The Essence of Reactivity

Ivan Perez
ivan.perezdominguez@nasa.gov
KBR Inc @ NASA Ames Research Center
Moffett Field, California, USA

Frank Dedden
frank@dedden.net

Abstract

Reactive programming, functional reactive programming, event-based programming, stream programming, and temporal logic all share an underlying commonality: values can vary over time. These languages differ in multiple ways, including the nature of time itself (e.g., continuous or discrete, dense or sparse, implicit or explicit), on how much of the past and future can be referenced, on the kinds of values that can be represented, as well as the mechanisms used to evaluate expressions or formulas. This paper presents a series of abstractions that capture the essence of different forms of time variance. By separating the aspects that differentiate each family of formalisms, we can better express the commonalities and differences between them. We demonstrate our work with a prototype in Haskell that allows us to write programs in terms of a generic interface that can be later instantiated to different abstractions depending on the desired target.

ACM Reference Format:
Ivan Perez and Frank Dedden. 2022. The Essence of Reactivity. *J. ACM* 37, 4, Article 111 (August 2022), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

There are multiple ways to represent systems that change over time. In functional reactive programming (FRP) (Elliott and Hudak 1997; Courtney and Elliott 2001; Nilsson, Courtney, and Peterson 2002) and reactive programming, a system is defined in terms of time-varying values (representing, for example, the mouse position or the state of an interactive application). In stream programming, a system is defined as an output stream (i.e. an infinite sequence), which may depend on external inputs also represented as streams (Ida, Tanaka, et al. 1982). In temporal logic, when the validity of the formula is evaluated, the values of the atomic propositions *over time* are needed. Temporal logics are often connected to runtime monitoring, where a temporal formula is evaluated against an input trace or sequence of values. Some runtime monitoring systems are implemented using ideas from reactive programming, stream programming and dataflow languages (Pike et al. 2010; d'Angelo et al. 2005; Halbwachs et al. 1991; Glory 1989).

The languages or formalisms used to define such systems differ in several ways. For example, in FRP as originally defined, time is notionally continuous, and grows strictly towards the future (Elliott and Hudak 1997). Implementations often limit the ability to sample time-varying values at arbitrary times, for reasons of both efficiency and correctness. In reactive programming, time is often considered discrete and not explicit. Reactive variables can be defined in terms of other variables, such that changes to variables will propagate across the program as needed (Perez and Nilsson 2015). In stream programming, time is also discrete, and not explicitly represented (Ida, Tanaka, et al. 1982). The implementation device for streams is often a lazy infinite list, although implementations may choose to define streams differently depending on the target application (Ida, Tanaka, et al. 1982; Pike et al. 2010; Perez, Dedden, and Goodloe 2020).

Due to the differences among these languages, its not always clear whether it is possible to translate an expression from one language to another, how to do so when possible, and whether there are subtle semantic differences that alter the meaning of a translated expression.

This paper proposes a series of abstractions that capture the essence of different families of temporal formalisms via multiple type classes in Haskell. By providing such classes, implementers can define time-varying systems at a high level, disregarding the specifics of a particular language or target platform. Such high-level interfaces enable for more reuse, since the same definition may be used with different instances, depending on the application. Defining operations at a high-level also allows us to focus on denotation, both in the implementation of such of operations, as well as in the tests for such operations.

We demonstrate our work with a sample application where the same definition can seamlessly be executed locally as a desktop application, compiled to hard-realtime C99, or transformed into a hardware specification.

Specifically, the contributions of this paper are as follows:

- We provide a series of type classes in Haskell that capture different aspects of temporal abstractions.
- We provide a prototype implementation of the instances of such classes for different existing frameworks and libraries.

Authors' addresses: Ivan Perez, ivan.perezdominguez@nasa.gov, KBR Inc @ NASA Ames Research Center, NASA Ames Research Center, Moffett Field, California, USA, 95032; Frank Dedden, frank@dedden.net.

2022. 0004-5411/2022/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

- We demonstrate the benefits of our approach with a sample application that can be realized via a selection of backends or instances of the classes proposed.

2 Background

To better illustrate the problem we seek to solve, let us compare two concrete realizations of two temporal abstractions: Yampa, which implements Functional Reactive Programming (FRP), and SimpleCP, a stream-based language inspired by Copilot, which is a domain-specific language that targets hard-realtime systems.

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP, Yampa and our stream-based language in this section. For further details, see earlier papers on FRP as embodied by Yampa (Courtney, Nilsson, and Peterson 2003; Elliott and Hudak 1997; Nilsson, Courtney, and Peterson 2002), and Copilot (Perez, Dedden, and Goodloe 2020). This presentation draws heavily from the summaries in (Perez, Bärenz, and Nilsson 2016) and (Perez, Dedden, and Goodloe 2020). Readers familiar with Yampa and Copilot can skip to Subsection 2.2.1.

2.1 Functional Reactive Programming

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

$$\text{Signal } \alpha \simeq \text{Time} \longrightarrow \alpha$$

Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an animation could be `Signal Picture` for some type `Picture` representing static images. 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 (Elliott 1998; Nilsson, Courtney, and Peterson 2002). 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. Instead of implementing signals directly as functions from time to values, FRP implementations address these concerns by introducing mechanisms limiting the ability to sample signals at arbitrary points in time.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP (Elliott and Hudak 1997) and Arrowized FRP (Nilsson, Courtney, and Peterson 2002). 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 *signal functions* (i.e., causal functions between signals), connected to the outside world only at the top level. Yampa, a domain-specific language (DSL) we compare in the rest of this section, implements ideas from Arrowized FRP.

2.1.1 Yampa.

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:

$$\text{Signal } \alpha \simeq \text{Time} \longrightarrow \alpha$$
$$\text{SF } \alpha\ \beta \simeq \text{Signal } \alpha \longrightarrow \text{Signal } \beta$$

When a value of type SF α β is applied to an input signal of type Signal α, it produces an output signal of type Signal β. Signal functions are first class entities in Yampa. Signals, however, are not: they only exist indirectly through the notion of a signal function. 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 $[0, t]$.

Programming in Yampa consists of defining signal functions compositionally using a library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the arrow framework proposed by Hughes (Hughes 2000). Some central primitives and combinators are `arr` that lifts an ordinary function to a stateless signal function, sequential composition (`>>>`), parallel composition (`&&&`), and the fixed-point combinator `loop`. In Yampa, they have the following types:

```
arr   :: (a -> b) -> SF a b
(>>>) :: SF a b -> SF b c -> SF a c
(&&&) :: SF a b -> SF a c -> SF a (b, c)
loop  :: SF (a, c) (b, c) -> SF a b
```

Yampa also offers a definition of `Event`, which represents a value that only exists at discrete points in time, as well as advanced notions of switching, which allow for different kinds of dynamism.

2.1.2 Example.

The following Yampa expression defines a signal function that calculates the average value of a list given as input:

```
avg :: SF [Double] Double
avg = arr (\ls -> sum ls / length ls)
```

The following definition calculates the vertical position of a falling ball given an initial position and velocity:

```
posY :: Double -> Double -> SF () Double
posY posY0 velY0 = constant (-9.8)
               >>> integral
               >>> (arr (+ velY0))
               >>> integral
               >>> (arr (+ posY0))
```

In this definition, we use Yampa's own signal function `integral`, which integrates the input signal over time and, in this particular case, has type `integral :: SF Double Double`.

Note that there are alternative ways of writing these expressions that may be easier to read (e.g., using arrow notation (Paterson 2001)). In the current document, we use these expressions simply to exemplify the framework and the key ideas. For alternative ways of writing expressions in Yampa, see (Courtney, Nilsson, and Peterson 2003).

2.2 Stream programming

Stream programming is a programming paradigm in which values are seen as infinite sequences (Ida, Tanaka, et al. 1982). There are multiple approaches towards defining a stream programming language. Ida et al. define streams by a combination of an accumulator, a generator (which calculates the next value of the accumulator), and a selector (which determines the current output based on the current value of the accumulator) (Ida, Tanaka, et al. 1982). Other authors just define a type of infinite lists (Moss and Danner 1997), or leverage an existing list type but define streams as the subset of lists that are infinite (Gibbons and Hutton 1999).

Streams can be defined in terms of other streams by applying transformations to them. Stream-based languages often provide a series of stream-building primitives and combinators (Ida, Tanaka, et al. 1982; Carlsson and Hallgren 1993). Such stream-building expressions are often amenable to formal proofs and equational reasoning (Hutton and Jaskelioff 2011). To ensure that streams can be evaluated efficiently and progressively, and that they are productive (well-defined), stream-based languages often impose restrictions on the dependencies between streams (Caspi and Pouzet 1998; Perez, Dedden, and Goodloe 2020).

2.2.1 SimpleCP. SimpleCP is a stream-based domain specific language embedded in Haskell, inspired by Copilot (Perez, Dedden, and Goodloe 2020). The main use case of SimpleCP is runtime monitoring or runtime verification, a technique in which a system is monitored at runtime to detect violations of expected properties or requirements.

In SimpleCP, a monitoring program is defined by a boolean-carrying stream, which denotes the points in time when a property is violated. SimpleCP streams can carry values of many types, including different kinds of numeric types, *structs* or records, and fixed-length arrays. For example, a stream that carries values of type `Bool` would have type `SimpleCP Bool`, and a stream that carries arrays with exactly 7 elements of type `Bool` would have type `SimpleCP (Array 7 Bool)`.

SimpleCP streams can be defined using a series of primitives and combinators. Primitives include the function `Const :: a -> SimpleCP a`, which defines a stream whose value is the same at all points in time. Combinators include pointwise transformations using a limited set of unary, binary and ternary functions, such as `not :: SimpleCP Bool -> SimpleCP Bool`, which negates a boolean stream, `(==) :: Eq a =>SimpleCP a -> SimpleCP a -> SimpleCP Bool`, which compares two streams for equality. Other combinators include explicit delays `Append :: Int -> SimpleCP a -> SimpleCP a -> SimpleCP a`, which allow us to prepend a limited number of samples from one stream to another, and `Drop :: Int -> SimpleCP a -> SimpleCP a`, which removes values from the head of a stream.

2.2.2 Example. The following SimpleCP expression defines a stream that calculates the average value of an array given as input:

```
avg :: SimpleCP (Array 3 Double) -> SimpleCP Double
avg inputArray = s / l
  where
    s :: SimpleCP Double
    s = inputArray .!! 0
      + inputArray .!! 1
      + inputArray .!! 2

    l :: SimpleCP Double
    l = Const 3.0
```

The following recursive definition produces the Fibonacci sequence:

```
fib :: SimpleCP Word64
fib = Append 2 zeroOne (Drop 1 fib + fib)


zeroOne :: SimpleCP Word64
zeroOne = Append 1 (constant 0)
        $ Append 1 (constant 1)
        $ zeroOne
```

This language is heavily inspired by the Copilot language. In Section 4, we discuss this choice and how similar ideas could be implemented in Copilot.

2.3 Problem statement

In spite of the similarities between these two frameworks, it is not currently possible to directly use a definition from one in the other.

For example, the definition of the `avg` function in Yampa uses `arr` to lift a pure function and apply it to all values of an input signal. In contrast, SimpleCP does not provide any form of lifting that works for all Haskell functions; only a limited subset of functions can be applied pointwise to all values of a stream (in the example above, the `(+)` and `(/)` functions are automatically applied pointwise).

Both libraries are also incompatible in terms of the values that can be carried by a stream or signal function. More specifically, not all Haskell types that can be used in a signal function in Yampa can also be used in a stream in SimpleCP (for example, SimpleCP does not allow arbitrary Haskell functions to be lifted into a stream and applied pointwise to another stream).

Note also that the `integral` signal function from Yampa does not exist in SimpleCP. We could implement a discrete time variant that sums all inputs, but the values would not be the same. This difference specifically points to the fact that time in Yampa is notionally continuous, while, in SimpleCP, it is discrete.

In the following, we present an encoding of different elements of time-variance, allowing us to write definitions that can be used with Yampa, SimpleCP, and other implementations, with minimal or no changes to the code.

3 Solution

In the following, we address the problem identified in prior sections by separating different aspects of temporal formalisms. We first present the ideas from an ideal, or mathematical, point of view, and later discuss how we can implement them.

3.1 Definitions

Time. For the purposes of this work, we consider time to be a set equipped with a notion of equality (=), a total order relation (≤), a measure of distance (δ), and notions of addition (+) and subtraction (−). Note that the set may not be closed under these operations; for example, subtraction may not be defined if the second argument is larger than the first. The relations $\neq$ and $<$ are defined in terms of = and ≤ in the usual way.

Except where otherwise stated, we use the letter T to refer to a generic time set, using subindices when multiple sets are being considered. The set of time is not fixed: different reactive implementations may be based on different sets of time.

Interval. We use the standard definition and notation of intervals and closed intervals. Given a set T with an order relation ≤ and given two elements $\alpha, \beta \in T$, an interval (α, β) is a set defined as $\{t \in T | \alpha < t < \beta\}$. An interval $[\alpha, \beta]$ is a set defined as $\{t \in T | \alpha \leq t \leq \beta\}$. The intervals $(\alpha, \beta]$ and $[\alpha, \beta)$ are defined in an analogous way.

Note that, because an interval is a subset of a set, it implicitly defines a set as well. Also, a set with minimal mn and maximal mx implicitly defines an interval (mn, mx). If the minimal is in the set (minimum), then the interval is closed on that end. The same is true for the maximal (maximum). A set with both minimum mn and maximum mx can be represented as $[mn, mx]$.

Ordered Partition. Intuitively, we define an *ordered partition* of an interval as an ordered sequence of non-overlapping subintervals whose union is the original interval. It is a way of splitting an interval into subintervals.

More formally, we define an order partition of an interval from `mn` to `mx` as a sequence of intervals $e_1, e_2, ..., e_n$ such that, for all i, j, if $i < j$ then $\forall u_i \in e_i, u_j \in e_j, u_i < u_j$, and such that the union $\cup_{i=1}^{n} e_i$ is equal to the set defined by the interval from `mn` to `mx`. Note that the definition applies whether the interval is closed or open on either end.

We define $\bigoplus$ as union of intervals that is only defined if the intervals form an ordered partition (that is, if the resulting union is an interval). Note that, by definition, the intervals in an ordered partition must be non-overlapping. Consequently, for three numbers $\alpha, \beta, \gamma \in T$, the following is ill-typed: $(\alpha, \beta] \bigoplus [\beta, \gamma)$. The addition of intervals cannot leave any holes, so $(\alpha, \beta) \bigoplus (\beta, \gamma)$ is also ill-typed.

We say that the intervals T_1, T_2 are a split of T at t, with $t \in T$, if T_1, T_2 are an ordered partition of T and $\forall t1 \in T_1, t1 \leq t$ and $\forall t2 \in T_2, t \leq t2$. If T_1, T_2 are a split of T, then $T_1 \bigoplus T_2 = T$.

Reactive Entities. We define a reactive entity as a value that varies over time. Given a time set T and a type α, a reactive entity of type α varying over a time domain T is denoted $S_T \alpha$.

We use the semantic function $at : S_T \alpha \longrightarrow T \longrightarrow \alpha$ to give meaning to a reactive entity.

We require reactive entities to be constrained applicatives, that is, applicative functors for some type representing functions, and for limited types of values. More specifically, given a constraint $Types_S$ and a type $Funs_S$, representing, respectively, the constraints on types and the type for functions for a given reactive entity S, and given a function $fun : Funs_S\ \alpha\ \beta \longrightarrow \alpha \longrightarrow \beta$ the following must be defined:

$$
\begin{aligned}
&\text{pure} : \text{Type}_S \alpha \Rightarrow \alpha \longrightarrow S_T \alpha \\
&\text{at}[\![pure\ \alpha]\!] t = \alpha \\
&\text{apply} \quad :(\text{Type}_S\ \alpha, \text{Type}_S\ \beta) \\
&\qquad\quad \Rightarrow S_T (\text{Fun}_S\ \alpha\ \beta) \longrightarrow S_T \alpha \longrightarrow S_T \beta \\
&\text{at}[\![\text{apply sf sv}]\!] t = \text{fun}(at[\![sf]\!] t)(at[\![sv]\!] t)
\end{aligned}
$$

Temporal Disjointness. We would like to operate over portions of reactive entities (that is, reactive entities that are only defined in limited time intervals). To that end, we define notions of temporal disjointness, or a partitioning of a reactive entity into subintervals, and temporal concatenation, or the joining of reactive entities over differet time intervals.

We say that a reactive type constructor S_T supports disjointness if, $\forall t \in T$, there exist left and right projections pL_t and pR_t such that $pL_t \times pR_t : S_T \alpha \longrightarrow S_{T_1} \alpha \times S_{T_2} \alpha$ such that T_1, T_2 are a split of T at t, and such that:

$$
\begin{aligned}
&pL_t : \text{Type}_S \alpha \Rightarrow S_T \alpha \longrightarrow S_{T_1} \alpha \\
&\text{at}[\![pL_t\ s]\!] t = \text{at}[\![s]\!] t \\
&pR_t : \text{Type}_S \alpha \Rightarrow S_T \alpha \longrightarrow S_{T_2} \alpha \\
&\text{at}[\![pR_t\ s]\!] t = \text{at}[\![s]\!] t
\end{aligned}
$$

Temporal Concatenation. We say that a reactive type S_T supports concatenation if, for any orderered partition T_1, T_2 of T, there is a function:

$$\oplus : S_{T_1}\alpha \longrightarrow S_{T_2}\alpha \longrightarrow S_T\alpha$$

$$\text{at}[\![s1 \oplus s2]\!]t = \begin{cases} \text{at}[\![s1]\!]t & \text{if } t \in T_1 \\ \text{at}[\![s2]\!]t & \text{if } t \in T_2 \end{cases}$$

Concatenation and disjointness are duals of each other: their composition on the left or the right is the identity (of either the S_T set or the $S_{t1}\alpha \times S_{t2}$ set). That is, if T_1, T_2 are the split of T at a time t, then the following equalities hold:

$$pL_t(s1 \oplus s2) = s1$$
$$pR_t(s1 \oplus s2) = s2$$

3.2 Encoding

We can implement the ideas discussed earlier in this section via a series of type classes that abstract the different aspects defined.

As discussed before, a reactive entity is a constrained applicative functor, with constraints for both the types of values carried by the reactive value, and the types of functions:

```
class ReactiveFramework r where
  type Fun r :: * -> * -> *
  type Type r a :: Constraint

  pureR  :: Type r a => a -> r a
  applyR :: Type r a
         => r (Fun r a b) -> r a -> r b
```

Note that this class is almost like `Applicative`, and it is equivalent if Fun is `(->)` and Type is `()`, provided that the necessary applicative and functor laws are also satisfied. An example of this is provided in the following.

Example. In the case of Yampa, signal functions that have been applied to an input type are applicative functors, any function can be lifted into a signal function and applied, and there are no constraints on the types carried by signal functions. Consequently, we can define the instance trivially in terms of the applicative interface:

```
instance ReactiveFramework (SF a) where
  type Fun (SF a) = (->)
  type Type (SF a) b = ()

  pureR  = pure
  applyR = (<*>)
```

The analogous definition for SimpleCP is more complicated, since SimpleCP restricts the kinds of functions that can be applied to a stream. The definition of a stream in SimpleCP is as follows:

```
data SimpleCP a where
  Const  :: a -> SimpleCP a

  Apply  :: SimpleCP (Lam a b)
         -> SimpleCP a
         -> SimpleCP b

  Drop   :: Int
         -> SimpleCP a
         -> SimpleCP a

  Append :: Int          -- number of samples
         -> SimpleCP a   -- take samples from this
         -> SimpleCP a   -- prepend samples to this
         -> SimpleCP a
```

The types of functions that can be applied to elements of a stream are defined by the following type Lam:

```
-- | Functions from @a@ to @b@, can be curried to
-- any possitive arity.
data Lam a b where
  Not :: Lam Bool Bool

  Add    :: Num n => Lam n (Lam n n)
  Mul    :: Num n => Lam n (Lam n n)
  Abs    :: Num n => Lam n n
  Signum :: Num n => Lam n n
  Neg    :: Num n => Lam n n
  Div    :: Fractional n => Lam n (Lam n n)

  And :: Lam Bool (Lam Bool Bool)
  Mux :: Lam Bool (Lam a (Lam a a)) -- ite

  ...other constructors...
```

It is now relatively straightforward to give an instance of ReactiveFramework:

```
instance ReactiveFramework SimpleCP where
  type Fun  SimpleCP = Lam
  type Type SimpleCP a = ()

  constR = Const
  applyR = Apply
```

The following definitions are valid in both reactive flavors:

```
true :: ( ReactiveFramework r, Type r Bool )
     => r Bool
true = constR True

false :: (ReactiveFramework r, Type r Bool )
      => r Bool
```

```
false = constR False
```

Some point-wise modifications require adding constraints on the types of functions. For example, we could have defined `false` as:

```
false' :: ( ReactiveFramework r
          , Type r Bool
          , Type r (Lam Bool Bool)
          , Fun r ~ Lam
          )
       => r Bool
false' = applyR (constR Not) true
```

The equivalent for Yampa would have different body and signature:

```
false'' :: ( ReactiveFramework r
           , Type r Bool
           , Type r (Bool -> Bool)
           , Fun r ~ (->)
           )
        => r Bool
false'' = applyR (constR not) true
```

We can, of course, abstract some function operating on values into type classes. For example, it is trivial to define **Num** and **Fractional** instances for both Yampa's SFs and SimpleCP's streams, allowing us to write expressions that are framework independent, such as:

```
twos :: ( ReactiveFramework r
        , Num a
        , Num (r a)
        )
        => r a
twos = 1 + 1
```

□

The type class above does not capture the notion of time variance, but rather the fact that we can produce and combine reactive entities, but there is nothing temporal or time-varying about it yet. In the following, we discuss three key ideas that give temporal meaning to reactive entities: the notion time, the notion of future, and the notion of past.

3.3 Time

To capture the notion of time, we use a dedicated type class:

```
class ReactiveFramework r => TemporalFramework r
  where
    type Time r :: *
```

Although in some frameworks, time is generally implicit, it is useful to define some combinators and primitives to explicitly indicate the nature of time (discrete vs continuous, signed vs unsigned, etc.) by picking a specific type.

Example. Time is one of the key distinguishing features between Yampa and SimpleCP, the former is notionally continuous, the latter is notionally discrete.

In our implementation, we capture that difference via instances that associate different types. In Yampa, time is represented as a double-precision floating-point number, for lack of a more-precise-yet-efficient type that could approximate real time:

```
instance TemporalFramework (SF a) where
  type Time (SF a) = Double
```

In contrast, SimpleCP defines time as follows:

```
instance TemporalFramework SimpleCP where
  type Time SimpleCP = Int
```

□

In the following, we capture the two time transformations we are interested in: time shifts, which relate to the notions of disjointness and concatenation.

3.3.1 Looking into the Past. In the context of FRP and stream programming, concatenation captures the idea that we can pre-pend part of a reactive entity (a signal function or a stream) to another reactive entity. Recall that the definition of $\oplus$ depends on the point of the split of the temporal domain in sub-intervals, as well as the two elements being concatenated. To make our interface easier to work with, we assume that the pre-pended element will be over an interval from 0 to t, and the second element will go from 0 to infinity, and the operations will automatically shift the second reactive entity to the range t to infinity. We first define entities over an interval 0 to t with the following type class:

```
class TemporalFramework r => FiniteFramework r
  where
    type Finite r :: * -> *

    finiteR :: Type r a
            => Time r
            -> r a
            -> Finite r a
```

We can now use the class to pre-pend a fixed amount of history to another reactive entity:

```
class FiniteFramework r => LeakyFramework r where
  appendR :: Type r a => Finite r a -> r a -> r a
```

Example. Yampa already has a type that is commonly used to signal the termination of a signal function, and that is `Event ()`. An `Event` is a value that only occurs at discrete points in time. Events are commonly used in switching, a mechanism in Yampa that allows executing a signal function until some point in time, at which a different signal function will be applied. Events are isomorphic to **Maybe**s, which are the basis for the implementation of Yampa's Tasks, that is, signal functions that may terminate with a result.

We can leverage this common representation to define signals that terminate as follows:

```
data FiniteSF a b = FiniteSF (SF a (b, Event ()))

instance FiniteFramework (SF a) where
  type Finite (SF a) = FiniteSF a

  -- finiteR :: Time (SF a)
             -> SF a b
             -> FiniteSF a b
  finiteR n sx =
      FiniteSF $ sx &&& (time >>> arr p)
    where
      p x = if x >= n then Event () else NoEvent
```

The function finiteR is a smart constructor that, given a signal function and some running time, it produces a finite signal function that terminates at that time. The signal function time is internal to Yampa, and it is a signal function that, at any point, produces the current time.

The implementation of appendR in Yampa can now rely on the existing switching mechanism:

```
instance LeakyFramework (SF a) where
  appendR (FiniteSF sx) sy = switch sx (const sy)
```

We also rely on an auxiliary type in SimpleCP to represent streams that terminate:

```
-- | A finite stream is simply delimited by a
-- point in time.
data FiniteSimpleCP a =
     FiniteSimpleCP
       (Time SimpleCP)
       (SimpleCP a)

instance FiniteFramework SimpleCP where
  type Finite SimpleCP = FiniteSimpleCP
  finiteR = FiniteSimpleCP
```

SimpleCP provides a constructor for appending, and so the type class instance for LeakyFramework only needs to extract elements from a FiniteSimpleCP:

```
instance LeakyFramework SimpleCP where
  appendR (FiniteSimpleCP n sx) sy =
    Append n sx sy
```

The following definitions are valid both in Yampa and in SimpleCP. First, let us define an entity that is false for several time units (which may be continuous or discrete), followed by the reactive entity that is constantly true:

```
append_spec :: forall r
            . ( LeakyFramework r
              , Num (Time r)
              , Type r Bool
              )
           => r Bool
append_spec =
  appendR (finiteR 5 (false :: r Bool)) true
```

We can use recursion to define an entity that alternates between true and false values:

```
alternating_spec :: forall r
                 . ( LeakyFramework r
                   , Num (Time r)
                   , Type r Bool
                   )
                => r Bool
alternating_spec =
    appendR
      (finiteR 2 (true_false :: r Bool))
      alternating_spec
  where
    true_false =
      appendR (finiteR 1 (true :: r Bool)) false
```

Assuming that we can use Num operations on both applied signal functions and streams, we can define a counter as follows:

```
counter :: forall r a
        . ( LeakyFramework r
          , Type r a
          , Num (Time r)
          , Num (r a)
          )
       => r a
counter =
  appendR (finiteR 1 (0 :: r a)) (1 + counter)
```

This counter is a valid definition in both frameworks, and it increments one step for every time unit.

Similarly, we can define a more complex function that calculates the average a reactive entity at different points in time. Note that we do not necessarily have a way to refer to the previous or the next samples, so we use an auxilary function f to convert a discrete number to a time.

```
-- | Takes the average over the next @n@ samples.
smoothN :: forall r a
        . ( LeakyFramework r
          , Type r a
          , Fractional a
          , Fractional (r a)
          )
       => (Int -> Time r) -> Int -> r a -> r a
```

```
smoothN f n sx = total / len
  where
    len   = constR $ realToFrac n
    total = sum steps

    steps = sx :
      [ appendR (finiteR (f n') (0 :: r a)) sx
      | n' <- [1..(n-1)]
      ]
```

□

3.4 Looking into the Future

The other common type of transformation in temporal frameworks is the notion of skipping a sample, or shifting a stream towards the past, thus looking into the future of the stream. In discrete frameworks, such as those based on lists, this can be implemented by dropping samples.

Note that, while the idea of prepending values to a reactive entity is almost always defined, the idea of dropping a portion of a reactive entity is not always defined. More specifically, we can define the projections pL_t and pR_t for some split time point t, but the resulting reactive element pR_t cannot always be "shifted" to a time 0. If the given definition depends on an external input, then we cannot arbitrarily perform time transformations on it and expect to be able to execute it in real time.

We define the following class representing reactive entities for which the right projection pR_t can be shifted to the same time base as the original domain:

```
class TemporalFramework r => NonCausalFramework r
  where
    dropR :: Type r a => Time r -> r a -> r a
```

Example. In Yampa, we can delay a signal function by prepending a different one to it, but we cannot suspend the execution of an SF until a later point, or bring the future of a signal function to the present. Therefore, we cannot implement an instance of NonCausalFramework.

The implementation for SimpleCP just uses the pre-existing constructor Drop:

```
instance NonCausalFramework SimpleCP where
  dropR = Drop
```

Using this instance we can now define a stream that drops samples from a pre-existing one:

```
drop_spec :: ( LeakyFramework r
             , NonCausalFramework r
             , Num (Time r)
             , Type r Bool
             )
          => r Bool
drop_spec = dropR 3 append_spec
```

For example, a function that takes the average of the last three values can be defined in terms of dropR instead of appendR:

```
smooth :: ( NonCausalFramework r
          , Type r a
          , Num (Time r)
          , Fractional (r a)
          )
       => r a -> r a
smooth sx = total / 3
  where
    total = sx + sx' + sx''
    sx'  = dropR 1 sx -- Previous value.
    sx'' = dropR 2 sx -- Value at time t - 2.
```

□

4 Evaluation

We have implemented the ideas presented in this paper in Haskell, for lists, for SimpleCP, for Yampa, and for dunai (based on a notion of Monadic Stream Functions).

Most of the functions and examples presented in the paper work for all frameworks (including dunai), with no changes to the signatures or the implementations. On occasion, we have had to apply specific point-wise functions to reactive entities, in which case we have had to assume a specific implementation of a function type. In many cases, such assumptions could be conveniently abstracted away in a type class.

The use of the generic interface requires adding type constraints to definitions, which, although useful, may sometimes be cumbersome. We cannot expect the typical developer using Yampa, dunai or a similar library to write an application to specify their signatures in terms of generic and overly verbose constraints. To some extent, this burden could be alleviated by means to type synonyms. We see the proposed interface as a suitable mechanism for library authors who would seek to generalize their libraries and make them available for as many targets as possible. In particular, this could apply, for example, to many definitions that currently form part of Yampa, dunai and copilot, but could be reused across libraries.

We initially designed an implementation on top of Copilot based on a more ad hoc interface for temporal transformations. While the code worked in most cases, the interface necessary for Copilot to work was too similar to Copilot itself, and it was difficutl to connect to other reactive abstractions, as well as to give constructs in other reactive languages meaning at a more abstract or mathematical level. We have opted instead for deriving the interface *from first principles*, by first attending to the mathematical properties we expect such a system to obey, and then creating an implementation that could be used for runtime monitoring and generation of C99 (like Copilot), but that is amenable to the API desired. We expect this experience to inform the next version of Copilot, which could present a cleaner design, and allow for constructs that the current version of the language disallows unnecessarily. More specifically, Copilot streams to which a drop is being applied must be guarded by an append operation, but the mathematical definitions in this paper suggest that this is not necessary if we can distinguish between a stream that starts at time 0 and a stream that starts at a future time. We have experimented with a variant of Copilot that uses dependent types to keep track of both the number of samples accumulated in memory (time in the past) and the future start time of a stream, to determine if streams are well-formed prior to compilation. Preliminary experiments indicate that such an encoding is possible in principle. However, the decision of whether such features should be added to the Copilot DSL may also depend on how it will affect the user experience, especially for the most inexperienced users, who may not have a Haskell background.

5 Related Work

The original FRP paper by Elliott and Hudak also proposes a definition of FRP derived from its denotation. Unlike our work, FRP picks a specific representation of time, time is notionally continuous, and allows for arbitrary time transformations (namely, distance-preserving shifts). FRP also introduces both notions of behaviors and events. In contrast, we do not enforce any specific notion of time, we only allow for specific time transformations. We also do not distinguish between different types of time-varying values, and leave those details to the implementations.

Finkbeiner et al. describe a process to automatically generate library-independent FRP control code from Temporal Stream Logic (TSL) specifications (Finkbeiner et al. 2019). There are a number of differences between our respective works. Finkbeiner et al. uses TSL as the source language to be used for specifications, which contain notions of updates, and which has pre-defined notions of predicates and functions. In contrast, our work does not contain notions of updates, and the types of supported functions and values depend on the backend of choice. Finkbeiner et al. also focus their work on FRP, and demonstrate it with three FRP-like libraries (Yampa, Threepenny and ClaSH). Our proposal is not specific to FRP, and can accomodate for other kinds of time-variance. Geier et al. also use TSL to implement an arcade game, and later realize it to an FPGA (Geier et al. 2019). In contrast, we use a source language that is closer in notation to the denotation of reactive languages, does not implement notions of updates and restricts the types of functions and values.

Existing frameworks can transform reactive or temporal languages to multiple targets, including Copilot (Perez, Dedden, and Goodloe 2020) and R2U2 (Schumann, Moosbrugger, and Rozier 2015). These frameworks fix the source language, and later transform into one of multiple target languages. The approach presented in this paper exploits the same ideas defined in tagless final style (Carette, Kiselyov, and Shan 2009), in the sense that the interface presented uses type classes to describe the different aspects of reactivity, and could easily be extended with new functions (as was the case for non-causal reactive entities). Via that mechanism, we provide an *extensible* family of source languages, and also enable giving meaning to those definitions via an extensible list of possible backends.

Perez et al. introduce Bearriver, an API-compatible Yampa replacement built on top of a different abstraction (Perez, Bärenz, and Nilsson 2016). The selection of which library to use for an application is made at link time, by making one library or the other available to the compiler (although one can use the GHC extension `PackageImports` to have both in scope at the same time). Instead, we allow for definitions where other backends and notions of time can be used. Unlike our work, Bearriver definitions are temporally limited to what Yampa and dunai support; in particular, the framework is causal and does not support arbitrary drops. In contrast, we allow for drops in definitions, but the signature of functions with drops requires mechanisms that implement non-causality.

Conal presents an implementation of a GHC compiler backend that can transform Haskell core into different kinds of target languages or representations (Elliott 2017). The use of such a backend makes it possible to compile to hardware circuits, perform automatic differentiation and incremental computation, among others, and provides an alternative way of compiling deeply embedded DSLs. In contrast, the approach presented here uses a tagless encoding of different aspects of reactivity, and then uses a type class instance to pick the specific backend and interpretation. Our approach is easier to implement and requires less knowledge of GHC's plugin system, at the expense of being potentially less versatile. However, the two approaches are not mutually exclusive, and Elliotts's work could be used as one more backend to compile reactive programs.

6 Conclusion

In this paper, we have presented a series of abstractions that capture different aspects of time variance. We first gave a denotational domain for our definitions, and later derived from them a practical implementation in the form of type classes. We have demonstrated that the ideas presented can be implemented using different backends, which could be further extended with other systems such as ClaSH, Bluespec or other FRP flavors.

Our presentation was based on an idealized version of a stream language inspired by, but not exactly like, Copilot. We plan to design an alternative version of the Copilot language that presents a cleaner interface and is easier to compile to multiple backends.

The examples used in this paper represent a small subset of all the library functions available in Yampa, dunai, Copilot and other FRP and reactive programming implementations. We plan to develop a library of generic definitions using the approach described in this paper that can be used with multiple backends.

References

Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2009. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages." *Journal of Functional Programming* 19 (5): 509–43.

Carlsson, Magnus, and Thomas Hallgren. 1993. "Fudgets: A Graphical User Interface in a Lazy Functional Language." In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 321–30.

Caspi, Paul, and Marc Pouzet. 1998. "A Co-Iterative Characterization of Synchronous Stream Functions." *Electronic Notes in Theoretical Computer Science* 11: 1–21.

Courtney, Antony, and Conal Elliott. 2001. "Genuinely Functional User Interfaces." In.

Courtney, Antony, Henrik Nilsson, and John Peterson. 2003. "The Yampa arcade." In *Haskell Workshop*, 7–18.

d'Angelo, Ben, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. "LOLA: Runtime Monitoring of Synchronous Systems." In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, 166–74. IEEE.

Elliott, Conal. 1998. "Functional Implementations of Continuous Modeled Animation." In *Principles of Declarative Programming*, edited by Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, 284–99. Berlin, Heidelberg: Springer Berlin Heidelberg.

———. 2017. "Compiling to Categories." In *Proceedings of the ACM on Programming Languages (ICFP)*. http://conal.net/papers/compiling-to-categories.

Elliott, Conal, and Paul Hudak. 1997. "Functional Reactive Animation." In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 263–73.

Finkbeiner, Bernd, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. "Synthesizing Functional Reactive Programs." In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 162–75.

Geier, Gideon, Philippe Heim, Felix Klein, and Bernd Finkbeiner. 2019. "Syntroids: Synthesizing a Game for FPGAs Using Temporal Logic Specifications." In *2019 Formal Methods in Computer Aided Design (FMCAD)*, 138–46. https://doi.org/10.23919/FMCAD.2019.8894261.

Gibbons, Jeremy, and Graham Hutton. 1999. "Proof Methods for Structured Corecursive Programs."

Glory, Anne-Cecile. 1989. "Vérification de Propriétés de Programmes Flots de Données Synchrones." PhD thesis, Université Joseph-Fourier-Grenoble I.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud. 1991. "The Synchronous Data Flow Programming Language LUSTRE." In *Proceedings of the IEEE*, 79:1305–20. 9. https://doi.org/10.1109/5.97300.

Hughes, John. 2000. "Generalising Monads to Arrows." *Science of Computer Programming* 37 (1): 67–111.

Hutton, Graham, and Mauro Jaskelioff. 2011. "Representing Contractive Functions on Streams." *Submitted to the Journal of Functional Programming*.

Ida, Tetsuo, Jiro Tanaka, et al. 1982. "Functional Programming with Streams." *IPSJ Research Report Programming* 1982 (49 (1982-PRO-003)): 49–56.

Moss, Lawrence S, and Norman Danner. 1997. "On the Foundations of Corecursion." *Logic Journal of the IGPL* 5 (2): 231–57.

Nilsson, Henrik, Antony Courtney, and John Peterson. 2002. "Functional Reactive Programming, Continued." In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, 51–64.

Paterson, Ross. 2001. "A New Notation for Arrows." *ACM SIGPLAN Notices* 36 (10): 229–40.

Perez, Ivan, Manuel Bärenz, and Henrik Nilsson. 2016. "Functional Reactive Programming, Refactored." *ACM SIGPLAN Notices* 51 (12): 33–44.

Perez, Ivan, Frank Dedden, and Alwyn Goodloe. 2020. "Copilot 3." NASA/TM-2020-220587. NASA Langley Research Center.

Perez, Ivan, and Henrik Nilsson. 2015. "Bridging the GUI Gap with Reactive Values and Relations." In *Haskell Symposium*, 47–58. Vancouver, BC, Canada.

Pike, Lee, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. "Copilot: A Hard Real-Time Runtime Monitor." In *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings 1*, 345–59. Springer.

Schumann, Johann, Patrick Moosbrugger, and Kristin Y. Rozier. 2015. "R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems." In *Runtime Verification*, edited by Ezio Bartocci and Rupak Majumdar, 233–49. Cham: Springer International Publishing.