Monadic Robotics

John Peterson
Yale University
peterson-john@cs.yale.edu, http://www.cs.yale.edu/homes/peterson-john.html
Greg Hager
The Johns Hopkins University
hager@cs.jhu.edu, http://www.cs.jhu.edu.dom/˜hager

Abstract

We have developed a domain specific language for the construction of robot controllers, Frob (Functional ROBotics). The semantic basis for Frob is Functional Reactive Programming, or simply FRP, a purely functional model of continuous time, interactive systems. FRP is built around two basic abstractions: behaviors, values defined continuously in time, and events, discrete occurances in time. On this foundation, we have constructed abstractions specific to the domain of robotics. Frob adds yet another abstraction: the *task*, a basic unit of work defined by a continuous behavior and a terminating event.

This paper examines two interrelated aspects of Frob. First, we study the design of systems based on FRP and how abstractions defined using FRP can capture essential domain-specific concepts for systems involving interaction over time. Second, we demonstrate an application of *monads*, used here to implement Frob tasks. By placing our task abstraction in a monadic framework, we are able to organize task semantics in a modular way, allowing new capabilities to be added without pervasive changes to the system.

We present several 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: it consists of feedback systems connecting the robot sensors and effectors. The more difficult task is to build complex behaviors by sequencing among various control disciplines, guided by overall plans and 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, making 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 on top of Haskell, a lazy, purely functional programming language[14]. 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 not cluttered by low level implementation details.

An advantage of Frob (as well as many DSLs) is that it is *architecture neutral*. That is, instead of defining a specific system architecture (organization or basic design pattern) it enables arbitrary architectures to defined in a high level, reusable manner. As an embedded DSL, Frob includes the capabilities of a fully-featured functional programming language, Haskell.

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 importantly, in a modular manner. We address monads from a practical vantage rather than a theoretical one, emphasizing their usage and benefits within our domain.

This paper contains many examples written in Haskell. Readers must have at least a passing familiarity with the syntax, primitives, and types of Haskell. Those unfamiliar with Haskell will find `www.haskell.org` has much helpful information. Although we make extended use of Functional Reactive Programming, we attempt to explain FRP constructs as they are used. 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 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

Programming robots operating in the real world provides a unique example of a programming system operating in conjunction with the physical world. As such, any system must coordinate multiple ongoing control processes, detect special events governing task execution, and supply data structures and language primitives appropriate to the domain.

Unsurprisingly, the development of robot programming systems has been an area of active research in robotics (see [2] for a recent collection of articles in this area). Many of these languages are realized by defining data structures and certain specialized library routines to existing languages (notably C [6, 5, 10, 16], Lisp [1], Pascal [11], and Basic [15]). In particular, most of these "languages" include special functions or commands that operate in the time domain. For example, VAL includes a command to move the robot to a given spatial location. This command operates asynchronously and has, thereby, the side-effect of "stitching together" multiple motions if they are supplied in rapid order. Likewise, other embedded languages such as the "Behavior Language" of Brooks [1] and the Reactive Control Framework of Khosla [16] provide rich programming environments for the coordination of time-domain processes. AML [17] is an exceptional case — it is a language designed from scratch. As such, it supplies similar capabilities to VAL, but with the addition of enhanced error handling capabilities for robot program execution.

Despite the proliferation of robot programming systems, relatively little work has been done on deriving a principled basis for them. For example, none of the languages cited above has a formal (or in many cases even informal!) semantics. Counter to this trend, Lyons [9] provides a compositional paradigm for expressing robot plans (collections of atomic actions) that are sequenced and coordinated using a rich set of primitives. The notion of continuous behavior and discrete event-based transitions is also introduced. However, no transparent implementation of the language (or even realistic examples) is presented.

In contrast, Frob has a transparent, extensible, and semantically clear basis, and is a practical, useful tool for implementing robot programming systems. While Frob is also a library embedded in an existing language, the abstractions defining Frob are at a higher level than those used to embed controllers in languages such as C. While a controller embedded in C or C++ could, in theory, handle the same sort of abstractions we have used in Frob, their implementation would be much more difficult and the reliability of the system would suffer in the absence of a good polymorphic type system.

2.2 About Haskell

Since Frob inherits the syntax, type system, and libraries of Haskell users of Frob must, of necessity, learn about Haskell. This section contains a brief overview of the Haskell features used in Frob; those familiar with Haskell may wish to skip to the next section.

Basic Haskell features include the following:

- Variables: these start with lower case letters, such as `x` or `robotPosition`. Variables may include `_` and `'` characters, so `x'` is a also a variable name. Operators are composed of punctuation, such as `-=>` or `.|..`.

- Function application: Haskell does not use `f(x,y)` style notation for function calling. Instead, the parenthesis and commas are omitted, as in `f x y`. Parentheses may be used for grouping, as in `f (g x) (h y)`, which would be written `f(g(x),h(y))` in other languages.

- Infix operators: examples include `x + y` or `e -=> f`. An infix operation is converted into an ordinary variable using parentheses, as in `(-=>)` or `(+)`. Ordinary functions, such as `f`, can be used in the infix style when surrounded by backquotes, as in ``x `f` y``, which is the same as `f x y`. Application takes precedence over infix operators, so `f x+y z` parses as `(f x)+(y z)`.

- Layout: indentation separates definitions: each definition in a list must start in the same column and the list is terminated by anything in a preceeding columm. For example, in

  ```
  let x = 1
      y = 2
    in x+2
  ```

  the indentation of the `y` must exactly match that of `x`.

- Definitions: the `=` in Haskell creates a definition. You can define a constant, as in `x = 3`, or a function, as in `f x y = x + y`. As with function application, no parenthesis are needed around the function parameters.

- Lambda abstractions: functions need not be named. The expression `\x y -> x + y` is an anonymous function. For example, you can pass a function as a parameter: `f (\y -> y + 2)`. There's no difference between `f x y = x + y` and `f = \x y -> x + y`.

- Type signatures: the polymorphic types in Haskell are quite descriptive. Types are inferred, allowing type signatures to be omitted, but for clarity we include signatures in all examples. Type signatures supply valuable documentation and make type errors easier to diagnose. The syntax of a signature declaration is `x :: Int`, where this defines the type of `x` to be `Int`. Type signatures are generally placed immediately preceding the associated definition.

- Function types: the type of a function from type `t1` to type `t2` is written `t1 -> t2`. A function with more than one argument will have more than one arrow in its type. Watch for parenthesis, though: the type `f :: (Int -> Int) -> Int` defines a function with one argument, a function from `Int` to `Int`, rather than two `Int` arguments; that would be `f :: Int -> Int -> Int`.

- Currying: you don't need to pass all of the arguments to a function at once. A call to `f :: Int -> Int -> Int` without the second argument, as in `f 3`, results in a function that takes the remaining argument.

- Polymorphic types: lower case identifiers in type expressions are type variables. These scope over a single type signature and denote type equality. Many types are parameterized; the parameters are passed using the same syntax that expressions use. For example, the signature

  ```
  (==>) :: Event a -> (a -> b) -> Event b
  ```

  defines an operator, `==>`, that takes an `Event` parameterized over some type `a` and a function from type `a` to another type `b`, yielding an `Event` parameterized over type `b`. Polymorphic types are an essential part of Frob; many built-in Frob operators are may be almost completely described by their type signature.

- Contexts: In Haskell, overloading is manifested in type signatures that contain a context: a set of constraints on type variables, prefixing an ordinary signature. For example, the type

  ```
  (>)   :: Ord a => a -> a -> Bool
  ```

indicates that two arguments to > are of the same type and that this type must be in class Ord. A Haskell type class is similar to a Java interface.

- Tuples and Lists: A tuple is a simple way of grouping two or more values. Tuples use parentheses and commas: (x,y) is an expression that combines x and y into a single tuple. The elements of a tuple may have different types. Lists are written using square brackets: [1,2,3] is a list of three integers. The : operator adds a new element to the front of a list; [1,2,3] = 1:2:3:[]. Many list functions are pre-defined in Haskell.
- The Maybe type: the type Maybe a denotes either a value of type a, Just x, or Nothing. Maybe is often used where C programmers would use a pointer that may be void.
- The void type: the type (), pronounced "void", is used in situations where no value is needed. The only value of this type is ().

2.3 Functional Reactive Programming

In developing Frob we have relied on our experience working with *Fran*, a DSL embedded in Haskell for *f*unctional *r*eactive *an*imation [4, 3]. Functional Reactive Programming can be thought of as Fran without animation: the basic events, behaviors, and reactivity without operations specific to graphics.

At the core of FRP is the notion of *dynamically evolving* values. A variable in FRP may denote an ongoing process in time rather than a static value. There are two sorts of evolving values: continuous behaviors and discrete events. A behavior is defined for all time values while events have a value only at some discrete set of times.

For a type t, the type Behavior t is an evolving quantity of type t. Behaviors are used to model continuous values: a value of type Behavior SonarReading represents the values taken from the sonars; Behavior Point2 represents the position of the robot. Expressions in the behavioral domain are not significantly different from static expressions. Through overloading, most static operators operate also in the dynamic world: users see little difference between programming with static values and with behaviors. For example, the following declaration is typical of Frob:

```
rerror :: Robot -> Behavior Float
rerror 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 functions for static value.

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:

```
atMin   :: 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 minimum.

The other essential abstraction supplied with FRP is the event. The type Event t denotes a process that generates discreet values (messages) 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; occurances happen when one of the robot bumper switches is activated. The console keyboard has type Event Char, where each keypress generates an event occurance.

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

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

This can be thought of monitoring a boolean behavior and sending a message when it becomes true. This definition uses predicate to generate an event when an underlying condition first holds:

```
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, represented using behaviors and events bundled into the `Robot` type, onto its effectors, behaviors and events that drive the wheels and any other systems controlled by the robot. The flow of time is hidden with the FRP abstractions; the user sees 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.

2.4 Monads as a Modular Abstraction Tool

Monads have been surrounded by a great deal of hype in the functional programming community. This has led those outside of this community to ask questions such as "What the heck are monads anyway?", "If monads are so useful why don't they have them in Java?", and "Do I have to understand category theory to write Haskell programs?". In this section we will attempt to demystify monads somewhat.

First, why don't C programmers or Java programmers use monads? The answer is really quite simple: monads don't actually do anything new. Monads are used for state, exceptions, backtracking, and many other things that programmers have long done without monads. What the monad does is allow many well-understood constructs to be explained conveniently in purely functional terms. Outside a purely functional language, it's usually easier (but maybe not better) to do what you want directly without involving monads.

In a pure language, though, monads are an ideal way to capture the essential semantics of a domain without compromising purity or modularity. Monads hide the "gears and wheels" of the domain from the user while also presenting a simple, intuitive interface to the user. The user (as opposed to the DSL designer) sees only sequencing and the return operator, together with 'magic' functions that reach inside to these gears and wheels in some way. Monadic programming is made more readable in Haskell using the `do` notation. Users of the DSL don't really have to know anything about monads at all; they simply "wire up" their program using `do` and the rest of the monadic internals are unseen.

The most important feature of the monadic approach is modularity: new features may be added without breaking existing code. Under the hood, though, interactions between different features in a monad are out in the open. This is the advantage of a purely functional style: the interplay among these various features is very explicit.

Another advantage of monads is that there are many "off the shelf" monadic constructions available. There is no need to re-invent a basic semantic building block such as exception handling; this is already well understood. A DSL designer may combine many such building blocks into a very domain-specific monad. There are also a number of algebraic properties that make monadic programs easier to understand and reason about.

Going back to a very concrete level, a monad in Haskell defined with an instance declaration that associates a type with the two monadic operations in the class `Monad`:.

```
class Monad m where
```

```
  (>>=)    :: m a -> (a -> m b) -> m b
  return   :: a -> m a
```

The operators are simple enough: `>>=` (or *bind*) is sequential composition while `return` defines an "empty" computation. A special syntax, `do` notation, makes calls to `>>=` more readable. In addition to this instance declaration, other functions may reach inside the monad, hooking into its internals. For example, consider the state monad. Here, the bind and return define the propagation of the state from computation to computation. To actually reach inside to the state, additional functions must be written to get at this internal state. The following example shows the declarations needed to define a monad over a container type `T` that, internally, maintains a state of type `S`:

```
data S = ...

-- A computation in T that returns type a
-- is a function that takes a state and returns
-- an updated state and an a.

data T a = T (\S -> (S, a))

instance Monad T where
  (T f1) >>= c2 =
    T (\state ->
        let (state', r) = f1 state
            T f2 = c2 r in
          f2 state')
  return k = T (\state -> (state, k))

getState :: T S
getState = T (\state -> (state, state))

setState :: S -> T ()
setState state = T (\_ -> (state, ()))

runT :: S -> T a -> (S, a)
runT state (T f) = f state
```

The `Monad` instance explicates the passing of the state from the first computation to the second. The `getState` and `setState` reach inside this particular monad to access the normally hidden state. Finally, the `runT` function runs a computation in monad `T`, passing in an initial state and producing the final state and returned value.

We may add new capabilities to the state monad (exception handling, for example) without changing any of the user level code that uses the monad. That is, we may often enrich a monad's vocabulary without altering "sentences" expressed in the old vocabulary.

Perhaps the best introduction to the practical use of monads is Wadler's "The Essence of Functional Programming"[18].

3 Implementing Frob

The basic implementation of Frob is discussed in [13] and [12]. 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 occurance of the terminating 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 that moves the system to a new mode of operation. Tasks are not restricted to the top level of the system: any reactive value (event or behavior) may be defined as a task; many tasks may be active at one time.

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

 -- standard continuation monad
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 finalB)
```

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`). Since `runAround` is not a terminating task there is no reason to pass a final behavior 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 `r` 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 here 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 unaltered. 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. The definition of `runTask` is now:

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

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 ()
setTaskSTate ts = Task (\_ c -> c ts ())
```

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

```
mkTask :: (TaskState ->
           (Behavior a, Event b)) ->
          Task a b
mkTask f =
   Task (\ts c ->
           let (b,e) = f ts in
              b `untilB` e ==> c)
```

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

```
goAhead, turnRight, runAround ::
    Task WheelControlB ()

goAhead    =
  mkTask (\ts ->
           let r = taskRobot ts in
            (pairB 10 0,
             predicate (frontSonar r < 20)))

turnRight  =
  mkTask (\ts ->
            let r = taskRobot ts in
             (pairB 0 0.5,
              predicate (frontSonar r > 30)))

runAround  =
  do goAhead
     turnRight
     runAround   -- loop forever
```

```
main =
   runController
      (\r -> runTask
               (TaskState {taskRobot = r})
               (runAround r)
                undefined)
```

Note that the composite task, runAround, is not aware of the propagation of the task state. We could have retained the old mkTask (without the 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 Task a b may succeed, returning a value of type b, or fail, raising an exception of type RoboErr. This is reflected in a new definition of the Task type in which a task may return either its terminating event value, b, or an error value, of type RoboErr.

```
data Task a b =
  Task (RState ->
        (RState -> (Either RoboErr b) -> a) -> a)
```

These primitives raise and catch exceptions:

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

We omit the definitions of these primitives and the modified Monad instance; these are standard constructions along the lines of those in [18]. However, we will examine the changes needed to mkTask. That is, the Monad instance itself is essentially independent of the underlying domain, defined in terms of standard monad constructions and unspecific to robotics while the task creator, however, is very domain-specific and must be modified to account for the presence of exceptions.

Here is a new version of mkTask that adds an error event to the basic definition of a task. We use a slightly different name, mkTaskE, so that the old interface, mkTask, remains valid. The new definitions are:

```
mkTask ::
   (TaskState -> (Behavior a, Event b)) ->
   Task a b
mkTask f =
   mkTaskE (\ts -> let (b,e) = f ts in
                    (b, e, neverE))

mkTaskE ::
  (Robot ->
   (Behavior a, Event b, Event RoboErr)) ->
  Task a b
mkTaskE f =
  Task (\ts c ->
          let (b,e,err) = f ts in
            b `untilB` ((e ==> Right) .|.
                        (err ==> Left))
            ==> c)
```

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 `.|.` operator is FRP construct that merges events, taking the first one to occur. The `==>` operator modifies the value of an event, so if `err` has type `Event RoboErr`, then `err` `==>` Right has type `Event (Either a RoboErr)`. The constructors `Left` and `Right` define 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, a primitive defined FRP to capture the value of a behavior at the time of an event occurance. 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 =
                   initialOrient ts + 90 in ...)
```

Empty tasks (those defined as a `return`) pass the state onto the next task unchanged.

3.2 Task Transformations

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     ::
    Event RoboErr -> Task a b -> Task a b
timeLimit    ::
    Time -> Task a b -> Task a (Maybe b)
withB        ::
    (TaskState -> Behavior a) ->
    Task b c ->
    Task b (c,a)
withExit     ::
    Event a -> Task b c -> Task b a
withMyResult ::
    (a -> Task a b) -> Task a b
withFilter   ::
    (a -> a) -> Task a b -> Task a b
withPicture  ::
    Behavior Picture -> Task a b -> Task a b
```

The operation of many of these functions is obvious from the type signature: an illustration of the value of polymorphic type signatures as documentation. 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 then 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 subtasks. The definition of `addError` looks something like this:

```
addError err tsk
  do oldErr <- previous global error event
     let newErr = err .|. oldErr
     place newErr into the task state
     execute tsk
     restore prior global error event
```

Of course, `mkTask` must be changed too. The event `err` now becomes `err .|. getGlobalErr ts`: the error event in the task state must be included in the error condition in the `untilB`. 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 err `untilB` e ==> c)
  where err = error "Premature task exit"
```

This function 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 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. A much better debugging technique is 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:

```
driveToWithPicture goal =
   driveTo goal `withPicture`
    paintAt goal (withColor red circle)
```

3.3 Parallel Tasks

So far, we have only modified tasks or combined them sequentially. Now, we wish to combine tasks by merging the results of multiple tasks running in parallel. The `withTask` function is the basic primitive for combining tasks in parallel:

```
withTask :: (t2b -> Task t1b t1e) ->
            (t1b ->
              Event (Either RoboErr t2e) ->
              Task t2b t2e) ->
            Task t2bt2e
```

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 `withTask` is as follows:

```
withTask t1f t2f =
 Task (\ts c ->
        let t1e = makeNewEvent
            t1b = (unTask t1)
                  (cloneState ts)
                  (sendTo t1e)
            t2b = (unTask t2) ts c
            t1  = t1f t2b
            t2  = t2f t1b t1e in
         t2b)
```

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

More importantly, notice the treatment of the task state. The task 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 t2 inherits these.

4 Examples

Assessing a DSL is often difficult; different DSL's are designed with different goals. Since our goal is to build a language that is declarative and descriptive, we choose to assess it with examples of code rather than performance figures. These examples are chosen to demonstrate expressiveness rather than computational speed.

4.1 The BUG Algorithm

First, we demonstrate the use of tasks to implement a well known control strategy. BUG [7] 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 resume travel. The following code skeleton implements BUG in terms of two primitive behaviors: driving straight to a goal, and following a wall. The `driveTo` task returns a boolean: true when the goal is reached, false when the robot is blocked. The `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        ::
    Point2 -> Task WheelControlB ()
bug g      =
     taskCatch (bug g)  -- restart on error
     (do finished <- driveTo g
         if finished
             then return ()
             else goAround g)

goAround   ::
    Task WheelControlB  ()
```

```
goAround g =
    do closestPoint <- circleOnceP g  -- circle
       circleTo closestPoint -- then back
       bug g -- restart

circleOnceP   ::
    Point2 -> 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) `atMin` (place r)

       circleOnce =
        do ts <- getTaskStatus
           let initp = initialPlace ts
               r = taskRobot ts
           -- get away from initial place
           timeLimit followWall 5
           followWall `withExit` (atPlace initp)
```

This definition is quite close to the informal definition of BUG. Some necessary details have been filled in: what to do if the wall disappears from the sensors (this is caught at the top level and restarts the system), how to circle (travel for 5 seconds to get away from the start point and then continue until the start point is re-attained).

4.2 A Process Architecture

As another example, consider Lyons' approach of capturing robotic action plans as networks of concurrent processes [8, 9]. 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 may be 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 inter-operates 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 or unaddressed: 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. Frob has not been used in any way that taxed system performance; the data rates from our sensors are so low that the controller has relatively little work to do.

One particularly large part of this domain that has yet to be addresses is real time. For example, we cannot express the priorities of various activities, allowing Frob to direct resources toward more critical systems when needed. No guarantees are made with respect to responsiveness or throughput. We expect that Frob is capable of addressing these issues but much more complex analysis and code generation may be required. However, for high level system control (as exemplified by the BUG algorithm) these issues are of less importance.

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. 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

[1] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Trans. on Robotics and Automation*, 2(1):24–30, March 1986.

[2] E. Coste-Maniére and B. Espiau, editors. *International Journal of Robotic Research, Special Issue on Integrated Architectures for Robot Control and Programming*, volume 17:4, 1998.

[3] 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}`.

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

[5] V. Hayward and J. Lloyd. *RCCL User's Guide*. McGill University, Montreál, Québec, Canada, 1984.

[6] K. Konolige. Colbert: A language for reactive control in sapphira. In G. Brewka, C. Habel, and B. Nebel, editors, *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] V.J. Lumelsky and A.A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Trans. on Automatic Control*, 31(11):1058–63, 1986.

[8] D. Lyons and M. Arbib. A formal model of computation for sensor-based robotics. *IEEE Trans. on Robotics and Automation*, 6(3):280–293, 1989.

[9] Damian M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(7):241–256, June 1993.

[10] J.L. Mundy. The image understanding environment program. *IEEE EXPERT*, 10(6):64–73, December 1995.

[11] Pattis, R et al. *Karel the Robot*. John Wiley & Sons, 1995.

[12] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, May 1999.

[13] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of PADL 99: Practical Aspects of Declarative Languages*, pages 91–105, Jan 1999.

[14] 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.

[15] B. Shimono. *VAL: A Versatile Robot Programming and Control Language*. IEEE Press, 1986.

[16] D.B. Stewart and P.K. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, 1996.

[17] R. H. Taylor, P.D. Summers, and J. M. Meyer. AML: A manufacturing language. *Int. J. of Robot Res.*, 1(3):3–18, 1982.

[18] P. Wadler. The essence of functional programming. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM SIGPLAN, January 1992.