Functional Reactive Types

Alan Jeffrey
Alcatel-Lucent Bell Labs

***Abstract*—Functional Reactive Programming (FRP) is an approach to streaming data with a pure functional semantics as time-indexed values. In previous work, we showed that Linear-time Temporal Logic (LTL) can be used as a type system for discrete-time FRP, and that functional reactive primitives perform two roles: as combinators for building streams of data, and as proof rules for constructive LTL. In this paper, we add a third role, by showing that FRP combinators can be used to define streams of types, and that these functional reactive types can be viewed both as a constructive temporal logic, and as the types for functional reactive programs. As an application of functional reactive types, we show that past-time LTL (pLTL) can be extended with FRP to get a logic pLTL+FRP. This logic is expressed as streams of boolean expressions, and so bounded satisfiability of pLTL can be translated to Satisfiability Modulo Theory (SMT). Thus, pLTL+FRP can be used as a constraint language for problems which mix properties of data with temporal properties.**

I. Introduction

The slogan "propositions are types, proofs are programs" can be indexed by time, to give a new slogan "temporal propositions are functional reactive types, proofs are functional reactive programs." This is the core idea behind the author's prior work [11], developed simultaneously by Jeltsch [14], showing that a constructive variant of linear-time temporal logic (LTL) [21] can be regarded as a type system, whose proof objects are expressed using discrete-time Functional Reactive Programming (FRP) [7].

In this paper, we further explore the connection between FRP and temporal logic, by showing that not only can FRP programs express the proofs of temporal propositions, they can also express the propositions themselves. We do this by defining FRP in a dependently typed functional language, in which there is no distinction between the language of types and the language of values. The type of streams is $[As]$ where As is a stream of types; the type $[A_0 :: A_1 :: A_2 :: \cdots]$ is inhabited by stream $(v_0 :: v_1 :: v_2 :: \cdots)$ where each v_i has type A_i. For example, the stream $(1 :: \mathsf{true} :: 2 :: \mathsf{false} :: \cdots)$ has type $[\mathbb{N} :: \mathbb{B} :: \mathbb{N} :: \mathbb{B} :: \cdots]$.

Constructively, types can be viewed as propositions: a proof of the proposition A is given by a value v of type A. A stream of types $(A_0 :: A_1 :: A_2 :: \cdots)$ can thus be seen as a temporal property, which is true at time k whenever A_k is inhabited. A stream of values $(v_0 :: v_1 :: v_2 :: \cdots)$ of type $[A_0 :: A_1 :: A_2 :: \cdots]$ provides a witness v_i for each A_i. Thus, inhabitance of the type $[As]$ can be viewed as a proof that As is true at all times, that is that As is a tautology.

In [11], we showed that the combinators of an FRP library can perform double duty: as well as their usual use as the building blocks of reactive programs, they can be seen as proof combinators for proving tautologies in a constructive variant of LTL. In this paper, we add another use for FRP combinators: they allow the construction of streams of types, which in turn can be used to give the types of the combinators themselves. Thus, functional reactive programs can be used to construct *functional reactive types*.

For example, consider the "head" and "tail" functions:

$$!(x :: xs) \equiv x \qquad \bullet\,(x :: xs) \equiv xs$$

Now, consider the type of the "head" function: it takes a stream $(v :: vs)$, whose type is $[A :: As]$, and returns a value v of type A. Now, A is the head of $(A :: As)$, so its type is:

$$! : [As] \rightarrow !As$$

Similarly, the type of "tail" is:

$$\bullet : [As] \rightarrow [\bullet As]$$

This simple example shows the triple play of functional reactive programs with functional reactive types:

1) As a function on streams of values, $\bullet$ is the familiar "tail" function.
2) As a function on streams of types, $\bullet$ is the "next time" modality of LTL: $\bullet As$ is inhabited at time k when As is inhabited at time $k+1$.
3) As a proof combinator, $\bullet$ takes a proof of $[As]$ and returns a proof of $[\bullet As]$, that is if As is tautology, then $\bullet As$ is a tautology.

In particular, we can define a recursive function indn which performs iteration over streams:

$$\mathsf{indn}(f :: fs)(x) \equiv (x :: \mathsf{indn}(fs)(f(x)))$$

As a function on streams, this is an "accumulating fold":

$$\begin{aligned}&\mathsf{indn}(f_0 :: f_1 :: f_2 :: \cdots)(x)\\ &\quad\equiv (x :: f_0(x) :: f_1(f_0(x)) :: f_2(f_1(f_0(x))) :: \cdots)\end{aligned}$$

which can be used to define functions such as sum, which provides a running total of a stream of numbers:

$$\begin{aligned}&\mathsf{sum}(x_0 :: x_1 :: x_2 :: \cdots)\\ &\quad\equiv (x_0 :: (x_1 + x_0) :: (x_2 + x_1 + x_0) :: \cdots)\end{aligned}$$

As a function on streams of types, it can be used to define modalities such as "always in the past" ($\Box$), where $\Box As$ is inhabited at time k whenever As is inhabited at every time $j \le k$, that is it gives a running product of a stream of types:

$$\begin{aligned}&\Box(A_0 :: A_1 :: A_2 :: \cdots)\\ &\quad\equiv (A_0 :: (A_1 \times A_0) :: (A_2 \times A_1 \times A_0) :: \cdots)\end{aligned}$$

As a proof combinator, it is an induction principle for LTL:

$$\mathsf{indn} : [As \Rightarrow \bullet As] \to (!As) \to [As]$$

where $\Rightarrow$ is the pointwise lifting of the function space to streams of types:

$$(A :: As) \Rightarrow (B :: Bs) \equiv (A \to B) :: (As \Rightarrow Bs)$$

That is, indn says that if truth of As at time k implies truth of As at time $k+1$, and As is true at time 0, then As is true at any time k, which is the usual induction scheme over $\mathbb{N}$. This use of induction over $\mathbb{N}$ to give an LTL type for recursive FRP programs is similar to Krishnaswami and Benton's [17] use of contraction maps in ultrametric spaces.

In the remainder of this paper, we will make the notion of functional reactive type more precise. Section II gives the mathematical preliminaries for the paper. Section III defines the stream combinators, and their types. Section IV shows how past-time LTL (pLTL) can be coded as streams of types.

As an example of pLTL with FRP, in Section V, we show how to define streams of expressions in a theory of booleans and integers, such that satisfiability of a boolean expression corresponds to satisfiability of a pLTL formula. This allows expression of pLTL formulae which makes use of stream expressions constructed using FRP combinators, for example:

$$\Box(\mathsf{sum}(xs) + ys = 0)$$

The expression language used is suitable for passing on to an SMT solver (see, for example de Moura and Bjørner's survey [5]), and so gives a simple algorithm for checking k-bounded satisfiability of pLTL+FRP: construct a stream of expressions $(E_0 :: E_1 :: E_2 :: \cdots)$, and pass expression E_k to an SMT solver. This algorithm has been implemented, using Z3 [4] as the solver, with promising execution times (sub-second on the examples tested).

Sections II–V of this paper are written in Literate Agda [1], and all formal definitions and results are typechecked Agda code [12].

II. Mathematical preliminaries

This paper uses Agda [1] as its expression of constructive mathematics and dependent type theory. We will elide some of Agda's technical machinery, such as extensionality, inferrable arguments, universes and universe polymorphism. In particular we will work as if the type of types were itself a type (in the formal development we use universe polymorphism instead of making this unsound assumption):

$$\star : \star$$

We write $\times$ for product, $\uplus$ for coproduct, $\bot$ for empty, and $\top$ for singleton:

$$\begin{aligned}
(\_ \times \_) &: \star \to \star \to \star \\
\mathsf{fst} &: (A \times B) \to A \quad \mathsf{snd} : (A \times B) \to B \\
\mathsf{both} &: (A \to B) \to (A \to C) \to A \to (B \times C) \\
\mathsf{map}^{\times} &: (A \to B) \to (C \to D) \to ((A \times C) \to (B \times D))
\end{aligned}$$

$$\begin{aligned}
(\_ \uplus \_) &: \star \to \star \to \star \\
\mathsf{inl} &: A \to (A \uplus B) \quad \mathsf{inr} : B \to (A \uplus B) \\
\mathsf{case} &: (A \to C) \to (B \to C) \to (A \uplus B) \to C \\
\mathsf{map}^{\uplus} &: (A \to B) \to (C \to D) \to ((A \uplus C) \to (B \uplus D))
\end{aligned}$$

$$\top : \star \quad * : \top$$

$$\bot : \star \quad \mathsf{contradiction} : \bot \to A$$

We write $\forall$ for universal and $\exists$ for existential quantification, (where $B : A \to \star$):

$$(\forall x \to B(x)) : \star \quad (\exists x \to B(x)) : \star$$

Universal quantification is a function space, and existential quantification is a product, inhabited by (where $L : A$, $M : B(L)$ and $N(x) : B(x)$):

$$(\lambda x \to N(x)) : (\forall x \to B(x)) \quad (L, M) : (\exists x \to B(x))$$

We write $(A \to^d B)$ as a synonym for universal quantification:

$$\begin{aligned}
(\_ \to^d \_) &: (\forall (A : \star) \to (A \to \star) \to \star) \\
(A \to^d B) &\equiv (\forall x \to B(x))
\end{aligned}$$

The type $(x \equiv y)$ is inhabited whenever x and y are propositionally equivalent; its only constructor is $*$ of type $(x \equiv x)$:

$$(\_ \equiv \_) : A \to A \to \star \quad * : (x \equiv x)$$

We use $\equiv$ to define $\leq$ on the natural numbers:

$$(\_ \leq \_) : \mathbb{N} \to \mathbb{N} \to \star \quad (m \leq n) \equiv (\exists \ell \to \ell + m \equiv n)$$

$\mathbb{P}(A)$ is the constructive powerset:

$$\mathbb{P} : \star \to \star \quad \mathbb{P}(A) \equiv (A \to \star)$$

We define the usual constructions on sets, in particular $x \in S$ is true whenever $S(x)$ is inhabited, and set comprehension is just an abbreviation for λ-abstraction:

$$(x \in S) \equiv S(x) \quad \{\, x \mid P(x) \,\} \equiv (\lambda x \to P(x))$$

All formal definitions and results in this paper are written in Literate Agda, and typecheck.

III. Functional Reactive Programs and Types

In this section, we formalize our notion of streams, and define FRP combinators which can be used to define streams of values, streams of types, and tautologies in constructive temporal logic.

We begin with the type of homogeneous streams, defined in Figure 1. The type A^ω is the type of homogeneous streams, all of whose elements are of type A. In keeping with the FRP tradition, streams are defined as functions $\mathsf{Time} \to A$, and since we are interested in discrete-time FRP, A^ω is just a

$$\begin{aligned}(\_^{\omega}) &: \star \to \star \\ A^{\omega} &= \mathbb{N} \to A \\[4pt] \langle \_ \rangle &: A \to A^{\omega} \\ \langle x \rangle(n) &= x\end{aligned}$$

Fig. 1. Homogeneous FRP

$$\begin{aligned}[\_] &: \star^{\omega} \to \star \\ [As] &= (\forall n \to As(n)) \\[4pt] ! &: [As] \to As(0) \\ !xs &= xs(0) \\[4pt] \bullet &: [As] \to (\forall n \to As(n+1)) \\ (\bullet xs)(n) &= xs(n+1) \\[4pt] (\_ :: \_) &: As(0) \to (\forall n \to As(n+1)) \to [As] \\ (x :: xs)(0) &= x \\ (x :: xs)(n+1) &= xs(n) \\[4pt] (\_ \$ \_) &: (\forall n \to \forall x \to Bs(n)(x)) \to \\ &\qquad (\forall xs \to \forall n \to Bs(n)(xs(n))) \\ (fs \$ xs)(n) &= fs(n)(xs(n)) \\[4pt] \mathsf{indn} &: (\forall n \to As(n) \to As(n+1)) \to \\ &\qquad As(0) \to [As] \\ \mathsf{indn}(fs)(x)(0) &= x \\ \mathsf{indn}(fs)(x)(n+1) &= fs(n)(\mathsf{indn}(fs)(x)(n))\end{aligned}$$

Fig. 2. Heterogeneous FRP

synonym for $\mathbb{N} \to A$. Of particular instance is the case where A is $\star$, since $\star^{\omega}$ is the type of streams of types, which are the functional reactive types we are interested in.

We define the constant stream $\langle k \rangle$, equal to $(k :: k :: k :: \cdots)$, which has type A^{ω} whenever k has type A. For example $\langle 1 \rangle$ has type $\mathbb{N}^{\omega}$, $\langle \_ + \_ \rangle$ has type $(\mathbb{N} \to \mathbb{N} \to \mathbb{N})^{\omega}$, $\langle \mathbb{N} \rangle$ has type $\star^{\omega}$, and $\langle \_ \times \_ \rangle$ has type $(\star \to \star \to \star)^{\omega}$.

We now turn our attention to heterogeneous streams, in Figure 2. The type $[As]$ (where $As$ is a stream of types, that is its type is $\star^{\omega}$) is inhabited by heterogeneous streams xs where each $xs(i)$ has type $As(i)$. In particular, note that:

$$A^{\omega} \equiv [\langle A \rangle]$$

That is, homogeneous streams are an instance of heterogeneous streams. In particular, the constant stream function can be given the type:

$$\langle \_ \rangle : A \to [\langle A \rangle]$$

This typing is well-founded because the type of $\langle \_ \rangle$ is defined to be $A \to A^{\omega}$ (which is well-founded) which is definitionally equivalent to $A \to [\langle A \rangle]$. We define $\langle \_ \rangle$ using homogeneous streams, as we could not define $\langle \_ \rangle$ to have type $A \to [\langle A \rangle]$ directly, since this would use the definition of $\langle \_ \rangle$ in its own type, which is not well-founded.

$$\begin{aligned}(\_ \wedge \_) &: \star^{\omega} \to \star^{\omega} \to \star^{\omega} \\ (As \wedge Bs) &= \langle \_ \times \_ \rangle \$ As \$ Bs \\[4pt] (\_ \vee \_) &: \star^{\omega} \to \star^{\omega} \to \star^{\omega} \\ (As \vee Bs) &= \langle \_ \uplus \_ \rangle \$ As \$ Bs \\[4pt] (\_ \Rightarrow \_) &: \star^{\omega} \to \star^{\omega} \to \star^{\omega} \\ (As \Rightarrow Bs) &= \langle \_ \to \_ \rangle \$ As \$ Bs \\[4pt] (\_ \Leftarrow \_) &: \star^{\omega} \to \star^{\omega} \to \star^{\omega} \\ (As \Leftarrow Bs) &= (Bs \Rightarrow As) \\[4pt] (\_ \Leftrightarrow \_) &: \star^{\omega} \to \star^{\omega} \to \star^{\omega} \\ (As \Leftrightarrow Bs) &= (As \Leftarrow Bs) \wedge (As \Rightarrow Bs) \\[4pt] (\_ \Rightarrow^{d} \_) &: (\forall (As : \star^{\omega}) \to [As \Rightarrow \langle \star \rangle] \to \star^{\omega}) \\ (As \Rightarrow^{d} Bs) &= \langle \_ \to^{d} \_ \rangle \$ As \$ Bs\end{aligned}$$

Fig. 3. Logical connectives as functional reactive types

We define the "head" function (!), "tail" function ($\bullet$) and "cons" function ($\_ :: \_$) which satisfy:

$$\begin{aligned}!(x :: xs) &\equiv x \\ \bullet(x :: xs) &\equiv xs \\ (!xs :: \bullet xs) &\equiv xs \\ (x :: \langle x \rangle) &\equiv \langle x \rangle\end{aligned}$$

The derived types for these functions are:

$$\begin{aligned}! &: [As] \to (!As) \\ \bullet &: [As] \to [\bullet As] \\ (\_ :: \_) &: A \to [As] \to [A :: As]\end{aligned}$$

In particular, for homogeneous streams, these functions have their expected type:

$$\begin{aligned}! &: A^{\omega} \to A \\ \bullet &: A^{\omega} \to A^{\omega} \\ (\_ :: \_) &: A \to A^{\omega} \to A^{\omega}\end{aligned}$$

The "apply" function ($\_\$\_$) applies a stream of functions pointwise to a stream of arguments to get a stream of results:

$$(f :: fs) \$ (x :: xs) \equiv (f(x) :: (fs \$ xs))$$

The type for "apply" is not particularly readable, but we can provide a derived type which is much more familiar. First, we take a detour into deriving the pointwise logical connectives $\wedge$, $\Rightarrow$ and so on, in Figure 3. These are all defined pointwise, for example:

$$[As \Rightarrow Bs] \equiv (\forall n \to As(n) \to Bs(n))$$

In particular, the dependent function space on streams has:

$$[As \Rightarrow^{d} Bs] \equiv (\forall n \to \forall x \to Bs(n)(x))$$

Thus, the "apply" function can be given the derived type:

$$(\_ \$ \_) : [As \Rightarrow^{d} Bs] \to (\forall xs \to [Bs \$ xs])$$

In particular, the independent function space is an instance of the dependent function space:

$$(As \Rightarrow Bs) \equiv (As \Rightarrow^{d} (\lambda n \to \lambda x \to Bs(n)))$$

$$\begin{aligned}
\langle\_\rangle &: A \to [\langle A\rangle] \\
! &: [As] \to (!As) \\
\bullet &: [As] \to [\bullet As] \\
(\_ :: \_) &: A \to [As] \to [A :: As] \\
(\_ \$ \_) &: [As \Rightarrow^d Bs] \to (\forall xs \to [Bs \$ xs]) \\
(\_ \$ \_) &: [As \Rightarrow Bs] \to [As] \to [Bs] \\
\mathsf{indn} &: [As \Rightarrow \bullet As] \to (!As) \to [As]
\end{aligned}$$

Fig. 4. Functional reactive types for FRP

and so "apply" has a specialized type for independent functions, which is the familiar "modus ponens" elimination rule for implication:

$$(\_ \$ \_) : [As \Rightarrow Bs] \to [As] \to [Bs]$$

Since:

$$(\langle A\rangle \Rightarrow \langle B\rangle) \equiv \langle A \to B\rangle$$

we get for homogeneous streams:

$$(\_ \$ \_) : (A \to B)^\omega \to A^\omega \to B^\omega$$

which, together with $\langle\_\rangle$, gives the structure of an applicative functor [18] to homogeneous streams.

In Figure 4, we restate the types for FRP, using functional reactive types.

The last combinator we consider is the induction rule for natural numbers. Its derived type is an induction rule for LTL:

$$\mathsf{indn} : [As \Rightarrow \bullet As] \to (!As) \to [As]$$

In particular, for homogeneous streams, indn is an indexed iterator:

$$\mathsf{indn} : (\mathbb{N} \to A \to A) \to A \to A^\omega$$

from which we can define the usual stream iterator:

$$\begin{aligned}
\mathsf{iterate} &: (A \to A) \to A \to A^\omega \\
\mathsf{iterate}(f) &= \mathsf{indn}\langle f\rangle
\end{aligned}$$

For example, we can define a clock which returns the current time as:

$$\begin{aligned}
\mathsf{now} &: \mathbb{N}^\omega \\
\mathsf{now} &= \mathsf{iterate}(\_ + 1)(0) \\
\mathsf{now}(n) &\equiv n
\end{aligned}$$

Induction can be used to define functions such as a "running total" function, defined in Figure 5, satisfying:

$$\begin{aligned}
!(\mathsf{sum}(xs)) &\equiv !xs \\
\bullet(\mathsf{sum}(xs)) &\equiv \bullet xs \ \mathsf{plus}\ \mathsf{sum}(xs)
\end{aligned}$$

For example:

$$\mathsf{sum}(3 :: 2 :: 1 :: \langle 0\rangle) \equiv (3 :: 5 :: 6 :: \langle 6\rangle)$$

The same technique is used in Figure 6 to define the temporal connectives from past-time LTL (pLTL) as functional reactive types. For example, the $\Box$ modality acts as an iterated product:

$$\begin{aligned}
!(\Box As) &\equiv !As \\
\bullet(\Box As) &\equiv (\bullet As) \wedge (\Box As)
\end{aligned}$$

$$\begin{aligned}
\mathsf{scan} &: [As \Rightarrow Bs \Rightarrow \bullet Bs] \to \\
&\quad (!Bs) \to [As] \to [Bs] \\
\mathsf{scan}(fs)(y)(xs) &= \mathsf{indn}(fs \$ xs)(y) \\[4pt]
\mathsf{scan}_1 &: [\bullet As \Rightarrow As \Rightarrow \bullet As] \to \\
&\quad [As] \to [As] \\
\mathsf{scan}_1(fs)(xs) &= \mathsf{scan}(fs)(!xs)(\bullet xs) \\[4pt]
\mathsf{scan}_2 &: [\bullet Bs \Rightarrow \bullet As \Rightarrow Bs \Rightarrow \bullet Bs] \to \\
&\quad [As] \to [Bs] \to [Bs] \\
\mathsf{scan}_2(fs)(xs)(ys) &= \mathsf{scan}(fs \$ \bullet ys)(!ys)(\bullet xs) \\[4pt]
(\_ \ \mathsf{plus}\ \_) &: \mathbb{N}^\omega \to \mathbb{N}^\omega \to \mathbb{N}^\omega \\
xs \ \mathsf{plus}\ ys &= \langle \_ + \_\rangle \$ xs \$ ys \\[4pt]
\mathsf{sum} &: \mathbb{N}^\omega \to \mathbb{N}^\omega \\
\mathsf{sum} &= \mathsf{scan}_1\langle \_ + \_\rangle
\end{aligned}$$

Fig. 5. Scan functions

$$\begin{aligned}
\bigcirc &: \star^\omega \to \star^\omega \\
\bigcirc As &= \top :: As \\[4pt]
\Box &: \star^\omega \to \star^\omega \\
\Box &= \mathsf{scan}_1\langle \_ \times \_\rangle \\[4pt]
\Diamond &: \star^\omega \to \star^\omega \\
\Diamond &= \mathsf{scan}_1\langle \_ \uplus \_\rangle \\[4pt]
(\_ \ \mathsf{S}\ \_) &: \star^\omega \to \star^\omega \to \star^\omega \\
(\_ \ \mathsf{S}\ \_) &= \mathsf{scan}_2\langle \_ \uplus \_ \times \_\rangle \\[4pt]
(\_ \rhd \_) &: \star^\omega \to \star^\omega \to \star^\omega \\
(\_ \rhd \_) &= \mathsf{scan}_2\langle \_ \times \_ \to \_\rangle
\end{aligned}$$

Fig. 6. Temporal connectives as functional reactive types

so, for instance:

$$\Box As(2) \equiv As(2) \times As(1) \times As(0)$$

and the S modality acts as an iterated nested product and coproduct:

$$\begin{aligned}
!(As \ \mathsf{S}\ Bs) &\equiv !Bs \\
\bullet(As \ \mathsf{S}\ Bs) &\equiv (\bullet Bs) \vee ((\bullet As) \wedge (As \ \mathsf{S}\ Bs))
\end{aligned}$$

so, for instance:

$$(As \ \mathsf{S}\ Bs)(2) \equiv Bs(2) \uplus As(2) \times (Bs(1) \uplus As(1) \times Bs(0))$$

We have now defined the logical connectives of pLTL as functional reactive types, and now look at how proof rules for pLTL can be encoded as functional reactive programs. In particular, we will show that FRP forms a category whose objects are functional reactive types, and whose morphisms are programs of type $[As \Rightarrow Bs]$, where $\wedge$ is product, $\vee$ is coproduct, $\Box$ is a comonad, and $\Diamond$ is a monad (and so form a model of constructive S4 modal logic [2]). The novelty here (compared to the author's previous work [11]) is that all the proof rules are defined just using the combinators in Figures 1 and 2, thus showing that it is sufficient to present streams as

$$\begin{aligned}
&(\mathsf{const}_1(k) : [\langle F\rangle \mathbin{\$} As]) \text{ where} \\
&\qquad (\forall A \to k : F(A)) \\
&\mathsf{const}_1(k) = \langle(\lambda A \to k)\rangle \mathbin{\$} As \\
&(\mathsf{const}_2(k) : [\langle F\rangle \mathbin{\$} As \mathbin{\$} Bs]) \text{ where} \\
&\qquad (\forall AB \to k : F(A)(B)) \\
&\mathsf{const}_2(k) = \langle(\lambda A, B \to k)\rangle \mathbin{\$} As \mathbin{\$} Bs \\
&(\mathsf{const}_3(k) : [\langle F\rangle \mathbin{\$} As \mathbin{\$} Bs \mathbin{\$} Cs]) \text{ where} \\
&\qquad (\forall ABC \to k : F(A)(B)(C)) \\
&\mathsf{const}_3(k) = \langle(\lambda A, B, C \to k)\rangle \mathbin{\$} As \mathbin{\$} Bs \mathbin{\$} Cs \\
&(\mathsf{const}_4(k) : [\langle F\rangle \mathbin{\$} As \mathbin{\$} Bs \mathbin{\$} Cs \mathbin{\$} Ds]) \text{ where} \\
&\qquad (\forall ABCD \to k : F(A)(B)(C)(D)) \\
&\mathsf{const}_4(k) = \langle(\lambda A, B, C, D \to k)\rangle \mathbin{\$} As \mathbin{\$} Bs \mathbin{\$} Cs \mathbin{\$} Ds
\end{aligned}$$

Fig. 7. Polymorphic constants

$$\begin{aligned}
\mathsf{ids} &: [As \Rightarrow As] \\
\mathsf{ids} &= \mathsf{const}_1(\mathsf{id}) \\
(\_ \cdot \_) &: [Bs \Rightarrow Cs] \to [As \Rightarrow Bs] \to [As \Rightarrow Cs] \\
(fs \cdot gs) &= \mathsf{const}_3(\_ \circ \_) \mathbin{\$} fs \mathbin{\$} gs \\
(fs \cdot \mathsf{ids}) &\equiv fs \\
(\mathsf{ids} \cdot fs) &\equiv fs \\
((fs \cdot gs) \cdot hs) &\equiv (fs \cdot (gs \cdot hs))
\end{aligned}$$

Fig. 8. Categorical structure of streams

an applicative functor with induction to deduce the categorical structure.

Before we can do this, however, we need to take a look at polymorphic constants such as the identity function. We are interested in finding an identity on streams with type:

$$\mathsf{ids} : [As \Rightarrow As]$$

The obvious definition is $\langle \mathsf{id} \rangle$, but we can only use this definition on homogeneous streams:

$$\langle \mathsf{id} \rangle : [\langle A \rangle \Rightarrow \langle A \rangle]$$

It cannot be given the more general type $[As \Rightarrow As]$ as the type id is being used at is $A(n) \to A(n)$, which depends on the time n. The constant stream $\langle k \rangle$ can only be used to construct homogeneous streams whose value and type do not depend on the time n. The same problem impacts all of the functions over heterogeneous streams such as function composition, projections and injections. We need a way to allow polymorphic constants whose type depends on the time n. We do this in Figure 7 where we define a polymorphic constant $\mathsf{const}_1(k)$ such that:

$$\mathsf{const}_1(k)(n) \equiv k$$

The difference between const_1 and $\langle \_ \rangle$ is its type, since $\mathsf{const}_1(k)$ allows k to be polymorphic and vary its type with time, whereas $\langle k \rangle$ requires k to be constant in its type.

$$(\mathsf{const}_1(k) : [\langle F\rangle \mathbin{\$} As]) \text{ where } (\forall A \to k : F(A))$$

$$\begin{aligned}
\mathsf{fsts} &: [(As \wedge Bs) \Rightarrow As] \\
\mathsf{fsts} &= \mathsf{const}_2(\mathsf{fst}) \\
\mathsf{snds} &: [(As \wedge Bs) \Rightarrow Bs] \\
\mathsf{snds} &= \mathsf{const}_2(\mathsf{snd}) \\
\mathsf{boths} &: [(As \Rightarrow Bs) \Rightarrow (As \Rightarrow Cs) \Rightarrow \\
&\qquad As \Rightarrow (Bs \wedge Cs)] \\
\mathsf{boths} &= \mathsf{const}_3(\mathsf{both}) \\
\mathsf{map}^{\wedge} &: [(As \Rightarrow Bs) \Rightarrow (Cs \Rightarrow Ds) \Rightarrow \\
&\qquad ((As \wedge Cs) \Rightarrow (Bs \wedge Ds))] \\
\mathsf{map}^{\wedge} &= \mathsf{const}_4(\mathsf{map}^{\times}) \\
fs &\equiv (\mathsf{fsts} \cdot (\mathsf{boths} \mathbin{\$} fs \mathbin{\$} gs)) \\
gs &\equiv (\mathsf{snds} \cdot (\mathsf{boths} \mathbin{\$} fs \mathbin{\$} gs)) \\
hs &\equiv (\mathsf{boths} \mathbin{\$} (\mathsf{fsts} \cdot hs) \mathbin{\$} (\mathsf{snds} \cdot hs)) \\
\mathsf{map}^{\wedge} \mathbin{\$} fs \mathbin{\$} gs &\equiv \mathsf{boths} \mathbin{\$} (fs \cdot \mathsf{fsts}) \mathbin{\$} (gs \cdot \mathsf{snds})
\end{aligned}$$

Fig. 9. Product structure of streams

$$\begin{aligned}
\mathsf{inls} &: [As \Rightarrow (As \vee Bs)] \\
\mathsf{inls} &= \mathsf{const}_2(\mathsf{inl}) \\
\mathsf{inrs} &: [Bs \Rightarrow (As \vee Bs)] \\
\mathsf{inrs} &= \mathsf{const}_2(\mathsf{inr}) \\
\mathsf{cases} &: [(As \Rightarrow Cs) \Rightarrow (Bs \Rightarrow Cs) \Rightarrow \\
&\qquad (As \vee Bs) \Rightarrow Cs] \\
\mathsf{cases} &= \mathsf{const}_3(\mathsf{case}) \\
\mathsf{map}^{\vee} &: [(As \Rightarrow Bs) \Rightarrow (Cs \Rightarrow Ds) \Rightarrow \\
&\qquad ((As \vee Cs) \Rightarrow (Bs \vee Ds))] \\
\mathsf{map}^{\vee} &= \mathsf{const}_4(\mathsf{map}^{\uplus}) \\
fs &\equiv ((\mathsf{cases} \mathbin{\$} fs \mathbin{\$} gs) \cdot \mathsf{inls}) \\
gs &\equiv ((\mathsf{cases} \mathbin{\$} fs \mathbin{\$} gs) \cdot \mathsf{inrs}) \\
hs &\equiv (\mathsf{cases} \mathbin{\$} (hs \cdot \mathsf{inls}) \mathbin{\$} (hs \cdot \mathsf{inrs})) \\
\mathsf{map}^{\vee} \mathbin{\$} fs \mathbin{\$} gs &\equiv \mathsf{cases} \mathbin{\$} (\mathsf{inls} \cdot fs) \mathbin{\$} (\mathsf{inrs} \cdot gs)
\end{aligned}$$

Fig. 10. Coproduct structure of streams

We are allowing k to have polymorphic type $k : F(A)$ for any $F : \star \to \star$, for example if we take k to be id and $F(A)$ to be $A \to A$ then we have:

$$\mathsf{const}_1(\mathsf{id}) : [As \Rightarrow As]$$

The definition of const_1 is:

$$\mathsf{const}_1(k) \equiv \langle(\lambda A \to k)\rangle \mathbin{\$} As$$

This typechecks because, although $\mathsf{const}_1(k)(n) = k$ has type $F(A_n)$ which depends on n, the function $(\lambda A \to k)$ has type $\forall A \to F(A)$ which does not depend on n. This use of dependent types allows us to define constants that are parametric in time. In Figure 7 we also define const_2, const_3 and so on. For example, taking $F(A)(B)(C)$ to be $(B \to C) \to (A \to B) \to (A \to C)$, composition of stream functions can be defined:

$$\begin{aligned}
(\_ \cdot \_) &: [Bs \Rightarrow Cs] \to [As \Rightarrow Bs] \to [As \Rightarrow Cs] \\
(fs \cdot gs) &\equiv \mathsf{const}_3(\_ \circ \_) \mathbin{\$} fs \mathbin{\$} gs
\end{aligned}$$

$$\begin{aligned}
\mathsf{map}^{\Box} &: [As \Rightarrow Bs] \to [\Box As \Rightarrow \Box Bs]\\
\mathsf{map}^{\Box}(fs) &= \mathsf{scan}(\mathsf{map}^{\wedge})(!fs)(\bullet fs)\\
\mathsf{extract} &: [\Box As \Rightarrow As]\\
\mathsf{extract} &= \mathsf{id} :: \mathsf{fsts}\\
\mathsf{duplicate} &: [\Box As \Rightarrow \Box(\Box As)]\\
\mathsf{duplicate} &= \mathsf{indn}(\mathsf{const}_3(\lambda g \to\\
&\qquad \mathsf{both}(\mathsf{id})(g \circ \mathsf{snd})))(\mathsf{id})\\
(\mathsf{extract} \cdot \mathsf{duplicate}) &\equiv \mathsf{ids}\\
(\mathsf{map}^{\Box}(\mathsf{extract}) \cdot \mathsf{duplicate}) &\equiv \mathsf{ids}\\
(\mathsf{duplicate} \cdot \mathsf{duplicate}) &\equiv (\mathsf{map}^{\Box}(\mathsf{duplicate}) \cdot \mathsf{duplicate})
\end{aligned}$$

Fig. 11. Comonadic structure of streams

$$\begin{aligned}
\mathsf{map}^{\Diamond} &: [As \Rightarrow Bs] \to [\Diamond As \Rightarrow \Diamond Bs]\\
\mathsf{map}^{\Diamond}(fs) &= \mathsf{scan}(\mathsf{map}^{\vee})(!fs)(\bullet fs)\\
\mathsf{return} &: [As \Rightarrow \Diamond As]\\
\mathsf{return} &= \mathsf{id} :: \mathsf{inls}\\
\mathsf{join} &: [\Diamond(\Diamond As) \Rightarrow \Diamond As]\\
\mathsf{join} &= \mathsf{indn}(\mathsf{const}_3(\lambda g \to\\
&\qquad \mathsf{case}(\mathsf{id})(\mathsf{inr} \circ g)))(\mathsf{id})\\
(\mathsf{join} \cdot \mathsf{return}) &\equiv \mathsf{ids}\\
(\mathsf{join} \cdot \mathsf{map}^{\Diamond}(\mathsf{return})) &\equiv \mathsf{ids}\\
(\mathsf{join} \cdot \mathsf{join}) &\equiv (\mathsf{join} \cdot \mathsf{map}^{\Diamond}(\mathsf{join}))
\end{aligned}$$

Fig. 12. Monadic structure of streams

So we have:

$$(fs \cdot gs)(n) \equiv fs(n) \circ gs(n)$$

Proposition 1: Heterogeneous streams form a category whose objects are elements of $\star^{\omega}$, and whose morphisms are elements of $[As \Rightarrow Bs]$. This category has products given by $As \wedge Bs$, coproducts given by $As \vee Bs$, a comonad given by $\Box As$, and a monad given by $\Diamond As$.

Proof: The constructions are given in Figures 8–12. ■

IV. pLTL AS FUNCTIONAL REACTIVE TYPES

In the previous section, we claimed that the logical connectives in Figure 3 and the temporal modalities introduced in Figure 6 corresponded to past-time LTL (pLTL). In this section, we shall make this correspondence precise.

The variant on pLTL we will use is given in Figure 13. We define a logic $\mathsf{pLTL}(\Sigma)$ which is pLTL over words drawn from an alphabet Σ. The only difference between this version of pLTL and the usual presentation is the modality (ϕconstrain ψ), which we are using as the dual of (ϕ since ψ): classically $\neg(\phi\,\mathsf{constrain}\,\psi)$ is $(\phi\,\mathsf{since}\,\neg\psi)$. We are using constrain rather than the more usual release modality (where (ϕ release ψ) is $\neg(\neg\phi\,\mathsf{since}\,\neg\psi)$) because, as we shall see below, constrain corresponds constructively to a function space, whereas it is not obvious what the constructive interpretation of release should be. The constrain modality was introduced (for future-time LTL) by McMillan [19] and further investigated by Namjoshi and Trefler [20], and was used in giving a characterization of rely-guarantee reasoning for cyclic systems in LTL.

$$\mathsf{pLTL} : \star \to \star$$

$$\begin{aligned}
\mathsf{true} &: \mathsf{pLTL}(\Sigma)\\
\mathsf{false} &: \mathsf{pLTL}(\Sigma)\\
\_\,\mathsf{and}\,\_ &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\_\,\mathsf{or}\,\_ &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\mathsf{box} &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\mathsf{dia} &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\_\,\mathsf{since}\,\_ &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\_\,\mathsf{constrain}\,\_ &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\mathsf{atom} &: \mathbb{P}(\Sigma) \to \mathsf{pLTL}(\Sigma)
\end{aligned}$$

Fig. 13. Syntax of pLTL

The semantics of pLTL is given in Figure 14. Here, $[\![\phi]\!](n)$ is the set of words drawn from Σ^{ω} which satisfy property ϕ at position n. We have presented pLTL in negation normal form, but we can define negation as:

$$\begin{aligned}
\mathsf{not} &: \mathsf{pLTL}(\Sigma) \to \mathsf{pLTL}(\Sigma)\\
\mathsf{not}(\mathsf{true}) &= \mathsf{false}\\
\mathsf{not}(\mathsf{false}) &= \mathsf{true}\\
\mathsf{not}(\phi\,\mathsf{and}\,\psi) &= \mathsf{not}(\phi)\,\mathsf{or}\,\mathsf{not}(\psi)\\
\mathsf{not}(\phi\,\mathsf{or}\,\psi) &= \mathsf{not}(\phi)\,\mathsf{and}\,\mathsf{not}(\psi)\\
\mathsf{not}(\mathsf{box}(\phi)) &= \mathsf{dia}(\mathsf{not}(\phi))\\
\mathsf{not}(\mathsf{dia}(\phi)) &= \mathsf{box}(\mathsf{not}(\phi))\\
\mathsf{not}(\phi\,\mathsf{since}\,\psi) &= \phi\,\mathsf{constrain}\,\mathsf{not}(\psi)\\
\mathsf{not}(\phi\,\mathsf{constrain}\,\psi) &= \phi\,\mathsf{since}\,\mathsf{not}(\psi)\\
\mathsf{not}(\mathsf{atom}(S)) &= \mathsf{atom}(S^{\mathsf{C}})
\end{aligned}$$

We can then show that this has the expected semantics (where S^{C} is the complement of S, defined to be $\Sigma^{\omega} \setminus S$):

$$\begin{aligned}
[\![\mathsf{not}(\phi)]\!](n) &\subseteq [\![\phi]\!](n)^{\mathsf{C}}\\
[\![\phi]\!](n)^{\mathsf{C}} &\subseteq^{*} [\![\mathsf{not}(\phi)]\!](n)
\end{aligned}$$

Here, $S \subseteq^{*} T$ means classical set inclusion, that is we use excluded middle in showing $S \subseteq T$. This is the only use of excluded middle in this paper.

We now show that the semantics of pLTL can be encoded as functional reactive types. This is not completely trivial as, for example, the semantics of box is defined using universal quantification rather than iterated product. To bridge this gap, we introduce alternative characterizations of the pLTL modalities as functional reactive types, which is closer to the semantics of pLTL.

We show that this alternative characterization of pLTL is isomorphic to the original, where isomorphism $As \approx Bs$ is given by a pair of stream functions:

$$\begin{aligned}
fs &: [As \Rightarrow Bs] & gs &: [Bs \Rightarrow As]\\
(fs \cdot gs) &\equiv \mathsf{ids} & (gs \cdot fs) &\equiv \mathsf{ids}
\end{aligned}$$

$$
\begin{aligned}
[\![\_]\!] &: \mathsf{pLTL}(\Sigma) \to \mathbb{N} \to \mathbb{P}(\Sigma^\omega) \\
[\![\mathsf{true}]\!](n) &= \Sigma^\omega \\
[\![\mathsf{false}]\!](n) &= \emptyset \\
[\![\phi \ \mathsf{and}\ \psi]\!](n) &= [\![\phi]\!](n) \cap [\![\psi]\!](n) \\
[\![\phi \ \mathsf{or}\ \psi]\!](n) &= [\![\phi]\!](n) \cup [\![\psi]\!](n) \\
[\![\mathsf{box}(\phi)]\!](n) &= \{\, w \mid (\forall m \to (m \le n) \to (w \in [\![\phi]\!](m)))\,\} \\
[\![\mathsf{dia}(\phi)]\!](n) &= \{\, w \mid (\exists m \to (m \le n) \times (w \in [\![\phi]\!](m)))\,\} \\
[\![\phi \ \mathsf{since}\ \psi]\!](n) &= \{\, w \mid (\exists \ell \to (\ell \le n) \times (\forall m \to (\ell < m) \to (m \le n) \to (w \in [\![\phi]\!](m))) \times (w \in [\![\psi]\!](\ell)))\,\} \\
[\![\phi \ \mathsf{constrain}\ \psi]\!](n) &= \{\, w \mid (\forall \ell \to (\ell \le n) \to (\forall m \to (\ell < m) \to (m \le n) \to (w \in [\![\phi]\!](m))) \to (w \in [\![\psi]\!](\ell)))\,\} \\
[\![\mathsf{atom}(S)]\!](n) &= \{\, w \mid (w(n) \in S)\,\}
\end{aligned}
$$

Fig. 14. Semantics of pLTL

$$
\begin{aligned}
\_\langle\_,\_] &: \star^\omega \to \mathbb{N} \to \mathbb{N} \to \star \\
As\langle \ell, n] &= (\forall m \to (\ell < m) \to (m \le n) \to As(m)) \\[1ex]
\square' &: \star^\omega \to \star^\omega \\
(\square' As)(n) &= (\forall m \to (m \le n) \to As(m)) \\[1ex]
\lozenge' &: \star^\omega \to \star^\omega \\
(\lozenge' As)(n) &= (\exists m \to (m \le n) \times As(m)) \\[1ex]
\_\,\mathsf{S}'\,\_ &: \star^\omega \to \star^\omega \to \star^\omega \\
(As\ \mathsf{S}'\ Bs)(n) &= (\exists \ell \to (\ell \le n) \times (As\langle \ell, n]) \times Bs(\ell)) \\[1ex]
\_\rhd'\_ &: \star^\omega \to \star^\omega \to \star^\omega \\
(As \rhd' Bs)(n) &= (\forall \ell \to (\ell \le n) \to As\langle \ell, n] \to Bs(\ell))
\end{aligned}
$$

$$
\begin{aligned}
\square As &\approx \square' As \\
\lozenge As &\approx \lozenge' As \\
As\ \mathsf{S}\ Bs &\approx As\ \mathsf{S}'\ Bs \\
As \rhd Bs &\approx As \rhd' Bs
\end{aligned}
$$

Fig. 15. Alternative characterization of pLTL as functional reactive types

The semantics of since and constrain are defined in terms of intervals, for example (ϕ since ψ) is true at time n if there is some $\ell \le n$ such that ψ is true at time ℓ and ϕ is true in the half-open interval $\langle \ell, n]$. For this reason, we introduce the interval type $As\langle \ell, n]$ inhabited by streams xs such that $xs(m)$ has type $As(m)$ for any $\ell < m \le n$. We use $As\langle \ell, n]$ in defining the alternate versions of $As\ \mathsf{S}\ Bs$ and $As \rhd Bs$, for example $As\ \mathsf{S}\ Bs$ is inhabited at time n whenever there is some $\ell \le n$ such that $As\langle \ell, n]$ and $Bs(\ell)$ are inhabited. This is formalized in Figure 15, including the isomorphisms between the previous presentation of pLTL and the alternative presentation (the proofs of these isomorphisms total about 200 lines of Agda).

Having provided the alternative definitions of pLTL modalities as functional reactive types, it is routine to translate pLTL into $\star^\omega$, in Figure 16. The formula ϕ from $\mathsf{pLTL}(\Sigma)$ is given an interpretation $\langle\!\langle \phi \rangle\!\rangle(w)$, where w is a word in Σ^ω. It is a direct induction to show that $w \in [\![\phi]\!](n)$ precisely when $\langle\!\langle \phi \rangle\!\rangle(w)(n)$ is inhabited.

Proposition 2:

$$(w \in [\![\phi]\!](n)) \equiv \langle\!\langle \phi \rangle\!\rangle(w)(n)$$

$$
\begin{aligned}
\langle\!\langle \_ \rangle\!\rangle &: \mathsf{pLTL}(\Sigma) \to \Sigma^\omega \to \star^\omega \\
\langle\!\langle \mathsf{true} \rangle\!\rangle(w) &= \langle \top \rangle \\
\langle\!\langle \mathsf{false} \rangle\!\rangle(w) &= \langle \bot \rangle \\
\langle\!\langle \phi \ \mathsf{and}\ \psi \rangle\!\rangle(w) &= \langle\!\langle \phi \rangle\!\rangle(w) \wedge \langle\!\langle \psi \rangle\!\rangle(w) \\
\langle\!\langle \phi \ \mathsf{or}\ \psi \rangle\!\rangle(w) &= \langle\!\langle \phi \rangle\!\rangle(w) \vee \langle\!\langle \psi \rangle\!\rangle(w) \\
\langle\!\langle \mathsf{box}(\phi) \rangle\!\rangle(w) &= \square'(\langle\!\langle \phi \rangle\!\rangle(w)) \\
\langle\!\langle \mathsf{dia}(\phi) \rangle\!\rangle(w) &= \lozenge'(\langle\!\langle \phi \rangle\!\rangle(w)) \\
\langle\!\langle \phi \ \mathsf{since}\ \psi \rangle\!\rangle(w) &= \langle\!\langle \phi \rangle\!\rangle(w)\ \mathsf{S}'\ \langle\!\langle \psi \rangle\!\rangle(w) \\
\langle\!\langle \phi \ \mathsf{constrain}\ \psi \rangle\!\rangle(w) &= \langle\!\langle \phi \rangle\!\rangle(w) \rhd' \langle\!\langle \psi \rangle\!\rangle(w) \\
\langle\!\langle \mathsf{atom}(S) \rangle\!\rangle(w) &= \langle S \rangle\ \$\ w
\end{aligned}
$$

Fig. 16. Translation of pLTL into functional reactive types

Proof: An induction on ϕ. ■

V. pLTL+FRP BOUNDED SATISFIABILITY

We now give an application of the use of functional reactive programs and functional reactive types. We show how FRP can be used to encode pLTL formulae as streams of boolean expressions, such that the boolean expression is true at time k precisely when the functional reactive type is inhabited at time k, and hence precisely when the pLTL formula is true at time k. This gives a simple algorithm for bounded satisfiability of pLTL: to check if a formula ϕ is satisfiable at time k, encode it as a stream of boolean expressions Es, and then check satisfiability of $Es(k)$.

In the case of pLTL, the use of SAT-solvers to encode bounded satisfiability is well-known [3]. What is new here is that FRP is being used to encode pLTL. This means that FRP expressions can be used in pLTL formulae, and so we have a strictly stronger logic in which one can express properties such as "the total sum of xs plus ys is always equal to 0," which cannot be expressed in pLTL due to the lack of data values. By encoding pLTL+FRP as streams of expressions, we can check satisfiability of pLTL+FRP formulae, by using a Satisfiability Modulo Theory (SMT) checker [5] to find a satisfying assignment for $Es(k)$.

The syntax of the expression language is given in Figure 17. For simplicity, we are just considering a theory with natural numbers, addition and equality, but this approach should apply

$$\begin{aligned}\mathsf{Sort} &: \star \\ \mathsf{Exp} &: (\mathsf{Sort} \to \star) \to \mathsf{Sort} \to \star\end{aligned}$$

$$\begin{aligned}\mathsf{bool} &: \mathsf{Sort} \\ \mathsf{nat} &: \mathsf{Sort}\end{aligned}$$

$$\begin{aligned}
\mathsf{true} &: \mathsf{Exp}(V)(\mathsf{bool}) \\
\mathsf{false} &: \mathsf{Exp}(V)(\mathsf{bool}) \\
\mathsf{const} &: \mathbb{N} \to \mathsf{Exp}(V)(\mathsf{nat}) \\
\_\,\mathsf{and}\,\_ &: \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \\
\_\,\mathsf{or}\,\_ &: \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \\
\_\,\mathsf{impl}\,\_ &: \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \to \mathsf{Exp}(V)(\mathsf{bool}) \\
\_\,\mathsf{add}\,\_ &: \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{nat}) \\
\_\,\mathsf{eq}\,\_ &: \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{bool}) \\
\_\,\mathsf{ne}\,\_ &: \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{nat}) \to \mathsf{Exp}(V)(\mathsf{bool}) \\
\mathsf{var} &: V(S) \to \mathsf{Exp}(V)(S)
\end{aligned}$$

Fig. 17. Syntax of expression language

$$\begin{aligned}
s[\![\_]\!] &: \mathsf{Sort} \to \star \\
s[\![\mathsf{bool}]\!] &= \mathbb{B} \\
s[\![\mathsf{nat}]\!] &= \mathbb{N}
\end{aligned}$$

$$\begin{aligned}
v[\![\_]\!] &: (\mathsf{Sort} \to \star) \to \star \\
v[\![V]\!] &= (\forall S \to V(S) \to s[\![S]\!])
\end{aligned}$$

$$\begin{aligned}
e[\![\_]\!] &: \mathsf{Exp}(V)(S) \to v[\![V]\!] \to s[\![S]\!] \\
e[\![\mathsf{var}(x)]\!](\rho) &= \rho(S)(x) \\
e[\![\mathsf{true}]\!](\rho) &= \mathsf{true} \\
e[\![\mathsf{false}]\!](\rho) &= \mathsf{false} \\
e[\![\mathsf{const}(n)]\!](\rho) &= n \\
e[\![E \mathbin{\mathsf{and}} F]\!](\rho) &= e[\![E]\!](\rho) \,\&\, e[\![F]\!](\rho) \\
e[\![E \mathbin{\mathsf{or}} F]\!](\rho) &= e[\![E]\!](\rho) \mid e[\![F]\!](\rho) \\
e[\![E \mathbin{\mathsf{impl}} F]\!](\rho) &= e[\![E]\!](\rho) \supset e[\![F]\!](\rho) \\
e[\![E \mathbin{\mathsf{add}} F]\!](\rho) &= e[\![E]\!](\rho) + e[\![F]\!](\rho) \\
e[\![E \mathbin{\mathsf{eq}} F]\!](\rho) &= e[\![E]\!](\rho) = e[\![F]\!](\rho) \\
e[\![E \mathbin{\mathsf{ne}} F]\!](\rho) &= e[\![E]\!](\rho) \neq e[\![F]\!](\rho)
\end{aligned}$$

Fig. 18. Semantics of expression language

to any theory with an SMT solver. The type of expressions $\mathsf{Exp}(V)(S)$ is parametrized by a set of sorted variables V and a sort S. For example, if we have:

$$\mathsf{x} : \mathsf{Var}(\mathsf{nat}) \quad \mathsf{y} : \mathsf{Var}(\mathsf{nat})$$

then we can construct an expression encoding $x + y = 0$ as:

$$((\mathsf{var}(\mathsf{x}) \mathbin{\mathsf{add}} \mathsf{var}(\mathsf{y})) \mathbin{\mathsf{eq}} \mathsf{const}(0)) : \mathsf{Exp}(\mathsf{Var})(\mathsf{bool})$$

The semantics of the expression language is given in Figure 18, and is given relative to a sort-respecting assignment of values to variables ρ. For example, if:

$$x \equiv \rho(\mathsf{nat})(\mathsf{x}) \quad y \equiv \rho(\mathsf{nat})(\mathsf{y})$$

then:

$$e[\![((\mathsf{var}(\mathsf{x}) \mathbin{\mathsf{add}} \mathsf{var}(\mathsf{y})) \mathbin{\mathsf{eq}} \mathsf{const}(0))]\!](\rho) \equiv ((x + y) = 0)$$

In Figure 19 we lift the syntax of expressions from single expressions to streams of expressions. Variables become timestamped variables, for example, writing:

$$\mathsf{Exps}(V)(S) = \mathsf{Exp}(\mathsf{Timestamped}(V))(S)^{\omega}$$

we have:

$$\mathsf{vars}(\mathsf{x}) : \mathsf{Exps}(\mathsf{Var})(\mathsf{nat})$$

Timestamped variables are of the form (x, k) where x is the name of the variable, and k is its timestamp:

$$\mathsf{vars}(x)(k) \equiv \mathsf{var}(x, k)$$

We can build up streams of arithmetic and boolean expressions, for example the expression "the total value of xs plus ys is 0" can be encoded:

$$\begin{aligned}
\mathsf{example} &: \mathsf{Exps}(\mathsf{Var})(\mathsf{bool}) \\
\mathsf{example} &= ((\mathsf{vars}(\mathsf{x}) \mathbin{\mathsf{adds}} \mathsf{vars}(\mathsf{y})) \mathbin{\mathsf{eqs}} \mathsf{consts}(0))
\end{aligned}$$

and if:

$$x \equiv \rho(\mathsf{nat})(\mathsf{x}, k) \quad y \equiv \rho(\mathsf{nat})(\mathsf{y}, k)$$

then:

$$e[\![\mathsf{example}(k)]\!](\rho) \equiv ((x + y) = 0)$$

In the same way as pLTL modalities are encoded as functions on streams of types, we can encode pLTL modalities as functions on streams of boolean expressions. For example, in the same ways as we defined:

$$\square \equiv \mathsf{scan}_1\langle\_ \times \_\rangle$$

we define:

$$\mathsf{historically} \equiv \mathsf{scan}_1\langle\_\,\mathsf{and}\,\_\rangle$$

For example, if:

$$x \equiv \rho(\mathsf{nat})(\mathsf{x}, 0) \quad y \equiv \rho(\mathsf{nat})(\mathsf{y}, 0)$$

then:

$$e[\![\mathsf{historically}(\mathsf{example})(0)]\!](\rho) \equiv ((x + y) = 0)$$

and if:

$$x' \equiv \rho(\mathsf{nat})(\mathsf{x}, k+1) \quad y' \equiv \rho(\mathsf{nat})(\mathsf{y}, k+1)$$

then:

$$\begin{aligned}
&e[\![\mathsf{historically}(\mathsf{example})(k+1)]\!](\rho) \\
&\equiv ((x' + y') = 0) \,\&\, e[\![\mathsf{historically}(\mathsf{example})(k)]\!](\rho)
\end{aligned}$$

We can now show that satisfaction of a stream of boolean expressions interpreting a pLTL formula is the same as inhabitance of the corresponding functional reactive type. First, we define satisfaction of an expression by an assignment ρ at time k to be when $e[\![Es(n)]\!]$ is true:

$$\begin{aligned}
\checkmark[\![\_]\!] &: \mathsf{Exp}(V)(\mathsf{bool})^{\omega} \to v[\![V]\!] \to \star^{\omega} \\
\checkmark[\![Es]\!](\rho)(k) &= e[\![Es(k)]\!](\rho) \equiv \mathsf{true}
\end{aligned}$$

We can now show that satisfaction of a pLTL formula corresponds precisely to inhabitance of a functional reactive type.

$$
\begin{aligned}
\mathsf{Timestamped} &: (\mathsf{Sort} \to \star) \to (\mathsf{Sort} \to \star) \\
\mathsf{Timestamped}(V)(S) &= V(S) \times \mathbb{N} \\[4pt]
\mathsf{stamped} &: V(S) \to \mathbb{N} \to \mathsf{Timestamped}(V)(S) \\
\mathsf{stamped}(x)(n) &= (x, n) \\[4pt]
\mathsf{vars} &: V(S) \to \mathsf{Exp}(\mathsf{Timestamped}(V))(S)^\omega \\
\mathsf{vars}(x) &= \langle \mathsf{var} \circ \mathsf{stamped}(x) \rangle \ \$\ \mathsf{now} \\[4pt]
\mathsf{consts} &: \mathbb{N} \to \mathsf{Exp}(\mathsf{Timestamped}(V))(\mathsf{nat})^\omega \\
\mathsf{consts}(n) &= \langle \mathsf{const}(n) \rangle \\[4pt]
(\_\ \mathsf{adds}\ \_) &: \mathsf{Exp}(V)(\mathsf{nat})^\omega \to \mathsf{Exp}(V)(\mathsf{nat})^\omega \to \mathsf{Exp}(V)(\mathsf{nat})^\omega \\
(Es\ \mathsf{adds}\ Fs) &= (\langle \_\ \mathsf{add}\ \_ \rangle \ \$\ Es\ \$\ Fs) \\[4pt]
\mathsf{total} &: \mathsf{Exp}(V)(\mathsf{nat})^\omega \to \mathsf{Exp}(V)(\mathsf{nat})^\omega \\
\mathsf{total} &= \mathsf{scan}_1 \langle \_\ \mathsf{add}\ \_ \rangle \\[4pt]
(\_\ \mathsf{eqs}\ \_) &: \mathsf{Exp}(V)(\mathsf{nat})^\omega \to \mathsf{Exp}(V)(\mathsf{nat})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
(Es\ \mathsf{eqs}\ Fs) &= (\langle \_\ \mathsf{eq}\ \_ \rangle \ \$\ Es\ \$\ Fs) \\[4pt]
(\_\ \mathsf{ands}\ \_) &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
(Es\ \mathsf{ands}\ Fs) &= (\langle \_\ \mathsf{and}\ \_ \rangle \ \$\ Es\ \$\ Fs) \\[4pt]
(\_\ \mathsf{ors}\ \_) &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
(Es\ \mathsf{ors}\ Fs) &= (\langle \_\ \mathsf{or}\ \_ \rangle \ \$\ Es\ \$\ Fs) \\[4pt]
\mathsf{historically} &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
\mathsf{historically} &= \mathsf{scan}_1 \langle \_\ \mathsf{and}\ \_ \rangle \\[4pt]
\mathsf{once} &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
\mathsf{once} &= \mathsf{scan}_1 \langle \_\ \mathsf{or}\ \_ \rangle \\[4pt]
(\_\ \mathsf{sinces}\ \_) &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
(Es\ \mathsf{sinces}\ Fs) &= \mathsf{scan}_2 \langle \_\ \mathsf{or}\ \_\ \mathsf{and}\ \_ \rangle (Es)(Fs) \\[4pt]
(\_\ \mathsf{constrains}\ \_) &: \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \to \mathsf{Exp}(V)(\mathsf{bool})^\omega \\
(\_\ \mathsf{constrains}\ \_) &= \mathsf{scan}_2 \langle \_\ \mathsf{and}\ \_\ \mathsf{impl}\ \_ \rangle
\end{aligned}
$$

Fig. 19. Streams of expressions

Proposition 3:

$$
\begin{aligned}
[(\checkmark[\![Es]\!](\rho) \wedge \checkmark[\![Fs]\!](\rho)) &\Leftrightarrow \checkmark[\![Es\ \mathsf{ands}\ Fs]\!](\rho)] \\
[(\checkmark[\![Es]\!](\rho) \vee \checkmark[\![Fs]\!](\rho)) &\Leftrightarrow \checkmark[\![Es\ \mathsf{ors}\ Fs]\!](\rho)] \\
[\Box(\checkmark[\![Es]\!](\rho)) &\Leftrightarrow \checkmark[\![\mathsf{historically}(Es)]\!](\rho)] \\
[\Diamond(\checkmark[\![Es]\!](\rho)) &\Leftrightarrow \checkmark[\![\mathsf{once}(Es)]\!](\rho)] \\
[(\checkmark[\![Es]\!](\rho)\ \mathsf{S}\ \checkmark[\![Fs]\!](\rho)) &\Leftrightarrow \checkmark[\![Es\ \mathsf{sinces}\ Fs]\!](\rho)] \\
[(\checkmark[\![Es]\!](\rho) \rhd \checkmark[\![Fs]\!](\rho)) &\Leftrightarrow \checkmark[\![Es\ \mathsf{constrains}\ Fs]\!](\rho)]
\end{aligned}
$$

Proof: For $\wedge$ and $\vee$ the proofs are direct. For the temporal modalities, the proof is by induction on time. ■

The interpretation of pLTL as streams of expressions has been implemented, and used as a high-level constraint language. An example constraint is shown in Figure 20. It makes use of the pLTL modalities `always` and `never`, and the derived "before" modality $Es \ll Fs$, defined to be $\Box(Es \Rightarrow \Box\neg Fs)$.

This constraint is interpreted as a stream of boolean expressions $Es$, and expression $Es(k)$ is passed to an SMT solver (we used Microsoft's Z3 [4]). In the example, the smallest k that was satisfiable was 17, which generated 646 boolean variables, 544 integer variables, and 1191 constraints. Z3 found a solution within 200ms.

VI. Conclusions

In this paper, we have shown that functional reactive programs in a dependently typed language are expressive enough to define their own types. In particular, functional reactive types can express past-time LTL, as well as the proof rules for constructive S4 modal logic.

As an example of the power of functional reactive programming and functional reactive types, we defined a language of expression streams, such that k-bounded inhabitance of a constructive pLTL formula corresponds precisely to satisfiability of the kth expression. We have used this to define a constraint language based on pLTL+FRP, which translates k-bounded satisfiability to a constraint solved by an SMT solver.

Sections II–V of this paper are written in Literate Agda, and all results in those sections have been mechanically verified by the Agda proof checker.

As future work, it would be interesting to explore the connection between functional reactive types and type systems such as session types [10], typestates [6] or stateful types [13], which allow the type of a stream xs to depend not just on the

```
xCost = 150*x1 + 330*x2 + 30*x3 + 40*x4 + 40*x5 + 60*x6 + 60*x7 + 150*x8
yCost = 150*y1 + 330*y2 + 30*y3 + 40*y4 + 40*y5 + 60*y6 + 60*y7 + 150*y8
cost = xCost + yCost
x = x1 | x2 | x3 | x4 | x5 | x6 | x7
y = y1 | y2 | y3 | y4 | y5 | y6 | y7

constraint = (
  always(cost <= 330)
  & never(x & y)
  & (sum(x1) == 1)  & (sum(x2) == 1) & (sum(x3) == 21) & (sum(x4) == 8)
  & (sum(x5) == 16) & (sum(x6) == 1) & (sum(x7) == 1)  & (sum(x8) == 1)
  & (sum(y1) == 1)  & (sum(y2) == 1) & (sum(y3) == 21) & (sum(y4) == 8)
  & (sum(y5) == 16) & (sum(y6) == 1) & (sum(y7) == 1)  & (sum(y8) == 1)
  & (x1 << x2)      & (x2 << x3)     & (x1 << x4)
  & (x3 << x5)      & (x4 << x5)     & (x5 << x6)
  & (x5 << x7)      & (x6 << x8)     & (x7 << x8)
  & (y1 << y2)      & (y2 << y3)     & (y1 << y4)
  & (y3 << y5)      & (y4 << y5)     & (y5 << y6)
  & (y5 << y7)      & (y6 << y8)     & (y7 << y8)
)
```

Fig. 20. An example constraint expressed in pLTL+FRP

current time, but also on the values $xs(i)$ for $i < j$. It is easy to define a functional reactive type $As(xs)$ which depends on xs, but we cannot type xs as $xs : As(xs)$ as this type is not well-formed, since xs mentions itself in its own type.

In [15], Jeltsch presents a categorical definition of an "abstract process category" as a way of capturing models of FRP. It would be interesting to know whether the structure defined in this paper is an instance of his definition.

It would also be interesting to investigate further the relationship between $As \rhd Bs$ and resumption models [8], since $(As \rhd Bs)(n+1)$ is $B(n) \times (A(n) \to (As \rhd Bs)(n))$, which is the type of a resumption. Resumptions have been used in modeling coinductive streams [9] and iteratees [16], and it would be interesting to know how they could be used in a setting of functional reactive types.

REFERENCES

[1] The Agda wiki. http://wiki.portal.chalmers.se/agda/.

[2] N. Alechina, M. Mendler, V. de Paiva, and E. Ritter. Categorical and kripke semantics for constructive S4 modal logic. In *Proc. Computer Science Logic*, pages 292–307, 2001.

[3] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[5] L. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[6] R. Deline and M. Fähndrich. Typestates for objects. In *Proc. European Conf. Object-Oriented Programming*, pages 465–490. Springer, 2004.

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

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

[9] 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.

[10] K. Honda. Types for dyadic interaction. In *Proc. Int. Conf. Concurrency Theory*, number 715 in Lecture Notes in Computer Science, pages 509–523. Springer, 1993.

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

[12] A. S. A. Jeffrey. Functional reactive types. http://ect.bell-labs.com/who/ajeffrey/papers/lics14.tgz, 2013.

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

[14] W. Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2013.

[15] W. Jeltsch. An abstract categorical semantics for functional reactive programming with processes. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2014.

[16] O. Kiselyov. Streams and iteratees. http://okmij.org/ftp/Streams.html.

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

[18] C. McBride and R. Paterson. Applicative programming with effects. *J. Functional Programming*, 18(1):1–13, 2008.

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

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

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