Arrows, Robots, and Functional Reactive Programming

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson

Yale University*
Department of Computer Science
paul.hudak@yale.edu, antony.courtney@yale.edu,
henrik.nilsson@yale.edu, john.c.peterson@yale.edu

Abstract. *Functional reactive programming*, or FRP, is a paradigm for programming *hybrid systems* – i.e., systems containing a combination of both continuous and discrete components – in a high-level, declarative way. The key ideas in FRP are its notions of continuous, time-varying values, and time-ordered sequences of discrete events.
Yampa is an instantiation of FRP as a domain-specific language embedded in Haskell. This paper describes Yampa in detail, and shows how it can be used to program a particular kind of hybrid system: a *mobile robot*. Because performance is critical in robotic programming, Yampa uses *arrows* (a generalization of monads) to create a disciplined style of programming with time-varying values that helps ensure that common kinds of time- and space-leaks do not occur.
No previous experience with robots is expected of the reader, although a basic understanding of physics and calculus is assumed. No knowledge of arrows is required either, although we assume a good working knowledge of Haskell.

This paper is dedicated in memory of
Edsger W. Dijkstra
for his influential insight that mathematical logic is and must be the basis for sensible computer program construction.

1 Introduction

Can functional languages be used in the real world, and in particular for real-time systems? More specifically, can the expressiveness of functional languages be used advantageously in such systems, and can performance issues be overcome at least for the most common applications?

For the past several years we have been trying to answer these questions in the affirmative. We have developed a general paradigm called *functional reactive*

* This research was supported in part by grants from the National Science Foundation (CCR9900957 and CCR-9706747), the Defense Advanced Research Projects Agency (F33615-99-C-3013 and DABT63-00-1-0002), and the National Aeronautics and Space Administration (NCC 2-1229). The second author was also supported by an NSF Graduate Research Fellowship.

programming that is well suited to programming *hybrid systems*, i.e. systems with both continuous and discrete components. An excellent example of a hybrid system is a *mobile robot*. From a *physical* perspective, mobile robots have continuous components such as voltage-controlled motors, batteries, and range finders, as well as discrete components such as microprocessors, bumper switches, and digital communication. More importantly, from a *logical* perspective, mobile robots have continuous notions such as wheel speed, orientation, and distance from a wall, as well as discrete notions such as running into another object, receiving a message, or achieving a goal.

Functional reactive programming was first manifested in *Fran*, a domain specific language (DSL) for graphics and animation developed by Conal Elliott at Microsoft Research [5, 4]. *FRP* [13, 8, 16] is a DSL developed at Yale that is the "essence" of Fran in that it exposes the key concepts without bias toward application specifics. *FAL* [6], *Frob* [11, 12], *Fvision* [14], and *Fruit* [2] are four other DSLs that we have developed, each embracing the paradigm in ways suited to a particular application domain. In addition, we have pushed FRP toward real-time embedded systems through several variants including *Real-Time FRP* and *Event-Driven FRP* [18, 17, 15].

The core ideas of functional reactive programming have evolved (often in subtle ways) through these many language designs, culminating in what we now call *Yampa*, which is the main topic of this paper.[1] Yampa is a DSL embedded in Haskell and is a refinement of FRP. Its most distinguishing feature is that the core FRP concepts are represented using *arrows* [7], a generalization of monads. The programming discipline induced by arrows prevents certain kinds of time- and space-leaks that are common in generic FRP programs, thus making Yampa more suitable for systems having real-time constraints.

Yampa has been used to program real industrial-strength mobile robots [10, 8][2], building on earlier experience with FRP and Frob [11, 12]. In this paper, however, we will use a *robot simulator*. In this way, the reader will be able to run all of our programs, as well as new ones that she might write, without having to buy a $10,000 robot. All of the code in this paper, and the simulator itself, are available via the Yampa home page at `www.haskell.org/yampa`.

The simulated robot, which we refer to as a *simbot*, is a *differential drive* robot, meaning that it has two wheels, much like a cart, each driven by an independent motor. The relative velocity of these two wheels thus governs the turning rate of the simbot; if the velocities are identical, the simbot will go straight. The physical simulation of the simbot includes translational inertia, but (for simplicity) not rotational inertia.

The motors are what makes the simbot go; but it also has several kinds of sensors. First, it has a bumper switch to detect when the simbot gets "stuck."

[1] Yampa is a river in Colorado whose long placid sections are occasionally interrupted by turbulent rapids, and is thus a good metaphor for the continuous and discrete components of hybrid systems. But if you prefer acronyms, Yampa was started at YAle, ended in Arrows, and had Much Programming in between.

[2] In these two earlier papers we referred to Yampa as *AFRP*.

That is, if the simbot runs into something, it will just stop and signal the program. Second, it has a range finder that can determine the nearest object in any given direction. In our examples we will assume that the simbot has independent range finders that only look forward, backward, left, and right, and thus we will only query the range finder at these four angles. Third, the simbot has what we call an "animate object tracker" that gives the location of all other simbots, as well as possibly some free-moving balls, that are within a certain distance from the simbot. You can think of this tracker as modelling either a visual subsystem that can see these objects, or a communication subsystem through which the simbots and balls share each others' coordinates. Each simbot also has a unique ID and a few other capabilities that we will introduce as we need them.

2 Yampa Basics

The most important concept underlying functional reactive programming is that of a *signal*: a continous, time-varying value. One can think of a signal as having polymorphic type:

```
Signal a = Time -> a
```

That is, a value of type `Signal a` is a function mapping suitable values of time (`Double` is used in our implementation) to a value of type `a`. Conceptually, then, a signal s's value at some time t is just $s(t)$.

For example, the velocity of a differential drive robot is a pair of numbers representing the speeds of the left and right wheels. If the speeds are in turn represented as type `Speed`, then the robot's velocity can be represented as type `Signal (Speed,Speed)`. A program controlling the robot must therefore provide such a value as output.

Being able to define and manipulate continuous values in a programming language provides great expressive power. For example, the equations governing the motion of a differential drive robot [3] are:

$$x(t) = \tfrac{1}{2} \int_0^t (v_r(t) + v_l(t)) \ \cos(\theta(t)) \ dt$$
$$y(t) = \tfrac{1}{2} \int_0^t (v_r(t) + v_l(t)) \ \sin(\theta(t)) \ dt$$
$$\theta(t) = \tfrac{1}{l} \int_0^t (v_r(t) - v_l(t)) \ dt$$

where $x(t)$, $y(t)$, and $\theta(t)$ are the robot's x and y coordinates and orientation, respectively; $v_r(t)$ and $v_l(t)$ are the right and left wheel speeds, respectively; and l is the distance between the two wheels. In FRP these equations can be written as:

```
x     = (1/2) * integral ((vr + vl) * cos theta)
y     = (1/2) * integral ((vr + vl) * sin theta)
theta = (1/l) * integral (vr - vl)
```

All of the values in this FRP program are implicitly time-varying, and thus the explicit time t is not present.[3] Nevertheless, the direct correspondence between the physical equations (i.e. the specification) and the FRP code (i.e. the implementation) is very strong.

2.1 Arrowized FRP

Although quite general, the concept of a signal can lead to programs that have conspicuous time- and space-leaks,[4] for reasons that are beyond the scope of this paper. Earlier versions of Fran, FAL, and FRP used various methods to make this performance problem less of an issue, but ultimately they all either suffered from the problem in one way or another, or introduced other problems as a result of fixing it.

In Yampa, the problem is solved in a more radical way: signals are simply not allowed as first-class values! Instead, the programmer has access only to *signal transformers*, or what we prefer to call *signal functions*. A signal function is just a function that maps signals to signals:

```
SF a b = Signal a -> Signal b
```

However, the actual representation of the type SF in Yampa is hidden (i.e. `SF` is abstract), so one cannot directly build signal functions or apply them to signals. Instead of allowing the user to define *arbitrary* signal functions from scratch (which makes it all too easy to introduce time- and space-leaks), we provide a set of primitive signal functions and a set of special composition operators (or "combinators") with which more complex signal functions may be defined. Together, these primitive values and combinators provide a *disciplined* way to define signal functions that, fortuitously, avoids time- and space-leaks. We achieve this by structuring Yampa based on *arrows*, a generalization of monads proposed in [7]. Specifically, the type `SF` is made an instance of the `Arrow` class.

So broadly speaking, a Yampa program expresses the composition of a possibly large number of signal functions into a composite signal function that is then "run" at the top level by a suitable interpreter. A good analogy for this idea is a state or IO monad, where the state is hidden, and a program consists of a linear sequencing of actions that are eventually run by an interpreter or the operating system. But in fact arrows are more general than monads, and in particular the composition of signal functions does not have to be completely linear, as will be illustrated shortly. Indeed, because signal functions are abstract, we should

[3] This implies that the sine, cosine, and arithmetic operators are over-loaded to handle signals properly.

[4] A time-leak in a real-time system occurs whenever a time-dependent computation falls behind the current time because its value or effect is not needed yet, but then requires "catching up" at a later point in time. This catching up process can take an arbitrarily long time, and may or may not consume space as well. It can destroy any hope for real-time behavior if not managed properly.

be concerned that we have a sufficient set of combinators to compose our signal functions without loss of expressive power.

We will motivate the set of combinators used to compose signal functions by using an analogy to so-called "point-free" functional programming (an example of which is the Bird/Meertens formalism [1]). For the simplest possible example, suppose that `f1 :: A -> B` and `f2 :: B -> C`. Then instead of writing:

```
g :: A -> C
g x = f2 (f1 x)
```

we can write in a point-free style using the familiar function composition operator:

```
g = f2 . f1
```

This code is "point-free" in that the values (points) passed to and returned from a function are never directly manipulated.

To do this at the level of signal functions, all we need is a primitive operator to "lift" ordinary functions to the level of signal functions:

```
arr :: (a -> b) -> SF a b
```

and a primitive combinator to compose signal functions:

```
(>>>) :: SF a b -> SF b c -> SF a c
```

We can then write:

```
g' :: SF A C
g' = arr g
   = arr f1 >>> arr f2
```

Note that `(>>>)` actually represents reverse function composition, and thus its arguments are reversed in comparison to `(.)`.

Unfortunately, most programs are not simply linear compositions of functions, and it is often the case that more than one input and/or output is needed. For example, suppose that `f1 :: A -> B, f2 :: A -> C` and we wish to define the following in point-free style:

```
h :: A -> (B,C)
h x = (f1 x, f2 x)
```

Perhaps the simplest way is to define a combinator:

```
(&) :: (a->b) -> (a->c) -> a -> (b,c)
(f1 & f2) x = (f1 x, f2 x)
```

which allows us to define `h` simply as:

```
h = f1 & f2
```

In Yampa there is a combinator `(&&&) :: SF a b -> SF a c -> SF a (b,c)` that is analogous to `&`, thus allowing us to write:

```
h' :: SF A (B,C)
h' = arr h
   = arr f1 &&& arr f2
```

As another example, suppose that `f1 :: A -> B` and `f2 :: C -> D`. One can easily write a point-free version of:

```
i :: (A,C) -> (B,D)
i (x,y) = (f1 x, f2 y)
```

by using `(&)` defined above and Haskell's standard `fst` and `snd` operators:

```
i = (f1 . fst) & (f2 . snd)
```

For signal functions, all we need are analogous versions of `fst` and `snd`, which we can achieve via lifting:

```
i' :: SF (A,C) (B,D)
i' = arr i
   = arr (f1 . fst) &&& arr (f2 . snd)
   = (arr fst >>> arr f1) &&& (arr snd >>> arr f2)
```

The "argument wiring" pattern captured by `i'` is in fact a common one, and thus Yampa provides the following combinator:

```
(***) :: SF b c -> SF b' c' -> SF (b,b') (c,c')
f *** g = (arr fst >>> f) &&& (arr snd >>> g)
```

so that `i'` can be written simply as:

```
i' = arr f1 *** arr f2
```

`g'`, `h'`, and `i'` were derived from `g`, `h`, and `i`, respectively, by appealing to one's intuition about functions and their composition. In the next section we will formalize this using type classes.

2.2 The Arrow Class

One could go on and on in this manner, adding combinators as they are needed to solve particular "argument wiring" problems, but it behooves us at some point to ask if there is a minimal universal set of combinators that is sufficient to express all possible wirings. Note that so far we have introduced three combinators – `arr`, `(>>>)`, and `(&&&)` – without definitions, and a fourth – `(***)` – was defined in terms of these three. Indeed these three combinators constitute a minimal universal set.

However, this is not the only minimal set. In fact, in defining the original `Arrow` class, Hughes instead chose the set `arr`, `(>>>)`, and `first`:

```
class Arrow a where
  arr   :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

where `first` is analogous to the following function defined at the ordinary function level:

```
firstfun f = \(x,y) -> (f x, y)
```

In Yampa, the type `SF` is an instance of class `Arrow`, and thus these types are consistent with what we presented earlier. To help see how this set is miminal, here are definitions of `second` and `(&&&)` in terms of the `Arrow` class methods:

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
           where swap pr = (snd pr, fst pr)

(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\b -> (b,b)) >>> (first f >>> second g)
```

In addition, here is an instance declaration that shows how Haskell's normal function type can be treated as an arrow:

```
instance Arrow (->) where
  arr f = f
  f >>> g = g . f
  first f = \(b,d) -> (f b, d)
```

With this instance declaration, the derivations of `g'`, `h'`, and `i'` in the previous section can be formally justified.

Exercise 1. Define (a) `first` in terms of just `arr`, `(>>>)`, and `(&&&)`, (b) `(***)` in terms of just `first`, `second`, and `(>>>)`, and (c) `(&&&)` in terms of just `arr`, `(>>>)`, and `(***)`.

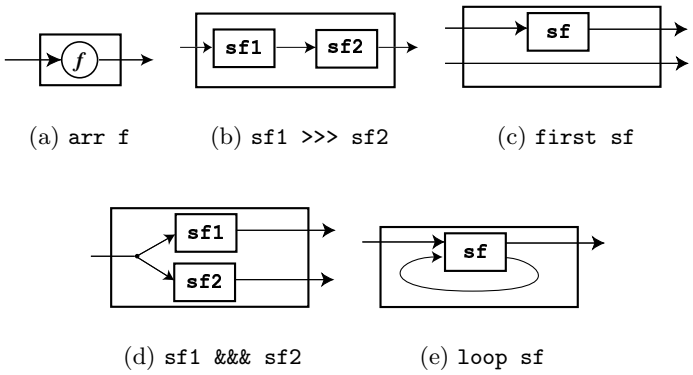
2.3 Commonly Used Combinators

In practice, it is better to think in terms of a commonly-used set of combinators rather than a minimal set. Figure 1 shows a set of eight combinators that we use often in Yampa programming, along with the graphical "wiring of arguments" that five of them imply.

Yampa also provides many convenient library functions that facilitate programming in the arrow framework. Here are some that we will use later in this paper:

```
identity :: SF a a
constant :: b -> SF a b
time     :: SF a Time
```

```
arr    :: Arrow a => (b -> c) -> a b c
(>>>)  :: Arrow a => a b c -> a c d -> a b d
(<<<)  :: Arrow a => a c d -> a b c -> a b d
first  :: Arrow a => a b c -> a (b,d) (c,d)
second :: Arrow a => a b c -> a (d,b) (d,c)
(***)  :: Arrow a => a b c -> a b' c' -> a (b,b') (c,c')
(&&&)  :: Arrow a => a b c -> a b c' -> a b (c,c')
loop   :: Arrow a => a (b,d) (c,d) -> a b c
```



(a) `arr f` (b) `sf1 >>> sf2` (c) `first sf`

(d) `sf1 &&& sf2` (e) `loop sf`

Fig. 1. Commonly Used Arrow Combinators

The `identity` signal function is analogous to the identity function in Haskell, and in fact is equivalent to `arr id`. The `constant` function is useful for generating constant signal functions, is analogous to Haskell's `const` function, and in fact is equivalent to `arr . const`. Finally, `time` is a signal function that yields the current time, and is equivalent to `constant 1.0 >>> integral`, where `integral` is a pre-defined Yampa signal function with type:[5]

```
integral :: SF Double Double
```

Yampa also defines a `derivative` signal function.

It is important to note that some signal functions are *stateful*, in that they accumulate information over time. `integral` is a perfect example of such a function: by definition, it sums instantaneuous values of a signal over time. Stateful signal functions cannot be defined using `arr`, which only lifts pure functions to the level of arrows. Stateful functions must either be pre-defined or be defined in terms of other stateful signal functions.

Stated another way, stateful signal functions such as integration and differentiation depend intimately on the underlying time-varying semantics, and so do not have analogous unlifted forms. Indeed, it is so easy to lift unary functions to the level of signal functions that there is generally no need to provide special signal function versions of them. For example, instead of defining a special `sinSF`, `cosSF`, etc., we can just use `arr sin`, `arr cos`, etc. Furthermore, with the binary lifting operator:

```
arr2 :: (a->b->c) -> SF (a,b) c
arr2 = arr . uncurry
```

we can also lift binary operators. For example, `arr2 (+)` has type `Num a => SF (a,a) a`.

2.4 A Simple Example

To see all of this in action, consider the FRP code presented earlier for the coordinates and orientation of the mobile robot. We will rewrite the code for the x-coordinate in Yampa (leaving the y-coordinate and orientation as an exercise).

Suppose there are signal functions `vrSF, vlSF :: SF SimbotInput Speed` and `thetaSF :: SF SimbotInput Angle`. The type `SimbotInput` represents the input state of the simbot, which we will have much more to say about later. With these signal functions in hand, the previous FRP code for `x`:

```
x = (1/2) * integral ((vr + vl) * cos theta)
```

can be rewritten in Yampa as:

[5] This function is actually overloaded for any vector space, but that does not concern us here, and thus we have specialized it to `Double`.

```
xSF :: SF SimbotInput Distance
xSF = let v = (vrSF &&& vlSF) >>> arr2 (+)
          t = thetaSF >>> arr cos
      in (v &&& t) >>> arr2 (*) >>> integral >>> arr (/2)
```

Exercise 2. Define signal functions `ySF` and `thetaSF` in Yampa that correspond to the definitions of `y` and `theta`, respectively, in FRP.

2.5 Arrow Syntax

Although we have achieved the goal of preventing direct access to signals, one might argue that we have lost the clarity of the original FRP code: the code for `xSF` is certainly more difficult to understand than that for `x`. Most of the complexity is due to the need to wire signal functions together using the various pairing/unpairing combinators such as `(&&&)` and `(***)`. Precisely to address this problem, Paterson [9] has suggested the use of special syntax to make arrow programming more readable, and has written a preprocessor that converts the syntactic sugar into conventional Haskell code. Using this special *arrow syntax*, the above Yampa code for `xSF` can be rewritten as:

```
xSF' :: SF SimbotInput Distance
xSF' = proc inp -> do
  vr    <- vrSF     -< inp
  vl    <- vlSF     -< inp
  theta <- thetaSF  -< inp
  i     <- integral -< (vr+vl) * cos theta
  returnA -< (i/2)
```

Although not quite as readable as the original FRP definition of `x`, this code is far better than the unsugared version. There are several things to note about the structure of this code:

1. The syntax `proc` *pat* `->` `...` is analogous to a Haskell lambda expression of the form `\` *pat* `->` `...` , except that it defines a signal function rather than a normal Haskell function.
2. In the syntax *pat* `<-` *SFexpr* `-<` *expr*, the expression *SFexpr* must be a signal function, say of type `SF T1 T2`, in which case *expr* must have type `T1` and *pat* must have type `T2`. This is analogous to *pat* `=` $expr_1$ $expr_2$ in a Haskell `let` or `where` clause, in which case if $expr_1$ has type `T1 -> T2`, then $expr_2$ must have type `T1` and *pat* must have type `T2`.
3. The overall syntax:

```
proc pat -> do
   pat1 <- SFexpr1 -< expr1
   pat2 <- SFexpr2 -< expr2
   ...
   returnA -< expr
```

defines a signal function. If *pat* has type `T1` and *expr* has type `T2`, then the type of the signal function is `SF T1 T2`. In addition, any variable bound by one of the patterns pat_i can only be used in the expression *expr* or in an expression $expr_j$ where $j > i$. In particular, it cannot be used in any of the signal function expressions $SFexpr_i$.

It is important to note that the arrow syntax allows one to get a handle on a signal's *values* (or *samples*), but *not* on the signals themselves. In other words, first recalling that a signal function `SF a b` can be thought of as a type `Signal a -> Signal b`, which in turn can be thought of as type `(Time -> a) -> (Time -> b)`, the syntax allows getting a handle on values of type `a` and `b`, but not on values of type `Time -> a` or `Time -> b`.

Figure 2(a) is a *signal flow diagram* that precisely represents the wiring implied by the sugared definition of `xSF'`. (It also reflects well the data dependencies in the original FRP program for `x`.) Figure 2(b) shows the same diagram, except that it has been overlaid with the combinator applications implied by the unsugared definition of `xSF` (for clarity, the lifting via `arr` of the primitive functions – i.e. those drawn with circles – is omitted). These diagrams demonstrate nicely the relationship between the sugared and unsugared forms of Yampa programs.

Exercise 3. Rewrite the definitions of `ySF` and `thetaSF` from the previous exercise using the arrow syntax. Also draw their signal flow diagrams.

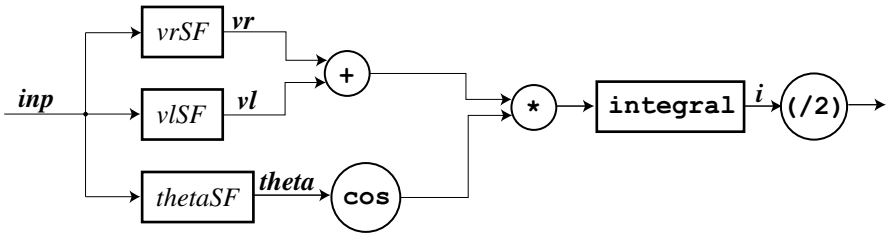
2.6 Discrete Events and Switching

Most programming languages have some kind of conditional choice capability, and Yampa is no exception. Indeed, given signal functions `flag :: SF a Bool` and `sfx, sfy :: SF a b`, then the signal function:

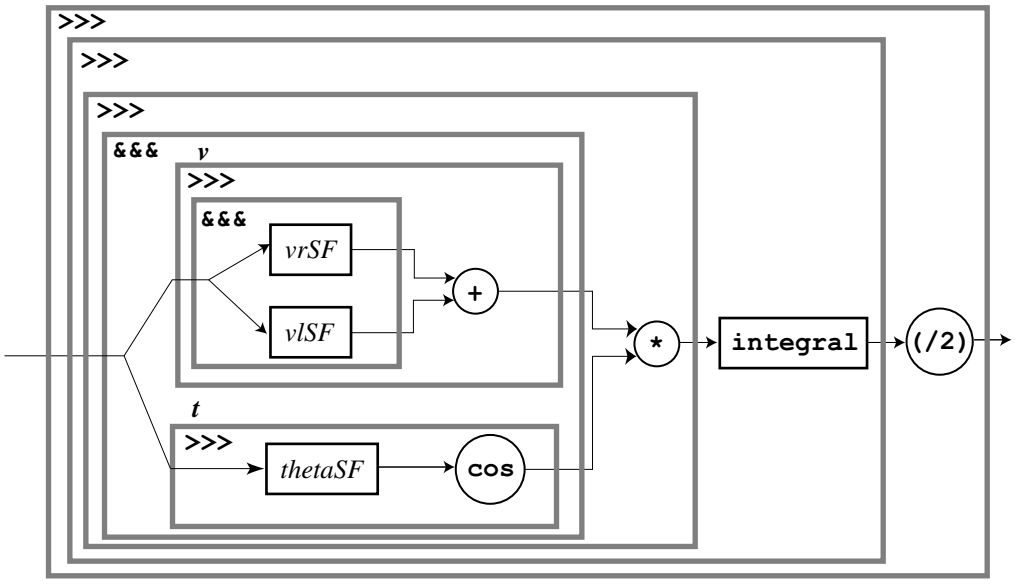
```
sf :: SF a b
sf = proc i -> do
  x <- sfx  -< i
  y <- sfy  -< i
  b <- flag -< i
  returnA -< if b then x else y
```

behaves like `sfx` whenever `flag` yields a true value, and like `sfy` whenever it yields false.

However, this is not completely satisfactory, because there are many situations where one would prefer that a signal function *switch into*, or literally become, some other signal function, rather than continually alternate between two signal functions based on the value of a boolean. Indeed, there is often a succession of new signal functions to switch into as a succession of particular events occurs, much like state changes in a finite state automaton. Furthermore, we would like for these newly invoked signal functions to start afresh from time zero, rather than being signal functions that have been "running" since the



(a) Sugared



(b) Unsugared

Fig. 2. Signal Flow Diagrams for `xSF`

program began. This relates precisely to the issue of "statefulness" that was previously discussed.

This advanced functionality is achieved in Yampa using *events* and *switching combinators*.

In previous versions of FRP, including Fran, Frob, and FAL, a significant distinction was made between continuous values and discrete events. In Yampa this distinction is not as great. Events in Yampa are just abstract values that are isomorphic to Haskell's `Maybe` data type. A signal of type `Signal (Event b)` is called an *event stream*, and is a signal that, at any point in time, yields either nothing or an event carrying a value of type `b`. A signal function of type `SF a (Event b)` generates an event stream, and is called an *event source*.

Note: Although event streams and continuous values are both represented as signals in Yampa, there are important semantic differences between them. For example, improper use of events may lead to programs that are not convergent, or that allow the underlying sampling rate to "show through" in the program's behavior. Semantically speaking, event streams in Yampa should not be "infinitely dense" in time; practically speaking, their frequency should not exceed the internal sampling rate unless buffering is provided.[6]

As an example of a well-defined event source, the signal function:

```
rsStuck :: SF SimbotInput (Event ())
```

generates an event stream whose events correspond to the moments when the robot gets "stuck:" that is, an event is generated every time the robot's motion is blocked by an obstacle that it has run into.

What makes event streams special is that there is a special set of functions that use event streams to achieve various kinds of switching. The simplest switching combinator is called `switch`, whose type is given by:

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

The expression `(sf1 &&& es) ‘switch‘ \e -> sf2` behaves as `sf1` until the first event in the event stream `es` occurs, at which point the event's value is bound to `e` and the behavior switches over to `sf2`.

For example, in order to prevent damage to a robot wheel's motor, we may wish to set its speed to zero when the robot gets stuck:

```
xspd :: Speed -> SF SimbotInput Speed
xspd v = (constant v &&& rsStuck) ‘switch‘ \() -> constant 0
```

It should be clear that stateful Yampa programs can be constructed using switching combinators.

Exercise 4. Rather than set the wheel speed to zero when the robot gets stuck, negate it instead. Then define `xspd` recursively so that the velocity gets negated *every time* the robot gets stuck.

[6] Certain input events such as key presses are in fact properly bufferred in our implementation such that none will be lost.

Switching semantics. There are several kinds of switching combinators in Yampa, four of which we will use in this paper. These four switchers arise out of two choices in the semantics:

1. Whether or not the switch happens exactly at the time of the event, or infinitesimally just after. In the latter case, a "`d`" (for "delayed") is prefixed to the name `switch`.
2. Whether or not the switch happens just for the first event in an event stream, or for every event. In the latter case, an "`r`" (for "recurring") is prefixed to the name `switch`.

This leads to the four switchers, whose names and types are:

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

An example of the use of `switch` was given above. Delayed switching is useful for certain kinds of recursive signal functions. In Sec. 2.7 we will see an example of the use of `drSwitch`.

As mentioned earlier, an important property of switching is that time begins afresh within each signal function being switched into. For example, consider the expression:

```
let sinSF = time >>> arr sin
in (sinSF &&& rsStuck) `switch` const sinSF
```

`sinSF` to the left of the switch generates a sinusoidal signal. If the first event generated by `rsStuck` happens at time t, then the `sinSF` on the right will begin at time 0, regardless of what the time t is; i.e. the sinusoidal signal will start over at the time of the event.

Useful event functions. `Event` is an instance of class `Functor`, and thus `fmap` can be used to change the value carried by an event. For example, we can increment the value of an event `e :: Event Double` by writing `fmap (+1) e`. Sometimes we don't care about the old value of an event when creating a new one, so Yampa also provides:

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

It is often desirable to *merge events*; for example, to form the disjunction of two logical events. The only problem is deciding what to do with simultaneous events. The most general form of merge:

```
mergeBy :: (a -> a -> a) -> Event a -> Event a -> Event a
```

allows the user to decide how to handle simultaneous events by providing a function to combine the event values. Alternatively, one may choose to give preference to the left or right event:

```
lMerge :: Event a -> Event a -> Event a
rMerge :: Event a -> Event a -> Event a
```

If there is no possibility of simultaneous events, `merge` may be used, which generates an error if in fact two events occur together:

```
merge :: Event a -> Event a -> Event a
```

So far we have only considered pre-existing events. Some of these may come from external sources, such as a bumper switch or communications subsystem, but it is often convenient to define our own events. Yampa provides a variety of ways to generate new events, the most important being:

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

The expression `boolSF >>> edge` generates an event every time the signal from `boolSF` goes from `False` to `True` (i.e. the "leading edge" of the signal). For example, if `tempSF :: SF SimbotInput Temp` is a signal function that indicates temperature, then:

```
alarmSF :: SF SimbotInput (Event ())
alarmSF = tempSF >>> arr (>100) >>> edge
```

generates an alarm event if the temperature exceeds 100 degrees.

Here are a few other useful event generation functions:

```
never      :: SF a (Event b)
now        :: b -> SF a (Event b)
after      :: Time -> b -> SF a (Event b)
repeatedly :: Time -> b -> SF a (Event b)
```

`never` is an event source that never generates any event occurrences. `now v` generates exactly one event, whose time of occurrence is zero (i.e. now) and whose value is `v`. The expression `after t v` generates exactly one event, whose time of occurrence is `t` and whose value is `v`. Similarly, `repeatedly t v` generates an event every `t` seconds, each with value `v`.

To close this section, we point out that the discrete and continuous worlds interact in important ways, with switching, of course, being the most fundamental. But Yampa also provides several other useful functions to capture this interaction. Here are two of them:

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

The signal function `hold v` initially generates a signal with constant value `v`, but every time an event occurs with value `v'`, the signal takes on (i.e. "holds") that new value `v'`. The signal function `accum v0` is assentially an event stream transformer. Each input event generates one output event. If f_n is the function corresponding to the nth input event, then the value v_n of the nth output event is just $f_n\ v_{n-1}$, for $n \geq 1$, and with $v_0 =$ `v0`.

For example, the following signal function represents the number of alarms generated from `alarmSF` defined earlier:

```
alarmCountSF :: SF SimbotInput Int
alarmCountSF = alarmSF >>> arr (`tag` (+1)) >>> accum 0 >>> hold 0
```

Indeed, the `accum` followed by `hold` idiom is so common that it is predefined in Yampa:

```
accumHold :: a -> SF (Event (a -> a)) a
accumHold init = accum init >>> hold init
```

Exercise 5. Suppose `v :: SF SimbotInput Velocity` represents the scalar velocity of a simbot. If we integrate this velocity, we get a measure of how far the simbot has traveled. Define an alarm that generates an event when either the simbot has traveled more than `d` meters, or it has gotten stuck.

2.7 Recursive Signals

Note in Fig. 1 the presence of the `loop` combinator. Its purpose is to define recursive signal functions; i.e. it is a fixpoint operator. The arrow syntax goes one step further by allowing recursive definitions to be programmed directly, which the preprocessor expands into applications of the `loop` combinator. In this case the user must include the keyword `rec` prior to the collection of recursive bindings.

For example, a common need when switching is to take a "snapshot" of the signal being switched out of, for use in computing the value of the signal being switched into. Suppose that there is an event source
`incrVelEvs :: SF SimbotInput (Event ())` whose events correspond to commands to increment the velocity. We can define a signal function that responds to these commands as follows:

```
vel :: Velocity -> SF SimbotInput Velocity
vel v0 = proc inp -> do
  rec e <- incrVelEvs -< inp
      v <- drSwitch (constant v0) -< (inp, e `tag` constant (v+1))
  returnA -< v
```

Note that `v` is recursively defined. This requires the use of the `rec` keyword, and also the use of a delayed switch to ensure that the recursion is well founded. Also note that the recurring version of switch is used, because we want the velocity update to happen on *every* event. Finally, note the use of `tag` to update the value of an event.

The need for a delayed switch is perhaps best motivated by analogy to recursively defined lists, or streams. The definition:

```
ones = 1 : ones
```

expresses the usual infinite stream of ones, and is obviously well founded, whereas the list:

```
ones = ones
```

is obviously not well founded. The value of 1 placed at the front of the list can be thought of as a delay in the access of `ones`. That is the idea behind a delayed switch, although semantically the delay is intended to be infinitesimally small, and in the implementation we avoid introducing a delay that could affect performance.

Exercise 6. Redefine `vel` using `dSwitch` instead of `drSwitch`, and without using the `rec` keyword. (Hint: define `vel` recursively instead of defining `v` recursively.)

3 Programming the Robot Simulator

3.1 Robot Input and Output

Generally speaking, one might have dozens of different robots, some real, some simulated, and each with different kinds of functionality (two wheels, three wheels, four wheels, cameras, sonars, bumper switches, actuators, speakers, flashing lights, missle launchers, and so on). These differences are captured in the input and output types of the robot. For example, there is only one kind of simulated robot, or simbot, whose input type is `SimbotInput` and whose output type is `SimbotOutput`.

[Note: The code described in this section works with Yampa version 0.9 (and 0.9.x patches), but some changes are anticipated for use with future Yampa versions 1.0 and higher. In particular, the module names will change. In Yampa 0.9 they are still known under their old names (`AFrob`, `AFrobRobotSim`, etc.) for backwards compatibility reasons.]

We refer to the collection of Yampa libraries that are robot-specific as *AFrob*. The AFrob library was written to be as generic as possible, and thus it does not depend directly on the robot input and output types. Rather, type classes are used to capture different kinds of functionality. Each robot type is an instance of some subset of these classes, depending on the functionality it has to offer.

For example, `SimbotInput` is a member of the type classes shown in the upper half of Fig. 3, and `SimbotOutput` is a member of the lower ones. The types `Velocity`, `Distance`, `Angle`, `RotVel`, `RotAcc`, `Length`, `Acceleration`, `Speed`, `Heading`, and `Bearing` are all synonyms for type `Double`. Type `Position2` is a synonym for `Point2 Position`, where:

```
data RealFloat a => Point2 a = Point2 !a !a
     deriving Eq
```

We will give examples of the use of many of these operations and type classes in the examples that follow. Before doing so, however, there is one other detail to describe about the output classes. Note in Fig. 3 that the methods in the last two classes return a type `MR a`, where `a` is constrained to be a `MergeableRecord`. This allows one to incrementally specify certain "fields" of the record, and to merge them later. There are two key operations on mergeable records:

```
mrMerge    :: MergeableRecord a => MR a -> MR a -> MR a
mrFinalize :: MergeableRecord a => MR a -> a
```

```
-- Input Classes And Related Functions
------------------------------------------------------------------------
class HasRobotStatus i where
    rsBattStat :: i -> BatteryStatus    -- Curent battery status
    rsIsStuck  :: i -> Bool             -- Currently stuck or not

data BatteryStatus = BSHigh | BSLow | BSCritical
     deriving (Eq, Show)

-- derived event sources:
rsBattStatChanged  :: HasRobotStatus i => SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())

class HasOdometry i where
    odometryPosition :: i -> Position2  -- Current position
    odometryHeading  :: i -> Heading    -- Current heading

class HasRangeFinder i where
    rfRange    :: i -> Angle -> Distance
    rfMaxRange :: i -> Distance

-- derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance

class HasAnimateObjectTracker i where
    aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
    aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
    tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                                Maybe Char -> SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i => SF i (Event Char)

-- Output Classes And Related Functions
------------------------------------------------------------------------
class MergeableRecord o => HasDiffDrive o where
  ddBrake   :: MR o                          -- Brake both wheels
  ddVelDiff :: Velocity -> Velocity -> MR o  -- set wheel velocities
  ddVelTR   :: Velocity -> RotVel   -> MR o  -- set vel. and rot.

class MergeableRecord o => HasTextConsoleOutput o where
  tcoPrintMessage :: Event String -> MR o
```

Fig. 3. Robot Input and Output Classes

For example, the expression:

```
sbo :: SimbotOutput
sbo = mrFinalize
       (ddVelDiff vel1 vel2 `mrMerge` tcoPrintMessage stringEvent)
```

merges the velocity output with a console message.

For simbots, it turns out that velocity control and message output are the only two things that can be merged, so the use of the `MergeableRecord` class may seem like an overkill. However, for other robots there may be many such mergeable outputs, and the functionality thus offered is quite convenient.

When two common outputs are merged, the result depends on how the `mrMerge` and `mrFinalize` methods are defined to behave. The designer of a particular instance of these methods might signal an error, accept one output or the other (for example, merging two calls to `ddVelDiff` yields the value of the first one), or combine the two (for example, merging two calls to `tcoPrintMessage` results in both messages being printed in order).

3.2 Robot Controllers

To control a robot we must define a *robot controller*, which, for the case of simbots, must have type:

```
type SimbotController =
           SimbotProperties -> SF SimbotInput SimbotOutput
```

`SimbotProperties` is a data type that specifies static properties of a simbot. These properties are accessed abstractly in that `SimbotProperties` is an instance of the `HasRobotProperties` type class:

```
class HasRobotProperties i where
  rpType     :: i -> RobotType    -- Type of robot
  rpId       :: i -> RobotId      -- Identity of robot
  rpDiameter :: i -> Length       -- Distance between wheels
  rpAccMax   :: i -> Acceleration -- Max translational acc
  rpWSMax    :: i -> Speed        -- Max wheel speed

type RobotType = String
type RobotId = Int
```

The simulator knows about two versions of the simbot, for which each of these properties is slightly different. The `RobotType` field is just a string, which for the simbots will be either `"SimbotA"` or `"SimbotB"`. The remaining fields are self-explanatory.

To actually run the simulator, we use the function:

```
runSim :: Maybe WorldTemplate ->
          SimbotController -> SimbotController -> IO ()
```

where a `WorldTemplate` is a data type that describes the initial state of the simulator world. It is a list of simbots, walls, balls, and blocks, along with locations of the centers of each:

```
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
    OTBlock   { otPos  :: Position2 } -- Square obstacle
  | OTVWall   { otPos  :: Position2 } -- Vertical wall
  | OTHWall   { otPos  :: Position2 } -- Horizontal wall
  | OTBall    { otPos  :: Position2 } -- Ball
  | OTSimbotA { otRId  :: RobotId,    -- Simbot A robot
                otPos  :: Position2,
                otHdng :: Heading   }
  | OTSimbotB { otRId  :: RobotId,    -- Simbot B robot
                otPos  :: Position2,
                otHdng :: Heading   }
```

The constants `worldXMin`, `worldYMin`, `worldXMax`, and `worldYMax` are the bounds of the simulated world, and are assumed to be in meters. Currently these values are -5, -5, 5, and 5, respectively (i.e. the world is 10 meters by 10 meters, with the center coordinate being $(0, 0)$). The walls are currently fixed in size at 1.0 m by 0.1 m, and the blocks are 0.5 m by 0.5 m. The diameter of a simbot is 0.5 m.

Your overall program should be structured as follows:

```
module MyRobotShow where

import AFrob
import AFrobRobotSim

main :: IO ()
main = runSim (Just world) rcA rcB

world :: WorldTemplate
world = ...

rcA :: SimbotController  -- controller for simbot A's
rcA = ...

rcB :: SimbotController  -- controller for simbot B's
rcB = ...
```

The module `AFrob` also imports the Yampa library. The module `AFrobRobotSim` is the robot simulator.

Note that many robots may be created of the same kind (i.e. simbot A or simbot B) in the world template, but the *same* controller will be invoked for all of them. If you want to distinguish amongst them, simply give them different `RobotID`'s. For example, if you have three simbot A robots, then your code for controller `rcA` can be structured like this:

```
rcA :: SimbotController
rcA rProps =
  case rpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```

3.3 Basic Robot Movement

In this section we will write a series of robot controllers, each of type `SimbotController`. Designing controllers for real robots is both an art and a science. The science part includes the use of *control theory* and related mathematical techniques that focus on differential equations to design optimal controllers for specific tasks. We will not spend any time on control theory here, and instead will appeal to the reader's intuition in the design of functional, if not optimal, controllers for mostly simple tasks. For more details on the kinematics of mobile robots, see [3].

Stop, go, and turn. For starters, let's define the world's dumbest controller – one for a stationary simbot:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

Or we could make the simbot move blindly forward at a constant velocity:

```
rcBlind1 _ = constant (mrFinalize $ ddVelDiff 10 10)
```

We can do one better than this, however, by first determining the maximal allowable wheel speeds and then running the simbot at, say, one-half that speed:

```
rcBlind2 rps =
    let max = rpWSMax rps
    in constant (mrFinalize $ ddVelDiff (max/2) (max/2))
```

We can also control the simbot through `ddVelTR`, which allows specifying the simbot's forward and rotational velocities, rather than the individual wheel speeds. For a differential drive robot, the maximal rotational velocity depends on the vehicle's forward velocity; it can rotate most quickly when it is standing still, and cannot rotate at all if it is going at its maximal forward velocity (because to turn while going at its maximal velocity, one of the wheels would have to slow down, in which case it would no longer be going at its maximal velocity). If the maximal wheel velocity is v_{max}, and the forward velocity is v_f, then it is easy to show that the maximal rotational velocity in radians per second is given by:

$$\omega_{max} = \frac{2(v_{max} - v_f)}{l}$$

For example, this simbot turns as fast as possible while going at a given speed:

```
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
    let vMax = rpWSMax rps
        rMax = 2 * (vMax - vel) / rpDiameter rps
    in constant (mrFinalize $ ddVelTR vel rMax)
```

Exercise 7. Link `rcBlind2`, `rcTurn`, and `rcStop` together in the following way: Perform `rcBlind2` for 2 seconds, then `rcTurn` for three seconds, and then do `rcStop`. (Hint: use `after` to generate an event after a given time interval.)

The simbot talks (sort of). For something more interesting, let's define a simbot that, whenever it gets stuck, reverses its direction and displays the message `"Ouch!!"` on the console:

```
rcReverse :: Velocity -> SimbotController
rcReverse v rps = beh `dSwitch` const (rcReverse (-v) rps)
  where beh = proc sbi -> do
                stuckE <- rsStuck -< sbi
                let mr = ddVelDiff v v `mrMerge`
                           tcoPrintMessage (tag stuckE "Ouch!!")
                returnA -< (mrFinalize mr, stuckE)
```

Note the use of a `let` binding within a `proc`: this is analogous to a `let` binding within Haskell's monadic `do` syntax. Note also that `rcReverse` is recursive – this is how the velocity is reversed everytime the simbot gets stuck – and therefore requires the use of `dSwitch` to ensure that the recursion is well founded. (It does not require the `rec` keyword, however, because the recursion occurs outside of the `proc` expression.) The other reason for the `dSwitch` is rather subtle: `tcoPrintMessage` uses `stuckE` to control when the message is printed, but `stuckE` also controls the switch; thus if the switch happened instantaneously, the message would be missed!

If preferred, it is not hard to write `rcReverse` without the arrow syntax:

```
rcReverse' v rps =
  (rsStuck >>> arr fun) `dSwitch` const (rcReverse' (-v) rps)
  where fun stuckE =
          let mr = ddVelDiff v v `mrMerge`
                     tcoPrintMessage (tag stuckE "Ouch!!")
          in (mrFinalize mr, stuckE)
```

Exercise 8. Write a version of `rcReverse` that, instead of knowing in advance what its velocity is, takes a "snapshot" of the velocity, as described in Sec. 2.7, at the moment the stuck event happens, and then negates this value to continue.

Finding our way using odometry. Note from Fig. 3 that our simbots have *odometry*; that is, the ability of a robot to track its own location. This capability on a real robot can be approximated by so-called "dead reckoning," in which the robot monitors its actual wheel velocities and keeps track of its position incrementally. Unfortunately, this is not particularly accurate, because of the errors that arise from wheel slippage, uneven terrain, and so on. A better technique is to use GPS (global positioning system), which uses satellite signals to determine a vehicle's position to within a few feet of accuracy. In our simulator we will assume that the simbot's odometry is perfect.

We can use odometry readings as *feedback* into a controller to stabilize and increase the accuracy of some desired action. For example, suppose we wish to move the simbot at a fixed speed in a certain direction. We can set the speed easily enough as shown in the examples above, but we cannot directly specify the direction. However, we can read the direction using the odometry function `odometryHeading :: SimbotInput -> Heading` and use this to control the rotational velocity.

(A note about robot headings. In AFrob there are three data types that relate to headings:

1. `Heading` is assumed to be in radians, and is aligned with the usual Cartesian coordinate system, with 0 radians corresponding to the positive x-axis, $\pi/2$ the positive y-axis, and so on. Its normalized range is $[-\pi, \pi)$.
2. `Bearing` is assumed to be in degrees, and is aligned with a conventional compass, with 0 degrees corresponding to north, 90 degrees to east, and so on. Its normalized range is $[0, 360)$.
3. `Angle` is assumed to be in radians, but is a *relative* measure rather than being aligned with something absolute.

AFrob also provide conversion functions between bearings and headings:

```
bearingToHeading :: Bearing -> Heading
headingToBearing :: Heading -> Bearing
```

However, in this paper we only use headings and relative angles.)

Getting back to our problem, if h_d and h_a are the desired and actual headings in radians, respectively, then the heading error is just $h_e = h_d - h_a$. If h_e is positive, then we want to turn the robot in a counter-clockwise direction (i.e. using a positive rotational velocity), and if h_e is negative, then we want to turn the robot in a clockwise direction (i.e. using a negative rotational velocity). In other words, the rotational velocity should be directly proportioonal to h_e (this strategy is thus called a *proportionate controller*). One small complication to this scheme is that we need to *normalize* $h_d - h_a$ to keep the angle in the range $[-\pi, \pi)$. This is easily achieved using Yampa's `normalizeAngle` function. Here is the complete controller:

```
rcHeading :: Velocity -> Heading -> SimbotController
rcHeading vel hd rps =
    let vMax = rpWSMax rps
        vel' = lim vMax vel
        k    = 2
    in proc sbi -> do
         let he    = normalizeAngle (hd - odometryHeading sbi)
         let vel'' = (1 - abs he / pi) * vel'
         returnA -< mrFinalize (ddVelTR vel'' (k*he))

lim m y = max (-m) (min m y)
```

The parameter k is called the *gain* of the controller, and can be adjusted to give a faster response, at the risk of being too fast and thus being unstable. `lim m y` limits the maximum absolute value of `y` to `m`.

Before the next example we will first rewrite the above program in the following way:

```
rcHeading' :: Velocity -> Heading -> SimbotController
rcHeading' vel hd rps =
  proc sbi -> do
    rcHeadingAux rps -< (sbi, vel, hd)

rcHeadingAux :: SimbotProperties ->
                SF (SimbotInput,Velocity,Heading) SimbotOutput
rcHeadingAux rps =
    let vMax = rpWSMax rps
        k    = 2
    in proc (sbi,vel,hd) -> do
         let vel' = lim vMax vel
         let he   = normalizeAngle (hd - odometryHeading sbi)
         let vel'' = (1 - abs he / pi) * vel'
         returnA -< mrFinalize (ddVelTR vel'' (k*he))
```

In the original definition, `vel` and `hd` were constant during the lifetime of the signal function, whereas in the second version they are treated as signals in

`rcHeadingAux`, thus allowing for them to be time varying. Although not needed in this example, we will need this capability below.

As another example of using odometry, consider the task of moving the simbot to a specific location. We can do this by computing a trajectory from our current location to the desired location. By doing this continually, we ensure that drift caused by imperfections in the robot, the floor surface, etc. do not cause appreciable error.

The only complication is that we must take into account our simbot's translational inertia: if we don't, we may overshoot the target. What we'd like to do is *slow down* as we approach the target (as for `rcHeading`, this amounts to designing a proportionate controller). Here is the code:

```
rcMoveTo :: Velocity -> Position2 -> SimbotController
rcMoveTo vd pd rps = proc sbi -> do
    let (d,h) = vector2RhoTheta (pd .-. odometryPosition sbi)
        vel   = if d>2 then vd else vd*(d/2)
    rcHeadingAux rps -< (sbi, vel, h)
```

Note the use of vector arithmetic to compute the difference between the desired position `pd` and actual position `odometryPosition sbi`, and the use of `vector2RhoTheta` to convert the error vector into distance `d` and heading `h`. `vel` is the speed at which we will approach the target. Finally, note the use of `rcHeadingAux` defined above to move the simbot at the desired velocity and heading.

Exercise 9. `rcMoveTo` will behave a little bit funny once the simbot reaches its destination, because a differential drive robot is not able to maneuver well at slow velocities (compare the difficulty of parallel parking a car to the ease of switching lanes at high speed). Modify `rcMove` so that once it gets reasonably close to its target, it stops (using `rcStop`).

Exercise 10. Define a controller to cause a robot to follow a sinusoidal path. (Hint: feed a sinusoidal signal into `rcHeadingAux`.)

Exercise 11. Define a controller that takes a list of points and causes the robot to move to each point successively in turn.

Exercise 12. (a) Define a controller that chases a ball. (Hint: use the `aotBalls` method in class `HasAnimateObjectTracker` to find the location of the ball.) (b) Once the ball is hit, the simulator will stop the robot and create an `rsStuck` event. Therefore, modify your controller so that it restarts the robot whenever it gets stuck, or perhaps backs up first and then restarts.

Home on the range. Recall that our simbots have *range finders* that are able to determine the distance of the nearest object in a given direction. We will assume that there are four of these, one looking forward, one backward, one to the left, and one to the right:

```
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

These are intended to simulate four sonar sensors, except that they are far more accurate than a conventional sonar, which has a rather broad signal. They are more similar to the capability of a laser-based range finder.

With a range finder we can do some degree of autonomous navigation in "unknown terrain." That is, navigation in an area where we do not have a precise map. In such situations a certain degree of the navigation must be done based on local features that the robot "sees," such as walls, doors, and other objects.

For example, let's define a controller that causes our simbot to follow a wall that is on its left. The idea is to move forward at a constant velocity v, and as the desired distance d from the wall varies from the left range finder reading r, adjustments are made to the rotational velocity ω to keep the simbot in line. This task is not quite as simple as the previous ones, and for reasons that are beyond the scope of this paper, it is desirable to use what is known as a PD (for "proportionate/derivative") controller, which means that the error signal is fed back proportionately and also as its derivative. More precisely, one can show that, for small deviations from the norm:

$$\omega = K_p(r - d) + K_d(\frac{dr}{dt})$$

K_p and K_d are the proportionate gain and derivative gain, respectively. Generally speaking, the higher the gain, the better the reponse will be, but care must be taken to avoid responding too quickly, which may cause over-shooting the mark, or worse, unstable behavior that is oscillatory or that diverges. It can be shown that the optimal relationship between K_p and K_d is given by:

$$K_p = vK_d^2/4$$

In the code below, we will set K_d to 5. For pragmatic reasons we will also put a limit on the absolute value of ω using the limiting function `lim`.

Assuming all of this mathematics is correct, then writing the controller is fairly straightforward:

```
rcFollowLeftWall :: Velocity -> Distance -> SimbotController
rcFollowLeftWall v d _ = proc sbi -> do
  let r = rfLeft sbi
  dr <- derivative -< r
  let omega = kp*(r-d) + kd*dr
      kd    = 5
      kp    = v*(kd^2)/4
  returnA -< mrFinalize (ddVelTR v (lim 0.2 omega))
```

Exercise 13. Enhance the wall-follower controller so that it can make left and right turns in a maze constructed only of horizontal and vertical walls. Specifically:

1. If the simbot sees a wall directly in front of itself, it should slow down as it approaches the wall, stopping at distance d from the wall. Then it should turn right and continue following the wall which should now be on its left. (This is an inside-corner right turn.)
2. If the simbot loses track of the wall on its left, it continues straight ahead for a distance d, turns left, goes straight for distance d again, and then follows the wall which should again be on its left. (This is an outside-corner left turn.)

Test your controller in an appropriately designed world template.

Exercise 14. As mentioned in the derivation above, the `rcFollowLeftWall` controller is only useful once the robot is close to being on track: i.e. at the proper distance from the wall and at the proper heading. If the robot is too far from the wall, it will tend to turn too much in trying to get closer, which makes the left range finder see an even *greater* distance, and the system becomes unstable. Designing a more robust wall follower is tricky business, and is best treated as *multi-mode* system, where the robot first seeks a wall, aligns itself parallel to the wall, and then tries to follow it. Design such a controller.

Mass hysteria. As mentioned earlier, the simulator can handle a number of simbots simultaneously. Groups of robots can exhibit all kinds of interesting and productive group behavior (or possibly mass hysteria), limited only by the cleverness of you, the designer. We will describe one simple kind of group behavior here, leaving others (such as the soccer match described in Ex. 16) to you.

The behavior that we will define is that of *convergence*. Assume that all simbots are initially moving in arbitrary directions and speeds. Each simbot will look at the positions of all of the others, and move toward the centroid (i.e. average) of those positions. If each robot does this continuously and independently, they will all end up converging upon the same point.

To achieve this, recall first the `HasAnimateObjectTracker` class:

```
class HasAnimateObjectTracker i where
    aotOtherRobots :: i -> [(RobotType, RobotId, Angle, Distance)]
    aotBalls       :: i -> [(Angle, Distance)]
```

The first of these operations permits us to determine the angle and distance of each of the other simbots. By converting these measurements to vectors, we can add them and take their average, then use `rcHeading` to steer the robot toward the resulting centroid.

Other than dealing with numeric conversions, the final code is fairly straightforward:

```
rcAlign :: Velocity -> SimbotController
rcAlign v rps = proc sbi -> do
  let neighbors = aotOtherRobots sbi
      vs = map (\(_,_,a,d) -> vector2Polar d a) neighbors
      avg = if vs==[] then zeroVector
            else foldl1 (^+^) vs ^/ intToFloat (length vs)
      heading = vector2Theta avg + odometryHeading sbi
  rcHeadingAux rps -< (sbi, v, heading)
intToFloat = fromInteger . toInteger
```

When observing the world through robot sensors, one should not make too many assumptions about what one is going to see, because noise, varying light conditions, occlusion, etc. can destroy those expectations. For example, in the case of the simbots, the simulator does not guarantee that all other robots will be visible through the animate object tracker. Indeed, at the very first time-step, *none* are visible. For reasons of causality, sensor data is delayed one time-step; but at the very first time step, there is no previous data to report, and thus the animate object tracker returns an empty list of other robots. This is why in the code above the list `vs` is tested for being empty.

Exercise 15. Write a program for two simbots that are traveling in a straight path, except that their paths continually interleave each other, as in a braid of rope. (Hint: treat the velocities as vectors, and determine the proper equations for two simbots to circle one another while maintaining a specified distance. Then add these velocities to the simbots' forward velocities to yield the desired behavior.)

Exercise 16. Write a program to play "robocup soccer," as follows. Using wall segments, create two goals at either end of the field. Decide on a number of players on each team, and write controllers for each of them. You may wish to write a couple of generic controllers, such as one for a goalkeeper, one for attack, and one for defense. Create an initial world where the ball is at the center mark, and each of the players is positioned strategically while being on-side (with the defensive players also outside of the center circle). Each team may use the same controller, or different ones. Indeed, you can pit your controller-writing skills against those of your friends (but we do not recommend betting money on the game's outcome).

4 Acknowledgements

We wish to thank all the members of the Yale Haskell Group for their support and feedback on many of the ideas in this paper. In particular, thanks to Zhanyong Wan, who undoubtedly would have been deeply involved in this work if he had not been so busy writing his thesis. Also thanks to Greg Hager and Izzet Pembeci at Johns Hopkins, who believed enough in our ideas to try them out on real robots. Finally, thanks to Conal Elliott, who started us on our path toward continuous nirvana, despite discrete moments of trauma.

References

1. Richard S. Bird. A calculus of functions for program derivation. In David A. Turner, editor, *Reseach Topics in Functional Programming*. Adison-Wesley, 1990.
2. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proc. of the 2001 Haskell Workshop*, September 2001.
3. Gregory Dudek and Michael Jenkin. *Computational Principles of Mobile Robots*. Cambride University Press, New York, 2000.
4. Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.
5. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.
6. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
7. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
8. Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *ACM SIGPLAN 2002 Haskell Workshop*, October 2002.
9. Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.
10. 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.
11. John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.
12. John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.
13. John Peterson, Zhanyong Wan, Paul Hudak, and Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001. Available at `http://www.haskell.org/frp/manual.html`.
14. Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. Int'l Conference on Software Engineering*, May 1999.
15. Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD thesis, Department of Computer Science, Yale University, December 2002.
16. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–252, Vancouver, BC, Canada, June 2000. ACM, ACM Press.
17. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, September 2001. ACM.
18. Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages*. ACM, Jan 2002.