Functional Reactive Programming, Continued*

Henrik Nilsson
Department of Computer Science
Yale University
New Haven, CT 06520-8285
nilsson@cs.yale.edu

Antony Courtney
Department of Computer Science
Yale University
New Haven, CT 06520-8285
courtney@cs.yale.edu

John Peterson
Department of Computer Science
Yale University
New Haven, CT 06520-8285
peterson-john@cs.yale.edu

Abstract

Functional Reactive Programming (FRP) extends a host programming language with a notion of time flow. Arrowized FRP (AFRP) is a version of FRP embedded in Haskell based on the arrow combinators. AFRP is a powerful synchronous dataflow programming language with hybrid modeling capabilities, combining advanced synchronous dataflow features with the higher-order lazy functional abstractions of Haskell. In this paper, we describe the AFRP programming style and our Haskell-based implementation. Of particular interest are the AFRP combinators that support dynamic collections and continuation-based switching. We show how these combinators can be used to express systems with an evolving structure that are difficult to model in more traditional dataflow languages.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*functional languages, data-flow languages, specialized application languages*

General Terms

Languages

Keywords

FRP, Haskell, functional programming, synchronous dataflow languages, hybrid modeling, domain-specific languages

*This material is based upon work supported in part by a National Science Foundation (NSF) Graduate Research Fellowship. We also gratefully acknowledge the support from the Defense Advanced Research Projects Agency (DARPA) MARS program. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF or DARPA.

Haskell'02, October 3, 2002, Pittsburgh, Pennsylvania, USA.

1 Introduction

Many interesting application domains are *reactive* rather than *transformational*. The input to a reactive system is not know in advance, but arrives continuously as the program executes. A reactive system is expected to interleave input and output, producing outputs in response to input stimuli as they arrive. A common approach to implementing reactive systems is to use a *synchronous dataflow* programming language, such as Signal [11], Lustre [4], or Lucid Synchrone [22]. In such languages, programs are built from a small set of primitive *processing elements* and a set of *composition operators*. Complete programs are formed by using the wiring primitives to compose processing elements into a hierarchical network or directed graph structure. The dataflow programming model thus provides a natural form of modularity for many applications, since larger programs are composed hierarchically from smaller components, each of which is itself a reactive program.

Haskell programs can be built in this style using lazy lists, as in the `interact` function. However, this approach is somewhat difficult to use (hence the move from stream based to monadic IO) and does not always yield the modular program designs that are needed for large scale applications. Functional Reactive Programming (FRP) serves to integrate reactivity directly in to the functional programming style while hiding the mechanism that controls time flow under an abstraction layer. Originally developed as part of the Fran animation system [10], FRP has evolved in two distinct directions. One use of FRP is as a "glue language" for combining host language components in ways that have well defined resource use characteristics; here the implementations compile directly to low level code and the use of functions is somewhat restricted. The RT-FRP [26] and E-FRP [27] systems are the result of this research. The other use of FRP utilizes the full power of Haskell. Here, expressiveness is the dominating concern. This approach has been used to create DSLs embedded in Haskell for a number of complex domains such as Frob[20] (robotics), FVision[21] (visual tracking), and Fruit[8] (user interfaces).

The Haskell-based version of FRP has now evolved into AFRP (Arrowized FRP) [8]. In AFRP, we make extensive use of John Hughes's arrow combinators [16] and Ross Paterson's arrow notation [18]. AFRP gives Haskell programmers some, if not most, of the expressive capabilities of synchronous dataflow languages, as well as basic hybrid modeling functionality. Unlike most dataflow languages, *signal functions*, the AFRP analogue to a dataflow processing element, are first class objects. AFRP thus supports higher-order network descriptions, allowing an unusual flexibility in describing structurally dynamic systems.

This paper mainly examines two recent additions to AFRP: continuation-based switching and dynamic collections of signal functions. These additions take the capabilities for describing structurally dynamic systems even further. Continuation based switching allows stateful signal functions to be started (i.e., connected to an input signal), stopped (disconnected), and then resumed again, potentially in a completely different part of the signal function network, and without loosing any information about the internal signal function state. Dynamic collections allow a *varying* number of signal functions to be connected into a network, again without loss of state information when the network structure is changed. There are many application domains in which the dataflow style of programming is natural, but where the highly dynamic structure of systems has limited or prevented its use. We believe that the new AFRP features considerably extends the applicability of the synchronous dataflow style.

In order to ensure an efficient implementation (one that is free of time and space leaks), signals (time-varying values) are not first class entities in AFRP, unlike the signal functions operating on them. This is one of the most substantial design differences between AFRP and earlier versions of FRP, for example Fran. While the operational advantages of a specification free of time and space leaks seems obvious enough, this design carries with it an apparent loss in expressive power: in Fran, the first-class nature of signals meant that the programmer could directly express dynamic collections of signals as a time-varying collection of time-varying values. Continuation-based switching and dynamic collections of signal functions give AFRP similar expressive power, but without compromising operational aspects.

The paper also outlines an experimental feature supporting embedding of subsystems with separate control of the time flow. This allows subsystems to operate at different, user-controllable fidelities, which is important for accurate simulation. It also allows the construction of multi-rate systems, which addresses an important operational concern. Furthermore, embedding offers a restricted but robust form of time transformation, without the performance problems associated with general time transformation encountered in previous FRP implementations.

The rest of the paper is organized as follows. Section 2 first presents basic concepts and AFRP fundamentals. It then continues with a brief introduction to continuation-based switching and signal function collections. Section 3 presents AFRP in greater detail by way of a fairly realistic example, inspired by work done on the FVision language. The emphasis is on the new features of AFRP. This is also where the experimental embedding functionality is discussed. Section 4 then considers the efficient implementation of AFRP in Haskell, including the ramifications of the new AFRP primitives. Section 5 presents related work, and then follows a section on future work and, finally, conclusions.

2 Arrowized Functional Reactive Programming

2.1 Basic Concepts

Two key concepts in FRP are signals (or fluents, time varying values) and functions from signals to signals. In previous Haskell-embedded implementations of FRP, signals were represented using an abstraction called *behavior*. All signals were implicitly connected to a common input whose type varied with the application domain. In Fran, this input contained the keyboard and mouse events while in Frob the system input contained the robot sensor information. This approach lacked modularity: it was impossible to compose components that used different input sources. It also brings an extra complexity to the semantics in the form of a *start time* that represents the time at which a component is "switched on" with respect to the global input signal. Thus the behaviors were fundamentally *signal functions* whose input signal was the system input from the start time forward. However, a primitive (`runningIn`) was provided to regain signal semantics where needed. The result was signals masquerading as signal functions, blurring the distinction between the two distinct notions.

Gradually it became clear that emphasizing the signal function aspect of the behavior abstraction (through combinators which allow e.g. the composition of such functions or computing some form of fixed point), while relegating signals to second class status, appeared to have a number of significant operational advantages and clarify semantical issues. There were also notational advantages in that a syntactic distinction between signals and signal functions was made. Taking this approach allowed us to recast FRP as an instance of John Hughes's Arrow framework, which directly gave us firm theoretical underpinnings and allowed us to leverage Ross Patterson's work on arrow syntax.

AFRP uses the `SF` type to denote signal functions. Conceptually, this is a function from *Signal* to *Signal*:

$$\mathbf{SF}\ a\ b \;=\; Signal\ a \;\rightarrow\; Signal\ b$$

where

$$\mathbf{Signal}\ a \;=\; Time \rightarrow\ a$$

for real-valued time. We reiterate that this is just a conceptual model: only signal functions are first class entities; signals only exist indirectly, through the signal functions. In particular, there is no type `Signal` in AFRP.

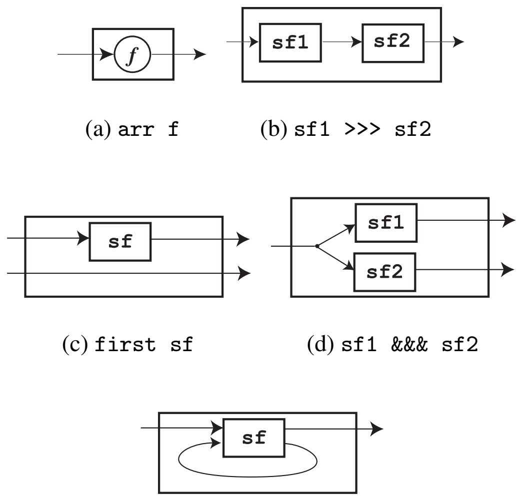
We can think of signals and signal functions using a simple circuit analogy. Line segments (or "wires") represent signals, with arrows indicating the direction of flow. Boxes (or "components") represent signal functions. Signal functions are not curried: multiple input or output signals are tupled together.

In order to ensure that signal functions are properly 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 $[0,t]$. All primitive signal functions in AFRP are causal and all signal function transformations preserve causality.

2.2 Discrete and Continuous Time

The essential abstraction that our system captures is *time flow*. As with the original FRP system, there are two distinct semantic domains for expressing the progress of time: *continuous time* and *discrete time*.

In a program that uses continuous time semantics, the execution of a program is an approximation to a continuous function of time. Continuous time is especially useful in applications that model or interact with the physical world. An advantage of this approach is that the mechanism by which the discrete approximation to the ideal semantics is obtained is hidden from the user. This removes a very operational aspect of programming and gives the system freedom to choose the best possible approximation to the continuous semantics. With continuous time there is no notion of "time steps"



(a) `arr f` (b) `sf1 >>> sf2`

(c) `first sf` (d) `sf1 &&& sf2`

(e) `loop sf`

Figure 1. Core Primitive Signal Functions

as a user-visible abstraction. Some constructs require conceptually infinitesimal delays, but there is no general notion of the "next" or "previous" time. Continuous time semantics are common in languages such as Simulink [24] or Modelica [17].

While continuous time is a useful abstraction, it can be difficult to control the computational resources of a program when the mechanism controlling time flow is opaque to the programmer. This can lead to systems in which it is difficult to understand the operation of a program or control exactly how computational resources are allocated. Many systems use discrete time semantics in which time is advanced in user-visible increments (steps). Here it is possible to reason precisely about program semantics and computational resources, but the program becomes less abstract. It also makes composition more difficult when subsystems operate at different rates.

FRP has traditionally bridged the gap between the continuous and discrete worlds. Although this work preserves this mixed semantic domain, we have paid more attention to the operational side of the implementation. Some of the abstractions we present here are used mainly to handle programming in the discrete time style. For example, the distinction between immediate and delayed switching is more useful in discrete time systems, and the way we currently handle embedding breaks the basic abstractions for representing time flow and is thus appropriate only for discrete time systems. While we would encourage users to use the continuous part of our system, we realize that it is often necessary to address operational aspects, and thus we consider it of prime importance to provide the facilities to do so.

2.3 Core Primitives

The core primitives for composing signal functions are shown in figure 1. These are all standard arrow combinators and have simple, intuitive definitions that follow directly from the diagrams. For example, `arr` is point-wise application:

```
arr :: (a -> b) -> SF a b
arr f = \s -> \t -> f (s t)
```

and `(>>>)` is just reverse function composition:

```
(>>>) :: SF a b -> SF b c -> SF a c
sf1 >>> sf2 = \s -> \t -> (sf2 (sf1 s)) t
            = sf2 . sf1
```

Note that the above definitions are in terms of the conceptual continuous-time model. The actual implementation is based on discrete sampling and is explained in detail in section 4.

The other three primitives provide mechanisms for specifying arbitrary wiring structures, using pairing to group signals. We have omitted the definitions, since they follow naturally from the above wiring diagrams, but the type signatures are as follows:

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

Although signals are not first class values in AFRP, Paterson's syntactic sugar for arrows [18] effectively allows signals to be named. This eliminates a substantial amount of plumbing, resulting in much more legible code. In this syntax, an expression denoting a signal function has the form:

`proc` *pat* `->` `do` [`rec`]
    pat_1 `<-` $sfexp_1$ `-<` exp_1
    pat_2 `<-` $sfexp_2$ `-<` exp_2
    ...
    pat_n `<-` $sfexp_n$ `-<` exp_n
    `returnA -<` *exp*

The keyword `proc` is analogous to the λ in λ-expressions, *pat* and pat_i are scalar patterns binding signal variables point-wise by matching on instantaneous signal values, *exp* and exp_i are scalar expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined point-wise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound point-wise in pat_i. The overall input to the signal function denoted by the `proc`-expression is bound by *pat*, and its output signal is defined by the expression *exp*. The signal variables bound in the patterns may occur in the scalar expressions, but *not* in the signal function expressions ($sfexp_i$). If the optional keyword `rec` is used, then signal variables may occur in expressions that textually precedes the definition of the variable, allowing recursive definitions (feedback loops). The syntactic sugar is implemented by a preprocessor which expands out the definitions using only the basic arrow combinators `arr`, `>>>`, `first`, and, if `rec` is used, `loop`.

For a concrete example, consider the following:

```
sf = proc (a,b) -> do
    c1 <- sf1 -< a
    c2 <- sf2 -< b
    c  <- sf3 -< (c1,c2)
    d  <- sf4 -< b
    returnA -< (d,c)
```

Here we have bound the resulting signal function to the variable `sf`, allowing it to be referred by name. Note the use of the tuple pattern for splitting `sf`'s input into two "named signals", `a` and `b`. Also note the use of tuple expressions for pairing signals, for example for feeding the pair of signals `c1` and `c2` to the signal function `sf3`.

2.4 Stateful Primitives

Functions that are converted to signal functions by `arr` are *stateless*: the value of the output signal at time t depends only on the value of the input signal at t. AFRP also provides signal functions that are *stateful*. Such primitives produce an output signal that may depend not just on the input signal at t, but at all times on $[0,t]$.

A basic stateful primitive is the *integral*[1]:

```
integral :: Fractional a => SF a a
```

The `integral` primitive computes the time integral of its input signal. Integrating `constant 1` yields the local time:

```
localTime :: SF a Time
localTime = constant 1.0 >>> integral
```

2.5 Events

An *event* is something which is understood as occurring at a single, discrete point in time, having no duration, such as a mouse button click or a signal exceeding some threshold. While many aspects of reactive systems are naturally modeled as continuous signals, i.e. total functions on a continuous interval of time, sequences of events are best reflected by a discrete-time signal defined only for the points in time at which the events occur. To model such a discrete-time signal, we introduce the `Event` type:

```
data Event a = Event a -- an event occurrence
             | NoEvent -- a non-occurrence
```

We call a *Signal* `(Event T)` for some type T an *event stream*. It captures the idea of a discrete-time signal in that the value of the signal is `(Event` v`)` for some value v only at the time of an event occurrence (the signal is “defined”), and `NoEvent` otherwise (the signal is “undefined”). The value v carried with an event may be used to convey extra information about the occurrence.

The `Event` type is isomorphic to `Maybe` and is overloaded in the same way, providing instances in `Functor` and other type classes. The function `tag` binds a given value to an event occurrence:

```
tag :: Event a -> b -> Event b
tag e t = fmap (const t) e
```

Events can be *merged* in a number of different ways. The `lMerge` function favors the left event in case of simultaneous occurrences:

```
lMerge :: Event a -> Event a -> Event a
lMerge (Event v1) e2 = (Event v1)
lMerge NoEvent    e2 = e2
```

while `mergeBy` combines events that occur at the same time using a user-supplied function:

```
mergeBy :: (a->a->a)->Event a->Event a->Event a
mergeBy f (Event v1) (Event v2) = Event (f v1 v2)
mergeBy f (Event v1) e2         = Event v1
mergeBy f NoEvent    e2         = e2
```

These functions are usually applied point-wise on event streams yielding a merged event stream.

[1] The given signature is simplified: the real AFRP primitive supports integration of vectors.

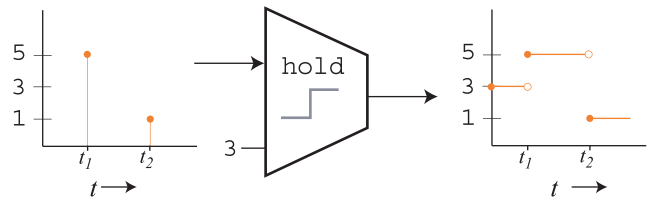


Figure 2. The hold function

Event streams may be synthesized from continuous signals. For example,

```
edge :: SF Bool (Event ())
```

is an *event source* that generates an event when a boolean signal changes from false to true. Note that this signal function is stateful.

Another stateful primitive is `hold`, a signal function that turns an event stream into a continuous signal by remembering the the most recent value of an event:

```
hold :: a -> SF (Event a) a
```

Figure 2 summarizes the semantics of `hold`.

2.6 Switching and Signal Collections

The AFRP switching primitives enable composition of signal functions by temporal sequencing: at discrete points in time, signalled by events, controlled is switched from one signal function to another. The simplest switching primitive is `switch`:

```
switch :: SF a (b,Event c)->(c->SF a b)->SF a b
```

Informally, `switch sf sfk` behaves as follows: At time $t = 0$, the initial signal function, `sf`, is applied to the input signal of the `switch` to obtain a pair of signals, *bs* (type: *Signal* `b`) and *es* (type: *Signal* `(Event c)`). The output signal of the `switch` is *bs* until the event stream *es* has an occurrence at some time t_e, at which point the event value is passed to `sfk` to obtain a signal function `sf'`. The overall output signal switches from *bs* to `sf'` applied to a suffix of the input signal starting at t_e.

The effect of switching may be *observed* at the output of the overall signal function either immediately at the time of the switching event or strictly after that time. That is, the question is whether the output value at the time of the event is given by the last output value of the signal function being switched out, or by the first value of the signal function being switched in. The choice depends on the application: delayed observation of the output from the newly started signal function is sometimes needed to break cyclic dependencies among signals. The AFRP library has two versions of every switching function, one with and the other without the delay.

The `rSwitch` function is similar to `switch`, except that the switch is *recurring*: it will switch on every occurrence of an event stream connected to its input, not just on the first occurrence. Given an initial subordinate signal function, defining the overall output signal, `rSwitch` creates a signal function that has two inputs: one for the input signal passed in to the current subordinate signal function, and the other for switching events, each carrying a new subordinate signal function to replace the previous one:

```
rSwitch :: SF a b -> SF (a,Event (SF a b)) b
```

The fact that events can carry signal functions nicely illustrates their first class nature: signal functions are just like any other values.

Signal function collections are used to condense groups of signal functions into a single signal function. The collection type is arbitrary; it needs only to be in the `Functor` class. The simplest operation groups signal functions sharing a common input:

```
parB :: Functor col=>col (SF a b)->SF a (col b)
```

The `B` suffix of `parB` signifies that, in the resulting signal function, the input signal is *b*roadcast to all signal functions in the collection. Sometimes, however, it is desirable to specify a *routing function* that determines the input signal to be delivered to each element of the collection; the `par` primitive is provided for this purpose:

```
par :: Functor col =>
  (forall sf . (a -> col sf -> col (b, sf)))
  ->  col (SF b c)
  ->  SF a (col c)
```

This primitive will apply its first argument (the routing function) point-wise to the external input signal and the collection of signal functions to obtain a collection of (input sample, signal function) pairs. Each signal function in the collection can thus be given its own input sample. One way of thinking about the routing function is as a way of controlling *perception*: the routing function determines the view of the world seen by each element of the collection. This can be used to, for example, allow a set of objects to perceive only their closest neighbor, or only those that are located in some specified field of view.

While `par` and `parB` give us basic facilities for maintaining collections of signal functions, the collections are fundamentally *static*: we cannot add or remove signal functions from the collection. For *dynamic* collections, we provide `pSwitch`, which allows the collection to be updated in response to events:

```
pSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b, sf)))
  -> col (SF b c)
  -> SF (a, col c) (Event d)
  -> (col (SF b c) -> d -> SF a (col c))
  -> SF a (col c)
```

The first two arguments are the routing function and initial collection, just as in `par`. The third argument is a signal function that observes the external input signal and the output signals of the collection, producing an event that triggers collection update. When the event occurs, the collection is reshaped by a function that produces a new collection given the value of the event.

The argument to the collection update function is of particular interest: it captures *continuations* of the running signal functions. Since the continuations are plain, ordinary signal functions, they can be resumed, discarded, stored, or combined with other signal functions. This gives AFRP a capability that has not previously been available in FRP: the ability to "freeze" running signal functions, turning them back into first class values (with preserved state, of course). In contrast, the collection argument passed to the routing function contains running signal functions. Due to the rank-2 universal quantification of `sf`, nothing much can be done with these, except passing them on, effectively giving them second class status.

`pSwitch` is a "switch once" combinator; another version, `rpSwitch` uses a recurring switch in the manner of `rSwitch`.

Although these switching combinators may appear complex and numerous, there is an underlying structure and relationship between all of the different switchers. For example, `rSwitch` is defined in terms of `switch` using a simple recursive definition:

```
rSwitch :: SF a b -> SF (a, Event (SF a b)) b
rSwitch sf = switch (first sf) rSwitch'
  where
    rSwitch' sf = switch (sf***notYet) rSwitch'
```

In the above definition, `(***)` is a derived combinator for parallel composition of two arrows, and `notYet` is a primitive signal function that suppresses an event occurrence at time $t = 0$, but otherwise behaves as the identity signal function:

```
(***)  :: SF a b -> SF c d -> SF (a,c) (b,d)
notYet :: SF (Event a) (Event a)
```

The `pSwitch` primitive is the most general of the switchers: all other switchers can be defined in terms of `pSwitch`.

3 Example: Traffic Surveillance by Visual Tracking

In this section we present an example that showcases various aspects of AFRP, in particular its capability to handle dynamically changing system configurations and the role first class signal function continuations plays in that context. The setting of the example is an imagined Unmanned Aerial Vehicle (UAV) for traffic surveillance. We will focus on describing a small part of its control system concerned with the detection of tailgating[2] among the vehicles currently in the field of vision. Figure 3(a) shows the UAV over a section of highway, and figure 3(b) the structure of the tailgating detector when three cars are in view. As cars enter or leave the field of view, or overtake, the structure of the tailgating detector must change accordingly.

We chose this example because the dynamically changing situation on the road provides a compelling case for employing the dynamic features of AFRP. Since our primary concern is AFRP, not the fine details of the example as such, we will make a number of simplifying assumptions to keep the example concise. These simplifications will not affect the essential structure of the solution.

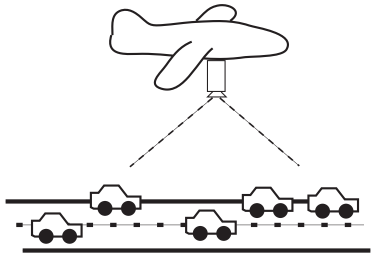
3.1 Interfacing to the Physical World

In our example, the UAV is equipped with a video camera providing a bird's-eye view of the highway below it. The video processing identifies and tracks individual ground vehicles, which we will just refer to as "cars". See figure 3.

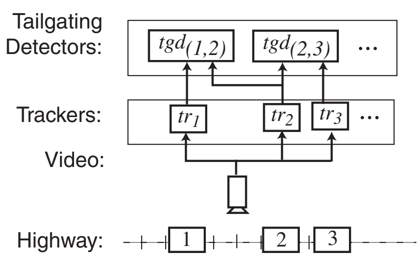
Prior work in domain specific languages based on FRP has used the XVision2 library. This has been imported into Haskell, and together with AFRP it forms the basis for FVision, a language for visual tracking [21]. For this example, we will thus assume that the functionality for creating trackers for tracking of an individual car in a video stream is available.

Once initialized, a tracker is just a signal function from video to a suitably abstract representation of a car. It is important to realize that trackers typically maintain internal state pertaining to the object being tracked, such as the position of the object in the previous

[2]To drive dangerously close behind another vehicle.



(a) UAV over highway.



(b) Tailgating detector.

Figure 3. UAV for traffic surveillance.

video frame, making it possible to find the position in the current frame efficiently. A tracker is thus a *stateful* signal function.

To keep things simple, we will assume that it is enough to regard the highway as a one-dimensional line. The position and velocity of a tracked car are thus also one-dimensional. Restricting tracking information to one dimension makes the tailgating detection somewhat inaccurate, since passing slowly in an adjacent lane will be detected as tailgating[3]. However, the main point here is the *structure* of the solution, and this structure would remain the same even if a more accurate representation of world were used. Car positions are measured relative to the point directly below the UAV; the positive direction coincides with the direction of travel of the UAV and the cars directly below it. The car velocity, however, is assumed to be absolute *ground* velocity, not velocity relative to the UAV. These assumptions lead to the following type definitions:

```
type Position = Double          -- [m]
type Velocity = Double          -- [m/s]
type Car = (Position, Velocity)
```

A car tracker has the following signature:

```
type CarTracker = SF (Video, UAVStatus)
                     (Car, Event ())
```

Here, `Video` is an abstract type representing a video frame. `UAVStatus` carries information such as current height and ground velocity of the UAV, making it possible for the tracker to figure out the scale of the received images and compute the ground position and velocity of tracked objects. The event output indicates lost tracking. Once tracking is lost, the last known position and velocity becomes the final, constant value of the `Car` output signal.

[3]Driving in another driver's blind-spot will also be detected as tailgating, but we consider this to be a feature, not a bug.

3.2 Simultaneous Tracking of Multiple Vehicles

Given trackers capable of tracking individual cars, we now have to devise a way of tracking multiple cars simultaneously. This involves running a number of trackers in parallel, taking into account that the number of tracked cars will change dynamically as cars enter and leave the field of vision. We assume that the arrival of a new car is signalled by an event carrying a tracker. Cars leaving the field of vision will cause the corresponding tracker to lose tracking, at which point we remove it from the collection of trackers. Furthermore, each tracked car will be associated be with a distinct identifier, giving each car in the system an identity. This yields the following signature for the Multi Car Tracker (`mct`):

```
type Id = Int
mct :: SF (Video, UAVStatus, Event CarTracker)
          [(Id,Car)]
```

Recall that trackers are *stateful* signal functions. Thus we need to add trackers to and delete trackers from the collection of running trackers without disrupting other trackers in the collection. The AFRP parallel switchers allow us to achieve this by making the entire collection of subordinate continuations available at the point of switching. This allows us to add and/or delete trackers from the collection, and then resume. While there are a number of parallel switching combinators provided by AFRP, we use `pSwitch` (see section 2.6 for the type signature) because it allows the signal function that controls switching to observe the output of the entire collection. This will enable us to remove a tracker from the collection when a lost tracking event is detected.

In order to use `pSwitch`, we first define a suitable collection type:

```
data MCTCol a = MCTCol Id [(Id, a)]
instance Functor MCTCol where
  fmap f (MCTCol n ias) =
    MCTCol n [ (i, f a) | (i, a) <- ias ]
```

The extra field of type `Id` is used for generating distinct identifiers.

We can now define `mct`:

```
mct = pSwitch route (MCTCol 0 [])
              addOrDelCTs
              (\cts' f -> mctAux (f cts'))
      >>> arr getCars
  where
    mctAux cts =
      pSwitch route cts
              (noEvent --> addOrDelCTs)
              (\cts' f -> mctAux (f cts'))
    route (v,s,_) = fmap (\ct -> ((v,s),ct))
```

The routing function `route` simply passes on the `Video` and `UAVStatus` part of the input to each running tracker. The event-generating signal function `addOrDelCTs`, defined below, emits an event, which in turn will cause a switch, whenever trackers need to be added or removed. This event carries a function which will perform the required mutation of the collection of signal function continuations when applied in the continuation argument passed to `pSwitch`. The `pSwitch` continuation argument is invoked at the time instant of the switching event. Thus, the signal function started by the continuation will initially see the same instantaneous input as the switch that switched into it. In this case, part of that input could be an event indicating the arrival of a new car. Thus we need to ensure that this event does not cause a new switch immediately, which would lead to an infinite loop. That is the purpose of the construct

```
noEvent --> addOrDelCTs
```

which overrides the the initial output from addOrDelCTs with noEvent, i.e. ensures that there will be no event occurrence at (local) time 0.

The definition of addOrDelCTs is straightforward:

```
addOrDelCTs = proc ((_, _, ect), ces) -> do
  let eAdd = fmap addCT ect
  let eDel = fmap delCTs
                 (catEvents (getEvents ces))
  returnA -< mergeBy (.) eAdd eDel
```

Any external tracker creating event is tagged with a function that will add the carried tracker to the collection of trackers. Similarly, from a list of events carrying the identities of trackers that has lost tracking, we form an event carrying a function that will remove those trackers from the collection. Finally we merge the two resulting event signals into a single signal using mergeBy. The function supplied to mergeBy is used to resolve the conflict in the case of simultaneous event occurrences. In this case we just use ordinary function composition to join the addition and deletion functions.

3.3 Detection of Tailgating

We now turn our attention to defining what the system's notion of tailgating should be. Since tailgating involves two cars, we will capture this notion using a binary predicate. However, to avoid capricious judgements, we want to evaluate the behavior of a potential tailgater over some amount of time, rather than at a single instant. Tailgating thus becomes a *temporal* predicate, which we can model with the following signature:

```
tailgating :: SF (Car, Car) (Event ())
```

The following criteria are good enough for defining tailgating for the purpose of our example:

1. car 1, the potential tailgater, is behind car 2;
2. the absolute speed of car 1 is greater than 5 m/s;
3. the relative speed of the cars is within 20 % of the absolute speed;
4. car 1 is no more than 5 s behind car 2; and
5. over an interval of 30 s, the average distance between the cars is less than 1 s.

Note that we measure distance in seconds, i.e. distances are normalized by dividing by the absolute speed. Also, to avoid reporting tailgating in a situation where the traffic is at a virtual standstill, we insist on a certain minimum speed.

Let us proceed ground up. We first define a signal function for computing the average distance (in seconds) between two cars. This is just a matter of integrating the normalized distance ($nd(t)$) and divide by the time passed:

$$\overline{nd} = \frac{1}{t}\int_0^t nd(t)\,\mathrm{d}t$$

However, we have to be careful with what we mean by "average" at time 0, when no time has passed. Since, as long as f is continuous at $t = 0$,

$$\lim_{t\to 0}\frac{1}{t}\int_0^t f(t)\,\mathrm{d}t = f(0)$$

we just return the instantaneous normalized distance at that point. These equations are rendered as follows in AFRP:

```
avgDist :: SF (Car, Car) Time
avgDist = proc ((p1, v1), (p2, v2)) -> do
  let nd = (p2 - p1) / v1
  ind <- integral  -< nd
  t   <- localTime -< ()
  returnA -< if t > 0 then ind / t else nd
```

Next, we define a temporal predicate which repeatedly evaluates the average distance over 30 second periods, and emits an event if the average distance was less than 1 s.

```
tooClose :: SF (Car, Car) (Event ())
tooClose = proc (c1, c2) -> do
  ead <- recur (snapAfter 30 <<< avgDist)
         -< (c1, c2)
  returnA -< (filterE (<1.0) ead) ‘tag‘ ()
```

This is a bit ad hoc: it would be better to compute a running average over the last 30 s. But that would require the use of a 30 s delay, and AFRP currently does not support that kind of time shifting operation.

Finally we insist that the instantaneous conditions for tailgating (the first four criteria) be satisfied before allowing tooClose to determine the outcome of the test.

```
tailgating = provided follow tooClose never
  where
    follow ((p1, v1), (p2, v2)) =
      p1 < p2 && v1 > 5.0
      && abs ((v2 - v1) / v1) < 0.2
      && (p2 - p1) / v1 < 5.0
```

The combinator provided

```
provided :: (a->Bool)->SF a b->SF a b->SF a b
```

applies its first argument point-wise to the input signal and switches into its first or second argument as the lifted predicate changes between True and False, respectively. The combinator never

```
never :: SF a (Event b)
```

is an event source that never has an occurrence.

3.4 Detection of Tailgating in a Group of Vehicles

In order to check for tailgating among all cars in view, we have to run a tailgating detection predicate for each pair of potential tailgater and tailgatee. Under our assumption of a one-dimensional world, this just amounts to running a predicate for each pair of adjacent cars in the field of view; see figure 3(b). We are going to use a parallel switch to maintain the collection of tailgating detectors. Thus we need a suitable collection type. Since each tailgating detector is related to two cars, a collection where each element is labelled by a pair of car identifiers will fit the bill.

```
newtype MTGDCol a = MTGDCol [((Id,Id), a)]
instance Functor MTGDCol where
  fmap f (MTGDCol iias) =
    MTGDCol [ (ii, f a) | (ii, a) <- iias ]
```

The signature for the Multi Tailgating Detector (`mtgd`) is as follows:

```
mtgd :: SF [(Id, Car)] (Event [(Id, Id)])
```

It receives a time-varying list of cars from the currently running car trackers where each car is associated with its identifier. When an instance of tailgating is detected, that is signalled by emitting an event identifying the tailgater and tailgatee. Though unlikely, it could be the case that two or more instances of tailgating are detected simultaneously. Thus each tailgating event actually carries a list of tailgater and tailgatee identifier pairs.

Since we need to run a tailgating detector for each pair of adjacent cars, we need to monitor the input list of cars and detect any change in the adjacency relation; i.e. cars entering or leaving the field of view or overtaking each other. Any such change is an event that should cause a switch into a new configuration. Sorting the list into order by car position facilitates change detection as well as routing the right pair of cars to each tailgating detector. As to which flavor of parallel switch to use, note that switching in this case does not depend on the output from the subordinate signal functions. Thus the recurring, parallel switch, `rpSwitch`, is suitable:

```
rpSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b, sf)))
  -> col (SF a b)
  -> SF (a, Event (col (SF b c)->col (SF b c)))
        (col c)
```

Using `rpSwitch`, `mtgd` can be defined as follows:

```
mtgd = proc ics -> do
  let ics' = sortBy relPos ics
  eno  <- newOrder -< ics'
  etgs <- rpSwitch route (MTGDCol [])
          -< (ics', fmap updateTGDs eno)
  returnA -< tailgaters etgs
  where
    tailgaters ::
      MTGDCol (Event ()) -> Event[(Id,Id)]
    tailgaters (MTGDCol iies) =
      catEvents [e `tag` ii | (ii,e) <- iies]
```

The signal function `newOrder` detects changes in the adjacency relation:

```
newOrder :: SF [(Id, Car)] (Event [Id])
```

It works by simply comparing the order among the cars at the previous time instant with the current order. Whenever they are not the same, an event is generated carrying a list detailing the new order.

The purpose of `route` is to route each pair of adjacent cars to the corresponding tailgating detector. We will order the collection of tailgating detectors by car position. Thus the routing is just a matter of zipping a list of of adjacent cars with the list of running tailgating detectors. Here is the code:

```
route ics (MTGDCol iitgs) = MTGDCol $
  let cs = map snd ics
  in [ (ii, (cc, tg))
     | (cc, (ii, tg)) <- zip (zip cs (tail cs))
                             iitgs]
```

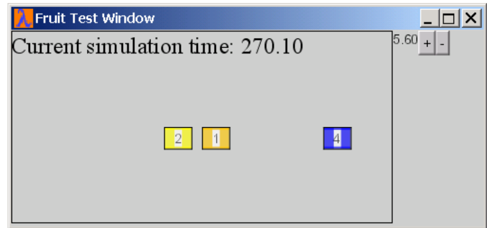


Figure 4. Animating the Tailgating Detector

Then we need to tag each change event with a function that updates the collection of tailgating detector continuations to reflect the new situation. Recall that each tailgating detector is a stateful signal function. Thus, if two cars that were adjacent before a change event also are adjacent after that event, then the tailgating detector pertaining to these two cars should be kept running across the event. On the other hand, tailgating detectors for cars no longer adjacent must be terminated. Finally we also need to start a tailgating detector for any new pair of adjacent cars. The function `updateTGs` computes an updated collection of tailgating detectors given the new order and the collection of continuations for the previously running detectors.

```
updateTGDs is (MTGDCol iitgs) = MTGDCol
  [ (ii, maybe tailgating id (lookup ii iitgs))
  | ii <- zip is (tail is) ]
```

Note the use of `lookup` for finding the continuations for the detectors that should keep running. Whenever `lookup` fails, that means that we have a new pair of adjacent cars, and therefore a new instance of the tailgating detector should be started.

Finally, we can tie the individual pieces together into a signal function that finds tailgaters:

```
findTailgaters ::
  SF (Video, UAVStatus, Event CarTracker)
     ([(Id, Car)], Event [(Id, Id)])
findTailgaters = proc (v, s, ect) -> do
  ics  <- mct  -< (v, s, ect)
  etgs <- mtgd -< ics
  returnA -< (ics, etgs)
```

3.5 Visualization

We have applied the functional reactive programming model to a wide variety of problem domains, such as control systems, computer vision and graphical user interfaces. Using AFRP as a common framework for these disparate domains has a number of benefits, such as enabling us to use a common vocabulary across domains, development of a library of domain-independent data- and time-flow patterns, and easy composition of systems with components drawn from different domains. For example, although the tailgating detector is really a problem from the domain of control systems, we were able to use Fruit, our AFRP-based graphical user interface toolkit [8], to quickly and easily develop an animated visualization of this application. A screenshot of this interface is shown in figure 4. The animated graphical user interface took less than two hours to develop, did not require any modification to the original tailgating detector or highway simulation, and is about fifty lines of code.

To start, we require a function that, given a car's `Id` and whether or not it is a tailgater (a `Bool`), returns a `Picture` of the car:

```
carPic :: Int -> Bool -> Picture
carPic i isTG =
  let
    cols = [red, orange, yellow, green,
            blue, cyan, magenta, pink]
    carColor = cols !! (i `mod` (length cols))
    drawBorder =
      if isTG
        then picFlatBorder (2,2,2,2) red
        else picFlatBorder (1,1,1,1) black
    carLabel =
      (withFont carFont $ picText (show i))
      <++> withColor white picMonochrome
  in
   place origin $ drawBorder $ withAlpha 0.65 $
    picFlatBorder (2,10,2,10) carColor carLabel
```

Note that `Picture`, above, denotes a static (non-animated) image, and hence the function above is just an ordinary Haskell function, not a signal function. The `<++>` operator is image composition (sometimes called `'over'`), and `$` is low-precedence function application (to reduce the need for parentheses). The body of `carPic` specifies that the car picture consists of the car's `Id` on a white background, enclosed in a rectangular border in the car color (determined by the car's `Id`), enclosed in a smaller rectangular border that is either black or red, depending on whether the car has been flagged as a tailgater. A translation is applied with `place` to position the upper-left hand corner of the picture at the `origin`

The output of `findTailgaters` is a pair of signals, where the first signal is a list of `(Id, Car)` pairs, and the second is an `(Event (Id,Id))` signal indicating a tailgating occurrence. To make the animation task simpler, we will write a stateful signal function that accumulates the tailgating events into a time-varying set of tailgaters, and uses this to produce an enriched time-varying list of cars, in which each car has a flag indicating whether or not it has ever been identified as a tailgater:

```
accumFlagged :: SF ([(Id,Car)],Event [(Id,Id)])
                   [(Id,Car,Bool)]
accumFlagged = proc (ics, tge) -> do
  tgs <- accumHold [] -<
    fmap (\tgPairs->(union (map fst tgPairs)))
         tge
  returnA -< map (\(i,c) -> (i,c,i `elem` tgs))
                 ics
```

The `accumHold` is used to accumulate the tailgating event into a time-varying set of all `Id`s that have been observed as tailgaters. The output signal simply maps over the `(Id,Car)` pairs, checking for membership in in the set of tailgaters (`tgs`).

Finally, we can write a function that, given a list of cars and their tail-gating status, returns a `Picture` of the UAV's perception of the highway:

```
tgPic :: [(Id,Car,Bool)] -> Picture
tgPic fcs =
  let
    drawCar (cid,(pos,_),istg) =
      translate (vector (pos*ppm+200) 100) %$
        carPic cid istg
    carPics = map drawCar fcs
    hwyPic = foldr (<++>) picEmpty carPics
  in hwyPic
```

In the above definition, each car is translated by a displacement vector, which will result in the upper-left-hand corner of the car being positioned at $(200, 100)$, plus a horizontal offset determined by the car's position relative to the UAV. Like `carPic`, this definition just defines how a static picture is computed from a *snapshot* of the list of cars, and hence is just an ordinary Haskell function, with no AFRP code.

With these definitions in place, animating the tailgating detector is essentially just a matter of using serial composition (`>>>`) to compose `findTailgaters`, `accumFlagged` and `arr tgPic`. Using `arr tgPic` lifts `tgPic` from operating on values to operating on signals: given a *time-varying* list of cars, `arr tgPic` returns a *time-varying* `Picture`, that is, an animation.

3.6 Embedding

Because AFRP is used as the foundation of both the animated graphical user interface and the UAV control system, we could execute the above composition directly to produce an animation. However, this results in executing the tailgating detector and the user interface in the same time frame, which has some significant drawbacks:

- The simulation of the tailgating detector can happen no faster (or slower) than real time, since the user interface executes in real time.
- The *fidelity* of the simulation is affected by the performance of the user interface, since complex rendering may require the implementation to either sample less frequently or drop output frames.

The first point indicates that there actually are two logically distinct time frames in the composed system: one is the real-time of the external world, in which the graphical user interface operates; the other is the simulated time frame of the tailgating simulation. They are logically related, but not necessarily through some simple, static time transformation function since the user may want to speed up or slow down the simulation at will. Compare fast-forwarding a video recording over uninteresting commercials, and then freezing at particularly exciting points for detailed scrutiny.

The second point is an operational concern. Ideally, an implementation would automatically arrange so that various subsystems are executed at optimal rates by judiciously balancing efficiency and real-time considerations on the one hand against numerical accuracy on the other, ensuring that the overall system behavior is a sufficiently good approximation of the conceptual continuous semantics. In practice this is very hard to do, and user intervention is probably going to be required for the foreseeable future. For example, while there exist sophisticated numerical simulation algorithms that adjust the size of the time steps automatically, the selection of the right such algorithm for the problem at hand has thus far not been successfully automated. Automatic partitioning into subsystems when there are conflicting requirements on the step sizes would also appear to be very difficult.

To address these issues, AFRP has an experimental primitive called `embedSynch` which creates a separate time frame in which a subsystem can be executed at a user-definable logical sampling rate. The embedded and the embedding time frames are synchronized at a controllable ratio, which allows the embedded time frame to be sped up or slowed down with respect to the embedding frame.

The signature of embedSynch (slightly simplified) is

```
embedSynch ::
  SF a b -> (a, [(DTime, a)]) -> SF Double b
```

The first argument is the signal function that is being embedded. The second is a discretized representation of the input signal in the form of an interleaving of sample values and time steps. This establishes the embeded time frame. The result is a signal function where the input signal dynamically controls the ratio between the embedded and the embedding time frame. Of course embedSynch is rather operational. It is also overly simplistic in some ways; for example, for simulation purposes, one might rather want the ability to choose the employed integration method, and have that method control the step size. But it is at least a start.

We use the embedSynch primitive to embed the UAV simulation and tailgating detector in the user interface. Thus the simulation gets its own time frame, with a simulated input signal and sampling fidelity that are completely independent of the user interface. The rate at which the output signal of the embedded signal function is sampled (relative to the user interface's time frame) can be controlled by the user interface, using the numeric up/down control to the right of the animation in figure 4.

Note that embedding is related to the idea of *time transformation* that has been implemented in some of the original FRP systems. While embedding in some ways is less expressive than the general transforms offered by those systems, it does not suffer from the operational problems associated with general time transformations, and it effectively allows transformation by a dynamic function (i.e., a signal), rather than a static one.

4 Implementation

Elliott [9] described a number of different approaches to functional implementations of Fran, the first language in the FRP family. One approach is based on synchronized stream processors. This approach was pursued further by Hudak and Wan [15, 25], and later also formed the basis for an early implementation of AFRP. Our current implementation uses continuations, inspired by a similar encoding used in the implementation of Fudgets [3]. In this section, we briefly review the synchronous stream-based implementation, present our alternative continuation-based encoding, and describe some simple enhancements to the continuation-based encoding that enable dynamic optimizations.

4.1 Synchronized Stream Processors

In the stream-based representation of signal functions, signals are represented as time-stamped streams, and signal functions are just functions from streams to streams:

```
type Time = Double
type SP a b = Stream a -> Stream b
newtype SF a b = SF (SP (Time,a) b)
```

The Stream type above can be implemented directly as a (lazy) list in Haskell, as described by Hudak [15].

In the above definition, each signal function (SF a b) is implemented as a Haskell function from a time-stamped stream of a values to a stream of b values. Time stamps may be omitted from the output stream because the implementation is *synchronous*: at every time step, a signal function will consume exactly one value from the head of its input stream and produce exactly one value on its output stream.

While a stream-based implementation is adequate for many purposes, it does have some substantial deficiencies:

- The synchronous nature of every primitive signal function is a critical requirement, but is not explicit in the implementation structure. That is, we require that each stream process consume exactly one input sample from its input stream and produce exactly one output sample on its output stream at every time step, but this is not explicit in the above type definition.
- At the implementation level, there is no way to identify signal functions that only react to *changes* in the input signal. As a consequence, sampling must occur at every time step, even though the program will only react to specific input events. Identifying signal functions that only react to changes in input would enable the implementation to make a blocking call to the operating system until an appropriate event occurs, a substantial performance improvement.
- The implementation does not retain enough information to do any runtime optimization of the dataflow graph.
- It does not seem to be possible[4] to implement first-class signal function continuations in a stream-based implementation, since the connection between a signal function and its input stream is hidden in a closure.

Since first-class signal function continuations are central to the way we handle structurally dynamic systems, we consider it essential to use an alternative representation of signal functions, presented below.

4.2 Continuation-Based Implementation

The Fudgets [3] graphical user interface toolkit is based on asynchronous stream processors. In [3], Carlsson and Hallgren present an alternative to the Stream a -> Stream b representation of stream processors based on *continuations*. Inspired by this, we have adopted a continuation based representation for AFRP. However, since we are working in a synchronous setting, there are substantial differences from the Fudgets implementation. A similar representation, called "residual behaviors", was explored as a possible implementation for Fran in [9].

We will start by explaining a simplified version of the signal function representation, shown below. Optimizations will be discussed later.

```
type DTime = Double

data SF a b =
  SF { sfTF :: DTime -> a -> (SF a b, b) }
```

In this implementation, each signal function is encoded as a *transition function*. The transition function takes as arguments the amount of time passed since the previous time step (DTime), and the current instantaneous value of the input signal (a). The time deltas are assumed to be strictly greater than 0. We will return to the question of what the first time delta should be below.

- a *continuation* (of type SF a b), determining how the signal function will behave at the next time step;

[4] At least not without stepping outside Haskell.

- an *output sample* (of type b), determining the output at the current time step.

The top-level function responsible for animating a signal function (called `reactimate`) runs in an infinite loop: It reads an input sample and the time from the external environment (typically via an I/O action), feeds this sample value (and corresponding `DTime`) to the `SF`'s transition function, and writes the output sample to the environment (also typically via an I/O action). The loop then repeats, but uses the *continuation* returned from the transition function on the next iteration.

4.3 Implementing Primitives

Most of the AFRP primitives have clear and simple implementations as continuations. For example:

```
constant :: b -> SF a b
constant b = SF {sfTF = \ _ _-> (constant b,b)}

identity :: SF a a
identity = SF {sfTF = \ _ a -> (identity,a)}
```

Of course these are just special cases of the point-wise lifting operator, `arr`:

```
sfArr :: (a -> b) -> SF a b
sfArr f = SF {sfTF = \ _ a -> (sfArr f,f a)}
```

The above primitives are all *stateless*. This fact is obvious from their definitions: the continuation returned from the transition function is exactly the signal function being defined. As an example of a *stateful* signal function, here is a simple implementation of `integral`:

```
integral :: Fractional a => SF a a
integral = SF {sfTF = sfAux 0}
  where
    sfAux :: Fractional a =>
               a -> DTime -> a -> (SF a a, a)
    sfAux acc dt a = (SF {sfTF = tf}, acc)
      where tf = sfAux (acc + a*realToFrac dt)
```

The auxiliary function `sfAux` uses partial application to capture the internal state of the integral in the accumulator (`acc`) argument of the transition function.

Many of the higher-order primitives (those that accept signal functions as arguments) are also straightforward. For example serial composition:

```
(>>>) :: SF a b -> SF b c -> SF a c
(SF {sfTF = tf1}) >>> (SF {sfTF = tf2}) =
    SF {sfTF = tf}
  where
    tf dt a = (sf1' >>> sf2', c)
      where
        (sf2', c) = tf2 dt b
        (sf1', b) = tf1 dt a
```

This definition follows naturally from the semantic definition of serial composition given in section 2.3. The transition function (`tf`) simply feeds the input sample and `DTime` to the first signal function (`tf1`) to obtain `sf1'` and `b`, feeds the resulting sample and `DTime` to `tf2` to obtain `sf2'` and `c`, and returns a continuation that is the composition of the continuations `sf1'` and `sf2'`, along with the output sample value `c`.

4.4 Encoding Variability

The continuation-based representation of `SF` allows for simple, precise operational definitions for the various combinators. However, this representation, while simple and general, hides some information that is potentially useful for optimization. For example, the concrete representation of `SF` makes no distinction between *stateless* and *stateful* signal functions.

To enable some simple runtime optimizations (described in the next section), we add extra constructors to the concrete representation of `SF` that encode certain predicates about the signal function. We also add a separate type for the initial continuation, since there is no delta time to be fed in at the very first time step:

```
data SF a b = SF {sfTF :: a -> (SF' a b,b)}

data SF' a b
  = SFGen {sfTF' :: DTime -> a -> (SF' a b,b)}
  | SFArr {sfTF' :: DTime -> a -> (SF' a b,b),
           sfAFun :: a -> b}
  | SFConst {sfTF' :: DTime -> a ->(SF' a b,b),
             sfCVal :: b}
```

Each of the constructors still carries a transition function. The interpretation of the constructors is as follows:

SFGen denotes the most general case of a signal function, where there is no particular "extra" information known about the transition function.

SFArr denotes a point-wise or "pure" signal function. At any time t, the output signal at t depends only on the input sample at t (but not on the time since the last sample). Since a point-wise function is "stateless", the continuation is always just the same signal function regardless of the input sample or `DTime`.

SFConst denotes a signal function that has "gone constant". The output value and continuation for a constant signal function do not change from one sample to the next, regardless of input sample or `DTime` value.

Formally, we can specify the properties captured by the constructors `SFArr` and `SFConst` by means of two predicates, *isArr* and *isConst*:

```
nextSF :: SF a b -> DTime -> a -> SF a b
nextSF sf dt a = fst ((sfTF sf) dt a)

sampleSF :: SF a b -> DTime -> a -> b
sampleSF sf dt a = snd ((sfTF sf) dt a)
```

$$
\begin{aligned}
\mathit{isGen}(\mathtt{sf}) &= \mathit{True}\\
\mathit{isArr}(\mathtt{sf}) &= \forall \mathtt{a}.\forall \mathtt{dt}.((\mathtt{nextSF\ sf\ dt\ a}) = \mathtt{sf}) \wedge\\
&\quad \forall \mathtt{a}.\forall \mathtt{dt}_1,\mathtt{dt}_2.(\mathtt{sampleSF\ sf\ dt}_1\ \mathtt{a}) =\\
&\qquad (\mathtt{sampleSF\ sf\ dt}_2\ \mathtt{a})\\
\mathit{isConst}(\mathtt{sf}) &= \mathit{isArr}(sf) \wedge\\
&\quad \forall \mathtt{a}_1,\mathtt{a}_2.\forall \mathtt{dt}_1,\mathtt{dt}_2.(\mathtt{sampleSF\ sf\ dt}_1\ \mathtt{a}_1) =\\
&\qquad (\mathtt{sampleSF\ sf\ dt}_2\ \mathtt{a}_2)
\end{aligned}
$$

The first part of the conjunction for *isArr* asserts that `sf`'s continuation is `sf` itself. The second part asserts that the sample value is the same regardless of the time delta (`dt`) between samples. The predicate *isConst* extends *isArr* with the requirement that the sample value is independent of delta time or input sample.

Since the first part of *isArr* specifies that the same signal function is returned as the continuation for the next time step, it follows trivially by induction that *isArr* and *isConst* will hold for all subsequent samples of a signal function, and that same value will be returned for all subsequent samples of a constant signal function. Also note that the following implications hold:

$$\mathit{isConst}(\texttt{sf}) \Rightarrow \mathit{isArr}(\texttt{sf}) \Rightarrow \mathit{isGen}(\texttt{sf})$$

Making the variability of signal functions explicit in the constructor enables two key optimizations:

- At the level where `reactimate` interacts with the operating system, knowing that a signal function is `SFConst` or `SFArr` makes it possible to avoid redundant polling. For example, a signal function with variability `SFArr` reacts only to changes in its input signal, not the progression of time. This enables `reactimate` to make a blocking call to the operating system while waiting for an input sample, avoiding redundant polling. At present, the utility of this is limited, but the idea could be carried further by refining the constructors.
- The information about signal functions encoded in each of these constructors enable certain dynamic optimizations to the dataflow graph, based on some simple algebraic identities.

4.5 Simple Dynamic Optimizations

As a signal function is animated, every signal function in the dataflow graph returns a new continuation at every time step. Encoding variability information in constructors enables the implementation to simplify the data flow graph if the graph reaches certain states as it is animated. For example, consider the AFRP primitive `once`:

```
once :: SF (Event a) (Event a)
```

This is a stateful filter that will only pass the *first* occurrence of its input signal to its output signal. Although the transition function for `once` has variability `SFGen` at initialization time, after the input signal has had an event occurrence, the continuation returned by `once` will be equivalent to `constant NoEvent`, with variability `SFConst`, and will therefore have no subsequent occurrences.

Such information can be used to optimize the dataflow graph dynamically by exploiting simple algebraic identities. For example, our implementation of serial composition exploits the following identities:

```
sf         >>> constant c = constant c
constant c >>> arr f      = constant (f c)
arr f      >>> arr g      = arr (g . f)
```

Our optimized implementation of serial composition uses pattern matching to identify the above cases; the implementation follows directly from the above identities, with a default case to be applied when none of the above optimizations are applicable.

We also provide optimized versions of some of the other wiring combinators, such as `first`, using the identities:

```
first (constant b) = arr (\(_, c) -> (b, c))
(first (arr f))    = arr (\(a, c) -> (f a, c))
```

One special case that we would have liked to encode in our constructors was `SFId`, to indicate the special case of the lifted identity function, `arr id`. For example, consider

```
initially :: a -> SF a a
```

This function behaves as the identity function, except at the instant $t = 0$, where its first argument (the initial value) is used as the output sample. If we could capture the fact that a signal function has become `(arr id)` in the representation of `SF'`, then we could exploit identities such as:

```
first (arr id) = arr id
```

Unfortunately, it appears that we would need dependent types to exploit such a constructor, since the type of the transition function (`sfTF'`) is too general. We could potentially keep around an extra function as a "proof" that we can perform the required coercions, but then the resulting code is no more efficient than using `SFArr` in the first place.

One last point to is that we must be careful to when propagating variability information. For example, even if both arguments to a switch are `SFArr`, the resulting signal function is still `SFGen`. This is because `switch` in itself is a stateful operation. We have explored adding another constructor to `SF'` that basically would capture the case that something could be `SFArr` for a while. This would yield more opportunities for blocking I/O. However, it is not part of the current implementation.

5 Related Work

Functional Reactive Programming grew out of Conal Elliot and Paul Hudak's work on Functional Reactive Animation [10]. Since then, the basic FRP framework has been implemented in a number of different ways, many of which have not been well described or published. Synchrony and continuous time have always been central aspects of FRP. This directly relates it to synchronous (dataflow) languages like Esterel [1], Lustre [4, 12], and Lucid Synchrone [6, 22] on the one hand, and to hybrid automata [14] and languages for hybrid modeling and simulation, like Simulink [24], on the other.

AFRP is intended to be a robust and expressive implementation of FRP, capable of describing reactive systems with a highly dynamic structure, such as graphical user interfaces or vision-based robot control systems [19], while retaining the fundamental advantages of the synchronous programming paradigm. Performance *guarantees* for space and time have so far been a secondary concern, although we have gone to great lengths to ensure that the system runs as smoothly as possible in practice. This puts AFRP in marked contrast to synchronous languages, as well as the RT-FRP line of work [26], where central aspects are guaranteed reactivity, execution in bounded space and time, and efficient implementation (including compilation to digital circuits [2]), but at the expense of requiring a fairly rigid system structure. For example, the closest thing there is to a `switch`-like construct in Lucid Synchrone is a reset operator [13], which causes a stream computation to start over.

AFRP shares with hybrid systems languages an inherent notion of continuous time and event-like abstractions for capturing discrete aspects of a system. However, the underlying numeric machinery of AFRP is currently much more simplistic than what is typical for hybrid modeling and simulation languages. For example, accurate location of the time of an event occurrence is often considered critical and requires complex algorithms in combination with language restrictions to be computationally tractable. Similarly, these languages often use sophisticated algorithms for integration with variable step size to ensure rapid computation as well as accurate results. The AFRP implementation does not currently do any of this. However, the ability of AFRP to express structurally dynamic systems, which typical hybrid modeling languages cannot deal with cleanly, makes it an interesting topic of future research to attempt to address such numerical concerns within AFRP.

Fudgets [3] has been a source of inspiration for the AFRP implementation. However, the asynchronous nature of Fudgets make it fundamentally different from AFRP. There is certainly an overlap of possible application domains (such as graphical user interfaces), but for areas where time and synchrony is inherent (animation, hybrid systems), we believe that a synchronous language is the obvious choice.

The continuation-based implementation of AFRP also bears a clear resemblance to other work in the area, such as the co-iterative characterization of Lucid Synchrone [5], the operational semantics of RT-FRP [26], and the "residual behaviors" implementation of Fran [9].

FranTk [23] provided support for dynamic collections and included a highly optimized implementation of the Fran core. However, FranTk's underlying Fran implementation was fundamentally imperative (depending on `unsafePerformIO` behind the scenes), and FranTk presented an imperative interface to the programmer (via the GUI monad) for creation or re-shaping of the dataflow graph. The imperative implementation resulted in some substantial semantical issues (including a basic flaw in referential transparency) that were never clearly resolved. In contrast, AFRP does not resort to `unsafePerformIO` or imperative programming in either the interface or the implementation. One drawback to our approach is that there are a number of undocumented aggressive optimizations present in FranTk's implementation that we, thus far, have been unable to include in our implementation.

6 Future Work

Our long-term goal is to produce an implementation of AFRP that combines the expressive power of synchronous dataflow languages, hybrid modeling languages, and Haskell, while executing programs in a reasonable amount of time and space. Since these goals, expressiveness and firm performance guarantees, are often at odds with each other, one could imagine a setting where performance aspects, along the lines of Lucid Synchrone's clock calculus or RT-FRP, are checked through *soft* type systems. Thus the user would have the power to make the best trade-off between expressiveness and performance for the problem at hand while remaining within a uniform framework. For example, in complex control systems, typically only some parts of the system have true real-time requirements. Thus, for some parts of the system, the full expressive power of AFRP can be used, while other parts where real-time performance are crucial would have to use a restricted version of AFRP that can provide the necessary performance guarantees.

Another line of interesting research is to explore the relationship between AFRP and hybrid modeling and simulation. A version of AFRP with the numerical sophistication required for reliable and efficient simulation would be a significant contribution to this community. A fairly recent development in the hybrid modeling area is *non-causal modeling* (or "object-oriented" modeling[5]), where models are expressed in terms of *non-directed* Hybrid Differential Algebraic Equations (DAEs), making then more reusable and declarative [7]. A successful example of such a non-causal modeling language is Modelica [17]. It would be interesting to see if such capabilities could be integrated smoothly into an AFRP-style system. We believe that by adding *signal relations* as first class entities in addition to signal functions we could integrate non-causal models into our system, but the technical challenges for efficient and sound numerical simulation would be substantial.

We are currently employing Ross Paterson's syntactic sugar for arrows [18] to facilitate writing AFRP programs. Compared with previous versions of FRP, or languages like Lustre or Lucid Synchrone, where function application syntax can be used even when the applied function is stateful, the AFRP syntax sometimes seems a bit clumsy. This is true in particular when simple mathematical expressions such as integrals have to be broken up since integration is a stateful operation. It would be interesting to see if there are other syntactical possibilities, possibly exploiting properties the AFRP arrows enjoy beyond the standard arrow properties. However, being explicit about stateful vs. stateless function application has its advantages, so it is not clear what the best trade-off is.

7 Conclusions

FRP is a novel way of programming interactive systems without resorting to brute-force imperative approaches such as the IO monad to express interaction. By extending traditional functional programming with abstractions that express time flow, we can retain the elegance and modularity of traditional functional programming in a domain where less expressive languages have predominated.

AFRP represents a new approach to the design of reactive languages. The use of arrows as the organizing abstraction for this system lends both syntactic and semantic clarity to our language. We have expanded the functionality of previous FRP implementations with flexible constructs for managing dynamic collections, and first-class continuations that can capture signal functions that are "in progress". This approach is particularly well suited for systems that contain changing groups of elements that interact with each other. Collection-based switching subsumes previous switching constructs and gives AFRP a significant advantage over other synchronous reactive languages.

We have also explored the use of embedding, allowing the user to have direct control over time flow within a component. While experimental and in some ways rather simplistic, this feature does offer the user a way of controlling the fidelity of subsystems, which is important for accurate simulation, as well as the ability to construct multi-rate systems, which addresses an important operational concern. Furthermore, it offers a restricted but robust form of time transformation, without the performance problems encountered in previous FRP implementations. This allows applications such as simulators to speed up or slow down time flow as needed, for example for real-time animation without affecting simulation fidelity.

These new AFRP constructs are demonstrated in a non-trivial application domain that showcases the ability of our system to capture complex communication patterns among a dynamic collection of objects. This example also shows how functional programming can be used to capture complex system structures in a succinct and reusable manner. Overall, we think AFRP has an expressive power and semantic simplicity that makes AFRP programs easy to understand even when describing structurally dynamic systems.

[5]Because the modeling focuses on physical objects; not to be confused with object-oriented programming.

Finally, we have described a new implementation of AFRP. This implementation addresses the new features as well as optimizations made possible by the continuation-based implementation style. The implementation appears to have more predictable performance characteristics than previous FRP implementations.

8 References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):217–248, 1992.

[2] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing Series. MIT Press, 2000.

[3] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, March 1998.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[5] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.

[6] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Submitted for publication, 2000.

[7] Franois E. Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands*, pages 53–64, 1996.

[8] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

[9] Conal Elliott. Functional implementations of continuous modelled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[11] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[13] Grégoire Hamon and Marc Pouzet. Modular resetting of synchronous data-flow programs. In *Principles and Practice of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[14] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logics in Computer Science (LICS 1996)*, pages 278–292, 1996.

[15] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, Cambridge, UK, 2000.

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

[17] Modelica – a unified object-oriented language for physical systems modeling: Language specification version 1.4. The Modelica Association, `http://www.modelica.org`, December 2000.

[18] Ross Paterson. A new notation for arrows. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, September 2001.

[19] Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, October 2002.

[20] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Fist International Workshop on Practical Aspects of' Declarative Languages (PADL)*, January 1999.

[21] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. FVision: A declarative language for visual tracking. In *Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages*, pages 304–321, January 2001. `http://haskell.org/frp/publication.html#fvision`

[22] Marc Pouzet, Paul Caspi, Pascal Couq, and Grégoire Hamon. Lucid Synchrone v2.0 – tutorial and reference manual. `http://www-spi.lip6.fr/lucid-synchrone/lucid_synchrone_2.0_manual.ps`, April 2001.

[23] Meurig Sage. Frantk: A declarative gui system for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.

[24] Using Simulink version 4. The MathWorks, Inc., `http://www.mathworks.com`, June 2001.

[25] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[26] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP'01)*, 2001.

[27] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*, January 2002.