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). 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.

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.

1 See http://haskell.org/haskellwiki/FRP for more references.
• 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”).

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

The original 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:

\[
\text{at} :: \text{Behavior} a \rightarrow B_a \\
\text{oecs} :: \text{Event} a \rightarrow E_a
\]

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.

\[
time :: \text{Behavior} \text{Time} \\
atime \equiv \text{id}
\]

2.1.1 Functor

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

\[
\text{lift}_a :: (a_1 \rightarrow \ldots \rightarrow a_n \rightarrow b) \\
\rightarrow (\text{Behavior} a_1 \rightarrow \ldots \rightarrow \text{Behavior} a_n \rightarrow \text{Behavior} b)
\]

Lifting is pointwise and synchronous:

\[
\text{at} (\text{lift}_a f (b_1 \ldots b_n)) = \lambda t \rightarrow f (b_1 \text{`at'} t) \ldots (b_n \text{`at'} t)
\]

The Functor instance for behaviors captures unary lifting, with fmap replacing FRP’s lift$_i$.

\[
fmap :: (a \rightarrow b) \rightarrow \text{Behavior} a \rightarrow \text{Behavior} b
\]

The semantic domain, functions, also form a functor:

\[
\text{instance} \ \text{Functor} ((\rightarrow) t) \ \text{where}
\]

\[
fmap \ f \ g = f \circ g
\]

The meaning of fmap on behaviors mimics fmap on the meaning of behaviors:

\[
\text{instance}_{\text{sem}} \ \text{Functor} \ \text{Behavior} \ \text{where}
\]

\[
at (\text{fmap} f \ b) = \text{fmap} \ f \ (at \ b) = f \circ at \ b
\]

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 (“$\text{instance}_{\text{sem}} \ldots$”) 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 $\langle\langle\rangle\rangle$ (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.

\[
\text{infixl} \ 4 \langle\langle\rangle\rangle
\]

\[
\text{class} \ \text{Functor} \ f \Rightarrow \text{Applicative} \ f \ \text{where}
\]

\[
\text{pure} :: a \rightarrow f a \\
\langle\langle\rangle\rangle :: (f a \rightarrow f b) ightarrow f a \rightarrow f b
\]

These two combinators suffice to define lift$_A_2$, lift$_A_3$, etc.

\[
\text{infixl} \ 4 \langle\langle\rangle\rangle
\]

\[
\langle\langle\rangle\rangle :: \text{Functor} \ f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b \\
f \langle\langle\rangle\rangle \ a = \text{fmap} \ f \ a \\
lift_{A_2} :: \text{Applicative} \ f \Rightarrow (a \rightarrow b \rightarrow c) \\
\rightarrow f a \rightarrow f b \rightarrow f c \\
lift_{A_2} f \ a \ b = f \langle\langle\rangle\rangle \ a \langle\langle\rangle\rangle \ b \\
lift_{A_3} :: \text{Applicative} \ f \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d) \\
\rightarrow f a \rightarrow f b \rightarrow f c \rightarrow f d \\
lift_{A_3} f \ a \ b \ c = \text{lift}_2 f \ a \ b \langle\langle\rangle\rangle \ c
\]

The left-associative ($\langle\langle\rangle\rangle$) is just a synonym for fmap—a stylistic preference—while lift$_{A_2}$, lift$_{A_3}$, etc. are generalizations of the monadic combinators lift$M_2$, lift$M_3$, etc.

CFRP’s lift$_0$ corresponds to pure, while lift$_2$, lift$_3$, etc correspond to lift$_{A_2}$, lift$_{A_3}$, etc., so the Applicative instance replaces all of the lift$_n$.\footnote{Haskellism: The $at$ function here is being used in both prefix form (on the left) and infix form (on the right).}

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

\footnote{Haskellism: Function application has higher (stronger) precedence than infix operators, so, e.g., $f \circ at \ b \equiv f \circ \text{(at} \ b)$.}

\footnote{The formulation of the lift$_n$ in terms of operators corresponding to pure and ($\langle\langle\rangle\rangle$) was noted in (Elliott 1998a, Section 2.1).}
instance Applicative ((→) t) where
  pure = const
  f <<< g = λt → (f t) (g t)

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

instance Applicative Behavior where
  at (pure a) = pure a
  = const a
  at (bf <<< bs) = at bs <<< at bf
  = λt → (bs 'at' t) (bf 'at' t)

So, given a function-valued behavior by and an argument-valued behavior bs, to sample by <<< bs at time t, sample by and bs at t and apply one result to the other.

This (∪) 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 0 and (∪) (Haskell’s mempty and mappend) replace the CFRP names neverE and (∪).

The event monoid differs from the list monoid in that delayOccs hides a subtle problem. If ee is an event whose occurrences contain new events like switcher at 0, then there will be equivalent to e at 0 and hence to e = λf → ee = λx → return (f x)

or more simply

ee = λf → fmap f ee

The resulting event contains occurrences for every pair of occurrences of ee and e, i.e., ((t, f max t) ∈ ees) and (t, x) ∈ ees ee. If there are m occurrences of ee and n occurrences of ee, then there will be m × n occurrences of e at eee. Since the maximum of two values is one value or the other, there are at most m + n different values. Hence the m × 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 an event.

switcher :: Behavior a → Event (Behavior a)
  → Behavior a

The behavior b0 'switcher' e acts like b0 initially. Each occurrence of the behavior-valued event e provides a new phase of behavior to switch to. Because the phases themselves (such as b0) may be reactive, each transition may cause the switcher behavior to lose interest in some events and start reacting to others.

The semantics of b0 'switcher' e chooses and samples either b0 or the last behavior from e before a given sample time t:

(b0 'switcher' e) 'at' t = last (b0 : before (ocs e) t) 'at' t

before :: Event a → Event [a]

As a simple and common specialization, stepper produces piecewise-constant behaviors (step functions, semantically):

stepper :: a → Event a → Behavior a
a0 'stepper' e = pure a0 'switcher' (pure (∪) e)
Hence
\[
\text{at} \ (a_0 \ \text{‘stepper’} \ e) = \lambda t \to \text{last} \ (a_0 : \text{become} \ (\text{occs} \ e) \ t)
\]

There is a subtle point in the semantics of switcher. Consider
\[
b_0 \ \text{‘stepper’} \ (e \oplus e').\]
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 \text{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: \(t_a \leq t_b\). If time is a flat domain (e.g., \text{Double}), this comparison could not take place until both \(t_a\) and \(t_b\) are known. Since occurrence times are not generally known until they actually arrive, this comparison would hold up event reaction until the \textit{later} of the two occurrences, at which time the \textit{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 \text{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 \textit{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,
\[
F_a = (\hat{T}, a)
\]
\[
\text{force} :: \text{Future} \ a \to F_a
\]

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 \(\hat{T}\).

4.1 Functor

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

\[
\text{instance} \ \text{Functor} \ ((., t) \ t)
\]
\[
\text{where} \ \text{fmap} \ h \ (t, a) = (t, h \ a)
\]

The semantic function, \text{force}, is a functor morphism:

\[
\text{instance}_{\text{sem}} \ \text{Functor} \ \text{Future} \ \text{where}
\]
\[
\text{force} \ (\text{fmap} \ h \ u) = \text{fmap} \ h \ (\text{force} \ u)
\]

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:

\[
\text{instance}_{\text{sem}} \ \text{Applicative} \ \text{Future} \ \text{where}
\]
\[
\text{force} \ (\text{pure} \ a) = (\emptyset, a)
\]
\[
\text{force} \ (\text{fmap} \ h \ (t, a)) = (t \oplus f, f \ x)
\]

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 \(\emptyset \oplus \emptyset = \max\). Since \(\emptyset\) is an identity for \(\oplus\), it follows that \(\emptyset = \min\text{Bound} \), and so \(\hat{T}\) must have a least element.

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

\[
\text{instance}_{\text{sem}} \ \text{Applicative} \ \text{Future} \ \text{where}
\]
\[
\text{force} \ (\text{pure} \ a) = (\emptyset, a)
\]
\[
\text{force} \ (t, a) = (\text{minBound}, a)
\]
\[
\text{force} \ (\text{fmap} \ h \ (t, a)) = (t \oplus f, f \ x)
\]
\[
\text{where}
\]
\[
(t, f) \oplus (t', x) = (t \oplus t', f \ x)
\]

\[
(t, f) = \text{force} \ u
\]
\[
(t, x) = \text{force} \ u_a
\]

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

4.3 Monad

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

\[
\text{instance} \ \text{Monoid} \ t \Rightarrow \text{Monad} \ ((., t) \ t) \ \text{where}
\]
\[
\text{return} \ a = (\emptyset, a)
\]
\[
(t_a, a) \bowtie h = (t_a \oplus t_a, b)
\]
\[
\text{where} \ (t_b, b) = h \ a
\]

Taking \text{force} to be a monad morphism (Wadler 1990),

\[
\text{instance}_{\text{sem}} \ \text{Monad} \ \text{Future} \ \text{where}
\]
\[
\text{force} \ (\text{return} \ a) = (\text{minBound}, a)
\]
\[
(t, a) \bowtie k = (t \oplus k, \text{force} \ b)
\]
\[
\text{where} \ (t_a, a) = \text{force} \ u
\]
\[
(t_b, b) = \text{force} \ (k \ a)
\]

Similarly, the \text{join} operation collapses a future future into a future.

\[
\text{join}_F :: \text{Future} \ (\text{Future} \ a) \to \text{Future} \ a
\]
\[
\text{force} \ (\text{join}_F \ uu) = \text{join} \ (\text{fmap} \ \text{force} \ (\text{force} \ uu))
\]

\(^5\) Mike Sperber noted this issue and addressed it as well Sperber (2001).
4.4 Monoid

A useful (⊕) for futures simply chooses the earlier one. Then, as an identity for (⊕), 0 must be the future that never arrives. (So T must have an upper bound.)

\[
\begin{align*}
\text{force}_0 & = (\text{maxBound}, \bot) \\
\text{force} (u_+ \oplus u_0) & = \text{if } t_0 \leq t_1 \text{ then } u_0 \text{ else } u_1 \\
\text{where} & \quad (t_1 \ldots) = \text{force } u_1 \\
& \quad (t_0, \ldots) = \text{force } u_0
\end{align*}
\]

(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 T), satisfies the properties encountered above:

- Ordered and bounded with lower and upper bounds of −∞ and ∞ (i.e., before and after all sample times), respectively.
- A monoid, in which 0 = −∞ and (⊕) = 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.

\[
\begin{align*}
\text{newtype Future } a & = \text{Fut } (\text{FTime } a) \\
\text{deriving } & \quad (\text{Functor, Applicative, Monad})
\end{align*}
\]

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

\[
\begin{align*}
\text{instance Monoid } (\text{Future } a) & \text{ where} \\
\text{force } 0 & = (\text{maxBound}, \bot) \\
\text{force} (u_+ \oplus u_0) & = \text{if } t_0 \leq t_1 \text{ then } u_0 \text{ else } u_1 \\
\text{where} & \quad (t_1 \ldots) = \text{force } u_1 \\
& \quad (t_0, \ldots) = \text{force } u_0
\end{align*}
\]

This definition of (⊕) has a subtle, but important, problem. Consider computing the earliest of three futures, (u_2 \oplus u_0) \oplus u_c, and suppose that u_c is earliest, so that t_c \leq t_2 \text{ min } t_0. No matter what the representation of FTime is, the definition of (⊕) above cannot produce any information about the time of u_2 \oplus u_0 until t_c \leq t_0 is determined. That test will usually be impossible until the earlier of those times arrives, i.e., until t_c \text{ min } t_0, which (as we’ve supposed) is after t_c.

To solve this problem, change the definition of (⊕) 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.

4.6 Future times

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

\[
\begin{align*}
\text{type } FTime & = \text{Max } (\text{AddBounds } (\text{Improving Time})) \\
\text{The } \text{Max } & \text{ wrapper adds the required monoid instance while inheriting Ord and Bounded.} \\
\text{newtype Max } a & = \text{Max } a \text{ deriving } (\text{Eq, Ord, Bounded}) \\
\text{instance } (\text{Ord } a, \text{ Bounded } a) & \Rightarrow \text{Monoid } (\text{Max } a) \text{ where} \\
\text{∅} & = \text{Max } \text{minBound} \\
\text{Max } a \oplus \text{Max } b & = \text{Max } (a \text{ max } b)
\end{align*}
\]

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

\[
\begin{align*}
\text{data AddBounds } a & = \\
& \quad \text{MinBound } | \text{NoBound } a | \text{MaxBound deriving Eq} \\
\text{instance Bounded } (\text{AddBounds } a) \text{ where} \\
\text{minBound} & = \text{MinBound} \\
\text{maxBound} & = \text{MaxBound}
\end{align*}
\]

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 (≤) (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 (≤) 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’ c, 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, sam-
plying 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 \( b \text{ switcher} e \), where the head behavior \( b \) is non-reactive, i.e., has no embedded \text{switcher} (or combinators defined via \text{switcher}), and the behaviors in \( e \) are also in RNF.

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

For example,

\[
\text{fmap} \ f \ (b \ '\text{switcher}' \ e) \equiv \text{fmap} \ f \ b \ '\text{switcher}' \ \text{fmap} \ f \ e
\]

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 time, they are not necessarily continuous. For instance, a behavior

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:

\[
\text{data \ Reactive} \ a = a \ '\text{Stepper}' \ \text{Event} \ a
\]

The meaning of a reactive value is given by translation into a reactive behavior, using \text{stepper}:

\[
\begin{align*}
\text{rat} :: \text{Reactive} \ a & \Rightarrow B_a \\
\text{rat} \ (a_0 \ '\text{Stepper}' \ e) &= \lambda t \rightarrow \text{last} \ (a_0 : \text{before} \ (\text{accs} \ e) \ t)
\end{align*}
\]

where \text{before} is as defined in Section 2.3.

With the exception of \text{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 \text{time}) have direct counterparts for reactive values. In addition, reactive values form a monad.

\[
\begin{align*}
\text{stepper}_R & :: a \rightarrow \text{Event} \ a \rightarrow \text{Reactive} \ a \\
\text{switcher}_R & :: \text{Reactive} \ a \rightarrow \text{Event} \ (\text{Reactive} \ a) \\
& \rightarrow \text{Reactive} \ a
\end{align*}
\]

\[
\begin{align*}
\text{instance} \ \text{Functor} & \ \text{Reactive} \\
\text{instance} \ \text{Applicative} & \ \text{Reactive} \\
\text{instance} \ \text{Monad} & \ \text{Reactive}
\end{align*}
\]

The semantic function, \text{rat}, is a morphism on \text{Functor}, \text{Applicative}, and \text{Monad}:

\[
\begin{align*}
\text{instance}_{\text{Functor}} \ \text{Reactive} \ \text{where} \\
\text{rat} \ (\text{fmap} \ f \ b) &= \text{fmap} \ f \ (\text{rat} \ b) \\
&= f \circ \text{rat} \ b
\end{align*}
\]

\[
\begin{align*}
\text{instance}_{\text{Applicative}} \ \text{Reactive} \ \text{where} \\
\text{rat} \ (\text{pure} \ a) &= \text{pure} \ a \\
&= \text{const} \ a \\
\text{rat} \ (r_\llhd \llhd \ r_s) &= \lambda t \rightarrow (r_\llhd \llhd \ \text{rat} \ t) \ (r_s \ '\text{rat}' \ t)
\end{align*}
\]

\[
\begin{align*}
\text{instance}_{\text{Monad}} \ \text{Reactive} \ \text{where} \\
\text{rat} \ (\text{return} \ a) &= \text{return} \ a \\
&= \text{const} \ a \\
\text{rat} \ (r \ \llRightarrow \ k) &= \lambda t \rightarrow (\text{rat} \ k) \ (\text{rat} \ r \ t) \\
&= \lambda t \rightarrow \text{rat} \ (k \ (\text{rat} \ r \ t)) \ t
\end{align*}
\]

The \text{join} operation may be a bit easier to follow.

\[
\begin{align*}
\text{join} \ (\text{join}_R \ rr) &= \text{join} \ (\text{fmap} \ \text{rat} \ (\text{rat} \ r)) \\
&= \text{join} \ (\text{rat} \circ \text{rat} \ rr) \\
&= \lambda t \rightarrow \text{rat} \ (\text{rat} \ rr \ t) \ t
\end{align*}
\]

Sampling \text{join}_R \ 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.

\[
\text{data} \ \text{Fun} \ t \ a = K \ a | \text{Fun} \ (t \rightarrow a)
\]

The semantics is given by a function that applies a \text{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, \text{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 \text{K} constructor to polynomials (adding a \text{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 \text{fmap} and \text{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.

\[
\text{type} \ \text{Behavior} = \text{Reactive} \circ \text{Fun} \ \text{Time}
\]

Type composition can be defined as follows:

\[
\text{newtype} \ (h \circ g) \ a = O \ (h \ (g \ a))
\]

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

\[
\begin{align*}
\text{instance} \ (\text{Functor} \ h, \text{Functor} \ g) & \Rightarrow \text{Functor} \ (h \circ g) \ \text{where} \\
\text{fmap} \ f \ (O \ hga) &= O \ (\text{fmap} \ (\text{fmap} \ f) \ hga)
\end{align*}
\]
data Fun t a = K a | Fun (t → a)
apply :: Fun t a → (t → a)
apply (K a) = const a
apply (Fun f) = f

instance Functor (Fun t) where
fmap f (K a) = K (f a)
fmap f (Fun g) = Fun (f ∘ g)

instance Applicative (Fun t) where
pure = K
K f <$> K x = K (f x)
cf <$> cx = Fun (apply cf <$> apply cx)

instance Monad (Fun t) where
return = pure
K a ≫ h = h a
Fun f ≫ h = Fun (f ≫ apply o h)

instance Arrow Fun where
arr = Fun
_- ≫ K b = K b
K a ≫ Fun g = K (g a)
Fun g ≫ Fun f = Fun (g ≫ f)
first = Fun ∘ first ∘ apply
second = Fun ∘ second ∘ apply
K a' ≪ K b' = K (a', b')
f ≪ g = first f ≪ second g

Figure 2. Constant-optimized functions

instance (Applicative h, Applicative g)
⇒ Applicative (h ∘ g) where
pure a = O (pure (pure a))
O hgf <$> O hgx = O (liftA2 (<<>) hgf hgx)

The semantics of behaviors combines the semantics of its two components.
at :: Behavior a → B,
at (O r f) = join (fmap apply (rat r f))
= λt → apply (rat r f) t

More explicitly,
O (f 'Stepper' e) 'at' t = last (f : before (occs e) t) t

This last form is almost identical to the semantics of switcher in Section 2.3.

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

The time combinator is non-reactive, i.e., purely a function of time:
time = O (pure (Fun id))

The Functor and Applicative instances are provided automatically from the instances for type composition (above), given the instances for Reactive and Fun (specified in Section 5 and to be defined in Section 7). Straightforward but tedious calculations show that time and the Functor and 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: [(t₀, a₀), (t₁, a₁), ...]. If such an occurrence list is nonempty, another view on it is as a time t₀, together with a reactive value having initial value a₀ and event with occurrences [(t₁, a₁), ...]. If the occurrence list is empty, then we could consider it to have initial time ∞ (maxBound), and reactive value of ⊥. 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:

-- for non-decreasing times
newtype Event a = Ev (Future (Reactive a))

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

occs :: Event a → E a
occs (Ev (Future (maxBound a))) = []
occs (Ev (Future (Ev a))) = (Ev a) : occs e'

Why use this representation of events instead directly mimicking the semantic model E? The future-reactive representation will be convenient in defined Applicative and Monad instances below. It also avoids a subtle problem similar to the issue of comparing future times using (⩽), 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:

data Reactive a = a 'Stepper' Event a
newtype Event a = Ev (Future (Reactive a))

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

7.1 Reactive values

7.1.1 Functor

As usual, 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.

instance Functor Reactive where
fmap f (a 'Stepper' e) = f a 'Stepper' fmap f e

7.1.2 Applicative

The Functor definition was straightforward, because the Stepper structure is easily preserved. Applicative is more challenging.

instance Applicative Reactive where ...

First the easy part. A pure value becomes reactive by using it as the initial value and ∅ as the (never-occurring) change event:
pure a = a 'Stepper' ∅
Consider next applying a reactive function to a reactive argument:

\[ r_f @ (f \ 'Stepper' \ Ev \ u_f) \bowtie r_g @ (x \ 'Stepper' \ Ev \ u_x) = f \ x \ 'Stepper' \ Ev \ u \]

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:

\[ fmap (\lambda r_f \rightarrow r_f \bowtie r_g) \ u_f \]

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

\[ fmap (\lambda r_g \rightarrow r_f \bowtie r_g) \ u_x \]

Combining these two alternatives:

\[ u = fmap (\lambda r_f \rightarrow r_f \bowtie r_g) \ u_f \oplus fmap (\lambda r_g \rightarrow r_f \bowtie r_g) \ u_x \]

More succinctly,

\[ u = (\bowtie r_g) \ u_f \oplus (\bowtie r_f) \ u_x \]

A wonderful thing about this \( \bowtie \) definition for \textit{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 \( \bowtie \), which can arise either explicitly or through \textit{liftA}_2, \textit{liftA}_3, etc. Consider \( u = \textit{liftA}_2 \ f \ r \ s \), or, equivalently, \( u \equiv (f \bowtie r) \bowtie s \), 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 \( f \bowtie r \) will be \( r_1 \), which then gets applied to \( s_0 \), i.e., \( u_1 \equiv f \ r_1 \ s_0 \). 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 \( f \bowtie r \) 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 \textit{Monad} instance is perhaps merely easily understood via its \textit{join}.

\[ \textit{join}_R :: \textit{Reactive} \ (\textit{Reactive} \ a) \rightarrow \textit{Reactive} \ a \]

The definition of \textit{join}_R is similar to \( \bowtie \) above:

\[ \textit{join}_R ((a \ 'Stepper' \ Ev \ u_r) 'Stepper' \ Ev \ u_r) = a 'Stepper' \ Ev \ u \]

where \( u = ... \)

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

\[ ('\textit{switcher'} \ Ev \ u_{r'}) \bowtie u_w \]

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

\[ \textit{join} \bowtie u_{r'} \]

Choose whichever comes first:

\[ u = (('\textit{switcher'} \ Ev \ u_{r'}) \bowtie u_w) \oplus (\textit{join} \bowtie u_{r'}) \]

Then plug this \textit{join} into a standard \textit{Monad} instance:

\[ \textit{instance Monad Reactive where} \]

\[ \text{return} = \text{pure} \]

\[ r \gg h = \textit{join}_R (\textit{fmap} \ h \ r) \]

---

8 Recall from Section 4.1 that \( \textit{fmap} \ f \ u \) 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, \textit{stepper} (on behaviors) is defined via \textit{switcher}. For reactive values, \textit{stepper}_R corresponds directly to the \textit{Stepper} constructor:

\[ \textit{stepper}_R :: a \rightarrow \textit{Event} \ a \rightarrow \textit{Reactive} \ a \]

\[ \textit{stepper}_R = \textit{Stepper} \]

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

\[ \textit{switcher}_R :: \textit{Reactive} \ a \rightarrow \textit{Event} \ (\textit{Reactive} \ a) \rightarrow \textit{Reactive} \ a \]

\[ r ' \textit{switcher}_R' e_r = \textit{join}_R (r ' \textit{stepper}_R' e_r) \]

### 7.2 Events

#### 7.2.1 Functor

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

\[ \text{instance Functor Event where} \]

\[ \text{fmap} \ f \ (\text{Ev} \ u) = \text{Ev} \ (\text{fmap} \ (\text{fmap} \ f) \ u) \]

#### 7.2.2 Monad

Assuming a suitable \textit{join} for events, the \textit{Monad} instance is simple:

\[ \text{instance Monad Event where} \]

\[ \text{return} \ a = \text{Ev} \ (\text{return} \ (\text{return} \ a)) \]

\[ r \gg h = \text{join}_E \ (\text{fmap} \ h \ r) \]

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

The \textit{join} operation collapses an event-valued event \textit{ee} into an event. Each occurrence of \textit{ee} delivers a new event, all of which get adjusted to insure temporal monotonicity and merged together into a single event. The event \textit{ee} can have infinitely many occurrences, each of which (being an event) can also have an infinite number of occurrences. Thus \textit{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.

\[ \text{join}_E :: \text{Event} \ (\text{Event} \ a) \rightarrow \text{Event} \ a \]

\[ \text{join}_E \ (\text{Ev} \ (\text{Fut} \ (t_0, e ' \textit{Stepper} \ ee'))) = \text{adjustTop}_E \ t_0 \ (\text{adjust}_E \ t_0 \ e \oplus \text{join}_E \ ee') \]

\[ \text{adjust}_E :: \text{FTime} \rightarrow \text{Event} \ a \rightarrow \text{Event} \ a \]

\[ \text{adjust}_{E} \ e@((\text{Ev} \ (\text{Fut} \ (\infty, \_))) = e \]

\[ \text{adjust}_{E} \ t_0 \ (\text{Ev} \ (\text{Fut} \ (t_0, a ' \textit{Stepper} \ e'))) = \text{Ev} \ (\text{Fut} \ (t_1, a ' \textit{Stepper} \ (\text{adjust}_{E} \ t_1 \ e))) \]

where

\[ t_1 = t_0 ' \textit{max}' t_0 \]

\[ \text{adjustTop}_E :: \text{FTime} \rightarrow \text{Event} \ a \rightarrow \text{Event} \ a \]

\[ \text{adjustTop}_E \ t_0 \ (\text{Ev} \ (\text{Fut} \ (t_0, r))) = \text{Ev} \ (\text{Fut} \ (t_0 ' \textit{max}' t_0, r)) \]

This definition of \textit{join}_E \ ee reaches into the future to get the first occurrence \( e \) of \textit{ee} and a remainder \textit{ee'} with the rest of the occurrences of \textit{ee}. It then merges an adjusted \( e \) with the recursive result of \textit{join-ing} \textit{ee'}. The adjustment made to \( e \) ensures that its occurrences are at least as late as the occurrence containing \( e \). The use of \textit{max} instead of a boolean comparison ensures that the adjusted time \( t_1 \) can immediately yield partial information, as discussed in Section 4.5. Unfortunately, it does interfere with the optimization of recursively adjusting only when \( t_0 < t_0 \). The entire \textit{join}_E result also makes a top-level adjustment, which has no semantic
effect (since those occurrences are already adjusted for times $t_0$ and later), but enables the infinitely cascaded ($\oplus$) to compare only finitely many first elements at a time. For instance, the outermost ($\oplus$) is $e_0 \oplus \text{adjustToFut}_t t_1 (e_1 \oplus \ldots)$, so occurrences of $e_0$ earlier than (or at) $t_1$ do not need be compared with occurrences of $e_1$.

7.2.3 Monoid

The Monoid instance relies on operations on futures:

\[
\begin{align*}
\text{instance Ord } t & \Rightarrow \text{Monoid } (\text{Event } a) \text{ where} \\
\emptyset & = \text{Ev} \emptyset \\
\text{Ev } u \oplus \text{Ev } v & = \text{Ev} (u \cdot \text{merge}_u \cdot v)
\end{align*}
\]

The never-occurring 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 $v_0$ and next future $u'$, then $v_0$ will be the initial value and $u' \cdot \text{merge}_u \cdot v$ 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 $u \cdot \text{merge}_u \cdot v'$ will be the next future.

\[
\text{merge}_u :: \text{Future } (\text{ Reactive } a) \rightarrow \text{Future } (\text{ Reactive } a) \\
\rightarrow \text{Future } (\text{ Reactive } a) \\
u \cdot \text{merge}_u \cdot v = (\text{inFutR} (\text{merge} v) \lll u) \oplus (\text{inFutR} (u \cdot \text{merge} v) \lll v)
\]

where

\[
\text{inFutR } f (r \cdot \text{Stepper} \cdot \text{Ev } u') = r \cdot \text{Stepper} \cdot \text{Ev } (f u')
\]

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 monotonicly 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:

\[
\text{type Sink } a = a \rightarrow \text{IO } ()
\]

For instance, a sink may render a number to a GUI widget or an image to a display window. The functions \(\text{sink}_{\text{R}}\) and \(\text{sink}_{\text{E}}\) consume values as generated by events and reactive values:

\[
\begin{align*}
\text{sink}_{\text{R}} :: \text{Sink } a \rightarrow \text{Reactive } a \rightarrow \text{IO } b \\
\text{sink}_{\text{E}} :: \text{Sink } a \rightarrow \text{Event } a \rightarrow \text{IO } b
\end{align*}
\]

The implementation is an extremely simple back-and-forth, with \(\text{sink}_{\text{R}}\) rendering initial values and \(\text{sink}_{\text{E}}\) waiting until the next event occurrence.

\[
\begin{align*}
\text{sink}_{\text{R}} \ \text{snk } (a \cdot \text{Stepper} \cdot e) & = \text{snk } a \quad \Rightarrow \text{sink}_{\text{E}} \ \text{snk } e \\
\text{sink}_{\text{R}} \ \text{snk } (\text{Ev } (\text{Fut } (t_r, r))) & = \text{waitFor } t_r \quad \Rightarrow \text{sink}_{\text{R}} \ \text{snk } r
\end{align*}
\]

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

A behavior contains a reactive value whose values are time functions, so it can be rendered using \(\text{sink}_{\text{R}}\) if we can come up with a appropriate sink for time functions.

\[
\begin{align*}
\text{sink}_{\text{R}} \ \text{snk } (\text{O } r_f) & = \text{do } \text{snk}_{\text{R}} \leftarrow \text{newTFunSink } \text{snk} \\
\text{sink}_{\text{R}} \ \text{snk } r_f
\end{align*}
\]

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

\[
\text{forkIO } (\text{forever } (f \lll \text{getTime} \lll \text{do } \text{snk}))
\]

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 \text{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 \text{Improving}, which adds partial information in the form of lower bounds. This information allows a time comparison $t_a \leq t_b$ to succeed when the earlier of $t_a$ and $t_b$ arrives instead of the later. It also allows $t_a \cdot \text{min}' t_b$ to start producing lower bound information either of $t_a$ and $t_b$ 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 deterministic 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 \text{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 \text{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).

\[
\text{getChanContents :: Chan } a \rightarrow \text{IO } [a]
\]

The idea is to make a channel, invoke \text{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 $t_r$.
- Choosing the earlier of two future values ((\(\oplus\)) from Section 4), uses \text{min} and (\(\leq\)) on future times.

Imagine that we can efficiently compare an improving value with an arbitrary known (exact) value.

7 The Haskell \text{Ordering} type contains \text{LT}, \text{EQ}, and \text{GT} to represent less-than, equal-to, and greater-than.
10

Simply efficient functional reactivity

```
compare_? :: Ord a ⇒ Improving a → a → Ordering
```

How might we use `compare_?` to compare two future times, \( t_a \leq t_b \)? We could either extract the exact time from \( t_a \) and compare it with \( t_b \), or extract the exact time from \( t_b \) and compare it with \( t_a \). 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 \( t_a \leq t_b \), 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

\[
t_a \leq t_b = ((t_a \cdot \text{compare}_?) \cdot \text{exact} t_b) \neq (GT) \cdot \text{unamb}'
\]

Next consider \( t_a \cdot \text{min} \cdot t_b \). The exact value can be extracted from the exact values of \( t_a \) and \( t_b \), or from \( (\leq) \) on IVs:

\[
\text{exact} (t_a \cdot \text{min} \cdot t_b) = \text{exact} t_a \cdot \text{min'} \cdot \text{exact} t_b
\]

\[
= \text{exact} (\text{if} (t_a \leq t_b) \text{ then } t_a \text{ else } t_b)
\]

How can we compute \((t_a \cdot \text{min'} \cdot t_b) \cdot \text{compare}_? t\) for an arbitrary exact value \( t\)? The answer is \( t_a \cdot \text{compare}_? t\) if \( t_a \leq t_b \) and \( t_b \cdot \text{compare}_? t\) 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 \( t_a \leq t_b \). If the answers agree, then we can use that answer immediately, without waiting to learn whether \( t_a \leq t_b \).

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

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
\[
\text{unamb} :: a → a → a
\]

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

\[
\text{compatible} a b = (a \equiv \top \lor b \equiv \top \lor a \equiv b)
\]

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

\[
\forall a. \text{compatible} a b ⇒ \text{unamb} a b = a \lor b
\]

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 `⊥` via `hang`, which blocks a thread indefinitely, while consuming negligible resources and generating no error. One use of `assuming` is to define `as Agree`, 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.
Integration is continuous accumulation on behaviors. The combiners \( \text{accumulate} \) and \( \text{accumulate} \) discretely accumulate the results of event occurrences.

\[
\begin{align*}
    \text{accumulate} & : a \rightarrow \text{Event} (a \rightarrow a) \rightarrow \text{Reactive} a \\
    \text{accumulate} & : a \rightarrow \text{Event} (a \rightarrow a) \rightarrow \text{Event} a
\end{align*}
\]

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

\[
\begin{align*}
    \text{a 'accumulate} e & = \text{a 'steppe} (' \text{a 'accumulate} e) \\
    \text{a 'accumulate} E v w & = E v (h \triangleleft w)
\end{align*}
\]

\textit{Filtering} It’s often useful to filter event occurrences, keeping some occurrences and dropping others. The \text{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 \text{Maybe} values, dropping each \text{Nothing} and unwrapping each \text{Just}.\footnote{My thanks to Cale Gibbard for this succinct formulation.}

\begin{verbatim}
joinMaybes :: MonadPlus m => m (Maybe a) -> m a
joinMaybes = (\$=maybe mzero return)
\end{verbatim}

The \text{MonadPlus} instance for \text{Event} uses \text{mzero} = \text{0} and \text{mplus} = (\oplus). The more common FRP event filter has the following simple generalization:

\begin{verbatim}
filterMP :: MonadPlus m => (a -> Bool) -> m a -> m a
filterMP p m = joinMaybes (liftM f m)
\end{verbatim}

\textit{where} \( f a | p a = \text{Just} a \)
\( | \text{otherwise} = \text{Nothing} \)

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 1999b), 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 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


