Causality For Free!

Parametricity Implies Causality for Functional Reactive Programs

Alan Jeffrey
Alcatel-Lucent Bell Labs
ajeffrey@bell-labs.com

Abstract

Functional Reactive Programming (FRP) is a model of reactive systems in which *signals* are time-dependent values, and *signal functions* are functions between signals. Signal functions are required to be *causal*, in that output behaviour at time t is only allowed to depend on input behaviour up to time t. In order to enforce causality, many FRP libraries are *arrowized*, in that they provide combinators for building signal functions, rather than allowing users to write functions directly. In this paper, we provide a definition of *deep* causality (which coincides with the usual definition on signals of base type, but differs on nested signals). We show that FRP types can be interpreted in System F_ω extended with a kind of time, and show that in this interpretation, a "theorems for free" argument shows that parametric functions are deep causal. Since all System F_ω functions are parametric, this implies that all implementable functions are deep causal. This model is the formal basis of the agda-frp-js FRP library for the dependently typed programming language Agda, which compiles to JavaScript and executes in the browser. Assuming parametricity of Agda, this allows reactive programs to be written as regular functions over signals, without sacrificing causality. All results in this paper have been mechanically verified in Agda.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Functional Reactive Programming, Parametricity

1. Introduction

Many classes of programs are *reactive*: they run for a long period of time during which they interact with their environment. Examples of reactive programs include control systems, servers, and any program with a graphical user interface.

Many reactive programs are implemented using an event-driven model, in which stateful components send and receive events which update their state, and may cause side-effects such as network traffic or screen updates. A popular example of event-driven programming is the Document Object Model (DOM) [26] event model, with bindings to ECMAScript [11] and executed in a browser context. The event-driven model forms the basis of Actors [17], and the Model View Controller architecture of Smalltalk [7].

The event-driven model has a number of challenging features, including:

- *Concurrency*: reactive programs often have concurrent features such as dealing with multiple simultaneous events. This either leads to multithreaded languages such as Java, with complex concurrency models [6, Ch. 17], or single-threaded languages such as ECMAScript [11] which do not naturally support multicore execution, and rely on cooperative multitasking.
- *Imperative programming*: components are stateful, and may respond to events by updating their internal state. These hidden side-effects can result in complex implicit component interdependencies.
- *Referential opacity*: since components support mutable state, component identity is important. The semantics for components is not referentially transparent, since creating a component and copying it is not equivalent to creating multiple components.
- *Callbacks*: the idiom for programming in an event-driven model is registering callbacks rather than blocking function calls. For example, in ECMAScript an HTTP request is not a blocking method call, but instead a non-blocking call which registers a callback to handle the result of the HTTP request. This essentially requires the programmer to convert their program to Continuation Passing Style (CPS) [32]. Manual CPS transformation can be error-prone, for example, calling the wrong continuation, or mistakenly calling a continuation twice. In the absence of call/cc, CPS transformation is a whole-program translation, so can require a large codebase to be rewritten.

Functional Reactive Programming allows reactive programs to be written in a pure functional style. Originally developed by Elliot and Hudak [14] as part of the Fran functional animation system, there are now a number of implementations, including Flapjax [28], Frappé [9], Froc [10], FrTime [8], Grapefruit [23], Reactive [12], Reactive-Banana [2], and Yampa [41],

Comparing FRP with the event-driven model, we have:

- *Pure functional model*: there are no implicit interactions caused by shared mutable state, and a simple concurrency model.
- *Referentially transparent*: signals can be copied without altering their semantics.
- *Direct*: FRP programs are given in direct style rather than CPS.

Comparing FRP with synchronous dataflow languages such as Esterel [5], some key distinctions are:

- *Fine-grained time*: FRP often models time as a continuous domain (such as $\mathbb{R}$) or using a much finer unit of time than the sample frequency of a synchronous language (such as 1ms).

- *Higher-order signals*: FRP allows signals of signals, which model dynamically reconfigurable dataflow networks.
- *Embedded DSL*: FRP is typically implemented as an embedded DSL library in a functional host language (often Haskell, but also Agda [19], ECMAScript [28], Java [9], OCaml [10], or Scheme [8]).

The semantics of an FRP program are defined in terms of *signals*, whose semantics are given as time-indexed values[1]:

$$\mathsf{Signal}\,A \quad \approx \quad \mathsf{Time} \to A$$

For example, the current state of the mouse button could be modeled by a mouseButton signal:

$$\begin{array}{rcl} \mathsf{mouseButton} & : & \mathsf{Signal}\,(\mathsf{MouseButtonState}) \\ \mathsf{mouseButton} & \approx & t \mapsto \begin{cases} \mathsf{down} & \text{if } t \in [2,5) \\ \mathsf{up} & \text{otherwise} \end{cases} \end{array}$$

which gives rise to an event signal of mouse clicks:

$$\begin{array}{rcl} \mathsf{mouseClick} & : & \mathsf{Signal}\,(\mathsf{Maybe}\,(\mathsf{MouseEvent})) \\ \mathsf{mouseClick} & \approx & t \mapsto \begin{cases} \mathsf{just\ clicked} & \text{if } t = 5 \\ \mathsf{nothing} & \text{otherwise} \end{cases} \end{array}$$

Functional reactive programs are defined using combinators which build functions over signals. There are two approaches to defining this combinator library:

- *Classic FRP*: in classic FRP, combinators are defined as functions over signals, for example:

  $$\begin{array}{rcl} \mathsf{map} & : & (A \to B) \to \mathsf{Signal}\,A \to \mathsf{Signal}\,B \\ \mathsf{map}\,f\,\sigma & \approx & t \mapsto f(\sigma(t)) \end{array}$$

  Examples of classic FRP systems include Fran [14], Grapefruit [23], Reactive [12] and Reactive-Banana [2].
- *Arrowized FRP*: in arrowized FRP, there is no explicit Signal type; instead there is an SF $A\,B$ type for *signal functions* from A to B, whose semantics is given as functions over signals:

  $$\mathsf{SF}\,A\,B \quad \subseteq \quad \mathsf{Signal}\,A \to \mathsf{Signal}\,B$$

  The SF type is required to form an *arrow* [18] with *loops* [31], for example the equivalent of map is:

  $$\begin{array}{rcl} \mathsf{arr} & : & (A \to B) \to \mathsf{SF}\,A\,B \\ \mathsf{arr}\,f & \approx & \sigma \mapsto t \mapsto f(\sigma(t)) \end{array}$$

  The arrow combinators support a point-free style of programming based on the structure of a traced Freyd category [35] (that is, a premonoidal category [34] with a cartesian centre and a premonoidal trace [4]). The reference implementation of arrowized FRP is Yampa [41].

Note that classic FRP can be seen as an instance of arrowized FRP, the difference is whether the equation for SF is an inclusion (up to isomorphism):

$$\mathsf{SF}\,A\,B \quad \subseteq \quad \mathsf{Signal}\,A \to \mathsf{Signal}\,B$$

or an equivalence (where the inclusion is on-the-nose definitional identity)

$$\mathsf{SF}\,A\,B \quad = \quad \mathsf{Signal}\,A \to \mathsf{Signal}\,B$$

There are (at least) two reasons for introducing arrowized FRP:

- *Semantics*: Elliott [13] argues that "one source of discomfort [with classic FRP] is that this model is *mostly junk*," and "this model allows responding to future input, violating a principle sometimes called *causality*, which is that outputs may depend on the past or present but not the future." Arrowized FRP allows for a model of SF which only includes causal functions.
- *Pragmatics*: Nilsson, Courtney and Peterson [30] say "In order to ensure an efficient implementation (one that is free of time and space leaks), signals (time-varying values) are not first class entities in AFRP, unlike the signal functions operating on them. This is one of the most substantial design differences between AFRP and earlier versions of FRP, for example Fran."

There is, however, a cost associated with arrowized FRP which is that signal functions are no longer expressed as host-language functions, and instead must be programmed using the point-free combinators. Programming directly in the point-free style can be cumbersome due to explicit wiring combinators; to mitigate this, Haskell provides a DSL for dataflow programming which compiles down to the arrow combinators. Even with this DSL, the programmer is faced with the complexity of a two-layer language whose semantics is a traced Freyd category.

In this paper we address the semantic challenge of giving a junk-free treatment of classic FRP. (For a discussion of the pragmatics of avoiding time leaks in classic FRP, see [23].) We show that given an appropriate definition of Signal, *all* implementable functions are causal. To sketch our approach, we first consider the definition of causality from [30]:

> The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$.

Rephrasing this slightly, we get:

> The output of a non-interfering function at security level ℓ is uniquely determined by the input on the interval $[\bot, \ell]$.

This is the standard definition of non-interference as an information flow security property [16]. Seen in this light, we can think of causality as a security policy: the future is confidential. This suggests that techniques which have been used to enforce information flow may also work to enforce causality. In this paper, we are inspired by the work of Pierce and Sumii [39], who use *relational parametricity* to establish non-interference properties.

Relational parametricity was introduced by Reynolds [36], to support reasoning about parametric polymorphism. Wadler [40] showed that parametricity gives "theorems for free", for example, map distributes through concatenation, just from its type. In this paper, we show how relational parametricity can be used to establish causality.

Investigating the relationship between parametricity and causality highlights some features of its definition which are not completely obvious. First, consider the canonical "predict the future" function:

$$\begin{array}{rcl} \phi & : & \mathsf{Signal}\,A \to \mathsf{Signal}\,A \\ \phi\,\sigma & \approx & t \mapsto \sigma(t+1) \end{array}$$

This function is non-causal, since its output at time t depends on it input at time $t + 1$. Making this more precise, define $\sigma =_u \tau$ on signals to mean "equal up to time u":

$$\sigma =_u \tau \text{ whenever } \sigma(t) = \tau(t) \text{ for any } t \leq u$$

from which we define f to be *causal* whenever:

$$f(\sigma) =_u f(\tau) \text{ for any } \sigma =_u \tau$$

It is clear that ϕ violates causality, since (if we take Time to be $\mathbb{N}$, and write signals using list notation):

$$[0,1,2,\ldots] =_0 [0,0,0,\ldots]$$
$$\phi\,[0,1,2,\ldots] = [1,2,3,\ldots] \neq_0 [0,0,0,\ldots] = \phi\,[0,0,0,\ldots]$$

[1] Note the use of $\approx$ in the semantics of Signal, since we are only defining the type of signals up to isomorphism. As we shall see later, the definition of Signal A is more complex, but it is isomorphic to Time $\to A$.

Things become less clear when we consider higher-order signals, for example:

$$\begin{array}{rcl} \eta & : & A \to \mathsf{Signal}\,A \\ \eta\,x & \approx & t \mapsto x \end{array}$$

The signal $\eta(x)$ is just a constant signal with value x, and looks like it should be unproblematic. Unfortunately, if we take A to be a signal type, then we have:

$$\begin{array}{c} [0,1,2,\ldots] =_0 [0,0,0,\ldots] \\ \eta\,[0,1,2,\ldots] \neq_0 \eta\,[0,0,0,\ldots] \end{array}$$

where the inequality follows from:

$$\eta\,[0,1,2,\ldots]\,0 = [0,1,2,\ldots] \neq [0,0,0,\ldots] = \eta\,[0,0,0,\ldots]\,0$$

That is, according to this definition, η is not causal. This is an example of a function which is surprisingly non-causal, but there are also functions that are surprisingly causal. Consider:

$$\begin{array}{rcl} \psi & : & \mathsf{Signal}\,(\mathsf{Signal}\,A) \to \mathsf{Signal}\,A \\ \psi\,\sigma & \approx & t \mapsto \sigma(t)(t+1) \end{array}$$

This is just a variant of the "predicting the future example", but is in fact causal. If we try to replay the argument which showed ϕ to be non-causal, we have:

$$\begin{array}{l} [[0,1,2,\ldots],\ldots]\,0 \\ \quad = [0,1,2,\ldots] \\ \quad \neq [0,0,0,\ldots] \\ \quad = [[0,0,0,\ldots],\ldots]\,0 \end{array}$$

and so:

$$[[0,1,2,\ldots],\ldots] \neq_0 [[0,0,0,\ldots],\ldots]$$

which means there is no violation of causality from:

$$\psi[[0,1,2,\ldots],\ldots] = [1,\ldots] \neq_0 [0,\ldots] = \psi[[0,0,0,\ldots],\ldots]$$

These examples demonstrate both unexpected non-causality (η) and unexpected causality (ψ). In both cases, the root cause is the same. In the definition of $=_u$:

$$\sigma =_u \tau \text{ whenever } \sigma(t) = \tau(t) \text{ for any } t \leq u$$

we used $=$ as the equivalence at time t, that is this definition is a *shallow* definition of causality. An alternative definition would be to ask for *deep* causality, where (at signal type):

$$\sigma =_u \tau \text{ whenever } \sigma(t) =_u \tau(t) \text{ for any } t \leq u$$

Note that ϕ is still non-causal using this definition, but that η is deep causal, and ψ is deep non-causal. Deep causality requires $=_u$ to be defined for non-signal types, for example on base types:

$$a =_u b \text{ whenever } a = b$$

and function types:

$$f =_u g \text{ whenever } f(a) =_u g(b) \text{ for any } a =_u b$$

Readers familiar with logical relations will note that this is precisely the definition of a (non-step-indexed) logical relation. This is the heart of our result: *every parametric function is deep causal.*

The distinction between shallow and deep causality impacts an implementation as well as its semantics. A system which models shallow causality is one in which signal boundaries introduce changes of clocks. For example, a shallow causal function of type $\mathsf{Signal}\,(\mathsf{Signal}\,A) \to \mathsf{Signal}\,A$ is allowed to read a signal from its input, and run that signal in a simulated time domain to predict its future behaviour. In contrast, a system which models deep causality is one in which the same clock is shared by all signals. This paper describes the formal model of agda-frp-js [19], which uses ECMAScript's time model throughout, and so implements deep causality.

Another approach to ensuring causality is advocated by Krishnaswami and Benton [25]. Their approach gives semantics in ultrametric spaces, in particular the function space $A \to B$ is the space of *nonexpansive maps*, which are causal by definition. Our approach is different: $A \to B$ is interpreted as plain old set-theoretic functions, and we rely on an appropriate coding of signals to achieve causality by way of parametricity.

The remainder of the paper supplies the technical details for this result. The paper is structured as follows:

- Section 2 gives a recap of our prior work [21] showing that FRP programs can be regarded as proof objects in a constructive variant of LTL [33].
- Section 3 gives a recap of Girard's System F_ω, including its parametricity theorem.
- Section 4 begins the new material with a presentation of System FRP_ω, which extends System F_ω with a kind of time, a type for the order on time, and proof objects capturing that time forms a total order. We show that many of the combinators of FRP can be coded in System FRP_ω. System FRP_ω also satisfies parametricity.
- Section 5 provides the definitions of signals for System FRP_ω.
- Section 6 contains the formal statement of deep causality, together with the result that all parametric functions are causal. Since all System FRP_ω functions are parametric, this implies that any FRP program implemented in System FRP_ω is causal.
- Section 7 has a discussion of the implementation of this work in Agda, which includes a compiler to ECMAScript, and mechanized proofs of the results in this paper.

This is the first result showing that a programming language can support FRP programs with signals as first-class citizens, without sacrificing causality, while still interpreting functions set-theoretically.

2. Recap of LTL as a type system for FRP

In previous work [21] we showed that FRP programs in a dependently typed programming language can be given types in a constructive variant of *Linear-time Temporal Logic* (*LTL*) [33], such that any well-typed FRP program is a proof of an LTL tautology. The correspondence between FRP programs and LTL proofs was discovered independently by Jeltsch [24]. The use of LTL to model properties of FRP programs was also investigated by Sculthorpe and Nilsson [38].

The motivation for considering a constructive LTL as a type system for FRP is that the type $\mathsf{Signal}\,A$ models signals whose value can change over time, but whose type cannot (the value must always have type A). For example, there is no way to model a signal whose value is a timestamp at some point in the past: an attempt to do so would be $\mathsf{Signal}\,(\mathsf{Past})$, but Past is not a type, it is a time-indexed type:

$$\mathsf{Past}\,(t) \quad = \quad \{\, s \mid s \leq t \,\}$$

Since types can be thought of as propositions, time-indexed types can be thought of as propositions parametrized over time, that is temporal propositions. This leads us to consider *reactive sets*:

$$\mathsf{RSet} \quad = \quad \mathsf{Time} \to \mathsf{Set}$$

for example:

$$\mathsf{Past} \quad : \quad \mathsf{RSet}$$

Jeltsch [23] proposed using signals indexed by *era* parameters, to avoid time leaks. The type $\mathsf{SSignal}\,A\,(s)$ is the type of signals with start time s, inhabited by signals σ where $\sigma(t)$ has type A for any time $t \geq s$. In terms of reactive sets:

$$\begin{array}{rcl} \mathsf{SSignal} & : & \mathsf{Set} \to \mathsf{RSet} \\ \mathsf{SSignal}\, A\,(s) & = & \prod_{t \geq s} A \end{array}$$

If there is a minimal time 0, then signals with era parameters generalize signals, since:

$$\mathsf{Signal}\, A = \mathsf{SSignal}\, A\,(0)$$

Given a set A, we can consider the constant reactive set $\langle A \rangle$. In temporal logic terms, $\langle A \rangle$ lifts a non-temporal proposition A to a temporal proposition:

$$\begin{array}{rcl} \langle\cdot\rangle & : & \mathsf{Set} \to \mathsf{RSet} \\ \langle A \rangle\,(t) & = & A \end{array}$$

In the other direction, given a reactive set A, we can consider the set $[A]$, which is inhabited by signals σ such that $\sigma(t)$ has type $A(t)$. In temporal logic terms, a proof of $[A]$ represents a proof that A is a tautology, that is $A(t)$ is provable at all times t.

$$\begin{array}{rcl} [\cdot] & : & \mathsf{RSet} \to \mathsf{Set} \\ [A] & = & \prod_{t} A(t) \end{array}$$

Given reactive sets A and B, we can form the pointwise function space $A \Rightarrow B$. In temporal logic terms, a proof of $A \Rightarrow B$ at time t is a proof that A at time t implies B at time t, which is the usual treatment of implication in LTL:

$$\begin{array}{rcl} (\cdot \Rightarrow \cdot) & : & \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet} \\ (A \Rightarrow B)\,(t) & = & A(t) \to B(t) \end{array}$$

Given reactive set A, we can form the modal type $\Box A$, which is inhabited at time t by signals σ such that $\sigma(u)$ has type $A(u)$ for any $u \geq t$. In temporal logic terms, $\Box A$ is the "globally true" modality for the future:

$$\begin{array}{rcl} \Box & : & \mathsf{RSet} \to \mathsf{RSet} \\ \Box A\,(t) & = & \prod_{u \geq t} A(u) \end{array}$$

This modality generalizes the signal type, since:

$$\mathsf{SSignal}\, A = \Box\langle A \rangle$$

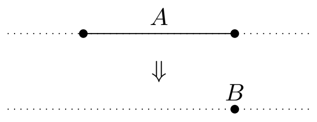
In [21], we investigated a semantics for arrowized FRP, based on the *constrains* modality of LTL [27, 29]. This modality $A \trianglerighteq B$ is inhabited at time t by functions f such that $f(\sigma)$ has type $B(u)$ whenever σ is a signal for A in the interval $[t, u]$. In temporal logic terms, this is a dual of "until" (since classically $A \trianglerighteq B$ is $\neg(A \;\underline{\mathsf{U}}\; \neg B)$) and is used to model rely/guarantee properties:

$$\begin{array}{rcl} (\cdot \trianglerighteq \cdot) & : & \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet} \\ (A \trianglerighteq B)\,(t) & = & \prod_{u \geq t} A[t, u] \to B(u) \end{array}$$

where:

$$\begin{array}{rcl} \cdot[\cdot,\cdot] & : & \mathsf{RSet} \to \mathsf{Time} \to \mathsf{Time} \to \mathsf{Set} \\ A[s, u] & = & \prod_{s \leq t \leq u} A(t) \end{array}$$

The reactive set $A \trianglerighteq B$ can be thought of as a type for history-dependent functions which, with start time t, can produce an output of type $B(u)$ at time $u \geq t$, given an input history of type $A[t, u]$:



Note that functions of this type are causal by definition. In [21], we constructed a model for arrowized FRP based on functions of type $A \trianglerighteq B$. In this paper, we show how to construct a model of classic FRP without sacrificing causality.

3. Recap of System F_ω

In this section, we introduce a kernel polymorphic functional programming language. We expect that the results would hold in a dependent setting (such as Agda) but we do not need the full power of dependent types, so we will start from Girard's System F_ω [15], a polymorphic language with higher-order kinds.

We recall the definition of System F_ω, including its syntax, type system, and denotational semantics. We also recall the definition of logical relations for System F_ω, and restate parametricity for System F_ω. In Section 4 we will extend System F_ω with a notion of time, which is a kernel of Haskell suitable for defining FRP. All results in this section have been formalized in Agda [20].

The syntax of System F_ω is presented in Figure 1, where:

- t and u range over type variables,
- x, y and z range over variables,
- K and L range over kinds, such as $(\mathsf{set} \to \mathsf{set}) \to \mathsf{set}$,
- Σ ranges over signatures of the form $t_1 : K_1, \ldots, t_m : K_m$,
- C ranges over constant types, such as $(\forall \mathsf{set})$ (which has kind $(\mathsf{set} \to \mathsf{set}) \to \mathsf{set}$),
- T and U range over types, such as the type for the identity function $(\forall \mathsf{set})(\lambda(t : \mathsf{set}) \,.\, ((\cdot \to \cdot)\, t\, t)$, which we write as $\forall(t : \mathsf{set})(t \to t)$,
- Γ ranges over contexts of the form $x_1 : T_1, \ldots, x_n : T_n$, and
- M and N range over terms, such as the polymorphic identity function $\Lambda(t : \mathsf{set}).\lambda(x : t).x$, which has type $\forall(t : \mathsf{set})(t \to t)$.

We define some shorthands:

$$\begin{array}{rcl} T \to U & = & (\cdot \to \cdot)\, T\, U \\ \forall(t : K) \,.\, T & = & (\forall K)(\lambda(t : K) \,.\, T) \end{array}$$

We will often elide the kind or type annotations from bound variables, for example writing $\forall t \,.\, T$ rather than $\forall(t : K) \,.\, T$. The notions of free variable (fv), free type variable (ftv), domain (dom), capture-avoiding substitution ($T[U/t]$) and η-β-convertibility ($=_{\eta\beta}$) are standard. The type rules for System F_ω are given in Figure 2, with judgements:

- $\Sigma \vdash \diamond$ "signature Σ is well-formed",
- $\Sigma; \Gamma \vdash \diamond$ "with respect to Σ, context Γ is well-formed", and
- $\Sigma \vdash T : K$ "with respect to Σ, type T has kind K",
- $\Sigma; \Gamma \vdash M : T$ "with respect to Σ and Γ, term M has type T".

For example. the identity function typechecks as:

$$\begin{array}{l} \vdash \forall(t : \mathsf{set})(t \to t) : \mathsf{set} \\ \vdash \Lambda(t : \mathsf{set}) \,.\, \lambda(x : t) \,.\, x : \forall(t : \mathsf{set})(t \to t) \end{array}$$

In Figure 3 and 4, we define the denotational semantics of System F_ω[2] where:

- $[\![K]\!] \in \mathsf{Set}$,
- $[\![\Sigma \vdash \diamond]\!] \in \mathsf{Set}$,
- $[\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) \in \mathsf{Set}$ where $\vec{A} \in [\![\Sigma \vdash \diamond]\!]$,

[2] In this presentation, for simplicity we allow $\mathsf{Set} \in \mathsf{Set}$. To make this presentation completely formal, it should be stratified into universes, and set should be parametrized on a universe. This is made formal in the Agda proofs of correctness [20].

$$
\begin{array}{rrcl}
\text{Kinds} & K, L & ::= & \mathsf{set} \mid K \to L \\
\text{Signatures} & \Sigma & ::= & \varepsilon \mid (\Sigma, t : K) \\
\text{Constant Types} & C & ::= & (\cdot \to \cdot) \mid (\forall K) \\
\text{Types} & T, U & ::= & C \mid \lambda(t : K) \,.\, T \mid T\,U \mid t \\
\text{Contexts} & \Gamma & ::= & \varepsilon \mid (\Gamma, x : T) \\
\text{Terms} & M, N & ::= & \lambda(x : T) \,.\, M \mid M\,N \mid x \mid \Lambda(t : K) \,.\, M \mid M\,T
\end{array}
$$

Figure 1. System F_ω syntax

$$
\frac{}{\varepsilon \vdash \diamond} \qquad \frac{\Sigma \vdash \diamond \quad t \notin \mathsf{dom}(\Sigma)}{\Sigma, t : K \vdash \diamond}
$$

$$
\begin{array}{rcl}
(\cdot \to \cdot) & : & \mathsf{set} \to \mathsf{set} \to \mathsf{set} \\
(\forall K) & : & (K \to \mathsf{set}) \to \mathsf{set}
\end{array}
$$

$$
\frac{\Sigma \vdash \diamond \quad C : K}{\Sigma \vdash C : K} \qquad \frac{\Sigma, t : K \vdash T : L}{\Sigma \vdash \lambda(t : K) \,.\, T : (K \to L)}
$$

$$
\frac{\Sigma \vdash T : K \to L \quad \Sigma \vdash U : K}{\Sigma \vdash T\,U : L} \qquad \frac{\Sigma \vdash \diamond \quad (t : K) \in \Sigma}{\Sigma \vdash t : K}
$$

$$
\frac{\Sigma \vdash \diamond}{\Sigma; \varepsilon \vdash \diamond} \qquad \frac{\Sigma; \Gamma \vdash \diamond \quad \Sigma \vdash T : \mathsf{set} \quad x \notin \mathsf{dom}(\Gamma)}{\Sigma; \Gamma, x : T \vdash \diamond}
$$

$$
\frac{\Sigma; \Gamma, x : T \vdash M : U}{\Sigma; \Gamma \vdash \lambda(x : T) \,.\, M : T \to U} \qquad \frac{\Sigma; \Gamma \vdash M : T \to U \quad \Sigma; \Gamma \vdash N : T}{\Sigma; \Gamma \vdash M\,N : U}
$$

$$
\frac{\Sigma; \Gamma \vdash \diamond \quad (x : T) \in \Gamma}{\Sigma; \Gamma \vdash x : T} \qquad \frac{\Sigma, t : K; \Gamma \vdash M : U \quad t \notin \mathsf{ftv}(\Gamma)}{\Sigma; \Gamma \vdash \Lambda(t : K) \,.\, M : \forall(t : K) \,.\, U}
$$

$$
\frac{\Sigma; \Gamma \vdash M : \forall(t : K) \,.\, T \quad \Sigma \vdash U : K}{\Sigma; \Gamma \vdash M\,U : T[U/t]} \qquad \frac{\Sigma; \Gamma \vdash M : T \quad \Sigma \vdash T =_{\eta\beta} U : \mathsf{set}}{\Sigma; \Gamma \vdash M : U}
$$

Figure 2. System F_ω judgements

- $[\![\Sigma \vdash T : K]\!](\vec{A}) \in [\![K]\!]$ where $\vec{A} \in [\![\Sigma \vdash \diamond]\!]$, and
- $[\![\Sigma; \Gamma \vdash M : T]\!](\vec{A}, \vec{a}) \in [\![\Sigma \vdash T : \mathsf{set}]\!](\vec{A})$ where $\vec{A} \in [\![\Sigma \vdash \diamond]\!]$ and $\vec{a} \in [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A})$.

For example. the identity function has semantics:

$$
\begin{array}{l}
[\![\vdash \forall(t : \mathsf{set})(t \to t) : \mathsf{set}]\!]() \\
\quad = \prod_{A \in \mathsf{Set}} A \to A \\
[\![\vdash \Lambda(t : \mathsf{set}) \,.\, \lambda(x : t) \,.\, x : \forall(t : \mathsf{set})(t \to t)]\!]() \\
\quad = A \mapsto a \mapsto a
\end{array}
$$

We have to provide some sanity checks, to ensure that this definition is well-formed. In the semantics of $\Sigma; \Gamma \vdash \Lambda t \,.\, M : \forall t \,.\, T$, there is a use of weakening, which is justified because:

$$[\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) = [\![\Sigma, t : K; \Gamma \vdash \diamond]\!](\vec{A}, A) \text{ when } t \notin \mathsf{ftv}(\Gamma)$$

In the semantics of $\Sigma; \Gamma \vdash M\,T : U[T/t]$ there is a use of substitutivity, which is justified because:

$$[\![\Sigma, t : K \vdash U : L]\!](\vec{A}, [\![\Sigma \vdash T : K]\!](\vec{A})) = [\![\Sigma \vdash U[T/t] : L]\!](\vec{A})$$

In the semantics of $\Sigma; \Gamma \vdash M : T =_{\eta\beta} U$ there is a use of η-β-equivalence, which is justified because:

$$[\![\Sigma \vdash T : K]\!](\vec{A}) = [\![\Sigma \vdash U : K]\!](\vec{A}) \text{ when } \Sigma \vdash T =_{\eta\beta} U : K$$

In Figure 5 we extend the semantics of System F_ω types from sets to relations (writing $A \leftrightarrow B$ for $\mathcal{P}(A \times B)$) where:

- $[\![K]\!]^2(A, B) \in \mathsf{Set}$ where $A, B \in [\![K]\!]$,
- $[\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B}) \in \mathsf{Set}$ where $\vec{A}, \vec{B} \in [\![\Sigma \vdash \diamond]\!]$,
- $[\![\Sigma; \Gamma \vdash \diamond]\!]^2(\vec{\mathcal{R}}) \in [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) \leftrightarrow [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{B})$ where $\vec{\mathcal{R}} \in [\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B})$, and
- $[\![\Sigma \vdash T : K]\!]^2(\vec{\mathcal{R}}) \in [\![K]\!]^2(A, B)$ where $A = [\![\Sigma \vdash T : K]\!](\vec{A})$ and $B = [\![\Sigma \vdash T : K]\!](\vec{B})$ and $\vec{\mathcal{R}} \in [\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B})$.

This specializes to the usual presentation of logical relations for System F, in particular at function type:

$$
\begin{array}{l}
(f, g) \in [\![\Sigma \vdash T \to U : \mathsf{set}]\!]^2(\vec{\mathcal{R}}) \text{ whenever} \\
\quad (a, b) \in [\![\Sigma \vdash T : \mathsf{set}]\!]^2(\vec{\mathcal{R}}) \text{ implies} \\
\qquad (f(a), g(b)) \in [\![\Sigma \vdash U : \mathsf{set}]\!]^2(\vec{\mathcal{R}})
\end{array}
$$

and at polymorphic type:

$$
\begin{array}{l}
(f, g) \in [\![\Sigma \vdash \forall t \,.\, T]\!]^2(\vec{\mathcal{R}}) \text{ whenever} \\
\quad (f(A), g(B)) \in [\![\Sigma, t : \mathsf{set} \vdash T : \mathsf{set}]\!]^2(\vec{\mathcal{R}}, \mathcal{R}) \\
\quad \text{for every } A, B \text{ and } \mathcal{R} : A \leftrightarrow B
\end{array}
$$

$$
\begin{array}{rcl}
[\![K]\!] & \in & \mathsf{Set} \\
[\![\mathsf{set}]\!] & = & \mathsf{Set} \\
[\![K \to L]\!] & = & [\![K]\!] \to [\![L]\!] \\
[\![\Sigma \vdash \diamond]\!] & \in & \mathsf{Set} \\
[\![\Sigma \vdash \diamond]\!] & = & [\![K_1]\!] \times \cdots \times [\![K_n]\!] \\
[\![C]\!] & \in & [\![K]\!] \text{ where } C : K \\
[\![(\cdot \to \cdot)]\!] & = & A \mapsto B \mapsto A \to B \\
[\![(\forall K)]\!] & = & F \mapsto \prod_{A \in [\![K]\!]} F(A) \\
[\![\Sigma \vdash T : K]\!](\vec{A}) & \in & [\![K]\!] \text{ where } \vec{A} \in [\![\Sigma \vdash \diamond]\!] \\
[\![\Sigma \vdash C : K]\!](\vec{A}) & = & [\![C]\!] \\
[\![\Sigma \vdash \lambda t \,.\, T : K \to L]\!](\vec{A}) & = & A \mapsto [\![\Sigma, t : K \vdash T : L]\!](\vec{A}, A) \\
[\![\Sigma \vdash T\,U : L]\!](\vec{A}) & = & [\![\Sigma \vdash T : K \to L]\!](\vec{A})([\![\Sigma \vdash U : L]\!](\vec{A})) \\
[\![\Sigma \vdash t_i : K_i]\!](\vec{A}) & = & A_i \\
[\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) & \in & \mathsf{Set} \text{ where } \vec{A} \in [\![\Sigma \vdash \diamond]\!] \\
[\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) & = & [\![\Sigma \vdash T_1 : \mathsf{set}]\!](\vec{A}) \times \cdots \times [\![\Sigma \vdash T_n : \mathsf{set}]\!](\vec{A})
\end{array}
$$

Figure 3. System F_ω type semantics, where $\Sigma = (t_1 : K_1, \ldots, t_m : K_m)$ and $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$

$$
\begin{array}{rcl}
[\![\Sigma; \Gamma \vdash M : T]\!](\vec{A}, \vec{a}) & \in & [\![\Sigma \vdash T : \mathsf{set}]\!](\vec{A}) \text{ where } \vec{A} \in [\![\Sigma \vdash \diamond]\!] \text{ and } \vec{a} \in [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) \\
[\![\Sigma; \Gamma \vdash \lambda x \,.\, M : T \to U]\!](\vec{A}, \vec{a}) & = & a \mapsto [\![\Sigma; \Gamma, x : T \vdash M : U]\!](\vec{A}, \vec{a}, a) \\
[\![\Sigma; \Gamma \vdash M\,N : U]\!](\vec{A}, \vec{a}) & = & [\![\Sigma; \Gamma \vdash M : T \to U]\!](\vec{A}, \vec{a})([\![\Sigma; \Gamma \vdash N : U]\!](\vec{A}, \vec{a})) \\
[\![\Sigma; \Gamma \vdash x_i : T_i]\!](\vec{A}, \vec{a}) & = & a_i \\
[\![\Sigma; \Gamma \vdash \Lambda t \,.\, M : \forall t \,.\, U]\!](\vec{A}, \vec{a}) & = & A \mapsto [\![\Sigma, t : K; \Gamma \vdash M : U]\!](\vec{A}, A, \vec{a}) \\
[\![\Sigma; \Gamma \vdash M\,T : U[T/t]]\!](\vec{A}, \vec{a}) & = & [\![\Sigma; \Gamma \vdash M : \forall (t : T) \,.\, U]\!](\vec{A}, \vec{a})([\![\Sigma \vdash T : K]\!](\vec{A}))
\end{array}
$$

Figure 4. System F_ω expression semantics

$$
\begin{array}{rcl}
[\![K]\!]^2(A, B) & \in & \mathsf{Set} \text{ where } A, B \in [\![K]\!] \\
[\![\mathsf{set}]\!]^2(A, B) & = & (A \leftrightarrow B) \\
[\![K \to L]\!]^2(F, G) & = & \prod_{\mathcal{R} \in [\![K]\!]^2(A,B)} [\![L]\!]^2(F(A), G(B)) \\
[\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B}) & \in & \mathsf{Set} \text{ where } \vec{A}, \vec{B} \in [\![\Sigma \vdash \diamond]\!] \\
[\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B}) & = & [\![K_1]\!]^2(A_1, B_1) \times \cdots \times [\![K_n]\!]^2(A_n, B_n) \\
[\![C]\!]^2 & \in & [\![K]\!]^2([\![C]\!], [\![C]\!]) \text{ where } C : K \\
[\![(\cdot \to \cdot)]\!]^2 & = & \mathcal{R} \mapsto \mathcal{S} \mapsto \{(f, g) \mid \forall (a, b) \in \mathcal{R} \,.\, (f(a), g(b)) \in \mathcal{S}\} \\
[\![(\forall K)]\!]^2 & = & \mathcal{R} \mapsto \{(f, g) \mid \forall \mathcal{S} \in [\![K]\!]^2(A, B) \,.\, (f(A), g(B)) \in \mathcal{R}(\mathcal{S})\} \\
[\![\Sigma \vdash T : K]\!]^2(\vec{\mathcal{R}}) & \in & [\![K]\!]^2([\![\Sigma \vdash T : K]\!](\vec{A}), [\![\Sigma \vdash T : K]\!](\vec{B})) \text{ where } \vec{\mathcal{R}} \in [\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B}) \\
[\![\Sigma \vdash C : K]\!]^2(\vec{\mathcal{R}}) & = & [\![C]\!]^2 \\
[\![\Sigma \vdash \lambda t \,.\, T : K \to L]\!]^2(\vec{\mathcal{R}}) & = & \mathcal{R} \mapsto [\![\Sigma, t : K \vdash T : L]\!]^2(\vec{\mathcal{R}}, \mathcal{R}) \\
[\![\Sigma \vdash T\,U : L]\!]^2(\vec{\mathcal{R}}) & = & [\![\Sigma \vdash T : K \to L]\!]^2(\vec{\mathcal{R}})([\![\Sigma \vdash U : K]\!]^2(\vec{\mathcal{R}})) \\
[\![\Sigma \vdash t_i : K_i]\!]^2(\vec{\mathcal{R}}) & = & \mathcal{R}_i \\
[\![\Sigma; \Gamma \vdash \diamond]\!]^2(\vec{\mathcal{R}}) & \in & [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{A}) \leftrightarrow [\![\Sigma; \Gamma \vdash \diamond]\!](\vec{B}) \text{ where } \vec{\mathcal{R}} \in [\![\Sigma \vdash \diamond]\!]^2(\vec{A}, \vec{B}) \\
[\![\Sigma; \Gamma \vdash \diamond]\!]^2(\vec{\mathcal{R}}) & = & [\![\Sigma \vdash T_1 : \mathsf{set}]\!]^2(\vec{\mathcal{R}}) \times \cdots \times [\![\Sigma \vdash T_n : \mathsf{set}]\!]^2(\vec{\mathcal{R}})
\end{array}
$$

Figure 5. System F_ω logical relations, where $\Sigma = (t_1 : K_1, \ldots, t_m : K_m)$ and $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$

For example, the relational semantics of the type of the identity function is:

$$\begin{aligned}
&[\![\vdash \forall(t:\mathsf{set})(t\to t):\mathsf{set}]\!]^2()\\
&\quad= \{\,(f,g)\mid \forall\mathcal{R}\in A\leftrightarrow B\,.\,\forall(a,b)\in\mathcal{R}\,.\\
&\qquad\quad (f(A)(a),g(B)(b))\in\mathcal{R}\,\}\\
&\quad\in (\textstyle\prod_{A\in\mathsf{Set}} A\to A)\leftrightarrow\\
&\qquad (\textstyle\prod_{A\in\mathsf{Set}} A\to A)\\
&\quad= [\![\vdash \forall(t:\mathsf{set})(t\to t):\mathsf{set}]\!]()\leftrightarrow\\
&\qquad [\![\vdash \forall(t:\mathsf{set})(t\to t):\mathsf{set}]\!]()
\end{aligned}$$

We can verify that if i is the semantics of the polymorphic identity function:

$$i = A\mapsto a\mapsto a$$

then i is related to itself in the logical relation for its type:

$$(i,i)\in[\![\vdash \forall(t:\mathsf{set})(t\to t):\mathsf{set}]\!]^2()$$

In fact, this property is true for any System F_ω term, which is the *parametricity* property.

THEOREM 1 (Parametricity of System F_ω).

$\vec{\mathcal{R}}\in[\![\Sigma\vdash\diamond]\!]^2(\vec{A},\vec{B})$ *and* $(\vec{a},\vec{b})\in[\![\Sigma;\Gamma\vdash\diamond]\!]^2(\vec{\mathcal{R}})$ *implies*
$(a,b)\in[\![\Sigma\vdash T:\mathsf{set}]\!]^2(\vec{\mathcal{R}})$ *where*
$a=[\![\Sigma;\Gamma\vdash M:T]\!](\vec{A},\vec{a})$ *and*
$b=[\![\Sigma;\Gamma\vdash M:T]\!](\vec{B},\vec{b})$.

This theorem has been mechanically verified [20].

4. System FRP_ω

In this section, we define System FRP_ω, which extends System F_ω with a kind time and appropriate types and expressions to express the order of time. We encode many of the FRP combinators from Section 2 in System FRP_ω, and state relational parametricity. In Section 6, parametricity is used to establish causality.

The syntax, type judgements, and semantics of System FRP_ω are given as an extension of System F_ω in Figures 6–9. We introduce a kind time of times (similar to Jelsch's [23] phantom types for eras), together with a type $t\leq u$ for the order on time, and constants refl, trans, antisym and case which internalize the properties required of a total order. The semantics of System FRP_ω are given with respect to a chosen total order $(\mathsf{Time},\leq)$. We can then define the kind of reactive types as:

$$\mathsf{rset} \quad = \quad \mathsf{time}\to\mathsf{set}$$

and define many of the combinators for reactive types in System FRP_ω (although we defer $[T]$ and $\Box T$ to Section 5):

$$\begin{aligned}
\langle\cdot\rangle &\;:\; \mathsf{set}\to\mathsf{rset}\\
\langle\cdot\rangle &\;=\; \lambda a\,.\,\lambda t\,.\,a\\[4pt]
(\cdot\Rightarrow\cdot) &\;:\; \mathsf{rset}\to\mathsf{rset}\to\mathsf{rset}\\
(\cdot\Rightarrow\cdot) &\;=\; \lambda a\,.\,\lambda b\,.\,\lambda t\,.\,a\,t\to b\,t\\[4pt]
\cdot[\cdot,\cdot] &\;:\; \mathsf{rset}\to\mathsf{time}\to\mathsf{time}\to\mathsf{set}\\
\cdot[\cdot,\cdot] &\;=\; \lambda a\,.\,\lambda s\,.\,\lambda u\,.\,\forall t\,.\,(s\leq t)\to(t\leq u)\to a\,t\\[4pt]
(\cdot\trianglerighteq\cdot) &\;:\; \mathsf{rset}\to\mathsf{rset}\to\mathsf{rset}\\
(\cdot\trianglerighteq\cdot) &\;=\; \lambda a\,.\,\lambda b\,.\,\lambda t\,.\,\forall u\,.\,a[t,u]\to b\,u
\end{aligned}$$

These System FRP_ω combinators have the same semantics as defined in Section 2 (in some cases up to isomorphism, written $\approx$):

$$\begin{aligned}
&[\![\Sigma\vdash\langle T\rangle:\mathsf{rset}]\!](\vec{A})\\
&\quad=\langle[\![\Sigma\vdash T:\mathsf{set}]\!](\vec{A})\rangle\\
&[\![\Sigma\vdash T\Rightarrow U:\mathsf{rset}]\!](\vec{A})\\
&\quad=[\![\Sigma\vdash T:\mathsf{rset}]\!](\vec{A})\Rightarrow[\![\Sigma\vdash U:\mathsf{rset}]\!](\vec{A})\\
&[\![\Sigma\vdash T[t,u]:\mathsf{set}]\!](\vec{A})\\
&\quad\approx[\![\Sigma\vdash T:\mathsf{rset}]\!](\vec{A})[[\![\Sigma\vdash t:\mathsf{time}]\!](\vec{A}),[\![\Sigma\vdash u:\mathsf{time}]\!](\vec{A})]\\
&[\![\Sigma\vdash T\trianglerighteq U:\mathsf{rset}]\!](\vec{A})\\
&\quad\approx[\![\Sigma\vdash T:\mathsf{rset}]\!](\vec{A})\trianglerighteq[\![\Sigma\vdash U:\mathsf{rset}]\!](\vec{A})
\end{aligned}$$

There is a canonical singleton interval:

$$\begin{aligned}
\mathsf{sing} &\;:\; \forall a\,.\,\forall s\,.\,a\,s\to a[s,s]\\
\mathsf{sing} &\;=\; \Lambda a\,.\,\Lambda s\,.\,\lambda x\,.\,\Lambda t\,.\,\lambda s{\leq}t\,.\,\lambda t{\leq}s\,.\\
&\qquad \mathsf{antisym}\,a\,s\,t\,s{\leq}t\,t{\leq}s\,x
\end{aligned}$$

and intervals can be concatenated:

$$\begin{aligned}
\mathsf{concat} &\;:\; \forall a\,.\,\forall s\,.\,\forall t\,.\,\forall u\,.\,a[s,t]\to a[t,u]\to a[s,u]\\
\mathsf{concat} &\;=\; \Lambda a\,.\,\Lambda s\,.\,\Lambda t\,.\,\Lambda u\,.\,\lambda\sigma\,.\,\lambda\tau\,.\,\Lambda v\,.\,\lambda s{\leq}v\,.\,\lambda v{\leq}u\,.\\
&\quad \mathsf{case}(a\,v)\,v\,t\\
&\qquad (\lambda v{\leq}t\,.\,\sigma\,v\,s{\leq}v\,v{\leq}t)\\
&\qquad (\lambda t{\leq}v\,.\,\tau\,v\,t{\leq}v\,v{\leq}u)
\end{aligned}$$

Let T be the trivial reactive set:

$$\mathsf{T}(t)=\{*\}$$

Since T is trivial, we have that $\mathsf{T}[s,u]$ is also trivial:

$$\mathsf{T}[s,u]\approx\{*\}$$

We have that logical relations over T identify subsets of Time:

$$\begin{aligned}
[\![\mathsf{rset}]\!]^2(\mathsf{T},\mathsf{T}) &\;=\; \textstyle\prod_{\mathcal{R}\in[\![\mathsf{time}]\!]^2(s,t)}[\![\mathsf{set}]\!]^2(\mathsf{T}(s),\mathsf{T}(t))\\
&\;=\; \textstyle\prod_{s=t}\mathsf{T}(s)\leftrightarrow\mathsf{T}(t)\\
&\;=\; \textstyle\prod_{t}\mathsf{T}(t)\leftrightarrow\mathsf{T}(t)\\
&\;=\; \textstyle\prod_{t}\{*\}\leftrightarrow\{*\}\\
&\;\approx\; \mathcal{P}(\mathsf{Time})
\end{aligned}$$

In the same way as for System F_ω, we can show that the semantics of System FRP_ω respects weakening, substitutivity and η-β-equivalence, and that System FRP_ω satisfies parametricity.

THEOREM 2 (Parametricity of System FRP_ω).

$\vec{\mathcal{R}}\in[\![\Sigma\vdash\diamond]\!]^2(\vec{A},\vec{B})$ *and* $(\vec{a},\vec{b})\in[\![\Sigma;\Gamma\vdash\diamond]\!]^2(\vec{\mathcal{R}})$ *implies*
$(a,b)\in[\![\Sigma\vdash T:\mathsf{set}]\!]^2(\vec{\mathcal{R}})$ *where*
$a=[\![\Sigma;\Gamma\vdash M:T]\!](\vec{A},\vec{a})$ *and*
$b=[\![\Sigma;\Gamma\vdash M:T]\!](\vec{B},\vec{b})$.

This theorem has been mechanically verified [20].

5. Signals

In this section, we show how System FRP_ω can be used to define signals, in such a way that all implementable functions are causal.

The key observation is that we consider System FRP_ω types with a chosen free type variable $\kappa:\mathsf{rset}$. This type variable is always instantiated as the trivial reactive set T, but parametricity ensures that programs cannot instantiate $\kappa(t)$ directly. Thus, variables of type $\kappa(t)$ can be used as tokens, which allow access to signals at time t. If causality is thought of as an information flow property, then $\kappa(t)$ can be thought of as the type of *capabilities* for t.

$$\begin{array}{rrcl} \text{Kinds} & K, L & ::= & \cdots \mid \mathsf{time} \\ \text{Constant Types} & C & ::= & \cdots \mid (\cdot \leq \cdot) \\ \text{Constant Terms} & c & ::= & \mathsf{refl} \mid \mathsf{trans} \mid \mathsf{antisym} \mid \mathsf{case} \\ \text{Terms} & M, N & ::= & \cdots \mid c \end{array}$$

Figure 6. System FRP_ω syntactic extensions of System F_ω

$$\frac{\Sigma; \Gamma \vdash \diamond \quad c : T}{\Sigma; \Gamma \vdash c : T}$$

$$\begin{array}{rcl} (\cdot \leq \cdot) & : & \mathsf{time} \to \mathsf{time} \to \mathsf{set} \\ \mathsf{refl} & : & \forall t \,.\, (t \leq t) \\ \mathsf{trans} & : & \forall s \,.\, \forall t \,.\, \forall u \,.\, (s \leq t) \to (t \leq u) \to (s \leq u) \\ \mathsf{antisym} & : & \forall a \,.\, \forall t \,.\, \forall u \,.\, (t \leq u) \to (u \leq t) \to a\, t \to a\, u \\ \mathsf{case} & : & \forall a \,.\, \forall t \,.\, \forall u \,.\, ((t \leq u) \to a) \to ((u \leq t) \to a) \to a \end{array}$$

Figure 7. System FRP_ω judgements

$$\begin{array}{rcl} [\![\mathsf{time}]\!] & = & \mathsf{Time} \\ [\![(\cdot \leq \cdot)]\!] & = & t \mapsto u \mapsto \begin{cases} \{*\} & \text{if } t \leq u \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

Figure 8. System FRP_ω type semantics

$$\begin{array}{rcl} [\![\mathsf{time}]\!]^2(t, u) & = & \begin{cases} \{*\} & \text{if } t = u \\ \emptyset & \text{otherwise} \end{cases} \\ [\![\Sigma \vdash (\cdot \leq \cdot)]\!]^2(\vec{\mathcal{R}}) & = & * \mapsto * \mapsto \{(*, *)\} \end{array}$$

Figure 9. System FRP_ω logical relations

$$[\![\Sigma; \Gamma \vdash c : T]\!](\vec{A}, \vec{a}) = [\![c]\!]$$

$$\begin{array}{rcl} [\![c]\!] & \in & [\![\vdash T : \mathsf{set}]\!] \text{ where } c : T \\ [\![\mathsf{refl}]\!] & = & t \mapsto * \\ [\![\mathsf{trans}]\!] & = & s \mapsto t \mapsto u \mapsto * \mapsto * \mapsto * \\ [\![\mathsf{antisym}]\!] & = & A \mapsto t \mapsto u \mapsto * \mapsto * \mapsto a \mapsto a \\ [\![\mathsf{case}]\!] & = & A \mapsto t \mapsto u \mapsto g \mapsto h \mapsto \begin{cases} g(*) & \text{if } t \leq u \\ h(*) & \text{otherwise} \end{cases} \end{array}$$

Figure 10. System FRP_ω expression semantics

Define $\Sigma \vdash_\kappa T : K$ to mean that T has kind K in type context Σ together with $\kappa : \mathsf{rset}$, and similarly for the other judgements:

$$\begin{array}{rcl} (\Sigma \vdash_\kappa T : K) & = & (\kappa : \mathsf{rset}, \Sigma \vdash T : K) \\ (\Sigma; \Gamma \vdash_\kappa \diamond) & = & (\kappa : \mathsf{rset}, \Sigma; \Gamma \vdash \diamond) \\ (\Sigma; \Gamma \vdash_\kappa M : T) & = & (\kappa : \mathsf{rset}, \Sigma; \Gamma \vdash M : T) \end{array}$$

Define $[\![\Sigma \vdash_\kappa T : K]\!]_\kappa$ to be the semantics of T, where κ is instantiated as the trivial reactive set T, and similarly for the other judgements:

$$\begin{array}{rcl} [\![\Sigma \vdash_\kappa T : K]\!]_\kappa(\vec{A}) & = & [\![\kappa : \mathsf{rset}, \Sigma \vdash T : K]\!](\mathsf{T}, \vec{A}) \\ [\![\Sigma; \Gamma \vdash_\kappa \diamond]\!]_\kappa(\vec{A}) & = & [\![\kappa : \mathsf{rset}, \Sigma; \Gamma \vdash \diamond]\!](\mathsf{T}, \vec{A}) \\ [\![\Sigma; \Gamma \vdash_\kappa M : T]\!]_\kappa(\vec{A}, \vec{a}) & = & [\![\kappa : \mathsf{rset}, \Sigma; \Gamma \vdash M : T]\!](\mathsf{T}, \vec{A}, \vec{a}) \end{array}$$

Note that we do *not* provide a similar definition of logical relations $[\![\Sigma \vdash_\kappa T : K]\!]^2_\kappa$, which would instantiate κ by the trivial logical relation. Instead, we allow κ to be instantiated by any logical relation $\mathcal{R} \in [\![\mathsf{rset}]\!]^2(\mathsf{T}, \mathsf{T})$. As we have seen, such logical relations identify subsets of Time, which we can think of as the times a program is allowed access to. In Section 6, we use this to show that parametricity implies causality.

The presence of κ allows us to define the reactive type $\Box T$ to be $\kappa \trianglerighteq T$. A witness of $\Box T(s)$ is a witness for $T(t)$ for any $s \leq t$, assuming a capability for $[s, t]$. Similarly, a witness for $[T]$ is given by a witness for $T\, t$, assuming a capability for t:

$$\begin{array}{rcl} [\cdot] & : & \mathsf{rset} \to \mathsf{set} \\ [\cdot] & = & \lambda a \,.\, \forall t \,.\, \kappa\, t \to a\, t \\ \Box & : & \mathsf{rset} \to \mathsf{rset} \\ \Box & = & \lambda a \,.\, \kappa \trianglerighteq a \end{array}$$

In particular, since κ is instantiated by T, we have the promised isomorphism between $\Box T$ and the temporal global future modality:

$$\begin{array}{l} [\![\Sigma \vdash_\kappa \Box T : \mathsf{rset}]\!]_\kappa(\vec{A})(s) \\ \quad = [\![\Sigma \vdash_\kappa \kappa \trianglerighteq T : \mathsf{rset}]\!]_\kappa(\vec{A})(s) \\ \quad \approx ([\![\Sigma \vdash_\kappa \kappa : \mathsf{rset}]\!]_\kappa(\vec{A}) \trianglerighteq [\![\Sigma \vdash_\kappa T : \mathsf{rset}]\!]_\kappa(\vec{A}))(s) \\ \quad = (\mathsf{T} \trianglerighteq [\![\Sigma \vdash_\kappa T : \mathsf{rset}]\!]_\kappa(\vec{A}))(s) \\ \quad = \prod_{t \geq s} \mathsf{T}[s, t] \to [\![\Sigma \vdash_\kappa T : \mathsf{rset}]\!]_\kappa(\vec{A})(t) \\ \quad \approx \prod_{t \geq s} [\![\Sigma \vdash_\kappa T : \mathsf{rset}]\!]_\kappa(\vec{A})(t) \\ \quad = \Box([\![\Sigma \vdash_\kappa T : \mathsf{rset}]\!]_\kappa(\vec{A}))(s) \end{array}$$

There is a similar proof of the isomorphism between $[T]$ and temporal tautologies. Note, however, that these isomorphisms are *not* parametric in κ, and so cannot be implemented in System FRP_ω. We can give $\Box$ functorial structure by defining:

$$\begin{array}{rcl} \mathsf{map} & : & \forall a \,.\, \forall b \,.\, [a \Rightarrow b] \to [\Box a \Rightarrow \Box b] \\ \mathsf{map} & = & \Lambda a \,.\, \Lambda b \,.\, \lambda f \,.\, \Lambda s \,.\, \lambda j \,.\, \lambda \sigma \,.\, \Lambda t \,.\, \lambda s{\leq}t \,.\, \lambda k \,. \\ & & \quad f\, t\, (\sigma\, t\, s{\leq}t\, k) \end{array}$$

and give $\Box$ comonadic structure by defining:

$$\begin{array}{rcl} \delta & : & \forall a \,.\, [\Box a \Rightarrow \Box\Box a] \\ \delta & = & \Lambda a \,.\, \Lambda s \,.\, \lambda j \,.\, \lambda \sigma \,.\, \Lambda t \,.\, \lambda s{\leq}t \,.\, \lambda k \,.\, \Lambda u \,.\, \lambda t{\leq}u \,.\, \lambda \ell \,. \\ & & \quad \sigma\, u\, (\mathsf{trans}\, s\, t\, u\, s{\leq}t\, t{\leq}u)(\mathsf{concat}\, \kappa\, s\, t\, u\, k\, \ell) \\ \xi & : & \forall a \,.\, [\Box a \Rightarrow a] \\ \xi & = & \Lambda a \,.\, \Lambda s \,.\, \lambda k \,.\, \lambda \sigma \,. \\ & & \quad \sigma\, s\, (\mathsf{refl}\, s)(\mathsf{sing}\, \kappa\, s\, k) \end{array}$$

Note that the definition of δ shows that System FRP_ω can implement functions which are deep causal, but not shallow causal. We cannot, however, define the "predicting the future" function in System FRP_ω. An attempt is (assuming a $(\cdot + 1)$ function on time):

$$\begin{array}{lcl} \phi & : & \forall a \,.\, [\Box\langle a\rangle \Rightarrow \Box\langle a\rangle] \\ \phi & = & \Lambda a \,.\, \Lambda s \,.\, \lambda j \,.\, \lambda\sigma \,.\, \Lambda t \,.\, \lambda s{\le}t \,.\, \lambda k \,.\, \sigma(t+1)(\cdots)(?) \end{array}$$

but there is no way to fill in the hole of type $\kappa[s, t+1]$; we can use $k : \kappa[s,t]$, but there is no way to fill the gap of type $\kappa(t, t+1]$.

6. Causality

We can now formally define causality, and show that parametricity implies causality. For simplicity, we will consider causality for monomorphic types, although we expect the results could be extended to polymorphic types.

When $\vdash_\kappa T : \mathsf{set}$ and $a, b \in [\![\vdash_\kappa T : \mathsf{set}]\!]_\kappa$ and $u \in \mathsf{Time}$, define the (*deep*) *causal equivalence* $T \vDash a =_u b$ as:

$(s \le t) \vDash * =_u *$
always

$T \to U \vDash f =_u g$
whenever $U \vDash f(a) =_u g(b)$ for any $T \vDash a =_u b$

$\Box T(s) \vDash \sigma =_u \tau$
whenever $T(t) \vDash \sigma(t) =_u \tau(t)$ for any $s \le t \le u$

$[T] \vDash \sigma =_u \tau$
whenever $T(s) \vDash \sigma(s) =_u \tau(s)$ for any s

We then define $f \in [\![\vdash_\kappa T \to U : \mathsf{set}]\!]_\kappa$ to be (*deep*) *causal* whenever:

$$U \vDash f(a) =_u f(b) \text{ for all } T \vDash a =_u b$$

or equivalently:

$$T \to U \vDash f =_u f$$

Causal equivalence is an instance of parametricity, as can be shown by constructing a logical relation T^2_u as:

$$\begin{array}{lcl} \mathsf{T}^2_u & : & [\![\mathsf{rset}]\!]^2(\mathsf{T}, \mathsf{T}) \\ & & = \prod_{\mathcal{R} \in [\![\mathsf{time}]\!]^2(s,t)} [\![\mathsf{set}]\!]^2(\mathsf{T}(s), \mathsf{T}(t)) \\ & & = \prod_{s=t}(\{*\} \leftrightarrow \{*\}) \\ \mathsf{T}^2_u & = & (s = t) \mapsto \left\{ \begin{array}{ll} \{(*,*)\} & \text{if } t \le u \\ \emptyset & \text{otherwise} \end{array} \right. \end{array}$$

We can then show that the logical relation generated by T^2_u is exactly $=_u$.

PROPOSITION 3. *$T \vDash a =_u b$ iff $(a, b) \in [\![\vdash_\kappa T : \mathsf{set}]\!]^2(\mathsf{T}^2_u)$.*

From this, and parametricity of System FRP_ω, we have that every function implementable in System FRP_ω is deep causal.

THEOREM 4. *Every $[\![\vdash_\kappa M : T \to U]\!]_\kappa$ is causal.*

This theorem has been mechanically verified [20].

7. Implementation in Agda

Figure 11 shows some simple applications running in a browser. These are implemented in Agda, using a classic FRP library whose semantics is given in the style of this paper [19]. There is a matching compiler to ECMAScript, and a run-time system implementing FRP (which uses the idiomatic HTML5 event model, and an observer pattern for event notification). For example, the clock application is defined:

$$\mathsf{main} = \mathsf{text}(\mathsf{map}\,\mathsf{toUTCString}(\mathsf{every}(1\,\mathsf{sec})))$$

where:

- $\mathsf{every}(1\,\mathsf{sec})$ is a signal of Time, which changes value every second,
- $\mathsf{map}\, f(\sigma)$ applies a function $f : A \to B$ to a signal σ of A to get a signal of B, in this case f is $\mathsf{toUTCString} : \mathsf{Time} \to \mathsf{String}$, and
- $\mathsf{text}(\sigma)$ converts a signal σ of String to a signal of DOM nodes.

The types of these combinators are (ignoring some technical issues about the type for DOM nodes):

$$\begin{array}{rcl} \mathsf{every} & : & \mathsf{Delay} \to [\Box\langle\mathsf{Time}\rangle] \\ \mathsf{map} & : & [A \Rightarrow B] \to [\Box A \Rightarrow \Box B] \\ \mathsf{text} & : & [\Box\langle\mathsf{String}\rangle \Rightarrow \Box\mathsf{DOM}] \end{array}$$

which gives the type of main as $[\Box\mathsf{DOM}]$, that is a signal of DOM nodes, suitable for rendering in a browser.

The library makes use of Agda's system for inferring optional arguments. A function $\lambda\{x : A\} \,.\, M$ has type $\forall\{x : A\} \,.\, T$ whenever M has type T. A function $M : \forall\{x : A\} \,.\, B$ can be applied to an argument $N : A$ to give a result $M\{N\} : B[N/x]$. Agda will infer optional arguments if they are not provided explicitly[3]. We use optional arguments in defining $[\cdot]$:

$$\begin{array}{lcl} [\cdot] & : & \mathsf{rset} \to \mathsf{rset} \\ [\cdot] & = & \lambda a \,.\, \forall\{t\} \,.\, \forall\{k : \kappa\, t\} \,.\, a\, t \end{array}$$

So, making the optional arguments explicit, main is defined:

```
main =
  λ{t} . λ{k} .
    text{t}{k}
      (map
        (λ{u} . λ{ℓ} . toUTCString)
        {t}{k}
        (every(1 sec){t}{k}))
```

which type checks since:

$$\begin{array}{rcl} \mathsf{main} & : & [\Box\mathsf{DOM}] = \forall\{t\} \,.\, \forall\{k\} \,.\, (\Box\mathsf{DOM}(t)) \\ \mathsf{main} & = & \lambda\{t\} \,.\, M_1 \\ M_1 & : & \Box\mathsf{DOM}(t) \\ M_1 & = & M_2\{t\}\{k\}(M_3) \\ M_2 & : & [\Box\langle\mathsf{String}\rangle \Rightarrow \Box\mathsf{DOM}] \\ & & \quad = \forall\{t\} \,.\, \forall\{k\} \,.\, (\Box\langle\mathsf{String}\rangle(t) \to \Box\mathsf{DOM}(t)) \\ M_2 & = & \mathsf{text} \\ M_3 & : & \Box\langle\mathsf{String}\rangle(t) \\ M_3 & = & M_4\{t\}\{k\}(M_6) \\ M_4 & : & [\Box\langle\mathsf{Time}\rangle \Rightarrow \Box\langle\mathsf{String}\rangle] \\ & & \quad = \forall\{t\} \,.\, \forall\{k\} \,.\, (\Box\langle\mathsf{Time}\rangle(t) \to \Box\langle\mathsf{String}\rangle(t)) \\ M_4 & = & \mathsf{map}\, M_5 \\ M_5 & : & [\langle\mathsf{Time}\rangle \Rightarrow \langle\mathsf{String}\rangle] \\ & & \quad = \forall\{u\} \,.\, \forall\{\ell\} \,.\, (\mathsf{Time} \to \mathsf{String}) \\ M_5 & = & \lambda\{u\} \,.\, \lambda\{\ell\} \,.\, \mathsf{toUTCString} \\ M_6 & : & \Box\langle\mathsf{Time}\rangle(t) \\ M_6 & = & M_7\{t\}\{k\} \\ M_7 & : & [\Box\langle\mathsf{Time}\rangle] = \forall\{t\} \,.\, \forall\{k\} \,.\, \Box\langle\mathsf{Time}\rangle\{t\} \\ M_7 & = & \mathsf{every}(1\,\mathsf{sec}) \end{array}$$

[3] In this paper, we are eliding the difference between implicit arguments and instance arguments, since they only differ in the algorithm used to infer missing arguments.

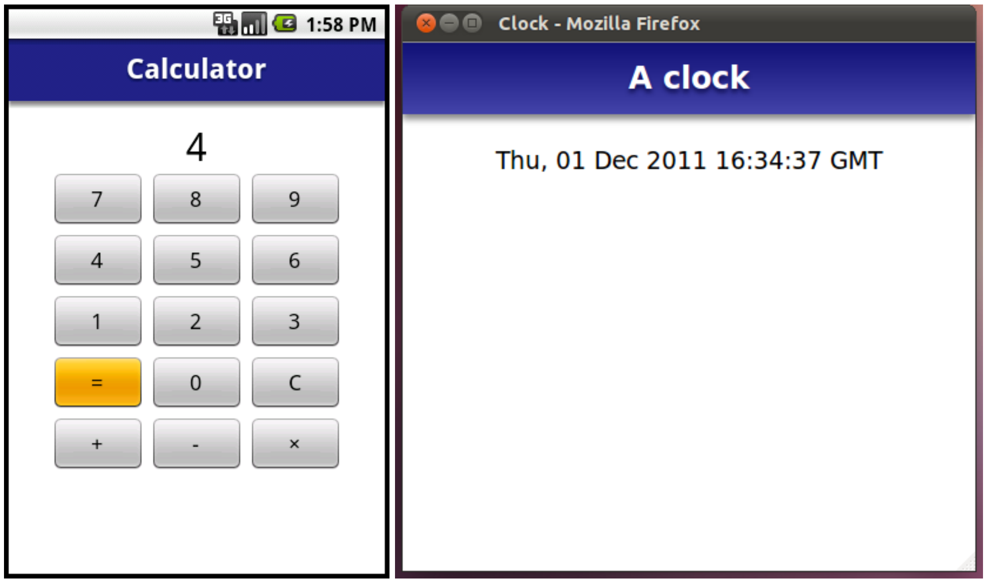


Figure 11. Example Agda programs running in the browser

Under the hood, the reactive type $\square A$ is implemented in ECMAScript, with an FFI binding to Agda. The implementation is based on Acar's [1] *self-adjusting computation*. Each signal is implemented as a node in a dataflow graph, which memoizes its current state. When a node changes state, it sends a notification to each of its downstream neighbours, which in turn may send further downstream notifications.

A simple application of the observer pattern results in *glitches*, which are notifications of transitory incorrect values. For example, in the dataflow graph for the expression $x = \neg x$, a state change to x sends a notification to the $=$ node and the $\neg$ node. If the $=$ node were to process the notification first, it would read a stale value from the $\neg$ node, so send a glitchy notification that its state is true.

To avoid glitches, we adopt a variant of Acar's technique [1], which is also used in [8, 10, 28]. Each node is *ranked*, such that every node has smaller rank than its observers. The run time system ensures that notifications are processed in rank order, which prevents glitches. For example, in the graph for $x = \neg x$, the $=$ node would be ranked greater than the $\neg$ node, so the $\neg$ node processes its notification before the $=$ node.

Each node in the dataflow graph maintains a set of pointers to its downstream observers, which has an impact on garbage collection, since these pointers may keep nodes alive unnecessarily. Since ECMAScript does not support weak pointers, we use a reference-counting scheme to remove any nodes with no observers. To ensure safety of this scheme, we maintain an invariant for any node of type $\square A(s)$, that after time s, we never add new observers, only remove them, so it is safe to remove a node which has no observers after time s. This garbage collection scheme is essentially the same as Jeltsch [23], but uses Agda's dependent types to express reactive types as temporal logic formulae, rather than relying on Haskell phantom types.

As well as an FRP implementation for GUI programming, the agda-frp-js library contains mechanizations of the theorems of this paper [20]. The definitions are essentially as given in this paper: the main differences are the use of de Bruijn indices for variables, and universe levels to avoid Set $\in$ Set.

The implementation of the FRP library and the compiler from Agda to ECMAScript is discussed in more detail in [22].

8. Conclusions and further work

In this paper, we have shown that for programs written in System FRP_ω, a kernel language for Haskell extended with time, every program is parametric. Moreover, we have shown that any parametric function is deep causal, and so every function implemented in System FRP_ω is deep causal. This allows programmers to write signal functions directly, rather than using an arrowized interface, without sacrificing causality. It provides the formal basis of the agda-frp-js [19] FRP library, which allows provably correct applications to run in a browser. This work leaves open some questions.

In this paper, we have considered System FRP_ω, which is the core of FRP programming in Haskell. System $\mathit{FRP}_\omega$is missing some important features, notably tagged unions, recursion and recursive types. We expect that tagged unions and recursion would not be problematic, but recursive types would introduce problems in the proofs that proceed by induction on type. Also, we have given a definition of causality for monotypes, and this should be generalized to polytypes.

We have not discussed the expressive power of System FRP_ω, and in particular, the existence of loop combinators. Currently System FRP_ω has no capabilities for induction over time; for discrete time models, such an induction combinator could be typed:

$$\forall a \,.\, \forall s \,.\, (\forall t \,.\, (s \leq t) \to a[s,t] \to a\, t) \to (\forall t \,.\, (s \leq t) \to a\, t)$$

We expect that such an induction combinator would preserve parametricity, and could be used to implement fixed points of type:

$$\forall a \,.\, [(\square a \Rightarrow \boxdot a) \Rightarrow \square a]$$

where $\boxdot$ is the type of *decoupled* signals:

$$\begin{array}{lcl} \boxdot & : & \mathsf{rset} \to \mathsf{rset} \\ \boxdot & = & \lambda a \,.\, \kappa \triangleright a \end{array}$$

where $A \triangleright B$ is the strict constrains modality:

$$\begin{array}{lcl} (\cdot \triangleright \cdot) & : & \mathsf{rset} \to \mathsf{rset} \to \mathsf{rset} \\ (\cdot \triangleright \cdot) & = & \lambda a \,.\, \lambda b \,.\, \lambda t \,.\, \forall u \,.\, a[t,u] \to b\, u \end{array}$$

which is defined in terms of semi-open intervals:

$$\cdot[\cdot,\cdot) : \mathsf{rset} \to \mathsf{time} \to \mathsf{time} \to \mathsf{set}$$
$$\cdot[\cdot,\cdot) = \lambda a\,.\,\lambda s\,.\,\lambda u\,.\,\forall t\,.\,(s \leq t) \to (t < u) \to a\,t$$

This would allow us to statically track coupled and decoupled signals, giving some of the power of Nilsson and Sculthorpe's [37] decoupling matrices.

The style of causality used here is non-monotone in that at function type the definition is:

$$T \to U \vDash f =_u g$$
$$\text{whenever } U \vDash f(a) =_u g(b) \text{ for any } T \vDash a =_u b$$

which is not monotone in u. A Kripke-style definition would be:

$$T \to U \vDash f =_u g$$
$$\text{whenever } U \vDash f(a) =_t g(b) \text{ for any } T \vDash a =_t b \text{ and } t \leq u$$

which is monotone in u. We never required monotonicity in our results, but it might be interesting to explore the relationship between causality and Kripke logical relations. As a special case, step-indexing [3] may shed light on FRP's loop combinators.

Acknowledgements. Many thanks to the anonymous referees

References

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon Univ., 2005.

[2] H. Apfelmus. Reactive-banana. `http://www.haskell.org/haskellwiki/Reactive-banana`.

[3] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Trans. Programming Languages and Systems*, 23(5):657683, 2001.

[4] N. Benton and M. Hyland. Traced premonoidal categories. *J. Theoretical Informatics and Applications*, 37:273–299, 2003.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Computer Programming*, 19(2):87–152, 1992.

[6] G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[7] S. Burbeck. Applications programming in smalltalk-80: How to use model-view-controller (MVC), 1987.

[8] G. H. Cooper and B. Adsul. Embedding dynamic dataflow in a call-by-value language. In *Proc. European Symp. on Programming*, pages 294–308, 2006.

[9] A. Courtney. Frappé: Functional reactive programming in Java. In *Proc. Symp. Pratical Aspects of Declarative Languages*, pages 29–44, 2001.

[10] J. Donham. Functional reactive programming in OCaml. `https://github.com/jaked/froc`.

[11] ECMAScript language specification. ECMA Standard 262, 5.1 Edition, 2011.

[12] C. Elliott. Push-pull functional reactive programming. In *Proc. Haskell Symp.*, 2009.

[13] C. Elliott. Garbage collecting the semantics of FRP, 2012. `http://conal.net/blog/posts/garbage-collecting-the-semantics-of-frp`.

[14] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. Int. Conf. Functional Programming*, pages 263–273, 1997.

[15] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[16] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[17] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. *Proc. Int. Joint Conf. Artificial Intelligence*, pages 235–245, 1973.

[18] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[19] A. S. A. Jeffrey. agda-frp-js. `https://github.com/agda/agda-frp-js/`, 2011.

[20] A. S. A. Jeffrey. agda-frp-js model. `https://github.com/agda/agda-frp-js/blob/master/src/agda/FRP/JS/Model.agda`, 2011.

[21] A. S. A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2012.

[22] A. S. A. Jeffrey. Provably correct web applications: FRP in Agda in HTML5. Submitted for publication, 2012.

[23] W. Jeltsch. Signals, not generators! In *Proc. Symp. Trends in Functional Programming*, pages 283–297, 2009.

[24] W. Jeltsch. The Curry-Howard correspondence between temporal logic and functional reactive programming. `http://www.cs.ut.ee/~varmo/tday-nelijarve/jeltsch-slides.pdf`, 2011.

[25] N. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *Proc. Int. Conf. Functional Programming*, pages 45–57, 2011.

[26] T. Leithead, J. Rossi, W3C D. Schepers, B. Höhrmann, P. Le Hégaret, and T. Pixley. Document object model (DOM) level 3 events specification. W3C Working Draft, 2012. `http://www.w3.org/TR/DOM-Level-3-Events/`.

[27] K. L. McMillan. Circular compositional reasoning about liveness. In *Proc. IFIP WG 10.5 Correct Hardware Design and Verification Methods*, pages 342–345, 1999.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proc, ACM Conf. Object-Oriented Programming Systems, Languages and Applications*, pages 1–20, 2009.

[29] K. S. Namjoshi and R. J. Trefler. On the competeness of compositional reasoning. In *Proc. Int. Conf. Computer Aided Verification*, pages 139–153, 2000.

[30] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. ACM Workshop on Haskell*, pages 51–64, 2002.

[31] R. Paterson. A new notation for arrows. In *Proc. ACM Int. Conf. Functional Programming*, pages 229–240, 2001.

[32] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[33] A. Pnueli. The temporal logic of programs. In *Proc. Symp. Foundations of Computer Science*, pages 46–57, 1977.

[34] A. J. Power and E. Robinson. Premonoidal categories and notions of computation. *Math. Structures in Comp. Sci.*, 7:453–468, 1997.

[35] A. J. Power and H. Thielecke. Closed Freyd- and kappa-categories. In *Proc. Int. Colloq. Automata, Languages and Programming*, pages 625–634. Springer, 1999.

[36] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, pages 513–523, 1983.

[37] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *Proc. ACM Int. Conf. Functional Programming*, pages 23–34, 2009.

[38] N. Sculthorpe and H. Nilsson. Keeping calm in the face of change: Towards optimisation of FRP by reasoning about change. *J. Higher-Order and Symbolic Computation*, 23(2):227–271, 2010.

[39] E. Sumii and B. C. Pierce. Logical relations for encryption. *J. Computer Security*, 11(4):521–554, 2003.

[40] P. Wadler. Theorems for free! In *Proc. Int. Conf. Functional Programming and Computer Architecture*, pages 349–359, 1989.

[41] Yale Haskell Group. Yampa library for programming hybrid systems. `http://www.haskell.org/haskellwiki/Yampa`.