Lambda in Motion: Controlling Robots With Haskell

John Peterson[1], Paul Hudak[1], and Conal Elliott[2]

[1] Yale University, peterson-john@cs.yale.edu and paul.hudak@yale.edu
[2] Microsoft Research, conal@microsoft.com

Abstract. We present our experiences using a purely functional language, Haskell, in what has been traditionally the realm of low-level languages: robot control. *Frob* (*F*unctional *Rob*otics) is a domain-specific language embedded in Haskell for robot control. Frob is based on *Functional Reactive Programming* (FRP), as initially developed for Fran, a language of reactive animations. Frob presents the interaction between a robot and its stimuli, both onboard sensors and messages from other agents, in a purely functional manner. This serves as a basis for composable high level abstractions supporting complex control regimens in a concise and reusable manner.

1 Introduction

Robotics is a an excellent problem domain to demonstrate the power and flexibility of declarative programming languages. Among the more interesting problem areas in this domain are control systems, constraint solving, reactive programming, sensor fusion, and real-time control. We have developed *Frob* (for *F*unctional *Rob*otics) as a domain-specific language, embedded in Haskell [PH97], for use in robotic systems. Frob hides the details of low-level robot operations and promotes a style of programming largely independent of the underlying hardware.

Our contributions include:

- Identification of factors that distinguish robotics programming from other uses of functional reactive programming.
- A characterization of certain aspects of robotics programming as *control system design*, and the use of continuous, mutually recursive equations in Frob to describe them.
- High-level abstractions that generalize patterns of control, including a monadic sequencing of control tasks.
- Combinators that intelligently fuse competing views of sensor data or strategies for robot control.

2 The Problem Domain

Our overall goal is to study control languages for general robotic systems. At present we have focused on a specific hardware configuration: a set of small mobile robots communicating via a radio modem. These robots are Nomadics Superscouts controlled by an onboard Linux system. The robots contain three types of sensors: a belt of 16 sonars, bumpers for collision detection, and a video camera. The sonars give only an approximate idea of how close the robot is to an object: objects which are too short or do not reflect sonar are not detected. The drive mechanism uses two independent drive wheels and is controlled by setting the forward velocity and turn rate. The robot keeps track of its location via dead reckoning.

The robots are controlled by a Frob program running on top of Hugs, a small portable Haskell interpreter. Libraries supplied by Nomadics, written in C++, perform low-level control and sensor functions. These libraries are imported into Hugs using GreenCard II, a C/C++ interface language for Haskell. The running Frob program interacts with a remote console window via telnet, allowing the program to respond to keyboard input as well as its sensors. Robots may also send messages to each other via the radio modem, allowing us to explore cooperative behaviors.

In developing Frob we have relied on our experience working with *Fran*, a DSL embedded in Haskell for *f*unctional *r*eactive *an*imation [EH97, Ell98b, Ell98a]. Specifically, we have borrowed the core *behavior* and *reactivity* components of Fran, which together we call *FRP* (for *functional reactive programming*). However, FRP itself is not enough to deal with the additional complexities that arise in the realm of robotics (for one thing, an animated figure will always do what you ask; but a robot will not!). Amongst the distinguishing features of Frob are:

1. Robotics is characterized by multiple, tightly coupled closed-loop control systems. The equational and highly recursive nature of functional programming lends itself well to this task.
2. Robots live in a real world where errors of all sorts can occur almost anywhere at anytime. In Frob, we can build flexible error handling schemes directly into our abstractions.
3. An equally important kind of failure arises from unreliable/noisy data from sensors, requiring filters or correlation measures with other sensors. Frob must deal intelligently with uncertainly in its inputs.
4. Robots are full of gadgets that often undergo configuration changes. These devices may run at varying rates and require various degrees of responsiveness from the controller. This raises the need for flexibility, modularity, and platform independence in our software.
5. Frob programs must allow precise control over many "lower-level" aspects of the system such as sensor sampling rates and other hardware-level concerns.
6. Finally, there is a need for *planning and strategizing*. One of our long-term goals is to program robots with some reasonable level of intelligence. High-level abstractions in a functional programming setting should help us once again to step back from low-level details and concentrate on the big picture.

3 Frob and Functional Reactive Programming

FRP defines two essential representations used in Frob: *behaviors* and *events*. Behaviors are conceptually continuous quantities that vary over time, similar to the following type:

```
type Behavior a = Time -> a   -- simplified
```

For example, a value of type `Behavior SonarReading` represents a time-varying sonar reading, `Behavior Velocity` represents a time-varying robot velocity, and `Behavior Time` represents time itself.

Using behaviors to model robotic components hides unnecessary details that clutter the controller software. Continuous values may be combined without regard to the underlying sampling rate, eliminating some of the difficulties encountered in multi-rate systems. For example, consider using a robot-mounted camera to determine the position of an object. The tracking software yields a vector relative to the robot position; the absolute position of the tracked object combines this relative position with the current robot position and orientation. As continuous signals, these values can be added directly even when the camera is clocked at a different rate than positional samples.

Not all values are best represented in a continuous manner, however. For example, the robot bumpers generate discrete events rather than continuous values. Similarly, devices such as the keyboard are best kept in the discrete domain. This leads to Frob's notion of an *event*, which can be thought of as a stream of time/value pairs:

```
type Event a = [(Time,a)]   -- simplified
```

For example, an `Event BumperEvent` occurs when one of the robot bumper switches is activated, and an `Event Char` occurs when a console key is pressed.

The interaction between behaviors and events is the basis of Frob's notion of *reactivity*. Events and behaviors may change course in response to a stimulating event. Fran (and Frob) define a large library of useful built-in combinators, allowing complex behaviors to be described and combined in useful ways. For example, this function:

```
goAhead :: Robot -> Time -> WheelControl
goAhead r timeMax =
  forward 30 `untilB`
      (predicate (time > timeMax) .|.
       predicate (frontSonarB r < 20))) -=> stop
```

can be read, for robot `r`: "Move forward at a velocity of 30 cm/sec, until either time exceeds the limit, `timeMax`, or an object appears closer than 20 cm, at which point stop." The operations used here are typical FRP primitives:

- `untilB` changes behavior in response to an event,
- `predicate` generates an event when a condition becomes true,

– `.|.` interleaves two events streams, and
– `-=>` associates a new value (in this case the behavior `stop`) with an event.

.

Although the interaction between Frob and the robot sensors is naturally implemented in terms of events (discrete samplings of sensor values), it is convenient to smooth out these values into a continuous behavior by extrapolating the discrete readings. Conversely, robot controls defined in a continuous manner must be sampled into discrete messages to the robot. It is generally more convenient to deal with continuous behaviors rather than discrete events. Frob generally provides both views of input sensors, both discrete and continuous, to the programmer, providing a choice of the most appropriate representation.

3.1 A Wall Follower

We now turn to a simple example of a robot control program: wall following. In this example, two sensor readings guide the robot: a front sensor determines the distance to any obstacle in front of the robot, and a side sensor measures the distance to the wall. In addition, the motor controller provides the actual robot velocity (this may differ from the velocity requested of the motors). Domain engineers typically describe such a task using differential equations. The equations describing this controller are:

$$
\begin{aligned}
v &= \sigma(v_{\max}, f - d^*) \\
\omega &= \sigma(\sin(\theta_{\max}) * v_{\mathrm{curr}}, s - d^*) - \dot{s}
\end{aligned}
$$

where $\sigma(x, y)$ is the limiting function $\sigma(x, y) = \max(-x, \min(x, y))$, d^* is the desired "setpoint" distance for objects to the front or side of the robot, $v_{\max}$ and $\theta_{\max}$ are the maximum robot velocity and body angle to the wall, respectively, and $\dot{s}$ denotes the derivative of s. In Frob, these equations become:

```
type FloatB       = Behavior Float
type WheelControl = Behavior (Float, Float)

basicWallFollower ::
    FloatB -> FloatB -> FloatB -> FloatB -> WheelControl

basicWallFollower vel side front setpoint =
 pairB v omega  -- creates a behavior of tuples
   where
    v           = limit vmax (front - setpoint)
    omega       = targetSideV - derivative side
    targetSideV = limit (vel * sin maxAngleToWall)
                        (setpoint - side)

limit high x = (-high) `max` x `min` high
```

The type `WheelControl` defines the output of this system, a pair of numbers defining the forward velocity and turn rate of the robot. Thus `basicWallFollower` is a direct encoding of the control equations, with `front` and `side` as the sonar readings, `vel` is the robot velocity, and `setpoint` as the distance the robot attempts to maintain from the wall.

The strategy expressed here is fairly simple: the robot travels at a maximum velocity until blocked in front, at which time it slows down as it approaches an obstacle. The value `targetSideV` is the rate that the robot should ideally approach the wall at; note this will be zero when `side` matches `setpoint`. This is limited by `maxAngleToWall` to prevent the robot from turning too sharply toward or away from the wall. The rotation rate, `omega`, is determined by the difference between the desired approach rate and the actual approach rate, as measured by taking the derivative of the side sonar reading.

It is important to emphasize the declarative nature of this short Frob program; it is essentially a rewrite of the differential equation specification into Haskell syntax. There is no loop to iteratively sample the sensors, compute parameters, update control registers, etc. In general, details pertaining to the flow of time are hidden from the programmer. Expressions such as `targetSideV - derivative side` represent operations on behaviors; overloading permits us to write them clearly and succinctly. Some operators, notably `derivative` in this example, directly exploit the time-varying nature of the signals. Finally, this code is independent of the kind of sensors used to measure the distances.

We demonstrate the use of reactivity by extending this example to pass control to a new behavior, represented as a continuation, when either the end of the side wall is encountered or an obstacle blocks the robot. A boolean value is passed into the continuation to distinguish the two cases.

```
wallFollower :: (Bool -> WheelControl) ->
       FloatB -> FloatB -> FloatB -> FloatB -> WheelControl
wallFollower cont side front rate dist =
  basicWallFollower side front rate dist `untilB`
    ((predicate (side > sideThreshold))  -=> cont True ) .|.
     ((predicate (front < frontThreshold)) -=> cont False)
```

This can be read: "Follow the wall until the side reading is greater than its threshold or the front reading is less than its threshold, then follow behavior `cont`."

4 More Abstractions

So far, we have demonstrated the ability of FRP to directly encode control equations and have used basic reactivity (`untilB`) to shape the overall system behavior. However, the real power of functional programming lies in the ability to define new, higher-level abstractions based on these primitive operators. Since Frob is embedded in Haskell, the full power of higher-order functional programming is available to the user. A number of useful abstractions are also pre-defined by Frob. In this section we examine some of these "robotic building blocks".

4.1 A Task Monad

Although FRP's reactivity is capable of describing arbitrary reactive behaviors, it is useful to add a layer on top of the basic FRP operations to lend more structure to the system. Indeed, the basic `untilB` operator is somewhat like a general "goto" construct: fundamental yet easily abused. It is often natural to think in terms of sequential *tasks*. The `wallfollower` function defined in Sect. 3 is an example of this: once the wall following task is complete, it calls a continuation to initiate the next task. We can alternatively use a *monad* to express sequential task combination. We also incorporate a number of other useful features into this monad, including:

- Exception handling.
- Implicit capture of task start time and robot state, allowing the task to be defined relative to the time or robot position at the start of the task. Tasks such as "wait 10 seconds" or "turn 90 degrees right" require this initial state information.
- Implicit propagation of the associated robot definition from task to task.
- A form of interrupt monitoring that adds interrupt events to a given task.

This monad hides the details of task sequencing, simplifying programming. Tasks are represented by the following types:

```
type TState = (Robot, RState, Event RoboErr)
type RState = (Time, Point2, Angle)
data Task b e =
  Task (TState -> (TState -> Either RoboErr e -> b) -> b)
```

The type `b` is the behavior defined by the task; `e` is the type of value returned at task completion. The task state, `TState`, contains the task's associated robot, the state of the robot at the start of the task, and a locally scoped error-catching event. The robot state, of type `RState`, saves the time, location, and heading of the robot at the start of the task. We use standard monadic constructions combining a state monad and an exception monad; the monad instance for `Task a` is straightforward:

```
instance Monad (Task a) where
  Task t1 >>= u =
    Task (\ts cont ->
            t1 ts (\ts' res ->
                     case res of
                       Right v -> case u v of
                                    Task t2 -> t2 ts' cont
                       Left l -> cont ts' (Left l)))
  return k = Task (\ts cont -> cont ts (Right k))
```

The state is managed in the underlying task functions. These are created by the `mkTask` function using a triple consisting of

1. the behavior during the task,
2. an event defining successful completion of the task, and
3. another event used to raise an error, if any.

Specifically:

```
mkTask :: (Robot -> RState ->
            (Behavior a, Event b, Event RoboErr))
          -> Task a b
mkTask f =
  Task (\(r,rstate,err) nextTask ->
          case f r rstate of
            (beh, success, failure) ->
               beh `untilB`
               ((success ==> Right) .|.
                (failure .|. err) ==> Left)
                `snapshot` stateOf r
                ==> \(res, rs') -> nextTask (r,rs',err) res)

stateOf r = pairB time (pairB (positionB r) (orientationB r))
```

This function defines the basic structure of a task: a behavior which continues until either a termination event or an error event stops it. The `snapshot` function captures the state of the robot, passing it into the next task. The termination event managed by the monad, `err`, supplies error handling from outside the context of the current task. This allows construction of composite tasks which share a common error exit; while `failure` defines an error event specific to a single task, the monad allows failure events to scope over larger sets of tasks.

These functions are the basic task combinators; their purpose is generally obvious from the type signature:

```
taskCatch     :: Task a b -> (RoboErr -> Task a b) -> Task a b
taskError     :: RoboErr -> Task a b  -- Raise an error

withTaskError :: Task a b -> Event RoboErr -> Task a b
timeLimit     :: Task a b -> Time -> b -> Task a b

getStartTime  :: Task a Time
getInitialPos :: Task Point2
getRobot      :: Task a Robot

runTask       :: Robot -> RState -> Task a b -> Behavior a
              -- Generate an error if the task terminates
```

The `withTaskError` and `timeLimit` functions use locally scoped error handlers (part of the `TState` type) to add termination conditions to an existing task, either with an arbitrary error event (`withTaskError`) or by aborting the task with a specified return value if it does not complete in a specified time period.

The `runTask` function converts a task into an ordinary behavior, aborting if the task terminates.

Using task functions, we now define interesting, easily understood robot control primitives represented as tasks and sequenced using Haskell's `do` notation. For example, using the `mkTask` function, the wall follower of Sect. 3.1 may be defined as follows:

```
wallFollower1 :: FloatB -> FloatB -> FloatB ->
                 FloatB -> Task RControl Bool
wallFollower1 side front rate d =
 mkTask
  (\r _ -> (wallFollower front side d rate,
           predicate (side > sideThreshold) -=> True .|.
           predicate (front < frontThreshold) -=> False,
           crash r -=> HitBumper)
```

This adds a new feature to the wall follower: if the robot's bumper, represented by the `crash` event, is triggered the error handler is called. The value `HitBumper` is part of the `RoboErr` type and tells the error handler the source of the error.

The use of a task monad does not imply single-threadedness. Tasks simply define behaviors; different tasks may be running simultaneously to control different parts of the robot. Tasks may be combined in the same manner that other behaviors are combined.

Here we present a few other common operations encoded as tasks. This turns the robot away from an obstacle:

```
turnAway :: Task WheelControl ()
turnAway = mkTask (\r _ -> (constantB (0, 0.1),
                            predicate (frontSonar r >* 30),
                            neverE))
```

The robot turns at a constant rate until the front sonar indicates that there is at least 30cm of open space in front of the robot.

This task turns 90 degrees:

```
turnRight :: Task WheelControl ()
turnRight = mkTask (\r rstate ->
     let target = constantB (orient rstate +@ (90 # deg)) in
         (constantB (0, 0.1),
          predicate (orientationB r === target),
          neverE))
```

The `===` operator is "nearly equal" comparison; exact comparison with the target orientation is not likely to succeed. The `+@` operation is angle addition; the result is normalized to the 0 to 2π range.

This task drives the robot to a specific location. We start with a behavior to drive the wheels:

```
goToward :: Point2B -> Robot -> Behavior RControl
goToward p r = pairB vel ang
 where
  err        = p .-. robotPosB r -- vector to target
  (delta, a) = vector2PolarCoords err  -- changed to polar
  da         = a -@ robotHeadingB r
  vel        = limit (maxVelocity r) (k1 * delta) -- velocity
  ang        = limit (maxTurn r) (k2 * da)        -- turn rate
```

Again, this program corresponds closely to the equations a a domain engineer would use to specify this system. Adding an event that detects arrival (within a suitable radius) yields a task:

```
travelTo :: Point2B -> Task RControl ()
travelTo p = mkTask (\r -> goToward p r,
                           predicate (dist p r < howClose),
                           senseBlocked r -=> Blocked)
```

This definition chooses (somewhat arbitrarily) to call the error handler when the sonars indicate obstacles ahead of the robot.

Finally, this function iterates a task until successful completion, using an error handler that restarts it repeatedly on failure.

```
keepTrying :: Task a b -> Task a c -> Task a b
keepTrying t err = t `taskCatch`
                         \_ -> do err; keepTrying t err
```

We use this strategy to combine to previously defined tasks into a composite behavior:

```
goto place = keepTrying (travelTo place) turnAway
```

As defined earlier, `travelTo` moves toward a specific location, terminating normally on arrival but raising an error if the robot is blocked. In this example, each time the robot 'steps aside' in `turnAway`, it tries again once the obstacle is out of the way.

4.2 Fusion

A common idiom in the robotics world is *fusion*. In general, fusion is the integration of routines which either observe similar parts of the real world or control the same (or similar) parts of the robot behavior. Some examples are:

- Combining sensor information into a unified picture of the outside world. For example, a robot with both vision and sonar has two different views of the same environment. Combining these views into a unified picture of the outside world is a crucial aspect of robotic control.

– Resolving conflicting doctrines. For example, an overall goal of traveling to a specific destination may conflict obstacles in the shortest path. The obstacle avoider may present a set of alternative paths; knowledge of the overall goal is needed to choose the route likely to be the best.
– Combining the behavior of more than one robot into an overall coordinated behavior.

Frob supports simple, reusable definitions of general fusion strategies.

One sort of fusion blends competing goals rather than selecting a current goal of attention. Perhaps the simplest example of this uses *force vectors*: imaginary forces pushing on the robot. These forces combine through vector addition. The following code assigns a force to a sonar reading:

```
sonarforce :: Robot -> SonarReading -> Vector2
sonarforce robot sonars =
    sum (zipWith f sonar (sonarAngles r) sonars)
  where
    f reading angle = vector2polar (distToForce reading) angle
    distToForce d | d >= maxSonar r = 0
                  | d < minValidSonar = 0
                  | otherwise = (-1) / (d*d)
```

This generates a force that increases as the sonar distance decreases and points in a direction opposite of the angle of the sonar (hence the `-1`). This next function generates a force pushing towards a goal, limited by a maximum force:

```
forceToGoal :: Point2 -> Point2 -> Vector2
forceToGoal there here = f where
  err        = there .-. here
  (power, a) = vector2PolarCoords err  -- changed to polar
  f          = vector2polar (limit power maxForce) a
```

Finally, this defines travel to destination, this time with obstacle avoidance:

```
goTowardF :: Point2B -> Robot -> Behavior RControl
goTowardF p r = followForce f r
 where f = lift1 (sonarForce r) (sonarB r) +
           (lift2 forceToGoal) p (positionB r)
```

The `goTowardF` function combines forces generated by obstacles on the sonars and a pull toward the destination to yield a simple controller that blends the competing goals of obstacle avoidance and travel to a specific place. The `lift` functions are necessary to convert scalar functions such as `sonarForce` into operations on behaviors. Not shown is the function `followForce`; it drives the robot based on the direction of force.

Sensor fusion is also neatly expressible in Frob. Consider an example taken from a robotic soccer competition. A team of vision-equipped robots tracks the soccer ball. The tracking software on each robot determines both the location of the ball when visible (based on the angle and the apparent size of the ball) and also assigns a confidence value to the reading, based on how sure the tracker is of the ball position. Each robot broadcasts the output of its tracker and these differing views of the ball are fused. This code generalizes this concept:

```
type WithConfidence a = (Float, a)
type WithConfidenceB a = Behavior (Float, a)
mergeB :: WithConfinenceB a -> WithConfinenceB a ->
          WithConfinenceB a
trackAll :: [Robot] -> WithConfidenceB Point2
trackerAll team  = foldr1 mergeB (map myTracker team)
```

We use the behavioral form of `withConfidence` to create a confidence varying over time. We decay confidence values, preferring recent low confidence readings over older high confidence readings. The following sampling code converts events (sensor readings) into a continuous behavior with linearly decaying confidence:

```
eventToBehaviorC :: Event (Float,a) -> a -> WithConfidenceB a
eventToBehaviorC e init =
    switcher (constantB (0, init)) e'
  where
    e' = e `withTimeE` \(eventTime, (c,val)) ->
                 pairB (c * decay eventTime) (constantB val)
    decay t0 = (1 - (time - constantB t0) / decayInterval)
               `max` 0
```

The `switcher` function assembles a behavior piecewise, with each event defining a new segment of the behavior. Initially the behavior is undefined until the first sensor reading arrives. At each occurance, a pair containing the sampled value, a constant, and its confidence, a linearly decaying value, is generated.

5 Implementing Frob

Frob is built by adding two components to the basic FRP engine, used unaltered from the Fran distribution. Underneath FRP is a layer that mediates between the IO actions that interact directly with the robot and the events native to FRP. On top of FRP are the various Frob abstractions for robotics, including coercions between the event streams generated by sampling and continuous behaviors.

Communication from FRP to the outside world is handled by event "listeners". A listener is a function that gets invoked on every occurrence of an event to which it is attached. That is, given an event of type `Event a`, its listeners are Haskell functions of type `a -> IO ()`. This mechanism is how information is moved out of the FRP domain into the robot.

Another mechanism, called "repeaters", supports getting information from the outside world *into* the FRP domain. A repeater is just a linked pair of listener and event. The event repeats whatever the listener is told. Creation of an FRP event based on some external real-world event is done by invoking `newRepeater` of type `IO (Listener a, Event a)` to get a pair `(l,e)`. Then whenever the external event occurs, imperative code tells the listener `l`, which causes the FRP event `e` to occur.

During initialization, event streams for all input sources are created using `newRepeater`. Listeners are attached to the output events of the user's program to forward each external event of interest. New repeaters and listeners may also be created on the fly, allowing new event streams to appear as the program executes.

Finally, the issue of *clocking* must be addressed. The timing of the input events (repeaters) defines the clocking of the system. There are no apriori constraints on this clocking: a repeater may be called at any time or rate; one sensor may be sampled frequently and another only occasionally. Input events may be driven by either interrupts or by polling. Once an event is injected into the system by a repeater, all computation contingent on that event is immediately performed, perhaps resulting in calls to repeaters.

At present, we use the simplest possible clocking scheme: a single heartbeat loop that runs as fast as possible, sampling all inputs and pushing their values into the system. The speed of the heartbeat loop depends on how long it takes to handle the incoming events. This is not ideal; if the heartbeat goes too slow the responsiveness of the system suffers. In the future we hope to explore more complex clocking schemes.

The type `Robot` is a container for all of the robot state, both dynamic (the events and behaviors defined by its sensors) and static (the description of hardware configuration). The dynamic values are generally provided in two forms: as the raw events generated by the hardware and as continuous behaviors. This "smoothing" of the event streams into a continuous form allows the sensors to be represented in a convenient way. A typical definition of `Robot` is:

```
data Robot = Robot {position     :: Event Point2,
                    positionB    :: Behavior Point2,
                    orientation  :: Event Float,
                    orientationB :: Behavior Float,
                    sonar        :: Event SonarReading
                    sonarB       :: Behavior SonarReading
                    maxVelocity  :: Float,
                    sonarAngles  :: [Float],
                    ...}
```

Similarly, the type `RobotControl` encapsulates all system outputs. The wheel controller, for example, is contained inside this structure. Output sampling is similar to input smoothing; here continuous behaviors are sampled (using the `snapshot` function) into discrete events. Both smoothing and sampling take place within the purely functional domain of FRP; only the listeners and talkers are relegated to the imperative world.

This code sketches the basic structure of Frob's main loop:

```
runRobot :: (Robot -> RobotControl) -> IO ()
```

```
runRobot f = do
   -- Create repeaters for all input values
   (inputEv1, listener1) <- newRepeater
     ...
   -- Combine all inputs into type "Robot"
   let robot = addSmoothing (makeRobot inputEv1 ...)
       -- Define output behaviors
       control = f robot
       -- convert to events
       (outputEvent1,...) = sampleBehaviors control
   -- Set up listeners for all output events
   addListener outputEvent1 (\t value -> ...)
   ...
   -- enter the heartbeat loop
   loop
 where loop =
     do -- for each repeater, get robot data and post to event
        i1 <- getRobot1
        listener1 i1
        ...
        loop
```

6 Experience

We have built a number of small controllers using Frob. Our experiences to date with the practical side of the system include:

- It was easy to use functional reactive programming to model our interaction with the outside world. Packaging up the robot controls as FRP events was simple and resulted in a much more declarative programming style than would be possible with an imperative language.
- Frob programs are far smaller than corresponding C++ programs for the relatively small systems developed so far. Abstractions such as tasks are much easier to implement and use in Frob than in C++. Experience with computer vision applications suggests that these same abstractions can be directly encoded in C++ (making judicious use of templates to handle polymorphism) but this requires considerably more effort and in most cases is not the obvious implementation choice in C++.
- The experimental nature of our robot control applications makes rapid prototyping particularly essential. New behaviors can be written quickly and succinctly in Frob, allowing the developer to spend more time actually experimenting with the robot.
- Performance has not been a problem, in spite of the fact that we are running Haskell using an interpreter (Hugs). The feedback control systems driving

the robot are very robust and slow clock rates or garbage collection delays do not perturb the system enough to cause any real problems.

7 Related Work

Several researchers have found declarative languages well suited for modeling pictures, 3D models, and even music. Examples include [Hen82, LZ87, Bar91, ZLL+88, FJ95, HMGW96]. Arya used a functional language to model 2D animations as lazy lists of pictures constructed using list combinators [Ary94]. While this work was quite elegant, the use of lists implies a discrete model of time, which is somewhat unnatural. The TBAG system modeled 3D animations as functions over continuous time, using a "behavior" type family [ESYAE94]. Although behaviors were based on continuous time, reactivity was handled imperatively through constraint assertion and retraction. Fran and DirectAnimation both grew out of the ideas in an earlier design called ActiveVRML [Ell96].

There has also been related work on concurrency and reactivity. CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called "events" [Rep91]. In CML, events are ultimately used to perform an action, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. Concurrent Haskell [JGF86] extends Haskell with a small set of primitives for explicit concurrency, designed around monadic I/O. While this system is purely functional in the technical sense, its semantics has a strongly imperative feel. In contrast, modeling entire behaviors as implicitly concurrent functions of continuous time yields what we consider a more declarative feel.

8 Conclusions

Assessing the effectiveness of Frob in the realm of robotics, it has succeeded in a number of ways:

- The basic building blocks of robotic systems such as differential equations, reactivity, and state machines, are all directly supported by Frob, allowing systems to be programmed directly from their specification.
- FRP effectively hides the imperative nature of the underlying robotic system, supporting a declarative programming style.
- The low-level details of the robotic system have been hidden under abstractions, allowing programming in a manner that is largely independent of the specific underlying hardware.
- We have been able to define a number of general control strategies in a modular and reusable way.

We are now in the process of incorporating vision-based controllers into our framework. We plan to use Frob for more complex systems soon. The latest release of our system is available on the web at `haskell.org/frob`.

9 Acknowledgements

This research was supported by NSF Experimental Software Systems grant CCR-9706747.

References

[Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.

[Bar91] Joel F. Bartlett. Don't fidget with widgets, draw! Technical Report 6, DEC Western Digital Laboratory, May 1991.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[Ell96] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996.

[Ell98a] Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, July 1998. Extended version with animations at `http://research.-microsoft.com/~conal/fran/{tutorial.htm,tutorialArticle.zip}`.

[Ell98b] Conal Elliott. Fran user's manual. `http://research.microsoft.com/-~conal/Fran/UsersMan.htm`, July 1998.

[ESYAE94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *Proceedings of SIGGRAPH '94*, pages 421–434. ACM SIGGRAPH, July 1994.

[FJ95] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of Glasgow Functional Programming Workshop*, July 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*, pages 179–187. ACM, 1982.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[JGF86] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM SIGPLAN, January 1986.

[LZ87] Peter Lucas and Stephen N. Zilles. Graphics in an applicative context. Technical report, IBM Almaden Research Center, July 1987.

[PH97] John Peterson and Kevin Hammond. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1997.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Conference on Programming Language Design and Implementation*, pages 293–305. SIGPLAN, June 1991.

[ZLL+88] S.N. Zilles, P. Lucas, T.M. Linden, J.B. Lotspiech, and A.R. Harbury. The Escher document imaging model. In *Proceedings of the ACM Conference on Document Processing Systems*, pages 159–168, December 1988.