Safe Functional Reactive Programming through Dependent Types

(Corrected Version, 14th December 2011)

Neil Sculthorpe Henrik Nilsson

School of Computer Science
University of Nottingham
United Kingdom
{nas,nhn}@cs.nott.ac.uk

Abstract

Functional Reactive Programming (FRP) is an approach to reactive programming where systems are structured as networks of functions operating on signals. FRP is based on the synchronous data-flow paradigm and supports both 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.

Statically guaranteeing correctness properties of programs is an attractive proposition. This is true in particular for typical application domains for reactive programming such as embedded systems. To that end, many existing reactive languages have type systems or other static checks that guarantee domain-specific properties, such as feedback loops always being well-formed. However, they are limited in their capabilities to support dynamism and higher-order data-flow compared with FRP. Thus, the onus of ensuring such properties of FRP programs has so far been on the programmer as established static techniques do not suffice.

In this paper, we show how dependent types allow this concern to be addressed. We present an implementation of FRP embedded in the dependently-typed language Agda, leveraging the type system of the host language to craft a domain-specific (dependent) type system for FRP. The implementation constitutes a discrete, operational semantics of FRP, and as it passes the Agda type, coverage, and termination checks, we know the operational semantics is total, which means our type system is safe.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages, data-flow languages, specialized application languages

General Terms Languages

Keywords dependent types, domain-specific languages, DSELs, FRP, functional programming, reactive programming, synchronous data-flow

1. Introduction

Functional Reactive Programming (FRP) grew out of Conal Elliott's and Paul Hudak's work on Functional Reactive Animation [Elliott and Hudak 1997]. 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. 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. The nature of the signals depends on the application domain. Examples include input from sensors in robotics applications [Peterson et al. 1999], video streams in the context of graphical user interfaces [Courtney and Elliott 2001] and games [Courtney et al. 2003, Cheong 2005], and synthesised sound signals [Giorgidze and Nilsson 2008].

A number of FRP variants exist. However, *the synchronous data-flow principle*, and support for both *continuous* and *discrete* time (*hybrid systems*), are common to most of them. There are thus close connections to synchronous data-flow languages such as Esterel [Berry and Gonthier 1992], Lustre [Halbwachs et al. 1991], and Lucid Synchrone [Caspi and Pouzet 1996, Pouzet 2006]; hybrid automata [Henzinger 1996]; and languages for hybrid modelling and simulation, such as Simulink [sim 2010]. However, FRP goes beyond most of these approaches by supporting *dynamism* (highly-dynamic system structure), and first-class signal functions (also known as *higher-order data-flow*).

Dynamism and higher-order data-flow are becoming important aspects of reactive programming as they are essential for implementing reconfigurable systems, including systems that receive software updates whilst running, which are increasingly prevalent [Colaço et al. 2004]. Statically guaranteeing central domain-specific correctness properties is consequently also becoming much more important, as dynamism and higher-order data-flow add levels of system complexity which make it correspondingly harder to test systems sufficiently thoroughly. Moreover, in many reactive application scenarios, the cost of failure is very high (for example, manual intervention may not be feasible: consider updating the software of a robot on Mars), thereby making it imperative to statically guarantee that the system will not fail.

Yampa [Nilsson et al. 2002] is an embedding of FRP in Haskell that supports dynamism (more so than previous Haskell-based FRP implementations) and first-class signal functions. However, from the perspective of reactive programming, the Haskell-based type system of Yampa is arguably not *safe*, as it does not enforce important domain-specific correctness properties. For example, there is nothing that prevents ill-formed feedback loops, which, if present,

can cause deadlock. Furthermore, even if a Yampa program is initially well-formed, there are no guarantees that it will remain so after dynamic reconfiguration. Conversely, there are reactive languages that statically do enforce such domain-specific properties (Lustre and Lucid Synchrone, for example), but their support for dynamism or higher-order data-flow is limited.

To address this problem, we develop a domain-specific type system for FRP that guarantees two central domain-specific correctness properties, well-formed feedback loops and proper initialisation, while still allowing for dynamism and first-class reactive entities. The type system is *safe* in that it guarantees that reactive programs are *productive* (guaranteed to deliver output at all points in time), under the assumption that the pure functional code embedded in the signal-processing network is total and terminating. This is accomplished through the domain-specific type system being dependent [Thompson 1991, Pierce 2002]: the types of signal functions are indexed on specific properties that they satisfy, allowing the corresponding properties of composite networks to be established compositionally through type-level computations.

The type system has been realised in the context of a prototype FRP implementation embedded in Agda [Norell 2007], a dependently-typed functional language. Agda bears many similarities to Haskell, but requires all functions to be total and terminating. The central part of the implementation is a function that constitutes a discretised operational semantics: given the time passed since the previous step and the current input, this semantic function maps a well-typed term representing the current configuration of (part of) a signal function network to the current output and a new, well-typed term representing the updated configuration. Because the semantic function is total and terminating, it constitutes a proof that the embedded type system guarantees the productivity of well-typed signal-function networks, which is the safety property with which we are concerned here.

A further benefit of making domain-specific properties manifest in the types of signal functions is that this clarifies their semantics, which, in turn, offers strong guidance as to their proper use. This is in stark contrast to Yampa, where subtle but crucial properties are often implicit, possibly leading to confusion about the exact relation between differently named combinators with the same type.

There are a couple of other innovative aspects to the FRP version developed in this paper. Firstly, there is a clear type-level distinction between continuous-time and discrete-time signals. In Yampa, the latter are just continuous-time signals carrying an option type. As a result, certain signal functions, such as the various delays, that in order to guarantee desirable semantical properties would have to treat continuous-time and discrete-time signals differently, actually treat them uniformly. This is another source of subtle bugs that can be eliminated by the more precise type system presented in this paper. Secondly, our development is structured around n-ary signal functions, through the notion of *signal vectors*. This enables a number of important optimisations, such as change propagation, to an extent that is not possible in Yampa [Sculthorpe and Nilsson 2009].

Note that our FRP type-system is, in principle, independent of the Agda-based FRP implementation presented here: it could be used with other realisations of FRP.

In summary, the main contributions of this paper are:

- A type system for FRP that
  - enforces well-formed feedback loops and proper initialisation;
  - guarantees productivity if all pure functions embedded in a network are total and terminating;
  - makes a clear type-level separation between continuous-time and discrete-time signals, ruling out additional kinds of ill-formed programs.
- A discrete, operational semantics for the version of FRP used in this paper.
- A machine-checked proof of the safety of the type system carried out through an embedding of the type system and the operational semantics in the dependently-typed language Agda. We outline the proof in the present paper; the full code is available from the first author's website[1].

The rest of the paper is structured as follows. Section 2 explains the fundamental concepts of FRP. Section 3 describes a new conceptual model that addresses some of the limitations in previous FRP models. Sections 4 and 5 demonstrate how the new conceptual model allows us to include feedback loops and uninitialised signals in an FRP program, whilst guaranteeing productivity at the type level. Finally, Section 6 describes a prototype FRP implementation using this type system, and gives its operational semantics.

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. In the case of Yampa, the host language is Haskell. The reactive level is concerned with *time-varying values*, which we call *signals*. At this level, combinators are used to construct *synchronous data-flow networks* by combining *signal functions*. The levels are, however, interdependent: the reactive level relies on the functional level for carrying out arbitrary pointwise computations on signals, while reactive entities, such as signal functions, are first class entities at the functional level.

2.1 Continuous-Time Signals

The core conceptual idea of FRP is that time is continuous. Signals are modelled as functions from time to value, where we take time to be the set of non-negative real numbers:

$$Time = \{\, t \in \mathbb{R} \mid t \geqslant 0 \,\}$$
$$Signal\ a \approx Time \to a$$

This conceptual model provides the foundation for an ideal FRP semantics. Of course, any digital implementation of FRP 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 this sampling rate is fixed, nor how this 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 [Wan and Hudak 2000]. But even then it is hard to know how densely one needs to sample before an answer is acceptably close to the ideal.

However, the focus of this paper is not directly the faithfulness of FRP implementations to any ideal semantics. Instead, our interest is to statically *rule out* programs that are bad; either because

[1] http://www.cs.nott.ac.uk/∼nas/icfp09.html

they lack meaning, or because they would be hard to run faithfully. This, in turn, is one step towards making it easier to implement FRP faithfully and allowing programmers to reason in terms of the ideal semantics with greater confidence. Thus, in this paper, we only provide a discrete operational FRP semantics as this is what we need for our purposes. But we will continue to refer to the ideal, conceptual model when it is expedient for providing the right intuitions.

2.2 Signal Functions

Signal functions are conceptually functions from signal to signal:

$$SF\ a\ b\ \approx\ Signal\ a \rightarrow Signal\ b$$

In Yampa, signal functions, rather than signals, are first class entities. Signals have no independent existence of their own; they exist only indirectly through the signal functions.

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

Causal Signal Function. A signal function is causal if, at any given time, its output can depend upon its past and present inputs, but not its future inputs:

$$\begin{aligned} SF\ a\ b\ =\ \{ & sf : Signal\ a \rightarrow Signal\ b \\ & \mid \forall\,(t : Time)\,(s_1\ s_2 : Signal\ a)\,. \\ & \quad (\forall\ t' \leqslant t.\ s_1\ t' \equiv s_2\ t') \Rightarrow (sf\ s_1\ t \equiv sf\ s_2\ t)\} \end{aligned}$$

In an implementation, signal functions that depend upon past inputs need to record past information in an internal state. For this reason, they are often called *stateful* signal functions.

Some signal functions are such that their output only depends on their input at the *current* point in time. We refer to these as *stateless* signal functions, as they require no internal state to be implemented:

$$\begin{aligned} SF_{stateless}\ a\ b\ =\ \{ & sf : Signal\ a \rightarrow Signal\ b \\ & \mid \forall\,(t : Time)\,(s_1\ s_2 : Signal\ a)\,. \\ & \quad (s_1\ t \equiv s_2\ t) \Rightarrow (sf\ s_1\ t \equiv sf\ s_2\ t)\} \end{aligned}$$

The terms *sequential* and *combinatorial* are also used for the same notions as stateful and stateless, respectively.

2.3 Why Not First Class Signals?

In Classic FRP (CFRP) [Elliott and Hudak 1997, Wan and Hudak 2000], the first class entities are *behaviours*, which are time-varying values corresponding to signals:

$$Behaviour\ a\ \approx\ Time \rightarrow a$$

CFRP programs are constructed by applying functions to behaviours, making CFRP programs look more like conventional functional programs than Yampa programs do. This is appealing in many ways. However, unless great care is exercised, first-class behaviours can lead to a number of performance problems. There are also thorny semantic problems related to composing behaviours temporally by switching from one to another [Nilsson et al. 2002].

In part to avoid these issues, the notion of signal function was adopted as the core concept for Yampa. The absence of first class signals makes it simple to process input as it arrives, which is the norm for synchronous data-flow languages. The semantics of switching also becomes obvious, paving the way for supporting structural dynamism [Nilsson et al. 2002].

This is not to say that first-class behaviours cannot be a viable approach in many cases: Elliott's recent work on the *Reactive* library has clearly shown this is not so [Elliott 2009]. However, we have chosen to stay with signal functions as the core concept because of its simplicity, robustness, and demonstrated flexibility.

3. The New Conceptual Model

In this section, we introduce a new conceptual FRP model that addresses some limitations of the Yampa design. We have discussed these problems in earlier work [Sculthorpe and Nilsson 2009], along with an initial version of this new model. In the following, we briefly review the problems of the Yampa design, and then introduce a refined version of the new model adapted to the setting of the present paper. With the new model as a basis, we then continue to develop a type system guaranteeing safety in the following sections.

3.1 Limitations of the Yampa Design

In Yampa, multiple signals are combined by tupling them together. There is no distinction between a pair of signals and a signal carrying a pair. For example, a signal function that conceptually maps a pair of signals carrying doubles to another pair of signals carrying doubles has the type:

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

This is exactly the same type as a signal function that maps a signal carrying pairs of doubles to another signal carrying pairs of doubles. Routing of signals between signal functions is mostly carried out at the functional level by lifting pure routing functions to the reactive level.

Unfortunately, this approach hides the routing from the reactive level, making it difficult to implement Yampa in a way that scales well (such as through direct point-to-point communication or change propagation [Sculthorpe and Nilsson 2009]). To overcome this, routing needs to be internalised at the reactive level, and the signal function notion needs to be refined so that a signal function truly maps multiple *individual* input signals to multiple *individual* output signals.

Another characteristic aspect of the Yampa design is that discrete-time signals are realised by continuous-time signals carrying an option type ($Signal\ (Maybe\ A)$). This is very convenient, as continuous-time and discrete-time signals can be freely mixed, but alas not sufficiently abstract: the ideal semantics of discrete-time signals cannot really be enforced, nor can it be exploited for optimising the implementation. It is thus desirous to make a clear type-level distinction between continuous-time signals and discrete-time signals, while retaining the convenience of the Yampa approach.

3.2 Signal Descriptors and Signal Vectors

To address the limitations of Yampa, we introduce the notion of a *signal vector*, a heterogeneous vector of signals, and redefine the conceptual notion of signal function to be a function on signal vectors. We also introduce two distinct kinds of signals: continuous-time signals, defined as before; and discrete-time, or *event*, signals, which are only defined at countably many points in time. Each point at which an event signal is defined is known as an *event occurrence*.

The crucial point is that we define these notions of different kinds of signals, 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 and exploiting those choices.

We proceed as follows. First we define *signal descriptors*. A signal descriptor is a *type* that describes key characteristics of a signal. Signal descriptors only exist at the type-level: there are no values having such types; in particular, a signal descriptor is not the (abstract) type of any signal.

Initially, we are interested in the time domain and the type (of the values carried by) the signal. Thus we introduce one descriptor for each kind of signal, each parametrised on the signal type:

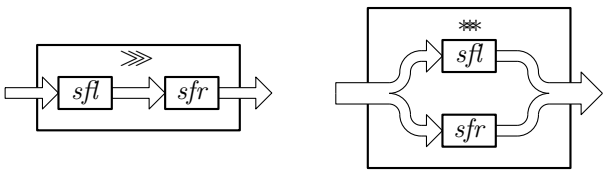

Figure 1. The Sequential ($\ggg$) and Parallel ($\ast\!\!\ast\!\!\ast$) Composition Combinators

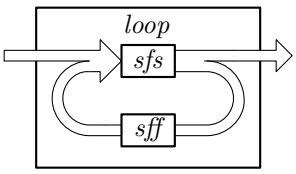

Figure 2. The Feedback Combinator ($loop$)

```
data SigDesc : Set where
  E : Set → SigDesc   -- discrete-time signals (events)
  C : Set → SigDesc   -- continuous-time signals
```

Note that Set is the "type of types" in Agda (similar to kind $*$ in Haskell)[2].

Next we introduce *signal vector descriptors*. A signal vector descriptor is simply a (type level) list of signal descriptors:

```
SVDesc : Set
SVDesc = List SigDesc
```

For the purpose of stating the new conceptual definition of signal functions, and for use in semantic definitions later, we postulate a function ($SVRep$) that maps a signal vector descriptor to some suitable type for representing a *sample* of signal vectors of that description, and use this to define signal vectors:

```
SVRep : SVDesc → Set
SigVec : SVDesc → Set
SigVec as ≈ Time → SVRep as
```

However, we do not require the existence of such a function: an implementation may opt to not represent signal vectors explicitly at all.

Finally, we refine the conceptual definition of signal functions:

```
SF : SVDesc → SVDesc → Set
SF as bs ≈ SigVec as → SigVec bs
```

3.3 Example Combinators and Primitives

To demonstrate the new conceptual model, we define some common primitive signal functions and combinators from Yampa. These primitives either operate at the reactive level, or mediate between the functional and reactive levels.

3.3.1 Sequential and Parallel Composition

Signal functions can be composed sequentially ($\ggg$) or in parallel ($\ast\!\!\ast\!\!\ast$) (see Figure 1):

```
_⋙_ : {as bs cs : SVDesc} →
      SF as bs → SF bs cs → SF as cs
_⁂_ : {as bs cs ds : SVDesc} →
      SF as cs → SF bs ds → SF (as ++ bs) (cs ++ ds)
```

(In Agda, _ is used to indicate the argument positions for infix and mixfix operators, while the 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.) Note that $\ast\!\!\ast\!\!\ast$ composes two signal functions that take different inputs. For parallel composition where both signal functions take the same input, there is the &&& combinator:

```
_&&&_ : {as bs cs : SVDesc} →
        SF as bs → SF as cs → SF as (bs ++ cs)
```

3.3.2 Switches

Signal function networks are made dynamic through the use of *switches*. Basic switches have the following type:

```
switch  : ∀ {as bs} → {e : Set} →
          SF as (E e :: bs) → (e → SF as bs) → SF as bs
dswitch : ∀ {as bs} → {e : Set} →
          SF as (E e :: bs) → (e → SF as bs) → SF as bs
```

(Agda allows the type of an implicit argument to be omitted when it is clear from the context. In the definitions above, both as and bs are clearly of type $SVDesc$ as they are used as arguments to the type constructor SF.) The behaviour of a switch is to run the subordinate signal function (the first explicit argument), emitting all but the head (the event) of the output vector as the overall output. When there is an event occurrence in the event signal, the value of that signal is fed into the function (the second explicit argument) to generate a residual signal function. The entire switch is then removed from the network and replaced with this residual signal function.

The difference between a $switch$ and a $dswitch$ (decoupled switch) is whether, at the moment of switching, the overall output is the output from the residual signal function ($switch$), or the output from the subordinate signal function ($dswitch$).[3]

A key point regarding switches is that the residual signal function does not start "running" until it is applied to the input signal at the moment of switching. Consequently, rather than having a single global $Time$, each signal function has its own *local time*.

Local Time. The time since this 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.

3.3.3 Loops

The $loop$ primitive provides the means for introducing feedback loops into signal function networks. A loop consists of two signal functions: a subordinate signal function (the first explicit argument) and a feedback signal function (the second explicit argument). The input of the feedback signal function is a suffix of the output of the subordinate signal function, and the output of the feedback signal function is a suffix of the input to the subordinate signal function:

```
loop : ∀ {as bs cs ds} →
       SF (as ++ cs) (bs ++ ds) → SF ds cs → SF as bs
```

Intuitively, we use the feedback signal function to connect some of the output signals of the subordinate signal function to some of its input signals, forming a feedback loop (see Figure 2).

[2] Strictly speaking, $SigDesc$ should have type $Set1$ (the type of Set). However, for clarity, we use the Agda option that accepts Set as the type of Set. We have successfully implemented the type system without this option, but, because Agda does not support universe polymorphism, the result is very repetitive code and loss of conceptual clarity.

[3] In Yampa, $dswitch$ also decouples part of its input from part of its output, but we do not assume any such behaviour here.

3.3.4 Primitive Signal Functions

We can lift pure functions to the reactive level using the primitives $pure$ and $pureE$[4]. Such lifted signal functions are always stateless:

$$pure \;\; : \{\, a\ b : Set \,\} \to (a \to b) \to SF\ [\ \mathsf{C}\ a\ ]\ [\ \mathsf{C}\ b\ ]$$
$$pureE : \{\, a\ b : Set \,\} \to (a \to b) \to SF\ [\ \mathsf{E}\ a\ ]\ [\ \mathsf{E}\ b\ ]$$

Note that we are using $[\ s\ ]$ as a synonym for $(s :: [])$.

We can lift *values* to the reactive level using the primitive $constant$. This creates a signal function with a constant, continuous-time, output:

$$constant : \forall\, \{\, as \,\} \to \{\, b : Set \,\} \to b \to SF\ as\ [\ \mathsf{C}\ b\ ]$$

Events can only be generated and accessed by event processing primitives. Examples include

- $edge$, which produces an event whenever the boolean input signal changes from false to true;
- $hold$, which emits as a continuous-time signal the value carried by its most recent input event;
- $never$, which outputs an event signal containing no event occurrences;
- now, which immediately outputs one event, but never does so again.

$$edge : SF\ [\ \mathsf{C}\ Bool\ ]\ [\ \mathsf{E}\ Unit\ ]$$
$$hold : \{\, a : Set \,\} \to a \to SF\ [\ \mathsf{E}\ a\ ]\ [\ \mathsf{C}\ a\ ]$$
$$never : \forall\, \{\, as \,\} \to \{\, b : Set \,\} \to SF\ as\ [\ \mathsf{E}\ b\ ]$$
$$now : \forall\, \{\, as \,\} \to SF\ as\ [\ \mathsf{E}\ Unit\ ]$$

The primitive pre conceptually introduces an infinitesimal delay:

$$pre : \forall\, \{\, a \,\} \to SF\ [\ \mathsf{C}\ a\ ]\ [\ \mathsf{C}\ a\ ]$$

To make this precise, the ideal semantics of pre is that it outputs whatever its input was *immediately prior* to the current time; that is, the *left limit* of the input signal at all points:

$$\forall\, (t : Time^{+})\ (s : Signal\ a).\ pre\ s\ t \;=\; \lim_{t' \to t^{-}} s\ t'$$

Here, $Time^{+}$ denotes positive time. Consequently, at any given point, the output of pre does not depend upon its present input, which is the crucial property of pre: see Section 4.

The primitive pre is usually implemented as a delay of one time step. Of course, this only approximates the ideal semantics. However, if the length of the time steps tends to zero, the semantics of such an implementation of pre converges to the ideal semantics.

Note that pre is only defined for continuous-time signals. This is because the left limit at any point of a discrete-time signal (a signal defined only at countably many points in time) is undefined. In our setting, this amounts to an event signal without any occurrences; which is a signal equivalent to the output from $never$. Applying pre to an event signal would thus be pointless (use $never$ instead), and any attempt to do so would likely be a mistake stemming from a misunderstanding of the semantics of pre. Disallowing pre on events thus eliminates a potential source of programming bugs.

In contrast, Yampa, because discrete-time signals are realised as continuous-time signals carrying an option type (see Section 3.1), cannot rule out pre being applied to event signals, nor can it guarantee the proper semantics of such an application.

Note also that pre is only defined for positive time. When the local time is zero (henceforth referred to as $time_0$), the output of pre is necessarily undefined as there are no prior points in time. Thus we need an $initialise$ combinator that defines a signal function's output at $time_0$:

$$initialise : \forall\, \{\, as\ b \,\} \to b \to SF\ as\ [\ \mathsf{C}\ b\ ] \to SF\ as\ [\ \mathsf{C}\ b\ ]$$

Initialisation is discussed further in Section 5.

[4] It is possible to have one $pure$ primitive that is overloaded to operate on either time domain, but we do not do so here for clarity.

3.4 Example

Let us illustrate the concepts and definitions that have been introduced thus far by constructing a simple signal function network. Its purpose is to monitor a real-valued continuous-time input signal and output the same signal until the input dips below 0. At this point, the output should be clamped to 0, and then remain at 0 from then on.

$$clamp : SF\ [\ \mathsf{C}\ \mathbb{R}\ ]\ [\ \mathsf{C}\ \mathbb{R}\ ]$$
$$clamp \;=\; switch\ ((pure\ (\lambda x \to x < 0) \ggg edge)\ \&\&\&\ pure\ id)\ (\lambda\_ \to constant\ 0)$$

4. Decoupled Signal Functions

As previously discussed, the $loop$ combinator allows feedback to be introduced into a network. This is an essential capability, as feedback is widely used in reactive programming.

However, feedback must not cause deadlock due to a signal function depending on its own output in an unproductive manner. To guarantee this, we conservatively prohibit *instantaneous cycles* in the network. This is a common design choice in reactive languages, but our way of enforcing it is different. We identify *decoupled* signal functions, essentially a class of signal functions that can be used safely in feedback loops, and index the type of a signal function by whether or not it is decoupled.

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:

$$SF_{dec}\ as\ bs \;=\; \{\, sf : SF\ as\ bs \mid \forall\, (t : Time)\ (sv_1\ sv_2 : SigVec\ as).\ (\forall\, t' < t.\ sv_1\ t' \equiv sv_2\ t') \Rightarrow (sf\ sv_1\ t \equiv sf\ sv_2\ t) \,\}$$

Decoupled Cycle. A cycle is decoupled if it passes through a decoupled signal function.

Instantaneous Cycle (Algebraic Loop). A cycle is instantaneous if it does not pass through a decoupled signal function.

In Yampa, the onus is on the programmer to ensure that all cycles are correctly decoupled. An instantaneous cycle will not be detected statically, and the program could well loop at run-time.

Many reactive languages deal with this problem by requiring a specific decoupling construct (a language primitive) to appear syntactically within the definition of any feedback loops. This works in a first order setting, but becomes very restrictive in a higher order setting as decoupled signal functions cannot be taken as parameters and used to decouple loops.

Our solution is to encode decoupledness information in the types of signal functions. This allows us to statically ensure that a well-typed program does not contain any instantaneous cycles. Furthermore, the decoupledness of a signal function will be visible in its type signature, providing guidance to an FRP programmer.

4.1 Decoupledness Descriptors

We introduce a data type of decoupledness descriptors:

```
data Dec : Set where
  dec : Dec   -- decoupled signal functions
  cau : Dec   -- causal signal functions
```

We then index SF with a decoupledness descriptor:

$$SF : SVDesc \to SVDesc \to Dec \to Set$$

We can now enforce that the feedback signal function within a *loop* is decoupled:

$$\begin{aligned} &loop : \forall\ \{as\ bs\ cs\ ds\} \to \{d : Dec\} \to \\ &\quad SF\ (as +\!\!+ cs)\ (bs +\!\!+ ds)\ d \to SF\ ds\ cs\ \mathsf{dec} \to \\ &\quad SF\ as\ bs\ d \end{aligned}$$

The primitive signal functions now need to be retyped to include appropriate decoupledness descriptors:

$$\begin{aligned} &pure : \forall\ \{a\ b\} \to (a \to b) \to SF\ [\ \mathsf{C}\ a\ ]\ [\ \mathsf{C}\ b\ ]\ \mathsf{cau} \\ &pureE : \forall\ \{a\ b\} \to (a \to b) \to SF\ [\ \mathsf{E}\ a\ ]\ [\ \mathsf{E}\ b\ ]\ \mathsf{cau} \\ &constant : \forall\ \{as\ b\} \to b \to SF\ as\ [\ \mathsf{C}\ b\ ]\ \mathsf{dec} \\ &edge : SF\ [\ \mathsf{C}\ Bool\ ]\ [\ \mathsf{E}\ Unit\ ]\ \mathsf{cau} \\ &hold : \forall\ \{a\} \to a \to SF\ [\ \mathsf{E}\ a\ ]\ [\ \mathsf{C}\ a\ ]\ \mathsf{cau} \\ &never : \forall\ \{as\ b\} \to SF\ as\ [\ \mathsf{E}\ b\ ]\ \mathsf{dec} \\ &now : \forall\ \{as\} \to SF\ as\ [\ \mathsf{E}\ Unit\ ]\ \mathsf{dec} \\ &pre : \forall\ \{a\} \to SF\ [\ \mathsf{C}\ a\ ]\ [\ \mathsf{C}\ a\ ]\ \mathsf{dec} \\ &initialise : \forall\ \{as\ b\} \to \{d : Dec\} \\ &\qquad \to b \to SF\ as\ [\ \mathsf{C}\ b\ ]\ d \to SF\ as\ [\ \mathsf{C}\ b\ ]\ d \end{aligned}$$

Notice that, from the definition of decoupled signal functions, it is evident that they are a subtype of causal signal functions ($\mathsf{dec} <: \mathsf{cau}$). This means that we can coerce a decoupled signal function into a causal one. Intuitively, we do this by forgetting that the decoupled signal function does not depend upon the present input. We provide a weakening primitive that performs this coercion:

$$\begin{aligned} &weaken : \forall\ \{as\ bs\ d\ d'\} \to \\ &\quad d <: d' \to SF\ as\ bs\ d \to SF\ as\ bs\ d' \end{aligned}$$

The primitive combinators compute the decoupledness descriptor of their composite signal function from the descriptors of their components. To do this, we use the join ($\vee$) and meet ($\wedge$) of the decoupledness descriptors (with respect to subtyping):

$$\begin{aligned} &\_\ggg\_ : \forall\ \{as\ bs\ cs\ d_1\ d_2\} \to \\ &\quad SF\ as\ bs\ d_1 \to SF\ bs\ cs\ d_2 \to SF\ as\ cs\ (d_1 \wedge d_2) \\ &\_***\_ : \forall\ \{as\ bs\ cs\ ds\ d_1\ d_2\} \to \\ &\quad SF\ as\ cs\ d_1 \to SF\ bs\ ds\ d_2 \to \\ &\quad SF\ (as +\!\!+ bs)\ (cs +\!\!+ ds)\ (d_1 \vee d_2) \\ &\_\&\&\&\_ : \forall\ \{as\ bs\ cs\ d_1\ d_2\} \to \\ &\quad SF\ as\ bs\ d_1 \to SF\ as\ cs\ d_2 \to \\ &\quad SF\ as\ (bs +\!\!+ cs)\ (d_1 \vee d_2) \\ &switch : \forall\ \{as\ bs\ e\ d_1\ d_2\} \to \\ &\quad SF\ as\ (\mathsf{E}\ e :: bs)\ d_1 \to (e \to SF\ as\ bs\ d_2) \to \\ &\quad SF\ as\ bs\ (d_1 \vee d_2) \\ &dswitch : \forall\ \{as\ bs\ e\ d_1\ d_2\} \to \\ &\quad SF\ as\ (\mathsf{E}\ e :: bs)\ d_1 \to (e \to SF\ as\ bs\ d_2) \to \\ &\quad SF\ as\ bs\ (d_1 \vee d_2) \end{aligned}$$

We have now ensured that all feedback in the network is well defined. In Yampa, badly defined cycles can cause the execution to loop at run-time, something that our type system guarantees will not occur.

Note that without indexing signal functions by their decoupledness, an Agda implementation using this type system (such as the one described in Section 6) would not pass Agda's termination checker.

4.2 Example: Switching Integration Methods

To demonstrate the usefulness of decoupledness descriptors, we give here a small example of how they can be used to allow dynamic switching between several integration signal functions. This is inspired by an example from Lucid Synchrone [Colaço et al. 2004], the synchronous data-flow language with the greatest similarity to FRP. Our example differs in that the integration signal function is being used to decouple a loop, allowing the decision of whether to switch integration functions to depend upon the current output.

To our knowledge, there is no other reactive language that could accept this program while also guaranteeing the absence of deadlock at run-time.

4.2.1 Recurring Switches

For this example we need to introduce an additional class of switching combinators: recurring switches (similar to *every* in Lucid Synchrone). The behaviour of a recurring switch is to apply its subordinate signal function to the tail of its input, producing the overall output. Whenever an event (the head of the input) occurs, the signal function carried by that event replaces the subordinate signal function. Recurring switches come in two varieties: like basic switches, they differ in whether the output at the instant of switching is from the new (*rswitch*) or old (*drswitch*) subordinate signal function.

$$\begin{aligned} &rswitch : \forall\ \{as\ bs\ d_1\ d_2\} \to SF\ as\ bs\ d_1 \to \\ &\quad SF\ (\mathsf{E}\ (SF\ as\ bs\ d_2) :: as)\ bs\ \mathsf{cau} \\ &drswitch : \forall\ \{as\ bs\ d_1\ d_2\} \to SF\ as\ bs\ d_1 \to \\ &\quad SF\ (\mathsf{E}\ (SF\ as\ bs\ d_2) :: as)\ bs\ (d_1 \vee d_2) \end{aligned}$$

4.2.2 Dynamic Integrator

There are many different ways to define an integration signal function, usually involving a trade-off between efficiency and accuracy. Some definitions of integration allow for decoupled behaviour, while others do not.

We first introduce some synonyms. We assume the signal to be integrated (*Input*) is a floating point number, as is the integrated signal (*Output*). An integrator (*Intgr*) is a causal signal function that integrates a signal, and a decoupled integrator (*dIntgr*) is one that is decoupled. We assume *integral* is an existing decoupled integrator.

$$\begin{aligned} Input &= Float \\ Output &= Float \\ Intgr &= SF\ [\ \mathsf{C}\ Input\ ]\ [\ \mathsf{C}\ Output\ ]\ \mathsf{cau} \\ dIntgr &= SF\ [\ \mathsf{C}\ Input\ ]\ [\ \mathsf{C}\ Output\ ]\ \mathsf{dec} \\ &integral : dIntgr \end{aligned}$$

Sometimes it can be advantageous to swap between integrators dynamically if the behaviour of the input signal changes radically (making sure to transfer the state). For the purposes of this example, we assume that we have a signal function (*intgrDecider*) that contains several integrators, and is capable of deciding, given the current input and output, whether to switch to one of them. For this, we require all such integrators to be decoupled (as the output is required to decide which integrator to use to compute that very output), and use them as the feedback signal function within a loop. The *intgrDecider* also receives as input new integrators, along with some decision rules that allow it to determine when they should be used.

We can now define a dynamic integrator (*dynIntgr*) that uses a *loop* to connect *intgrDecider* with a *drswitch* containing an initial decoupled integrator:

$$\begin{aligned} &DecisionRules : Set \\ &NewIntgr = dIntgr \times DecisionRules \\ &intgrDecider : SF\ (\mathsf{E}\ NewIntgr :: \mathsf{C}\ Input :: \mathsf{C}\ Output :: [\,]) \\ &\qquad\qquad (\mathsf{C}\ Output :: \mathsf{E}\ dIntgr :: \mathsf{C}\ Input :: [\,]) \\ &\qquad\qquad \mathsf{cau} \\ &dynIntgr : SF\ (\mathsf{E}\ NewIntgr :: \mathsf{C}\ Input :: [\,])\ [\ \mathsf{C}\ Output\ ]\ \mathsf{cau} \\ &dynIntgr = loop\ intgrDecider\ (drswitch\ integral) \end{aligned}$$

This program could be defined in Yampa. However, there would be no restriction on the decoupledness of the integrators. It would be possible to provide a new, non-decoupled integrator as input, which would then cause deadlock if it was ever used. Here, the types ensure this will never happen.

5. Uninitialised Signals

Recall the *pre* signal function from Section 3.3. This primitive is very common in reactive programming, and is the standard means of decoupling feedback. However, the output of *pre* is undefined at $time_0$. It is for this reason that the *initialise* primitive exists, which defines the output of a signal function at $time_0$.

It is possible to sidestep this issue by combining *pre* and *initialise* into one combinator (such as the *iPre* primitive in Yampa). We do not do this for several reasons:

- We may not want, or be able (if no initialisation value is available), to initialise the signal at the usage of *pre*, but only elsewhere in the program.
- An uninitialised signal can pass through a stateless signal function without causing an error, it just produces an uninitialised output signal.
- Some signals may not require initialising, such as within the residual signal function of a *dswitch* (see Section 5.2).

In Yampa, the onus is on the programmer to ensure that any uninitialised signals are correctly initialised where necessary. When this is not done correctly, it can cause run-time errors.

5.1 Initialisation Descriptors

Our solution is to add initialisation information to the signal descriptors. This guides the FRP programmer when writing programs, and allows the type checker to reject any programs where uninitialised signals could cause a run-time error. (Without this addition we could not implement uninitialised signals in Agda using this type system; the totality checker would reject it.)

Note that the property of a signal being defined or not at $time_0$ is only of interest for continuous-time signals. Event signals are, by definition, only defined at discrete points in time, and thus there is no need to initialise them if they are not defined at $time_0$.

```
data Init : Set where
  ini : Init   -- initialised signals
  uni : Init   -- uninitialised signals

data SigDesc : Set where
  E : Set → SigDesc
  C : Init → Set → SigDesc
```

The primitive signal functions that mention continuous-time signals in their types now need to be retyped:

```
pure : ∀ {a b} → {i : Init} →
       (a → b) → SF [ C i a ] [ C i b ] cau

constant : ∀ {as b} → b → SF as [ C ini b ] dec

edge : SF [ C ini Bool ] [ E Unit ] cau

hold : ∀ {a} → a → SF [ E a ] [ C ini a ] cau

pre : ∀ {a} → SF [ C ini a ] [ C uni a ] dec

initialise : ∀ {as b d} → {i : Init} →
             b → SF as [ C i b ] d → SF as [ C ini b ] d
```

Initialised signals are subtypes of uninitialised signals (ini <: uni), as they can be coerced by forgetting the value at $time_0$. We extend the *weaken* primitive to reflect this:

```
weaken : ∀ {as as' bs bs' d d'} →
         as' <: as → bs <: bs' → d <: d' →
         SF as bs d → SF as' bs' d'
```

Note that, as is usual for function types, the subtyping is contravariant on the input signal vector. A signal vector is a subtype of another if all signals in the former are subtypes of the respective signals in the latter (we do not use any width subtyping).

5.2 Switching into Uninitialised Signals

Our signal descriptors allow us to deal with uninitialised signals at $time_0$, but, crucially, not at any other time. As previously discussed, signal functions exist in their own local time. What is $time_0$ for one part of the system may not be for another part. We must ensure that an uninitialised signal does not "escape" from a subnetwork that locally is at $time_0$, into an outer network that is not.

For example, consider *switch*. When the residual signal function is switched in, it could produce uninitialised output at its (local) $time_0$. But this uninitialised signal then escapes, potentially causing a run-time error.

In fact, switches are the only place that this can occur, as it is only switches that create sub-networks at a different local time.

We resolve this by requiring that all output signals from the residual signal function are initialised, enforcing this at the type level:

```
initc : SVDesc → SVDesc
initc = map initcAux
  where initcAux : SigDesc → SigDesc
        initcAux (E a)   = E a
        initcAux (C _ a) = C ini a

switch : ∀ {as bs e d1 d2} →
         SF as (E e :: bs) d1 →
         (e → SF as (initc bs) d2) →
         SF as bs (d1 ∨ d2)
```

We do not require this constraint for decoupled switches, as their output at $time_0$ is defined as being the output from the subordinate signal function. The ($time_0$) output from the residual signal function is discarded, so it does not matter if it is uninitialised.

Furthermore, this means that the initialisation of the residual signal function's output does not affect the overall initialisation of the switch construct. We redefine *dswitch* to reflect this flexibility:

```
dswitch : ∀ {as bs bs' e d1 d2} →
          SF as (E e :: bs) d1 →
          (e → SF as bs' d2) →
          initc bs <: bs' →
          SF as bs (d1 ∨ d2)
```

6. Safety and Semantics

Agda has completeness and termination checkers, ensuring that Agda programs are total and terminating. Thus our FRP embedding within Agda has these assurances as well.

To run signal functions, our prototype implementation operates by running the network iteratively over a discrete sequence of time steps. At each time step, the input is sampled and fed into the network, along with the time delta since the preceding time step. The network then updates any internal state, and produces an output sample. This is the same approach as taken by Yampa and many other reactive languages. We give the operational semantics of this in Figure 4. These semantics correspond directly to the Agda function we use to execute one step of a network ($\Longrightarrow$):

```
Δt : Set
Δt = Time

⟹ : ∀ {as bs d} →
     Δt × SF as bs d × SVRep as → SF as bs d × SVRep bs
```

This (one time step) evaluation function is accepted by Agda; therefore the evaluation of each time step is guaranteed to terminate,

$$
\begin{array}{l}
\textbf{data}\ SF : SVDesc \to SVDesc \to Dec \to Set\ \textbf{where} \\
\quad prim \quad : \forall \{ as\ bs\ State \} \to (\Delta t \to State \to SVRep\ as \to State \times SVRep\ bs) \quad \to State \to SF\ as\ bs\ \mathsf{cau} \\
\quad dprim \quad : \forall \{ as\ bs\ State \} \to (\Delta t \to State \to (SVRep\ as \to State) \times SVRep\ bs) \to State \to SF\ as\ bs\ \mathsf{dec} \\
\quad \_ \ggg \_ \quad : \forall \{ as\ bs\ cs\ d_1\ d_2 \} \to SF\ as\ bs\ d_1 \to SF\ bs\ cs\ d_2 \qquad \to SF\ as\ cs\ (d_1 \wedge d_2) \\
\quad \_ *** \_ \quad : \forall \{ as\ bs\ cs\ ds\ d_1\ d_2 \} \to SF\ as\ cs\ d_1 \to SF\ bs\ ds\ d_2 \qquad \to SF\ (as +\!\!+ bs)\ (cs +\!\!+ ds)\ (d_1 \vee d_2) \\
\quad loop \quad : \forall \{ as\ bs\ cs\ ds\ d \} \to SF\ (as +\!\!+ cs)\ (bs +\!\!+ ds)\ d \to SF\ ds\ cs\ \mathsf{dec} \qquad \to SF\ as\ bs\ d \\
\quad switch \quad : \forall \{ as\ bs\ e\ d_1\ d_2 \} \to SF\ as\ (\mathsf{E}\ e :: bs)\ d_1 \to (e \to SF\ as\ bs\ d_2) \qquad \to SF\ as\ bs\ (d_1 \vee d_2) \\
\quad dswitch : \forall \{ as\ bs\ e\ d_1\ d_2 \} \to SF\ as\ (\mathsf{E}\ e :: bs)\ d_1 \to (e \to SF\ as\ bs\ d_2) \qquad \to SF\ as\ bs\ (d_1 \vee d_2)
\end{array}
$$

Figure 3. Implementation Core Primitives

$$\frac{f\ \delta t\ s\ as \mapsto (s'\ ,\ bs)}{(prim\ f\ s\ ,\ as) \xRightarrow{\delta t} (prim\ f\ s'\ ,\ bs)}\ \text{PRIM}$$

$$\frac{f\ \delta t\ s \mapsto (g\ ,\ bs) \qquad g\ as \mapsto s'}{(dprim\ f\ s\ ,\ as) \xRightarrow{\delta t} (dprim\ f\ s'\ ,\ bs)}\ \text{DPRIM}$$

$$\frac{(sfl\ ,\ as) \xRightarrow{\delta t} (sfl'\ ,\ bs) \qquad (sfr\ ,\ bs) \xRightarrow{\delta t} (sfr'\ ,\ cs)}{(sfl \ggg sfr\ ,\ as) \xRightarrow{\delta t} (sfl' \ggg sfr'\ ,\ cs)}\ \text{SEQ}$$

$$\frac{svsplit\ asbs \mapsto (as,\ bs) \qquad (sfl\ ,\ as) \xRightarrow{\delta t} (sfl'\ ,\ cs) \qquad (sfr\ ,\ bs) \xRightarrow{\delta t} (sfr'\ ,\ ds) \qquad cs +\!\!+ ds \mapsto csds}{(sfl *** sfr\ ,\ asbs) \xRightarrow{\delta t} (sfl' *** sfr'\ ,\ csds)}\ \text{PAR}$$

$$\frac{sff \xRightarrow{\delta t}_{\phi_1} (sff_\phi\ ,\ cs) \qquad as +\!\!+ cs \mapsto ascs \qquad (sfs\ ,\ ascs) \xRightarrow{\delta t} (sfs'\ ,\ bsds) \qquad svsplit\ bsds \mapsto (bs\ ,\ ds) \qquad (sff_\phi\ ,\ ds) \xRightarrow{\delta t}_{\phi_2} sff'}{(loop\ sfs\ sff\ ,\ as) \xRightarrow{\delta t} (loop\ sfs'\ sff'\ ,\ bs)}\ \text{LOOP}$$

$$\frac{(sf\ ,\ as) \xRightarrow{\delta t} (sf'\ ,\ NoEvent :: bs)}{(switch\ sf\ f\ ,\ as) \xRightarrow{\delta t} (switch\ sf'\ f\ ,\ bs)}\ \text{SW-NOEV}$$

$$\frac{(sfs\ ,\ as) \xRightarrow{\delta t} (sfs'\ ,\ Event\ e :: bss) \qquad f\ e \mapsto sfr \qquad (sfr\ ,\ as) \xRightarrow{0} (sfr'\ ,\ bsr)}{(switch\ sfs\ f\ ,\ as) \xRightarrow{\delta t} (sfr'\ ,\ bsr)}\ \text{SW-EV}$$

$$\frac{(sf\ ,\ as) \xRightarrow{\delta t} (sf'\ ,\ NoEvent :: bs)}{(dswitch\ sf\ f\ ,\ as) \xRightarrow{\delta t} (dswitch\ sf'\ f\ ,\ bs)}\ \text{DSW-NOEV}$$

$$\frac{(sfs\ ,\ as) \xRightarrow{\delta t} (sfs'\ ,\ Event\ e :: bss) \qquad f\ e \mapsto sfr \qquad (sfr\ ,\ as) \xRightarrow{0} (sfr'\ ,\ bsr)}{(dswitch\ sfs\ f\ ,\ as) \xRightarrow{\delta t} (sfr'\ ,\ bss)}\ \text{DSW-EV}$$

Figure 4. Operational Semantics for the $\xRightarrow{\delta t}$ Evaluation Relation

producing output. Although execution of an FRP program is usually modelled as non-terminating (an infinite sequence of steps), we can nevertheless guarantee that they are *productive* because each individual step is productive.

As our implementation allows feedback loops and uninitialised signals, this demonstrates that our type system allows for the construction of safe programs with these features.

6.1 Prototype Implementation

We will not discuss the implementation in detail, but give the key points that are required to understand the semantics.

Signal functions are represented internally as a data type, of which the constructors are five primitive combinators and two primitive signal functions (see Figure 3). We will call these the *core primitives*.

Note that this implementation does not allow arbitrary routing to be expressed at the *reactive level*; routing at the functional level is required for some network structures. This design choice was made for simplicity of presentation, as routing is not the primary concern of this paper. We stress that this is a limitation of this particular implementation, not of the type system in general.

The two core signal functions are $prim$ (for causal signal functions) and $dprim$ (for decoupled signal functions). The internal structures of these two primitives reflect the properties we require them to have.

A causal signal function must be able to produce output at the current time, provided it has access to all past and present inputs. We realise this by giving $prim$ an internal state (with which to record past inputs), and a function that maps the time delta (since the preceding time step), state and input to an updated state and output.

A decoupled signal function must be able to produce output at the current time, provided it has access to all past inputs. Thus its internal function requires only the time delta and state to produce an output. In order to manage updating the state, it also produces a function mapping input to an updated state. The key point here is that while the input will be required to fully evaluate the signal function at the current time step, the output can be produced before that input is provided.

Be aware that $prim$ and $dprim$ are intended to be hidden from the FRP programmer. The FRP primitives (such as $edge$ and pre, and many more not described in this paper) are defined in terms of them internally.

Notice that the semantics of $loop$ (rule LOOP in Figure 4) require the use of two auxiliary evaluation relations. These are *phase 1 evaluation* ($\Longrightarrow_{\phi_1}$) and *phase 2 evaluation* ($\Longrightarrow_{\phi_2}$), both of which are only defined on decoupled signal functions. They are required to allow partial evaluation of a decoupled signal function, producing its output without requiring its current input (phase 1), and then to allow this partially evaluated signal function (denoted SF_ϕ, see Figure 5) to be updated by providing the input (phase 2). The semantics of these two relations are given in Figures 6 and 7. The types of their corresponding Agda functions are:

$$\Longrightarrow_{\phi_1} : \forall\ \{as\ bs\} \to \Delta t \times SF\ as\ bs\ \mathsf{dec} \to SF_\phi\ as\ bs \times SVRep\ bs$$

$$\Longrightarrow_{\phi_2} : \forall\ \{as\ bs\} \to \Delta t \times SF_\phi\ as\ bs \times SVRep\ as \to SF\ as\ bs\ \mathsf{dec}$$

The semantics also make use of $svsplit$, an auxiliary function that splits a signal vector into two parts, determined by the required type of the output:

$$svsplit : \forall\ \{as\ bs\} \to SVRep\ (as +\!\!+ bs) \to SVRep\ as \times SVRep\ bs$$

We use $\mapsto$ to denote evaluation at the functional level.

Finally, we let the representation of an event signal be either $NoEvent$ or $Event\ e$; in the latter case e is the value carried by the event.

6.2 Semantics of Uninitialised Signals

The semantics given here omit any mention of signal initialisation. This is because these semantics only apply to time steps after the initialisation step (at $time_0$). The initialisation step requires a slightly different (and more complicated) set of semantic rules that we do not give in this paper. Be aware that when our evaluation rules are used with a zero time delta (as in SW-EVENT), we should actually be using the $time_0$ semantics.

7. Related Work

The synchronous data-flow languages [Benveniste et al. 2003, Halbwachs 1993, 1998] have long modelled reactive programs as synchronous data-flow networks. These languages usually have static first-order structures, allowing them to prevent undesirable network structures by performing a static analysis at compile time. While not having the expressiveness of FRP, they can provide strong space and time guarantees on their programs.

A typical example is Lucid Synchrone [Pouzet 2006], which does not allow instantaneous feedback loops, nor the construction of any nodes (signal functions) that produce uninitialised output. To guarantee that this is not the case, the decoupling and initialisation primitives (pre and $\rightarrow$) have to appear *syntactically* within a recursive node definition.

There has been recent work to extend Lucid Synchrone with higher order features [Colaço et al. 2004], setting it apart from the other synchronous languages. However, higher order nodes cannot be used to decouple feedback loops due to the aforementioned syntactic requirements.

FRP approaches the problem from the other direction. Most FRP implementations are highly expressive, but lack termination guarantees, and space and time bounds. Real-Time FRP (RT-FRP) [Wan et al. 2001], a small and experimental CFRP variant, is a notable exception that does provide some guarantees. RT-FRP disallows instantaneous feedback through the type system, which, like Lucid Synchrone, insists on the explicit insertion of the decoupling primitive ($delay$) into any recursive calls. However, RT-FRP has very limited capabilities for abstracting over and combining reactive entities, essentially only being concerned with monolithic reactive expressions. Also, there are no uninitialised signals in RT-FRP, as $delay$ requires an initialisation value as an additional parameter (like $iPre$ in Yampa).

FrTime [Cooper and Krishnamurthi 2006] is another recent FRP incarnation, though distinguished in that it is embedded in DrScheme rather than Haskell. FrTime takes the same approach to decoupling as Lucid Synchrone, requiring explicit decoupling and initialisation within a node definition.

8. Further Work

We have focused primarily on basic switches in this paper, giving brief mention to recurring switches. However, as demonstrated by Yampa [Nilsson et al. 2002], much more general switches are possible that allow for greater dynamism of network structure. Our next task is to extend our implementation with such switches.

Yampa is structured using *Arrows* [Hughes 2000], and thus many of its (and our) combinators are arrow combinators. Programming directly with arrow combinators is awkward for more complicated arrows, and so a syntactic sugar has been devised to aid writing of arrow code [Paterson 2001] (similar to monadic *do* notation). A recent development has been the *Arrow Calculus* [Lindley et al. 2008], an alternative (and equivalent) notation for arrows, with a

$$
\begin{array}{l}
\textbf{data}\ SF_\phi : SVDesc \to SVDesc \to Set\ \textbf{where} \\
\quad dprim_\phi \quad : \forall\{as\ bs\ State\} \to (\Delta t \to State \to (SVRep\ as \to State) \times SVRep\ bs) \to (SVRep\ as \to State) \to SF_\phi\ as\ bs \\
\quad \_\ggg_{\phi l}\_ \quad : \forall\{as\ bs\ cs\ d\} \to SF_\phi\ as\ bs \to SF\ bs\ cs\ d \to SF_\phi\ as\ cs \\
\quad \_\ggg_{\phi r}\_ \quad : \forall\{as\ bs\ cs\ d\} \to SF\ as\ bs\ d \to SF_\phi\ bs\ cs \to SF_\phi\ as\ cs \\
\quad \_***_\phi\_ \quad : \forall\{as\ bs\ cs\ ds\} \to SF_\phi\ as\ cs \to SF_\phi\ bs\ ds \to SF_\phi\ (as +\!\!+ bs)\ (cs +\!\!+ ds) \\
\quad loop_\phi \quad : \forall\{as\ bs\ cs\ ds\} \to SF_\phi\ (as +\!\!+ cs)\ (bs +\!\!+ ds) \to SF\ ds\ cs\ \mathsf{dec} \to SVRep\ cs \to SF_\phi\ as\ bs \\
\quad switch_\phi \quad : \forall\{as\ bs\ e\} \to SF_\phi\ as\ (\mathsf{E}\ e :: bs) \to (e \to SF\ as\ bs\ \mathsf{dec}) \to SF_\phi\ as\ bs \\
\quad dswitch_\phi : \forall\{as\ bs\ e\} \to SF_\phi\ as\ (\mathsf{E}\ e :: bs) \to (e \to SF\ as\ bs\ \mathsf{dec}) \to SF_\phi\ as\ bs
\end{array}
$$

Figure 5. Partially Evaluated Signal Functions

$$\frac{f\ \delta t\ s \mapsto (s_\phi\ ,\ bs)}{dprim\ f\ s \xRightarrow{\delta t}_{\phi_1} (dprim_\phi\ f\ s_\phi\ ,\ bs)}\ \Phi_1\text{-DPRIM}$$

$$\frac{sfl \xRightarrow{\delta t}_{\phi_1} (sfl_\phi\ ,\ bs) \qquad (sfr\ ,\ bs) \xRightarrow{\delta t} (sfr'\ ,\ cs)}{sfl \ggg sfr \xRightarrow{\delta t}_{\phi_1} (sfl_\phi \ggg_{\phi l} sfr'\ ,\ cs)}\ \Phi_1\text{-SEQ-L}$$

$$\frac{sfr \xRightarrow{\delta t}_{\phi_1} (sfr_\phi\ ,\ cs)}{sfl \ggg sfr \xRightarrow{\delta t}_{\phi_1} (sfl \ggg_{\phi r} sfr_\phi\ ,\ cs)}\ \Phi_1\text{-SEQ-R}$$

$$\frac{sfl \xRightarrow{\delta t}_{\phi_1} (sfl_\phi\ ,\ cs) \qquad sfr \xRightarrow{\delta t}_{\phi_1} (sfr_\phi\ ,\ ds) \qquad cs +\!\!+ ds \mapsto csds}{sfl *** sfr \xRightarrow{\delta t}_{\phi_1} (sfl_\phi ***_\phi sfr_\phi\ ,\ csds)}\ \Phi_1\text{-PAR}$$

$$\frac{sfs \xRightarrow{\delta t}_{\phi_1} (sfs_\phi\ ,\ bsds) \qquad svsplit\ bsds \mapsto (bs\ ,\ ds) \qquad (sff\ ,\ ds) \xRightarrow{\delta t} (sff'\ ,\ cs)}{loop\ sfs\ sff \xRightarrow{\delta t}_{\phi_1} (loop_\phi\ sfs_\phi\ sff'\ cs\ ,\ bs)}\ \Phi_1\text{-LOOP}$$

$$\frac{sf \xRightarrow{\delta t}_{\phi_1} (sf_\phi\ ,\ NoEvent :: bs)}{switch\ sf\ f \xRightarrow{\delta t}_{\phi_1} (switch_\phi\ sf_\phi\ f\ ,\ bs)}\ \Phi_1\text{-SW-NOEV}$$

$$\frac{sfs \xRightarrow{\delta t}_{\phi_1} (sfs_\phi\ ,\ Event\ e :: bss) \qquad f\ e \mapsto sfr \qquad sfr \xRightarrow{0}_{\phi_1} (sfr_\phi\ ,\ bsr)}{switch\ sfs\ f \xRightarrow{\delta t}_{\phi_1} (sfr_\phi\ ,\ bsr)}\ \Phi_1\text{-SW-EV}$$

$$\frac{sf \xRightarrow{\delta t}_{\phi_1} (sf_\phi\ ,\ NoEvent :: bs)}{dswitch\ sf\ f \xRightarrow{\delta t}_{\phi_1} (dswitch_\phi\ sf_\phi\ f\ ,\ bs)}\ \Phi_1\text{-DSW-NOEV}$$

$$\frac{sfs \xRightarrow{\delta t}_{\phi_1} (sfs_\phi\ ,\ Event\ e :: bss) \qquad f\ e \mapsto sfr \qquad sfr \xRightarrow{0}_{\phi_1} (sfr_\phi\ ,\ bsr)}{dswitch\ sfs\ f \xRightarrow{\delta t}_{\phi_1} (sfr_\phi\ ,\ bss)}\ \Phi_1\text{-DSW-EV}$$

Figure 6. Operational Semantics for the $\xRightarrow{\delta t}_{\phi_1}$ Evaluation Relation

$$\frac{s_\phi\ as \mapsto s'}{(dprim_\phi\ f\ s_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} dprim\ f\ s'}\ \Phi_2\text{-DPRIM}$$

$$\frac{(sfl_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sfl'}{(sfl_\phi \ggg_{\phi l} sfr'\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sfl' \ggg sfr'}\ \Phi_2\text{-SEQ-L}$$

$$\frac{(sfl\ ,\ as) \xrightarrow{\delta t} (sfl'\ ,\ bs) \qquad (sfr_\phi\ ,\ bs) \xrightarrow{\delta t}_{\phi_2} sfr'}{(sfl \ggg_{\phi r} sfr_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sfl' \ggg sfr'}\ \Phi_2\text{-SEQ-R}$$

$$\frac{svsplit\ asbs \mapsto (as,\ bs) \qquad (sfl_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sfl' \qquad (sfr_\phi\ ,\ bs) \xrightarrow{\delta t}_{\phi_2} sfr'}{(sfl_\phi ***_\phi sfr_\phi\ ,\ asbs) \xrightarrow{\delta t}_{\phi_2} sfl' *** sfr'}\ \Phi_2\text{-PAR}$$

$$\frac{as +\!\!+ cs \mapsto ascs \qquad (sfs_\phi\ ,\ ascs) \xrightarrow{\delta t}_{\phi_2} sfs'}{(loop_\phi\ sfs_\phi\ sff'\ cs\ ,\ as) \xrightarrow{\delta t}_{\phi_2} loop\ sfs'\ sff'}\ \Phi_2\text{-LOOP}$$

$$\frac{(sf_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sf'}{(switch_\phi\ sf_\phi\ f\ ,\ as) \xrightarrow{\delta t}_{\phi_2} switch\ sf'\ f}\ \Phi_2\text{-SW-NOEV}$$

$$\frac{(sf_\phi\ ,\ as) \xrightarrow{\delta t}_{\phi_2} sf'}{(dswitch_\phi\ sf_\phi\ f\ ,\ as) \xrightarrow{\delta t}_{\phi_2} dswitch\ sf'\ f}\ \Phi_2\text{-DSW-NOEV}$$

Figure 7. Operational Semantics for the $\xrightarrow{\delta t}_{\phi_2}$ Evaluation Relation

structure bearing more resemblance to the lambda calculus. We are interested in using the arrow calculus for programming at the reactive level (where it could replace our core routing combinators).

An important aspect of FRP is its capacity for dynamism and higher-order data-flow. It should be possible to exploit this by receiving new programs (signal functions) as system inputs at runtime. This is not yet possible in any FRP implementations, but there has been work in Haskell to allow dynamic loading of new code, which would make a good starting point for future work in this direction [Pang et al. 2004].

In this paper, we keep track of decoupledness by noting which signal functions are decoupled. We could take a more fine grained approach by recording the decoupledness of each output signal with respect to each input signal. This would give each signal function a matrix of decoupledness descriptors, allowing for far more precise tracking of decoupledness than the comparatively conservative method we use in this paper. For example, the following signal function would have all output signals decoupled from all input signals in such a setting, whereas here it is typed as causal:

```
parPre : SF (C ini ℝ :: C ini ℝ :: []) (C uni ℝ :: C uni ℝ :: []) cau
parPre = (pre *** pure id) >>> (pure id *** pre)
```

Such a setting would allow us, for example, to define a loop combinator similar to Yampa's, in which it is not necessary to explicitly separate a feedback signal function from the subordinate signal function.

Finally we note that our current prototype is a proof of concept implementation only. The long term aim of our work is to develop an efficient scalable implementation of FRP that incorporates the safety properties described in this paper. For example, while it would not be possible to *prove* the safety of an embedded implementation in Haskell in the manner we have done here, it might be possible to encode the domain-specific type constraints in a Haskell embedding. Note that it would not matter to the end use that the implementation does not carry the proof as long as the implementation is correct. However, earlier attempts of ours to work inside Haskell were discouraging (for example, there were problems encoding associativity of vector concatenation). That was in part what prompted us to switch to a framework where we would not have to worry about language restrictions to realise our design. But it may be that recent Haskell extensions such as type-level functions would suffice. The alternative would be to implement a more conventional stand-alone type checker. It may still be possible to do that in an embedded setting using *quasiquoting* [Mainland et al. 2008, Giorgidze and Nilsson 2011].

9. Conclusions

In this paper we presented a domain-specific type system for FRP. This type system ensures that signal function networks with feedback loops and uninitialised signals will be *productive*, without

sacrificing support for higher-order data-flow and structural dynamism.

The type system also makes a distinction between the *functional* and *reactive* levels of an FRP program, restricting some concepts (such as time domain) to the reactive level.

We demonstrated the expressiveness of the type system with a simple example of a well formed program (Section 4.2) that would either be statically rejected, or permit non-productive networks to be dynamically constructed, in any other reactive language we are aware of.

We have implemented a prototype interpreter for our FRP programs in Agda. We have given a simplified version of the operational semantics of this interpreter in this paper (omitting initialisation at $time_0$), and the full implementation is available from the first author's website.

As our implementation is accepted by Agda, we know it is safe and productive.[5] The implementation also constitutes a machine-checked proof of the safety of our type system for FRP, in principle allowing it to be used with confidence by other FRP implementations.

Acknowledgments

We would like to thank George Giorgidze, Nils Anders Danielsson and the anonymous reviewers for their helpful comments and feedback.

References

Simulink User's Guide, Version 7.6. 3 Apple Hill Drive, Natick, MA, 2010. URL `www.mathworks.com/help/toolbox/simulink/`.

Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on embedded systems*, 91(1):64–83, 2003.

Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming (ICFP '96)*, pages 226–238. ACM, 1996.

Mun Hon Cheong. Functional programming and 3D games. BEng thesis, University of New South Wales, 2005.

Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Embedded Software (EMSOFT '04)*, pages 230–239. ACM, 2004.

Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming (ESOP '06)*, pages 294–308. Springer, 2006.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop (Haskell '01)*, pages 41–69. Elsevier, 2001.

Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Haskell Workshop (Haskell '03)*, pages 7–18. ACM, 2003.

Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium (Haskell '09)*, pages 25–36. ACM, 2009.

Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP '97)*, pages 263–273. ACM, 1997.

George Giorgidze and Henrik Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In *Practical Aspects of Declarative Languages (PADL '08)*, pages 282–298. Springer, 2008.

George Giorgidze and Henrik Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Implementation and Application of Functional Languages (IFL '08)*, pages 138–155. Springer, 2011.

[5] Assuming there are no flaws in the Agda system itself.

Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. The Springer International Series in Engineering and Computer Science. Springer, 1993.

Nicolas Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Computer Aided Verification (CAV '98)*, pages 1–16. Springer, 1998.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

Thomas A. Henzinger. The theory of hybrid automata. In *Logics in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Society, 1996.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. Technical report, School of Informatics, University of Edinburgh, 2008.

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *International Conference on Functional Programming (ICFP '08)*, pages 335–345. ACM, 2008.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop (Haskell '02)*, pages 51–64. ACM, 2002.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Haskell Workshop (Haskell '04)*, pages 10–21. ACM, 2004.

Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming (ICFP '01)*, pages 229–240. ACM, 2001.

John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages (PADL '99)*, pages 91–105. Springer, 1999.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Marc Pouzet. *Lucid Synchrone, version 3: Tutorial and reference manual*. Université Paris-Sud, LRI, 2006. URL `www.di.ens.fr/ pouzet/lucid-synchrone`.

Neil Sculthorpe and Henrik Nilsson. Optimisation of dynamic, hybrid signal function networks. In *Trends in Functional Programming (TFP '08)*, pages 97–112. Intellect, 2009.

Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Programming Language Design and Implementation (PLDI '00)*, pages 242–252. ACM, 2000.

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, pages 146–156. ACM, 2001.