LTL types FRP

Linear-time Temporal Logic Propositions as Types
Proofs as Functional Reactive Programs

Alan Jeffrey

Alcatel-Lucent Bell Labs

ajeffrey@bell-labs.com

Abstract

Functional Reactive Programming (FRP) is a form of reactive programming whose model is pure functions over signals. FRP is often expressed in terms of arrows with loops, which is the type class for a Freyd category (that is a premonoidal category with a cartesian centre) equipped with a premonoidal trace. This type system suffices to define the dataflow structure of a reactive program, but does not express its temporal properties. In this paper, we show that Linear-time Temporal Logic (LTL) is a natural extension of the type system for FRP, which constrains the temporal behaviour of reactive programs. We show that a constructive LTL can be defined in a dependently typed functional language, and that reactive programs form proofs of constructive LTL properties. In particular, implication in LTL gives rise to stateless functions on streams, and the “constrains” modality gives rise to causal functions. We show that reactive programs form a partially traced monoidal category, and hence can be given as a form of arrows with loops, where the type system enforces that only decoupled functions can be looped.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Verification

Keywords Linear-time Temporal Logic, Functional Reactive Programming, Dependent Types

1. Introduction

Functional Reactive Programming (FRP) is a form of reactive programming whose model is pure functions over signals, which are time-dependent values. FRP was first introduced by Elliott in the context of functional animation [7, 9], and has since formed the basis of a number of reactive Domain Specific Embedded Languages, notably Fruit [5], Grapefruit [18] and Yampa [14],

Since the development of Yampa, FRP is often expressed in terms of arrows [15] with loops [26], which is the type class for a Freyd category [28] (that is a premonoidal category [29] with a cartesian centre) equipped with a premonoidal trace [3]. This type system suffices to define the dataflow structure of a reactive program, but does not express its temporal properties.

Sculthorpe and Nilsson [32] have shown that FRP can be implemented in a dependently typed language, but their goals were rather different. Although their programs are reactive, their types are not, in contrast to ours. Moreover, they are interested in run-time safety, not logical soundness, and so they disable the termination checker.

In this paper, we show that Linear-time Temporal Logic (LTL), introduced by Pnueli [27], is a natural extension of the type system for FRP, which constrains the temporal behaviour of reactive programs. In particular, the notions of *stateless function*, *causal function* and *decoupled function* which occur in FRP have natural expressions as LTL operators.

We show how LTL can be embedded in a dependently typed programming language such as Agda, and that LTL formulae can be used as *reactive types* for FRP programs, such that any well-typed program constitutes a proof of an LTL formula. Paraphrasing Curry–Howard, we consider *LTL propositions as types*, and *proofs as FRP programs*. This correspondence between LTL propositions and types for FRP was discovered simultaneously by Jeltsch [19].

We show that many FRP combinators can be given natural LTL types, and that the LTL types express the temporal behaviours of programs, for example allowing us to distinguish between a program which must be used immediately, and one which may be used at some point in the future. The most interesting typing is that of the *loop* combinator, which allows the construction of cyclic dataflow graphs. Its type is that of a partial trace [11], but it is derivable from the known result [23, 25] that LTL can be used to provide compositional proofs for cyclic rely/guarantee systems.

As well as a model for FRP, we provide a sketch of an implementation, based on *interval types*, and makes use of an embedded process language describing the I/O behaviour of causal functions.

2. Model

In this section, we discuss the signals model of FRP, and show that LTL can be used as a type system for FRP programs, such that well-typed FRP programs constitute proofs of LTL formulae.

We give definitions in pseudo-Agda, making use of dependent functions, dependent products, and inductive and coinductive datatypes. Most of Agda’s notation is hopefully familiar, but we call attention to Agda’s notation for inferrable arguments. The types:

$$\forall\{x\}A \qquad \exists\{x\}A$$

are universal (resp. existential) quantification over x, which may occur free in A. When instantiated, the witness for x may be provided explicitly:

$$a\{w\} \qquad (\{w\}, a)$$

PLPV'12, January 24, 2012, Philadelphia, PA, USA.

or implicitly, in which case it is to be inferred from context.

In many cases, we will follow the Agda naming convention for variables, and name them after their type, for example $s{\leq}t$ is a variable of type $s \leq t$. We will write $s{\leq}s$ and $s{\leq}t{\leq}u$ for uses of reflexivity and transitivity respectively, and similarly for other arithmetic properties such as $t{-}1{\leq}t$.

We will sometimes write pattern matching using an if-then-else syntax, which may bind variables. For example, a comparison test of s and t, making use of the result $s \leq t$ in the true branch and $s > t$ in the false branch is written:

$$\mathsf{if}\,(s \leq t)\,\mathsf{then}\,(\cdots s{\leq}t \cdots)\,\mathsf{else}\,(\cdots s{>}t \cdots)$$

In some cases, we will define functions, and provide a separate argument for termination, rather than weave the proof of termination in with the function definition, as required by Agda in cases when its termination-checking algorithm fails to prove totality.

2.1 Signals

The fundamental unit of data in a functional reactive program is a *signal* (also called a *behaviour*). This is a value which varies with time, for example the state of a mouse button might be tracked as:

$$\mathsf{mouseButton}(t) = \begin{cases} \mathsf{down} & \text{if } t \in [2,5) \\ \mathsf{up} & \text{otherwise} \end{cases}$$

which gives rise to an event stream of mouse clicks:

$$\mathsf{mouseClick}(t) = \begin{cases} \mathsf{just\ clicked} & \text{if } t = 5 \\ \mathsf{nothing} & \text{otherwise} \end{cases}$$

Signals are typed using:

$$\begin{aligned} \mathsf{Signal} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Signal}(A) &= \mathsf{Time} \to A \end{aligned}$$

for example:

$$\begin{aligned} \mathsf{mouseButton} &: \mathsf{Signal}(\mathsf{MouseButtonState}) \\ \mathsf{mouseClick} &: \mathsf{Signal}(\mathsf{Maybe}(\mathsf{MouseEvent})) \end{aligned}$$

We give the interface to the Time type in Figure 1, from which it follows that Time forms a decidable total order. In this paper, we assume a discrete time model, as this makes many definitions related to fixed points much simpler. Note that we do not assume that the sample interval for signals is the same as the discrete unit of time, for example an implementation might use POSIX time, where the unit is 1ms, which will usually be much less than the interval between signal state changes. We leave dense time FRP to future work, as discussed in Section 4.

2.2 Reactive types

Most work on FRP has been carried out in non-dependent languages such as Haskell. In a dependent language such as Agda, there is the possibility for a richer type system, where not only can the signal's value vary with time, but so can its type. We propose that a natural type language for reactive programs is one with *reactive types*, that is elements of $\mathsf{Time} \to \mathsf{Set}$, as defined in Figure 2.

For each type A there is a constant reactive type $\langle A \rangle$, but not all reactive types are constant. For example the reactive type Past contains times before the current time:

$$\begin{aligned} \mathsf{Past} &: \mathsf{RSet} \\ \mathsf{Past}(t) &= \exists\{s\}(s \leq t) \end{aligned}$$

A reactive type A gives rise to a type $[\![A]\!]$, whose elements are the signals σ such that $\sigma\{t\} : A(t)$ for all times t. This notation generalizes the Signal type constructor, since there is an isomorphism:

$$\mathsf{Signal}(A) \approx [\![\langle A \rangle]\!]$$

$$\begin{aligned} \mathsf{Time} &: \mathsf{Set} \\ (\cdot \leq \cdot) &: \mathsf{Time} \to \mathsf{Time} \to \mathsf{Set} \\ (\cdot + \cdot) &: \mathsf{Time} \to \mathbb{N} \to \mathsf{Time} \\ (\cdot - \cdot) &: \mathsf{Time} \to \mathsf{Time} \to \mathbb{N} \end{aligned}$$

$+$ is associative, cancellative, and has right unit 0

$t - u$ is the least n such that $t \leq u + n$

$t \leq u$ iff $\exists n \,.\, t + n = u$

Figure 1. Time model

$$\begin{aligned} \mathsf{RSet} &: \mathsf{Set}_1 \\ \mathsf{RSet} &= \mathsf{Time} \to \mathsf{Set} \\ \langle\cdot\rangle &: \mathsf{Set} \to \mathsf{RSet} \\ \langle A \rangle(t) &= A \\ [\![\cdot]\!] &: \mathsf{RSet} \to \mathsf{Set} \\ [\![A]\!] &= \forall\{t\}A(t) \end{aligned}$$

Figure 2. Reactive types

$$\begin{aligned} T, F &: \mathsf{RSet} \\ T(t) &= 1 \\ F(t) &= 0 \\ (\cdot \wedge \cdot), (\cdot \vee \cdot), (\cdot \Rightarrow \cdot) &: \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet} \\ (A \wedge B)(t) &= A(t) \times B(t) \\ (A \vee B)(t) &= A(t) + B(t) \\ (A \Rightarrow B)(t) &= A(t) \to B(t) \end{aligned}$$

Figure 3. Propositional logic as reactive types

for example:

$$\mathsf{mouseButton} : [\![\langle \mathsf{MouseButtonState} \rangle]\!]$$

Constant signals can be given constant type:

$$\begin{aligned} \eta &: \forall\{A\}A \to [\![\langle A \rangle]\!] \\ \eta(a)\{t\} &= a \end{aligned}$$

Signals of signals give rise to signals:

$$\begin{aligned} \mu &: \forall\{A\}[\![\langle [\![\langle A \rangle]\!] \rangle]\!] \to [\![\langle A \rangle]\!] \\ \mu(\sigma)\{t\} &= \sigma\{t\}\{t\} \end{aligned}$$

This gives rise to a monad $[\![\langle\cdot\rangle]\!] : \mathsf{Set} \to \mathsf{Set}$, as an instance of the read-only state monad, where the state is the current time.

Type combinators can be lifted pointwise from types to reactive types, for example we can define a lifted version of the Maybe type constructor as:

$$\begin{aligned} \mathsf{E} &: \mathsf{RSet} \to \mathsf{RSet} \\ \mathsf{E}(A)(t) &= \mathsf{Maybe}(A(t)) \end{aligned}$$

for example:

$$\mathsf{mouseClick} : [\![\mathsf{E}\langle \mathsf{MouseEvent} \rangle]\!]$$

In Figure 3 we show how logical combinators can be lifted from types to reactive types. This gives rise to a category $\mathbf{RSet}$, with:

- objects are reactive types,

$$(\cdot[\cdot,\cdot]),(\cdot[\cdot,\cdot)),(\cdot\langle\cdot,\cdot\rangle) : \mathsf{RSet} \to \mathsf{Time} \to \mathsf{Time} \to \mathsf{Set}$$
$$A[s,u] = \forall\{t\}(s \le t) \to (t \le u) \to A(t)$$
$$A[s,u) = \forall\{t\}(s \le t) \to (t < u) \to A(t)$$
$$A\langle s,u\rangle = \exists\{t\}(s \le t) \times (t \le u) \times A(t)$$

Figure 4. Types for reactive signals over an interval

$$\bigcirc, \Diamond, \Box : \mathsf{RSet} \to \mathsf{RSet}$$
$$(\bigcirc A)(t) = A(t+1)$$
$$(\Diamond A)(t) = \exists\{u\}(t \le u) \times A(u)$$
$$(\Box A)(t) = \forall\{u\}(t \le u) \to A(u)$$

$$(\cdot \mathsf{U} \cdot), (\cdot \underline{\mathsf{U}} \cdot), (\cdot \rhd \cdot), (\cdot \unrhd \cdot), (\cdot \rightsquigarrow \cdot) : \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet}$$
$$(A \mathsf{U} B)(t) = \exists\{u\}(t \le u) \times A[t,u) \times B(u)$$
$$(A \underline{\mathsf{U}} B)(t) = \exists\{u\}(t \le u) \times A[t,u] \times B(u)$$
$$(A \rhd B)(t) = \forall\{u\}(t \le u) \to A[t,u) \to B(u)$$
$$(A \unrhd B)(t) = \forall\{u\}(t \le u) \to A[t,u] \to B(u)$$
$$(A \rightsquigarrow B)(t) = \forall\{u\}(t \le u) \to A[t,u] \to B\langle t,u\rangle$$

Figure 5. Temporal modalities as reactive type combinators

- morphisms are elements of $[\![A \Rightarrow B]\!]$, and
- identity and composition inherited pointwise from $\mathbf{Set}$.

It is routine to check that the cartesian closed structure of $\mathbf{Set}$ lifts to $\mathbf{RSet}$, and that $\langle\cdot\rangle$ gives rise to a functor from $\mathbf{Set}$ to $\mathbf{RSet}$.

2.3 Linear-time Temporal Logic

Reactive types are types with one parameter of type Time, and so the slogan *propositions as types* becomes *propositions with one time parameter as reactive types*, and since this exactly describes propositions in a temporal logic, we have *temporal propositions as reactive types*. In particular, we can define the modalities of Linear-time Temporal Logic (LTL) [27] as combinators of reactive types. LTL has two primitive modalities:

- *next*: $\bigcirc A$ is true at time s whenever A is true at time $s+1$, and
- *until*: $A \mathsf{U} B$ is true at time s whenever there is some time $t \ge s$ such that A is true in the interval $[s,t)$ and B is true at time t.

LTL is usually presented as a classical logic, in which there are derived modalities, such as:

- *future*: $\Diamond A$ is true whenever A is true at some future time,
- *globally*: $\Box A$ is true whenever A is true at all future times,
- *non-strict until*: $A \underline{\mathsf{U}} B$ is true at time s whenever there is some time $t \ge s$ such that A is true in the interval $[s,t]$ and B is true at time t,
- *constrains*: $A \rhd B$ is true at time s whenever, for all times $t \ge s$, if A is true in the interval $[s,t)$ then B is true at time t, and
- *non-strict constrains*: $A \unrhd B$ is true at time t whenever, for all times $t \ge s$, if A is true in the interval $[s,t]$ then B is true at time t.

The "constrains" modality was introduced by McMillan [23] and studied further by Namjoshi and Trefler [25], as the basis of compositional reasoning about rely/guarantee properties of parallel systems. We also make use of a modality which classically collapses to $A \Rightarrow B$, but constructively defines a choice function:

$$\mathsf{extend} : \forall\{A,B\}[\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!]$$
$$\mathsf{extend}(f)(a)(s{\le}t) = f(a(s{\le}t))$$
$$\mathsf{extract} : \forall\{A\}[\![\Box A \Rightarrow A]\!]$$
$$\mathsf{extract}(a) = a(s{\le}s)$$
$$\mathsf{duplicate} : \forall\{A\}[\![\Box A \Rightarrow \Box\Box A]\!]$$
$$\mathsf{duplicate}(a)(s{\le}t)(t{\le}u) = a(s{\le}t{\le}u)$$

Figure 6. Comonad structure of $\Box$

$$[\cdot] : \forall\{A\}[\![A]\!] \to [\![\Box A]\!]$$
$$[a](s{\le}t) = a$$
$$(\cdot \langle * \rangle \cdot) : \forall\{A,B\}[\![\Box(A \Rightarrow B) \Rightarrow \Box A \Rightarrow \Box B]\!]$$
$$(f \langle * \rangle \sigma)(s{\le}t) = f(s{\le}t)(\sigma(s{\le}t))$$

Figure 7. Applicative structure of $\Box$

- *choice*: $A \rightsquigarrow B$ is true at time s whenever, for all times $u \ge s$, if A is true in the interval $[s,u]$ then B is true at some time $t \in [s,u]$.

The derivations of these modalities in a classical logic are:

$$\Diamond A = T \mathsf{U} A$$
$$\Box A = \neg(\Diamond \neg A)$$
$$A \underline{\mathsf{U}} B = A \mathsf{U} (A \wedge B)$$
$$A \rhd B = \neg(A \mathsf{U} \neg B)$$
$$A \unrhd B = A \neg(A \underline{\mathsf{U}} \neg B)$$
$$A \rightsquigarrow B = \neg((A \wedge \neg B) \underline{\mathsf{U}} T)$$

In a constructive logic, such as a dependent type theory, the derivations using De Morgan duality are not valid, and so we give the definitions of the modalities directly in Figures 4 and 5. In particular, note that $A \Rightarrow B$, $A \unrhd B$ and $A \rhd B$ are all function spaces:

- elements of $A \Rightarrow B$ are *stateless functions*, whose output value at time t only depends on the input value at time t,
- elements of $A \unrhd B$ are *causal functions*, whose output value at time t depends on a history of inputs, and
- elements of $A \rhd B$ are *decoupled functions*, whose output value at time t depends on a history of inputs, but cannot depend on the input value at time t.

Since Figures 3–5 are a direct translation of the semantics of LTL into dependent type theory, we have a direct soundness result (assuming that dependent type theory is sound for classical logic):

for any LTL formula F with uninterpreted atoms $\vec{A}$

if $\vdash p : \forall\{\vec{A}\}[\![F]\!]$ then F is a tautology

Most of our work will be done in LTL with future, but some material will require a modality from LTL with past:

- *yesterday*: $\ominus A$ is true at time s whenever A is true at time $s-1$,
- *non-strict since*: $A \underline{\mathsf{S}} B$ is true at time t whenever there is some time $s \le t$ such that A is true in the interval $[s,t]$ and B is true at time s,
- *once*: $\diamondminus A$ is true whenever A is true at some past time, and
- *historically*: $\boxminus A$ is true whenever A is true at all past times.

$$\mathsf{arr} : \forall\{A,B\}[\![\Box(A \Rightarrow B) \Rightarrow (A \trianglerighteq B)]\!]$$
$$\mathsf{arr}(f)(s{\le}t)(\sigma) = f(s{\le}t)(\sigma(s{\le}t)(t{\le}t))$$
$$\mathsf{identity} : \forall\{A\}[\![A \trianglerighteq A]\!]$$
$$\mathsf{identity} = \mathsf{arr}[\lambda a\ .\ a]$$
$$(\cdot\ \mathsf{before}\ \cdot) : \forall\{A,s,u,v\}A[s,v] \to (u \le v) \to A[s,u]$$
$$(\sigma\ \mathsf{before}\ u{\le}v)(s{\le}t)(t{\le}u) = \sigma(s{\le}t)(t{\le}u{\le}v)$$
$$(\cdot\ \mathsf{after}\ \cdot) : \forall\{A,s,t,v\}A[s,v] \to (s \le t) \to A[t,v]$$
$$(\sigma\ \mathsf{after}\ s{\le}t)(t{\le}u)(u{\le}v) = \sigma(s{\le}t{\le}u)(u{\le}v)$$
$$(\cdot\ \$\ \cdot) : \forall\{A,B,s,u\}(A \trianglerighteq B) \to A[s,u] \to B[s,u]$$
$$(f\ \$\ \sigma)(s{\le}t)(t{\le}u) = f(s{\le}t)(\sigma\ \mathsf{before}\ t{\le}u)$$
$$(\cdot \ggg \cdot) : \forall\{A,B,C\}[\![(A \trianglerighteq B) \Rightarrow (B \trianglerighteq C) \Rightarrow (A \trianglerighteq C)]\!]$$
$$(f \ggg g)(s{\le}t)(\sigma) = g(s{\le}t)(f\ \$\ \sigma)$$

Figure 8. Enriched categorical structure of $\trianglerighteq$

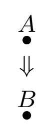
These are just the duals of $\bigcirc A$, $A\ \underline{\mathsf{U}}\ B$, $\Diamond A$ and $\Box A$, with $\ge$ replacing $\le$. Figure 6 gives the constructions forming a comonad for $\Box$, and Figure 7 gives the constructions for an applicative functor. Together, these show that $\Box$ is a model of S4 modal logic. Similar constructions give that $\bigcirc$ and $\ominus$ are applicative functors, $\Diamond$ and $\diamondminus$ are applicative monads, and $\boxminus$ is an applicative comonad. Since $\boxminus$ is a comonad, we can build the Kleisli category $\boxminus\mathbf{RSet}$:

- objects are reactive types,
- morphisms are elements of $[\![\boxminus A \Rightarrow B]\!]$, and
- identity and composition as usual for a Kleisli construction.

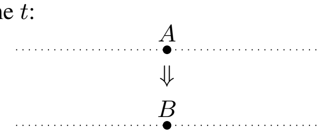
Figure 8 shows the constructions for an $\mathbf{RSet}$-enriched category with homobjects given by $\trianglerighteq$. Let $\trianglerighteq\mathbf{RSet}$ be the category with:

- objects are reactive types,
- morphisms are elements of $[\![A \trianglerighteq B]\!]$, and
- identity and composition given in Figure 8.

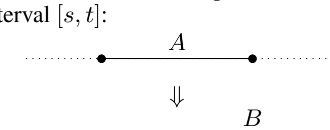
These can be visualized as morphisms in $\mathbf{Set}$ witnessing an implication with no place in time:



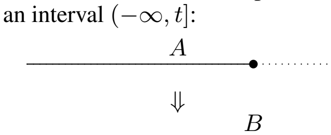
morphisms in $\mathbf{RSet}$ witness an implication whose hypothesis is true at some time t:



morphisms in $\trianglerighteq\mathbf{RSet}$ witness an implication whose hypothesis is true over an interval $[s,t]$:



and morphisms in $\boxminus\mathbf{RSet}$ witness an implication whose hypothesis is true over an interval $(-\infty, t]$:



$$\mathsf{fst} : \forall\{A,B\}[\![(A \wedge B) \trianglerighteq A]\!]$$
$$\mathsf{fst} = \mathsf{arr}[\lambda(a,b)\ .\ a]$$
$$\mathsf{snd} : \forall\{A,B\}[\![(A \wedge B) \trianglerighteq B]\!]$$
$$\mathsf{snd} = \mathsf{arr}[\lambda(a,b)\ .\ b]$$
$$(\cdot\ \&\&\&\ \cdot) : \forall\{A,B,C\}[\![(A \trianglerighteq B) \Rightarrow (A \trianglerighteq C) \Rightarrow (A \trianglerighteq B \wedge C)]\!]$$
$$(f\ \&\&\&\ g)(s{\le}t)(\sigma) = (f(s{\le}t)(\sigma), g(s{\le}t)(\sigma))$$

Figure 9. Enriched product structure of $\wedge$

In each case, the hypothesis contains progressively more information, which leads to functors:

$$\mathbf{Set} \hookrightarrow \mathbf{RSet} \hookrightarrow \trianglerighteq\mathbf{RSet} \hookrightarrow \boxminus\mathbf{RSet}$$

Note that the embedding $\trianglerighteq\mathbf{RSet} \hookrightarrow \boxminus\mathbf{RSet}$ requires the existence of a least element of time $-\infty$, from which we define:

$$\mathsf{bounded} : \forall\{A,B\}[\![A \trianglerighteq B]\!] \to [\![\boxminus A \Rightarrow B]\!]$$
$$\mathsf{bounded}(f)(\sigma) = f(-\infty{\le}t)(\lambda{-\infty}{\le}s\ .\ \sigma)$$

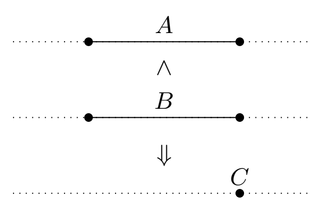
These categories all have finite products, inherited from $\mathbf{Set}$, for example the product structure of $\trianglerighteq\mathbf{RSet}$ is given in Figure 9.

$\mathbf{RSet}$ inherits its coproducts from $\mathbf{Set}$, but $\trianglerighteq\mathbf{RSet}$ only has weak coproducts, and $\boxminus\mathbf{RSet}$ does not have coproducts. In $\trianglerighteq\mathbf{RSet}$ we can construct a mediating morphism:

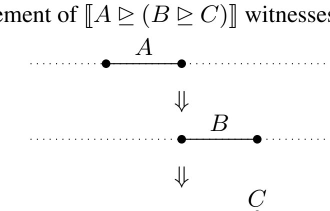
$$\mathsf{cond} : \forall\{A,B,C\} \to [\![A \trianglerighteq C]\!] \to [\![B \trianglerighteq C]\!] \to [\![(A \vee B) \trianglerighteq C]\!]$$

If $\sigma(u) = \mathsf{inl}(a)$ then $\mathsf{cond}(f)(g)(s{\le}u)(\sigma) = f(t{\le}u)(\tau)$ where $\tau \in A[t,u]$ is the longest segment such that $\mathsf{inl}(\tau)$ is a suffix of σ, and symmetrically if $\sigma(u) = \mathsf{inr}(b)$. This satisfies the commuting diagram of a coproduct, but is not unique.

All of the categories but $\trianglerighteq\mathbf{RSet}$ are cartesian closed: to see why $\trianglerighteq\mathbf{RSet}$ is not cartesian closed, consider an element of $[\![(A \wedge B) \trianglerighteq C]\!]$, which can be visualized as witnessing:



whereas an element of $[\![A \trianglerighteq (B \trianglerighteq C)]\!]$ witnesses:



Namjoshi (personal communication) has shown that in $\mathbf{RSet}$, $\underline{\mathsf{S}}$ (with its arguments flipped) is the left adjoint of $\trianglerighteq$, and so $\mathbf{RSet}$ does have a closed structure for $\trianglerighteq$, even if $\trianglerighteq\mathbf{RSet}$ does not.

Harel, Kozen and Parikh [12] introduced a "chop" modality on paths, studied by Rosner and Pnueli [31], where $A_1 \mathsf{C} A_2$ is true on a signal σ when σ can be partitioned into σ_1, σ_2 such that σ_1 satisfies A_1 and σ_2 satisfies A_2. We expect that this is the left adjoint to $\trianglerighteq$, and would provide a monoidal closed structure. However, this would take us out of LTL and into a logic on paths, so we leave this issue for future work.

2.4 Reactive functions

The fundamental unit of computation in a functional reactive program is a *signal function*. This is a function from signals to signals, for example a function which monitors a mouse state, and returns mouse-clicked events is:

$$\begin{aligned}&\mathsf{clickMonitor} : [\![\langle\mathsf{MouseButtonState}\rangle]\!] \to [\![\mathsf{E}\langle\mathsf{MouseEvent}\rangle]\!]\\ &\mathsf{clickMonitor}\{t\}(\sigma) =\\ &\quad \mathsf{if}\ (\sigma(t) = \mathsf{up}\ \&\ \sigma(t-1) = \mathsf{down})\\ &\quad \mathsf{then}\ (\mathsf{just\ clicked})\ \mathsf{else}\ (\mathsf{nothing})\end{aligned}$$

The type of all functions on signals is too generous, however, as it contains functions whose output value in the present can depend on input values in the future. We require functions to be *causal*, which (in the non-dependent case) leads to the type of signal functions:

$$\mathsf{SF} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{SF}\,A\,B = (f : \mathsf{Signal}(A) \to \mathsf{Signal}(B)) \times (\mathsf{Causal}(f))$$

where the causal functions are those which respect equivalence up to the current time:

$$\mathsf{Causal}(f) = \forall\{\sigma, \sigma', u\}(\sigma \approx_u \sigma') \to (f(\sigma) \approx_u f(\sigma'))$$
$$\sigma \approx_u \sigma' = \forall\{t\}(t \le u) \to (\sigma(t) = \sigma'(t))$$

The type $\mathsf{SF}\,A\,B$ can be encoded in LTL over the past, as:

$$\mathsf{SF}\,A\,B \approx [\![\boxminus\langle A\rangle \Rightarrow \langle B\rangle]\!]$$

that is, signal functions are given a semantics in $\boxminus\mathbf{RSet}$. Since $\boxminus\mathbf{RSet}$ is a category with finite products, it satisfies the requirements of Hughes's *arrows* [15], since arrows are the type class for Freyd categories [28], and any category with finite products is trivially a Freyd category.

Jeltsch [18] has proposed a modification to the SF type, to add a *era* parameter, giving the start time of the input and output signals. In Haskell, the era parameter has to be modelled as a phantom type, but in a dependent language, we can make it a time parameter:

$$\mathsf{SF}' : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Time} \to \mathsf{Set}$$
$$\mathsf{SF}'\,A\,B\,s = (f : \mathsf{Signal}(A) \to \mathsf{Signal}(B)) \times (\mathsf{Causal}_s(f))$$

This requires a modification to the definition of causality, to include a start time as well as the current time:

$$\mathsf{Causal}_s(f) = \forall\{\sigma, \sigma', u\}(\sigma \approx_{[s,u]} \sigma') \to (f(\sigma) \approx_{[s,u]} f(\sigma'))$$
$$\sigma \approx_{[s,u]} \sigma' = \forall\{t\}(s \le t) \to (t \le u) \to (\sigma(t) = \sigma'(t))$$

The type $\mathsf{SF}'\,A\,B\,s$ can be encoded in LTL, as:

$$\mathsf{SF}'\,A\,B \approx \langle A\rangle \trianglerighteq \langle B\rangle$$

that is, signal functions are given a semantics in $\trianglerighteq\mathbf{RSet}$. Again, since $\trianglerighteq\mathbf{RSet}$ has finite products, it satisfies the requirements of arrows. An important subset of signal functions is the *decoupled* functions, whose output can depend on input in the strict past:

$$\mathsf{SF}'' : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Time} \to \mathsf{Set}$$
$$\mathsf{SF}''\,A\,B\,s = (f : \mathsf{Signal}(A) \to \mathsf{Signal}(B)) \times (\mathsf{Decoupled}_s(f))$$
$$\mathsf{Decoupled}_s(f) = \forall\{\sigma, \sigma', u\}(\sigma \approx_{[s,u)} \sigma') \to (f(\sigma) \approx_{[s,u]} f(\sigma'))$$
$$\sigma \approx_{[s,u)} \sigma' = \forall\{t\}(s \le t) \to (t < u) \to (\sigma(t) = \sigma'(t))$$

The type $\mathsf{SF}''\,A\,B\,s$ can be encoded in LTL, as:

$$\mathsf{SF}''\,A\,B \approx \langle A\rangle \rhd \langle B\rangle$$

Note that $\rhd\mathbf{RSet}$ does *not* form a category: it has a composition operation, but not identities, as the identity function is only decoupled on singleton types. This is similar to the situation of contraction maps in a complete metric space: identities are non-expanding, but not contracting.

In the remainder of the paper, we focus on $\trianglerighteq\mathbf{RSet}$. We expect that the results would carry over to $\boxminus\mathbf{RSet}$.

$$\begin{aligned}&\mathsf{constant} : \forall\{A, B\}[\![\Box B \Rightarrow A \rhd B]\!]\\ &\mathsf{constant}(\tau)(s{\le}t)(\sigma) = \tau(s{\le}t)\end{aligned}$$
$$\begin{aligned}&\mathsf{localTime} : \forall\{A\}[\![A \rhd \langle\mathsf{Time}\rangle]\!]\\ &\mathsf{localTime}(s{\le}t)(\sigma) = t\end{aligned}$$
$$\begin{aligned}&\mathsf{initially} : \forall\{A\}[\![A \Rightarrow A \trianglerighteq A]\!]\\ &\mathsf{initially}\{s\}(a)\{t\}(s{\le}t)(\sigma) =\\ &\quad \mathsf{if}\ (s = t)\ \mathsf{then}\ (a)\ \mathsf{else}\ (\sigma(s{\le}t)(t{\le}t))\end{aligned}$$
$$\begin{aligned}&\mathsf{decouple} : \forall\{A\}[\![\ominus A \Rightarrow A \rhd \ominus A]\!]\\ &\mathsf{decouple}(a)(s{\le}t)(\sigma) =\\ &\quad \mathsf{if}\ (s \le t-1)\ \mathsf{then}\ (\sigma(s{\le}t{-}1)(t{-}1{\le}t))\ \mathsf{else}\ (a)\end{aligned}$$

Figure 10. Example FRP primitives, with LTL types

2.5 FRP Combinators

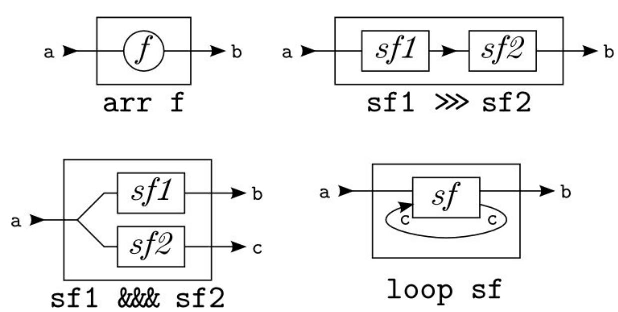
We have already seen some of the combinators of FRP, in the constructions which showed that $\trianglerighteq\mathbf{RSet}$ forms a category with finite products. These allow the construction of dataflow programs, for example (using primitives discussed below) we can construct $\mathsf{clickMonitor}$ as:

$$\begin{aligned}&\mathsf{clickMonitor} : [\![\langle\mathsf{MouseButtonState}\rangle \trianglerighteq \langle\mathsf{MouseEvent}\rangle]\!]\\ &\mathsf{clickMonitor} = \mathsf{arr}[\cdot = \mathsf{up}] \ggg \mathsf{edge} \ggg \mathsf{tag}[\mathsf{mouseClick}]\end{aligned}$$

Note that the combinators $\ggg$ and &&& respect decoupling:

- if $f : [\![A \rhd B]\!]$ and $g : [\![B \trianglerighteq C]\!]$ then $f \ggg g : [\![A \rhd C]\!]$,
- if $f : [\![A \trianglerighteq B]\!]$ and $g : [\![B \rhd C]\!]$ then $f \ggg g : [\![A \rhd C]\!]$, and
- if $f : [\![A \rhd B]\!]$ and $g : [\![A \rhd C]\!]$ then $f \mathbin{\&\&\&} g : [\![A \rhd B \wedge C]\!]$.

This typing can be made precise by indexing the $\trianglerighteq$ type by a coupledness flag, as was shown by Sculthorpe and Nilsson [32]. Together with the loop combinator discussed below, $\ggg$ and &&& allow dataflow networks to be built, and can be visualized as:



This visualization can be carried out for any traced monoidal category, as surveyed, for example, by Selinger [33].

2.6 FRP Primitives

FRP libraries feature a number of primitives, from which reactive programs can be built compositionally. In Figure 10 we show how some prototypic primitives can be given a semantics in terms of signal functions, and their types in LTL.

The LTL types are more expressive than the usual types, for example in Yampa [34] the type for `constant` is (in our notation) $B \to [\![\langle A\rangle \trianglerighteq \langle B\rangle]\!]$. In comparison, the LTL type $[\![\Box B \Rightarrow A \rhd B]\!]$ makes it clear that the B argument is required at all times in the future, not just now (hence the $\Box B$ modality), and the function is decoupled (hence the $\rhd$ type).

$\mathsf{never} : \forall\{A, B\}[\![A \rhd \mathsf{E}B]\!]$
$\mathsf{never}(s{\le}t)(\sigma) = \mathsf{nothing}$

$\mathsf{now} : \forall\{A, B\}[\![B \Rightarrow A \rhd \mathsf{E}B]\!]$
$\mathsf{now}\{s\}(b)\{t\}(s{\le}t)(\sigma) =$
$\quad \mathsf{if}\ (s = t)\ \mathsf{then}\ (\mathsf{just}(b))\ \mathsf{else}\ (\mathsf{nothing})$

$\mathsf{later} : \forall\{A, B\}[\![\Diamond B \Rightarrow A \rhd \mathsf{E}B]\!]$
$\mathsf{later}(\{u\}, s{\le}u, b)\{t\}(s{\le}t)(\sigma) =$
$\quad \mathsf{if}\ (t = u)\ \mathsf{then}\ (\mathsf{just}(b))\ \mathsf{else}\ (\mathsf{nothing})$

$\mathsf{tag} : \forall\{A, B\}[\![\Box B \Rightarrow \mathsf{E}A \trianglerighteq \mathsf{E}B]\!]$
$\mathsf{tag}(b)(s{\le}t)(\sigma) =$
$\quad \mathsf{if}\ (\sigma(s{\le}t)(t{\le}t) = \mathsf{just}(a))$
$\quad \mathsf{then}\ (\mathsf{just}(b(s{\le}t)))\ \mathsf{else}\ (\mathsf{nothing})$

$\mathsf{edge} : [\![[\mathsf{Bool}] \trianglerighteq \mathsf{E}T]\!]$
$\mathsf{edge}(s{\le}t)(\sigma) =$
$\quad \mathsf{if}\ (s < t\ \&\ !\sigma(s{<}t)(t{-}1{\le}t)\ \&\ \sigma(s{\le}t)(t{\le}t))$
$\quad \mathsf{then}\ (\mathsf{just}())\ \mathsf{else}\ (\mathsf{nothing})$

$\mathsf{hold} : \forall\{A\}[\![A \Rightarrow \mathsf{E}A \rhd \Diamond\!\!\!\!\,\cdot A]\!]$
$\mathsf{hold}(a)(s{\le}u)(\sigma) =$
$\quad \mathsf{if}\ (\mathsf{last}(s{\le}u)(\sigma) = (s{\le}t, t{\le}u, \mathsf{just}(b)))$
$\quad \mathsf{then}\ (t{\le}u, \mathsf{just}(b)y)\ \mathsf{else}\ (s{\le}u, a)$

Figure 11. Example FRP event primitives, with LTL types

$\mathsf{first} : \forall\{A\}[\![\mathsf{E}A \rightsquigarrow \mathsf{E}A]\!]$
$\mathsf{first}\{s\}\{t\}(s{\le}t)(\sigma) = f(m)(s{\le}t{-}m)\ \mathsf{where}$
$\quad m = t - s$
$\quad f(0)(s{\le}t{-}0) = (s{\le}t, t{\le}t, \sigma(s{\le}t)(t{\le}t))$
$\quad f(n+1)(s{<}t{-}n) =$
$\quad\quad \mathsf{if}\ (\sigma(s{<}t{-}n)(t{-}(n{+}1){\le}t) = \mathsf{just}(a))$
$\quad\quad \mathsf{then}\ (s{<}t{-}n, t{-}(n{+}1){\le}t, \mathsf{just}(a))$
$\quad\quad \mathsf{else}\ (f(n)(s{\le}t{-}n))$

$\mathsf{last} : \forall\{A\}[\![\mathsf{E}A \rightsquigarrow \mathsf{E}A]\!]$
$\mathsf{last}(s)(t)(s{\le}t)(\sigma)$ is defined similarly

Figure 12. Example FRP choice primitives, with LTL types

$\mathsf{switch} : \forall\{A, B, C\}$
$\quad [\![(A \trianglerighteq (B \wedge \mathsf{E}C)) \Rightarrow \Box(C \Rightarrow (A \trianglerighteq B)) \Rightarrow (A \trianglerighteq B)]\!]$
$\mathsf{switch}(f)(g)(s{\le}u)(\sigma) =$
$\quad \mathsf{if}\ (\mathsf{first}(s{\le}u)(\mathsf{snd}\ \$\ (f\ \$\ \sigma)) = (s{\le}t, t{\le}u, \mathsf{just}(c)))$
$\quad \mathsf{then}\ (g(s{\le}t)(c)(t{\le}u)(\sigma\ \mathsf{after}\ s{\le}t))$
$\quad \mathsf{else}\ (\mathsf{fst}(s{\le}u)(f\ \$\ \sigma))$

$\mathsf{rswitch} : \forall\{A, B\}[\![(A \trianglerighteq B) \Rightarrow (A \wedge \mathsf{E}(A \trianglerighteq B)) \trianglerighteq B]\!]$
$\mathsf{rswitch}(f)(s{\le}u)(\sigma) =$
$\quad \mathsf{if}\ (\mathsf{last}(s{\le}u)(\mathsf{snd}\ \$\ \sigma) = (s{\le}t, t{\le}u, \mathsf{just}(g)))$
$\quad \mathsf{then}\ (g(t{\le}u)(\mathsf{fst}\ \$\ (\sigma\ \mathsf{after}\ s{\le}t)))$
$\quad \mathsf{else}\ (f(s{\le}u)(\mathsf{fst}\ \$\ \sigma))$

Figure 13. FRP switching combinators

LTL types allow for nested signals via the $\Box$ modality, which indicates a stream of future values, for example $[\![\Box\Box A]\!]$ is inhabited by signals of signals of type A.

The decouple function is used to introduce minimal decoupling: it acts as an identity, but with a 1 unit delay. This is reflected in its type: it inputs A and returns $\ominus A$. In the most common case, A is a constant type, and since $\langle A\rangle = \ominus\langle A\rangle$, decouple specializes to have type $\langle A\rangle \rhd \langle A\rangle$.

2.7 Event primitives

As well as continuous behaviours, FRP supports an event model, whose semantics is given by signals of type $\mathsf{E}A$, that is at time t they are either nothing (no event has arrived) or $\mathsf{just}(a)$ (an event a of type $A(t)$ has arrived). In Figure 11 we show how some event primitives can be given a semantics in terms of signal functions, and their types in LTL.

Again, the LTL types are descriptive of the temporal behaviour of the primitives, for example now and later (which have the same type in Yampa) now have different types: now returns its argument immediately, so the argument has type B, whereas later returns its argument at some point in the future, so the argument has type $\Diamond B$.

The edge function is a primitive for converting signals into events, and the hold function converts events into signals. The implementation of hold makes use of the $\rightsquigarrow$ modality, since it uses last, which chooses the last event from an event stream. It is defined in Figure 12. Note that this uses induction over delays, and is one of the places where we assume a discrete model of time. We discuss this further in Section 4.

2.8 Switching

Many FRP implementations include a notion of *switching* between signal functions, which supports starting and stopping signal functions based on events. For example a function which returns true after an event has occurred is:

$$\mathsf{switch}(\mathsf{constant}[\mathsf{false}]\ \&\&\&\ \mathsf{identity})[\lambda x\ .\ \mathsf{constant}[\mathsf{true}]]$$

Sample switching combinators are given in Figure 13, based on the corresponding Yampa combinators. There are two switches, depending on whether the switch should only react to the first switching event (switch) or every switching event (rswitch). Their semantics is defined in terms of first and last.

Again, note that the LTL type for switch makes it clear that the event being switched to is run in the future, since it is given by a function of type $\Box(C \Rightarrow (A \trianglerighteq B))$.

2.9 Loops

Much of the power of FRP comes from the ability to form feedback loops, using a function of type (in our notation):

$$[\![((A \wedge B) \trianglerighteq (A \wedge C)) \Rightarrow B \trianglerighteq C]\!]$$

which is required to satisfy the equations of a *traced premonoidal category* [3]. In Haskell, this is an instance of the type class of arrows with loops [26]. A consequence of the existence of loops is that every type is inhabited, for example we can construct:

$$f : [\![(F \wedge T) \trianglerighteq (F \wedge F)]\!]$$
$$f = \mathsf{arr}[\lambda(x, y)\ .\ (x, x)]$$

Tracing f gives a function $T \trianglerighteq F$, which can be used to inhabit the empty type. This construction is not problematic in Haskell, where there is a canonical $\bot$ element inhabiting all types, but it is problematic in total languages such as Agda. We could try to fix this by adding a seed value:

$$[\![((A \wedge B) \trianglerighteq (A \wedge C)) \Rightarrow A \Rightarrow B \trianglerighteq C]\!]$$

$$\begin{array}{l}
\mathsf{fix} : \forall\{A\}[\![(A \rhd A) \Rightarrow \Box A]\!] \\
\mathsf{fix}\{s\}(f)(s{\le}u) = f(s{\le}u)(\sigma) \text{ where} \\
\quad \sigma : \forall\{u\}A[s,u] \\
\quad \sigma(s{\le}t)(t{<}u) = f(s{\le}t)(\sigma) \\
\mathsf{ifix} : \forall\{A,B\}[\![(A \wedge B \rhd A) \Rightarrow B \rhd A]\!] \\
\mathsf{ifix}\{s\}(f)\{v\}(s{\le}v)(\tau) = \mathsf{fix}(g)(s{\le}v)(v{\le}v) \text{ where} \\
\quad A' : \mathsf{RSet} \\
\quad A'(t) = (t \le v) \to A(t) \\
\quad g : (A' \rhd A')s \\
\quad g\{u\}(s{\le}u)(\sigma)(u{\le}v) = f(s{\le}u)(\rho) \text{ where} \\
\quad\quad \rho : (A \wedge B)[s,u] \\
\quad\quad \rho(s{\le}t)(t{<}u) = (\sigma(s{\le}t)(t{<}u)(t{\le}u{\le}v), \tau(s{\le}t)(t{<}u{\le}v)) \\
\mathsf{loop} : \forall\{A,B,C\}[\![((A \wedge B) \rhd (A \wedge C)) \Rightarrow B \rhd C]\!] \\
\mathsf{loop}(f) = (\mathsf{ifix}(f \ggg \mathsf{fst}) \;\&\&\&\; \mathsf{identity}) \ggg f \ggg \mathsf{snd}
\end{array}$$

Figure 14. FRP loop combinators

However, since A might vary over time, this construction is still unsound, for example we could set:

$$A(t) = \mathsf{if}\ (t \le 0)\ \mathsf{then}\ (1)\ \mathsf{else}\ (0)$$

and replay the previous example. The solution to this is to only give fixed points to *decoupled* functions, that is the type of loop is:

$$\mathsf{loop} : \forall\{A,B,C\}[\![((A \wedge B) \rhd (A \wedge C)) \Rightarrow B \rhd C]\!]$$

This type is inhabited because decoupled functions have fixed points:

$$\mathsf{fix} : \forall\{A\}[\![(A \rhd A) \Rightarrow \Box A]\!]$$

from which we can construct indexed fixed points:

$$\mathsf{ifix} : \forall\{A,B\}[\![(A \wedge B \rhd A) \Rightarrow B \rhd A]\!]$$

which in turn is enough to define loop. The fact that $\rhd$ has fixed points is a known result for LTL, and is the basis of rely/guarantee reasoning for parallel composition of systems [23, 25].

The details are given in Figure 14. Note that the termination of fix relies on $<$ forming a well-ordering over a closed interval. In a discrete time model, this is immediate. If we were to replay this for a dense time model, we would introduce ϵ-decoupled functions, for some $\epsilon > 0$. As Krishnaswami and Benton [21] showed, decoupled functions form contraction maps in an ultrametric space of functions, and so have unique fixed points. The use of ultrametric spaces to model fixed points in timed reactive systems goes back to Reed and Roscoe's work on Timed CSP [30].

Note that since loop can only be applied to decoupled functions, and not functions in general, it does *not* form a trace. Instead, it forms a *partial* trace, in the sense of Haghverdi and Scott [11]. This is unsurprising, as complete metric spaces are one of the motivating examples for partial traces.

Note, however, that Haghverdi and Scott's definition of the trace class for a complete metric space has that a function $f : A \times B \to A \times C$ is traceable whenever, for all $b \in B$, the function $\lambda a\, .\, \pi(f(a,b)) : A \to A$ has a unique fixed point, *not* whenever f is a contraction map. In our setting, the type system is giving a static approximation to the trace class, since if $f : [\![A \rhd B]\!]$ then f is a contraction map, and so is in the trace class, but not conversely.

Sculthorpe and Nillson [32] have developed a more refined type system for tracking function decoupling. For a function of type $[\![A_1 \wedge \cdots \wedge A_m \trianglerighteq B_1 \wedge \cdots \wedge B_n]\!]$, the type system carries an $m \times n$ matrix, such that (i,j) is marked as decoupled whenever the B_j output can only depend on the A_i input in the strict past.

$$\begin{array}{l}
\mathsf{data}\ \mathsf{Time}^\infty : \mathsf{Set}\ \mathsf{where} \\
\quad \infty : \mathsf{Time}^\infty \\
\quad \mathsf{fin} : \mathsf{Time} \to \mathsf{Time}^\infty \\
\mathsf{data}\ (\cdot \preceq \cdot) : \mathsf{Time}^\infty \to \mathsf{Time}^\infty \to \mathsf{Set}\ \mathsf{where} \\
\quad \infty : (\forall\{t\} t \preceq \infty) \\
\quad \mathsf{fin} : \forall\{s,t\}(s \le t) \to (\mathsf{fin}(s) \preceq \mathsf{fin}(t)) \\
(\cdot \prec \cdot) : \mathsf{Time}^\infty \to \mathsf{Time}^\infty \to \mathsf{Set} \\
(s \prec t) = (s \preceq t) \times ((t \preceq s) \to 0)
\end{array}$$

Figure 15. Time bounds

$$\begin{array}{l}
\mathsf{data}\ \mathsf{Interval} : \mathsf{Set}\ \mathsf{where} \\
\quad [\cdot) : \forall\{s,t\}(s \prec t) \to \mathsf{Interval} \\
\mathsf{Int}^\infty : \mathsf{Interval} \to \mathsf{Time}^\infty \to \mathsf{Set} \\
\mathsf{Int}^\infty[s{\prec}u)(t) = (s \preceq t) \times (t \prec u) \\
\mathsf{Int} : \mathsf{Interval} \to \mathsf{Time} \to \mathsf{Set} \\
\mathsf{Int}[s{\prec}u)(t) = \mathsf{Int}^\infty[s,u)(\mathsf{fin}(t)) \\
(\cdot \sqsubseteq \cdot) : \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{Set} \\
([t{\prec}u) \sqsubseteq [s{\prec}v)) = (s \preceq t) \times (u \preceq v) \\
(\cdot \sim \cdot) : \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{Set} \\
[s{\prec}t) \sim [u{\prec}v) = (t = u) \\
(\cdot \frown \cdot \because \cdot) : (i,j : \mathsf{Interval}) \to (i \sim j) \to \mathsf{Interval} \\
[s{\prec}t) \frown [t{\prec}u) \because t{=}t = [s{\prec}t{\prec}u)
\end{array}$$

Figure 16. Time intervals

Our type system approximates these matrices: $A \rhd B$ corresponds to a matrix which is everywhere decoupled, and $A \trianglerighteq B$ approximates any matrix. We speculate that their type system satisfies the requirements of a partial trace, but leave this for future work.

3. Implementation

In the previous section we gave a model for FRP in a dependently typed language. Unfortunately, while the model is executable, it is not efficiently executable for two reasons:

- it is a *polling pull* implementation, where the receiver of data is required to sample a signal, and
- it suffers from *time leaks* in that the entire input history must be recorded, and cannot be garbage collected.

For these reasons, we investigate an alternative implementation strategy. The implementation replaces the pull strategy by a push strategy, in which the producer of data can push a *segment* of a signal to a reactive program: in return it will receive back segments of output generated by the program, together with a continuation, in the style of Carlsson and Hallgren's Fudgets [4]. It is similar to Elliott's *push-pull* FRP [8], but (because it assumes the underlying I/O model is asynchronous) does not require any threading support.

3.1 Time intervals

The implementation is based on segments of signals, that is a signal defined over a semi-open interval $[s,t)$, where $s < t$. We also allow infinite segments, defined over the interval $[s,\infty)$: for this reason, we introduce the type Time^∞ of *time bounds*, which extends Time

$\mathsf{MSet} : \mathsf{Set}_1$
$\mathsf{MSet} =$
$\quad (A : \mathsf{Interval} \to \mathsf{Set}) \times$
$\quad (\forall\{i,j\}(i{\sim}j) \to A(i \frown j \because i{\sim}j) \to (A(i) \times A(j))) \times$
$\quad (\forall\{i,j\}(i \sqsubseteq j) \to A(j) \to A(i))$

$\mathsf{data}\ \mathsf{ISet} : \mathsf{Set}_1\ \mathsf{where}$
$\quad [\cdot] : \mathsf{MSet} \to \mathsf{ISet}$
$\quad (\cdot \Rrightarrow \cdot) : \mathsf{MSet} \to \mathsf{ISet} \to \mathsf{ISet}$

$\mathsf{I}[\![\cdot]\!] : \mathsf{ISet} \to \mathsf{Interval} \to \mathsf{Set}$
$\mathsf{I}[\![[A, \mathsf{split}, \mathsf{subsum}]]\!](i) = A(i)$
$\mathsf{I}[\![A \Rrightarrow B]\!](i) = \mathsf{I}[\![[A]]\!](i) \to \mathsf{I}[\![B]\!](i)$

$\mathsf{M}[\![\cdot]\!] : \mathsf{ISet} \to \mathsf{Interval} \to \mathsf{Set}$
$\mathsf{M}[\![[A]]\!](i) = \mathsf{I}[\![[A]]\!](i)$
$\mathsf{M}[\![A \Rrightarrow B]\!](i) = \forall\{j\}(j \sqsubseteq i) \to \mathsf{I}[\![A \Rrightarrow B]\!](j)$

$\mathsf{m2i} : \forall\{A, i\}\mathsf{M}[\![A]\!](i) \to \mathsf{I}[\![A]\!](i)$
$\mathsf{m2i}\{[A]\}(\sigma) = \sigma$
$\mathsf{m2i}\{A \Rrightarrow B\}(f) = f(i{\sqsubseteq}i)(\sigma)$

$\mathsf{i2m} : \forall\{A, i\}(\forall\{j\}(j \sqsubseteq i) \to \mathsf{I}[\![A]\!](j)) \to \mathsf{M}[\![A]\!](i)$
$\mathsf{m2i}\{[A]\}(\sigma) = \sigma(i{\sqsubseteq}i)$
$\mathsf{m2i}\{A \Rrightarrow B\}(f) = f$

$[\![\cdot]\!] : \mathsf{ISet} \to \mathsf{Set}$
$[\![A]\!] = \forall\{i\}\mathsf{I}[\![A]\!](i)$

Figure 17. Interval types

with ∞. The order on time can be extended to one on time bounds:

$$\mathsf{fin}(s) \preceq \mathsf{fin}(t) \prec \infty \qquad \text{when } s \le t$$

A *time interval* is of the form $[s{\prec}t)$, interpreted as:

$$t \in \mathsf{Int}[s{\prec}u) \text{ iff } s \preceq \mathsf{fin}(t) \text{ and } \mathsf{fin}(t) \prec u$$

There is a natural notion of inclusion order on intervals, such that:

$$i \sqsubseteq j \text{ iff } \mathsf{Int}(i) \subseteq \mathsf{Int}(j)$$

Two intervals i and j are *concatenatable* (written $i \sim j$) whenever they are of the form $[s{\prec}t)$ and $[t{\prec}u)$. Their *concatenation* (written $i \frown j \because i{\sim}j$) is $[s{\prec}u)$. These are formalized in Figures 15 and 16.

3.2 Interval types

Since the implementation is based on segments rather than signals (that is functions over intervals rather than points in time) the type system of the implementation is also based on *interval types*. A first cut definition for A to be an interval type is just:

$$A : \mathsf{Interval} \to \mathsf{Set}$$

Unfortunately, this type is not rich enough to define the FRP combinators. For example, consider the combinator f &&& g. When pushed an input segment ρ, it pushes ρ to both f and g, receiving back output σ and τ respectively. Now if σ and τ are segments over the same interval, then f &&& g can just return (σ, τ). However, consider when σ is over a smaller interval than τ: we need to *split* τ into subsegments τ_1 and τ_2, where τ_1 is over the same interval as σ; f &&& g can then return (σ, τ_1), and buffer τ_2 to be output later. As well as splitting, we provide a *subsumption* operation, which, for intervals $i \sqsubseteq j$, takes a segment σ over j, and returns a segment over i. Splitting and subsumption are interderivable, the only reason for supporting both is efficiency.

$\mathsf{splitM}[\![\cdot]\!] : (A : \mathsf{ISet}) \to \forall\{i,j\}$
$\quad (i{\sim}j) \to \mathsf{M}[\![A]\!](i \frown j \because i{\sim}j) \to (\mathsf{M}[\![A]\!](i) \times \mathsf{M}[\![A]\!](j))$
$\mathsf{splitM}[\![[A, \mathsf{split}, \mathsf{subsum}]]\!](t{=}t)(\sigma) = \mathsf{split}(t{=}t)(\sigma)$
$\mathsf{splitM}[\![A \Rrightarrow B]\!]\{[s{\prec}t)\}\{[t{\prec}u)\}(t{=}t)(f) = (f_1, f_2)\ \mathsf{where}$
$\quad f_1(s{\preceq}r, v{\preceq}t) = f(s{\preceq}r, v{\preceq}t{\preceq}u)$
$\quad f_2(t{\preceq}r, v{\preceq}u) = f(s{\preceq}t{\preceq}r, v{\preceq}u)$

$\mathsf{subsumM}[\![\cdot]\!] : (A : \mathsf{ISet}) \to \forall\{i,j\}$
$\quad (i \sqsubseteq j) \to \mathsf{M}[\![A]\!](j) \to \mathsf{M}[\![A]\!](i)$
$\mathsf{subsumM}[\![[A, \mathsf{split}, \mathsf{subsum}]]\!](i{\sqsubseteq}j)(\sigma) = \mathsf{subsum}(i{\sqsubseteq}j)(\sigma)$
$\mathsf{subsumM}[\![A \Rrightarrow B]\!](i{\sqsubseteq}j)(f) = \lambda h{\sqsubseteq}i\ .\ f(h)(h{\sqsubseteq}i{\sqsubseteq}j)$

$\mathsf{mset} : \mathsf{ISet} \to \mathsf{MSet}$
$\mathsf{mset}(A) = (\mathsf{M}[\![A]\!], \mathsf{splitM}[\![A]\!], \mathsf{subsumM}[\![A]\!])$

Figure 18. Translation of ISet into MSet

Summarizing, as well as $A : \mathsf{Interval} \to \mathsf{Set}$, we make use of two operations:

- split of type $A(i \frown j \because i{\sim}j) \to (A(i) \times A(j))$, and
- subsum of type $(i \sqsubseteq j) \to A(j) \to A(i)$.

We call an A equipped with split and subsum a *monotone interval type*, and formalize it as an MSet, defined in Figure 17.

Unfortunately, while we can use MSet to represent monotone operations such as products, we cannot use it to represent functions, which are anti-monotone in their first argument. An attempt to define a function space for MSets as:

$$\begin{array}{l} (\cdot \Rightarrow \cdot) : \mathsf{MSet} \to \mathsf{MSet} \to \mathsf{MSet} \\ (A \Rightarrow B) = (C, \mathsf{split}, \mathsf{subsum})\ \mathsf{where} \\ \quad C(i) = A(i) \to B(i) \\ \quad \mathsf{split} = ? \\ \quad \mathsf{subsum} = ? \end{array}$$

would fail, as there is no implementation of split or subsum. As is usual for the definition of a monotone function space over an order, we need to parameterize functions over all possible sub-intervals, that is we should have above:

$$C(i) = \forall\{j\}(j \sqsubseteq i) \to A(j) \to B(j)$$

For this reason, rather than taking MSets as primitive, we instead work with ISets, which are MSets closed under function spaces, as defined in Figure 17, for example if A, B and C are MSets, then $A \Rrightarrow B \Rrightarrow [C]$ is an ISet. There are two natural semantics for ISet:

- the *non-monotone* semantics $\mathsf{I}[\![\cdot]\!]$, where $A \Rrightarrow B$ is interpreted at i as the non-monotone $\mathsf{I}[\![A]\!](i) \to \mathsf{I}[\![B]\!](i)$, and
- the *monotone* semantics $\mathsf{M}[\![\cdot]\!]$, where $A \Rrightarrow B$ is interpreted at i as the monotone $\forall\{j\}(j \sqsubseteq i) \to \mathsf{I}[\![A \Rrightarrow B]\!](j)$.

These semantics, together with functions i2m and m2i which map between them, are given in Figure 17. Since $\mathsf{M}[\![A]\!]$ is monotone, we can define a function in Figure 18:

$$\mathsf{mset} : \mathsf{ISet} \to \mathsf{MSet}$$

Using this, we can define $(\cdot \Rightarrow \cdot)$ on ISets as in Figure 19.

$$A \Rightarrow B = \mathsf{mset}(A) \Rrightarrow B$$

At top level, the semantics exposed to the user is $[\![A]\!]$, which is defined for interval types in the same way as for reactive types:

$$[\![A]\!] = \forall\{i\}\mathsf{I}[\![A]\!](i)$$

$$(\cdot \Rightarrow \cdot) : \mathsf{ISet} \to \mathsf{ISet}$$
$$(A \Rightarrow B) = (\mathsf{mset}(A) \Rrightarrow B)$$
$$(\cdot \,\$\, \cdot) : \forall\{A, B, i\}\mathsf{M}[\![A \Rightarrow B]\!](i) \to \mathsf{M}[\![A]\!](i) \to \mathsf{M}[\![B]\!](i)$$
$$f \,\$\, \sigma = \mathsf{i2m}(\lambda j{\sqsubseteq}i \,.\, \mathsf{subsumM}[\![A]\!](j{\sqsubseteq}i)(\sigma))$$

Figure 19. Implementation of $\Rightarrow$

$$\uparrow : \mathsf{Time} \to \mathsf{Interval}$$
$$\uparrow t = [\mathsf{fin}(t){\prec}\infty)$$
$$\Uparrow : \mathsf{Interval} \to \mathsf{Interval}$$
$$\Uparrow [t{\prec}u) = [t{\prec}\infty)$$
$$\Box : \mathsf{ISet} \to \mathsf{ISet}$$
$$\Box A = [B, \mathsf{split}, \mathsf{subsum}] \text{ where}$$
$$\quad B(i) = \forall\{t\}(t \in \mathsf{Int}(i)) \to \mathsf{M}[\![A]\!](\uparrow t)$$
$$\quad \ldots$$
$$\mathsf{extend} : \forall\{A, B\}[\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!]$$
$$\mathsf{extend}\{A\}(f)(\sigma)(t{\in}i) =$$
$$\quad \mathsf{i2m}(\lambda j{\sqsubseteq}{\uparrow}t \,.\, f(\mathsf{subsumM}[\![A]\!](j{\sqsubseteq}{\uparrow}t)(\sigma(t{\in}i))))$$
$$\mathsf{extract} : \forall\{A\}[\![\Box A \Rightarrow A]\!]$$
$$\mathsf{extract}\{A\}(\sigma) = \mathsf{m2i}(\mathsf{subsumM}[\![A]\!](i{\sqsubseteq}{\Uparrow}i)(\sigma))$$
$$\mathsf{duplicate} : \forall\{A\}[\![\Box A \Rightarrow \Box\Box A]\!]$$
$$\mathsf{duplicate}\{A\}(\sigma)(t{\in}i)(u{\in}{\uparrow}t) = \mathsf{subsumM}[\![A]\!]({\uparrow}u{\sqsubseteq}{\uparrow}t)(\sigma(t{\in}i))$$
$$[\cdot] : \forall\{A\}[\![A]\!] \to [\![\Box A]\!]$$
$$[\sigma](t{\in}i) = \mathsf{i2m}(\lambda j{\sqsubseteq}{\uparrow}t \,.\, \sigma)$$
$$(\cdot \langle * \rangle \cdot) : \forall\{A, B\}[\![\Box(A \Rightarrow B) \Rightarrow \Box A \Rightarrow \Box B]\!]$$
$$(f \langle * \rangle \sigma)(t{\in}i) = f(t{\in}i) \,\$\, \sigma(t{\in}i)$$

Figure 20. Implementation of $\Box$

In particular, $[\![A \Rightarrow B]\!]$ is interpreted as a function space, so users can write proofs of implications as functions, for instance:

$$\mathsf{flip} : \forall\{A, B, C\}[\![(A \Rightarrow B \Rightarrow C) \Rightarrow B \Rightarrow A \Rightarrow C]\!]$$
$$\mathsf{flip}(f)(b)(a) = f \,\$\, a \,\$\, b$$

3.3 Temporal modalities

In Figure 20, we give the implementation of $\Box$, together with the structure of an applicative comonad. The other temporal modalities are implemented similarly.

3.4 Causal function space

We now turn to implementing the causal function space $A \trianglerighteq B$. The state of a causal function is modelled as a *process* of type:

$$(A \,@\, s \multimap B \,@\, u)$$

which has:

- *inputs* of the form $A[s, t)$ for some $t \succ s$, after which the function will be in state $(A \,@\, t \multimap B \,@\, u)$, and
- *outputs* of the form $B[u, v)$ for some $v \succ u$, after which the function will be in state $(A \,@\, s \multimap B \,@\, v)$.

$$\mathsf{codata}\ (\cdot \,@\, \cdot \multimap \cdot \,@\, \cdot)(A : \mathsf{Interval} \to \mathsf{Set})(s : \mathsf{Time}^\infty)$$
$$\quad (B : \mathsf{Interval} \to \mathsf{Set})(u : \mathsf{Time}^\infty) : \mathsf{Set} \text{ where}$$
$$\quad\quad \mathsf{inp} : (s \preceq u \prec \infty) \to$$
$$\quad\quad\quad (\forall\{t, s{\prec}t\}A[s{\prec}t) \to (A \,@\, t \multimap B \,@\, u)) \to$$
$$\quad\quad\quad\quad (A \,@\, s \multimap B \,@\, u)$$
$$\quad\quad \mathsf{out} : \forall\{v, u{\prec}v\}B[u{\prec}v) \to (A \,@\, s \multimap B \,@\, v) \to$$
$$\quad\quad\quad (A \,@\, s \multimap V \,@\, u)$$
$$\quad\quad \mathsf{done} : (u = \infty) \to$$
$$\quad\quad\quad (A \,@\, s \multimap V \,@\, u)$$
$$(\cdot \trianglerighteq \cdot) : \mathsf{ISet} \to \mathsf{ISet} \to \mathsf{ISet}$$
$$(A \trianglerighteq B) = [C, \mathsf{split}, \mathsf{subsum}] \text{ where}$$
$$\quad C(i) = \forall\{t\}(t \in \mathsf{Int}(i)) \to (\mathsf{M}[\![A]\!] \,@\, t \multimap \mathsf{M}[\![B]\!] \,@\, t)$$
$$\quad \ldots$$

Figure 21. Implementation of $\trianglerighteq$

Informally, such a process is one which has already received input up to time s, and has produced output up to time u. The initial state of a causal function at time t is a process where $s = t = u$.

To ensure that functions are causal, we require that processes are only input-enabled when $s \preceq u$: this ensures that any output can only depend on input in the past, not the future. Processes are defined in Figure 21, and are given as a coinductive syntax, with terms of the form:

- $\mathsf{inp}(s{\preceq}u{\prec}\infty)P$ where P is a function consuming a segment of type $A[s, t)$ and producing a process of type $A \,@\, t \multimap B \,@\, u$,
- $\mathsf{out}(\sigma)P$ where σ is a segment of type $B[u, v)$ and P is a process of type $A \,@\, t \multimap B \,@\, v$, or
- $\mathsf{done}(u{=}\infty)$.

Note that a process can terminate when $u = \infty$, that is when it has produced all of its output, even if $s \prec \infty$ and so there may still be outstanding input. For example, an identity process can be defined:

$$P : \forall\{t\}(A \,@\, t \multimap A \,@\, t)$$
$$P\{\infty\} = \mathsf{done}(\infty{=}\infty)$$
$$P\{t\} = \mathsf{inp}(t{\preceq}t{\prec}\infty)(\lambda\sigma \,.\, \mathsf{out}(\sigma)P)$$

This coinductive presentation of causal functions is similar to Hennessy and Plotkin's resumption model of concurrency [13], Ghani, Hancock and Pattinson's eater model of stream consumers [10] and Jeffrey and Rathke's model of streaming I/O in Agda [17].

In Figure 22 we give the categorical structure of causal functions. The identity is just inherited from Set. Of more interest is composition $f \ggg g$, which is defined in terms of process chaining:

$$P \gg Q : A \,@\, s \multimap C \,@\, u$$

where:

- P is a process of type $A \,@\, s \multimap B \,@\, t$, and
- Q is a process of type $B \,@\, t \multimap C \,@\, u$.

The runtime behaviour of $P \gg Q$ is:

- If Q is output-enabled, then so is $P \gg Q$.
- If P and Q are both input-enabled, then so is $P \gg Q$.
- If P is output-enabled and Q is input-enabled, then P's output is passed to Q.
- If Q is terminated, then so is $P \gg Q$.

This notion of composition is very similar to chaining in process calculi [24], or zig-zag plays in games semantics [1, 16].

$$\mathsf{arr} : \forall\{A, B\}[\![\Box(A \Rightarrow B) \Rightarrow (A \trianglerighteq B)]\!]$$
$$\mathsf{arr}(f)(t{\in}i) = P(t, f(t{\in}i)) \text{ where}$$
$$P(\infty, f) = \mathsf{done}(\infty{=}\infty)$$
$$P(t, f) = \mathsf{inp}(t{\preceq}t{\prec}\infty)P' \text{ where}$$
$$P'\{t{\prec}u\}(\sigma) = \mathsf{out}(f_1 \,\$\, \sigma)P(u, f_2) \text{ where}$$
$$(f_1, f_2) = \mathsf{splitM}[\![A \Rightarrow B]\!](u{=}u)(f)$$

$$\mathsf{identity} : \forall\{A\}[\![A \trianglerighteq A]\!]$$
$$\mathsf{identity} = \mathsf{arr}[\lambda a \,.\, a]$$

$$(\cdot \ggg \cdot) : \forall\{A, B, C\}[\![(A \trianglerighteq B) \Rightarrow (B \trianglerighteq C) \Rightarrow (A \trianglerighteq C)]\!]$$
$$(f \ggg g)(t{\in}i) = f(t{\in}i) \gg g(t{\in}i) \text{ where}$$
$$P \gg \mathsf{out}(\tau)Q =$$
$$\mathsf{out}(\tau)(P \gg Q)$$
$$\mathsf{inp}(s{\preceq}t{\prec}\infty)P \gg \mathsf{inp}(t{\preceq}u{\prec}\infty)Q =$$
$$\mathsf{inp}(s{\preceq}t{\preceq}u{\prec}\infty)(\lambda\sigma \,.\, (P(\sigma) \gg \mathsf{inp}(t{\preceq}u{\prec}\infty)Q))$$
$$\mathsf{out}(\sigma)P \gg \mathsf{inp}(t{\preceq}u{\prec}\infty)Q =$$
$$P \gg Q(\sigma)$$
$$P \gg \mathsf{done}(u{=}\infty) =$$
$$\mathsf{done}(u{=}\infty)$$

Figure 22. Implementation of categorical structure

The proof that $P \gg Q$ terminates is slightly subtle, because the process syntax is defined coinductively, and so a proof is required to show that there is not an infinite number of outputs that can be passed from P to Q without an external interaction. In a discrete time domain, the result follows, because each output is required to be non-empty, and $\succ$ is well-founded on an interval $[s, u)$ when $u \prec \infty$. The need to show $P \gg Q$ to be well-defined is the reason why we place the precondition $u \prec \infty$ on input, which in turn is the reason for supporting done (as otherwise there would be no inhabitants of $A \mathbin{@} s \multimap B \mathbin{@} \infty$).

3.5 Products

In Figure 23 we give the implementation of product structure. On interval types, products are just inherited from $\mathbf{Set}$:

$$\mathsf{M}[\![A \wedge B]\!](i) = \mathsf{M}[\![A]\!](i) \times \mathsf{M}[\![B]\!](i)$$

The interesting definition is the mediating function $f \mathbin{\&\&\&} g$, which is defined in terms of a mediating process:

$$P \mathbin{\&\&} Q : A \mathbin{@} s \multimap (B \wedge C) \mathbin{@} t$$

where:

- P is a process of type $A \mathbin{@} s \multimap B \mathbin{@} t$, and
- Q is a process of type $A \mathbin{@} s \multimap C \mathbin{@} t$.

The runtime behaviour of $P \mathbin{\&\&} Q$ is:

- If P or Q are input-enabled then $P \mathbin{\&\&} Q$ is input-enabled, and the input segment is copied to both P and Q.
- If P or Q are terminated, then so is $P \mathbin{\&\&} Q$.
- If P and Q are both output-enabled, then we have three cases to consider:
  - P's output σ is shorter than Q's output τ, in which case we split τ into (τ_1, τ_2), output (σ, τ_1) and keep τ_2 in Q's output buffer,
  - P's output σ is the same length as Q's output τ, in which case we output (σ, τ), or
  - P's output σ is longer than Q's output τ, in which case we split σ into (σ_1, σ_2), output (σ_1, τ) and keep σ_2 in P's output buffer.

$$(\cdot \wedge \cdot) : \mathsf{ISet} \to \mathsf{ISet} \to \mathsf{ISet}$$
$$(A \wedge B) = [C, \mathsf{split}, \mathsf{subsum}] \text{ where}$$
$$C(i) = \mathsf{M}[\![A]\!](i) \times \mathsf{M}[\![B]\!](i)$$
$$\cdots$$

$$\mathsf{fst} : \forall\{A, B\}[\![A \wedge B \trianglerighteq A]\!]$$
$$\mathsf{fst} = \mathsf{arr}[\lambda(a, b) \,.\, a]$$

$$\mathsf{snd} : \forall\{A, B\}[\![A \wedge B \trianglerighteq B]\!]$$
$$\mathsf{snd} = \mathsf{arr}[\lambda(a, b) \,.\, b]$$

$$(\cdot \mathbin{\&\&\&} \cdot) : \forall\{A, B, C\}[\![(A \trianglerighteq B) \Rightarrow (A \trianglerighteq C) \Rightarrow (A \trianglerighteq B \wedge C)]\!]$$
$$(f \mathbin{\&\&\&} g)(t{\in}i) = f(t{\in}i) \mathbin{\&\&} g(t{\in}i) \text{ where}$$
$$P \mathbin{\&\&} \mathsf{inp}(s{\preceq}u{\prec}\infty)Q =$$
$$\mathsf{inp}(s{\preceq}u{\prec}\infty)(\lambda\rho \,.\, (P/\rho) \mathbin{\&\&} Q(\rho))$$
$$\mathsf{inp}(s{\preceq}u{\prec}\infty)P \mathbin{\&\&} Q =$$
$$\mathsf{inp}(s{\preceq}u{\prec}\infty)(\lambda\rho \,.\, P(\rho) \mathbin{\&\&} (Q/\rho))$$
$$P \mathbin{\&\&} \mathsf{done}(t{=}\infty) =$$
$$\mathsf{done}(t{=}\infty)$$
$$\mathsf{done}(t{=}\infty) \mathbin{\&\&} Q =$$
$$\mathsf{done}(t{=}\infty)$$
$$\mathsf{out}(t{\prec}u)(\sigma)P \mathbin{\&\&} \mathsf{out}(t{\prec}v)(\tau)Q =$$
$$\mathsf{if}\ (u \prec v)$$
$$\mathsf{then}\ (\mathsf{out}(t{\prec}u)(\sigma, \tau_1)(P \mathbin{\&\&} \mathsf{out}(u{\prec}v)(\tau_2)Q) \text{ where}$$
$$(\tau_1, \tau_2) = \mathsf{splitM}[\![C]\!](u{=}u)(\tau))$$
$$\mathsf{else\ if}\ (u = v)$$
$$\mathsf{then}\ (\mathsf{out}(t{\prec}u)(\sigma, \tau)(P \mathbin{\&\&} Q))$$
$$\mathsf{else}\ (\mathsf{out}(t{\prec}u)(\sigma_1, \tau)(\mathsf{out}(v{\prec}u)(\sigma_2)P \mathbin{\&\&} Q) \text{ where}$$
$$(\sigma_1, \sigma_2) = \mathsf{splitM}[\![B]\!](u{=}u)(\sigma))$$

Figure 23. Implementation of product structure

$$(\cdot/\cdot) : \forall\{A, B, s{\prec}t, u\}$$
$$(A \mathbin{@} s \multimap B \mathbin{@} u) \to A[s{\prec}t) \to (A \mathbin{@} t \multimap B \mathbin{@} u)$$
$$\mathsf{inp}(s{\preceq}u{\prec}\infty)P/\sigma = P(\sigma)$$
$$\mathsf{out}(\tau)P/\sigma = \mathsf{out}(\tau)(P/\sigma)$$
$$\mathsf{done}(u{=}\infty)/\sigma = \mathsf{done}(u{=}\infty)$$

Figure 24. Implementation of the "after" operation on processes

The use of split in defining the product structure is the motivation for introducing MSet. In the input-enabled case, we make use of an auxiliary operator P/σ, defined in Figure 24, which applies a process P to an input segment σ.

3.6 Loops

In Figure 25 we give the implementation of decoupled functions, which is the same as for causal functions, except that the precondition on input is strengthened from $(s \preceq u \prec \infty)$ to $(s \prec u \prec \infty)$, that is, a decoupled process's output can only depend on input in the strict past.

For decoupled functions, looping is implemented in Figure 26 in terms of three tracing processes:

$$
\begin{array}{l}
\mathsf{codata}\ (\cdot \mathbin{@} \cdot \multimapdot \cdot \mathbin{@} \cdot)(A : \mathsf{Interval} \to \mathsf{Set})(s : \mathsf{Time}^{\infty}) \\
\quad (B : \mathsf{Interval} \to \mathsf{Set})(u : \mathsf{Time}^{\infty}) : \mathsf{Set}\ \mathsf{where} \\
\quad\quad \mathsf{inp} : (s \prec u \prec \infty) \to \\
\quad\quad\quad (\forall \{t, s{\prec}t\} A[s{\prec}t) \to (A \mathbin{@} t \multimapdot B \mathbin{@} u)) \to \\
\quad\quad\quad\quad (A \mathbin{@} s \multimapdot B \mathbin{@} u) \\
\quad\quad \mathsf{out} : \forall \{v, u{\prec}v\} B[u{\prec}v) \to (A \mathbin{@} s \multimapdot B \mathbin{@} v) \to \\
\quad\quad\quad (A \mathbin{@} s \multimapdot V \mathbin{@} u) \\
\quad\quad \mathsf{done} : (u = \infty) \to \\
\quad\quad\quad (A \mathbin{@} s \multimapdot V \mathbin{@} u) \\
(\cdot \rhd \cdot) : \mathsf{ISet} \to \mathsf{ISet} \to \mathsf{ISet} \\
(A \rhd B) = [C, \mathsf{split}, \mathsf{subsum}]\ \mathsf{where} \\
\quad C(i) = \forall \{t\}(t \in \mathsf{Int}(i)) \to (\mathsf{M}[\![A]\!] \mathbin{@} t \multimapdot \mathsf{M}[\![B]\!] \mathbin{@} t) \\
\quad \ldots
\end{array}
$$

Figure 25. Implementation of $\rhd$

- $\mathsf{tr}(P) : B \mathbin{@} s \multimapdot C \mathbin{@} s$, where:
  - P is a process of type $(A \wedge B) \mathbin{@} s \multimapdot (A \wedge C) \mathbin{@} s$,
- $\mathsf{tr}_{\prec}(s{\prec}u)(\rho)(P) : B \mathbin{@} s \multimapdot C \mathbin{@} u$, where:
  - ρ is a segment of type $\mathsf{M}[\![A]\!][s{\prec}u)$, and
  - P is a process of type $(A \wedge B) \mathbin{@} s \multimapdot (A \wedge C) \mathbin{@} u$, and
- $\mathsf{tr}_{\succ}(s{\succ}u)(\sigma)(P) : B \mathbin{@} s \multimapdot C \mathbin{@} u$, where:
  - σ is a segment of type $\mathsf{M}[\![B]\!][u{\prec}s)$, and
  - P is a process of type $(A \wedge B) \mathbin{@} u \multimapdot (A \wedge C) \mathbin{@} u$.

The three processes correspond to the three cases given by considering how much B input has been provided, compared to the amount of A output which has been generated:

- $\mathsf{tr}(P)$ covers the case where output is provided up to time u, and input is provided up to time $s = u$, so no buffering is required,
- $\mathsf{tr}_{\prec}(s{\prec}u)(\rho)P$ covers the case where output is provided up to time u, and input is provided up to time $s \prec u$, where ρ is the buffered output, and
- $\mathsf{tr}_{\succ}(s{\succ}u)(\sigma)P$ covers the case where output is provided up to time u, and input is provided up to time $s \succ u$, where σ is the buffered input.

The runtime behaviour of tracing is:

- If P is terminated, then we are terminated.
- If $s \prec u$ then we have an output buffer ρ, and we are input-enabled. After inputting a segment σ of type $\mathsf{M}[\![B]\!][s{\prec}t)$ we compare t and u:
  - If $t \prec u$ then we split ρ into (ρ_1, ρ_2), apply P to (ρ_1, σ), and continue with remaining output buffer ρ_2.
  - If $t = u$ then we apply P to (ρ, σ), and continue without any buffering.
  - If $t \succ u$ then we split σ into (σ_1, σ_2), apply P to (ρ, σ_2), and continue with remaining input buffer σ_2.
- If $s \succ u$ then P must be output-enabled, with output (ρ, τ), so we output τ, and continue in a similar fashion.
- If $s = u$ then P must be output-enabled, with output (ρ, τ), so we we output τ, and continue with output buffer ρ.

Showing that tr is well-defined is direct, since it is a guarded recursion. If we tried replaying the same definition for $\multimap$ rather

$$
\begin{array}{l}
\mathsf{loop} : \forall \{A, B, C\} [\![((A \wedge B) \rhd (A \wedge C)) \Rightarrow B \rhd C]\!] \\
\mathsf{loop}(f)(t{\in}i) = \mathsf{tr}(f(t{\in}i))\ \mathsf{where} \\
\quad \mathsf{tr}(\mathsf{out}(s{\prec}u)(\rho, \tau)P) = \mathsf{out}(s{\prec}u)(\tau)(\mathsf{tr}_{\prec}(s{\prec}u)(\rho)(P)) \\
\quad \mathsf{tr}(\mathsf{done}(s{=}\infty)) = \mathsf{done}(s{=}\infty) \\
\quad \mathsf{tr}_{\prec}(s{\prec}u)(\rho)P = \\
\quad\quad \mathsf{if}\ (u = \infty) \\
\quad\quad \mathsf{then}\ (\mathsf{done}(u{=}\infty)) \\
\quad\quad \mathsf{else}\ (\mathsf{inp}(s{\prec}u{\prec}\infty)(\lambda s{\prec}t\ .\ \lambda\sigma\ . \\
\quad\quad\quad \mathsf{if}\ (t \prec u) \\
\quad\quad\quad \mathsf{then}\ (\mathsf{tr}_{\prec}(t{\prec}u)(\rho_2)(P/(\rho_1, \sigma))\ \mathsf{where} \\
\quad\quad\quad\quad (\rho_1, \rho_2) = \mathsf{splitM}[\![A]\!](t{=}t)(\rho)) \\
\quad\quad\quad \mathsf{else\ if}\ (t = u) \\
\quad\quad\quad \mathsf{then}\ (\mathsf{tr}(P/(\rho, \sigma))) \\
\quad\quad\quad \mathsf{else}\ (\mathsf{tr}_{\succ}(t{\succ}u)(\sigma_2)(P/(\rho, \sigma_1))\ \mathsf{where} \\
\quad\quad\quad\quad (\sigma_1, \sigma_2) = \mathsf{splitM}[\![B]\!](u{=}u)(\sigma)))) \\
\quad \mathsf{tr}_{\succ}(s{\succ}u)(\sigma)(\mathsf{done}(u{=}\infty)) = \mathsf{done}(u{=}\infty) \\
\quad \mathsf{tr}_{\succ}(s{\succ}u)(\sigma)(\mathsf{out}(u{\prec}v)(\rho, \tau)P) = \\
\quad\quad \mathsf{out}(u{\prec}v)(\tau) \\
\quad\quad\quad \mathsf{if}\ (s \prec v) \\
\quad\quad\quad \mathsf{then}\ (\mathsf{tr}_{\prec}(s{\prec}v)(\rho_2)(P/(\rho_1, \sigma))\ \mathsf{where} \\
\quad\quad\quad\quad (\rho_1, \rho_2) = \mathsf{splitM}[\![A]\!](s{=}s)(\rho)) \\
\quad\quad\quad \mathsf{else\ if}\ (s = v) \\
\quad\quad\quad \mathsf{then}\ (\mathsf{tr}(P/(\rho, \sigma))) \\
\quad\quad\quad \mathsf{else}\ (\mathsf{tr}_{\succ}(s{\succ}v)(\sigma_2)(P/(\rho, \sigma_1))\ \mathsf{where} \\
\quad\quad\quad\quad (\sigma_1, \sigma_2) = \mathsf{splitM}[\![B]\!](v{=}v)(\sigma))
\end{array}
$$

Figure 26. Implementation of loop

than $\multimapdot$, it would fail because we would have to provide a definition for the case where $s \succ u$ but P is input-enabled.

4. Future work

There are a number of open problems, of which the most important is that the implementation is a skeleton, and needs to be fleshed out to include more FRP combinators. Currently, there is no mechanized proof that the implementation matches the specification. The combinators are very similar to those of the Agda streaming I/O library [17], which has been mechanized, so we expect the proofs should go through, but this is still future work.

Assuming the soundness of the underlying logic, we have a direct soundness result, that every functional reactive program proves a tautology in LTL. The other direction is trickier, because it requires finding a complete derivation system for constructive LTL. There are existing complete systems for classical LTL [22], but the constructive case is less well developed. Kojima and Igarashi [20] have investigated the fragment with just a $\bigcirc$ modality, it would be interesting to see if their framework extends to the $\trianglerighteq$ modality.

We have given a model of stateful reactive programs, but not of stateful type functions. For example, an RSet is isomorphic to $[\![\langle \mathsf{Set} \rangle]\!]$, and reactive type constructors can be seen as a reactive programs, for example $(\cdot \trianglerighteq \cdot) : [\![\Box\langle \mathsf{Set} \rangle \Rightarrow \Box\langle \mathsf{Set} \rangle \Rightarrow \langle \mathsf{Set} \rangle]\!]$. It might be interesting to investigate stateful type-level reactive functions, including those with loops, thus lifting causal functions from the program level to the type level.

The implementation described here is based on interval types, rather than reactive types. This might be a better match to an *interval temporal logic* rather than LTL, which would admit the "chop" modality, and may give cartesian-closed structure to $\trianglerighteq$.

The treatment of fixed points is very similar to the ultrametric semantics of Krishnaswami and Benton [21], which leads to a question of what the right notion of partial trace is for a complete metric space? Neither Haghverdi and Scott's [11], not Abramsky, Blute and Panangaden's [2] notions of trace class are satisfied by contraction maps (in both cases, Vanishing 2 is the main stumbling block). It would also be interesting to investigate the appropriate graphical presentation of a partial trace, as this would determine the presentation of FRP programs as dataflow graphs.

Finally, the model is a discrete-time model rather than dense-time. The unit of communication is a segment which may be much longer than the unit of time, so there is still a benefit from the FRP approach, but the assumption of discrete time is at odds with the usual FRP semantics. There are two places where discrete time was used: in the proof that composition is well-defined, and in the choice functions which return the first or last events in a signal.

The difficulty with dense time for composition is caused by so-called *Zeno processes* which perform output which is successively smaller, for example:

$$P : (t : \mathsf{Time}) \to \mathsf{M}[\![B]\!][t, \infty) \to \mathbb{R} \to (A \mathbin{@} s \multimap B \mathbin{@} t)$$
$$P(t)(\sigma)(d) = \mathsf{out}(\sigma_1)(P(t+d)(\sigma_2)(d/2)) \text{ where}$$
$$(\sigma_1, \sigma_2) = \mathsf{splitM}[\![B]\!](t{+}d{=}t{+}d)(\sigma)$$

Starting with $t = 0$ and $d = 1/2$, this process will output the $[0, 1/2)$-prefix of σ, then the $[1/2, 3/4)$-prefix, then the $[3/4, 7/8)$-prefix, and so on, never reaching time 1. Chaining P into a process that is always input-enabled in the interval $[0, 1)$ results in an infinite amount of chatter, with no external interaction. This is a difficult problem to deal with: banning Zeno processes without banning safe uses of split is difficult.

In a dense time model, there are no choice functions, due to Zeno event signals. Consider a signal which alternates between a true event and a false event at time $1 - 2^{-n}$, which has no canonical last event in the interval $[0, 1]$. Any attempt to ban Zeno event signals would run afoul of the edge function, which can convert an arbitrary function of type $\mathsf{Time} \to \mathsf{Bool}$ into a signal. In the presence of edge, we require that any function of type $\mathsf{Time} \to \mathsf{Bool}$ have only finitely many edges over a closed interval. This, for example, rules out $\mathbb{Q}$ as a time model, although there may be appropriate models of the exact reals, such as [6].

Acknowledgments

Thanks to Kedar Namjoshi and Corin Pitcher for discussions about this paper, and to the anonymous referees for valuable feedback.

The dataflow diagrams on page 5 are from the Yampa documentation [34].

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. and Computation*, 163:409–470, 1996.

[2] S. Abramsky, R. Blute, and P. Panangaden. Nuclear and trace ideals in tensored *-categories. *J. Pure and Applied Algebra*, 143:3–47, 1999.

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

[4] M. Carlsson and T. Hallgren. *Fudgets: Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, 1998.

[5] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Proc. Haskell Workshop*, 2001.

[6] A. Edalat and P. J. Potts. A new representation for exact real numbers. In *Proc. Math. Foundations of Progamming Semantics*, pages 119–132, 1997.

[7] C. Elliott. Functional implementations of continuous modeled animation. In *Proc. PLILP/ALP*, 1998.

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

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

[10] N. Ghani, P. Hancock, and D. Pattinson. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3), 2009.

[11] E. Haghverdi and P. J. Scott. Towards a typed geometry of interaction. In *Proc. Computer Science Logic*, pages 216–231, 2005.

[12] D. Harel, D. C. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. In *Proc. IEEE Symp. Foundations of Computer Science*, pages 129–142, 1980.

[13] M. Hennessy and G. D. Plotkin. Full abstraction for a simple programming language. In *Proc. Math. Foundations of Computer Science*, number 74 in Lecture Notes in Computer Science, pages 108–120. Springer, 1979.

[14] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2003.

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

[16] J. M. E. Hyland and C.-H. Ong. On full abstraction for PCF. *Inf. and Computation*, 163:285–408, 2000.

[17] A. S. A. Jeffrey and J. Rathke. The lax braided structure of streaming i/o. In *Proc. Conf. Computer Science Logic*, 2011.

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

[19] W. Jeltsch. Programming in linear temporal logic. `http://cs.ioc.ee/~tarmo/tsem10/jeltsch-slides.pdf`, 2011.

[20] K. Kojima and A. Igarashi. Constructive linear-time temporal logic: Proof systems and Kripke semantics. *Inf. and Computation*, to appear.

[21] N. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proc. IEEE Logic in Computer Science*, 2011.

[22] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Sci.*, 83:91–130, 1991.

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

[24] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

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

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

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

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

[30] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Sci.*, 58:249–261, 1988.

[31] R. Rosner and A. Pnueli. A choppy logic. In *Proc. IEEE Symp. Logic in Computer Science*, pages 306–313, 1986.

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

[33] P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, chapter 4, pages 289–356. Springer, 2011.

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