Types and Applications for Functional Reactive Programming

Jonathan Schuster

1 Introduction

In this talk, I'm going to give an introduction to functional reactive programming and some of its uses. We'll look at the history of the idea, some of its key features, and how it might be used in various settings. I like to think of it as seeing the full timeline of a single idea: we'll go from a problem in need of a solution, to a proposed solution, to refinements to that solution, and finally to applications to problems we face today (specifically, programming for networks).

2 Motivation

Let's take a few moments to think about our typical programming model. Often, when we're describing a language or talking about extending a language with some new feature, we use a model that looks a lot like what you see in the lambda calculus: we give a single input value to a program, the program runs for a while and internally works with the input, and eventually it returns a single output value. Internally, there might be complicated things going on inside the program, like mutation, non-local control, etc., but the user only sees the final output value that is eventually returned.

This model is easy to understand and reason about, but unfortunately, the real world is not so simple. In the real world, things are constantly in motion, everything is changing over time, and our program is expected to change with it. Instead of having just one input, our program can take any number of inputs over time, and it can have any number of outputs, as well. If an event suddenly occurs, our program might be expected to change its process entirely based on that change. We can apply this model to many different scenarios, such as software to control vehicles like cars and airplanes, GUI programs, and robots. In short, our programs need to *react* to the changes around them.

Even if we ignore changing input values, some programs might be expected to behave differently based on the passage of time itself—for example, think of media players or animations, which need to produce new output multiple times per second.

So, we can see that we have two types of programs: We call the first type of program *transformational*, since its job is to transform the given input into an

output, and we call the second type *reactive*, since the program should react to the changes in its environment.

The problem we want to solve is, how can we model this ever-changing world and write programs that can handle all of its changes? The typical solution seen today is to write the normal imperative code that runs in an infinite loop, continuously creates the output needed, and reacts to events either by creating a new thread for each event or periodically polling for any new events that have come in. Saving state in our program allows us to hold onto any old values that may have changed over time.

However, moving into the imperative world forces us to lose many of the nice reasoning models we had in our functional, transformational world. Furthermore, this model mixes together two separate parts of our program: the "framework" of our program that handles communicating with the real world, checking for events, etc., and the actual computation that we're trying to express. For example, if you think of a program that runs an animation, our code to run a loop and draw a frame every n milliseconds would be mixed up with the code that calculates, given the current time, where we want a certain point to be. We want a model that allows for a clean separation of these two parts of our programs and gives us back a model that's easy to reason about, hopefully similar to our familiar lambda calculus. This is the intent of functional reactive programming.

3 Functional Reactive Programming

Functional reactive programming (a.k.a. FRP) [1] is a programming model created to separate the implementation details of our reactive programs from the domain issues we wanted to express. The key idea of FRP is that everything is expressed in terms of *signals*. A signal is just an entity whose value changes over time. At a high level, you can think of a signal as a function from **Time** to some output value of type τ:

$$\textbf{Signal } \tau = \textbf{Time} \rightarrow \tau$$

However, a signal is not just a typical function as we know them from the lambda calculus. For example, we can examine signals for their edges (the points where the value changes), and create new signals based on these edges. There are other differences we'll see throughout the rest of this talk.

We can use signals to represent all sorts of time-varying values. In an animation, this might be the color or size of some image. In a mobile robot, signals could act as input such as light sensors or collision detection, or output like the motor speed for each wheel. In a video player, we might have one signal for the current frame being displayed, and other signals like volume and pitch for the audio output.

We can also model events, such as keyboard presses or mouse clicks, as signals. An event signal is just a signal that, at any given time, either has a value of that event's type, or an appropriate null value. In Haskell, we can

represent this using the Maybe type. For example, the signal for a keyboard press event would have the type **Signal (Maybe Char)**.

Effectively, when we write an FRP program, we're really just computing a new signal based on the input signals given. As a simple example, imagine we have a robot whose x-coordinate is defined by the following formula:

$$x(t) = \frac{1}{2} \int_0^t (v_r(t) + v_l(t)) \cos(\theta(t)) \, \mathrm{d}t$$

Using FRP, we could write a signal for this formula like so (assuming we already have signals `vr`, `vl`, and `theta`; and the various necessary math operators have been lifted to the level of functions):

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

Depending on what sort of program we're writing, some underlying framework can take this signal computation and use it as needed, while handling all of the low-level issues with dealing with the surrounding environment. The framework handles all of the implementation/operational details for us, such as sampling rates, polling, gathering input, etc., and allows us to work in a simpler model. This leads to all sorts of benefits, like better modularity, easier reasoning about programs, and programming at a higher level of abstraction.

3.1 Signal Functions and Arrows

What I've described so far is FRP as described in the original paper, *Functional Reactive Animation*. However, there's a problem: this level of expressiveness in our signal code can cause space and time leaks. Intuitively, the problem is that functions allow the programmer to access the values of a signal from any point in the past up to the current time. This makes it too easy to accidentally write functions that have to compute every previous value of the signal, since signals do not normally hold onto their state. For example, if a signal relies on just its single previous value, it might have to compute every value from its past just to reconstruct that value. This can cause space leaks in that we now need $O(n)$ amount of memory to store the previous values during computation (where n is the number of previous steps). It can also cause time leaks in that our operations might take linear time instead of constant time. This is especially problematic in real-time systems, a common application of FRP.

To solve this problem, we instead restrict our language so that programmers actually write in terms of *signal functions*, which you can think of as functions from signals to signals:

$$\mathbf{SF}\ \tau_1\ \tau_2 = \mathbf{Signal}\ \tau_1 \to \mathbf{Signal}\ \tau_2$$

This is the method used by Yampa [5, 3], the FRP implementation from the Yale Haskell group and the subject of both *Functional Reactive Programming, Continued*, and *Arrows, Robots, and Functional Reactive Programming*.

Although the type is described above as a function, the type **SF** is actually abstract—its representation is completely hidden from us, allowing the underlying framework to implement it in whatever way it sees fit. By doing this, we have completely removed the programmer's access to signals: in other words, signals are no longer first class! This prevents the programmer from introducing certain kinds of space and time leaks, because they can no longer access values of the signal from arbitrarily far back in the past.

FRP programmers still need to create and manipulate signal functions, though. To do so, we structure signal functions with *arrows*. In high-level terms, an arrow is a general abstraction of computation. If you're familiar with monads, arrows are actually a generalization of monads (although don't worry if you don't know monads—they're not necessary for this talk). The key benefits to arrows are that we can write arrows that deal with the current instantaneous values of our signals, even though we have no access to the signals themselves, and we can compose arrows in a number of ways. The application of arrows is abstracted away from us, though, so we then rely on the underlying FRP implementation to apply them whenever a new value of a given signal is needed.

To work with signals, we are given a number of *combinators* (functions that work on signal functions). The combinator **arr** lifts pure (stateless) functions to signal functions:

$$\mathbf{arr} : (\tau_1 \to \tau_2) \to \mathbf{SF}\ \tau_1\ \tau_2$$

The >>> combinator does the expected composition, piping the output of the first signal function into the input of the second:

$$>>> : \mathbf{SF}\ \tau_1\ \tau_2 \to \mathbf{SF}\ \tau_2\ \tau_3 \to \mathbf{SF}\ \tau_1\ \tau_3$$

Finally, **&&&** creates a signal function that splits a signal into a pair of signals, applying the first signal function to the first part of the pair, and the second function to the second part. You can think of this as parallel application:

$$\&\&\& : \mathbf{SF}\ \tau_1\ \tau_2 \to \mathbf{SF}\ \tau_1\ \tau_3 \to \mathbf{SF}\ \tau_1\ (\tau_2 \times \tau_3)$$

These three combinators define the core rules for arrows. There are also a number of other combinators that can be built from these and are included in the Yampa system.

Also, to allow for computing on previous values of a signal, and generally allowing for signals with "state", some primitive signal functions are provided to manage these computations for us. Among these are **integral**, which computes the integral of a signal from time 0 to the current time, and **hold**, which holds the value of an event as the current value until another event occurs.

Let's look at an example that puts all these combinators together. Recall our x-coordinate formula from earlier:

$$x(t) = \frac{1}{2}\int_0^t (v_r(t) + v_l(t))cos(\theta(t))\,\mathrm{d}t$$

Our signal function for this formula looks like the following (assuming supplied signal functions `vrSF`, `vlSF`, and `thetaSF`; and the binary equivalent of **arr**, **arr2**):

```
xSF = let v = (vrSF &&& vlSF) >>> arr2 (+)
          c = thetaSF >>> arr cos
      in (v &&& c) >>> arr (*) >>> integral >>> arr (/2)
```

If you look hard enough, you can see that this does indeed implement the given function. Howerver, it's hard to see that when using typical function application and infix operator syntax for our combinators and signal functions. Fortunately, Yampa has a separate syntax used for this purpose:

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

The above syntax defines a new signal function, `xSF`. The binding structure is similar to Scheme's `let*`, with each variable to the left of an arrow being bound in each successive line. The `proc` keyword indicates the beginning of this form, and `inp` is a variable that is bound in the scope of this form to current instantaneous value of the input signal to this signal function. On each arrow line, the term on the right is an expression indicating the instantaneous value to be sent to the signal function indicated in the middle of the arrow. Finally, the left-hand side can be a pattern to bind the result of the signal function's computation.

You can see that, while not as clean as the original signal definition, this syntax is much more readable than the version we defined previously. The above form will desugar to something like what we defined above, using the core combinators as needed.

3.2 Events and Models of Time

As mentioned above, we can implement events as **Maybe** types. However, events are only defined at instantaneous points in time. This brings up the issue of modeling time. In the Yampa system, time is modeled with both continuous and discrete methods, although the emphasis is more on continuous time. Behind the scenes, when our implementation is dealing with a continuous time signal, it's typically going to get values of this signal by polling periodically—since some value is always defined on that signal, polling will always return some reasonable value.

However, this is not true for discrete time signals, such as events. Since events only happen at infinitely small moments in time, there's a good chance polling would miss some events altogether. In these cases, the implementation would most likely rely on some sort of push/interrupt mechanism to feed it new values when an event occurs, which would kickstart the evaluation of signal functions that rely on that event.

There are various signal functions that move between continuous and discrete time. For example, **edge** outputs a signal function that generates an event whenever the value of its input signal changes. On the other hand, **hold** does the opposite: its output signal's value is always the value of the last event that occurred (or some default value).

Another important signal function is **switch**, which has the following type:

$$\textbf{switch} : \textbf{SF}\ \tau_1\ (\tau_2 \times \textbf{Event}\ \tau_3) \rightarrow (\tau_3 \rightarrow \textbf{SF}\ \tau_1\ \tau_2) \rightarrow \textbf{SF}\ \tau_1\ \tau_2$$

As you can see, this signal function takes as input a signal function that generates a value and an event, as well as a normal function for generating a new signal function based on the value of the event. The semantics of switch are that it acts like the signal function in the first argument until an event occurs, at which point it calls the function in the second argument to generate a new signal function to use from that point forward.

There are other varieties of **switch**. **dSwitch** delays the switch by exactly one time-step, so that the switch does not happen until just after the event occurred (which allows some loops to be well-founded). **rSwitch** is like the normal **switch**, except that it generates a new signal every time an event occurs, not just the first time. Finally, **drSwitch** combines the functionality of **dSwitch** and **rSwitch**.

Here's an example of a use of **switch**. We want to program a robot such that, whenever it hits a wall, we reverse its velocity (so that it moves away from the wall). We are given signal functions `currVel` and `hitWall`, which provide the current velocity and an event whenever we hit a wall. Here's what our signal function would look like:

```
fixVelocity = switch (currVel &&& hitWall)
              lambda (e). (currVel >>> arr (lambda (x). -1 * x))
```

This signal function will initially act exactly like `currVel`, since it's the first part of our pair in the input to `switch`. As soon as we get an event from `hitWall`, we will use the the function given as the second argument to produce the new signal function, which just passes the current velocity to a signal function that will negate it.

3.3 Loops

Finally, we can also have loops in our FRP programs. Specifically, we can use the output of a signal function as part of its own input. Since this would cause an infinite loop if a signal function's output was always dependent on its current instantaneous input, we need to avoid these sorts of dependencies using constant signal functions, or delays such as in **dSwitch**. Some research has investigated ensuring well-formed loops using type systems, but that is beyond the scope of this talk.

Here is a quick example, again using events and switches. This example shows a signal function that increments a robots velocity by one every time a particular button is pressed:

```
vel = proc inp -> do rec
  e <- incrVelEvs -< inp
  v <- drSwitch (constant 0) -< (inp, tag e (constant (v + 1)))
  returnA -< v
```

(`tag` is just a function that transforms its event argument e (if it's not a null value) into the value given by its second argument)

Two things to note here: first, the `rec` keyword at the beginning indicates that these definitions are mutually recursive (similar to a `letrec` in Scheme). Second, note that `v`'s definition depends on itself: it uses `v + 1` in its definition. However, this is still well-formed, since `drSwitch` guarantees that `v` only depends on its *previous* value.

3.4 Putting it all Together

Finally, let's look at one more example that combines some of these concepts. We want to build a controller for a robot that goes faster depending on how loud the user yells at it. Specifically, the requirements are the following:

1. Initially, the velocity should be proportional to the average volume the robot has sensed since its program started.
2. If the user ever presses a certain button, the robot converts into a mode where it just moves forward at some constant velocity (say, 5.0).

We will assume we have the signal functions localtime (to get the elapsed time), identity (the identity signal function), and buttonE (an event signal that triggers when the user pushes the button). We also assume that we have a function getVolume that gets the current volume from the current value of the robot's overall input signal. We'll break this up into two parts; first, the signal function to turn the volume into a velocity:

```
velFromSound = proc inp -> do
  vol <- identity -< getVolume inp
  ivol <- integral -< vol
  t <- localtime -< ()
  avgVol <- identity -< ivol / t
  returnA -< k * avgVol
```

In the above function, we use `integral` to calculate the integral of the volume over the course of the entire program's execution. We then divide that by the elapsed time to get the average volume, and multiply that by the constant `k` to get our final velocity.

Next, we want to integrate this with logic to switch between velocity modes as needed:

```
velController = switch (velFromSound &&& buttonE)
                       lambda (e). constant 5.0
```

Our overall controller is, again, a relatively simple switch statement that uses our volume-driven velocity until the button press occurs, at which point it switches to the `constant 5.0` velocity signal function.

4 Nettle: Applying FRP to programmable routers

FRP has been around for about 15 years now, but one great test of a research idea's impact is to ask, "is it being used to solve real, every-day problems?". In fact, FRP is being used today in an area of work today known as *software-defined networking*. Before we get into how people use FRP in this area, though, we need to understand what a software-defined network (or SDN) is. If you saw Jennifer Rexford's talk at POPL this year, this next bit may be a review for you.

Today, one can describe routers as having two separate parts: the data plane and the control plane. The data plane is the part of the router responsible for accepting incoming packets and forwarding them on through the correct port according to some rules in an internal flow table. The control plane is in charge of keeping that flow table up to date using some routing protocol that it's been configured to use. The routing protocols available on a given router are typically built into the device (for performance reasons), and have limited configurability. Network administrators have limited parameters of the various protocols that they can tweak, and they have no way at all of using a completely new or different protocol on these routers. What's more, since these boxes have lots of custom circuitry to implement these protocols, they tend to be expensive.

Software-defined networking takes a different approach. It says that instead of having a large number of static, expensive, complicated boxes making up our network, the network should be comprised of two separate types of entities: smart but slow controllers, and fast but dumb switches. The idea is that a network would have a small number of controllers running on commodity server hardware handling the actual routing protocols, which would then install new rules as needed into the flow tables of the switches in the network. By doing this, the routers could be simplified so as to run on common switches, while at the same time becoming more flexible and able to run more protocols.

The most common API used for this purpose is OpenFlow. A switch that implements OpenFlow allows controllers on the network to send it commands to update its routing table, and the switch in turn sends any packets it doesn't know how to handle to the controller. The controller handles each of these packets in turn and installs new rules on the switches as appropriate.

Researchers have come up with a variety of ways to program these networks, but one system of note is Nettle [6], also out of the Yale Hakell group. Their method is to model the controller software as an FRP program. Its input is a stream of OpenFlow events, such as IncomingPacket, SwitchJoin, SwitchLeave, etc., and its output is a stream of OpenFlow commands to send back to the switches. In full, the signature looks like this:

```
Controller = SF (Event (Switch ID, SwitchMessage)) (Event SwitchCommand)
```

4.1 Examples

Let's look at some examples of how this system works. The first example is a controller that just drops every packet it receives. It's not a very useful controller, but it does give a simple illustration of the idea:

```
controller = proc msgE -> do
  cmdE <- never -< msgE
  returnA -< cmdE
```

The variable `msgE` represents a value from our signal of switch messages. Note that we're dealing entirely with event signals, and therefore are working in a discrete time system. We pass this value through the signal function `never`, which is a signal function whose output type is an event signal, but which never generates an actual event—the value of the signal is always the null value. The resulting value is used as the output of our overall signal function. Putting this all together, we can see that this controller will never generate any commands, regardless of the messages it receives.

Now let's try something slightly more complex (and certainly more useful). Our next controller will, for every message it receives, tell the switch to flood the packet to the rest of the network. This isn't terribly efficient, but it will at least eventually get packets where they need to go. In addition, we'll clear the routing table of any switch that joins the network, to ensure a somewhat consistent state. We'll build this router in pieces:

```
clearOnJoin = proc msgE -> do
  returnA -< liftE f (switchJoinE msgE)
  where f (sid, _) = clearTable sid

floodPackets = proc msgE -> do
  returnA -< liftE f (packetInE msgE)
  where f (sid, packetMsg) = sendPacketIn flood (sid, packetMsg)

controller = proc msgE -> do
  clearE <- clearOnJoin -< msgE
  floodE <- floodPackets -< msgE
  returnA -< mergeEventsBy (+) (clearE, floodE)
```

There are a few new operations here that we need to define. First, `liftE` has type `(a -> b) -> Event a -> Event b` and essentially raises pure functions up to the level of events. Next, `switchJoinE` and `packetInE` are projection operations that select the switch join and packet messages, respectively, out of the stream of switch messages. At the OpenFlow level, `clearTable` and `sendPacketIn` are constructors for switch commands. Finally, `mergeEventsBy` takes two events and merges them into one, using the provided operator to handle conflicts when both events occur at once. In this case, `(+)` is the sequencing operator, so both commands would be sent.

Finally, to make something more practical, we want to create a *learning switch*. The idea behind a learning switch is that whenever a switch receives a packet marked as being from a certain address, and that packet is received through a given port, we know that there is a path to that address whose next hop is through that port. This information allows us to construct a routing table. In typical routing situations, this construction would happen on the router, but in our case, it will all be done on the controller, which will then relay this information to the switches in the network. Again, we will build this controller by piecing together a few different signal functions.

```
getNextHops = proc msgE -> do
  hostMapE <- accum empty -< liftE updateMap (packetInE msgE)
  hostMap <- hold empty -< hostMapE
  returnA -< hostMap

controller = proc msgE -> do
  nextHops <- getNextHops -< msgE
  tableModE <- identity <- mapFilterE f (packetInE msgE)
  clearE <- clearOnJoin -< msgE
  floodE <- floodPackets -< msgE
  returnA <- mergeEventsBy (+) (clearE, tableModE, floodE)
  where f (packetE) = packetToCmd nextHops packetE
```

Again, there are some new combinators to discuss. `updateMap` takes a packet and turns it into an entry in our next hop map. `mapFilterE` is a combinator that takes a function from a type `a` to a type `Maybe b` as well as an `Event a`, and turns it into an `Event b`, where there is only an event if the given function returns a `Something b` value instead of `Nothing`— in other words, we're mapping over our elements, and then filtering them to include those with actual values. Finally, `packetToCmd` will generate a rule insertion command if it can find a suitable entry in the next hops table, or a `Nothing` value otherwise.

You can see that by putting these together, we can create a sensible controller. Over time, the getNextHops rule accumulates a table of next hops, and new packets that come into the network get rules assigned to their appropriate switch. Any incoming switch messages that don't have a corresponding rule in the next hops map have their packet flooded over the network.

These have all been relatively primitive examples, but you can imagine that by using these tools and this programming model, you could program the controller to implement more complicated protocols, like OSPF or BGP, or even features like firewalls or network address translation. FRP, as used in Nettle, gives programmers a model that's easy to reason about and abstracts away many of the details of event-driven programming.

4.2 Extensions and Other Issues

There are some other interesting issues in using FRP for SDNs that deserve a brief menion here. First, you may have noticed that the above programs use a

completely discrete time system. This makes sense given our goals, but there are uses for continuous time systems in SDNs, too. For example, one could have a continuous signal that represents traffic coming through each switch, and develop a load balancer based on that.

Second, the NetCore system [4, 2] (formerly known as Frenetic) uses FRP for a similar purpose, but at a higher level. When programming directly with OpenFlow rules, such as in Nettle, it's easy to introduce race conditions in which two separate but conflicing rules are sent to a switch at roughly the same time, and it's unclear which one will override the other. Also, working at this level can make it hard to compose rules together, especially when trying to write larger and more complicated programs. Frenetic works at a higher-level, abstracting these issues away from the programmer and allowing them to work entirely in terms of patterns and rules.

5 Conclusion

To wrap up, we have seen the evolution of functional reactive programming from its beginning to where it stands today. We saw some of the initial problems with the traditional computation model that motivated a move to a more sophisticated system like FRP. We then saw how the initial attempts at this caused some initial issues, which encouraged a move towards signal functions, modeled as arrows. We also saw how the idea of arrows allows us to abstract the idea of computation and gives us a nice semantics for composing compositions together. Also in the Yampa system, we discussed its events and continuous versus discrete time, as well as recursive signal functions, both of which are very useful when programming with FRP. Finally, we looked at Nettle to see that FRP is indeed useful enough to be used in real-world applications, specifically software-defined networks. There are many more applications of FRP being used, and research into the subject is ongoing.

6 Reading Summaries

6.1 Functional Reactive Programming, Continued

In this paper, Nilsson et al. present their version of arrowized FRP (which would later become Yampa). Their intent is to avoid some of the operational and semantic problems that came about in the original FRP system. To accomplish this, they restrict the programmer to writing programs in terms of signal functions, intstead of signals directly. This restriction makes certain kinds of problematic programs impossible to write and clarifies some of the semantics. They also describe how they used the concept of arrows to implement signal functions, and they explain the combinators that work on arrows. The paper also describes some other details of their system, such as stateful signal functions and their method for dealing with discrete time and events. From there, they

go in depth into some of the implementation details, including some possible optimizations, and conclude.

I think the idea of working directly with signal functions makes sense, and it does seem to clarify some confusion between what's a signal and what's a signal function. However, they didn't explain how signals caused space-time leaks very well, which largely motivated the whole idea. Perhaps it had been covered in previous work, but a more detailed explanation would have helped.

Also, the optimizations mentioned at the end in 4.4 and 4.5 seemed a bit tacked-on. They could possibly be useful, but they're not explored enough here for me to make a good judgment. They seem to merit a future paper, and should have just been in the "Future Work" section.

6.2 Arrows, Robots, and Functional Reactive Programming

This paper was similar to FRP Continued in many ways, although it had a different audience. Whereas FRP Continued was published at the Haskell workshop, this paper was published at the Advanced Functional Programming Summer School in Oxford, so its intent is more to teach the general ideas rather than to describe new research that's been done. Again, the paper explains signal functions, arrows, arrow combinators, events, and recursive signals. The rest of the paper is dedicated to an extended example, exploring different parts of the Yampa system and providing exercises for students.

Although the paper does not present anything entirely new, it is a good source for learning the Yampa system as a whole, from a user's perspective. It mostly skips over the motivation for moving from signals to signal functions, but it does a good job of explaining how to work with signal functions and the provided combinators. The examples in the second half also provide larger examples of FRP programs that weren't available in the earlier paper.

The one weakness of this paper is that although its intended audience is end users of Yampa, it could have gone into more depth about some of the motivations behind these ideas. Although the audience may not be researchers who plan on extending this work, it's still useful for a programmer using the system to understand why one way of working with the system is a good idea and another is not.

6.3 Nettle: Taking the Sting Out of Programming Network Routers

In the Nettle paper, Voellmy and Hudak describe Nettle, an FRP-based (specifically, Yampa-based) approach for writing a controller for a software-defined network. Their approach assumes the network is running the OpenFlow API, which allows the controller to receive messages from switches in the network and remove and install rules on those switches. The system is largely modeled as a discrete system, where the input signal is a stream of switch message events, and the output stream is a stream of switch command events, which will then be

distributed to the network by the underlying implementation. They give some primitive functions for working with messages, rules, and commands, and show how these can be used to build up some controllers that model simple routing algorithms. At the end, they also mention how continuous time signals might be used to model data like traffic throughout the network, and they note this could be used for building things like load balancers.

Previous solutions to the SDN controller problem have mostly been either too low-level (e.g. NOX) or too static (e.g. FML) for real-world use. Nettle presents a solution that allows programmers to work at a higher level and respond to changes in the network, while still having a simple, clean model of the overall system. However, controller programmers will probably want an even higher-level model, such as in Frenetic, to deal with issues like rule composition and race conditions.

Also, while the examples given in the paper do get increasingly more complex, even the most sophisticated one is still rather simple. A better test of this system would be to implement one of the classic routing protocols on it, such as a distance-vector or link-state protocol, and examine how difficult this task is.

References

[1] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, volume 32 of *ICFP '97*, pages 263–273, New York, NY, USA, August 1997. ACM.

[2] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *ICFP '11*, volume 46, pages 279–291, New York, NY, USA, September 2011. ACM.

[3] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. volume 2638 of *Lecture Notes in Computer Science*, chapter 6, pages 159–187. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003.

[4] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 217–230, New York, NY, USA, 2012. ACM.

[5] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[6] Andreas Voellmy and Paul Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *PADL*, pages 235–249, 2011.