Frob: Functional Reactive Programming Applied to Robotics

Gregory Hager
Johns Hopkins University
Baltimore, MD
hager@cs.jhu.edu

John Peterson
Yale University
New Haven, CT
peterson-john@cs.yale.edu

Henrik Nilsson
Yale University
New Haven, CT
henrik.nilsson@yale.edu

ABSTRACT

We have applied methodologies developed for *domain-specific embedded languages* to create a high-level robot control language called *Frob*, for *F*unctional *Rob*otics. The goal of Frob is to provide an environment where robot software can be clearly, cleanly, and correctly specified, while suppressing unnecessary implementation detail. To this end, Frob incorporates three important notions: an implicit representation of time, equational specification of continuous-time behaviors and asynchronous events, and functions to construct complex, reusable abstractions. We show that these ideas can be combined to produce many commonly used "higher-level" abstractions for specifying robot programs. Experience with this approach to robot programming suggests that it leads to effective development of robot software while producing programs that are readable and can be reasoned about in a formal sense.

Keywords

Domain-specific Languages, Robotics Languages, Functional Reactive Programming

1. INTRODUCTION

Advances in computing and related mechatronics technology have led to a situation where extremely capable robotic systems are now cheap and widely available. As a result, developing flexible, robust robotic systems is rapidly being reduced to a problem of effective software engineering. In particular, the (as yet) experimental nature of building robotic systems places a strong emphasis on *rapid prototyping* and *software reusability.*

In this paper, we present *Frob* (for *F*unctional *Rob*otics), a unified framework for rapidly and reliably creating robotic software ranging from simple behavioral loops to large systems involving complex control strategies and/or multiple interacting modules.

Experience suggests that effective robot software construction is an incremental process of development, testing, and refinement. Thus, a good development environment should allow a user to pull standard tools (e.g. wall followers, control algorithms, and so forth) "off the shelf" and to customize those tools to a given application, both at the level of basic functionality, and at the level of systems architecture. It should be easy to test the resulting software system against a wide variety of operating conditions with minimal debugging of component algorithms.

We have adapted a software methodology developed in the field of domain-specific languages (DSLs) known as functional reactive programming (FRP). FRP is used to express two programming language features critical to robotics: time flow, as characterized by both continuous- and discrete-time signals, and reaction, allowing a system to reconfigure itself in response to stimulus. Frob is a thin layer built on top of FRP that defined abstractions specific to robotics. Indeed, the essential idea behind our approach is the ability to express general architectures using abstractions that hide the implementation of the architecture from the user, and make the structure of the program clear and easily understandable.

FRP is not a language in itself but rather a layer that is built on an existing host language. We have used two different host languages in our research: C++ and Haskell, a purely functional language. In this paper we will use the Haskell version of FRP to express programs, but it would also be possible to implement these programs in C++ directly.

In the remainder of this paper, we first introduce the basic concepts of Frob with particular emphasis on its ability to abstract over time-varying phenomena using FRP. We then describe how these basic structures can be used to develop the notion of a *task*, a more powerful and intuitive programming structure for robotic systems. We also show how tasks can in turn be used to develop other well-known software architectures, such as state machines.

2. AFRP

Functional Reactive Programming (FRP) is a general framework for programming hybrid systems in a high-level, declarative manner. The key notions provided by FRP are *signals* and *signal functions.* A *signal* is, conceptually, a function of time, used to represent time-varying values. A signal can be either a function of continuous time, i.e. the signal carries a value at every point in time, or a function of discrete time, meaning that it only carries a value at certain points which are referred to as *events.* AFRP (Arrowized FRP) is a new formulation of FRP based on arrows, a mathematical structure from category theory that provides the theoretical basis for the semantics of AFRP. AFRP solves a number of problems in the original FRP implementation and includes a special syntax that makes AFRP programs easier to write and understand.

A *signal function* is a *pure* function that maps a stimulating signal onto a responding signal. Signal functions are the basic building blocks of AFRP programs. They are first class entities: they have a type, they can be bound to variables, and they can be passed to and returned from functions. AFRP provides a rich library of built-in signal functions that can be used to describe complex signal flow patterns in a system. In addition, AFRP provides *dynamic switching* to change the system topology in reaction to various conditions.

2.1 Types and Notations

AFRP has been constructed as an embedded language using Haskell as a host language. The types and syntax of the following examples are all derived from Haskell. We will try to explain the notation as we go but familiarity with Haskell helps a great deal. Information on the Haskell language can be found at `www.haskell.org`.

AFRP is a *typeful* style of programming. We use type signatures to verify the correctness of a system and to describe its components to the user. These signatures make extensive use of *type variables* - variables that are part of the type language instead of the value language. The type of a signal function in AFRP is written `SF a b`: this should be understood as a function mapping a signal carrying values of type `a` to a signal carrying values of type `b`. The variables `a` and `b` indicate that a signal function may have any input or output type. Syntactically, type variables are lowercase letters while type constants are names that begin with an uppercase character such as `SF`. We use a `::` to associate a type with a value. For example,

```
wallDistance :: SF RobotInput Float
```

declared that `wallDistance` is a function that maps an input signal of type `RobotInput` onto an output signal of type `Float`.

Each signal function has only one input and output. If multiple inputs or outputs are needed we use *tuples* to bundle signals together. Tuples elements or types are enclosed in parenthesis and separated by commas. Thus `(2, True)` is a tuple containing two values, the integer 2 and the boolean `True`. Types containing tuples use a similar notation. This example defines a tuple constant:

```
p  ::  (Integer, Bool)
p  =   (1,False)
```

The following type declaration describes a signal function that has two inputs and three outputs:

```
monitorPath :: SF (RobotInput, Point2) ->
                  (Float, Float, Bool)
```

A signal function is defined using *arrow notation.* This notation is a syntactic sugar [2] that allows the use of variable names to represent signals. Within the arrows, *signal variables* get bound to the value of the related signal at the current point in time, i.e. a "sample".

The syntax of a signal function definition is as follows:

```
sf = proc patInput -> do
       pat1 <- st1 -< exp1
       pat2 <- st2 -< exp2
       ...
       patN <- stN -< expN
       returnA -< expOutput
```

The patterns, `patInput` and `pat1` through `patN`, are used to destructure signals that are grouped in tuples or other data structures. The only patterns we will need are a single variable or a tuple of variables. The expressions, `exp1` through `expN` and `expOutput`, are used to form signals using both the instantaneous signal values defined in the patterns and non-signal expressions. For instance, consider the following "wiring diagram"

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

This defines a component (signal function) that maps a pair of input signals onto a pair of output signals. The input is a tuple of two signals, named `a` and `b` in this component. The output is also a tuple, here defined using signals named `c` and `d`. The internal wiring also defines two local signals, `c1` and `c2`. Each of the sub-components is connected to the inputs or other sub-components. For example, the input to `sf3` is a tuple containing the outputs of `sf1` and `sf2`.

The type signatures of these signal functions must be compatible with each other. For example, the following types would be valid in this example:

```
sf  :: SF (Integer, Float) (Vector2, Bool)
sf1 :: SF Integer Float
sf2 :: SF Integer Vector2
sf3 :: SF (Integer, Vector2) Vector2
sf4 :: SF Float Bool
```

The values defined on the left of the arrows contain signal samples. These samples can be used to do pointwise computations. For example, consider an integrator:

```
integral :: SF (Float, Time) Float
integral = proc (v, t) -> do
  t' <- delay 0 -< t
  r  <- delay 0 -< r + (t - t') * v
  returnA -< r
```

There are a number of things going on here: a built-in signal function, `delay`, is being used to hold onto the previous value of the time and the integral. The expression that computes the next value of the integral, `r + (t - t') * v` uses operators such as `+` in a pointwise fashion, evaluating this expression repeatedly as time progresses. Finally, the definition of `r` is recursive. That is, this wiring diagram defines a feedback loop through the signal `r`. Note that the signal functions appear only inside arrows.

This notation describes complex interconnections of signal functions in a readable manner. Using a small set of primitive signal functions such as `delay`, we can build complex robot controllers entirely from a few basic signal functions.

2.2 Events

So far we have used only continuous signals. AFRP also defines events, signals which have values only at specific times. We use the type `Event` to represent these signals. Sampling an event valued signal at a time when the event does not occur yields `NoEvent`. At an occurrence, the value is `Event` v, where v is the value of the event at the time of occurrence. The Haskell definition of this type is as follows.

```
data Event a = NoEvent | Event a
```

These are the basic event functions:

```
mapE            :: (a -> b) -> Event a -> Event b
tag             :: Event a -> b -> Event b
mergeE          :: Event a -> Event a -> Event a
pairEvents      :: Event a -> Event b -> Event (a,b)
filterEvent     :: (a -> Bool) -> Event a -> Event a
```

Event values can be altered or replaced. The `mapE` function alters the value of an event using a function while `tag` replaces the value without regard to the old one. Event signals can be merged or filtered to produce new signals from existing ones. The `merge` function combines event streams, dropping the right hand event if both streams have occurrences at the same time (it is also possible to favor the right hand signal). The `pairE` function detects simultaneous occurrence of events and the `filterEvent` function erases event occurrences according to a filtering function.

2.3 Built-in Signal Functions

The following functions are some of the basic signal functions that define AFRP.

```
delay          :: a -> SF a a
edge           :: SF Bool (Event ())
switch         :: SF a b -> SF (a,Event (SF a b)) b
hold           :: a -> SF (Event a) a
```

The `delay` function delays the input stream one computational step. The argument to delay is the initial output value. In general, all feedback loops need to have a delay somewhere. The `edge` function generates an event when a boolean value changes. The `switch` function reconfigures the signal topology in response to an event. The system is initially the signal function given as the first argument to `switch`. When an event arrives, it brings a new topology which overwrites the current signal function. The `hold` function retains the most recent value in an event stream. This is similar to a latch in hardware design.

3. FROB

Frob is a language for robotic programming built on top of AFRP. Frob programs are signal functions that take an input signal that encapsulates the perception of the robot, including console events from a remote console as well as cameras, sonars, or odometry on the robot. The output signal sends commands to the robot and other devices such as a remote display. For example, a controller may send the camera image overlaid with graphics to a console window. The latter allows the operator to instruct the robot and to visualize how well the robot performs a specific task.

Every class of robot used by Frob has two specific data types that encapsulate the sensing environment and actuators of the robot. Frob uses *type classes*, generic interfaces to specific capabilities such as drive motors or sonars, to allow reuse across different hardware platforms. Rather than dwell on the details of type classes, we will use specific data types, `RobotInput` and `RobotOutput` in these examples.

To illustrate how the continuous-time aspects of Frob can be used, consider a simple feedback control algorithm[1] for wall following using range (e.g. sonar) data. The system to be controlled is a differential drive robot with control inputs v and ω representing the desired translational and rotational velocity of the robot, respectively. The algorithm makes use of the readings of the front and side range sensors, f and s, as well as the current velocity v_{curr} of the robot. In equations, we write

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

where $\sigma(x, y)$ is the limiting function

$$\sigma(x, y) \quad = \quad \max(-x, \min(x, y))$$

d^* is the desired (static) "setpoint" distance for objects to the front or side of the robot, and $v_{\max}$ and $\theta_{\max}$ are the maximum robot velocity and body angle to the wall, respectively. The strategy expressed here is fairly simple: the robot travels at its maximum velocity until blocked in front, at which time it slows down as it approaches the obstacle. It turns toward or away from the wall based on its distance relative to d^*, but at no time is the side range sensor allowed to be more that $\theta_{\max}$ degrees away from the perpendicular to the wall.

The above equations can be understood as describing a signal function that maps the current speed signal and two sonar signals to a pair of control signals giving the desired translational and rotational velocity of the robot. In Frob, this means a signal function from a signal of triples to a signal of pairs. (An n-tuple of signals is isomorphic to a signal of n-tuples for our purposes.) Furthermore, we would like to retain the *static* parameterization on the setpoint distance. Thus, in Frob, the result is a *static* function from the setpoint distance to a signal function from velocity and sonar signals to control signals:

```
wallFollow :: Dist -> SF (Vel,Dist,Dist) (Vel,RotVel)
```

The definition of `wallFollow` is mostly a transliteration of the equations above. The first equation says that v, at *every point* in time, is proportional to the difference between the distance to the nearest obstacle in front of the robot and the desired setpoint distance at the *same* point in time, limited by the maximal velocity. Such point-wise equations can be transliterated directly into the arrow notation:

```
let v = limit v_max (f - d_star)
```

The equation for ω can be handled in almost the same way, except for the time derivative of the sonar signal s. Differentiation is an operation on a *signal*, as opposed to individual points on a signal in isolation. The differentiation operator in Frob is thus a signal function with the following signature:

```
derivative :: SF Double Double
```

[1]Here and elsewhere we have, in the interest of brevity, suppressed algorithm details (*e.g.* gain coefficients) which are not essential to our presentation of Frob.

Using the arrow application notation, the equation $s_{\mathrm{dot}} = \dot{s}$ can be rendered in Frob as follows:

```
s_dot <- derivative -< s
```

This may not look like an equation, but it actually is. The difference is that this is an equation on *signals*, whereas the more familiar looking syntax is used for point-wise equations on signal *values*. Frob makes a careful distinction (at the type level) between signals and signal vales. This may seem to complicate the notation in a simple setting like this one, in particular in comparison with standard mathematical notation, but it turns out that this distinction is highly beneficial in practice. Furthermore, one could explore additional syntactical shortcuts which would make the notation even more like standard mathematical notation, but we have not yet done so.

Given these considerations, the control equations can be rendered into the following, complete, definition of `wallFollow`, including some auxiliary definitions:

```
wallFollow :: Dist -> SF (Vel,Dist,Dist) (Vel,RotVel)
wallFollow d_star = proc (v_curr, f, s) -> do
    let v = limit v_max (f - d_star)
    s_dot <- derivative -< s
    let omega = limit (v_curr * sin theta_max)
                      (d_star - s)
                - s_dot
    returnA -< (v, omega)

limit high x = max (-high) (min x high)
theta_max = 10
v_max     = 0.5
```

Even to someone who has never seen Frob, the correspondence between this program and the equations above should be very clear. In particular, note that time is *implicit* in the function, just as it was in the original set of equations.

The function `reactimateRobot` is used to connect a controller to the sensors and actuators of a robot. It has the following signature:

```
reactimateRobot :: SF RobotInput RobotOutput -> IO ()
```

Both `RobotInput` and `RobotOutput` are effectively records of signals. Thus, to run our controller, we need extract the relevant sensor information from the robot input signal and feed it to the controller, and we need to embed the generated control signal in the robot output.

For extraction, we will assume that the following signal functions are available:

```
velocity   :: SF RobotInput Velocity
frontSonar :: SF RobotInput Distance
sideSonar  :: SF RobotInput Distance
```

Note that these are signal functions and not ordinary static functions. Thus they can be used for various form of stateful pre-processing of the input signals. For example, the sonar inputs are naturally discrete events (echos). We will assume that `frontSonar` and `sideSonar` smooths the input in such a way that the result is a continuous-time, one-time differentiable signal.

The output is a little more involved since we potentially need to compose actuator output from more than one source. To that end, we provide functions which map desired actuator values to a *function* that will modify the robot output in the desired way. For example, the function `setVelTR` sets the desired values for rotational and translational velocity:

```
setVelTR :: Velocity -> RotVel
            -> (RobotOutput -> RobotOutput)
```

The output from more than one control source can now be combined by composing the resulting functions of type `RobotOutput->RobotOutput` using ordinary function composition. The result is a function of the same type which finally can be turned into `RobotOutput` by applying it to a suitably chosen base value. This is done by `roMake`:

```
roMake :: (RobotOutput -> RobotOutput) -> RobotOutput
```

Putting this all together, we can "package" the wall follower with the appropriate sonar for following a wall on the right by writing:

```
wallFollow1 :: Distance -> SF RobotInput RobotOutput
wallFollow d_star = proc i -> do
    v_curr     <- velocity   -< i
    f          <- frontSonar -< i
    s          <- sideSonar  -< i
    (v, omega) <- wallFollow -< (v_curr, f, s)
    returnA -< roMake (setVelTR v omega)
```

We can then execute right-wall following at a distance of 0.2 meters by writing:

```
main = reactimateRobot (wallFollow1 0.2)
```

Note that, since `wallFollow` is defined on behaviors, 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. Some operators, notably `derivative` in this example, directly exploit the time-varying nature of the signals. Finally, note that the code is independent of the kind of sensors used to measure the distances or, more generally, how it is derived. For example, we could easily compose `wallFollow` with a signal function that performs filtering to clean up the incoming sonar data.

4. TASKS

A task is an activity that terminates when some condition is met. For example, a task can capture the idea of driving the robot to a specific location: there is both a continuous behavior (driving) and a terminating event (arrival). More complex tasks can be achieved by combining simpler ones.

We have defined a `Task` construct that allows the user to build, modify, and sequence tasks[1]. Essentially, a task can be understood as a signal function that indicates when it is done by generating an event. Thus, the three types which determine the type of a task are the type of the input signal, the type of the output signal, and the type of the event. Consider the following signature:

```
driveTo :: Point2 -> Task RobotInput RobotOutput Bool
```

This states that `driveTo` is a function that takes a single argument, a 2-D point, and returns a task that defines a signal function from `RobotInput` to `RobotOutput`. When the task is complete it generates a value of type `Bool`. The idea here is that the task will attempt to drive to the indicated position, returning True if it succeeded, False otherwise.

Signal functions can be turned into tasks and vice versa by means of the following two combinators:

```
mkTask  :: SF a (b, Event c) -> Task a b c
runTask :: Task a b c -> SF a (Either b c)
```

The first one captures the underlying intuition that a task can be seen as a signal function with an event output indicating task termination. The second turns a task into a signal function where the output is the signal function's output until it terminates, and then the value of the terminating event; hence the type `Either b c`.

The arrow combinator `&&&` for parallel composition of signal functions is useful for creating signal functions of the right type:

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

For example, given an arbitrary signal function `sf::SF a b`, we can turn it into a non-terminating task as follows:

```
mkTask (sf &&& never)
```

where `never::SF a (Event b)` is a standard event source that never never has any occurrences.

The `do` syntax is used to sequence tasks:

```
  task1 = do t1
             t2

  task2 = do x <- t1
             t2
```

In `task1`, robot will first perform task `t1` and then `t2`. In the second kind of binding in `task2`, the first task `t1` returns a value related to the outcome of the task and this value can be used in the definitions of subsequent task(s).

In general, tasks can be used to describe a system in terms of mode transitions. Structures such as finite state machines are easily encoded using tasks. For example, consider the following example:

```
task1 = do r1 <- task2
           if r1 < 0.1 then task1 else
              do r2 <- task3
                 if r2 < 0.1 then task1 else task4
```

This task has three different states and two cycles. Note the use of the `if` construct for switching between tasks.

In addition to sequencing, the following operators create the basic vocabulary of tasks:

```
constT    :: b -> Task a b c
snapT     :: Task a b a
timeOut   :: Task a b c -> DTime ->
             Task a b (Maybe c)
abortWhen :: Task a b c -> SF a (Event d) ->
             Task a b (Either c d)
withSF    :: Task a b c -> SF a d -> Task a b (c,d)
```

`constT` creates a task with a constant output value. `snapT` is an immediately terminating task that returns the current value of the input signal. `timeOut` adds a time constraint to an existing task by terminating it after the specified time passes. In this case the task value is `Nothing` denoting termination because of time, otherwise it is `Just c`, `c` being the normal termination value. Similarly `abortWhen` can be used for adding another termination criteria to a task. The value of the new task will be of type `Either c d`, `Left c` denoting normal termination and `Right d` meaning terminated by the new criteria.

We can illustrate these concepts by extending the previous wall following algorithms to implement the BUG navigation algorithm. Informally, the idea of the BUG algorithm is as follows. When in free space, the robot drives toward a 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 using the wall follower defined in the previous section, dressed up as a non-terminating task, and some additional components:

```
wallFollowT d_star =
    mkTask (wallFollow1 d_star &&& never)
driveTo :: Point2 -> Task RobotInput RobotOutput Bool
atPlace :: Point2 -> SF RobotInput Event ()
atMin   :: SF (a, Double) a
```

The task `driveTo` drives straight toward a goal and returns a boolean event: true when the goal is reached, false when the robot is blocked. The event source `atPlace` occurs when the robot gets sufficiently close to a given place. The signal function `atMin` outputs the value of the first signal associated with the smallest value of the second signal encountered so far.

The top level structure of `bug` relies primarily on sequencing using the `do` syntax:

```
bug goal = do
    finished <- driveTo goal;
    if (not finished)
        then do
            goAround goal
            bug goal
        else
            return ()

goAround goal = do
    closestPoint <- findClosestPlace goal
    circleTo closestPoint
```

Note the tail-recursive call to `bug` within the `if`.

We now have to circle the object to find the closest point. Circling can be accomplished as follows:

```
circleTo p = wallFollowT 0.20 `abortWhen` atPlace p

circleOnce = do
    initp <- robotPlace
    wallFollowT 0.20 `timeOut` 5.0
    circleTo initp
```

Here, we have "customized" the wall following algorithm in two ways. In `circleTo`, we force termination of the (non-terminating) `wallFollowT` behavior when it arrives at place `p`. In `circleOnce`, the `wallFollowT` task is given a 5 second time limit, just enough time to move away from its initial position. Then, wall following is continued until it returns to its starting point.

One interesting facet of the algorithm is how we determine the closest point to the goal. The signal function `atMin` outlined above comes in handy here.

```
closestPlace goal = proc i -> do
    p <- atMin (place i, distance (place i) goal)
    returnA -< p

findClosestPlace goal =
    circleOnce `withSF` closestPlace goal
```

Here, we use `withSF` to get a "snapshot" of the `atMin` behavior when `circleOnce` terminates. Since it constantly records the robot position at the minimum distance to the goal, the terminating event of `findClosestPlace` will contain that value.

5. CONCLUSIONS

To obtain the latest version of Frob and AFRP, please visit `www.haskell.org/frob` for our most recent software. To date we have only used Frob on a Pioneer robot with stereo vision and on a Nomadics scout. Porting Frob to new hardware settings is relatively easy. We have also implemented FRP directly in C++, allowing this same style of programming to be used without incorporating Haskell code into the system.

Our experience with FRP has been quite promising. We have found it to be a flexible and powerful method for integrating the various time-based computations required in vision, robotics, graphics, and human-machine interaction. Furthermore, the recent development of AFRP has been a great practical step forward in terms of the scalability of our systems.

At the same time, we have learned several lessons in the use of FRP. First, we have found that most toolkits cannot be treated as black boxes with a single, high level interface. For example, our vision code is imported at a level which allows us to redefine some of the underlying C++ abstractions (e.g. a visual tracker) in FRP. This level of interface is far more flexible and useful when integration occurs. Second, we have found that there is often a natural evolution from high-level (FRP) code to low-level (e.g. C++) code as applications develop and tool-box deficiencies appear. As a result, there is a natural merging of functionality over the life of a project. Finally, we have found that working in FRP is ideal for the prototyping "beyond the state of the art." The flexible facilities of FRP (and its embedding in Haskell) make it an ideal platform to discover and develop domain abstractions.

6. REFERENCES

[1] G. Hager and J. Peterson. A transformational approach to the design of robot software. In *Robotics Research: The Ninth International Symposium*, pages 257–264. Springer Verlag, 2000.

[2] R. Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.