Author's Version
The final publication is available at www.springerlink.com
DOI 10.1007/s10990-011-9068-x

Keeping Calm in the Face of Change

Towards Optimisation of FRP by Reasoning about Change

Neil Sculthorpe · **Henrik Nilsson**

November 6, 2011

Abstract Functional Reactive Programming (FRP) is an approach to reactive programming where systems are structured as networks of functions operating on signals (time-varying values). FRP is based on the synchronous data-flow paradigm and supports both (an approximation to) continuous-time and discrete-time signals (hybrid systems). What sets FRP apart from most other languages for similar applications is its support for systems with dynamic structure and for higher-order reactive constructs.

This paper contributes towards advancing the state of the art of FRP implementation by studying the notion of signal change and change propagation in a setting of structurally dynamic networks of n-ary signal functions operating on mixed continuous-time and discrete-time signals. We first define an ideal denotational semantics (time is truly continuous) for this kind of FRP, along with temporal properties, expressed in temporal logic, of signals and signal functions pertaining to change and change propagation. Using this framework, we then show how to reason about change; specifically, we identify and justify a number of possible optimisations, such as avoiding recomputation of unchanging values. Note that due to structural dynamism, and the fact that the output of a signal function may change because time is passing even if the input is unchanging, the problem is significantly more complex than standard change propagation in networks with static structure.

Keywords Change Propagation · Domain-Specific Languages · Functional Reactive Programming · Hybrid Systems · Synchronous Data-flow

Neil Sculthorpe
School of Computer Science, University of Nottingham, United Kingdom
Tel.: +44 (0)115 846 7664
Fax: +44 (0)115 951 4254
E-mail: nas@cs.nott.ac.uk

Henrik Nilsson
School of Computer Science, University of Nottingham, United Kingdom
Tel.: +44 (0)115 846 6506
Fax: +44 (0)115 951 4254
E-mail: nhn@cs.nott.ac.uk

1 Introduction

Functional Reactive Programming (FRP) grew out of Conal Elliott's and Paul Hudak's work on Functional Reactive Animation [15]. The idea of FRP is to allow the full power of modern Functional Programming to be used for implementing *reactive systems*: systems that interact with their environment in a timely manner[1]. This is achieved by describing systems in terms of functions mapping *signals* (time-varying values) to signals, and combining such *signal functions* into signal-processing networks, through plain function composition and other combining forms. The nature of the signals depends on the application domain. Examples include sensor input in robotics applications [34], video streams in the context of graphical user interfaces [11] and games [12,7], and synthesised sound signals [16].

A number of FRP variants exist. We discuss the basics of the original approach [15,39], now commonly referred to as Classic FRP (CFRP), in Section 3, and we give an overview of a number of others in Section 9. However, the *synchronous data-flow principle*, and support for both *continuous* and *discrete* time (hybrid systems), are common to most of the variants. There are thus close connections to synchronous data-flow languages such as Esterel [3], Lustre [20], and Lucid Synchrone [35]; hybrid automata [21]; and languages for hybrid modelling and simulation, such as Simulink [1]. However, FRP goes beyond most of these approaches by supporting *dynamism* (highly dynamic system structure), and first-class reactive constructs[2] (higher-order data-flow). Support for structural dynamism significantly extends the range of reactive systems that can be described naturally and easily. Typical examples include video games [12,7]; virtual reality applications [4]; and maintaining models of a changing number of entities in view, say for UAV[3] applications [27].

It is well-known how to implement first-order synchronous data-flow networks with static structure efficiently [24,20]. However, higher-order data-flow and dynamic system structure in combination with support for hybrid systems raise new implementation challenges. FRP implementations usually adopt either a *push* or *pull* driven implementation strategy [14]. The essence of the push-driven approach is reaction to events occurring at some specific point in time by pushing changes through the system. This is a good fit for discrete-time signals. Pull is the opposite, where the need to compute the current value of a signal necessitates computing the current values of any signals it depends on, thus pulling data through the network. This is a good fit for continuous-time signals. Thus push and pull have complementary strengths, but combining the approaches into one system has turned out to be hard.

In this paper, by studying the notion of signal change and how change propagates in a structurally dynamic signal-processing network, we contribute to the state of the art of FRP implementation by identifying when computation is unnecessary and thus could be avoided. We hope this will help reconcile the advantages of push and pull. Note that structural dynamism, and the fact that FRP signals can change just because time passes, make the problem significantly more complex than standard change propagation in a network with a static structure.

The FRP variant that is the starting point for this paper is Yampa [27], a domain-specific embedding in Haskell and one of the most expressive FRP implementations to date when it comes to structural dynamism. Yampa takes signal functions to be the primary reactive ab-

[1] A response is expected within an amount of time that is "reasonable" for the application at hand. We do thus not predicate reactivity on hard real-time guarantees; most FRP variants only achieve soft real-time.

[2] Signals or signal functions, depending on the FRP variant.

[3] Unmanned Aerial Vehicle

straction; signals are secondary, existing only indirectly through the notion of signal functions. Many other approaches to FRP, including CFRP, opt to make signals their primary notion. We give the necessary background and discuss why this is an important FRP design consideration in sections 2 and 3, and in Section 3.5 we explain in some detail our reasons for choosing signal functions as the base. In brief, the signal function approach in principle allows for a strict separation between the reactive and functional layers. This in turn facilitates implementation in many ways, and provides certain conceptual and expressivity advantages.

However, note that a setting of signal functions is also the *natural* choice for studying change and change propagation in signal processing networks. We need to understand the properties of the network nodes, and these nodes *are* signal functions, regardless of the surface syntax used to set up the network. Thus, much of our study is relevant to FRP in general, not just to FRP versions based on signal functions.

We argue that an FRP approach based on signal functions has some distinct advantages, and note that Yampa, currently the main such FRP variant, is demonstrably useful for fairly demanding applications [7, 16, 4]. However, it is also clear that the current version of Yampa has a number of conceptual as well as practical issues that, amongst other things, limit its scalability. A discussion can be found in Section 4.

These issues were what initially prompted us to start investigating a new FRP model based on *n-ary* (multi-input and output) signal functions and differentiated kinds of signals [36, 37]. Working towards overcoming the limitations of the present Yampa approach is an additional goal of this paper, so we adopt this model, under the name N-ary FRP, and develop it further and in more detail. We then use this model to identify and study general temporal properties relevant to *any* faithful implementation of the model, and how these can be used to justify a range of optimisations. For example, given a signal-function network, we can characterise exactly which signals remain unchanged for arbitrary combinations of changing or unchanging input, and where updates nonetheless are necessary due to internal state changes.

In more detail, the contributions of this paper are as follows:

- We define a denotational semantics for an ideal, mixed continuous-time and discrete-time N-ary FRP. While a concrete digital implementation would have to approximate some aspects of this semantics, it does capture a number of temporal aspects that a faithful implementation would have to, and can, respect exactly.
- We identify a number of temporal properties of n-ary signal functions that are useful for studying the behaviour of networks of such signal functions, especially concerning how changes propagate, and we define these properties exactly using temporal logic.
- We characterise the primitive N-ary FRP signal functions and combinators, as defined by the denotational semantics, in terms of which of our temporal properties they satisfy or preserve, and we study the relations between these properties.
- We demonstrate how our temporal properties can be used to justify optimisations and identify optimisation opportunities in concrete networks of n-ary signal functions, assuming that the implementation is faithful to the temporal properties: that is, assuming that the implementation of each primitive satisfies the same temporal properties as the denotational model of the primitive.

A limitation of the present paper, compared both with Yampa and some of our earlier work [37], is that we only consider *acyclic* networks, leaving consideration of cyclic networks (feedback) as future work.

The notation we use in this paper is mostly that of *Agda* [30], a dependently typed functional language with many similarities to Haskell. We have chosen Agda as it can be used for defining the semantics of FRP, for proving properties about that semantics, and as a host language for an FRP embedding. However, as our interest is in a conceptual model, rather than a specific implementation, we occasionally make use of more general mathematical notation when defining the semantics. We also borrow some syntax from Haskell in order to clarify the presentation. In particular, we allow: pattern matching under lambdas, operator sections, case expressions, pattern guards, list comprehensions and overloading. In all cases that we do so, it is possible to translate into equivalent (but more verbose) Agda code in a fairly straightforward manner[4]. To make clear the distinction between conceptual definitions and embedded FRP code, we use the $\approx$ symbol when defining an entity conceptually.

The rest of the paper is structured as follows:

- Section 2 explains the fundamental concepts of FRP.
- Section 3 reviews Classic FRP, and motivates the first-class signal-function abstraction.
- Section 4 describes the new conceptual N-ary FRP model.
- Section 5 defines the primitives of the N-ary FRP language, and then demonstrates how they can be used to construct N-ary FRP programs.
- Section 6 discusses optimisation opportunities for FRP implementations.
- Section 7 defines a number properties of signals and signal functions using temporal logic, in particular properties related to change and how signal functions propagate change.
- Section 8 describes how the properties from Section 7 give rise to optimisations.
- Section 9 considers related work.
- Section 10 considers future work.
- Section 11 provides some concluding remarks.

2 FRP Fundamentals

FRP programs can be considered to have two levels to them: a *functional level* and a *reactive level*. The functional level is a pure functional language. FRP implementations are usually embedded in a host language, and in these cases the functional level is provided entirely by the host. For example, Haskell is the host language of CFRP [15, 39] and Yampa [27]. The reactive level is concerned with time-varying values called *signals*. At this level, functions operating on signals are used to construct synchronous data-flow networks. The levels are, however, interdependent. The reactive level relies on the functional level for carrying out arbitrary pointwise computations on signals, while reactive constructs are first-class entities at the functional level.

2.1 Continuous-Time Signals

Time is considered to be continuous in FRP. Signals are thus modelled as functions from continuous-time to value, where we take time to be the set of non-negative real numbers:

$$\mathit{Time} \approx \{t \in \mathbb{R} \mid t \geqslant 0\}$$
$$\mathit{Signal}\ A \approx \mathit{Time} \to A$$

[4] The code can be viewed on the first author's website: http://www.cs.nott.ac.uk/∼nas/hosc10.html

This conceptual model provides the foundation for an ideal FRP semantics. Of course, any digital implementation of continuous signals will have to execute over a discrete series of time steps, and will consequently only approximate the ideal semantics. The advantage of the conceptual model is that it abstracts away from such implementation details. It makes no assumptions as to the rate of sampling, whether the sampling rate is fixed, or how sampling is performed. It also avoids many of the problems of composing subsystems that have different sampling rates. The ideal semantics is helpful for understanding FRP programs, at least to a first approximation. It is also abstract enough to leave FRP implementers considerable freedom.

That said, implementing FRP completely faithfully to the ideal semantics is challenging. At the very least, a faithful implementation should, for "reasonable programs", converge to the ideal semantics in the limit as the sampling interval tends to zero [39]. But even then it is hard to know how densely one needs to sample before an answer is acceptably close to the ideal.

2.2 Signal Functions

Signal functions are conceptually functions on signals:

$$SF\ A\ B \approx Signal\ A \rightarrow Signal\ B$$

In the N-ary FRP model we define later in this paper (Section 4), as well as in Yampa [27], it is signal functions, rather than signals, that are first-class entities. Signals have no independent existence of their own; they exist only indirectly through the signal functions.

What if plain signals are needed; that is, a time-varying value that depends on no input? Well, a signal function that takes a unit signal as input essentially serves the same purpose. Alternatively, a signal function that completely ignores its input, and therefore is polymorphic in the type of the input signal, could be used. (However, see the discussion in Section 2.5: these are really *signal generators*.)

To make it possible to implement signal functions such that output is produced in lock-step with the input arriving, as is required for a system to be reactive, we constrain signal functions to be *temporally causal*:

Definition 1 (Causal Signal Function) A signal function is *temporally causal* if, at any given time, its output can depend upon its past and present inputs, but not its future inputs.

There are other notions of causality, but, throughout this paper, when we say causality we will always mean temporal causality. We define causality formally in Section 7.3.2.

2.3 Discrete-Time Signals

Conceptually, discrete-time signals (often called event signals) are signals whose domain of definition is an at-most-countable set of points in time. Each point of definition signifies some event that is without any extent in time. Inclusion of discrete-time signals, along with operations on them and operations for mediating between continuous-time and discrete-time signals, is what makes most FRP variants capable of handling hybrid systems.

However, different FRP variants have taken different approaches to the nature of discrete-time signals. One possibility is to make a fundamental distinction between continuous-time

and discrete-time signals on the grounds that they enjoy distinct properties. Separating them facilitates taking advantage of these differences for being more precise about applicable operations or for optimisation purposes. We refer to this approach as *multi-kinded* FRP as there is more than one kind of signal. For example, CFRP is multi-kinded.

Another possibility is to define discrete-time signals as a subtype of continuous-time signals by lifting the range of signals using an option type. We refer to this approach as *single-kinded* FRP as there fundamentally is only one kind of signal. For example[5]:

```
data Event (A : Set) where
  noEvent :       Event A
  event   : A →   Event A
```

A discrete-time signal carrying elements of type *A* would then be represented as a signal of type *Signal* (*Event A*), with a value of noEvent whenever the discrete-time signal is not defined, and a value of event *v* whenever the discrete-time signal is defined with value *v*.

Yampa is single-kinded: a uniform treatment of continuous-time and discrete-time signals fits well with the idea of signal functions being the core concept and there only being one kind of signal function. However, single-kindedness has some drawbacks. We will return to this in Section 4.

2.4 Structural Dynamism

Most FRP variants support *structural dynamism*. This means that the structure of the signal-processing network may change at run-time, and, furthermore, that new signals or signal functions may be computed at run-time. We refer to such changes in the network structure as *structural switches*.

A common way to allow for structural dynamism is to provide one or more *switching combinators*. As structural switches are discrete instantaneous occurrences, event signals are used to control when they happen. A switch occurs at the point in time of the first occurrence in the event signal. This point is the *moment of switching*. The details of switching combinators vary between FRP systems, but the essential idea is that, at the moment of switching, one signal, called the *subordinate* signal, is removed from the network, and a new signal, called the *residual* signal, is inserted in its place. At the moment of switching, the subordinate signal is *switched out* and the residual signal is *switched in*.

Switching combinators are often designed to allow the residual signal to *depend* on the value carried by the event that triggered the switch. This means that, in general, the residual signal is computed at the moment of switching. This has important consequences. First, it cannot be assumed that switching only happens within a predetermined finite set of system configurations. Second, it raises the question as to over what range of time the residual signal is defined: from the system start time or from the time it was switched in? We return to this discussion in the next section.

For a concrete example of a switching combinator, see Section 3.3 that provides a formal definition of such a combinator in the setting of CFRP.

What if we are in a setting where signal functions, not signals, is the primary reactive abstraction? In that case, switching takes place between signal functions, not signals. Other than that, the ideas are very similar. See Section 5.1.2 for a definition of that style of switching combinator.

[5] Agda notation: *Set* is the type of types, similar to kind $*$ in Haskell.

2.5 Signal Generators

As previously mentioned, switching combinators defined on signals could either "start" the residual signal at the same time as the subordinate signal, or when it is switched in. Note that if all provided switching combinators adhere to the first option, then the start times of *all* signals in the entire system would always coincide with the system start time.

The first choice is problematic if the residual signal depends on the value of the triggering event, as this is not known until the moment of switching. Consequently, when the switch occurs, the signal has to be retroactively computed up to that moment. In an implementation, this requires all past input to be remembered, a so-called *space leak*, and a catch-up computation to be performed, a so-called *time leak*. This is particularly troublesome if all provided switching combinators are of the first kind, as that would mean that all newly switched-in signals are subject to catch-up computations from the system start time. Furthermore, no input could ever be discarded: an increasingly cumbersome prospect the longer the up-time of the system. Consequently, most FRP variants with first-class signals choose the second option: to start the residual signal at the moment of switching.

However, once we have signals that can start at different times, the conceptual model of signals as functions from time to value is no longer sufficient: the value of a signal no longer just depends on the time at which it is sampled, but also the time at which it starts. To express this, the concept of a *signal generator* is needed:

$$\begin{aligned} &\mathit{StartTime} &&= \mathit{Time} \\ &\mathit{SampleTime} &&= \mathit{Time} \\ &\mathit{SignalGenerator}\ A &&\approx \mathit{StartTime} \to \mathit{SampleTime} \to A \end{aligned}$$

Or, equivalently, a signal generator is a function that, given a start time as an argument, produces a signal as the result:

$$\mathit{SignalGenerator}\ A \approx \mathit{StartTime} \to \mathit{Signal}\ A$$

The key point is that two signals created from the same signal generator can be (and often are) different if started at different times.

3 Classic FRP

To give further background on FRP, we take a look at the original FRP work known as Classic FRP (CFRP) in this section. This should also give a better understanding of the relevance of, and relation between, the various FRP notions discussed in the previous section (specifically signals, signal generators and signal functions). There are several variants of CFRP, but they are all based around the idea of multi-kinded first-class signals: *Behaviours* (continuous-time signals) and *Events* (discrete-time signals). In the following we introduce a basic CFRP language and give some examples of CFRP programming. However, as CFRP is not the principal topic of this paper, only primitives required for our examples are discussed. We conclude the section with a discussion on first-class signals vs. first-class signal functions, as a motivation for the work presented in the rest of the paper.

3.1 Behaviours and Events

We said that CFRP has first-class signals called *Behaviours* and *Events*. In fact, in most CFRP variants, *Behaviours* and *Events* are *signal generators*. Conceptually then, a *Behaviour* is a function that maps a start time and a sample time to a value:

$$\mathit{Behaviour}\ A \approx \mathit{StartTime} \to \mathit{SampleTime} \to A$$

An *Event* is similar, except that it produces a (time-ordered and finite) list of all event occurrences *up to* the sample time:

$$\mathit{Event}\ A \approx \mathit{StartTime} \to \mathit{SampleTime} \to \mathit{List}\ (\mathit{Time} \times A)$$

3.2 CFRP Primitives

We now introduce some CFRP primitives, along with their conceptual definitions. The utility functions used in these definitions can be found in Appendix A. We adopt the naming convention of adding a 'B' or 'E' suffix to distinguish between similar functions that operate on *Behaviours* and *Events*, respectively. In most implementations, some form of overloading is usually employed.

We begin with a family of lifting combinators that allow us to lift pure functions from the functional level to operate on *Behaviours* and *Events* in a pointwise fashion[6]:

$$\mathit{constant} : \{A : \mathit{Set}\} \to A \to \mathit{Behaviour}\ A$$
$$\mathit{constant}\ a \approx \lambda\ t_0\ t_1 \to a$$
$$\mathit{liftE} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{Event}\ A \to \mathit{Event}\ B$$
$$\mathit{liftE}\ f\ \mathit{ev} \approx (\mathit{result2} \circ \mathit{map} \circ \mathit{second})\ f\ \mathit{ev}$$
$$\mathit{liftB} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{Behaviour}\ A \to \mathit{Behaviour}\ B$$
$$\mathit{liftB}\ f\ \mathit{beh} \approx \mathit{result2}\ f\ \mathit{beh}$$
$$\mathit{liftB2} : \{A\ B\ C : \mathit{Set}\} \to (A \to B \to C) \to \mathit{Behaviour}\ A \to \mathit{Behaviour}\ B \to \mathit{Behaviour}\ C$$
$$\mathit{liftB2}\ f\ \mathit{beh}_1\ \mathit{beh}_2 \approx \lambda\ t_0\ t_1 \to f\ (\mathit{beh}_1\ t_0\ t_1)\ (\mathit{beh}_2\ t_0\ t_1)$$

A more interesting primitive is the *integral* function that integrates a *Behaviour* with respect to time. Note that unlike the liftings above, the value of the output *Behaviour* at any given time depends upon past inputs.

$$\mathit{integral} : \mathit{Behaviour}\ \mathbb{R} \to \mathit{Behaviour}\ \mathbb{R}$$
$$\mathit{integral}\ \mathit{beh} \approx \lambda\ t_0\ t_1 \to \int_{t_0}^{t_1} (\mathit{beh}\ t_0\ t)\ dt$$

It is also useful to have an integration function that has an initial value other than zero. We can define such an initialised integration within the CFRP language (rather than as a primitive):

$$\mathit{iIntegral} : \mathbb{R} \to \mathit{Behaviour}\ \mathbb{R} \to \mathit{Behaviour}\ \mathbb{R}$$
$$\mathit{iIntegral}\ x = \mathit{liftB}\ (+x) \circ \mathit{integral}$$

Finally, we introduce a primitive function that mediates between *Behaviours* and *Events*:

$$\mathit{when} : \{A : \mathit{Set}\} \to (A \to \mathit{Bool}) \to \mathit{Behaviour}\ A \to \mathit{Event}\ A$$

[6] Agda notation: Curly braces are used to enclose implicit arguments: arguments that only have to be provided at an application site if they cannot be inferred from the context. Implicit type arguments are often used to define polymorphic functions.

We have omitted the conceptual definition of *when* as it is quite involved. It can be found in Wan and Hudak [39]. Informally, the resultant *Event* contains an occurrence at each point in time that the predicate (the first explicit argument) applied to the value of the *Behaviour* (the second explicit argument) changes from false to true. The value of the occurrence is the value of the *Behaviour* at that point in time. Let us emphasise that events occur only when the result of the predicate *changes*, not whenever it holds.

3.3 Switching between Behaviours

As discussed in Section 2.4, switching combinators are a crucial aspect of FRP as they allow us to construct dynamic programs. Here we define a CFRP switching combinator:

$$
\begin{aligned}
&\mathit{untilB} : \{A : \mathit{Set}\} \to \mathit{Behaviour}\ A \to \mathit{Event}\ (\mathit{Behaviour}\ A) \to \mathit{Behaviour}\ A \\
&\mathit{untilB}\ \mathit{beh}_1\ \mathit{ev} \approx \lambda\ t_0\ t_1 \to \mathbf{case}\ \mathit{ev}\ t_0\ t_1\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad\qquad\ \to \mathit{beh}_1\ t_0\ t_1 \\
&\qquad\qquad\qquad\qquad (t_e, \mathit{beh}_2) :: \_ \to \mathit{beh}_2\ t_e\ t_1
\end{aligned}
$$

The first argument (beh_1) is the subordinate *Behaviour*, the second argument (*ev*) is the *Event* that controls the switch occurrence. The value of the *Event* is a *Behaviour*, and it is this *Behaviour* that will be switched in as the residual *Behaviour*. The residual *Behaviour* will not start until it is switched in, and at the moment of switching the overall value is taken from the residual *Behaviour*.

Recall that an alternative design choice would be to have the residual *Behaviour* start at the same time as the subordinate *Behaviour*. The semantics of such a switching combinator would be:

$$
\begin{aligned}
&\mathit{untilB}' : \{A : \mathit{Set}\} \to \mathit{Behaviour}\ A \to \mathit{Event}\ (\mathit{Behaviour}\ A) \to \mathit{Behaviour}\ A \\
&\mathit{untilB}'\ \mathit{beh}_1\ \mathit{ev} \approx \lambda\ t_0\ t_1 \to \mathbf{case}\ \mathit{ev}\ t_0\ t_1\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad\qquad\ \to \mathit{beh}_1\ t_0\ t_1 \\
&\qquad\qquad\qquad\qquad (t_e, \mathit{beh}_2) :: \_ \to \mathit{beh}_2\ t_0\ t_1
\end{aligned}
$$

However, if *all* switches were of the *untilB*′ type, then t_0 would always be 0, the global system start time. This means that the start time parameter becomes redundant, and the definitions of *Behaviour* and *Event* become signals as opposed to signal generators. But as we have discussed, this leads to severe performance problems and so tends to be avoided.

3.4 Example: Bouncing Balls

As an example, we will construct a simple model of bouncing balls. This a hybrid model, because the continuous motion of the balls is broken by discrete events (when the ball hits the ground). For simplicity, we assume no air resistance and consider only one dimension (the height of the ball above the ground). To demonstrate the modularity and higher-order benefits of FRP, we will consider distinct balls that behave differently when they impact the ground.

We represent the configuration of a ball by a pair of its height and velocity:

$$
\begin{aligned}
&\mathit{Acceleration} = \mathbb{R} \\
&\mathit{Velocity} = \mathbb{R} \\
&\mathit{Height} = \mathbb{R} \\
&\mathit{Ball} = \mathit{Height} \times \mathit{Velocity}
\end{aligned}
$$

For the purposes of this example, we assume our units are metres and seconds. We thus set the gravitational constant:

```
g : Acceleration
g = 9.81
```

We now construct a *Behaviour* that models a freely falling ball. This is achieved by integrating the acceleration (in this case caused by gravity) to compute the velocity, and integrating the velocity to compute the height. The *Behaviour* is parameterised on an initial ball configuration:

```
falling : Ball → Behaviour Ball
falling (h0, v0) = let a = constant (−g)
                       v = iIntegral v0 a
                       h = iIntegral h0 v
                   in
                     liftB2 (,) h v
```

The next step is to model interaction with the ground. We define a predicate to detect when a ball impacts the ground, and a function that negates a ball's velocity:

```
detectImpact : Ball → Bool
detectImpact (h, v) = (h <= 0) && (v < 0)
negateVel : Ball → Ball
negateVel (h, v) = (h, −v)
```

We now turn our attention to the bounce itself. A bounce is a discrete occurrence that will cause a discontinuity in the behaviour of the ball. Clearly then, a bounce is an event, and a bounce detector is a function mapping *Behaviour Ball* to *Event Ball* (the value of the event is the configuration of the ball at the moment of impact):

```
detectBounce : Behaviour Ball → Event Ball
detectBounce = when detectImpact
```

We can now define a *Behaviour* for a ball that bounces perfectly elastically:

```
elasticBall : Ball → Behaviour Ball
elasticBall b = let beh = falling b
                in
                  untilB beh (liftE (elasticBall ∘ negateVel) (detectBounce beh))
```

Intuitively, this says that an elastic ball should behave as a falling ball until a bounce is detected. At which point, the ball should have its velocity negated, and then have its configuration used to initialise a new *elasticBall*.

Note that *elasticBall* is recursively defined. When the bounce occurs, a new *elasticBall* *Behaviour* begins, taking the final ball configuration from the previous *Behaviour* as its initial configuration. We now see the usefulness of not starting a *Behaviour* until it is switched in. Imagine the ball first bounces after 5 seconds. If *elasticBall* had been defined using *untilB′*, then the residual *Behaviour* immediately after being switched in would be the ball's configuration 5 seconds after that bounce! This is not to say that there are never situations when it is desirous to have *Behaviours* starting before they are switched in though, as we will see shortly.

First however, we define a *Behaviour* for a ball that collides perfectly inelastically with the ground:

```
inelasticBall : Ball → Behaviour Ball
inelasticBall b = let beh = falling b
                  in
                    untilB beh (liftE (λ _ → constant (0,0)) (detectBounce beh))
```

Notice the similarity of the *elasticBall* and *inelasticBall* definitions. There is obviously an opportunity for abstraction here, so we define a more general model of a bouncing ball that is parameterised on the *Behaviour* to switch in when bouncing:

```
bouncingBall : (Ball → Behaviour Ball) → Ball → Behaviour Ball
bouncingBall f b = let beh = falling b
                   in
                     untilB beh (liftE f (detectBounce beh))
```

We can then redefine our two balls as:

```
elasticBall   = bouncingBall (elasticBall ∘ negateVel)
inelasticBall = bouncingBall (λ _ → constant (0,0))
```

Finally, we add the capacity for the ball to be arbitrarily moved to a new position (and given a new velocity) by some external actor. We model this as an *Event*, with the event values being new ball configurations. An event occurrence therefore represents a repositioning of the ball (which we will call a reset). The intuitive way to express this would seem to be as follows:

```
resetBB : (Ball → Behaviour Ball) → Event Ball → Ball → Behaviour Ball
resetBB f ev b = untilB (bouncingBall f b) (liftE (resetBB f ev) ev)
```

Thus, *resetBB* (resettable bouncing ball) behaves as *bouncingBall* until a reset event occurs, at which point it recursively starts *resetBB*, using the same *Event* but a new initial ball configuration.

However, this doesn't do what we want, because *resetBB* is defined in terms of *untilB*. Thus, when the switch occurs, not only is the motion of the ball reset, but so too is the *Event*. Consequently, the first event occurrence will trigger the reset repeatedly, and any events thereafter will be ignored. For example, if the first event occurs after 3 seconds, then the reset will be triggered every 3 seconds, regardless of any other events. This is not what we intended. We will discuss this issue further later, as it is one of the motivating factors behind first-class signal functions. For now, we will explain how this is dealt with in CFRP.

One could imagine providing switches of both the *untilB* and *untilB′* variety. However, what CFRP variants addressing this problem do is to provide a family of *runningIn* primitives that allow *Behaviours* and *Events* to start running before they are switched in. This is achieved by fixing the start time of the *Behaviour* or *Event* such that when it is switched in its start time does not change. In effect, the *runningIn* primitives coerce *Behaviours* and *Events* from signal generators to signals, thus providing the programmer with both first-class signal generators and first-class signals. These signals (running *Behaviours* or *Events*) can then be used in the definitions of other *Behaviours* and *Events* that have not yet been switched in.

There are four functions in the *runningIn* family, one for each possible pair combination of *Event* and *Behaviour*. We first consider *runningInBB*, which starts a behaviour inside a behaviour. It has the following type and semantics:

```
runningInBB : {A B : Set} → Behaviour A → (Behaviour A → Behaviour B) → Behaviour B
runningInBB beh f ≈ λ t0 → f (λ _ → beh t0) t0
```

The first argument (*beh*) is the *Behaviour* we wish to start running. The second argument (*f*) is a function that uses this *Behaviour* (which is really a signal, despite the lack of type

distinction) to define another *Behaviour*. The semantics say that *beh* can be used in the definition of the second *Behaviour*, but that whenever *beh* is switched in, the local start time is ignored and the start time of the *runningInBB* expression is used instead.

The *runningIn* primitive that we need for our bouncing balls is *runningInEB*, which starts an *Event* inside a *Behaviour*:

$$runningInEB : \{A\ B : Set\} \to Event\ A \to (Event\ A \to Behaviour\ B) \to Behaviour\ B$$
$$runningInEB\ ev\ f \approx \lambda\ t_0 \to f\ (\lambda\ t_e \to dropWhile\ ((<\ t_e) \circ fst) \circ ev\ t_0)\ t_0$$

The semantics are similar to *runningInBB*, except that we apply $dropWhile\ ((<\ t_e) \circ fst)$ to the running *Event*. This is because the meaning of an *Event* is all event occurrences between the start time and the sample time (whereas a *Behaviour* is only concerned with the sample time). While the *Event* should start running before it is switched in, only events that occur after it is switched in should be observable.

We can now redefine *resetBB* with the behaviour we require:

$$\begin{aligned}
&resetBB' : (Ball \to Behaviour\ Ball) \to Event\ Ball \to Ball \to Behaviour\ Ball \\
&resetBB'\ f\ ev\ b\ =\ runningInEB\ ev\ (\lambda\ rev \to resetBBaux\ rev\ b) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad\quad resetBBaux : Event\ Ball \to Ball \to Behaviour\ Ball \\
&\qquad\qquad\quad resetBBaux\ rev\ b' \ =\ untilB\ (bouncingBall\ f\ b')\ (liftE\ (resetBBaux\ rev)\ rev)
\end{aligned}$$

The *bouncingBall Behaviour* is reset, but not the *Event* that triggers the resets.

3.5 First-Class Signals or First-Class Signal Functions?

The notion of a signal is absolutely central to any FRP instance. As discussed, a way to start the computation of a signal at any desired point in time, not just when the overall system starts, is key if we wish to support a dynamic system structure, both for reasons of expressivity and to avoid time and space leaks. This suggested a notion of signal generators as the central first-class abstraction at the functional level. But first-class generators alone are not enough: the ability to refer to already existing signals from within the definition of a generator is needed as well, suggesting that signals too should be first-class entities. In the overview of CFRP we encountered one particular approach for achieving this, the *runningIn* primitive, even if a signal through that particular formulation ends up being disguised as a *Behaviour* or *Event*; that is, as a signal generator. As a more recent example, Elerea [32] also provides both signals and signal generators as first-class abstractions, but this time carefully distinguished at the type level. Either way, once signals are first-class entities, signal functions come for free.

However, an alternative is to make signal functions the central first-class abstraction. They will then play the role of generators, as a signal will be generated whenever a signal function is applied to a signal, either when the system first starts or when a signal function is switched in at some later point in time. This way, the ability to make a generated signal depend on already existing signals comes for free. Thus, signals no longer have to be a first-class notion at the functional level, but can be relegated to secondary status, existing only indirectly through the signal-function abstraction. This is the approach taken by Yampa [27].

So, which option should one choose? First-class signals (and generators), or first-class signal functions? There are pros and cons to each, many related to the specifics of a particular setting (embedded or stand-alone implementation, the facilities of the host language if an embedded approach is chosen, intended application area, etc.), and some somewhat subjective. Moreover, they are not mutually exclusive; for example, Grapefruit [23] provides

first-class signal and signal function abstractions, albeit motivated by somewhat different considerations from those we outlined above.

In this paper we have chosen to develop and study a Yampa-inspired FRP variant where signal functions are the primary notion and signals are secondary. As a motivation, we conclude this section with a brief discussion on some of the advantages we think this approach offers: the nub is that making the signal-function notion primary allows for a stricter separation between the functional and reactive layers. However, this is not to say that CFRP-like approaches are not viable; recent FRP implementations [9, 14, 23, 31, 32] have shown that they are. Nor is it to say that the work in this paper applies exclusively to FRP approaches based on signal functions being the central abstraction; we reiterate that the nodes of a signal-processing network are signal functions in the sense discussed in this paper, meaning that many aspects of the present work are relevant to FRP in general.

3.5.1 Implementation Implications

Implementing first-class signals efficiently in their full generality has turned out to be very hard. The essential difficulty is that signals are *time-varying* entities occurring at the functional level where everything notionally must be *time-invariant* so as to not break referential transparency. The key to solving this apparent contradiction is to adopt the view that the signal abstraction represents the *entire* signal, which is time invariant. But this does not change the fact that signals, if space and time leaks are to be avoided, have to be *implemented* as truly time-varying values by updating them as soon as there is a change. Note that if signals are truly first-class, then they can be put into data structures or be part of closures, and be kept there for a long time without any connection to the outside world.

To our knowledge, all practically useful FRP implementations supporting first-class signals resort to imperative techniques to address this. For example, *runningIn* was implemented by updating the running *Behaviour* or *Event* as a side effect (using Haskell's *unsafePerformIO*) of consuming the produced signal (that need not depend on the running *Behaviour* or *Event* at all points of time; in fact, normally would not). For another example, Elerea [32] maintains a pool of (weak) references to all active stateful signal computations to enable all of them to be updated, regardless of whether or not the result of an individual computation is currently being used, by making a sweep over the pool at every time step.

In contrast, an approach based on signal functions can be implemented remarkably simply and purely functionally. In essence, a signal function is just a state transition function taking an input sample and current state to an output sample and new state. As the composition of such state transition functions is another state transition function, the entire system just becomes a state transition function. Signal functions themselves are *time-invariant*, so giving them first-class status at the functional level is trivial.

Another issue concerns sharing. As signal generators essentially are functions mapping a start time to a signal, the normal lazy evaluation machinery of a language like Haskell is not enough to ensure that signals generated by the same generator applied to the same start time are shared. This leads to a lot of redundant computation unless addressed, in particular for recursively defined signal generators. The usual solution is to employ some form of memoisation (again using imperative techniques). The memoisation is often done behind the scenes, as part of the abstractions; but at least one implementation, Elerea, albeit for somewhat different reasons, provides an explicit memoisation primitive as memoising everything is usually redundant and has a negative impact on performance. In contrast, with signal functions, it is easy to arrange that each signal sample is computed exactly once and distributed to where it is needed, thus avoiding any risk of lost sharing.

Of course, once everything works, what matters to an end user is not the complexity of an implementation, but the facilities provided, how easy they are to use for the purpose at hand, and how good the performance ultimately is. As to the performance of FRP implementations based on signals vs. signal functions, it is safe to say that the jury is still out: lots of research, implementation, and practical evaluation is still needed. There may not even be a simple, conclusive answer. However, we note that Yampa, despite its scalability issues, has proved to be quite efficient for many applications as witnessed by video-game implementations [12, 7] or the Yampa synthesiser [16]. We speculate that this in no small part is due to the implementation being purely functional, and functional compilers being good at compiling purely functional code. Moreover, we note that the work on causal commutative arrows [25] has shown that switch-free signal-function networks can be executed very efficiently.

3.5.2 Routing

In a language with first-class signal functions, synchronous data-flow networks can be constructed using routing combinators that operate on signal functions. This has the potential of internalising *all* routing at the reactive level, giving much greater scope for optimisation than when the routing is hidden at the host-language level. We note that Yampa, which is structured using arrows, is a half-way house in this respect because of the way the arrow framework is set up: some routing is through combinators, some takes place at the functional level. An explicit goal of our work on *N*-ary FRP is to do all routing at the reactive level. While using routing combinators is more awkward than just applying functions to arguments, we envision that syntax along the lines of Paterson's arrow notation [33] would alleviate the burden.

3.5.3 Signal-Function Objects

By making the notion of a signal function a first-class abstraction, an FRP implementer has great freedom in choosing its representation and, subsequently, in exploiting information manifest in this representation. For example, Yampa encodes simple properties about signal functions in their representation, which in favourable circumstances allows compositions of signal functions to be fused for better performance [26]. One of the goals of the present work is to identify properties of signal functions that could enable such optimisation in a more systematic and formally justifiable manner: see Section 7.

Similarly, as we have shown in earlier work [37], being able to associate additional information with signal functions *at the type level* allows certain safety guarantees, such as absence of instantaneous feedback loops, to be enforced statically. If signal functions were ordinary host-language functions on signals, it would not be possible to take such information into account if it truly relates to the function as opposed to its argument or result.

Finally, Yampa allows a switched-in signal function to be "frozen": effectively unapplied from its input signals and switched out of the network. The result is an aged version of the initial signal function; that is, its internal state at the time of being switched out is maintained. At some later point, the signal function can be switched in again. This is a powerful capability, forming the basis of Yampa's collection-based switching primitives that allow highly dynamic signal-function networks to be described. The same fundamental mechanism is also used in the virtual-reality project FRVR [4] where, through a Yampa extension, it is used to implement an undo facility by capturing the system state as frozen signal functions at various points in time. This allows interaction to resume from any saved point at a

later stage, thereby undoing the effects of any intervening interaction. It would seem hard to replicate the freezing functionality in a setting with first-class signals.

3.5.4 Other Applications

Signal functions also have applications beyond FRP, making them interesting to study in their own right. The connections to the synchronous data-flow languages and to modelling and simulation languages such as Simulink were mentioned in Section 1. Functional Hybrid Modelling (FHM) [28] is an approach to modelling and simulation, in part inspired by FRP, where signal functions are generalised to relations on signals. For efficient simulation, while still allowing dynamic structure, these relations are compiled to native simulation code using the LLVM just-in-time compiler [17]. As the notions of signal relations and signal functions are related, and as it would be desirable to have signal functions in the FHM setting, the work in this paper is potentially of use for FHM. Conversely, FHM's compilation-based implementation strategy could potentially be applied in FRP applications.

4 Signal Kinds

In Section 2 we gave a conceptual definition of signal functions that map a single signal to a single signal. We refer to FRP models with such signal functions as the central abstraction as Unary FRP. The Yampa implementation is based on single-kinded Unary FRP. While both simple and expressive, single-kinded Unary FRP has a number of inherent problems, practical as well as conceptual. In this section we review these problems, and introduce a further refined (but still high-level and general) conceptual model based on multi-kinded n-ary signal functions that we will refer to as N-ary FRP. This model will serve as the foundation for the rest of this paper.

4.1 Routing Limitations and Artificial Interdependencies

In Unary FRP, signal functions have only a single input and single output. Consequently, the only way to represent signal functions operating on, or returning, more than one signal is to exploit the fact that a product of signals is (in this model) isomorphic to a single signal carrying a product of elements of the constituent signals. For example, a signal function that maps a pair of signals carrying doubles to another pair of signals carrying doubles has type:

$$SF\ (Double, Double)\ (Double, Double)$$

This means that there is no distinction (and cannot be) between a signal that carries a "genuine" pair of values, and one that is the result of pairing two independent signals.

Moreover, exploiting this isomorphism is often the only way to route signals between signal functions: Signals are grouped together into a single signal according to the structure of signal-function composition, and then, at the functional level, values of this signal are regrouped so as to enable decomposition according to the structure of the receiving signal function.

Unfortunately, this approach hides the routing from the reactive level, and creates artificial interdependencies between independent individual signals. This makes it difficult to

implement the Unary FRP model in a way that scales well, such as through direct point-to-point communication between signal functions or minimisation of redundant computation through change propagation (see Section 8.4) [36].

The Unary FRP model certainly does not rule out all optimisation opportunities, as evidenced by the latest Yampa implementation [26]. However, overcoming these limitations in a more comprehensive and systematic way necessitates internalising the routing at the reactive level, as well as introducing *n*-ary signal functions that truly map multiple *independent* input signals to multiple *independent* output signals. Thus we take this approach in the following. However, first we need to revisit the nature of signals.

4.2 Different Kinds of Signal

As discussed in Section 2.3, most versions of FRP cater for the implementation of hybrid systems by supporting multi-kinded signals. However, we also saw that in single-kinded FRP, discrete-time signals were defined in terms of continuous-time signals by lifting the signal range using an option type. This means that there is nothing that rules out semantic infelicities such as dense event occurrences: event signals where events are always occurring, regardless of how densely the signal is sampled. This violates the conceptual model of at-most-countably many event occurrences. While such option types are typically kept abstract to prevent the programmer from accidentally creating dense events, it is nevertheless fairly easy for a "mischievous" programmer to do so deliberately. This means an implementation cannot safely carry out optimisations that are predicated on events occurring non-densely, even though that is the intent.

Another problem of single-kinded signals is that some operations need to be done in different ways on the two kinds of signal in order to maintain central properties of the signal kind in question. For example, in a typical sampled implementation, it may be necessary to insert or delete samples of continuous-time signals to mediate between different sampling rates. However, for event signals, duplicating event occurrences would often be disastrous. There may be specific versions of such operations that work correctly for events, but as any operation that works on polymorphic signals is also applicable to event signals, there is nothing to enforce that these specific operations are used in place of the generic ones.

Furthermore, we can observe that many continuous-time signals are piecewise constant (mainly due to their interaction with discrete-time signals). However, if all signals are continuous-time signals, without any further guaranteed properties, then there is not much that can be gained from this observation.

This is all in sharp contrast to multi-kinded FRP (such as CFRP) that makes a strict distinction between continuous-time and discrete-time signals, allowing the differences to be used for both gaining semantic precision and better implementation.

For reasons such as these, it is desirable to make a clear type-level distinction between different kinds of signal. To this end, we have identified three useful kinds of signal:

- Event Signals: These signals are only defined at an at-most-countable set of points in time. Each point at which an event signal is defined is known as an *event occurrence*.
- Step Signals: These signals are piecewise constant. They are always defined, but their value only changes at an at-most-countable set of points in time.
- Continuous Signals: These signals are always defined.

4.3 *N*-ary Signal Functions

To address the routing limitations of Unary FRP, and to cater for multi-kinded signals, we introduce *n*-ary signal functions, signal functions that can have more than one input or output, by defining signal functions on *signal vectors* rather than signals. Signal vectors are conceptually products of heterogeneous signals. However, note that signals do *not* nest: there are never any signals carrying signals.

Now, the crucial point is that we define the different kinds of signal, and vectors of such signals, only as an integral part of the signal-function abstraction: they have no independent existence of their own and are thus completely internalised at the reactive level. This means that the FRP implementer has great freedom in choosing representations of signals, signal functions, and the routing between them; and in exploiting those choices.

4.3.1 Signal-Vector Descriptors

We begin by defining *signal-vector descriptors*. A signal-vector descriptor is a type-level value that describes key characteristics of a signal vector. Signal-vector descriptors only exist at the type level of *N*-ary FRP, and will only be used to index signal-function types.

We are interested in the time domain and the type (of the values carried by) a signal. Thus we introduce one descriptor for each kind of signal, each parameterised on the signal type, and a pairing descriptor[7] to construct vectors of more than one signal:

```
data SVDesc : Set where
  C   : Set                    → SVDesc   -- continuous signal
  E   : Set                    → SVDesc   -- event signal
  S   : Set                    → SVDesc   -- step signal
  _,_ : SVDesc → SVDesc → SVDesc          -- product of signals
```

4.3.2 Signal Vectors

We now refine the conceptual definition of signals as follows:

- Continuous signals remain functions from time to value, as before.
- Step signals are modelled as an initial value, along with a function from time to a finite list of changes. These changes are represented as pairs of a (strictly positive) time delta and a value.
- Event signals are modelled as *Maybe* an initial event, along with a function from time to a finite list of event occurrences. These occurrences are represented as pairs of a (strictly positive) time-delta and a value.

We will refer to lists of time-delta–value pairs as *change lists*, and to functions mapping time to change lists as *change prefixes*. We also introduce the notation $Time^+$ for the set of strictly positive time, and the synonym Δt for time deltas:

$$Time^+ \approx \{t \in \mathbb{R} \mid t > 0\}$$
$$\Delta t = Time^+$$
$$ChangeList : Set \to Set$$
$$ChangeList\ A = List\ (\Delta t \times A)$$

[7] Agda notation: Infix (and more generally mixfix) functions and constructors are defined by underscores denoting the positions of the arguments.

$$\begin{aligned}&\mathit{ChangePrefix} : \mathit{Set} \to \mathit{Set}\\&\mathit{ChangePrefix}\ A = \mathit{Time} \to \mathit{ChangeList}\ A\end{aligned}$$

Signal vectors are thus defined:

$$\begin{aligned}&\mathit{SigVec} : \mathit{SVDesc} \to \mathit{Set}\\&\mathit{SigVec}\ (\mathsf{C}\ A) \quad = \mathit{Time} \to A\\&\mathit{SigVec}\ (\mathsf{E}\ A) \quad = \mathit{Maybe}\ A \times \mathit{ChangePrefix}\ A\\&\mathit{SigVec}\ (\mathsf{S}\ A) \quad = A \times \mathit{ChangePrefix}\ A\\&\mathit{SigVec}\ (as, bs) = \mathit{SigVec}\ as \times \mathit{SigVec}\ bs\end{aligned}$$

We then refine signal functions to operate over signal vectors:

$$\begin{aligned}&\mathit{SF} : \mathit{SVDesc} \to \mathit{SVDesc} \to \mathit{Set}\\&\mathit{SF}\ as\ bs \approx \mathit{SigVec}\ as \to \mathit{SigVec}\ bs\end{aligned}$$

4.3.3 Why Change Prefixes?

The desired properties of an event signal are that events occur countably and not simultaneously. The change-prefix representation enforces this as follows:

As we want our model to enforce causality, a change prefix maps a time to a finite list of changes, *up to that point in time*. Crucially, this means that at any point in time, we do not know the times or values of future events. As we require that the change list is finite, this also ensures that events are at most countable: there may be countably infinite events in the limit as time tends towards infinity, but only a finite number of events up to any specific point. We use strictly positive time deltas to ensure that events cannot occur simultaneously. However, a consequence of this is that the change list cannot represent an event at the first point in time (which we will refer to henceforth as time_0), and so we pair it with a *Maybe* value to represent the possibility of an initial event.

The definition of step signals is the same idea, but with the change list representing changes of the signal value, rather than event occurrences. Instead of the possibility of an initial event, a step signal always has an initial value.

However, there are some required properties that change prefixes do not enforce. First, we want to ensure that the change list produced by a change prefix is a prefix of all change lists produced at future sample times (intuitively, history must not be "rewritten"). We say that a change prefix is *stable* if it has this property. Second, the change list produced at any sample time must not extend beyond that sample time (intuitively, it must not "see into the future"). We say that a change prefix is *contained* if it has this property. We could incorporate these properties into the change-prefix data structure, but that would substantially complicate the definitions in this paper. Thus, we state them here as side conditions that are required to hold for all change prefixes in our model[8]:

$$\begin{aligned}&\mathit{Stable} : \{A : \mathit{Set}\} \to \mathit{ChangePrefix}\ A \to \mathit{Set}\\&\mathit{Stable}\ cp = \{t_1\ t_2 : \mathit{Time}\} \to (t_1 < t_2) \to (cp\ t_1 \equiv \mathit{takeIncl}\ t_1\ (cp\ t_2))\\&\mathit{Contained} : \{A : \mathit{Set}\} \to \mathit{ChangePrefix}\ A \to \mathit{Set}\\&\mathit{Contained}\ cp = \{t : \mathit{Time}\} \to (\mathit{lastChangeTime}\ (cp\ t) \leqslant t)\end{aligned}$$

The definitions of *takeIncl* and *lastChangeTime* can be found in Appendix A.

[8] Agda notation: We are working in dependent-type theory [29], where a proposition is represented as a type (*Set*). The elements of that type are the proofs of the proposition; thus to prove a proposition, one has to produce an inhabitant of that type. Hence *True* is represented by the unit type, and *False* by the empty type. We can refine the domain of a function by requiring a proof that the argument meets some condition. For example, $(t_1 < t_2)$ is a *type*, the elements of which are proofs that t_1 is indeed less than t_2.

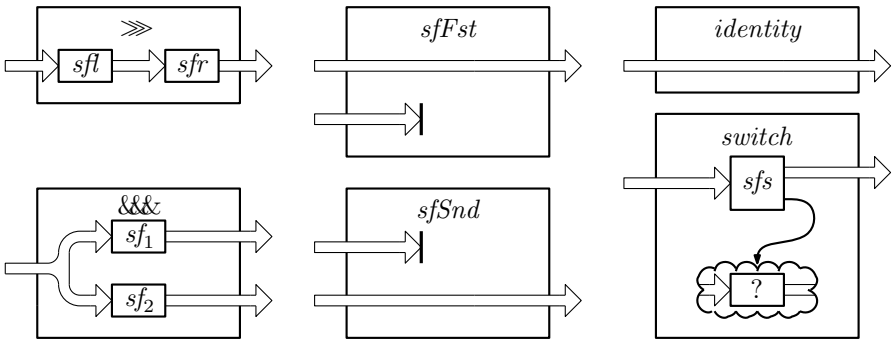

Fig. 1 Routing Primitives

5 *N*-ary FRP

An FRP language consists of a set of primitives: signal functions, signal-function combinators, and lifting functions that convert pure functions into signal functions. FRP programs are constructed by using the combinators to compose the primitive and lifted signal functions into signal-function networks. The key point about these primitives is that they only allow the construction of signal functions that respect the conceptual model.

In this section we first introduce the primitives of *N*-ary FRP, giving their definitions at the *conceptual* level. We then demonstrate *N*-ary FRP programming by defining some new combinators and signal functions using those primitives.

The utility functions we use in our definitions can be found in Appendix A. When the (conceptual) definition of a signal function is particularly verbose, we give only its type and relegate the definition to Appendix B.

5.1 Routing Primitives

As previously discussed, we wish to express all routing at the reactive level. To this end, there is a set of combinators and primitives that exist purely for routing purposes. All routing should be expressed using these primitives (as opposed to lifting routing functions from the functional level) so that an implementation can fully exploit this information.

Let us reiterate: we do not expect an *implementation* of FRP to be structured in a way that corresponds directly to the conceptual definitions below. All that is required is that the *semantics* of the implemented routing corresponds to our conceptual model.

The routing primitives can be divided into: those for *acyclic static routing*; *switching combinators*, which add the capacity for dynamism; and *feedback combinators*, which add the capacity for cyclic networks.

5.1.1 Acyclic Static Routing

We can define all acyclic static routing at the reactive level using five primitives: *identity*, *sfFst*, *sfSnd*, ⋙ and &&&. These primitives can be represented graphically, as shown in Fig. 1. This set of primitives is minimal in the sense that any acyclic static network structure can be described by them, yet none of these primitives can be defined in terms of the other

four. There are of course other sets of minimal combinators that can likewise express such routing. In Section 5.4.1 we demonstrate the expressiveness of these primitives by using them to define several other routing combinators.

We begin with the identity signal function:

$$identity : \{as : SVDesc\} \to SF\ as\ as$$
$$identity \approx \lambda\ as \to as$$

The projection signal functions extract the first or second component of a signal vector:

$$sfFst : \{as\ bs : SVDesc\} \to SF\ (as, bs)\ as$$
$$sfFst \approx fst$$
$$sfSnd : \{as\ bs : SVDesc\} \to SF\ (as, bs)\ bs$$
$$sfSnd \approx snd$$

The sequential-composition combinator composes two signal functions:

$$\_ \ggg \_ : \{as\ bs\ cs : SVDesc\} \to SF\ as\ bs \to SF\ bs\ cs \to SF\ as\ cs$$
$$sf_1 \ggg sf_2 \approx sf_2 \circ sf_1$$

The fan-out combinator applies two signal functions to the same input in parallel:

$$\_\&\&\&\_ : \{as\ bs\ cs : SVDesc\} \to SF\ as\ bs \to SF\ as\ cs \to SF\ as\ (bs, cs)$$
$$sf_1 \&\&\& sf_2 \approx \lambda\ as \to (sf_1\ as, sf_2\ as)$$

5.1.2 Switching

One of the main things that sets FRP apart from the synchronous data-flow languages is its highly dynamic nature. Yampa, for example, provides a family of switching combinators that operate on signal functions. New first-class signal functions can be created, and first-class signal functions can be switched in to replace running signal functions. Running signal functions can be "frozen" (transformed back into first-class entities, maintaining any internal state), removed from the network, and then later switched in again if desired. [27]

There is a similar family of switching combinators in *N*-ary FRP, but in this paper we present only one as a primitive:

$$switch : \{as\ bs : SVDesc\} \to \{A : Set\} \to SF\ as\ (bs, \mathsf{E}\ A) \to (A \to SF\ as\ bs) \to SF\ as\ bs$$

Informally, the behaviour of *switch* is to apply the subordinate signal function (the first explicit argument) to the input signal. The first component of the subordinate's output is emitted as the overall output until there is an occurrence in the event signal (the second component of the subordinate's output). The switching function (the second explicit argument) is then applied to the value of that event to produce a residual signal function. This residual signal function is then applied to the input signal, starting at the time of the event occurrence, and henceforth the overall output is taken from the residual signal function.

The formal definition of *switch* over signal vectors is somewhat involved in our conceptual model, but to give the idea we define the specialised case of one continuous output signal. The full definition can be found in Appendix B.

$$switchC : \{as : SVDesc\} \to \{A\ B : Set\} \to SF\ as\ (\mathsf{C}\ B, \mathsf{E}\ A) \to (A \to SF\ as\ (\mathsf{C}\ B)) \to SF\ as\ (\mathsf{C}\ B)$$
$$switchC\ sf\ f \approx \lambda\ s_a\ t \to \mathbf{let}\ (s_b, s_e) = sf\ s_a$$
$$\mathbf{in\ case}\ fstOcc\ s_e\ t\ \mathbf{of}$$
$$\quad \mathsf{nothing} \to s_b\ t$$
$$\quad \mathsf{just}\ (t_e, e) \to (f\ e)\ (advance\ t_e\ s_a)\ (t - t_e)$$

The key point here is that the residual signal function ($f\ e$) only "starts" at the moment of switching. Thus we have to "advance" (time-shift) the input signal so that the residual signal function can only observe the input signal after the switch occurs. Furthermore, we modify the sampling time to match this time shifting, so that the residual signal function does not examine the future of the advanced signal.

Note that this differs from the CFRP switching combinators (see Section 3.3), which had signal generators and start times. Here, each signal function is running in its own *local time*, and thus always starts at (local) $time_0$.

Definition 2 (Local Time) The time since a signal function was applied to its input signal. This will have been either when the entire system started, or when the sub-network containing the signal function in question was switched in.

5.1.3 Feedback

An important facility in FRP (and synchronous data-flow generally) is to be able to introduce feedback into a network. However, when doing so, one has to be careful not to introduce ill-defined feedback that could cause an implementation to loop at run-time. Ideally, one wants a language that disallows ill-defined feedback, without enforcing conservative restrictions on the well-defined feedback.

We have considered this issue in earlier work [37], but for the purposes of this paper we will concentrate only on acyclic networks, and so do not give any feedback combinators. We discuss feedback further in Section 10.

5.2 Primitive Signal Functions

In this section we give the primitive signal functions, along with their conceptual definitions. In some cases, the definition of a signal function differs when applied to (or producing) different kinds of signals. We define a separate version of the signal function for each such signal kind. In an implementation, some form of overloading mechanism would probably be employed on top of these.

We begin with a signal function that emits constant output:

$$constantS : \{as : SVDesc\} \to \{A : Set\} \to A \to SF\ as\ (\mathsf{S}\ A)$$
$$constantS\ a \approx const\ (a, const\ [\,])$$

Note that *constantS* is polymorphic in its input signal-vector descriptor. As discussed in Section 2.2, this is the way of embedding what are really signal generators into signal functions. That is, *constantS* is a generator of constant step signals.

Similarly, the primitives *never* and *now* are also embedded event-signal generators:

- *never* generates an event signal with no event occurrences;
- *now* generates an event signal containing exactly one event occurrence at $time_0$.

$$never : \{as : SVDesc\} \to \{A : Set\} \to SF\ as\ (\mathsf{E}\ A)$$
$$never \approx const\ (\mathsf{nothing}, const\ [\,])$$
$$now : \{as : SVDesc\} \to SF\ as\ (\mathsf{E}\ Unit)$$
$$now \approx const\ (\mathsf{just\ unit}, const\ [\,])$$

We can mediate between event and step signals using *hold* and *edge*:

– *hold* emits a step signal carrying the value of its most recent input event;
– *edge* emits an event whenever the value of the Boolean input step signal changes from false to true:

$$
\begin{aligned}
&\mathit{hold} : \{A : \mathit{Set}\} \to A \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{S}\ A)\\
&\mathit{hold}\ a \approx \mathit{first}\ (\mathit{fromMaybe}\ a)\\
&\mathit{edge} : \mathit{SF}\ (\mathsf{S}\ \mathit{Bool})\ (\mathsf{E}\ \mathit{Unit})\\
&\mathit{edge} \approx \lambda\ (b, cp) \to (\mathsf{nothing}, \mathit{edgeAux}\ b \circ cp)\\
&\quad \textbf{where}\\
&\qquad \mathit{edgeAux} : \mathit{Bool} \to \mathit{ChangeList\ Bool} \to \mathit{ChangeList\ Unit}\\
&\qquad \mathit{edgeAux}\ \_\quad [\,] \qquad\qquad\qquad\quad = [\,]\\
&\qquad \mathit{edgeAux}\ \mathsf{true}\ ((\_, b) \quad :: \delta bs) = \mathit{edgeAux}\ b\ \delta bs\\
&\qquad \mathit{edgeAux}\ \mathsf{false}\ ((\_, \mathsf{false}) :: \delta bs) = \mathit{edgeAux}\ \mathsf{false}\ \delta bs\\
&\qquad \mathit{edgeAux}\ \mathsf{false}\ ((\delta, \mathsf{true}) :: \delta bs) = (\delta, \mathsf{unit}) :: \mathit{edgeAux}\ \mathsf{true}\ \delta bs
\end{aligned}
$$

We can integrate a real-valued step or continuous signal with respect to time. The output of such an integration is always a continuous signal. Note that while an implementation will only be able to approximate the integral of an arbitrary continuous signal, it can compute the exact[9] integral of a step signal:

$$
\begin{aligned}
&\mathit{integralS} : \mathit{SF}\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\\
&\mathit{integralC} : \mathit{SF}\ (\mathsf{C}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\\
&\mathit{integralC} \approx \lambda\ s\ t_1 \to \textstyle\int_0^{t_1} (s\ t)\ dt
\end{aligned}
$$

The signal function *when* applies a predicate to a continuous input signal, producing an event occurrence as output whenever that predicate changes from false to true. Note that, as with *edge*, this is only at the moment of change: another event will not occur until the predicate has ceased to hold and then become true again.

$$\mathit{when} : \{A : \mathit{Set}\} \to (A \to \mathit{Bool}) \to \mathit{SF}\ (\mathsf{C}\ A)\ (\mathsf{E}\ A)$$

The *delay* primitives delay a signal vector by a specified amount of time. Note that in the case of continuous and step signals, we have to initialise the signal for the delay period:

$$
\begin{aligned}
&\mathit{delayE} : \{A : \mathit{Set}\} \to \mathit{Time}^+ \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\\
&\mathit{delayE}\ d \approx \lambda\ (ma, cp) \to (\mathsf{nothing}, \mathit{delayCP}\ d\ ma\ cp)\\
&\mathit{delayS} : \{A : \mathit{Set}\} \to \mathit{Time}^+ \to A \to \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{S}\ A)\\
&\mathit{delayS}\ d\ a_0 \approx \lambda\ (a_1, cp) \to (a_0, \mathit{delayCP}\ d\ (\mathsf{just}\ a_1)\ cp)\\
&\mathit{delayC} : \{A : \mathit{Set}\} \to \mathit{Time}^+ \to (\mathit{Time} \to A) \to \mathit{SF}\ (\mathsf{C}\ A)\ (\mathsf{C}\ A)\\
&\mathit{delayC}\ d\ f \approx \lambda\ s\ t \to \textbf{if}\ t < d\ \textbf{then}\ f\ t\ \textbf{else}\ s\ (t - d)
\end{aligned}
$$

Finally, to allow us to combine step and continuous signals, we provide a coercion signal function that converts a step signal to a continuous signal:

$$
\begin{aligned}
&\mathit{fromS} : \{A : \mathit{Set}\} \to \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{C}\ A)\\
&\mathit{fromS} \approx \mathit{valS}
\end{aligned}
$$

5.3 Lifting Functions

There is a family of lifting functions that allow us to lift pure functions from the functional level to the reactive level in a pointwise fashion:

[9] Up to the limit of the underlying numeric representation, typically floating-point numbers.

$$liftC : \{A\ B : Set\} \to (A \to B) \to SF\ (\mathsf{C}\ A)\ (\mathsf{C}\ B)$$
$$liftC\ f \approx mapC\ f$$
$$liftS : \{A\ B : Set\} \to (A \to B) \to SF\ (\mathsf{S}\ A)\ (\mathsf{S}\ B)$$
$$liftS\ f \approx mapS\ f$$
$$liftE : \{A\ B : Set\} \to (A \to B) \to SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ B)$$
$$liftE\ f \approx mapE\ f$$
$$liftC2 : \{A\ B\ Z : Set\} \to (A \to B \to Z) \to SF\ (\mathsf{C}\ A, \mathsf{C}\ B)\ (\mathsf{C}\ Z)$$
$$liftC2\ f \approx uncurry\ (mapC2\ f)$$
$$liftS2 : \{A\ B\ Z : Set\} \to (A \to B \to Z) \to SF\ (\mathsf{S}\ A, \mathsf{S}\ B)\ (\mathsf{S}\ Z)$$
$$liftS2\ f \approx uncurry\ (mapS2\ f)$$

We have deliberately omitted *liftE2*, because the intended meaning of such a combinator is not obvious. Consider: there are two input event signals, and one output event signal. At any point in time, if there is no occurrence on either input signal, then there shouldn't be an occurrence on the output signal. And if there are event occurrences on both input signals, then it seems reasonable that there should be an event occurrence on the output. But what about the case when there is an event occurrence on one input signal and not the other? Should there be an event occurrence on the output or not?

To address this question, we define two separate primitives: *merge* and *join*. The behaviour of *merge* is to produce an event occurrence when either input has an occurrence; the behaviour of *join* is to produce an event only when both inputs have an event occurrence:

$$merge : \{A\ B\ Z : Set\} \to (A \to Z) \to (B \to Z) \to (A \to B \to Z) \to SF\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)$$
$$merge\ fa\ fb\ fab \approx uncurry\ (mergeE2\ fa\ fb\ fab)$$
$$join : \{A\ B\ Z : Set\} \to (A \to B \to Z) \to SF\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)$$
$$join\ f \approx uncurry\ (joinE2\ f)$$

Finally, *sampleWith* merges event signals with continuous or step signals, producing an output event occurrence exactly when there is an occurrence on the input event signal:

$$sampleWithC : \{A\ B\ Z : Set\} \to (A \to B \to Z) \to SF\ (\mathsf{C}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)$$
$$sampleWithC\ f \approx uncurry\ (mapCE\ f)$$
$$sampleWithS : \{A\ B\ Z : Set\} \to (A \to B \to Z) \to SF\ (\mathsf{S}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)$$
$$sampleWithS\ f \approx uncurry\ (mapSE\ f)$$

5.4 Examples

Having introduced the primitives of *N*-ary FRP, we will now write some *N*-ary FRP programs using those primitives. We are no longer working at the conceptual level: thus signal functions are now abstract, and signals do not exist.

5.4.1 Additional Combinators

We begin by defining some useful routing combinators (shown in Fig. 2):

$$toFst : \{as\ bs\ cs : SVDesc\} \to SF\ as\ cs \to SF\ (as, bs)\ cs$$
$$toFst\ sf = sfFst \ggg sf$$
$$toSnd : \{as\ bs\ cs : SVDesc\} \to SF\ bs\ cs \to SF\ (as, bs)\ cs$$
$$toSnd\ sf = sfSnd \ggg sf$$
$$\_ *** \_ : \{as\ bs\ cs\ ds : SVDesc\} \to SF\ as\ cs \to SF\ bs\ ds \to SF\ (as, bs)\ (cs, ds)$$
$$sf_1 *** sf_2 = toFst\ sf_1\ \&\&\&\ toSnd\ sf_2$$

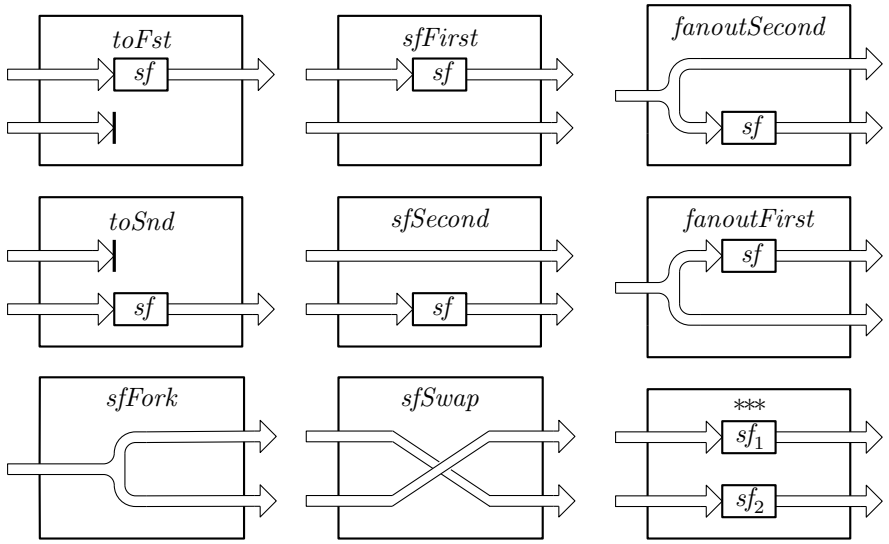

Fig. 2 Additional routing combinators

```
sfFirst : {as bs cs : SVDesc} → SF as bs → SF (as,cs) (bs,cs)
sfFirst sf = sf *** identity
sfSecond : {as bs cs : SVDesc} → SF bs cs → SF (as,bs) (as,cs)
sfSecond sf = identity *** sf
sfFork : {as : SVDesc} → SF as (as,as)
sfFork = identity &&& identity
sfSwap : {as bs : SVDesc} → SF (as,bs) (bs,as)
sfSwap = sfSnd &&& sfFst
fanoutFirst : {as bs : SVDesc} → SF as bs → SF as (bs,as)
fanoutFirst sf = sf &&& identity
fanoutSecond : {as bs : SVDesc} → SF as bs → SF as (as,bs)
fanoutSecond sf = identity &&& sf
```

We also define a switching combinator called *switchWhen* that will be convenient later:

```
switchWhen : {as bs : SVDesc} → {A : Set}
             → SF as bs → SF bs (E A) → (A → SF as bs) → SF as bs
switchWhen sf sfe = switch (sf >>> fanoutSecond sfe)
```

Roughly, *switchWhen* is the same as *switch*, except the subordinate signal function has been split into two: one to produce the output and one to produce the event. You can consider *switchWhen* to be a *switch* specialised to the case where:

- the event that causes the switch to occur only depends on the output of the subordinate signal function, and
- the output of the subordinate signal function does not depend on the event.

Notice that we did not use the functional level in any of these definitions—the set of routing primitives is sufficient. This is key: as discussed in Section 3.5.2, it is one of our objectives to express all routing at the reactive level.

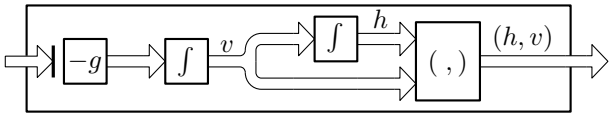

Fig. 3 A signal-function network modelling falling balls

5.4.2 Simple Signal Functions

Let us construct some simple signal-function networks. First we define a signal function that outputs the current local time by integrating the constant 1:

```
localTime : {as : SVDesc} → SF as (C ℝ)
localTime = constantS 1 >>> integralS
```

Next we define a signal function that emits a single event after a specified amount of time has passed:

```
after : {as : SVDesc} → Time → SF as (E Unit)
after t = now >>> delayE t
```

When working with continuous signals, it is useful to produce constant continuous signals:

```
constantC : {as : SVDesc} → {A : Set} → A → SF as (C A)
constantC a = constantS a >>> fromS
```

Finally, we define initialised versions of the integration signal functions:

```
iIntegralS : ℝ → SF (S ℝ) (C ℝ)
iIntegralS x = integralS >>> liftC (+ x)
iIntegralC : ℝ → SF (C ℝ) (C ℝ)
iIntegralC x = integralC >>> liftC (+ x)
```

5.4.3 Bouncing Balls Revisited

The example signal-function networks we have seen so far have been static in structure. As an example of a dynamic network, we will revisit the bouncing balls example from Section 3.4. We will reuse the definitions from the functional level (i.e. those that do not contain *Events* or *Behaviours*), but replace the reactive level definitions (i.e. those that do).

We begin by defining a signal function to model a falling ball:

```
falling : {as : SVDesc} → Ball → SF as (C Ball)
falling (h, v) = constantS (−g) >>> iIntegralS v >>> fanoutFirst (iIntegralC h) >>> liftC2 (,)
```

For this signal function, the code is less clear than its CFRP equivalent. You may find the graphical representation in Fig. 3 helpful. In Yampa, Paterson's arrow notation [33] is used for signal functions such as this to make the code clearer [27] (and we would expect an implementation of *N*-ary FRP to provide similar notation). However, the definitions of *bouncingBall*, *elasticBall* and *inelasticBall* closely follow those of CFRP:

```
detectBounce : SF (C Ball) (E Ball)
detectBounce = when detectImpact
```

$$bouncingBall : \{as : SVDesc\} \to (Ball \to SF\ as\ (\mathsf{C}\ Ball)) \to Ball \to SF\ as\ (\mathsf{C}\ Ball)$$
$$bouncingBall\ f\ b = switchWhen\ (falling\ b)\ detectBounce\ f$$
$$elasticBall : \{as : SVDesc\} \to Ball \to SF\ as\ (\mathsf{C}\ Ball)$$
$$elasticBall = bouncingBall\ (elasticBall \circ negateVel)$$
$$inelasticBall : \{as : SVDesc\} \to Ball \to SF\ as\ (\mathsf{C}\ Ball)$$
$$inelasticBall = bouncingBall\ (\lambda\_ \to constantC\ (0,0))$$

Finally, we add the capacity for the ball to be reset:

$$resetBB : (Ball \to SF\ (\mathsf{E}\ Ball)\ (\mathsf{C}\ Ball)) \to Ball \to SF\ (\mathsf{E}\ Ball)\ (\mathsf{C}\ Ball)$$
$$resetBB\ f\ b = switch\ (fanoutFirst\ (bouncingBall\ f\ b))\ (resetBB\ f)$$

This signal function is much easier to define in *N*-ary FRP than CFRP. Primarily, this is because of the modular nature of signal functions. They are parameterised on their input, and so we can define a signal function that receives input from outside the *switch*, thereby allowing us to retain (rather than resetting) the input signal when the switch occurs. In CFRP the *runningIn* primitive was required to achieve this.

6 FRP Optimisation

In order to be reactive (delivering timely responses), any FRP implementation must be discretely sampled. Consequently, if notionally continuous-time signals are provided, a concrete implementation can only approximate the ideal semantics.

However, we aim to make the approximation as faithful as possible. Here, the semantic distinction between different kinds of signals helps. As discussed in Section 4.2, we can statically rule out certain uses of signals by making the kinds manifest in the type system. This allows employing an implementation strategy that is appropriate for a specific kind of signal, but which would have risked breaking the abstractions had said uses not been ruled out. Moreover, this also opens up opportunities for signal-kind–specific optimisations.

In this section, we first briefly review the two basic FRP implementation strategies. We then discuss the archetypal optimisation opportunities we would like to identify in an FRP system, as a background and motivation for the signal function properties in the next section.

6.1 Basic FRP Implementation Strategies

An FRP instance typically employs either a *pull-based* (demand-driven) or *push-based* (data-driven) implementation approach [14].

A pull-based approach repeatedly samples the output signals over a sequence of time steps, recomputing every signal at each step. This is a good approach for signals that change often, as is common for continuous-time signals. In fact, the more frequent the changes, the more efficient this approach. However, signals that change only rarely have their value unnecessarily recomputed repeatedly. This is inefficient and scales poorly, as the amount of work is proportional to the number of signals, not to the signal activity.

In contrast, a push-based approach only recomputes signals when the signals they depend on change. This is a natural fit for discrete-time signals: when nothing changes, no updates are needed. However, in FRP there are signals that depend on time (and can thus change even if their inputs do not), as well as continuous-time signals that change frequently, often at every sample step. The former implies that just reacting to external events is not

enough. The second, that there can be a substantial overhead for using a push-based approach for continuous-time signals as the costs associated with pushing are incurred very frequently.

Ideally, one would like to employ both strategies selectively, to reap the benefits of each. A solid understanding of how change works in signal function networks is a good first step in that direction.

6.2 Optimisation Opportunities

For most networks, many signals will be unchanging for significant periods of time, with changes occurring sparsely compared to the sampling rate. We would like an implementation to be able to optimise as much as possible based on this, without breaking any abstractions.

There are three archetypal ways in which we can optimise a signal function network:

- Eliminate any signal function whose output is not used.
- Avoid recomputation of signals whose values will never change.
- Apply a change-propagation execution scheme (where valid).

For the first, we need to keep track of which signals are used. Given the dynamic nature of FRP, and the combinator style used to construct networks, it is very common for signals to be used for a while, but then later ignored. Often, the signal function that produces them is still in the network, and consequently being executed. To allow such signal functions to be garbage collected we need to track signal functions that do not use their inputs, as from that we can determine which signals are not used.

For the second, we need to know that a signal will not change at any point in the future. If that is the case, then we can just compute the value of the signal once, and then employ constant propagation.

For the third, we need to identify which signal functions are such that their output will not change unless their input does. Then, when executing, if we know that the input has not changed, we can avoid recomputing the output.

By identifying the three different kinds of signals, we can more precisely track the properties of signals and signal functions, and hence are able to apply more precise optimisations. For example, we know that continuous signals are likely to be always changing, no matter how rapid the sampling rate. On the other hand, step signals tend to change only sparsely, and are thus likely to benefit greatly from change-propagation optimisations. By combining this knowledge with knowledge about how signal functions are affected by change, it becomes possible to select appropriate implementation and optimisation strategies in a fine-grained manner.

7 Properties of Signals and Signal Functions

In this section we define some properties of signals and signal functions that could be exploited by an implementation to enable the optimisations suggested in Section 6.2. As many of the definitions in this section refer to time-varying properties, we first introduce some combinators from *temporal logic* [22,38] to simplify the definitions.

7.1 Temporal Logic

First we introduce a type of temporal predicate:

$$\mathit{TimePred} = \mathit{Time} \to \mathit{Set}$$

We then define pointwise versions of the standard logical operators[10]:

$$\_\vee\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$(\varphi \vee \psi)\, t \;\; = \varphi\, t \uplus \psi\, t$$
$$\_\wedge\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$(\varphi \wedge \psi)\, t \;\; = \varphi\, t \times \psi\, t$$
$$\_\Rightarrow\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$(\varphi \Rightarrow \psi)\, t = \varphi\, t \to \psi\, t$$
$$\neg\_ : \mathit{TimePred} \to \mathit{TimePred}$$
$$(\neg\, \varphi)\, t = \varphi\, t \to \mathit{False}$$

We now introduce two unary temporal operators: **G** (*Global*) and **H** (*History*). They should be read as "at all points in the future" and "at all points in the past", respectively:

$$\mathbf{G} : \mathit{TimePred} \to \mathit{TimePred}$$
$$(\mathbf{G}\, \varphi)\, t = (t' : \mathit{Time}) \to (t' > t) \to \varphi\, t'$$
$$\mathbf{H} : \mathit{TimePred} \to \mathit{TimePred}$$
$$(\mathbf{H}\, \varphi)\, t = (t' : \mathit{Time}) \to (t' < t) \to \varphi\, t'$$

Notice that the definitions of **G** and **H** exclude the current time. However, we can define reflexive variants that include the current time:

$$\mathbf{G^r} : \mathit{TimePred} \to \mathit{TimePred}$$
$$\mathbf{G^r}\, \varphi = \varphi \wedge \mathbf{G}\, \varphi$$
$$\mathbf{H^r} : \mathit{TimePred} \to \mathit{TimePred}$$
$$\mathbf{H^r}\, \varphi = \varphi \wedge \mathbf{H}\, \varphi$$

We next introduce the synonym *Between*, as an aid to defining further temporal operators. *Between* $t_0\ \varphi\ t_1$ should be taken to mean that φ holds over the interval (t_0, t_1):

$$\mathit{Between} : \mathit{Time} \to \mathit{TimePred} \to \mathit{Time} \to \mathit{Set}$$
$$\mathit{Between}\, t_0\, \varphi\, t_1 = (t : \mathit{Time}) \to (t_0 < t) \to (t < t_1) \to \varphi\, t$$

We will now define some binary temporal operators. We begin with **S** (*Since*). φ **S** ψ should be read as "φ since ψ", and should be taken to mean that there is a point in the past at which ψ held, and φ has held since that point:

$$\_\mathbf{S}\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$(\varphi\ \mathbf{S}\ \psi)\, t = \Sigma\, \mathit{Time}\, (\lambda\, t' \to (t' < t) \times \psi\, t' \times \mathit{Between}\, t'\, \varphi\, t)$$

Our next binary operator is **W** (*Wait For* or *Weak Until*). φ **W** ψ should be read as "φ waits for ψ", and should be taken to mean that φ will hold until ψ holds:

$$\_\mathbf{W}\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$(\varphi\ \mathbf{W}\ \psi)\, t = \mathbf{G}\, (\mathit{Between}\, t\, (\neg\, \psi) \Rightarrow \mathit{Between}\, t\, \varphi)\, t$$

Note that the definition of **S** requires that ψ must have held at some point in the past, whereas **W** does not require ψ to ever hold.

[10] Agda notation: $\uplus$ is the Agda sum type, see Appendix A.

Our final operator is a reflexive variant of **W** that requires φ to hold at the current time if ψ does not:

$$\_\mathbf{W^r}\_ : \mathit{TimePred} \to \mathit{TimePred} \to \mathit{TimePred}$$
$$\varphi\ \mathbf{W^r}\ \psi\ =\ \psi \vee (\varphi \wedge (\varphi\ \mathbf{W}\ \psi))$$

We now define some functions for introducing and eliminating temporal predicates. We can eliminate a temporal predicate by requiring it to hold at all points in time:

$$\mathit{Always} : \mathit{TimePred} \to \mathit{Set}$$
$$\mathit{Always}\ \varphi\ =\ (t : \mathit{Time}) \to \varphi\ t$$

We can introduce a temporal predicate by requiring the result of a time function to equal its result at a specified point in time:

$$\mathit{EqualAt} : \{A : \mathit{Set}\} \to (\mathit{Time} \to A) \to \mathit{Time} \to \mathit{TimePred}$$
$$\mathit{EqualAt}\ f\ t'\ t\ =\ f\ t \equiv f\ t'$$

7.2 Change

As previously discussed, many of our optimisations rely on some notion of *change*. However, most obvious definitions of change are implementation specific. In a sampled implementation, an obvious definition would be to say that a signal has changed if its current sample differs from its previous sample. This would make sense for continuous signals, but not for events which are supposed to occur in isolation. Two adjacent identical event occurrences should be two changes, not a lack of change. This is also specialised to the implementation; in our conceptual model there is no notion of time samples.

Consequently, we use a more precise definition of change that respects the conceptual model of multi-kinded signals:

Definition 3 (Signal Change) At any given point in time, a signal is either *changing* or *unchanging* (exclusively):

- A continuous signal is *unchanging* at a time t iff there exists a non-empty closed interval bounded to the right by t over which the signal remains constant.
- An event signal is *unchanging* at all points in time at which there is no event occurrence.
- A step signal is *changing* at time_0 (see Section 8.5) and at all points in time at which it assumes a new value. It is *unchanging* at all other points.
- A signal vector is *unchanging* if all signals in that vector are *unchanging*.

$$\mathit{UnchangingC} : \{A : \mathit{Set}\} \to \mathit{SigVec}\ (\mathsf{C}\ A) \to \mathit{TimePred}$$
$$\mathit{UnchangingC}\ s\ t\ =\ ((\mathit{EqualAt}\ s\ t)\ \mathbf{S}\ (\mathit{EqualAt}\ s\ t))\ t$$

$$\mathit{UnchangingE} : \{A : \mathit{Set}\} \to \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{TimePred}$$
$$\begin{aligned}\mathit{UnchangingE}\ (ma, cp)\ t \mid\ & t == 0 = \mathit{IsNothing}\ ma\\ \mid\ & t > 0 = \mathit{IsNothing}\ (\mathit{lookupCP}\ cp\ t)\end{aligned}$$

$$\mathit{UnchangingS} : \{A : \mathit{Set}\} \to \mathit{SigVec}\ (\mathsf{S}\ A) \to \mathit{TimePred}$$
$$\begin{aligned}\mathit{UnchangingS}\ (\_, cp)\ \ t \mid\ & t == 0 = \mathit{False}\\ \mid\ & t > 0 = \mathit{IsNothing}\ (\mathit{lookupCP}\ cp\ t)\end{aligned}$$

$$\mathit{Unchanging} : \{as : \mathit{SVDesc}\} \to \mathit{SigVec}\ as \to \mathit{TimePred}$$
$$\begin{aligned}\mathit{Unchanging}\ \{\mathsf{C}\ \_\}\ s &= \mathit{UnchangingC}\ s\\ \mathit{Unchanging}\ \{\mathsf{E}\ \_\}\ s &= \mathit{UnchangingE}\ s\\ \mathit{Unchanging}\ \{\mathsf{S}\ \_\}\ s &= \mathit{UnchangingS}\ s\\ \mathit{Unchanging}\ \{\_,\_\}\ (s_1, s_2) &= \mathit{Unchanging}\ s_1 \wedge \mathit{Unchanging}\ s_2\end{aligned}$$

$$\begin{aligned}&\mathit{Changing} : \{\mathit{as} : \mathit{SVDesc}\} \to \mathit{SigVec\ as} \to \mathit{TimePred}\\&\mathit{Changing}\ s\ =\ \neg\,(\mathit{Unchanging}\ s)\end{aligned}$$

The definition for *UnchangingC* can be read as: "There was a point in the past such that the value of the signal was equal to the current value of the signal, and since that point they have remained equal."

Having defined what it means for a signal to be *unchanging* at an arbitrary point in time, we can use the temporal-logic combinators to express the property that a signal will remain *unchanging* henceforth. We define two variants of this property, one reflexive, one not:

Definition 4 (Changeless Signal) We say that a signal is *changeless* if it will be *unchanging* at all future points in time:

$$\begin{aligned}&\mathit{Changeless} : \{\mathit{as} : \mathit{SVDesc}\} \to \mathit{SigVec\ as} \to \mathit{TimePred}\\&\mathit{Changeless}\ s\ =\ \mathbf{G}\ (\mathit{Unchanging}\ s)\end{aligned}$$

Definition 5 (Reflexively Changeless Signal) We say that a signal is *reflexively changeless* if it is *unchanging* now, and will remain *unchanging* henceforth:

$$\begin{aligned}&\mathit{Changeless}^r : \{\mathit{as} : \mathit{SVDesc}\} \to \mathit{SigVec\ as} \to \mathit{TimePred}\\&\mathit{Changeless}^r\ s\ =\ \mathbf{G}^{\mathbf{r}}\ (\mathit{Unchanging}\ s)\end{aligned}$$

7.3 Signal-Function Properties

In this section we define some properties of signal functions that are useful for optimisation. We will define both time-invariant properties (those that will always hold), as well as time-varying properties. The former are suited to static optimisation, whereas the latter present opportunities for dynamic optimisation.

7.3.1 Time-Varying Equality

To express some of the properties in this section we will require a pointwise temporal equality of signal vectors. To achieve this, we define a *Sample* of a signal vector, which represents its value at a single point in time:

$$\begin{aligned}&\mathit{Sample} : \mathit{SVDesc} \to \mathit{Set}\\&\mathit{Sample}\ (\mathsf{C}\ A) \ \ =\ A\\&\mathit{Sample}\ (\mathsf{E}\ A) \ \ =\ \mathit{Maybe}\ A\\&\mathit{Sample}\ (\mathsf{S}\ A) \ \ =\ A\\&\mathit{Sample}\ (\mathit{as}, \mathit{bs}) \ =\ \mathit{Sample\ as} \times \mathit{Sample\ bs}\end{aligned}$$

We then define a function *at* that computes this sample, given a time point:

$$\begin{aligned}&\mathit{at} : \{\mathit{as} : \mathit{SVDesc}\} \to \mathit{SigVec\ as} \to \mathit{Time} \to \mathit{Sample\ as}\\&\mathit{at}\ \{\mathsf{C}\ \_\}\ s \qquad t\ =\ s\ t\\&\mathit{at}\ \{\mathsf{S}\ \_\}\ s \qquad t\ =\ \mathit{valS}\ s\ t\\&\mathit{at}\ \{\mathsf{E}\ \_\}\ s \qquad t\ =\ \mathit{occ}\ s\ t\\&\mathit{at}\ \{\_,\_\}\ (s_1, s_2)\ t\ =\ (\mathit{at}\ s_1\ t, \mathit{at}\ s_2\ t)\end{aligned}$$

The pointwise equality that we need is then defined as a temporal predicate that holds at any given point in time if the samples of the two vectors are equal:

$$\begin{aligned}&\_\equiv_s\_ : \{\mathit{as} : \mathit{SVDesc}\} \to \mathit{SigVec\ as} \to \mathit{SigVec\ as} \to \mathit{TimePred}\\&(s_1 \equiv_s s_2)\ t\ =\ \mathit{at}\ s_1\ t \equiv \mathit{at}\ s_2\ t\end{aligned}$$

7.3.2 Time-Invariant Properties

To begin, we formally define the causality property from Section 2.2:

$$\begin{aligned}&\mathit{Causal} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{Set}\\ &\mathit{Causal}\ \{\mathit{as}\}\ \mathit{sf} = (s_1\ s_2 : \mathit{SigVec}\ \mathit{as}) \to \mathit{Always}\ (\mathbf{H}^{\mathbf{r}}\ (s_1 \equiv_s s_2) \Rightarrow (\mathit{sf}\ s_1 \equiv_s \mathit{sf}\ s_2))\end{aligned}$$

Some signal functions are such that their output at any point in time only depends on their input at that same point in time. These are known as *stateless* signal functions:

Definition 6 (Stateless Signal Function) A signal function is *stateless* if, at any given time, its output can depend upon its current input, but not its past or future inputs:

$$\begin{aligned}&\mathit{Stateless} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{Set}\\ &\mathit{Stateless}\ \{\mathit{as}\}\ \mathit{sf} = (s_1\ s_2 : \mathit{SigVec}\ \mathit{as}) \to \mathit{Always}\ ((s_1 \equiv_s s_2) \Rightarrow (\mathit{sf}\ s_1 \equiv_s \mathit{sf}\ s_2))\end{aligned}$$

Signal functions are often implemented as having an internal state, in which they store any required information about past inputs. Thus the system does not have to record globally all past signal information; each signal function will store what it requires. It is this common implementation choice that leads to the name *stateful* for signal functions that require such a state, and *stateless* for those that do not. In other settings, the terms *sequential* and *combinatorial* are used for the same notions, respectively.

Another interesting property of signal functions is whether they are *decoupled* or not:

Definition 7 (Decoupled Signal Function) A signal function is *decoupled* if, at any given time, its output can depend upon its past inputs, but not its present and future inputs:

$$\begin{aligned}&\mathit{Decoupled} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{Set}\\ &\mathit{Decoupled}\ \{\mathit{as}\}\ \mathit{sf} = (s_1\ s_2 : \mathit{SigVec}\ \mathit{as}) \to \mathit{Always}\ (\mathbf{H}\ (s_1 \equiv_s s_2) \Rightarrow (\mathit{sf}\ s_1 \equiv_s \mathit{sf}\ s_2))\end{aligned}$$

A *decoupled* signal function is a special case of the more general notion of a *contractive* function. Identifying decoupled signal functions is particularly useful as they can be used in feedback loops to guarantee well defined feedback (see Section 8.7) [37].

7.3.3 Time-Varying Properties

So far, the signal-function properties we have considered have been time-invariant: they have held for the entire lifetime of the signal function, and will continue to do so eternally. However, we now consider some time-varying properties of signal functions. These properties may come to hold at some point during execution, usually as a result of structural switches. These properties are valuable because they identify opportunities for run-time optimisations.

We begin with *sources*, signal functions that ignore their input:

Definition 8 (Source Signal Function) A signal function is a *source* if its current and future outputs do not depend on its current or future inputs:

$$\begin{aligned}&\mathit{Source} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{TimePred}\\ &\mathit{Source}\ \{\mathit{as}\}\ \mathit{sf}\ t = (s_1\ s_2 : \mathit{SigVec}\ \mathit{as}) \to (\mathbf{H}\ (s_1 \equiv_s s_2) \Rightarrow \mathbf{G}^{\mathbf{r}}\ (\mathit{sf}\ s_1 \equiv_s \mathit{sf}\ s_2))\ t\end{aligned}$$

This is a time-varying property because *sources* can arise dynamically; typically as a result of switching in a *source*.

We next overload the *changeless* and *reflexively changeless* properties from signals onto signal functions:

Definition 9 (Changeless Signal Function) A signal function is *changeless* if its output signal is *changeless*:

$$\begin{aligned}&\mathit{Changeless} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{TimePred}\\&\mathit{Changeless}\ \{\mathit{as}\}\ \mathit{sf}\ t\ =\ (s : \mathit{SigVec}\ \mathit{as}) \to \mathit{Changeless}\ (\mathit{sf}\ s)\ t\end{aligned}$$

Definition 10 (Reflexively Changeless Signal Function) A signal function is *reflexively changeless* if its output signal is *reflexively changeless*:

$$\begin{aligned}&\mathit{Changeless}^{r} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{TimePred}\\&\mathit{Changeless}^{r}\ \{\mathit{as}\}\ \mathit{sf}\ t\ =\ (s : \mathit{SigVec}\ \mathit{as}) \to \mathit{Changeless}^{r}\ (\mathit{sf}\ s)\ t\end{aligned}$$

Finally we define two very similar properties: *change propagating* and *change dependent*. *Change propagating* is the stronger property, but for our suggested optimisations *change dependent* will be sufficient (see sections 8.2 and 8.4).

Definition 11 (Change-Propagating Signal Function) A signal function is *change propagating* if, now and henceforth, its output will be *unchanging* whenever its input is *unchanging*:

$$\begin{aligned}&\mathit{ChangePrp} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{TimePred}\\&\mathit{ChangePrp}\ \{\mathit{as}\}\ \mathit{sf}\ t\ =\ (s : \mathit{SigVec}\ \mathit{as}) \to \mathbf{G}^{\mathbf{r}}\ (\mathit{Unchanging}\ s \Rightarrow \mathit{Unchanging}\ (\mathit{sf}\ s))\ t\end{aligned}$$

Definition 12 (Change-Dependent Signal Function) A signal function is *change dependent* if its output will be *unchanging* until its input is *changing*:

$$\begin{aligned}&\mathit{ChangeDep} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{TimePred}\\&\mathit{ChangeDep}\ \{\mathit{as}\}\ \mathit{sf}\ t\ =\ (s : \mathit{SigVec}\ \mathit{as}) \to (\mathit{Unchanging}\ (\mathit{sf}\ s)\ \mathbf{W}^{\mathbf{r}}\ \mathit{Changing}\ s)\ t\end{aligned}$$

Note that *change dependent* differs from our other properties in that a *change-dependent* signal function may cease to be *change dependent* in the future. This can only happen if a structural switch occurs within the *change-dependent* signal function. Thus, if a *change-dependent* signal function contains no switching combinators, then it is also *change propagating*.

7.3.4 Implied Properties

Many of the signal-function properties imply others directly. We list these implications below, omitting those that follow from transitivity.

Note that for clarity of presentation we will usually omit quantification of variables when giving implications. In these cases, any free variables should be assumed to be universally quantified at the top level. Also, note that the differing notation is due to some of the properties being time-varying, and some time-invariant.

$$\begin{aligned}&\mathit{Stateless}\ \mathit{sf} \uplus \mathit{Decoupled}\ \mathit{sf} &&\to \mathit{Causal}\ \mathit{sf}\\&\mathit{Changeless}^{r}\ \mathit{sf} &&\Rightarrow \mathit{Changeless}\ \mathit{sf}\\&\mathit{ChangePrp}\ \mathit{sf} &&\Rightarrow \mathit{ChangeDep}\ \mathit{sf}\\&\mathit{Changeless}^{r}\ \mathit{sf} &&\Rightarrow \mathit{ChangePrp}\ \mathit{sf} \wedge \mathit{Source}\ \mathit{sf}\\&\mathit{ChangeDep}\ \mathit{sf} \wedge \mathit{Source}\ \mathit{sf} &&\Rightarrow \mathit{Changeless}^{r}\ \mathit{sf}\end{aligned}$$

7.3.5 Properties of Primitives

All of the primitive signal functions we have defined are *causal*. This is required so that our signal functions can be realised by an implementation. The other properties that hold for the primitives are as follows:

- *Stateless*: *identity*, *sfFst*, *sfSnd*, *constant*, *never*, *lift*, *merge*, *join*, *fromS*, *sampleWith*
- *Decoupled*: *constant*, *never*, *now*, *integral*, *delay*
- *Source*: *constant*, *never*, *now*
- *Changeless*: *constant*, *never*, *now*
- $Changeless^r$: *never*
- *ChangePrp*: *identity*, *sfFst*, *sfSnd*, *never*, *hold*, *edge*, *lift*, *merge*, *join*, *fromS*, *when*, *sampleWith*
- *ChangeDep*: *identity*, *sfFst*, *sfSnd*, *never*, *hold*, *edge*, *lift*, *merge*, *join*, *fromS*, *when*, *sampleWith*

Note that for families of primitives that share the same properties, only the family name has been given (e.g. *delay* rather than *delayC*, *delayE*, *delayS*).

7.3.6 Properties of Combinators

The three primitive combinators (⋙, &&& and *switch*) preserve the properties of their constituent signal functions as follows[11]:

$$
\begin{array}{lll}
Causal\ sf_1 & \times\ Causal\ sf_2 & \rightarrow Causal\ (sf_1 \ggg sf_2) \\
Stateless\ sf_1 & \times\ Stateless\ sf_2 & \rightarrow Stateless\ (sf_1 \ggg sf_2) \\
Decoupled\ sf_1 & \uplus\ Decoupled\ sf_2 & \rightarrow Decoupled\ (sf_1 \ggg sf_2) \\
Source\ sf_1 & \vee\ Source\ sf_2 & \Rightarrow Source\ (sf_1 \ggg sf_2) \\
 & Changeless\ sf_2 & \Rightarrow Changeless\ (sf_1 \ggg sf_2) \\
 & Changeless^r\ sf_2 & \Rightarrow Changeless^r\ (sf_1 \ggg sf_2) \\
ChangePrp\ sf_1 & \wedge\ ChangePrp\ sf_2 & \Rightarrow ChangePrp\ (sf_1 \ggg sf_2) \\
ChangeDep\ sf_1 & \wedge\ ChangeDep\ sf_2 & \Rightarrow ChangeDep\ (sf_1 \ggg sf_2) \\
Causal\ sf_1 & \times\ Causal\ sf_2 & \rightarrow Causal\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Stateless\ sf_1 & \times\ Stateless\ sf_2 & \rightarrow Stateless\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Decoupled\ sf_1 & \times\ Decoupled\ sf_2 & \rightarrow Decoupled\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Source\ sf_1 & \wedge\ Source\ sf_2 & \Rightarrow Source\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Changeless\ sf_1 & \wedge\ Changeless\ sf_2 & \Rightarrow Changeless\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Changeless^r\ sf_1 & \wedge\ Changeless^r\ sf_2 & \Rightarrow Changeless^r\ (sf_1 \mathbin{\&\&\&} sf_2) \\
ChangePrp\ sf_1 & \wedge\ ChangePrp\ sf_2 & \Rightarrow ChangePrp\ (sf_1 \mathbin{\&\&\&} sf_2) \\
ChangeDep\ sf_1 & \wedge\ ChangeDep\ sf_2 & \Rightarrow ChangeDep\ (sf_1 \mathbin{\&\&\&} sf_2) \\
Causal\ sf & \times\ (\forall\ \{e\} \rightarrow Causal\ (f\ e)) & \rightarrow Causal\ (switch\ sf\ f) \\
Decoupled\ sf & \times\ (\forall\ \{e\} \rightarrow Decoupled\ (f\ e)) & \rightarrow Decoupled\ (switch\ sf\ f) \\
Source\ sf & \wedge\ (\lambda\ t \rightarrow \forall\ \{e\} \rightarrow Source\ (f\ e)\ t) & \Rightarrow Source\ (switch\ sf\ f) \\
ChangePrp\ sf & \wedge\ (\lambda\ t \rightarrow \forall\ \{e\} \rightarrow ChangePrp\ (f\ e)\ t) & \Rightarrow ChangePrp\ (switch\ sf\ f) \\
ChangeDep\ sf & & \Rightarrow ChangeDep\ (switch\ sf\ f) \\
Changeless^r\ sf & & \Rightarrow Changeless^r\ (switch\ sf\ f)
\end{array}
$$

The quantification over the event (e) in the temporal predicates requires us to explicitly route through the time argument, rather than it being handled implicitly by the temporal combinators.

Notice that most of the properties are only preserved by *switch* if the residual signal function also has that property. As the residual signal function is computed dynamically by a host-language function, the implementation has very little knowledge of this signal function. Consequently, optimisations based on these properties are limited. We return to this in Section 8.7.

[11] Agda notation: The $\forall\ \{e\}$ is shorthand for the universal quantification of implicit arguments whose type can be inferred automatically.

However, the *reflexively changeless* and *change-dependent* properties *are* preserved by *switch*. Intuitively, this is because a structural switch cannot occur until the subordinate signal function emits an event (a change). This will never occur for a *reflexively changeless* signal function, and the *change-dependent* property only guarantees the output to remain unchanging until the input changes. Thus optimisation of switches based on these two properties is much more feasible, as no information is required about the residual signal function.

Finally, the interaction between properties gives rise to the following implications:

$$\mathit{Changeless}\ sf_1 \ \wedge \mathit{ChangePrp}\ sf_2 \ \Rightarrow \mathit{Changeless}\ (sf_1 \ggg sf_2)$$
$$\mathit{Changeless}^r\ sf_1 \wedge \mathit{ChangeDep}\ sf_2 \Rightarrow \mathit{Changeless}^r\ (sf_1 \ggg sf_2)$$

8 Suggested Optimisations

In this section we overview the change-based optimisations that are possible on a signal-function network. We do not discuss how to *implement* such optimisations, as that depends on the details of the specific FRP implementation involved. Some optimisations would no doubt be more applicable for some implementations than others.

We describe many of the optimisations in this section as *static*; that is, they can be applied at compile time. A common way to implement signal functions is as state transition functions in a data-flow network. Such functions execute over a discrete sequence of time steps, mapping an input sample and state to an output sample and state at each step. Each signal function maintains an internal state, rather than relying on a global state. In this style of implementation, our static optimisations could also be applied dynamically (at run-time) after each structural switch in the network, as new optimisations may be possible for the new network configuration. The capacity for dynamic optimisation is the reason that many of our properties are time-varying: they allow for optimisations that are only valid at certain points in time.

8.1 Eliminating Unused Signal Functions

Any signal function whose output signal is not used can be eliminated. This could arise either because the signal is eliminated by routing primitives (and thus never reaches another signal function), or because all signal functions that do receive it are *sources*. This is essentially reactive-level garbage collection, exploiting the properties of our routing combinators and signal functions to identify these unused signal functions. This is a static optimisation, because we know that, except at the moments of switching, there will be no change in which signals are used.

The latest version of Yampa [10, 26] uses this technique to some degree, but is limited by the Unary FRP model. As discussed in Section 4.1, much of the routing of the arrow framework is carried out by lifted pure functions, hiding the routing from the reactive level.

8.2 Compressing Changeless Signals

If a signal is *reflexively changeless*, then we know it is constant. Repeatedly recomputing a constant value (in a pull-based implementation) is a waste of computational resources and should be avoided. As a static optimisation, the signal can be compressed (i.e. compute the

value and then discard the signal function that computes it) and then constant propagation applied. This, in turn, could cause other signal functions to become *sources*. As a *change-dependent source* is *reflexively changeless*, this could lead to *change-dependent* signal functions becoming *reflexively changeless*, and thus present further optimisation opportunities.

In terms of the implications from Section 7.3:

$$\begin{array}{ll} \mathit{Changeless}^{r}\ \mathit{sf}_1 \wedge \mathit{ChangeDep}\ \mathit{sf}_2 & \Rightarrow \mathit{Changeless}^{r}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\ \mathit{ChangeDep}\ \mathit{sf} \ \wedge \mathit{Source}\ \mathit{sf} & \Rightarrow \mathit{Changeless}^{r}\ \mathit{sf} \end{array}$$

8.3 Switch Elimination

We can eliminate switching combinators that will never switch out the subordinate signal function. This will sometimes occur as part of the aforementioned optimisation to compress changeless signals, but only in the case that the output of the subordinate signal function is *reflexively changeless*.

However, there is another static optimisation that can do better. If we know that the event signal produced by the subordinate signal function of *switch* is *reflexively changeless*, then we know that the subordinate signal function will never be switched out. Consequently, a valid optimisation is to remove the *switch* combinator and replace it with the subordinate signal function, discarding the event signal.

$$\mathit{Changeless}^{r}\ (\mathit{snd}\ (\mathit{sf}\ s)) \Rightarrow \mathbf{G}^{\mathbf{r}}\ ((\mathit{switch}\ \mathit{sf}\ f)\ s \equiv_s (\mathit{sf} \ggg \mathit{sfFst})\ s)$$

Eliminating switching combinators can be considerably beneficial, as they often obstruct other optimisations (such as causal-commutative-arrow optimisation [25]).

8.4 Change Propagation

The motivation for change propagation optimisations is that the output of a signal function may often be *unchanging* over a period of time, often as a consequence of its input being *unchanging*. The idea is to identify where this is the case, and then not recompute that unchanging output. This approach is inherent to push-based implementations of FRP (such as FrTime [9]), wherein a signal is never recomputed unless there is a change in its input. It is also present in push-pull implementations (such as Grapefruit [23] and Reactive [14]) that make use of push-based execution for step and event signals, and pull-based implementation for continuous signals. However, some change propagation is still possible to some degree for the "pulled" signals of such systems, and is also useful for entirely pull-based systems (such as Yampa [27]).

In our setting, it is the signal functions that we have identified as *change-dependent* that we can exploit for change propagation. Consider a discretely sampled pull-based implementation where signal functions are executed every time step, mapping an input sample to an output sample. If a signal function is *change-dependent*, and its current input is *unchanging*, then its current output is guaranteed to be *unchanging*:

$$\mathit{ChangeDep}\ \mathit{sf} \wedge \mathit{Unchanging}\ s \Rightarrow \mathit{Unchanging}\ (\mathit{sf}\ s)$$

Thus there is no need to compute an output sample because: for event signals there is no event occurrence; and for step and continuous signals the value is known from the previous sample.

Implementation note: If signal functions are implemented with an internal state (as in Yampa), then any *change-dependent* signal functions must be implemented such that there is no modification of that state whenever their input is *unchanging*. Without this condition, state updates would be lost and the optimisation would be invalidated.

Change propagation is hindered if information about which signals are *unchanging* is lost. For example, the latest version of Yampa [10, 26] employs some limited change propagation. However, as discussed in Section 4.1, there is no difference in Yampa between a tuple of signals and a signal of tuples. Thus a change to one signal in a tuple appears to be a change to all signals in the tuple. FrTime, on the other hand, has very effective change propagation because it performs run-time equality checks to compare recomputed values with the previous value, to determine if it really has changed.

8.5 Interaction between Optimisations and Switching

When giving the definition of change, we defined step signals to be *changing* at $time_0$ (we did the same for continuous signals, though this may be less obvious). This may seem counter-intuitive: for example, a constant signal may seem never to change. However, we chose this definition with optimisation in mind.

The reason pertains to the dynamic nature of signal-function networks. Each signal function runs in its own *local time* (see Section 5.1.2). Consequently, what is $time_0$ to one signal function may not be to another. In particular, after a structural switch, the residual signal function will be at its local $time_0$, whereas the network external to the switching combinator will not. Assume the output is a constant step signal. The initial value of that signal appears as a change to the rest of the network, as this is a new value that has not been seen before. If the signal was treated as *reflexively changeless*, then the network could be incorrectly optimised based on the assumption that the value of the signal is the same as it was at previous time points.

8.6 Decoupled Switches

There is a design choice for switching combinators as to whether the output at the moment of switching should be taken from the residual or subordinate signal. Many FRP implementations provide two versions of each switching combinator to cater for both. The switching combinators we have defined in this paper take their output from the residual signal. Those that take their output from the subordinate signal are often known as *decoupled switches*.

We mention decoupled switches because they can be a pitfall when performing optimisations of sampled implementations. This is related to the issue in Section 8.5. The problem is that from the point of view of the external network, the value of any step or continuous signals in the residual signal function of a decoupled switch appears to change not at the moment of switching, but immediately afterwards. Thus, even if the residual signal function is a constant signal, there would appear to be a change in this *changeless* signal after $time_0$.

8.7 Implementing Signal-Function Properties

As discussed in Section 3.5.3, one of the advantages of a first-class signal-function abstraction is that additional information can be associated with it. Thus signal functions can record

FRP Variant	Signal Kind		
N-ary FRP	Event Signal	Step Signal	Continuous Signal
Reactive	Event	Reactive Value	Time Function
Grapefruit	Discrete Signal	Segmented Signal	Continuous Signal

Table 1 Naming conventions for signal kinds

internally which properties they satisfy. Provided the implementer identifies the properties of all the primitives, the properties of any composite signal function can be computed from those of its components, using the implications in Section 7.3.6.

In most cases these properties would be kept internal to the implementation and hidden from the FRP programmer. However, there are properties that it is useful to make visible. For example, consider the *decoupled* signal-function property. In a language that allows cyclic networks, requiring a *decoupled* signal function on all feedback paths prevents any instantaneous cyclic dependencies. If the *decoupled* property is encoded in the *type* of signal functions, then the host-language type checker can rule out instantaneous feedback. [37]

Another advantage of encoding properties in types is that we can infer properties of switching combinators that depend on the residual signal function. We do not know the value of the residual signal function in advance, but we do know its type and thus can perform optimisations based on any properties in that type.

9 Related Work

Devising semantic models for FRP that respect the abstractions of discrete and continuous time is nothing new. Daniels [13] has constructed a formal semantics for an idealised CFRP language (that assumes no approximation errors in the implementation), while Wan and Hudak [39] have shown that with certain constraints, a sampled implementation of CFRP can be ensured to converge to the semantics. The main differences between such works and ours is that they consider signals (not signal functions) as the central abstraction, and that they do not distinguish between continuous and step signals.

Two Haskell-embedded FRP implementations currently under development are *Reactive* [14] and *Grapefruit* [23]. They both identify the three signal kinds (see Table 1), and use push-based approaches for the implementation of step and event signals. Signals are first class in both systems, though Grapefruit also has a first-class signal function construct. Switching combinators switch between signals in Reactive, and signal functions in Grapefruit.

Central to FRP's hybrid capabilities is the notion of events occurring at specific points in time, and specifying reactions to such events. This means asking whether some event has occurred yet or not. A natural way of doing this is to compare the time associated with the event with the present time. However, this directly leads to a causality problem: how can the precise future time of an event that has not yet occurred be known in general? Predicating an FRP semantics on such a capability would inevitably render the whole model non-causal, severely limiting its usefulness for describing the meaning of FRP programs, especially when fixed points (some form of feedback) is involved.

The key to resolving this dilemma is to concentrate on the original question above, has an event occurred yet or not, not the exact future time of its occurrence. In the original work on Fran [15], this was achieved through a careful definition of a customised time domain with an ordering that permitted deciding whether one time value is before another without

knowing the exact value of the second. The same problem is addressed in a similar way in Reactive by making events "future values". Grapefruit deals with the issue by considering all possible interleavings of future event occurrences, relying on laziness to ensure that only the correct interleaving is evaluated. In our semantic model, we have addressed the problem more directly by building a notion of observation *only up to some specific point in time* into the definitions of event and step signals. This leads to a clear and simple semantics as it does not rely on any auxiliary notions, and also to a finitary semantics for events and changes. Our approach is unlikely to be very useful as a direct basis for implementation, but then the goal of our semantics is not to serve as a basis for some specific implementation, but rather to serve as a reference relevant for *any* implementation.

Elerea ("Eventless Reactivity") [31,32] is another Haskell embedding of FRP currently in development. Elerea has first-class signals and signal generators (as distinct types), but is otherwise in many ways similar to Yampa, being a single-kinded pull-based discretely sampled system. In contrast to Yampa, Elerea doesn't abstract away from the discrete implementation. Yampa provides a set of primitives that operate on conceptually continuous signals and conceptually discrete events, trying to hide the sampling rate from the programmer. Elerea, on the other hand, exposes the sampling rate, reducing the number of primitives required. Similarly, whereas Yampa provides an abstract event type that is *internally* a continuous signal carrying an option type, Elerea directly uses continuous signals carrying option types (or Booleans) for signals with discrete behaviour. Uniquely, Elerea provides a monadic *join* for signals:

$$join : Signal\ (Signal\ A) \rightarrow Signal\ A$$

One application of this is supporting dynamic collections of signals, allowing Elerea to be used for expressing highly structurally dynamic reactive systems, such as video games, in much the same way as Yampa.

The synchronous data-flow languages [2,18,19] have long modelled reactive programs as synchronous data-flow networks. These languages treat time as discrete, and have static first-order structures. Optimising such networks is well studied [24,20]. However, they lack the dynamism and higher-order reactive constructs of FRP; though there has been some work on extending Lucid Synchrone in this direction [6,8].

FrTime [9,5] is a push-based FRP language (embedded in Scheme) with first-class signals, which uses a variety of optimisation techniques. The inherent change propagation of the push-based execution is enhanced by performing run-time equality checks on the values of recomputed signals to determine whether they really have changed. It also uses a static optimisation called *lowering*, which reduces a data-flow network by fusing together composite signal functions into single signal functions (discarding the routing information). In FrTime, this technique is only applied to signal functions that are lifted pure functions. For example, (in our setting) a typical lowering optimisation would look like:

$$lift\ f \ggg lift\ g = lift\ (g \circ f)$$

FrTime's lowering optimisations are applied statically at compile time, which allows for substantial optimisation of source code, but does not allow dynamic optimisation of the network after structural switches. Lowering optimisations are also applied by Elerea [31] and Yampa [26], albeit not to the extent of FrTime. However, Yampa can lower some stateful signal functions as well as stateless ones. Yampa performs its lowering optimisations dynamically, which suffers from additional run-time overhead, but does allow for continued optimisation after structural changes.

A recent development has been a static optimisation technique for *Causal Commutative Arrows* [25] that lowers any static arrow network (which may include cycles) to a single arrow with a single internal state. When applied to some examples from Yampa (where a signal function is an arrow), the elimination of most of the arrow infrastructure has resulted in impressive performance gains. However, this technique does not extend to networks containing switching combinators, and so cannot be applied to arbitrary Yampa programs.

10 Future Work

All of the code in this paper has been formulated directly in Agda without the syntactic sugar we used for presentational purposes. Formally proving the properties of signals and signal functions from Section 7 is ongoing work; the proofs formalised thus far can be can be found on the first author's website.

In this paper we have only considered acyclic networks. Feedback is an essential facility for a synchronous data-flow language, and thus we need to extend our routing primitives with a feedback combinator. Ensuring that such a combinator is well-defined in a highly dynamic setting is not trivial. We have previously [37] proposed a feedback combinator, along with a type system extension that can guarantee the combinator is well-defined, but have not yet incorporated it in our new conceptual model. The *decoupled* property is key to the above type system, as well as to several other FRP primitives that we have not defined in this paper. For example, the signal function *pre*, which appears in most reactive languages, conceptually introduces an infinitesimal delay in a signal. However, most implementations give *pre* the behaviour of a one–time-sample delay, which does not respect its conceptual definition. Furthermore, having done this, a programmer can (indirectly) gain access to the sampling rate, and thence define various things that break the signal abstractions (such as events occurring at all points in time). We aim to extend our model with decoupling primitives, and to precisely address the notions of feedback and decoupling.

Finally, our long term goal is an efficient scalable implementation of FRP that respects the conceptual definitions of signal functions. To achieve efficiency, we believe such an implementation would dynamically (i.e. at run-time) employ the optimisations suggested in this paper, optimising after each structural switch.

11 Conclusions

In this paper we introduced a conceptual model of FRP that we call N-ary FRP, and we defined it through an ideal denotational semantics. This model is a development of our previous attempts to model n-ary (multi-input and multi-output) signal functions that operate over distinct kinds of signals.

With this model as a base, we identified and formally defined, using temporal logic, several important temporal properties of signals and signal functions pertaining to change and change propagation. These properties hold in our model, and we would expect them to hold in any implementation that we would consider "faithful" to the semantics. Having defined these properties, we described how they relate to optimisation techniques for FRP implementations, and what properties have to hold for certain optimisations to be valid.

Reasoning about change in the setting of FRP is challenging due to structural dynamism and changes due just to time passing. For example, it is very easy to introduce invalid optimisations by failing to appreciate subtle aspects of the semantics. Having a formal framework

that allows optimisation opportunities to be identified and properly justified is thus a useful aid for FRP implementers.

Acknowledgements We would like to thank George Giorgidze, Florent Balestrieri and the anonymous reviewers for their detailed comments and feedback.

References

1. Simulink User's Guide, Version 7.6. 3 Apple Hill Drive, Natick, MA (2010). URL www.mathworks.com/help/toolbox/simulink/
2. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE, Special issue on embedded systems **91**(1), 64–83 (2003)
3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2), 87–152 (1992)
4. Blom, K.J.: Dynamic interactive virtual environments. Ph.D. thesis, Department of Informatics, University of Hamburg (2009)
5. Burchett, K., Cooper, G.H., Krishnamurthi, S.: Lowering: A static optimization technique for transparent functional reactivity. In: Partial Evaluation and Program Manipulation (PEPM '07), pp. 71–80. ACM (2007)
6. Caspi, P., Pouzet, M.: Synchronous Kahn networks. In: International Conference on Functional Programming (ICFP '96), pp. 226–238. ACM (1996)
7. Cheong, M.H.: Functional programming and 3D games. BEng thesis, University of New South Wales (2005)
8. Colaço, J.L., Girault, A., Hamon, G., Pouzet, M.: Towards a higher-order synchronous data-flow language. In: Embedded Software (EMSOFT '04), pp. 230–239. ACM (2004)
9. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: European Symposium on Programming (ESOP '06), pp. 294–308. Springer (2006)
10. Courtney, A.: Modeling user interfaces in a functional language. Ph.D. thesis, Yale University (2004)
11. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Haskell Workshop (Haskell '01), pp. 41–69. Elsevier (2001)
12. Courtney, A., Nilsson, H., Peterson, J.: The Yampa arcade. In: Haskell Workshop (Haskell '03), pp. 7–18. ACM (2003)
13. Daniels, A.: A semantics for functions and behaviours. Ph.D. thesis, University of Nottingham (1999)
14. Elliott, C.: Push-pull functional reactive programming. In: Haskell Symposium (Haskell '09), pp. 25–36. ACM (2009)
15. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming (ICFP '97), pp. 263–273. ACM (1997)
16. Giorgidze, G., Nilsson, H.: Switched-on Yampa: Declarative programming of modular synthesizers. In: Practical Aspects of Declarative Languages (PADL '08), pp. 282–298. Springer (2008)
17. Giorgidze, G., Nilsson, H.: Mixed-level embedding and JIT compilation for an iteratively staged DSL. In: Functional and Constraint Logic Programming (WFLP '10), pp. 48–65. Springer (2011)
18. Halbwachs, N.: Synchronous Programming of Reactive Systems. The Springer International Series in Engineering and Computer Science. Springer (1993)
19. Halbwachs, N.: Synchronous programming of reactive systems, a tutorial and commented bibliography. In: Computer Aided Verification (CAV '98), pp. 1–16. Springer (1998)
20. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language Lustre. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
21. Henzinger, T.A.: The theory of hybrid automata. In: Logics in Computer Science (LICS '96), pp. 278–292. IEEE Computer Society (1996)
22. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, chap. 3, pp. 172–255. Cambridge University Press (2004)
23. Jeltsch, W.: Signals, not generators! In: Trends in Functional Programming (TFP '09), pp. 145–160. Intellect (2010)
24. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers **36**(1), 24–35 (1987)
25. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows and their optimization. In: International Conference on Functional Programming (ICFP '09), pp. 35–46. ACM (2009)

26. Nilsson, H.: Dynamic optimization for functional reactive programming using generalized algebraic data types. In: International Conference on Functional Programming (ICFP '05), pp. 54–65. ACM (2005)
27. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell Workshop (Haskell '02), pp. 51–64. ACM (2002)
28. Nilsson, H., Peterson, J., Hudak, P.: Functional hybrid modeling. In: Practical Aspects of Declarative Languages (PADL '03), pp. 376–390. Springer (2003)
29. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory. Oxford University Press (1990)
30. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
31. Patai, G.: Eventless reactivity from scratch. In: Draft Proceedings of Implementation and Application of Functional Languages (IFL '09), pp. 126–140 (2009)
32. Patai, G.: Efficient and compositional higher-order streams. In: Functional and Constraint Logic Programming (WFLP '10), pp. 137–154. Springer (2011)
33. Paterson, R.: A new notation for arrows. In: International Conference on Functional Programming (ICFP '01), pp. 229–240. ACM (2001)
34. Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with Haskell. In: Practical Aspects of Declarative Languages (PADL '99), pp. 91–105. Springer (1999)
35. Pouzet, M.: Lucid Synchrone, version 3: Tutorial and reference manual. Université Paris-Sud, LRI (2006). URL www.di.ens.fr/ pouzet/lucid-synchrone
36. Sculthorpe, N., Nilsson, H.: Optimisation of dynamic, hybrid signal function networks. In: Trends in Functional Programming (TFP '08), pp. 97–112. Intellect (2009)
37. Sculthorpe, N., Nilsson, H.: Safe functional reactive programming through dependent types. In: International Conference on Functional Programming (ICFP '09), pp. 23–34. ACM (2009)
38. Venema, Y.: Temporal logic. In: The Blackwell Guide to Philosophical Logic, chap. 10, pp. 203–223. Blackwell (2001)
39. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: Programming Language Design and Implementation (PLDI '00), pp. 242–252. ACM (2000)

A Utilities

This appendix contains the definitions of some of the data types and utility functions that we use. This is not exhaustive; we assume familiarity with data types and functions from the Haskell Prelude. The complete code is available on the first author's website.

A.1 Data Types

We assume familiarity with the data types *Unit*, *Bool*, *List* and *Maybe*. Sum and product types are given below. Note that the usual product ($\times$) is defined in terms of Agda's dependent product (Σ):

```
data _⊎_ (A B : Set) : Set where
  inl : A → A ⊎ B
  inr : B → A ⊎ B
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) → B a → Σ A B
_×_ : Set → Set → Set
A × B = Σ A (λ _ → B)
```

We define propositional equality as follows:

```
data _≡_ {A : Set} (a : A) : A → Set where
  refl : a ≡ a
```

A.2 Combinators

We define some basic function combinators:

```
const : {A B : Set} → A → B → A
const a _ = a

first : {A B C : Set} → (A → C) → A × B → C × B
first f (a, b) = (f a, b)

second : {A B C : Set} → (B → C) → A × B → A × C
second f (a, b) = (a, f b)

result : {A B C : Set} → (B → C) → (A → B) → (A → C)
result f g = f ∘ g

result2 : {A B C D : Set} → (C → D) → (A → B → C) → (A → B → D)
result2 f g a = f ∘ g a
```

We also define some functions over *Maybe* types:

```
maybeMap : {A B : Set} → (A → B) → Maybe A → Maybe B
maybeMap f nothing = nothing
maybeMap f (just a) = just (f a)

maybeMap2 : {A B C : Set} → (A → B → C) → Maybe A → Maybe B → Maybe C
maybeMap2 f nothing mb = nothing
maybeMap2 f (just a) mb = maybeMap (f a) mb

maybeMerge : {A B C : Set} → (A → C) → (B → C) → (A → B → C)
           → Maybe A → Maybe B → Maybe C
maybeMerge fa fb fab nothing nothing = nothing
maybeMerge fa fb fab nothing (just b) = just (fb b)
maybeMerge fa fb fab (just a) nothing = just (fa a)
maybeMerge fa fb fab (just a) (just b) = just (fab a b)
```

A.3 Functions on Signals

We now define some functions over signals, change prefixes and change lists; for use at the conceptual level. We begin with some look-up functions that determine if there is a change at a given time point:

```
lookupCL : {A : Set} → ChangeList A → Time → Maybe A
lookupCL []          _          = nothing
lookupCL ((δ, a), δas) t | t <  δ = nothing
                         | t == δ = just a
                         | t >  δ = lookupCL δas (t − δ)
lookupCP : {A : Set} → ChangePrefix A → Time → Maybe A
lookupCP cp t = lookupCL (cp t) t
```

We can compute the value of a step signal at a given time point:

```
valS : {A : Set} → SigVec (S A) → Time → A
valS (a0, cp) t = case reverse (cp t) of
                    []          → a0
                    (_, a1) :: _ → a1
```

Similarly, we can determine if an event is occurring at a given time point:

```
occ : {A : Set} → SigVec (E A) → Time → Maybe A
occ (ma, cp) t | t == 0 = ma
               | t >  0 = lookupCP cp t
```

We now define some mappings:

$$
\begin{aligned}
&\mathit{mapCL} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ B\\
&\mathit{mapCL} = \mathit{map} \circ \mathit{second}\\
&\mathit{mapCP} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{ChangePrefix}\ A \to \mathit{ChangePrefix}\ B\\
&\mathit{mapCP} = \mathit{result} \circ \mathit{mapCL}\\
&\mathit{mapC} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{SigVec}\ (\mathsf{C}\ A) \to \mathit{SigVec}\ (\mathsf{C}\ B)\\
&\mathit{mapC} = \mathit{result}\\
&\mathit{mapE} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{SigVec}\ (\mathsf{E}\ B)\\
&\mathit{mapE}\ f\ (ma, cp) = (\mathit{maybeMap}\ f\ ma, \mathit{mapCP}\ f\ cp)\\
&\mathit{mapS} : \{A\ B : \mathit{Set}\} \to (A \to B) \to \mathit{SigVec}\ (\mathsf{S}\ A) \to \mathit{SigVec}\ (\mathsf{S}\ B)\\
&\mathit{mapS}\ f\ (a, cp) = (f\ a, \mathit{mapCP}\ f\ cp)
\end{aligned}
$$

Mapping over two signals is somewhat more involved. To aid in these definitions, we also define *mapCPtime*, which allows the mapped function to depend upon the time point at which it is applied:

$$
\begin{aligned}
&\mathit{mapCLtime} : \{A\ B : \mathit{Set}\} \to (\mathit{Time} \to A \to B) \to \mathit{Time} \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ B\\
&\mathit{mapCLtime}\ f\ d\ [\,] = [\,]\\
&\mathit{mapCLtime}\ f\ d\ ((\delta, a) :: \delta as) = \mathbf{let}\ d' = d + \delta\ \mathbf{in}\ (\delta, f\ d'\ a) :: \mathit{mapCLtime}\ f\ d'\ \delta as\\
&\mathit{mapCPtime} : \{A\ B : \mathit{Set}\} \to (\mathit{Time} \to A \to B) \to \mathit{ChangePrefix}\ A \to \mathit{ChangePrefix}\ B\\
&\mathit{mapCPtime}\ f = \mathit{result}\ (\mathit{mapCLtime}\ f\ 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapC2} : \{A\ B\ Z : \mathit{Set}\} \to (A \to B \to Z) \to \mathit{SigVec}\ (\mathsf{C}\ A) \to \mathit{SigVec}\ (\mathsf{C}\ B) \to \mathit{SigVec}\ (\mathsf{C}\ Z)\\
&\mathit{mapC2}\ f\ s_1\ s_2\ t = f\ (s_1\ t)\ (s_2\ t)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapS2} : \{A\ B\ Z : \mathit{Set}\} \to (A \to B \to Z) \to \mathit{SigVec}\ (\mathsf{S}\ A) \to \mathit{SigVec}\ (\mathsf{S}\ B) \to \mathit{SigVec}\ (\mathsf{S}\ Z)\\
&\mathit{mapS2}\ f\ (a, cp_a)\ (b, cp_b) = (f\ a\ b, \lambda\ t \to \mathit{mergeS}\ a\ b\ (cp_a\ t)\ (cp_b\ t))\\
&\quad \mathbf{where}\\
&\qquad \mathit{mergeS} : A \to B \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ B \to \mathit{ChangeList}\ Z\\
&\qquad \mathit{mergeS}\ a_0\ b_0\ [\,]\ \delta bs = \mathit{mapCL}\ (f\ a_0)\ \delta bs\\
&\qquad \mathit{mergeS}\ a_0\ b_0\ \delta as\ [\,] = \mathit{mapCL}\ (\lambda\ a_n \to f\ a_n\ b_0)\ \delta as\\
&\qquad \mathit{mergeS}\ a_0\ b_0\ ((\delta_a, a_1), \delta as)\ ((\delta_b, b_1), \delta bs)\\
&\qquad\quad \mid \delta_a <\ \ \delta_b = (\delta_a, f\ a_1\ b_0) :: \mathit{mergeS}\ a_1\ b_0\ \delta as\ ((\delta_b - \delta_a, b_1) :: \delta bs)\\
&\qquad\quad \mid \delta_a == \delta_b = (\delta_a, f\ a_1\ b_1) :: \mathit{mergeS}\ a_1\ b_1\ \delta as\ \delta bs\\
&\qquad\quad \mid \delta_a >\ \ \delta_b = (\delta_b, f\ a_0\ b_1) :: \mathit{mergeS}\ a_0\ b_1\ ((\delta_a - \delta_b, a_1) :: \delta as)\ \delta bs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mergeE2} : (A \to Z) \to (B \to Z) \to (A \to B \to Z) \to \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{SigVec}\ (\mathsf{E}\ B) \to \mathit{SigVec}\ (\mathsf{E}\ Z)\\
&\mathit{mergeE2}\ fa\ fb\ fab\ (ma, cp_a)\ (mb, cp_b) = (\mathit{maybeMerge}\ fa\ fb\ fab\ ma\ mb, \lambda\ t \to \mathit{mergeCL}\ (cp_a\ t)\ (cp_b\ t))\\
&\quad \mathbf{where}\\
&\qquad \mathit{mergeCL} : \mathit{ChangeList}\ A \to \mathit{ChangeList}\ B \to \mathit{ChangeList}\ Z\\
&\qquad \mathit{mergeCL}\ [\,]\ \delta bs = \mathit{mapCL}\ fb\ \delta bs\\
&\qquad \mathit{mergeCL}\ \delta as\ [\,] = \mathit{mapCL}\ fa\ \delta as\\
&\qquad \mathit{mergeCL}\ ((\delta_a, a), \delta as)\ ((\delta_b, b), \delta bs) \mid \delta_a == \delta_b = (\delta_a, fab\ a\ b) :: \mathit{mergeCL}\ \delta as\ \delta bs\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \delta_a <\ \ \delta_b = (\delta_a, fa\ a) :: \mathit{mergeCL}\ \delta as\ ((\delta_b - \delta_a, b) :: \delta bs)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \delta_a >\ \ \delta_b = (\delta_b, fb\ b) :: \mathit{mergeCL}\ ((\delta_a - \delta_b, a) :: \delta as)\ \delta bs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{joinE2} : \{A\ B\ Z : \mathit{Set}\} \to (A \to B \to Z) \to \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{SigVec}\ (\mathsf{E}\ B) \to \mathit{SigVec}\ (\mathsf{E}\ Z)\\
&\mathit{joinE2}\ f\ (ma, cp_a)\ (mb, cp_b) = (\mathit{maybeMap2}\ f\ ma\ mb, \lambda\ t \to \mathit{joinCL}\ 0\ (cp_a\ t)\ (cp_b\ t))\\
&\quad \mathbf{where}\\
&\qquad \mathit{joinCL} : \mathit{Time} \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ B \to \mathit{ChangeList}\ Z\\
&\qquad \mathit{joinCL}\ \_\ [\,]\ \_ = [\,]\\
&\qquad \mathit{joinCL}\ \_\ \_\ [\,] = [\,]\\
&\qquad \mathit{joinCL}\ d\ ((\delta_a, a) :: \delta as)\ ((\delta_b, b) :: \delta bs) \mid \delta_a == \delta_b = (d + \delta_a, f\ a\ b) :: \mathit{joinCL}\ 0\ \delta as\ \delta bs\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \delta_a <\ \ \delta_b = \mathit{joinCL}\ (d + \delta_a)\ \delta as\ ((\delta_b - \delta_a, b) :: \delta bs)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \delta_a >\ \ \delta_b = \mathit{joinCL}\ (d + \delta_b)\ ((\delta_a - \delta_b, a) :: \delta as)\ \delta bs
\end{aligned}
$$

```
mapCE : {A B Z : Set} → (A → B → Z) → SigVec (C A) → SigVec (E B) → SigVec (E Z)
mapCE f s (mb, cp) = (maybeMap (f (s 0)) mb, mapCPtime (f ∘ s) cp)
mapSE : {A B Z : Set} → (A → B → Z) → SigVec (S A) → SigVec (E B) → SigVec (E Z)
mapSE f s (mb, cp) = (maybeMap (f (valS s 0)) mb, mapCPtime (f ∘ valS s) cp)
```

Next we define some utility functions on change lists and change prefixes. The time of the last change in a change list is the sum of its time deltas:

```
lastChangeTime : {A : Set} → ChangeList A → Time
lastChangeTime = sum ∘ map fst
```

We can take the prefix of a change list up to a specified point in time (inclusive or exclusive):

```
takeIncl : {A : Set} → Time → ChangeList A → ChangeList A
takeIncl _ [] = []
takeIncl t ((δ, a) :: δas) | t <  δ = []
                           | t >= δ = (δ, a) :: takeIncl (t − δ) δas
takeExcl : {A : Set} → Time → ChangeList A → ChangeList A
takeExcl _ [] = []
takeExcl t ((δ, a) :: δas) | t <= δ = []
                           | t >  δ = (δ, a) :: takeExcl (t − δ) δas
```

The *delayCL* function delays a change list by increasing the first time delta. The *delayCP* function delays a change prefix by reducing the sample time by the delay period (*d*), and then delaying the resultant change list by that amount. It also takes *Maybe* an initial value as an argument, allowing an initial change, if any, to be inserted into the resultant change list.

```
delayCL : {A : Set} → Time+ → ChangeList A → ChangeList A
delayCL d [] = []
delayCL d ((δ, a) :: δas) = (d + δ, a) :: δas
delayCP : {A : Set} → Time+ → Maybe A → ChangePrefix A → ChangePrefix A
delayCP d ma cp t | t <  d = []
                  | t >= d = case ma of
                               nothing → delayCL d (cp (t − d))
                               just a  → (d, a) :: cp (t − d)
```

A.4 Switching Utilities

In this section we define utility functions that are only used by *switch* (see Appendix B).

The *advance* function shifts the time frame of a signal forwards by a given amount of time (*d*), discarding everything before time *d* (some authors call this *ageing* the signal). This is similar to the *delay* signal function, except it looks into the future instead of the past. Unlike *delay* this is acausal, and so wouldn't make sense as a signal function. However, *advance* is only used as a utility by *switch*, connecting a signal from outside *switch* to the local time of the residual signal function (because from the point of view of the residual signal function, the local time of the external network is in the future). Semantically this is achieved by advancing any sample point by *d*, and, in the case of step and event signals, discarding the prefix of the signal up to time *d*:

```
advanceCL : {A : Set} → Time → ChangeList A → ChangeList A
advanceCL d [] = []
advanceCL d ((δ, a) :: δas) | δ <= d = advanceCL (d − δ) δas
                            | δ >  d = (δ − d, a) :: δas
```

```
advanceCP : {A : Set} → Time → ChangePrefix A → ChangePrefix A
advanceCP d cp t = advanceCL d (cp (t + d))

advance : {as : SVDesc} → Time → SigVec as → SigVec as
advance {C _} d s       = λ t → s (t + d)
advance {S _} d s       = (valS s d, advanceCP d (snd s))
advance {E _} d s       = (occ s d, advanceCP d (snd s))
advance {_,_} d (s1,s2) = (advance d s1, advance d s2)
```

The *svAtTime* function allows us to construct a signal vector based on the sample time, even though that sample time is not yet known:

```
svAtTime : {as : SVDesc} → (SampleTime → SigVec as) → SigVec as
svAtTime {C _} f = λ t → f t t
svAtTime {E _} f = (fst (f 0), λ t → snd (f t) t)
svAtTime {S _} f = (fst (f 0), λ t → snd (f t) t)
svAtTime {_,_} f = (svAtTime (fst ∘ f), svAtTime (snd ∘ f))
```

The *fstOcc* function returns the first event occurrence of an event signal, provided it occurs before a specified point in time (inclusive):

```
fstOcc : {A : Set} → SigVec (E A) → Time → Maybe (Time × A)
fstOcc (just a, _)     _ = just (0, a)
fstOcc (nothing, cp) t   = case cp t of
                             []       → nothing
                             (δa :: _) → just δa
```

The *takeExclEnd* function applies a change prefix to a (strictly positive) time, discards any change at precisely that time, and returns the remaining change list and the time delta since the final change:

```
takeExclEnd : {A : Set} → ChangePrefix A → Time+ → ChangeList A × Δt
takeExclEnd cp t = let δas = takeExcl t (cp t)
                   in (δas, t − lastChangeTime δas)
```

Finally, we define some splicing functions that compose two signal vectors *temporally*, cutting the first vector at the given event occurrence time:

```
spliceC : {A : Set} → SigVec (C A) → SigVec (C A) → EventTime+ → SigVec (C A)
spliceC _ s2 te t = s2 (t − te)

spliceS : {A : Set} → SigVec (S A) → SigVec (S A) → EventTime+ → SigVec (S A)
spliceS (a1, cp1) (a2, cp2) te = let (δas1, δ) = takeExclEnd cp1 te
                                 in (a1, λ t → δas1 ++ (δ, a2) :: cp2 (t − te))

spliceE : {A : Set} → SigVec (E A) → SigVec (E A) → EventTime+ → SigVec (E A)
spliceE (ma1, cp1) (ma2, cp2) te = let (δas1, δ) = takeExclEnd cp1 te
                                   in (ma1, λ t → let δas2 = cp2 (t − te)
                                                  in δas1 ++ case ma2 of
                                                               nothing → delayCL δ δas2
                                                               just a2 → (δ, a2) :: δas2)

splice+ : {as : SVDesc} → SigVec as → SigVec as → EventTime+ → SigVec as
splice+ {C _} s1 s2 te = spliceC s1 s2 te
splice+ {S _} s1 s2 te = spliceS s1 s2 te
splice+ {E _} s1 s2 te = spliceE s1 s2 te
splice+ {_,_} (sa1, sb1) (sa2, sb2) te = (splice+ sa1 sa2 te, splice+ sb1 sb2 te)

splice : {as : SVDesc} → SigVec as → SigVec as → EventTime → SigVec as
splice s1 s2 te | te == 0 = s2
                | te >  0 = splice+ s1 s2 te
```

B Signal-Function Conceptual Definitions

Some of the primitive signal functions in Section 5 were given without conceptual definitions. Those definitions are given in this appendix.

$$
\begin{array}{l}
\mathit{switch} : \{\mathit{as}\ \mathit{bs} : \mathit{SVDesc}\} \to \{A : \mathit{Set}\} \to (\mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A)) \to (A \to \mathit{SF}\ \mathit{as}\ \mathit{bs}) \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \\
\mathit{switch}\ \mathit{sf}\ f \approx \lambda\ s_a \to \mathbf{let}\ (s_b, s_e) = \mathit{sf}\ s_a \\
\qquad\qquad \mathbf{in}\ \mathit{svAtTime}\ (\lambda\ t \to \mathbf{case}\ \mathit{fstOcc}\ s_e\ t\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathsf{nothing} \to s_b \\
\qquad\qquad\qquad\qquad \mathsf{just}\ (t_e, e) \to \mathit{splice}\ s_b\ ((f\ e)\ (\mathit{advance}\ t_e\ s_a))\ t_e)
\end{array}
$$

$$
\begin{array}{l}
\mathit{integralS} : \mathit{SF}\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R}) \\
\mathit{integralS} \approx \lambda\ (a_0, \mathit{cp})\ t \to \mathbf{let}\ \delta\mathit{as} = \mathit{cp}\ t \\
\qquad\qquad\qquad\qquad\qquad\ \ \delta s = \mathit{map}\ \mathit{fst}\ \delta\mathit{as} +\!\!+ (t - \mathit{lastChangeTime}\ \delta\mathit{as}) :: [\,] \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{as} = a_0 :: \mathit{map}\ \mathit{snd}\ \delta\mathit{as} \\
\qquad\qquad\qquad\qquad\ \ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{sum}\ (\mathit{zipWith}\ (*)\ \delta s\ \mathit{as})
\end{array}
$$

The following conceptual definition of *when* is inspired by Wan and Hudak's definition [39]. The key difference, beside an adaptation to our model, is that we directly characterise when a time-varying Boolean is sufficiently well-behaved to make the definition of *when* meaningful. In Wan and Hudak's definition, this is indirect from the lack of a solution satisfying their stated semantic conditions. Intuitively, a time-varying Boolean is well-behaved if there exists some temporal imprecision $\varepsilon : \Delta t$ such that a list of *all* times at which there is a transition from false to true over any given interval is computable, with each time point within ε from its true occurrence time according to the ideal semantics. This means that a sampled implementation will converge to the ideal semantics for well-behaved predicate and signal combinations as the sampling interval tends to 0.

We first define a number of auxiliary predicates on time-varying Booleans $p : \mathit{Time} \to \mathit{Bool}$ and time points $t : \mathit{Time}$:

$$
\begin{array}{l}
P\ p\ t = \exists\,(\varepsilon : \Delta t).\,(\forall\,\tau \in (t-\varepsilon, t).\,\neg\,(p\ \tau)) \wedge (\forall\,\tau \in (t, t+\varepsilon).\,p\ \tau) \\
N\ p\ t = \exists\,(\varepsilon : \Delta t).\,(\forall\,\tau \in (t-\varepsilon, t).\,p\ \tau) \wedge (\forall\,\tau \in (t, t+\varepsilon).\,\neg\,(p\ \tau)) \\
U\ p\ t = \exists\,(\varepsilon : \Delta t).\,(\forall\,\tau \in (t-\varepsilon, t+\varepsilon).\,\neg\,(p\ \tau)) \vee (\forall\,\tau \in (t-\varepsilon, t+\varepsilon).\,p\ \tau) \\
P_l\ p\ t = p\ t \wedge (\exists\,(\varepsilon : \Delta t).\,(\forall\,\tau \in (t-\varepsilon, t).\,\neg\,(p\ \tau)))
\end{array}
$$

P holds if p has a positive transition at point t, N if p has a negative transition at this point, and U if p is unchanging at this point. Note that P and N are not concerned with the value of p at t, only that there exist open intervals to the left and right of the point where p is constant. Similarly, U holds if there exists a neighbourhood around t, this time including t, where p is constant. Finally, P_l is the left-biased version of P that only considers an interval to the left of t. However, this time, the value of p at t is considered.

We can now define a predicate W that holds if a time-varying Boolean is well-behaved on an open interval (t_0, t_1):

$$W\ p\ t_0\ t_1 = \text{finite}\,\{\tau \mid \tau \in (t_0, t_1), P\ p\ \tau\} \wedge (\forall\,\tau \in (t_0, t_1).\,P\ p\ \tau \vee N\ p\ \tau \vee U\ p\ \tau)$$

The finiteness condition rules out the time-varying Boolean oscillating between false and true infinitely often over a finite interval. The second part says that it must be possible to characterise every interior point either as a positive transition, a negative transition, or a point where no change occurs. This rules out "spikes": points where the value of the time-varying Boolean differs from its value in all neighbourhoods of that point.

The finite ascending list of time points of positive transitions for a time-varying Boolean p over an interval $(0, t]$ can now be defined:

```
poccs : (Time → Bool) → Time → List Time
poccs p t | W p 0 t = [τ | τ ← (0,t),P p τ] ++ [t | P_l p t]
```

Finally, we define *when* using *poccs*. Note that *poccs* is only defined for well-behaved time-varying Booleans. The semantics of *when* applied to an ill-behaved predicate and signal composition is thus that it diverges:

```
when : {A : Set} → (A → Bool) → SF (C A) (E A)
when p ≈ λ s → (nothing, whenAux)
  where
    whenAux t = let ts = poccs (p ∘ s) t
                in [(t − t', s t) | (t,t') ← zip ts (0 :: ts)]
```