Monadic Robotics
John Peterson
Greg Hager
Yale University
peterson-john@cs.yale.edu hager-greg@cs.yale.edu

Abstract

We have developed a domain specific language, Frob (for Functional Robotics), for the construction of robot controllers. The semantic basis for Frob is Functional Reactive Programming (FRP), a purely functional model of continuous time, interactive systems. FRP supplies two essential abstractions: behaviors and events. Behaviors are continuous time-varying values and while events are streams of discrete occurrences in time. On this foundation, we have constructed abstractions specific to robotics. One important element of Frob is the task, which defines a basic unit of work in terms of a continuous behavior and a terminating event.

This paper examines two interrelated aspects of Frob. We study the design of systems based on FRP and how abstractions defined with FRP can capture essential domain-specific concepts. Also, we demonstrate an application of monads to organize essential system semantics in a modular way, allowing new capabilities to be added without pervasive changes to the system. Frob tasks are incorporated into a monadic framework, hiding the details of their semantic underpinnings from Frob users.

We demonstrate a number of basic robot control algorithms specified using Frob. These programs are clear, succinct, and modular, demonstrating the power of our approach.

1 Introduction

A successful DSL combines the vocabulary (values and primitive operations) of an underlying domain with abstractions that capture useful patterns in the vocabulary. Ideally, these abstractions organize the vocabulary into structures that support clarity and modularity in the domain of interest. In robotic control, this basic vocabulary is quite simple: in consists of feedback systems connecting the robot sensors and its effectors. A more difficult task is to build complex behaviors by sequencing among various control disciplines, guided by overall plans or objectives. Controllers must be robust and effective, capable of complex interactions with an uncertain environment. While basic feedback systems are well understood, constructing controllers remains a serious software engineering challenge. Many different high-level architectures have been proposed but no one methodology addresses all problems. In short, this is an ideal area for the application of DSL technology.

Frob is an embedded DSL for robotic control systems. Frob is built on top of Functional Reactive Programming (FRP), which is in turn built from Haskell, a lazy, purely functional programming language[10]. Frob hides the details of low-level robot operations and promotes a style of programming largely independent of the underlying hardware. It also promotes a declarative style of specification: one that is cluttered by the details of how to elaborate the behavior.

As an embedded DSL, Frob includes the capabilities of a fully-featured functional programming language. Thus, Frob is best thought of a flexible framework for expressing robot programs rather than an embodiment of a specific system architecture.
This paper addresses both the Frob language itself, its capabilities, usage, and effectiveness, and the implementation of Frob. In particular, we examine the use of a monad to implement one of the essential semantic components of Frob. We demonstrate how "off the shelf" monadic constructs may be incorporated into a domain specific language to express its semantic foundation clearly and, more important, in a modular manner. We address monads from a practical vantage rather than a theoretical one; we emphasize their usage and benefits within our domain.

This paper contains many examples written in Haskell, but only a passing knowledge of Haskell, as well as an understanding of polymorphic type signatures, is needed to follow the general ideas. Although we make extended use of Functional Reactive Programming, we attempt to explain FRP constructs as they are used. Finally, no prior understanding of monads is required.

The remainder of this paper is organized as follows. Section 2 discusses the domain of robot control and essentials of both FRP and monads. Section 3 demonstrates the construction of the task monad in an incremental manner, adding features one by one and examining the impact on the system as the definition of a task changes. In Section 4, a number of non-trivial examples of Frob programming are presented. Section 5 concludes.

2 Background

2.1 The Problem Domain

At present Frob is implemented on a small mobile robots, 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 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 data rate from the sensors (except the camera) is quite low and the performance of non vision-based controllers is generally not an issue.

A simulator allows Frob programs to be tested independently of the robots, although simulating video input to the robots is difficult. The simulator has additional capabilities such as animation that can be exploited by a Frob program.

2.2 Functional Reactive Programming

In developing Frob we have relied on our experience working with Fran, a DSL embedded in Haskell for functional reactive animation [5, 4]. Specifically, we have borrowed the core behavior and reactivity components of Fran, FRP, to serve as a foundation for Frob.

FRP defines two essential representations used in Frob: behaviors and events. Behaviors are quantities that vary over continuous time. For a type t, the type \( \text{Behavior } t \) is an evolving value of type t. Behaviors are used to model continuous values; a value of type \( \text{Behavior SonarReading} \) represents the values taken from the sonars; \( \text{Behavior Point2} \) represents the position of the robot. Expressions in the behavioral domain are not significantly different from static (not time-varying) expressions. Aside from some operator renaming, users see little difference between programming with static values and with behaviors. For example, the following declaration is typical of Frob:
2.2 Functional Reactive Programming

```haskell
rererror :: Robot -> Behavior Float
rererror r = limit (velocity r * sin thetamax) (setpoint - leftSonar r)
  where limit m v = (-m) 'max' v 'min' m
```

This example shows a function mapping robot sensors (as selected by the `velocity` and `leftSonar` functions) onto a time-varying float, part of a larger control system. The details of this example are unimportant; the point is that writing functions over behaviors is little different from writing ordinary functions. Users of Frob have little trouble working with behaviors.

Behaviors hide the underlying details of clocking and sampling, presenting an illusion of continuous time. Behaviors also support operators not found in the static world: both integral and derivative, for example, exploit time flow. As a further example of the expressive power of behaviors, consider the following:

```haskell
atMax :: Ord a => Behavior a -> Behavior b -> Behavior b
```

This returns the value of the second behavior at the time the first behavior is at its maximum.

The other essential abstraction supplied with FRP is the event. The type `Event t` denotes a process that generates discreet values of type `t` at specific instances. Some components of the system are best represented by events rather than behaviors. For example, the bumpers are of type `Event BumperEvent`; occurrences happen when one of the robot bumper switches is activated. The console keyboard has type `Event Char`, where each keypress generates an event occurrence.

Events may be synthesized from boolean behaviors, using the `predicate` function:

```haskell
predicate :: Behavior Bool -> Event ()
```

Again, this function is easily understood by users familiar with the underlying domain. This demonstrates a typical event in a Frob program:

```haskell
stopit :: Robot -> Event ()
stopit r = predicate (time > timeMax || frontSonarB r < 20)
```

This event occurs when either the current time passes some maximum value or when an object appears less than 20 cm away on the front sonar of the robot.

Within FRP, a robot controller is simply a function from the robot sensors, generally wrapped up as components of the `Robot` type using behaviors and events, onto its effectors where behaviors drive the wheels and any other systems controlled by the robot. The flow of time is hidden with the FRP abstractions and the user sees only a purely functional mapping from inputs to outputs.

Is this all we need? Perhaps; complex robot controllers can be constructed using only the basic FRP primitives. However, such controllers have a number of problems:

- While FRP is well suited for the low-level control systems present in this domain, it lacks higher level constructs needed to plainly express robot behaviors at a high level.
Controllers may be complicated by "plumbing" code needed to propagate values in a functional manner.

Hard to understand FRP constructs are sometimes required. While users easily comprehend basic event and behavioral operators, FRP is also littered with arcane (but essential) operators such as snapshot, switcher, or withElem that are unfamiliar to users and are not a natural part to the underlying domain.

Our goal is to create better abstractions: ones that embody patterns that are familiar to domain engineers and that have well-defined semantic properties.

2.3 Monads as a Modular Abstraction Tool

In Haskell, a monad is any type that is an instance of the class Monad:

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

The operators are simple enough: >>= is sequential composition while return defines an "empty" computation. A special syntax, do notation, that makes calls to >>= more readable, as will be seen in later examples. As seen by the the programmer, a monad consists of a set of predefined building blocks (functions with the monadic type) and a sequencing operation (do) that glues these blocks together. Modularity results from the fact that new building blocks can be added without changing either the interface of existing building blocks or the way in which the blocks are sequenced. That is, new capabilities generally increase the monad's vocabulary without altering "sentences" expressed in the old vocabulary. Furthermore, many useful semantic structures have been captured in monads, yielding a rich library of semantic techniques for us to draw on.

Perhaps the best introduction to the practical use of monads is Wadler's "The Essence of Functional Programming" [12].

3 Implementing Frob

The basic implementation of Frob is discussed in [9] and [8]. Here, we examine only tasks and their use of monads.

3.1 The Basic Task Monad

Rather than present the full definition of Frob tasks up front, we will instead develop the task abstraction incrementally, adding features one by one and showing how each incremental extension in the expressiveness of tasks affects programs and the task implementation. Our purpose here is twofold: to show the ability of functional reactive programming to define the abstractions needed
for our domain and, more importantly, to show how the use of a monad to organize the task structure promotes modularity.

The essential idea behind a task is quite simple: the type `Task a b` defines a behavior (actually, any reactive value), `a`, over some duration and then exits with a value of type `b`. In terms of FRP, a task is represented as

```
behavior `untilB` event ==> nextTask
```

where `untilB` switches the behavior upon occurrence of the event. The `==>` operator passes the value generated by the event to the next task. Tasks are a natural abstraction in this domain: they couple a continuous control system (a behavior) with an event moves the system to a new mode of operation.

Initially, the task monad requires only one instrument from the monadic toolbox: a continuation to carry the computation to the next task. This is implemented by a type, `Task`, and an instance declaration for the standard Haskell `Monad` class:

```
data Task a b = Task ((b -> a) -> a)
unTask (Task t) = t
instance Monad Task where
  (Task f) >>= g = Task (\c -> f (\r -> unTask (g r) c))
  return k = Task (\c -> c k)
```

This defines a structure for combining computations (the glue); we still need to define the computations themselves. Here is a simple task creation function:

```
mkTask :: (Behavior a, Event b) -> Task a b
mkTask (b, e) = Task (\c -> b `untilB` e ==> c)
```

Now that we can create tasks and sequence tasks, how do we get out of the `Task` world? After all, the robot controller is defined in terms of behaviors, not tasks. That is, we need to convert a task into a behavior. This brings up a small problem: what to do when the task completes? That is, what is the initial value of the continuation argument? One way out of this dilemma is to pass in an additional behavior to take control after the task exits:

```
runTask :: Task a b -> a -> a
runTask (Task t) finalB = t (const b)
```

We now have everything needed to write a simple robot controller. Here are two simple tasks:

```
goAhead, turnRight, runAround :: Robot -> Task WheelControlB ()
goAhead r = mkTask (pairB 10 0) (predicate (frontSonar r < 20))
turnRight r = mkTask (pairB 0 0.5) (predicate (frontSonar r > 30))
runAround r = do goAhead r
              turnRight r
              runAround r -- loop forever
main = runController (\r -> runTask (runAround r) undefined)
```
The wheel controls are defined by a pair of numbers, constructed using `pairB`, with the first being the forward velocity and the second the turn rate. The `runController` function executes a controller (a function from sensors `Robot` to effectors `WheelControlB`). Note that `runAround` is not a terminating task so there is no reason to pass a default value to `runTask`.

Starting with this foundation (the monad of continuations, `mkTask` to build atomic tasks, and `runTask` to pull a behavior out of the task monad), we will now add some new features.

In the previous example, the robot description had to be passed explicitly into each part of the controller. We can pass this description implicitly rather than explicitly by building it into the task monad directly. That is, we want to pass the robot description to `runTask` and then have it appear wherever needed without adding extra parameters to everything. In particular, the place we really need it to appear is `mkTask`, since the behavior and event are generally functions of the current robot.

We define a new type to encapsulate `task state`:

```
data TaskState = TaskState {taskRobot :: Robot}
```

For now, our state has only one element: the current robot. The type `TaskState` is defined using Haskell record syntax, which then defines `taskRobot` as a selector function to extract the robot from the task state. As more components are added to the task state, the definition of `TaskState` will change but code referring to state values will remain unchanged. We now add this state to the definition of `Task`. A task is given an initial state (the first parameter) and then passes a (potentially updated) state to the continuation carrying the next task:

```
data Task a b = Task (TaskState -> (TaskState -> b -> a) -> a)
instance Monad Task where
  (Task f) >>= g = Task (\ts c -> f ts (\ts' r -> unTask (g r) ts' c))
  return k = Task (\ts c -> c ts k)
```

Again, this instance definition is “off the shelf”: a standard combination of continuations and state. The `runTask` function now needs an initial state to pass into the first task. An appropriate definition of `runTask` is now:

```
runTask :: TaskState -> Task a b -> a -> a
runTask ts (Task t) finalB = t ts (_ _ -> b)
```

In general, the call to `runTask` will need to fill in initial values for all components of the task state.

We’ve put `TaskState` into the monad, but how can we get it back out again? That is, how can tasks access information inside `TaskState`? These monadic operators directly manipulate the current `TaskState`:

```
getTaskState :: Task a TaskState
getTaskState = Task (\ts c -> c ts ts)
setTaskState :: TaskState -> Task a TaskState
setTaskState ts = Task (\c c -> c ts (c))
```
3.1 The Basic Task Monad

We also make the state available to tasks defined via \texttt{mkTask}. The argument to \texttt{mkTask} is now a function from the current task state onto the behavior and event defining the task:

\texttt{mkTask : (TaskState \rightarrow (Behavior a, Event b)) \rightarrow Task a b}

\texttt{mkTask f \equiv Task (\lambda ts c \rightarrow}
\hspace*{1cm} \texttt{let (b,e) = f ts in}
\hspace*{1cm} \texttt{b `untilB` e ==> c)}

The definitions in the previous example are now simplified: the robot is propagated to the tasks implicitly rather than explicitly:

\texttt{goAhead, turnRight, runAround :: Task WheelControlB ()}
\texttt{goAhead \equiv mkTask (\lambda ts \rightarrow let r = taskRobot ts in}
\hspace*{1cm} \texttt{(pairB 10 0, predicate (frontSonar r < 20))}
\texttt{turnRight \equiv mkTask (\lambda ts \rightarrow let r = taskRobot ts in}
\hspace*{1cm} \texttt{(pairB 0 0.5, predicate (frontSonar r > 30))}
\texttt{runAround \equiv do goAhead}
\hspace*{1cm} \texttt{turnRight}
\hspace*{1cm} \texttt{runAround -- loop forever}
\texttt{main \equiv runController (\lambda r \rightarrow runTask (TaskState \{ taskRobot = r \}) (runAround r) undefined)}

Note that the composite task, \texttt{runAround}, is not aware of the propagation of the task state. We could have retained the old \texttt{mkTask} (without the \texttt{TaskState} argument) for compatibility but have chosen not to. This change could, though, have been made without invalidating any user code.

We have, so far, exploited well known monadic structures for continuations and state. One more basic monadic construction is of use: exceptions. With exceptions, tasks of type \texttt{Task a b} may succeed, returning a value of type \texttt{b}, or fail, raising an exception of type \texttt{RoboErr}. This is reflected in a new definition of the \texttt{Task} type:

\texttt{data Task a b = Task (RState \rightarrow (RState \rightarrow (Either RoboErr b) \rightarrow a) \rightarrow a)}

These primitives raise and catch exceptions:

\texttt{taskCatch :: Task a b \rightarrow (RoboErr \rightarrow Task a b) \rightarrow Task a b}
\texttt{taskError :: RoboErr \rightarrow Task a b -- Raise an error}

We omit the definitions of these primitives and the modified \texttt{Monad} instance; these are standard constructions along the lines of those in [12]. However, we will examining the changes needed to \texttt{mkTask}. That is, the \texttt{Monad} instance itself is essentially independent of the underlying domain; the task creator, however, is domain-specific and must be modified to account for the presence of exceptions.

Here is a new version of \texttt{mkTask} that adds an error event to the basic definition of a task. We use a slightly different name, \texttt{mkTaskE}, so that the old interface, \texttt{mkTask}, remains valid. The new definitions are:
The only change here is that the terminating event is either the normal exit event with the Right
constructor added or an error event, as tagged by Left. The .\l. operator is FRP construct that
merges events, taking the first one. The ==> operator modifies the result of an event, so if err has
type Event RoboErr, then err ==> Right has type Event (Either a RoboErr), where Left and
Right are constructors for the Either type.

Next, consider a task such as “turn 90 degrees right”. Can we encode this easily as a Frob task?
Not yet! The problem is that a task don’t know the orientation of the robot at the start of the task.
We can build a control system to turn to a specified heading, but how do we know what the goal
should be? The answer lies in the task state. During task transitions (the untilB in mkTaskE), we
should also take note of where the robot is, which way its pointing, and other useful information.

In this simplified example, we capture the current robot location when moving from task to
task. The monad remains unchanged (except for a new field in the TaskState structure) but the
task builder must be modified as follows:

type RobotStatus = Point2

snapRobot :: Event a -> Robot -> Event (a,Radians)
snapRobot e r = e \"snapshot\" (orientationB r)

mkTaskE f = Task (\ts c ->
  let (b,e,err) = f ts in
  b \"untilB\"
    ((e ==> Right \|. err ==> Left) \"snapRobot\" (tsRobot ts)
      ==> (\(res,rstate) ->
        c (addRstate ts rstate) res)

The terminating event of the behavior is augmented with the state of the robot at the time of the
event by the the snapshot function, part of FRP. It samples a behavior during an event occurrence
and adds the behavior value to the event value. Tasks will now find this initial orientation as part
of the task state. This, a “turn right” task would be as follows:

turnRight = mkTask (\ts -> let goal = initialOrientation ts + 90 in ...)

Empty tasks (those generated by return) may pass the state onto the next task unchanged.
Having described the elementary task operations in detail, we now examine, briefly, other task operations. Given an existing task, what useful task transformations can be implemented? Examples include:

- `addError`:
  \[
  \text{Event RoboErr} \rightarrow \text{Task a b} \rightarrow \text{Task a b}
  \]

- `timeLimit`:
  \[
  \text{Time} \rightarrow \text{Task a b} \rightarrow \text{Task a (Maybe b)}
  \]

- `withB`:
  \[
  \text{(TaskState} \rightarrow \text{Behavior a)} \rightarrow \text{Task b c} \rightarrow \text{Task b (c,a)}
  \]

- `withExit`:
  \[
  \text{Event a} \rightarrow \text{Task b c} \rightarrow \text{Task b a}
  \]

- `withMyResult`:
  \[
  \text{(a} \rightarrow \text{Task a b)} \rightarrow \text{Task a b}
  \]

- `withFilter`:
  \[
  \text{(a} \rightarrow \text{a)} \rightarrow \text{Task a b} \rightarrow \text{Task a b}
  \]

- `withPicture`:
  \[
  \text{Behavior Picture} \rightarrow \text{Task a b} \rightarrow \text{Task a b}
  \]

While the full implementations of these functions is beyond the scope of this paper, we will provide a basic outline of each of these functions and how they are supported by the Task monad.

Most of the interesting semantic extensions to the system involve the basic definition of an atomic task. By bringing values from the task state into this definition, we can parameterize sequences of tasks rather than atomic ones. For example, consider `addError`: this function adds a new error event to an existing task. Note that the error event specified in `mkTaskE` applies only to an atomic task. The task passed to `addError`, however, may consist of many sequenced atomic tasks. The definition of `addError` looks something like this:

\[
\text{addError err tsk} =
\begin{align*}
& \text{do oldErr <- previous global error event} \\
& \quad \text{let newErr = err |.| oldErr} \\
& \quad \text{place newErr into the task state} \\
& \quad \text{execute tsk} \\
& \quad \text{restore prior global error event}
\end{align*}
\]

Of course, `mkTask` must be changed too. The event `err` now becomes `err |.| getGlobalErr`. In the `untilB` case, the error event in the task state must be included in the error condition. This sort of "scoped reactivity" is not easily expressed in basic FRP; using the task monad makes it much easier to implement this feature.

The `timeLimit` function aborts a task if it does not complete within a specified time. It is implemented using `addError` to attach an event that to the associated task which occurs at the specified time. This requires both the exception and state capabilities of the underlying monad.

The `withB` function defines a behavior to run in parallel with a task. When the task exits, the value of the behavior is added to the task’s result value. This is implemented by building a task that attaches a snapshotting function to the incoming continuation.

The `withExit` function aborts a task upon an event. If the task completes before the aborting event, an error occurs. This is implemented directly in FRP by `untilB`, as in this simplified definition:
withExit e (Task t) =
Task (\ts c -> t ts (error "Premature task exit") 'untilB'
     e => c)

Note that this is implemented directly at the continuation level.

The withMyResult function it allows a task to observe its own result, which is often needed in
the differential equations that define a controller.

withMyResult f =
Task (\ts c -> let r = (unTask t) ts c
         t = f r in
         e)

Another place the in which the atomic definition of a task may be further parameterized is the
resulting behavior. That is, instead of

b 'untilB' (e ...)

we modify the resulting behavior using a filter in the task state:

((getfilter ts) b) 'untilB' (e ...)

This is implemented in a manner similar to withError, using

withFilter :: (a -> a) -> Task a b -> Task a b

Finally, we discuss a more global change to the task structure. Debugging controllers is difficult:
it is hard to visualize the operation of a control system based on printing out numbers on the screen
as the controller executes. Instead, we wish to display diagnostic information graphically in the robot
simulator, painting various cues onto the simulated world to graphically convey information. For this,
we augment the behavior defined by a task to include an animation. That is, using the task monad
we provide an implicit channel to convey diagnostic information along with the behavior. This
modification requires changes to the definition of Task: a is replaced by (a, Behavior Picture),
the type now produced by runTask. This change does not affect user-level code; the extra picture
is implicit in every task. When a program is running on a real robot, the animation coming out of
runTask is ignored.

This is the definition of withPicture:

withPicture :: Behavior Picture -> Task a b -> Task a b
withPicture p t = addFilter (addPicture p) t

The addPicture function introduces an additional picture to an augmented behavior. For example,
this function makes driveToGoal easier to understand in simulation: expressed as

driveToWithPicture goal = driveTo goal 'withPicture'
        (move goal (withColor red circle))
3.3 Parallel Tasks

So far, we have only modified tasks or combined them sequentially. But we also wish to combine tasks in parallel. That is, execute more than one task at a time, combining their results. The \texttt{withTask} function is the basic primitive for combining tasks in parallel:

\[
\text{withTask} :: (t_2b \to \text{Task } t_1b \ t_1e) \to \\
(t_1b \to \text{Event} (\text{Either RoboErr } t_2e) \to \text{Task } t_2b \ t_2e) \\
\to \text{Task } t_2bt_2e
\]

This initiates two tasks, each observing the behavior defined by the other. The termination of the first task, either through an exception or normal termination, may be observed by the other task as an event.

The code associated with \texttt{withTask} is as follows:

\[
\text{withTask } t_1f \ t_2f = \\
\text{Task } (ts \ c \to \text{let } t_1e = \text{makeNewEvent} \\
\ t_1b = \text{unTask } t_1 (\text{cloneState } ts) \ (\text{sendTo } t_1e) \\
\ t_2b = \text{unTask } t_2 ts c \\
\ t_1 = t_1f \ t_2b \\
\ t_2 = t_2f \ t_1b \ t_1e \text{ in} \\
\ t_2b)
\]

Note the somewhat imperative treatment of the terminating event of \texttt{t1}. The \texttt{sendTo} and \texttt{makeNewEvent} functions exploit FRP internals. This is an expedient but semantically unusual way of dealing with events. The \texttt{sendTo} function becomes an undefined behavior after sending the termination message; reference to the value of \texttt{t1} after this event will result in a runtime error.

More importantly, notice the treatment of the task state. The task \texttt{t1} needs to receive a “fresh copy” of the overall task state. Local error handlers and filters are removed from the state so that only \texttt{t2} inherits these.

4 Examples

4.1 The BUG Algorithm

We demonstrate the use of tasks to implement a well known control strategy. BUG is an algorithm to navigate around obstacles to a specified goal. When an obstacle is encountered, the robot circles the obstacle, looking for the point closest to the goal and the returns to this point to result travel. The following code skeleton implements BUG in terms of two primitive behaviors: driving to a goal, and following a wall. The \texttt{driveTo} task returns a boolean: true when the goal is reached, false when the robot is blocked. The \texttt{followWall} runs indefinitely, traveling in circles around an obstacle. If for some reason the wall disappears from the sonars, this task raises an exception.
-- Basic tasks and events (not shown)
followWall :: Task WheelControlB ()
driveTo :: Point2 -> Task WheelControlB Bool
atPlace :: Robot -> Point2 -> Event ()

bug :: Point -> Task WheelControlB ()
bug g = taskCatch (bug g)
    (do finished <- driveTo g
         if finished then return () else goAround g)

goAround :: Task WheelControlB ()
goAround g = do closestPoint <- circleOnceP g
                circleTo closestPoint
                bug g

circleOnceP :: Point -> Task WheelControlB Point2
circleOnceP g = do (_,p) <- withB closestP (circleOnce g)
                 return p
                 where closestP ts = let r = taskRobot ts in
                      (distance (place r) g) ‘atMax’ (place r)
circleOnce = do ts <- getTaskStatus
               let goal = initialPlace ts; r = taskRobot ts
               timeLimit followWall 5 -- get away from initial place
               followWall 'withExit' (atPlace goal)

4.2 A Process Architecture

As another example, consider Lyons’ approach of capturing robotic action plans as networks of concurrent processes [6, 7]. Frob tasks can easily mimic Lyons’ processes. His conditional composition operation is identical to >>= in the task monad with exceptions. Of more interest is parallel composition: his P ∣ Q executes processes P and Q in parallel, with ports connecting P and Q. This can be directly implemented with withTask. His disabling composition, P # Q is also contained in withTask, however withError is also needed to correctly disable the resulting task when one task aborts.

The lazy evaluation semantics of Frob permits the following operations (synchronous concurrent composition and asynchronous concurrent composition) to be implemented directly:

P <> Q = do { v<>P; Q; (P<>Q) }
P >< Q = do { v<>P; (Q ∣ (P><Q)) }
5 Conclusions

We have demonstrated both a successful DSL for robotic control and shown how a set of tools developed in the functional programming community enable the construction of complex DSLs with relatively little effort.

Assessing a DSL (or any programming language) is difficult at best. We feel the success of Frob is demonstrated in a number of ways:

- Users from outside of the FP community find that Frob is easy to use and well-suited to the task of robot control. The abstractions supplied by Frob are easily understood at an intuitive level. While there is a definite learning curve, especially with respect to the Haskell type system, users soon become accustomed to polymorphic typing and find Haskell types much more descriptive than those of object oriented systems.

- We have encoded a number of well-known algorithms and architectural styles in Frob and found the results to be elegant, concise, and modular.

- As an embedded DSL, Frob interoperates easily with other DSLs. Preliminary work on combining Frob with FVision (a very different DSL for vision processing) suggests that at least some DSLs may be combined to great advantage.

- Monads support relatively painless evolution of DSL semantics. DSLs are, rather naturally, somewhat of a moving target: as domain engineers and DSL implementors work together, improvements in semantic expressiveness are continually being developed. The monadic framework has allowed this semantic evolution to proceed without requiring constant rewriting of existing code.

Some issues are still unresolved: we have not yet ported the system to new types of robots or implemented systems in which system performance is critical. We also have yet to experiment with multi-robot systems. Also, data rates from the sensors are low enough that performing and performance comparisons between Frob and C++ controllers is not very meaningful. Once we start to work with vision-based controllers we expect to be able to study some of these issues more thoroughly.

We have used Frob to teach an undergraduate robotics course. Frob was quite successful in allowing assignments that traditionally required many pages of C++ code to be programmed in only a page or two of code. While there was an admittedly steep learning curve, students eventually became quite productive. The addition of graphic feedback in the simulator was especially useful to them.

Turning to the issue of DSL construction, this research shows that monads are an important tool for attaining program modularity. The definition of task monad evolved significantly over the term but the interfaces remained the same, allowing all student code to run unchanged as Frob evolved. Monads effectively hide potentially complex machinery and provide a framework whereby new functionality can be added to a system with minimal impact on existing code.

All Frob software, papers, and manuals are available at http://haskell.org/frob. Nomadics has agreed to license their simulator to Frob users at no cost, allowing our software to be used by those without real robots to control.
6 Acknowledgements

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

References


