Back to the Future: Time Travel in FRP

Ivan Perez
School of Computer Science
University of Nottingham
United Kingdom
ixp@cs.nottingham.ac.uk

Abstract

Functional Reactive Programming (FRP) allows interactive applications to be modelled in a declarative manner using time-varying values. For practical reasons, however, operational constraints are often imposed, such as having a fixed time domain, time always flowing forward, and limiting the exploration of the past.

In this paper we show how these constraints can be overcome, giving local control over the time domain, the direction of time and the sampling step. We study the behaviour of FRP expressions when time flows backwards, and demonstrate how to synchronize subsystems running asynchronously and at different sampling rates. We have verified the practicality of our approach with two non-trivial games in which time control is central to the gameplay.

CCS Concepts • Theory of computation → Functional constructs; • Software and its engineering → Control structures;

Keywords functional reactive programming, game programming, time, stream programming, monadic streams, Haskell

ACM Reference Format:
https://doi.org/10.1145/3122955.3122957

1 Introduction

Functional Reactive Programming (FRP) [5, 6, 11] is a declarative approach to implementing reactive applications centred around programming with time-varying values. FRP systems are described by defining how these time-varying values, or signals, depend on each other over time (as opposed to operational descriptions of how to react to individual events).

Certain constraints are usually required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be causal: they cannot depend on other signals at future times.

The nature and progress of time is central to meeting these requirements. In most implementations, the time domain is the same for the whole system, time always advances towards the future, and synchronicity or asynchronicity is global to the framework. For instance, in implementations like Yampa [5, 11], access to other signals is limited to the present, time has floating-point precision and the whole system advances in unison on a synchronous clock. In others time is hidden, and different subsystems run asynchronously, being recalculated as the values they depend on change [12].

These constraints on time and how it progresses do not always accommodate certain applications well. For instance, many board games, like chess, are conceptually discrete, but implementations may include visual transitions and animations that happen in continuous time. In other games, like Braid [8], time can advance towards the past or non-linearly, enabling interesting features.

This paper presents a uniform way of addressing these limitations by focusing on the nature of time and the coordination of systems running on different time domains.

Specifically, the contributions of this paper are:
- We extend an existing implementation to perform time transformations, within or between time domains.
- We study the behaviour of FRP systems when such transformations reverse the direction of time, and provide variants of existing FRP constructs that are temporally consistent when time flows backwards.
- We extend an existing implementation to give local time control, and handle the coordination of asynchronous subsystems with possibly different time domains.
- We demonstrate our approach with two non-trivial games in which time control and reversal are central to the gameplay.

2 Background

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP and Yampa in the following. For further details, see earlier papers on FRP and Arrowized FRP (AFRP) as embodied by Yampa [5, 6, 11]. This presentation draws heavily from the summaries in [5, 14].

2.1 Functional Reactive Programming

FRP is a programming paradigm to describe hybrid systems that operate on time-varying data. FRP is structured around the concept of signal, which conceptually can be seen as a function from time to values of some type:

\[ \text{Signal} \; \alpha \approx \text{Time} \rightarrow \alpha \]

Time is (notionally) continuous, and is represented as a non-negative real number. The type parameter \(\alpha\) specifies the type of values carried by the signal. For example, the type of an animation would be \(\text{Signal Picture}\) for some type \(\text{Picture}\) representing static pictures. Signals can also represent input data, like the position of the mouse on the screen.

Additional constraints are required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to...
be causal: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time.

The space of FRP frameworks can be subdivided in two main branches, namely Classic FRP [6] and Arrowized FRP [11]. Classic FRP programs are structured around signals or a similar notion representing internal and external time-varying data. In contrast, Arrowized FRP programs are defined using causal functions between signals, or signal functions, connected to the outside world only at the top level.

Arrowized FRP renders declarative and efficient code, and facilitates debugging [15]. In the following we turn our attention to Arrowized FRP as embodied by Yampa, and later explain current limitations that our framework addresses.

2.2 Fundamental Concepts

Yampa is based on two concepts: signals and signal functions. A signal, as we have seen, is a function from time to values of some type, while a signal function is a function from Signal to Signal:

\[
\begin{align*}
\text{Signal } \alpha & \approx \text{Time } \rightarrow \alpha \\
\text{SF } \alpha \beta & \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
\end{align*}
\]

When a value of type SF \( \alpha \beta \) is applied to an input signal of type Signal \( \alpha \), it produces an output signal of type Signal \( \beta \). Signal functions are first class entities in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function.

In order to ensure that signal functions are executable, we require them to be causal: The output of a signal function at time \( t \) is uniquely determined by the input signal on the interval \([0, t]\).

2.3 Composing Signal Functions

Programming in Yampa consists of defining signal functions compositionally using Yampa’s library of primitive signal functions and a set of combinators. Yampa’s signal combinators are an instance of the arrow framework proposed by Hughes [7]. Some central arrow combinators are arr that lifts an ordinary function to a stateless signal function, composition \( \gg \), parallel composition \&\&\&, and the fixed point combinator loop. In Yampa, they have the following types:

\[
\begin{align*}
\text{arr} & :: (a \rightarrow b) \rightarrow \text{SF } a \ b \\
(\gg) & :: \text{SF } a \ b \rightarrow \text{SF } b \ c \rightarrow \text{SF } a \ c \\
(\&\&\&) & :: \text{SF } a \ b \rightarrow \text{SF } a \ c \rightarrow \text{SF } a \ (b, c) \\
\text{loop} & :: \text{SF } (a, c) \ (b, c) \rightarrow \text{SF } a \ b
\end{align*}
\]

We can think of signals and signal functions using a simple flow chart analogy. Line segments (or “wires”) represent signals, with arrowheads indicating the direction of flow. Boxes represent signal functions, with one signal flowing into the box’s input port and another signal flowing out of the box’s output port. Figure 1 illustrates the aforementioned combinators using this analogy. Through the use of these and related combinators, arbitrary signal function networks can be expressed.

2.4 Time-Variant Signal Functions: Integrals and Derivatives

Signal functions must remain causal and leak-free, and so Yampa introduces limited ways of depending on past values of other signals. Integrals and derivatives are important for application domains like games, multimedia and physical simulations, and they have well-defined continuous-time semantics. Their types in Yampa are as follows (\( v \) represents the type of vectors, and \( s \) the type of scalars):

\[
\begin{align*}
\text{integral} & :: \text{VectorSpace } v s \Rightarrow \text{SF } v v \\
\text{derivative} & :: \text{VectorSpace } v s \Rightarrow \text{SF } v v
\end{align*}
\]

Time-aware primitives like the above make Yampa specifications highly declarative. For example, the position of a falling mass starting from a position \( p0 \) with initial velocity \( v0 \) is calculated as:

\[
\begin{align*}
\text{fallingMass} & :: \text{Double } \rightarrow \text{Double } \rightarrow \text{SF } () \text{ Double} \\
\text{fallingMass } p0 \ v0 & = \text{arr } (\text{const } (-9.8)) \\
& \gg \text{integral} \gg \text{arr } (+v0) \\
& \gg \text{integral} \gg \text{arr } (+p0)
\end{align*}
\]

which resembles well-known physics equations (i.e. “the position is the integral of the velocity with respect to time”) even more when expressed using Paterson’s Arrow notation [13]:

\[
\begin{align*}
\text{fallingMass} & :: \text{Double } \rightarrow \text{Double } \rightarrow \text{SF } () \text{ Double} \\
\text{fallingMass } p0 \ v0 & = \text{proc } () \Rightarrow \text{do} \\
& v \leftarrow \text{arr } (+v0) \ll \text{integral} \ll (-9.8) \\
& p \leftarrow \text{arr } (+p0) \ll \text{integral} \ll v \\
& \text{return } A \ll p
\end{align*}
\]

2.5 Events and Event Sources

To model occurrences at discrete points in time, Yampa introduces the Event type [11]:

\[
\text{data Event } a = \text{NoEvent } | \text{Event } a
\]

A signal function whose output signal is of type Event \( T \) for some type \( T \) is called an event source. The value carried by an event occurrence may be used to convey information about the occurrence. The operator tag is often used to associate such a value with an occurrence:

\[
\text{tag } :: \text{Event } a \rightarrow b \rightarrow \text{Event } b
\]

2.6 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as mode switches. This is accomplished through a family of switching primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is switch:

\[
\text{switch } :: \text{SF } a \ (b, \text{Event } c) \rightarrow (c \rightarrow \text{SF } a \ b) \rightarrow \text{SF } a \ b
\]
Back to the Future: Time Travel in FRP

The switch combinator switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that is active initially. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, switch applies its second argument to the value tagged to the event and switches into the resulting signal function.

Yampa also includes parallel switching constructs that maintain dynamic collections of signal functions connected in parallel [11]. Signal functions can be added to or removed from such a collection at runtime in response to events, while preserving any internal state of all other signal functions in the collection (see Fig. 2). The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems.

2.7 Animating Signal Functions

To actually execute a Yampa program, i.e. the top-level signal function representing an entire system, we need some way to connect the program’s input and output signals to the external world. Yampa provides the function reactimate (here slightly simplified) for this purpose:

```haskell
reactimate :: IO (DTime, a) → (b → IO ()) → SF a b → IO ()
```

Reactimate approximates the continuous-time model presented here by performing discrete sampling of the signal function, feeding input to and processing output from the signal function at each time step. The first argument to reactimate is an IO action that will obtain the next input value along with the amount of time elapsed (or "delta" time, DTime) since the previous sample. For example, in a setting of music with CD audio quality, the delta time would be set to correspond to a system sampling frequency of 44.1 kHz, i.e. 1/44100 s, and the input could be note-on and note-off messages from a connected synthesizer keyboard. The second argument is a function that, given an output value, produces an IO action that will process the output in some way. For example, it could send sound samples to the audio subsystem for immediate playback. The third argument is the signal function to be animated.

2.8 Limitations of Current AFRP Systems

Yampa has a number of limitations, described in the following. Most of these also apply to other current AFRP systems.

Time Domain FRP is conceptually continuous, and Yampa uses a double-precision floating-point number (Double) to represent time and time steps. However, many games run on a discrete clock, while others require a rational clock with arbitrary precision or no clock at all. Having a fixed representation of time for all the components in the game limits the applicability of FRP to parts defined over different time domains.

Time Step Yampa signal functions progress all with the same sampling step, which limits the constructs that can be defined and their reusability. For instance, it is impossible to make a signal function progress twice as fast without modifying its implementation or making all signal functions progress twice as fast.

Time Direction FRP is required to always progress towards the future, and Yampa makes all time steps positive. However, games like Braid [8] enable time reversal, making it possible to rewind parts of the game. Other games allow the player to jump forward and backwards along the timeline, and even provide multiple accesses to the same timeline with actions in the past immediately affecting a distant future.

Synchronicity Yampa is synchronous, and all signal functions progress in unison. When given control over the time step, or when they operate over different time domains (e.g. discrete-time vs continuous-time), some FRP Signal Functions may need to be sampled more often than others.

3 Synchronous Time Transformations Towards the Future

In Arrowized FRP implementations like Yampa all signal functions progress with a global clock and move forward by the same amount of time. This makes Yampa signal functions somewhat limited.

For instance, imagine that we have a signal function that represents a ball going around in circles:

```haskell
circlingBall :: SF a (Double, Double)
circlingBall = proc (_) → do
  t ← time ←< ()
  let radius = 100 -- pixels
      (x, y) = (radius * cos t, radius * sin t)
  returnA ← < (x, y)
```

No mechanism within Yampa allows us to run circlingBall faster or slower than it normally would without affecting other signal functions. For example, if we wanted to show two balls, one moving twice as fast as the other, we would need to modify the implementation of circlingBall.

3.1 Point-wise Time Transformations

In Yampa, signal functions are progressed by sampling them with positive time deltas, which represent the time difference between the previous and the current sample.

We can make a signal function progress at a different speed by directly transforming those time deltas. If the transformation does not depend on the past state or the global time, then we can use the following function:

```haskell
timeTransform :: (DTime → DTime) → SF a b → SF a b
```

Note that this does not make the argument signal function "tick" more or less frequently than others, but simply makes the internal function "believe" that more or less time has passed between samples. DTime is Yampa’s type representing time deltas. For the time being, we consider DTime to be a positive Double, which maintains causality at a conceptual level even as we transform time.
Example Given `circlingBall` defined before, we run two such animations in parallel, one twice as fast as the other, as follows:

```haskell
twoBalls :: SF a ((Double, Double), (Double, Double))
twoBalls = circlingBall &&& timeTransform (+2) circlingBall
```

We can also make signal functions tick with a fixed step (e.g. `timeTransform (const (1 / 60))`), or with a minimum or maximum time step (e.g. `timeTransform (max (1 / 120))`).

3.2 State-depending Time Transformations

One might want the time transformation to be applied only in some circumstances, for instance, when the player collects a special “power-up” or presses a certain key. This feature is common in games; a notable example is Max Payne’s Bullet Time1, which gives the player the ability to slow time down and aim weapons with more precision.

To facilitate this task we provide:

```haskell
timeTransformSF :: SF a (DTime → DTime) → SF a b → SF a b
```

At every point in time, the first signal function produces a time transformation operating on the time deltas that can depend on the global time or the history of the system. This transformation is applied to the current time delta, which is then used to progress the second signal function.

Example Given a user input controller with an action button to determine if time should run in slow-motion:

```haskell
data Input = Input { slow :: Bool }
```

We adapt our example to use that feature as follows:

```haskell
game :: SF Input ((Double, Double), (Double, Double))
game = timeTransformSF timeProgression twoBalls
timeProgression :: SF Input (DTime → DTime)
timeProgression = arr (λc → if slow c then (0.1∗) else id)
```

We use `timeProgression` to make time deltas smaller upon user request. This example shows that time transformations are composable: `twoBalls` internally applies another time transformation to the second ball. In Section 7 we present more sophisticated ways of using these transformations.

4 Synchronous Time Transformations Towards the Past

In FRP, time always flows towards the future. Frameworks like Yampa implement this feature by applying positive time deltas to signal functions. In implementations like Elerea [12] time is implicit and time deltas are always of one unit of time. Applying positive time deltas helps enforce two additional requirements of FRP: signals must be causal (they cannot depend on the future), and they must be referentially transparent.

Nevertheless, non-linear time is a central feature in many games. Some games let the player jump back and forth between different points in a timeline, while in others time can flow backwards [8]. Implementations like Dunai [14] and Netwire [19] are fully polymorphic in the time domain, letting us, in principle, use more structured representations of time, and negative time deltas.

In this section we show how to make time flow back in Arrowized FRP. We first introduce a simple version of time reversal, and analyse how existing FRP constructs behave under such conditions. We then introduce variants of existing combinators that behave consistently when time flows back. Finally, we show how to cache results to deal with inherently non-reversible signals, and study how to limit the amount of history that can be explored.

4.1 Reversing Time

Combinators like `timeTransform`, introduced in the previous section, could in principle be used to negate time deltas:

```haskell
reverseSF :: SF a b → SF a b
reverseSF = timeTransform (λdt → −dt)
```

Note that here we no longer assume that `DTime` is necessarily positive. For the sake of simplicity, we abuse the type `SF` to refer to signal functions with positive or negative time deltas.

Example The examples introduced in the previous section work equally well when time is reversed. For instance, we can run `circlingBall` as follows to see the ball move in the opposite direction:

```haskell
circlingBallR :: SF a (Double, Double)
circlingBallR = timeTransform (λdt → −dt) circlingBall
```

Time reversal is most interesting when applied only at certain times or depending on the game state. We extend the example defined for slowing time down as follows:

```haskell
game′ :: SF Input ((Double, Double), (Double, Double))
game′ = timeTransformSF timeProgression game
timeProgression :: SF Input (DTime → DTime)
timeProgression = arr (λc → if backwards c then ((−1)∗) else id)
data Input = Input { backwards :: Bool, slow :: Bool }
```

Here, we use `timeProgression` to make time deltas negative if the user requests it, by extending our type for user input with a `backwards` boolean record field. This definition nests three time transformations, defined in `game′`, `game`, and `twoBalls`.

4.2 Reversible Signal Functions

In principle, one would expect FRP signals and signal functions to be temporally consistent: if we roll time back, then signals should hold the values they had in the past. Providing temporal consistency presents challenges of its own both for input signals and for signal functions. Some signals may represent external data (e.g. mouse position) and providing temporal consistency would require keeping an unbounded amount of data, leading to space/time leaks. Additionally, not all Signal Functions behave consistently when time is reversed. Yampa and other Arrowized FRP frameworks are implemented using continuations, and signal functions change over time, often "forgetting" their past.

We refer to signal functions that behave coherently when time is reversed as temporally consistent, and otherwise we call them temporally directed.

In this section we explore temporal consistency of signal functions, but the mechanisms proposed can be used for external input signals as well. We focus exclusively on the meaning of signal functions and combinators, and disregard implementation details such as if a particular function in Yampa throws an error when applied to negative time deltas.

---

1 http://www.rockstargames.com/maxpayne/main.html
4.2.1 Overview of Core FRP Constructs

Fig. 3 presents a list of FRP definitions that define the core of Yampa [3], and indicates which ones are temporally consistent (Fig. 3a) and which ones are temporally directed (Fig. 3b).

\[
\begin{align*}
arr &: (a \rightarrow b) \rightarrow SF a b \\
(\gg\gg) &: SF a b \rightarrow SF b c \rightarrow SF a c \\
first &: SF a c \rightarrow SF (a, b) (c, b) \\
time &: SF a Time \\
integral &: (Floating a) \Rightarrow SF a a \\
loop &: SF (a, c) (b, c) \rightarrow SF a b
\end{align*}
\]

(a)

\[
iPre &: a \rightarrow SF a a \\
switch &: SF a (b, Event c) \rightarrow (c \rightarrow SF a b) \rightarrow SF a b
\]

(b)

**Figure 3.** FRP primitives: temporally consistent (a) and temporally directed (b).

The primitives `arr`, `time` and `integral` are temporally consistent, while `first`, `(\gg\gg)` and `loop` are so only when applied to temporally consistent arguments.

Both `switch` and `iPre` are temporally directed. Switch modifies the signal function being applied and, to be reversed, needs to remember the original transformation before the point of switching. `iPre` introduces a delay and, to reverse it, we need to cache part of the input signal.

In the following we focus our attention on these two temporally directed constructs and provide temporally consistent alternatives.

4.2.2 Reversible Switching

Switching in FRP allows a transformation to be substituted for another at a certain point during the execution. In its simplest form, switching is achieved with the following function:

\[switch :: SF a (b, Event c) \rightarrow (c \rightarrow SF a b) \rightarrow SF a b\]

A signal function is active initially, producing an output and an event. The output defines the overall output by default while the event does not take place. However, if the event carries a value, then the default output is ignored and the function given as second argument is used to start a new signal function to be used for the present and all future times.

For instance, imagine that we have a signal function that represents a ball bouncing up and down. For simplicity, let us deal only with the ball’s vertical coordinate.

\[\begin{align*}
\text{bouncingBall} &: Double \rightarrow Double \rightarrow SF () Double \\
\text{bouncingBall} p0 v0 &=
\text{switch} (\text{fallingBall} p0 v0 \gg\gg (\text{arr} \text{ fist} \&\& \text{hitFloor})) \\
&\quad (\lambda(p, v) \rightarrow \text{bouncingBall} p (-v))
\end{align*}\]

\[\begin{align*}
\text{fallingBall} &: Double \rightarrow Double \rightarrow SF () (Double, Double) \\
\text{fallingBall} p0 v0 &= \text{proc} () \rightarrow do \\
& v \leftarrow (v0+) \ll\ll \text{integral} \ll (-9.8) \quad -- \text{m/s} \\
& p \leftarrow (p0+) \ll\ll \text{integral} \ll v \\
& \text{returnA} \ll\ll (p, v)
\end{align*}\]

The function `bouncingBall` starts with an initial position and velocity, behaving like a falling mass until it hits the floor. When it does, the velocity is negated, the ball starts moving up, and the same is repeated again.

If we run the previous signal function with initial position 100.0 and zero velocity, and sample it every second, we obtain the following (number of decimal digits modified to facilitate readability):

\[ [100.0, 90.2, 70.6, 41.19, 2.0, -47.0, 11.79, 60.8, 100.0, 129.4, ...] \]

The third sample shows that the ball temporarily collides with the floor, but it bounces back towards positive positions. Note that ball bounces above 100, but this is merely an artifact of using Euler integration with a very large sampling step. This calculation error approximates zero as the sampling step becomes smaller.

The previous definition works well when time moves forward, but not when time flows backwards. For example, if we run it again but reverse time after the bounce, we observe:

\[ [100.0, 90.2, 70.6, 41.19, 2.0, -47.0, 11.79, -37.2, -96.0, -164.6, ...] \]

The condition that triggers bouncing requires that the velocity be negative, indicating that the ball is falling, something that is not true when we reverse time after bouncing (the ball looks like it is falling, but only because we are rewinding the simulation).

While it might be possible to modify this particular definition to detect the bounce backwards, we provide a general way of reversing switching with the following combinator:

\[\text{revSwitch} :: SF a (b, Event c) \rightarrow (c \rightarrow SF a b) \rightarrow SF a b\]

which keeps the original signal function after the switch and uses it if time is reversed to a point before the switch took place. This makes expressions that use `revSwitch` temporally consistent, so long as all arguments are temporally consistent as well. Rewriting the original signal function with this switching combinator we obtain:

\[\begin{align*}
\text{bouncingBallR} &: Double \rightarrow Double \rightarrow SF () Double \\
\text{bouncingBallR} p0 v0 &= \\
& \text{revSwitch} (\text{fallingBall} p0 v0 \gg\gg (\text{arr} \text{ fist} \&\& \text{hitFloor})) \\
&\quad (\lambda(p, v) \rightarrow \text{bouncingBall} p (-v))
\end{align*}\]

Note that `revSwitch` stores the original signal function used before switching, preventing it from being garbage collected. This may require additional treatment in the presence of recursion.

In some games time is only “reversed” for certain elements or actions, while some changes cannot be undone and persist when time flows back. The existence of both a forgetful switch and a time-reversible switch is a feature, and will be used in Section 7 to implement an interesting gameplay.

4.2.3 Caching and Reversible Delays

The primitive `iPre` introduces an “infinitesimal” delay in a signal. Its default behaviour, as defined by its operational semantics [3], completely ignores time deltas. Even if the direction of time is reversed, `iPre` behaves the same way.

To provide a temporally consistent variant we need to remember the past of the signal and re-create it if necessary. This presents a
challenge if the value requested was never sampled in the first place. For that purpose we first provide an auxiliary caching facility:

\[
\text{cache} :: \left(\text{Time}, a\right) \rightarrow \text{Maybe} \left(\text{Time}, a\right) \rightarrow \text{Time} \rightarrow a \rightarrow SF\ a\ a
\]

This signal function behaves like the identity when time flows forward, and shows old values when time flows back. The function argument interpolates values from existing samples.

**Example** Caching can be used with any signal, including those that represent user input. Let us demonstrate with an example of an animation of a ball going in circles around the mouse position.

We define a type for user input containing the current mouse coordinates:

\[
data\ \text{Input} = \text{Input}\ \{\ \text{playerPos} :: \text{Pos2D}\ \}
\]

\[
type\ \text{Pos2D} = (\text{Float}, \text{Float})
\]

We now define a signal function for the moving ball, displaced by some base coordinates:

\[
\text{ballAroundMousePos} :: SF\ \text{Input}\ \text{Pos2D}
\]

\[
\text{ballAroundMousePos} = \text{arr}\ \text{playerPos} \gg \text{ballAroundPos}
\]

\[
\text{ballAroundPos} :: SF\ \text{Pos2D}\ \text{Pos2D}
\]

\[
\text{ballAroundPos} = \text{proc}\ (x, y) \rightarrow \text{do}
\]

\[
t \leftarrow \text{time} \leftarrow ()
\]

\[
\text{let}\ \text{radius} = 100
\]

\[
\left(x', y'\right) = \left(x + \text{radius} \times \cos\ t, y + \text{radius} \times \sin\ t\right)
\]

\[
\text{return} \leftarrow \left(x', y'\right)
\]

If we reverse this simulation, the ball will move in clockwise (opposite) direction, but around the present, not the past, mouse position. We can reverse \text{ballAroundPos}, but the \text{Input} signal is not consistent with its past. Instead, we define the following alternative:

\[
\text{ballAroundMousePosR} :: SF\ \text{Input}\ \text{Pos2D}
\]

\[
\text{ballAroundMousePosR} = \text{arr}\ \text{playerPos} \gg \text{cache}\ \text{interpolateMousePos}\ \gg \text{ballAroundPos}
\]

\[
\text{interpolateMousePos} :: \left(\text{Time}, \text{Pos2D}\right) \rightarrow \text{Maybe} \left(\text{Time}, \text{Pos2D}\right) \rightarrow \text{Time} \rightarrow \text{Pos2D}
\]

\[
\text{interpolateMousePos} t1 p1 \rightarrow\ \text{Nothing}\ \rightarrow\ p1\ \text{ interpolateMousePos} t1\left(p1x, p1y\right) \left(\text{just} \left(t2, (p2x, p2y)\right)\right) t = p\ \text{where}
\]

\[
p = (p1x + (p2x - p1x) \times dtp, p1y + (p2y - p1y) \times dtp)
\]

\[
dtp = (t - t1) / t2 - t1
\]

When executed backwards, the ball will move towards old mouse positions. Intermediate positions, not sampled during the original execution, will be interpolated linearly from existing samples.

We can implement an alternative version of \text{iPre} by using caching, together with an interpolation function. For instance, if we assume that the first sample always needs to be used in between samples, the following implementation would suffice:

\[
\text{revIPre} :: a \rightarrow SF\ a\ a
\]

\[
\text{revIPre} a0 = \text{iPre}\ a0\ \gg\ \text{cache}\ \left(\lambda\ a\ a\rightarrow a\right)
\]

**4.2.4 Limiting History**

Caching introduces the possibility of unbounded memory leaks. If one needs to remember an unlimited amount of history, the memory consumption of the program will grow during execution.

We address these concerns with two alternative caching functions that limit how much past is kept. The first alternative is the following function:

\[
\text{cacheMax} :: DTime
to\ ((\text{Time}, a) \rightarrow \text{Maybe} \left(\text{Time}, a\right) \rightarrow \text{Time} \rightarrow a) \rightarrow SF\ a\ a
\]

This function behaves almost like \text{cache}, except that the amount of history is limited by the argument of type \text{DTime}. As time moves forward, old samples are dropped.

This function limits the amount of history with a time delta, but it may still consume too much memory if time deltas become very small. We provide a second, more general function that gives total control over the history kept after each sample:

\[
\text{cacheWith} :: \left([(\text{Time}, a)] \rightarrow [(\text{Time}, a)]\right) \rightarrow ((\text{Time}, a) \rightarrow \text{Maybe} \left(\text{Time}, a\right) \rightarrow \text{Time} \rightarrow a) \rightarrow SF\ a\ a
\]

This second approach is more suitable when we prefer to drop intermediate values and interpolate them when necessary, or when we need to limit history based on the number of samples.

**Example** Using these functions is now straightforward. We only need to add the amount of history that we want to preserve:

\[
\text{ballAroundMousePosC} :: SF\ \text{Input}\ \text{Pos2D}
\]

\[
\text{ballAroundMousePosC} = \text{arr}\ \text{playerPos} \gg \text{cacheMax}\ 10\ \text{interpolateMousePos}\ \gg\ \text{ballAroundPos}
\]

This will behave like \text{ballAroundMousePosR}, except that it will behave as if, for all history before 10 seconds from the present, the mouse position has not changed.

These functions effectively limit the amount of memory consumed. Figure 4 shows the memory profile of this example with unbounded (Fig. 4a) and with bounded caching (Fig. 4b). The sudden drops in the profile correspond to the activation of time reversal.

**4.3 Checkpoints**

To complement the possibility of progressing towards the past, we also provide a way to create checkpoints, that is, to “bookmark” a point in history and jump directly to it in a timeline. This is different from caching, in the sense that we are not saving the value of a signal or applying a negative time delta, but rather keeping a signal function and restoring it.

Games often use checkpoints to let the players resume from a point in the middle of a level when the in-game character dies, without losing all progress (Fig. 5).

To implement checkpoints we provide the following interface:

\[
\text{checkpoint} :: SF\ a\ b, \text{Event} () \rightarrow SF\ a\ b
\]

The argument signal function must produce two additional events apart from its main output. The first is an event that indicates when a checkpoint is reached. The second is an event that indicates when the checkpoint needs to be restored. Unlike unbounded caching, checkpoints do not create unbounded memory leaks: a checkpoint combinator may keep, at most, one past signal function in memory.
Example Imagine that, in a game, a checkpoint is located at a certain vertical position \texttt{checkPointY}. We can define a player that detects when it reaches the checkpoint and indicates when it needs to go back to it as follows:

\[
\text{playerWithSave} :: \text{SF Input (Pos2D, Event ()), Event ()}
\]

\[
\text{playerWithSave} = \text{proc input } \rightarrow \text{ do}
\]

\[
\text{ (playerPos, dead) } \leftarrow \text{ player } \rightarrow \text{ input}
\]

\[
\text{reachedCheckpoint } \leftarrow \text{ edge } \leftarrow (\text{snd playerPos} \> \text{checkPointY})
\]

\[
\text{let restoreCheckpoint } = \text{ if dead then Event () else noEvent returnA } \leftarrow (\text{playerPos, reachedCheckpoint, restoreCheckpoint})
\]

-- Defined elsewhere

\[
\text{player} :: \text{SF Input (Pos2D, Bool)}
\]

We now use the above to restart the game from the checkpoint if the in-game character dies:

\[
\text{immortalPlayer} :: \text{SF Input Pos2D}
\]

\[
\text{immortalPlayer} = \text{checkpoint playerWithSave}
\]

More realistically, checkpoints would surround the complete game, as opposed to only the player, and their use would be limited by the number of lives the player has left.

5 Transforming between Time Domains

FRP requires that time be continuous, which does not accommodate all kinds of games well. Many games, like board games, are inherently discrete, so having the natural numbers as the only time domain or simply having no time at all might be enough to define them and, in certain circumstances, more efficient. Other games might have compound and non-linear representations of time.

The definitions presented so far follow the convention that the implicit type \texttt{DTime} represents time deltas. In Yampa, \texttt{DTime} is a \texttt{Double} and is required to be positive.

More general FRP implementations like Dunai [14] allow the time domain to be altered. The basic construct in Dunai is that of a Monadic Stream Function, a type equipped with a \texttt{step} function:

\[
\text{data MSF m a b} = \text{MSF}
\]

\[
\text{\{} stepMSF :: a } \rightarrow \text{ m (b, MSF m a b) }\}
\]

The meaning of executing an MSF is given by applying it to the first sample of an input stream, obtaining the first output, and using the continuation to transform the tail of the stream. For simplicity, in the following we omit the monad constraint on \texttt{m}.

When the monad is the \texttt{Reader} monad with \texttt{DTime} in the environment, the monadic stream function becomes isomorphic to Yampa’s Signal Functions (disregarding minor optimisation details). Yampa’s operational semantics is also defined by a step function, and input samples include a value and the time delta since the previous sample [3].

Monadic Stream Functions are more expressive than Signal Functions: we can use more structured types for time (negative or positive, natural, etc.), and also different monads to nest effects. Control monads naturally give rise to common FRP combinators. The \texttt{Maybe} monad, when used in a monadic stream functions, gives rise to terminating stream functions. The \texttt{Either} monad gives rise to switching, and the using the list monad provides parallelism with broadcasting. Monads like the \texttt{Writer} monad can be used to log information about Monadic Stream Functions (debug log, etc.) and to implement simple versions of continuous collision detection.

One could use this generality of Dunai to include more sophisticated time domains, like branching time, or simpler time domains,
like the natural numbers. The following combinator, a generalization of \text{timeTransformSF}, allows us to embed a stream function in one time domain in a larger system with a different time domain:
\[
\text{timeTransformMSF} :: \text{MSF} (\text{ReaderT} t_1 \text{m}) a (t_1 \rightarrow t_2) \\
\quad \rightarrow \text{MSF} (\text{ReaderT} t_2 \text{m}) a b \\
\quad \rightarrow \text{MSF} (\text{ReaderT} t_1 \text{m}) a b
\]

The use of this kind of transformation to run two MSFs within the same domain, one faster than the other, is demonstrated in [14].

**Example** A common usage for \text{timeTransformMSF} would be to transform between the different types and units that multimedia frameworks like Simple Direct-media Layer (SDL) use to represent time. For instance, we can run a game that uses natural numbers for time as part of a larger system in continuous time as follows:

\[
game :: \text{MSF} (\text{Reader DTime}) \text{GameInput GameVisuals} \\
game = \text{timeTransformMSF} \text{(constant round)} \text{discreteGame} \\
\quad \gg \text{animations}
\]

\[
discreteGame :: \text{MSF} (\text{Reader Int}) \text{GameInput GameState} \\
\quad \gg \text{animations}
\]

\[
discreteGame :: \text{MSF} (\text{Reader DTime}) \text{GameState GameVisuals}
\]

### 5.1 Asynchronous Coordination

The previous construct runs both the discrete game and the continuous animations synchronously, which is not always ideal.

Consider, for example, the case of having a discrete \text{board} game which is animated in continuous time. In such a setting, changes to the board are applied at a discrete point, but may take some time to animate and be presented to the user.

When a change takes place, the user still sees the old board, or an intermediate state, but the conceptual game representation may already reflect the new game state. If the user acts on the board during a transition period, the result may be counter-intuitive. Input events would be applied to a discrete game that already reflects the result after the piece is moved, which is not what the user is seeing at that time.

This problem is common in GUI frameworks, and different implementations opt for different design decisions. On Android, for instance, UI events are applied on the conceptual model regardless of whether the UI has "caught" up. The lack of visual feedback to indicate that the system is waiting or processing data in the background may generate frustration on the part of the user, whose actions are applied on UI elements that were not yet present when the event actually took place.

#### 5.1.1 Freezing

One can synchronize the conceptual and visual layers of an application by making the conceptual layer progress only when the visual layer is done animating previous changes.

An ad-hoc way to synchronize the two layers can be achieved by making the visual layer output events that indicate when it has caught up with the conceptual changes. This, however, would require extending the type for input events to include visual events, and having the conceptual game be aware of and deal with data from the visual layer.

We can define this problem in terms of the time domains of different signal functions and implement a general mechanism for discrete-continuous synchronization. This requires the introduction of two extensions: making it possible to temporarily pause the game (i.e. allowing time deltas to be zero), and making the visual layer determine the time delta of the conceptual layer.

First, we make the discrete part of the game run on a monad that indicates whether the game has stepped forward or not. In terms of time, we are extending the type of the time deltas, from being always exactly one to being zero or one.

**type DiscreteMonad m = ReaderT Bool m**

Referential transparency in FRP would require that a zero time delta produce the same output as in the previous iteration. We can provide a mechanism to step a game forward only if the clock has a non-zero time delta (\text{True} in our \text{DiscreteMonad}), as follows:

\[
\text{possiblyRun} :: b \rightarrow \text{MSF} m a b \rightarrow \text{MSF} (\text{ReaderT} \text{Bool} m) a b \\
\text{possiblyRun bo msf} = \text{MSF} \lambda a \rightarrow \text{do} \\
\quad \text{run} \leftarrow \text{ask} \\
\quad \text{if} \ \text{run} \ \text{then} \ \text{do} \ (b, \text{msf}^\prime) \leftarrow \text{lift} (\text{stepMSF} \text{msf} a) \\
\quad \quad \text{return} \ (b, \text{possiblyRun} \ b \ \text{msf}^\prime) \\
\quad \text{else} \ \text{return} \ (b, \text{possiblyRun} \ b \ \text{msf})
\]

This function "freezes" the internal signal function if the boolean in the reader environment is \text{False}. When that happens, it returns the last known output or a default initial output.

We can now enclose the visual signal function in a similar wrapping, with the following monad:

**type ContinuousMonad m = StateT DTime m**

This lets the visual layer determine the time step that the conceptual layer should use, which corresponds to whether it is done animating the last change (\text{True}) or not (\text{False}).

If we now make the reader monad of the discrete part take the boolean from the state environment, and reset such state to \text{True} before every step of the evaluation of the visual layer, then we will effectively have synchronized both:

\[
game :: \text{MSF} \text{(ContinuousMonad m)} \text{Input VisualState} \\
game = \text{readerToState} \ (\text{possiblyRun} \ \text{defGameState} \text{game}) \\
\quad \gg \text{withSideEffect} \_ \ (\text{put} \ \text{True}) \\
\quad \gg \text{visualLayer} \\
\]

Auxiliary definitions

\[
defGameState :: \text{GameState} \\
defGameState :: \text{MSF} m \text{Input GameState} \\
\text{visualLayer} :: \text{MSF} \text{(ContinuousMonad m)} \text{GameState VisualState} \\
\text{readerToState} :: \text{MSF} (\text{Reader s m}) a b \rightarrow \text{MSF} \text{(StateT s m)} a b
\]

The real-time animation detects changes to the discrete part and decides whether it should animate them and take some time to do so. When it does, it changes the state of the monad to \text{False}, indicating that it is trying to catch up, and so the rest of the game does not run.

More generally, this kind of synchronization, with one signal function deciding when to step another signal function and by how much, requires that the former writes in a context the time deltas of the latter, and the appropriate wiring functions to convert between contexts or monads.

### 5.2 Self-clocking Signal Functions

The previous construct is a special case of a more general construct that lets a signal function control its own time step. The basic construct for local time control is a self-clocking signal function:
type SelfClockingMSF m a b =
  MSF (StateT (DT ime,Bool) (ReaderT DT ime m)) a b

This kind of signal function gets the time delta from a reader context, and stores in the state the actual time delta used, and whether it could run again within the time left. We only require the write capacity of StateT, but we avoid WriterT due to its additional Monad constraint.

In terms of time we are giving this stream function self-timing abilities, and we are extending the time domain with a Bool that indicates whether a simulation cycle has been completed.

For instance, imagine that a signal function advances with a fixed time delta of 16ms regardless of actual time delta provided. If the actual delta is 44ms, we should be able to run it twice within that time. The first iteration would write (16, True) in the state, indicating that there are iterations pending and that it could run again within the given time delta. A second iteration of the MSF would then return, in the state, (16, False), with False indicating that not enough time has passed to run again.

We define the following combinator that runs an MSF, possibly multiple times, accumulating the outputs at those times:

runSelfClocking :: SelfClockingMSF m a b
  → MSF (ReaderT DT ime m) a [(DT ime, b)]

A more convenient way of running a self-clocking function, by merging all outputs, as the following:

resample :: (DT ime → [(DT ime, b)]) → b
  → MSF (ReaderT DT ime m) a b
  → MSF (ReaderT DT ime m) a b

The first argument to resample might return only the last sample, the average, or extrapolate it in some other way.

Example Imagine that we want to run a signal function with a maximum time delta of 16ms (60 times per second), possibly sampling multiple times within each simulation step. We could define our self-clocking MSF as follows:

game60FPS' :: MSF (Reader DT ime) a b
  → SelfClockingMSF Identity a b

game60FPS' game = MSF $ λa → do
  dt ← lift $ ask
  let realDT = min dt (1 / 60)
  lastStep = dt < (1 / 60)
  put (realDT, lastStep)
  (output, game') ← lift $ withReader (const realDT)
  (stepMSF game a)
  return (output, game60FPS' game')

Within each step, this MSF puts in the state the actual time delta used for the simulation, and whether it was the last simulation step.

We now run a game with a minimum sampling rate as follows:

game60FPS = MSF (Reader DT ime) Input GameState

This approach keeps the same type signature that realGame had, effectively giving us local time control.

6 Implementation

We have implemented time transformations, time reversal and asynchronous MSF coordination in Dunai [14]. Dunai has been used to implement an API compatible replacement for Yampa capable of running non-trivial FRP games, including commercial games for Android and iOS.

The implementation of synchronous time transformations in Dunai is trivial, as Dunai facilitates transforming the monad and uses a simple continuation-based interface. For example, we can implement our generic function timeTransformMSF as:

```
timeTransformMSF :: Monad m
  ⇒ MSF (ReaderT t1 m) a (t1 → t2)
  → MSF (ReaderT t2 m) a b
  → MSF (ReaderT t1 m) a b

timeTransformMSF timeSF sf = MSF $ λa → do
  (f, timeSF') ← stepMSF timeSF a
  (b, sf') ← withReader f (stepMSF sf a)
  return (b, timeTransformMSF timeSF' sf')
```

The other two time transformation functions are now straightforward, using the type synonym SF a b = MSF (Reader DT ime) a b:

```
timeTransform :: (DT ime → DT ime) → SF a b → SF a b
  ⇒ timeTransform = timeTransformMSF (constant f)

timeTransformSF :: SF a (DT ime → DT ime) → SF a b → SF a b
  ⇒ timeTransformSF = timeTransformMSF
```

The implementation of reversible switching, caching, and all temporal aids requires keeping part of the history (a signal function, an input stream, a global clock) and modifying it to either get elements from the history when times goes backwards or adding new elements when time flows forward.

We have implemented time transformations also in Yampa. While some Yampa combinators check that time deltas are strictly positive and may throw an error otherwise, most functions work irrespectively of the direction of time. In the next section we present an FRP game that uses time transformations and time reversal, and works both in Yampa and in Dunai.

Synchronization of elements running on multiple clocks and time domains is straightforward in Dunai, as shown in the previous section. The only two facilities that are primitive are time-polymorphic variants of possiblyRun and runSelfClocking. The implementation of these features in Yampa would require the extension of output types with additional synchronization information, demonstrating one of the limitations of Yampa.

7 Experience

We have tested our proposal with two games: Lightsmash (Fig. 6) and Pang-a-lambda (Fig. 7).

Lightsmash is a tile-matching board game with discrete changes and continuous animations. The synchronization of these two aspects follows the presentation in Section 5.1. In this game there are two kinds of animations: one when tiles are dropped, and another when a group of tiles needs to be eliminated. The only adaptation required was to consider those two changes, conceptually occurring at the same time, as occurring at successive points in
discrete time, allowing the visual layer to animate one after the other. Otherwise, the discrete part of the game has no knowledge of visualization.

Figure 6. The board game Lightsmash, by Keera Studios\(^3\), running on an Android device.

Pang-a-lambda (Fig. 8) is a 2D platform game in which a player shoots balls bouncing around the screen. When hit by a bullet, balls split in two, until they are too small, when they just disappear. The objective is to eliminate all the balls. This game is available on hackage\(^3\) for desktop, and also works on mobile platforms (iOS, Android) (Fig. 7).

Figure 7. Screenshot of the version of Pang-a-lambda for mobile platforms, developed by Keera Studios.

This game has many opportunities to manipulate the progress of time. In this section we focus on three features: slowing time down, pausing specific elements, and reversing time.

Following the design proposal of previous Yampa games [5], both the player and balls in our game are elements of a type \(\text{ObjectSF}\), a signal function from \(\text{ObjectInput}\) to \(\text{Object}\). The type \(\text{ObjectInput}\) contains the input from the player plus information about previous collisions, needed for the physics and for the gameplay.

\[
\text{data Object} = \text{Object} \\
\begin{aligned}
\text{objectKind} &:: \text{ObjectKind} \\
\text{objectPos} &:: \text{Pos2D} \\
\text{objectVel} &:: \text{Vel2D} \\
\text{...} \\
\end{aligned}
\]

\[
\text{data ObjectKind} = \begin{cases} 
\text{Ball} & \text{Double} \quad \text{-- radius} \\
\text{Player} & \text{Int} \quad \text{-- lives x vulnerable} \\
\text{...} \\
\end{cases}
\]

**Slowing Time Down** The player can decide when time should slow down by pressing a special key. This required modifying the type of the input to our system to know if time should be transformed:

\[
\text{data Controller} = \text{Controller} \\
\begin{aligned}
\text{controllerSlow} &:: \text{Bool} \\
\text{...} \\
\end{aligned}
\]

Slowing time down can only be activated for a limited time. We implemented such a feature with a signal that counts how long it has been in use. When the feature is not being used, the time-slowdown power is “recharged”:

\[
\text{slowDown} :: \text{SF ObjectInput} (\text{DTime} \rightarrow \text{DTime}) \\
\text{slowDown} = \text{proc} (\text{oi}) \rightarrow \text{do} \\
\text{rec let} \\
\text{c} = \text{userInput} \text{oi} \\
\text{slow} = \text{controllerSlow} \text{c} \\
\text{unit} = \text{if} \ | \ (\text{power}') \geq 0 \wedge \text{slow} \rightarrow (-1) \\
\quad | \ (\text{power}') \geq \text{maxPower} \rightarrow 0 \\
\quad | \ \text{otherwise} \rightarrow 1 \\
\text{power} \leftarrow (\text{maxPower}+) \ll \text{integral} \ll \text{unit} \\
\text{let} \text{power}' = \text{min} \text{maxPower} (\text{max 0 power}) \\
\text{diF} = \text{if} \ \text{slow} \wedge (\text{power}') > 0 \ \text{then} \ (0.1+) \ \text{else} \ \text{id} \\
\text{returnA} \ll \text{diF} \\
\text{where} \\
\text{maxPower} = 5
\]

The power to slow time down should decrease over time when it is being used, and increase (“recharge”) when not in use. Time in FRP can be defined as the integral of 1 over time, so our power can be defined as the integral of 1 or \(-1\) over time, depending on whether the player has power left. Power is always capped between 0 and \(\text{maxPower}\). The function applied to the time deltas is \((0.1+)\) if the user wants to slow time down and there is power left, or no transformation otherwise. The use of recursive arrow syntax lets us keep the power bounded within the desired limits.

Our definitions for the player, game objects, etc. remain unchanged by the introduction of this time transformation. However, we must select which elements we want affected by slowing down the clock: while balls, items and the game time limit should be affected by this feature, the player still moves at a normal speed. We did this by manually surrounding those elements that can be slowed down with \(\text{timeTransformSF}\).
We consider this ability to affect time without having to modify the implementations of game signal functions a demonstration of the modularity of our proposal and one of its strengths.

**Pausing Time** We wanted to be able to also pause time under certain circumstances. Stopping time only affects balls and moving blocks, but not the player or the ability to fire bullets. When bullets hit a ball, the ball still splits normally and the new smaller balls start moving, even when others are not. This makes the feature useful for the player, while still challenging and fun.

We introduced this feature in the game by using a time progression function that stops the clock for five seconds. The game clock keeps ticking, only the ball halts in the air:

\[
\text{timeProgressionStop} :: SF \text{ObjectInput} (\text{DTime} \to \text{DTime})
\]

\[
\text{timeProgressionStop} = \text{arr userInput} \gg \text{stopClock}
\]

\[
\text{stopClock} :: SF \text{Controller} (\text{DTime} \to \text{DTime})
\]

\[
\text{stopClock} = \text{switch}
\]

\[
(\text{arr} \ (\lambda c \to \text{if controllerHalt} \ c \text{ then} \ (\text{const} \ 0 \ \text{Event} \ ()))
\]

\[
\text{else} \ (\text{id} \ \text{noEvent})))
\]

\[
(\lambda_ \to \text{switch} \ (\text{const} \ (\text{const} \ 0) \ \&\& \ \text{after} \ (5 \ ()))
\]

\[
(\lambda_ \to \text{stopClock}))
\]

The clock proceeds normally (id) unless the user activates the halting power (controllerHalt). If so, then the signal function switches. After the switch, it constantly halts time (const 0 :: DTime \to DTime), for five seconds (after 5 ()), and then it starts over again.

Our player can now stop the clock and move around while the balls are frozen in the air (Fig. 8a). Balls still react to user events and their corresponding arrow signal functions are being executed but, in terms of simulating their movement, no time has passed. Yet, if a ball is hit while it is frozen, it still splits in two (Fig. 8b).

**Reversing Time** Like slowing time down, reversing time is requested by the user by pressing a key. In our game, time reversal affects the balls but not the player. To make the game harder, we limit the use of time reversal to once per level. This limit is implemented using time directed switches: the game proceeds normally until time reversal is requested, when activated it can be used for up to 5 seconds, and then it always goes towards the future.

\[
\text{reverseTime} :: SF \text{ObjectInput} (\text{DTime} \to \text{DTime})
\]

\[
\text{reverseTime} = \text{switch} \ (\text{constant} \ \text{id} \ \&\& \ \text{reverseActivated})
\]

\[
(\lambda_ \to \text{switch} \ (\text{reverseHeld} \ \&\& \ \text{reverseOver}))
\]

\[
(\lambda_ \to \text{identity}))
\]

\[
\text{reverseHeld} :: SF \text{ObjectInput} (\text{DTime} \to \text{DTime})
\]

\[
\text{reverseHeld} = \text{arr} \ \text{switch}
\]

\[
\text{if} \ \text{controllerReverse} \ \text{userInput} \ \text{io} \ \text{then} \ ((-1)\circ) \ \text{else} \ \text{id}
\]

\[
\text{reverseActivated} :: SF \text{ObjectInput} (\text{Event} ())
\]

\[
\text{reverseActivated} = \text{arr} \ (\text{controllerReverse} \ \text{userInput}) \ \gg \ \text{edge}
\]

\[
\text{reverseOver} :: SF \text{ObjectInput} (\text{Event} ())
\]

\[
\text{reverseOver} = \text{proc} \ (\text{oi}) \to \text{do}
\]

\[
\text{overDueToUser} \leftarrow \text{edge} \leftarrow \text{not} \ \text{controllerReverse} \ \text{userInput} \ \text{oi}
\]

\[
\text{overDueToTime} \leftarrow \text{after} \ (5) \ \leftarrow \ ()
\]

\[
\text{returnA} \leftarrow \text{Imerge} \ \text{overDueToUser} \ \text{overDueToTime}
\]

To make time reversal possible, we surround the elements affected by time reversal in a time transform, and make all the signal functions they use are temporally consistent. For example, bouncing balls are now implemented as:

\[
\text{bouncingBall} :: \text{Double} \to \text{String} \to \text{Pos2D} \to \text{Vel2D} \to \text{ObjectSF}
\]

\[
\text{reverseSwitch} (\text{progressAndBounce} \text{ size bid p0 v0})
\]

\[
(\lambda(p, v) \to \text{bouncingBall size bid p v})
\]

\[
\text{progressAndBounce} :: \text{Double} \to \text{String} \to \text{(Pos2D, Vel2D)}
\]

\[
\to \text{SF ObjectInput (Object, Event (Pos2D, Vel2D))}
\]

\[
\text{progressAndBounce size bid (p0, v0)} = \text{proc} \ (i) \to \text{do}
\]

\[
o \leftarrow \text{freeBall size bid p0 v0} \leftarrow \text{i}
\]

\[
b \leftarrow \text{ballBounce bid} \leftarrow (i, o)
\]

\[
\text{returnA} \leftarrow (o, b)
\]

The function bouncingBall needs an initial position and velocity. Every time there is a bounce, it takes a snapshot of the point of collision and corrected velocity, and starts again. The auxiliary SF ballBounce detects when the ball actually bounces against any other element. It needs the ObjectSF’s input, which contains knowledge of collisions, to detect whether the ball has bounced.

**8 Related Work**

Our work proposes the introduction of time transformations and local control over the sampling step in FRP. The original implementation of Fran [6] did not require causality by construction and allowed time transformations. However, a direct implementation of signals as functions from time to value opens the door to causality violations and memory leaks [4]. Our approach, which operates on time deltas instead of absolute time, and uses time-consistent constructs with limited history, avoids leaks and maintains causality, in the same way that Arrowized FRP does.

Uniti [18] signals can be also sampled at any time, even before the conceptual beginning of the simulation. The implementation of
signals as functions renders problems similar to the original Fran, while our arrowized implementation provides similar versatility while limiting time and space leaks. Like our work, Uniti provides control of the sampling rate and local clocks.

Netwire [19] also uses continuations to implement transformations to input values. Being parameterised over a monad, wires can be used to implement FRP much like Dunai does. However, Netwire does not allow, in principle, running wires backwards in time, and no constructs are provided to guarantee temporal consistency when time flows towards the past.

Varying uses a basic construct isomorphic to Monadic Stream Functions, and our proposal could be implemented in that abstraction as well. Varying is mainly focused on reactivity and stream processing, as opposed to our focus on time transformations and the combination of multiple monads.

Implementations like Elerea [12], Sodium, reactive-banana, as well as the work of Krishnaswami [10], all operate in a step-based fashion with time being a natural number. In these variants the sampling time is the index of a sample in an input stream, and time deltas are of one unit of time. FRPNow! [16] introduces a more complex definition of clock, but progression towards the future occurs in discrete steps and time deltas are of one unit. All these implementations have the limitation that time flows necessarily forward, and there are no ways of progressing towards the past.

Recent work on Testing and Debugging signal functions [15] also records signal functions to travel backwards in time. Signal Functions and inputs are recorded at every point, to guarantee that bugs are fully reproducible. In contrast, our proposal relies on reversing signal functions and could deliver inconsistent results due to numerical rounding errors. Also, this approach is used externally to a program’s definition, while we make use of time transformations and time travel within the game itself.

Games like Braid [8] implement time travel and time reversal. To guarantee that past states are numerically identical when time is reversed, Braid records the game state at every sample, instead of using reversible signal functions or saving only some samples and interpolating intermediate steps.

The synchronization of multiple systems through an explicit notion of clocks is present in Clash [1], Lucid Synchrone [17], recent work by Bärenz [2], and other dataflow languages. In contrast, our framework aims for simplicity for the user, by keeping this notion out of the type level. Furthermore, the use of a completely IO-free framework to synchronize systems running at different rates enables referentially transparent testing as described in [15].

9 Future Work

We started this paper by showing how existing FRP frameworks are constrained in certain situations to avoid time and space leaks. The amount of history preserved by each of these combinators could be introduced at the type level, and leak-avoidance proved statically. We have not specified a formal semantics for the time manipulation constructs we have provided. A denotational semantics for some time transformations was already defined by Courtney [3], but never introduced in Yampa. Unless a notion of branching time is introduced, time reversal remains an operational concern. Currently, the operational semantics of Yampa is not time-consistent when time flows backwards. Monadic Stream Functions coincide with the domain used to describe Yampa’s operational semantics for the monad Reader DTIme. We leave a thorough description of the operational semantics under time reversal as future work.

We have also shown how to coordinate signal functions on different time domains, focusing on hybrid systems. Our proposal would be aided by a first-class notion of change for values, minimizing unnecessary recomputations. Similar attempts suggest that Arrows might be too permissive for incremental FRP, and Generalized Arrows might facilitate the definition of arrows that only run for changes to the input or due to the progress of time.

Acknowledgments

The author thanks M. Bärenz, H. Nilsson, G. Hutton and P. Capriotti for discussions, and for comments on earlier drafts of this paper. The background section is based on prior work by Nilsson [5, 14].

References


4http://hackage.haskell.org/package/varying
5http://hackage.haskell.org/package/sodium
6https://hackage.haskell.org/package/reactive-banana
7http://www.gdcvault.com/play/1012210/