ &\qquad\qquad\qquad (inFutR\ (u\text{`}merge\text{`}) \mathbin{<\$>} v)\\ &\textbf{where}\\ &\quad inFutR\ f\ (r\ \text{`}Stepper\text{`}\ Ev\ u') = r\ \text{`}Stepper\text{`}\ Ev\ (f\ u')\end{aligned}$$

8. Monotonic sampling

The semantics of a behavior is a function of time. That function can be applied to time values in any order. Recall in the semantics of $switcher$ (Section 2.3) that sampling at a time t involves searching through an event for the last occurrence before t. The more occurrences take place before t, the costlier the search. Lazy evaluation can delay computing occurrences before they're used, but once computed, these occurrences would remain in the events, wasting space to hold and time to search.

In practice, behaviors are rendered forward in time, and so are sampled with monotonically increasing times. Making this usage pattern explicit allows for much more efficient sampling.

First, let's consider reactive values and events. Assume we have a consumer for generated values:

$$\textbf{type}\ Sink\ a = a \to IO\ ()$$

For instance, a sink may render a number to a GUI widget or an image to a display window. The functions $sink_R$ and $sink_E$ consume values as generated by events and reactive values:

$$\begin{aligned}&sink_R :: Sink\ a \to Reactive\ a \to IO\ b\\ &sink_E :: Sink\ a \to Event\ \quad a \to IO\ b\end{aligned}$$

The implementation is an extremely simple back-and-forth, with $sink_R$ rendering initial values and $sink_E$ waiting until the next event occurrence.

$$\begin{aligned}&sink_R\ snk\ (a\ \text{`}Stepper\text{`}\ e) \qquad\ \ = snk\ a \qquad\quad \gg sink_E\ snk\ e\\ &sink_E\ snk\ (Ev\ (Fut\ (\hat{t}_r, r))) = waitFor\ \hat{t}_r \gg sink_R\ snk\ r\end{aligned}$$

Except in the case of a predictable event (such as a timer), $waitFor\ \ \hat{t}_r$ blocks simply in evaluating the time $\hat{t}_r$ of a future event occurrence. Then when evaluation of $\hat{t}_r$ unblocks, the real time *is* (very slightly past) $\hat{t}_r$, so the actual $waitFor$ need not do any additional waiting.

A behavior contains a reactive value whose values are time functions, so it can be rendered using $sink_R$ if we can come up with a appropriate sink for time functions.

$$\begin{aligned}&sink_B :: Sink\ a \to Behavior\ a \to IO\ b\\ &sink_B\ snk\ (O\ r_f) = \textbf{do}\ snk_F \leftarrow newTFunSink\ snk\\ &\qquad\qquad\qquad\qquad\qquad\ sink_R\ snk_F\ r_f\end{aligned}$$

The procedure $newTFunSink$ makes a sink that consumes successive time functions. For each consumed constant function $K\ a$, the value a is rendered just once (with snk). When a non-constant function $Fun\ f$ is consumed, a thread is started that repeatedly samples f at the current time and renders:

$$forkIO\ (forever\ (f \mathbin{<\$>} getTime \gg\!= snk))$$

In either case, the constructed sink begins by killing the current rendering thread, if any. Many variations are possible, such as using a GUI toolkit's *idle* event instead of a thread, which has the benefit of working with thread-unsafe libraries.

9. Improving values

The effectiveness of future values, as defined in Section 4, depends on a type wrapper $Improving$, which adds partial information in the form of lower bounds. This information allows a time comparison $\hat{t}_a \leqslant \hat{t}_b$ to suceed when the earlier of $\hat{t}_a$ and $\hat{t}_b$ arrives instead of the later. It also allows $\hat{t}_a\ \text{`}min\text{`}\ \hat{t}_b$ to start producing lower bound information before *either* of $\hat{t}_a$ and $\hat{t}_b$ is known precisely.

Fortunately, exactly this notion was invented, in a more general setting, by Warren Burton. "Improving values" (Burton 1989, 1991) provide a high-level abstraction for parallel functional programming with determinate semantics.

An improving value (IV) can be represented functionally as a list of lower bounds, ending in the exact value. An IV representing a simple value (the $exactly$ function used in Section 4.6), is a singleton list (no lower bounds). See (Burton 1991, Figure 3) for details.

Of course the real value of the abstraction comes from the presence of lower bounds. Sometimes those bounds come from max, but for future times, the bounds will come to be known over time. One possible implementation of future times would involve Concurrent Haskell channels (Peyton Jones et al. 1996).

$$getChanContents :: Chan\ a \to IO\ [a]$$

The idea is to make a channel, invoke $getChanContents$, and wrap the result as an IV. Later, lower bounds and (finally) an exact value are written into the channel. When a thread attempts to look beyond the most recent lower bound, it blocks. For this reason, this simple implementation of improving values must be supplied with a steady stream of lower bounds, which in the setting of FRP correspond to event non-occurrences.

Generating and manipulating numerous lower bounds is a significant performance drawback in the purely functional implementation of IVs. A more efficient implementation, developed next, thus benefits FRP and other uses of IVs.

10. Improving on improving values

In exploring how to improve over the functional implementation of improving values, let's look at how future times are used.

- Sampling a reactive value requires comparing a sample time t with a future time $\hat{t}_{r'}$.
- Choosing the earlier of two future values ($(\oplus)$ from Section 4), uses min and $(\leqslant)$ on future times.

Imagine that we can efficiently compare an improving value with an arbitrary known (exact) value:[7]

[7] The Haskell $Ordering$ type contains LT, EQ, and GT to represent less-than, equal-to, and greater-than.

$$compare_I :: Ord\ a \Rightarrow Improving\ a \rightarrow a \rightarrow Ordering$$

How might we use $compare_I$ to compare two future times, $\hat{t}_a \leqslant \hat{t}_b$? We could either extract the exact time from $\hat{t}_a$ and compare it with $\hat{t}_b$, or extract the exact time from $\hat{t}_b$ and compare it with $\hat{t}_a$. These two methods produce the same information but usually not at the same time, so let's choose the one that can answer most promptly. If indeed $\hat{t}_a \leqslant \hat{t}_b$, then the first method will likely succeed more promptly and otherwise the second method. The dilemma in choosing is that we have to know the answer before we can choose the best method for extracting that answer.

Like many dilemmas, this one results from either/or thinking. A third alternative is to try both methods in parallel and just use whichever result arrives first. Assume for now the existence of an "unambiguous choice" operator, $unamb$, that will try two methods to solve a problem and return whichever one succeeds first. The two methods are required to agree when they both succeed, for semantic determinacy. Then

$$\begin{aligned}\hat{t}_a \leqslant \hat{t}_b = & ((\hat{t}_a \text{ `}compare_I\text{` } exact\ \hat{t}_b) \not\equiv GT) \text{ `}unamb\text{`} \\ & ((\hat{t}_b \text{ `}compare_I\text{` } exact\ \hat{t}_a) \not\equiv LT)\end{aligned}$$

Next consider $\hat{t}_a \text{`}min\text{`} \hat{t}_b$. The exact value can be extracted from the exact values of $\hat{t}_a$ and $\hat{t}_b$, or from $(\leqslant)$ on IVs:

$$\begin{aligned}exact\ (\hat{t}_a \text{ `}min\text{` } \hat{t}_b) &= exact\ \hat{t}_a \text{ `}min\text{` } exact\ \hat{t}_b \\ &= exact\ (\mathbf{if}\ (\hat{t}_a \leqslant \hat{t}_b)\ \mathbf{then}\ \hat{t}_a\ \mathbf{else}\ \hat{t}_b)\end{aligned}$$

How can we compute $(\hat{t}_a \text{`}min\text{`} \hat{t}_b) \text{`}compare_I\text{`}\, t$ for an arbitrary exact value t? The answer is $\hat{t}_a$ `$compare_I$` t if $\hat{t}_a \leqslant \hat{t}_b$, and $\hat{t}_b$ `$compare_I$` t otherwise. However, this method, by itself, misses an important opportunity. Suppose both of these tests can yield answers before it's possible to know whether $\hat{t}_a \leqslant \hat{t}_b$. If the answers agree, then we can use that answer immediately, without waiting to learn whether $\hat{t}_a \leqslant \hat{t}_b$.

With these considerations, a new representation for IVs suggests itself. Since the only two operations we need on IVs are $exact$ and $compare_I$, use those two operations *as* the IV representation. Figure 3 shows the details, with $unamb$ and $asAgree$ defined in Section 11. Combining $(\leqslant)$ and min into $minLE$ allows for a simple optimization of future $(\oplus)$ from Section 4.5.

```
-- An improving value. Invariant:
--   compare_I iv ⊒ compare (exact iv)
data Improving a =
  Imp { exact :: a, compare_I :: a → Ordering }

exactly :: Ord a ⇒ a → Improving a
exactly a = Imp a (compare a)

instance Eq a ⇒ Eq (Improving a) where
  Imp a _ ≡ Imp b _ = a ≡ b

instance Ord a ⇒ Ord (Improving a) where
  s  ⩽   t = snd (s `minLE` t)
  s `min` t = fst (s `minLE` t)
  s `max` t = fst (s `maxLE` t)

-- Efficient combination of min and (⩽)
minLE :: Ord a ⇒ Improving a → Improving a
        → (Improving a, Bool)
Imp u uComp `minLE` Imp v vComp =
  (Imp uMinV wComp, uLeqV)
 where
  uMinV = if uLeqV then u else v
    -- u ⩽ v: Try u `compare` v and v `compare` u.
  uLeqV = (uComp v ≢ GT) `unamb` (vComp u ≢ LT)
  minComp = if uLeqV then uComp else vComp
    -- (u `min` v) `compare` t: Try comparing according to
    -- whether u ⩽ v, or use either answer if they agree.
  wComp t = minComp t `unamb`
              (uComp t `asAgree` vComp t)

-- Efficient combination of max and (⩾)
maxLE :: Ord a ⇒ Improving a → Improving a
        → (Improving a, Bool)
  -- ... similarly ...
```

Figure 3. Improved improving values

11. Unambiguous choice

The representation of improving values in Section 10 relies on an "unambiguous choice" operator with determinate semantics and an underlying concurrent implementation.

```
  -- precondition: compatible arguments
unamb :: a → a → a
```

In order to preserve simple, determinate semantics, $unamb$ may only be applied to arguments that agree where defined.

$$compatible\ a\ b = (a \equiv \bot \lor b \equiv \bot \lor a \equiv b)$$

$unamb$ yields the more-defined of the two arguments.

$$\forall a\ b . compatible\ a\ b \Rightarrow unamb\ a\ b = a \sqcup b$$

Operationally, $unamb$ forks two threads and evaluates one argument in each. When one thread finishes, the other thread is killed, and the computed value is returned.

Figure 4 shows one way to implement $unamb$, in terms of an ambiguous choice operator, amb. The latter, having indeterminate (ambiguous) semantics, is in the IO type, using $race$ to run two concurrent threads. For inter-thread communication, the $race$ function uses a Concurrent Haskell MVar (Peyton Jones et al. 1996) to hold the computed value. Each thread tries to execute an action and write the resulting value into the shared MVar. The $takeMVar$ operation blocks until one of the threads succeeds, after which both threads are killed (one perhaps redundantly).[8]

The $assuming$ function makes a conditional strategy for computing a value. If the assumption is false, the conditional strategy yields $\bot$ via $hang$, which blocks a thread indefinitely, while consuming neglible resources and generating no error. One use of $assuming$ is to define $asAgree$, which was used in Figure 3.

12. Additional functionality

All of the usual FRP functionality can supported, including the following.

Integration Numeric integration requires incremental sampling for efficiency, replacing the $apply$ interface from Section 5.3 by $applyK$ from Section 8. The residual time function returned by $applyK$ remembers the previous sample time and value, so the next sampling can do a (usually) small number of integration steps. (For accuracy, it is often desirable to take more integration steps than samples.) Integration of reactive behaviors can work simply by integrating each non-reactive phase (a time function) and accumu-

[8] My thanks to Spencer Janssen for help with this implementation.

```
  -- Unambiguous choice on compatible arguments.
unamb :: a → a → a
a `unamb` b = unsafePerformIO (a `amb` b)
  -- Ambiguous choice, no precondition.
amb :: a → a → IO a
a `amb` b = evaluate a `race` evaluate b
  -- Race two actions in separate threads.
race :: IO a → IO a → IO a
race :: IO a → IO a → IO a
a `race` b =
  do v  ← newEmptyMVar
     t_a ← forkIO (a >>= putMVar v)
     t_b ← forkIO (b >>= putMVar v)
     x  ← takeMVar v
     killThread t_a
     killThread t_b
     return x
  -- Yield a value if a condition is true.
assuming :: Bool → a → a
assuming c a = if c then a else hang
  -- The value of agreeing values (or hang)
asAgree :: Eq a ⇒ a → a → a
a `asAgree` b = assuming (a ≡ b) a
  -- Never yield an answer. Identity for unamb.
hang :: a
hang = unsafePerformIO hangIO
  -- Block forever, cheaply
hangIO :: IO a
hangIO = do forever (threadDelay maxBound)
            return ⊥
```

Figure 4. Unambiguous choice implementation

lating the result, thanks the interval-additivity property of definite integration ($\int_a^c f \equiv \int_a^b f + \int_b^c f$).

Accumulation Integration is continuous accumulation on behaviors. The combinators $accum_E$ and $accum_R$ discretely accumulate the results of event occurrences.

$$accum_R :: a \rightarrow Event\ (a \rightarrow a) \rightarrow Reactive\ a$$
$$accum_E :: a \rightarrow Event\ (a \rightarrow a) \rightarrow Event\ a$$

Each occurrence of the event argument yields a function to be applied to the accumulated value.

```
a `accum_R` e = a `stepper` (a `accum_E` e)
a `accum_E` Ev u_r = Ev (h <$> u_r)
  where
    h (f `Stepper` e') = f a `accum_R` e'
```

Filtering It's often useful to filter event occurrences, keeping some occurrences and dropping others. The *Event* monad instance allows a new, simple and very general definition of that includes event filtering as a special case. One general filtering tool consumes *Maybe* values, dropping each *Nothing* and unwrapping each *Just*.[9]

[9] My thanks to Cale Gibbard for this succinct formulation.

```
joinMaybes :: MonadPlus m ⇒ m (Maybe a) → m a
joinMaybes = (>>= maybe mzero return)
```

The *MonadPlus* instance for *Event* uses $mzero = \emptyset$ and $mplus = (\oplus)$. The more common FRP event filter has the following simple generalization:

```
filter_MP :: MonadPlus m ⇒ (a → Bool) → m a → m a
filter_MP p m = joinMaybes (liftM f m)
  where
    f a | p a       = Just a
        | otherwise = Nothing
```

13. Related work

The most closely related FRP implementation is the one underlying the Lula system for design and control of lighting, by Mike Sperber (2001). Like the work described above, Lula-FRP eliminated the overhead of creating and processing the large numbers of event non-occurrences that have been present, in various guises, in almost all other FRP implementations. Mike noted that the pull-based event interface that motivates these non-occurrences also imposes a reaction latency bounded by the polling frequency, which detracts noticeably from the user experience. To eliminate non-occurrences and the resulting overhead and latency, he examined and addressed subtle issues of events and thread blocking, corresponding to the those discussed in Section 4.5. Mike's solution, like the one described in Section 10 above, involved a multi-threaded implementation. However, it did not guarantee semantic determinism, in case of simultaneous or nearly-simultaneous event occurrences. The implementation of event operations was rather complex, especially for event merging. The supporting abstractions used above (future values, improving values, and unambiguous choice) seem to be helpful in taming that complexity. Lula-FRP's behaviors still used a pure pull interface, so the latency solution was limited to direct use of events rather than reactive behaviors. The reactive value abstraction used above allows behavior reactions at much lower latency than the sampling period. Unlike most published FRP implementations, Lula-FRP was implemented in a strict language (Scheme). For that reason, it explicitly managed details of laziness left implicit in Haskell-based implementations.

"Event-Driven FRP" (E-FRP) (Wan et al. 2002) also has similar goals. It focused on event-driven systems, i.e., ones in which limited work is done in reaction to an event, while most FRP implementations repeatedly re-evaluate the whole system, whether or not there are relevant changes. Like RT-FRP (Wan et al. 2001), expressiveness is restricted in order to make guarantees about resource-bounded execution. The original FRP model of continuous time is replaced by a discrete model. Another restriction compared with the semantics of the original FRP (preserved in this paper) is that events are not allowed to occur simultaneously.

Peterson et al. (2000) explored opportunities for parallelism in implementing a variation of FRP. While the underlying semantic model was not spelled out, it seems that semantic determinacy was not preserved, in contrast to the semantically determinate concurrency used in this paper (Section 11).

Nilsson (2005) presented another approach to FRP optimization. The key idea was to recognize and efficiently handle several FRP combinator patterns. In some cases, the standard Haskell type system was inadequate to capture and exploit these patterns, but generalized algebraic data types (GADTs) were sufficient. These optimizations proved worthwhile, though they did introduce significant overhead in run-time (pattern matching) and code complexity. In contrast, the approach described in the present paper uses very simple representations and unadventurous, Hindley-Milner types. Another considerable difference is that (Nilsson 2005) uses an arrow-based formulation of FRP, as in Fruit (Courtney and Elliott 2001) and Yampa (Nilsson et al. 2002). The nature of the *Arrow* interface is problematic for the goal of minimal re-evaluation. Input events and behaviors get combined into a single input, which then changes whenever any component changes. Moreover, because the implementation style was demand-driven, event latency was still tied to sampling rate.

FranTk is a GUI library containing FRP concepts but mixing in some imperative semantics (Sage 2000). Its implementation was based on an experimental data-driven FRP implementation (Elliott 1998b), which was itself inspired by *Pidgets++* (Scholz and Bokowski 1996). Pidgets++ used functional values interactively recomputed in a data-driven manner via one-way constraints. None of these three systems supported continuous time, nor implemented a pure FRP semantics.

In some formulations of FRP, simultaneous occurrences are eliminated or merged (Nilsson et al. 2002; Wan and Hudak 2000; Wan et al. 2001), while this paper retains such occurrences as distinct. In some cases, the elimination or merging was motivated by a desire to reduce behaviors and events to a single notion. This desire is particularly compelling in the arrow-based FRP formulations, which replace behaviors (or "signals") and events with a higher level abstraction of "signal transformers". Although simultaneity is very unlikely for purely physical events, it can easily happen with FRP's *compositional* events.

14. Future work

- Much more testing, measurement, and tuning is needed in order to pragmatically and quantitatively evaluate the implementation techniques described in this paper, especially the new implementation of improving values described in Section 10. How well do the techniques work in a complex application?
- Can these ideas be transplanted to arrow-based formulations of FRP? How can changes from separately-changing inputs be kept from triggering unnecessary computation, when the arrow formulations seem to require combining all inputs into a single varying value?
- Explore other uses of the unambiguous choice operator defined in Section 11, and study its performance, including the kinds of parallel search algorithms for which improving values were invented (Burton 1989, 1991).
- Experiment with relaxing the assumption of temporal monotonicity exploited in Section 8. For instance, a zipper representation for bidirectional sampling could allow efficient access to nearby *past* event occurrences as well as future ones. Such a representation may be efficient in time though leaky in space.
- Type class morphisms are used to define the the semantics of every key type in this paper *except* for events. Can this exception be eliminated?

15. Acknowledgments

I'm grateful to Mike Sperber for the conversation that inspired the work described in this paper, as well as his help understanding Lula-FRP. Ivan Tomac provided suggestions and testing on earlier versions of the software.

References

F. Warren Burton. Indeterminate behavior with determinate semantics in parallel programs. In *International conference on Functional programming languages and computer architecture*, pages 340–346. ACM, 1989.

F. Warren Burton. Encapsulating nondeterminacy in an abstract data type with deterministic semantics. *Journal of Functional Programming*, 1(1):3–20, January 1991.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, September 2001.

Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP*, 1998a.

Conal Elliott. An imperative implementation of functional reactive animation. Unpublished draft, 1998b. URL `http://conal.net/papers/new-fran-draft.pdf`.

Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer, September 1998.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *International conference on Functional programming*, pages 54–65. ACM Press, 2005.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64. ACM Press, October 2002.

John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, 1999.

John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. *Lecture Notes in Computer Science*, 1753, 2000.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Symposium on Principles of Programming Languages*, January 1996.

Meurig Sage. FranTk – a declarative GUI language for Haskell. In *International Conference on Functional Programming*, pages 106–118. ACM, ACM Press, September 2000.

Enno Scholz and Boris Bokowski. PIDGETS++ - a C++ framework unifying postscript pictures, GUI objects, and lazy one-way constraints. In *Conference on the Technology of Object-Oriented Languages and Systems*. Prentice-Hall, 1996.

Michael Sperber. *Computer-Assisted Lighting Design and Control*. PhD thesis, University of Tübingen, June 2001.

Philip Wadler. Comprehending monads. In *Conference on LISP and Functional Programming*, pages 61–78. ACM, 1990.

Zhanyong Wan and Paul Hudak. Functional Reactive Programming from first principles. In *Conference on Programming Language Design and Implementation*, 2000.

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming*, September 2001.

Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages*, January 2002.