Nettle: Functional Reactive Programming for OpenFlow Networks

Andreas Voellmy Ashish Agarwal Paul Hudak

July 1, 2010

Yale University
Department of Computer Science
New Haven, CT 06520

Research Report YALEU/DCS/RR-1431

andreas.voellmy@yale.edu
ashish.agarwal@yale.edu
paul.hudak@yale.edu

Abstract

We describe a language-centric approach to solving the complex, low-level, and error-prone nature of network control. Specifically, we have designed a domain-specific language called *Nettle*, embedded in Haskell, that allows programming *OpenFlow* networks in an elegant, declarative style. Nettle is designed in layers to accommodate a family of DSLs targeted for specific network applications. The primary core of Nettle is based on the principles of *functional reactive programming* (FRP). Aside from its useful signal abstraction, FRP facilitates the integration of components written in different higher-level DSLs. We demonstrate our methodology by writing several non-trivial OpenFlow controllers.

1 Introduction

Networks continue to increase in importance and complexity, yet the means to configure them remain primitive and error prone. There is no precise language for describing what a network should do, nor how it should behave. At best, network operators document their complex requirements informally, but then are faced with the daunting and unreliable task of translating their specifications by hand into the low-level, device-specific, often arcane scripts used to control today's switches and routers. This low-level programming model often results in devices and protocols interacting in unexpected and unintended ways, and gives little hope in validating high-level protocols and policies such as those related to traffic engineering, business relationships, security, and so on.

Part of the problem is that most conventional routers are not only low-level, they are also decidedly impoverished in their expressive power, and inflexible in their configuration capabilities – it is sometime difficult to get even the most basic configurations to work correctly. Another problem is that conventional routers are designed to work autonomously, and not collaboratively with other routers to achieve a more sophisticated global behavior.

We believe that these problems can be overcome through the use of advanced high-level programming languages and tools that allow one to express the overall network behavior as a single program expressed in a declarative style. Although this idea has been suggested by several researchers [3, 7], the development of an actual solution has been elusive. There are two aspects of our approach that we believe will result in a successful outcome: First, we abandon conventional routers in favor of *OpenFlow switches* [1]. OpenFlow presents a unified, flexible, dynamic, remotely programmable interface that allows network switches to be controlled from a logically centralized location.

Second, we use advanced programming language ideas to ensure that our programming model is expressive, natural, concise, and designed precisely for networking applications. In particular, we borrow ideas from *functional reactive programming* (FRP) and adopt the design methodology of *domain-specific language* (DSL) research.

Our overall approach, which we call *Nettle*, allows us to radically rethink the problem of network configuration. Rather than configure a collection of black boxes, we *control* a collection of programmable components. In doing this, we enable the development of new control algorithms and most importantly, more powerful and natural control languages. This approach makes it feasible to provide application-specific languages, to express interdomain business policies or security requirements, for example. It enables programmers to design new dynamic traffic control algorithms not provided in today's devices.

2 Approach

Because computer networks come in many different shapes, sizes, and purposes, it would be a mistake to try designing single language that "did it all." Our approach is to instead design an *extensible family of DSLs*, each capturing an important network abstraction. For example, we may have one DSL for access control policies, another for traffic engineering strategies, and another for expressing interdomain contracts. And because the family is extensible, new abstractions can easily

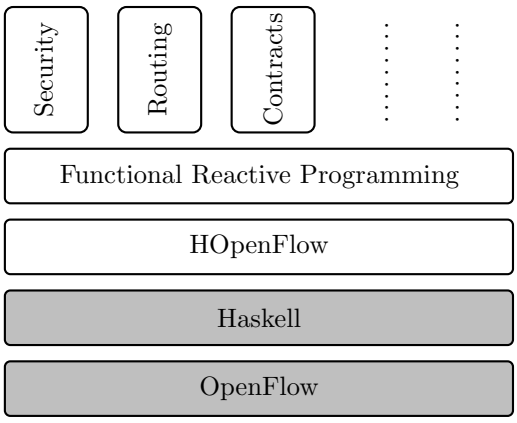

Figure 1: Nettle layered system architecture.

be added.

Furthermore, to avoid creating small, isolated DSLs, we rely on the technique of *embedding* each DSL into a host language. We choose Haskell [11] as our host because of its remarkable flexibility in supporting embedded DSLs [5]. This approach allows our DSLs to share a common "look and feel" through the adoption of the same language infrastructure, such as variable naming conventions, function definitions, primitive data types, a powerful type system, and so on. This not only relieves us from the burden of implementing these general features, allowing us to focus on domain-specific concepts, but more importantly allows the DSLs to *interoperate* with one another.

Figure 1 illustrates our approach. At the bottom are OpenFlow switches, an enabling technology for our work. One level up is Haskell, our host language. Above that is a library, HOpenFlow, that abstractly captures the OpenFlow protocol.

The next layer in our stack is an instantiation of a language in the Functional Reactive Programming (FRP) paradigm. FRP is a family of languages that provide an expressive and mathematically sound approach to programming real-time interactive systems in a declarative manner. FRP-based languages have been used successfully in computer animation, robotics, control systems, GUIs, interactive multimedia, and other areas in which there is a combination of both continuous and discrete entities [9, 10, 12, 4]. Network control programs share many of these characteristics, and we expect that FRP will be a key technology in the modular implementation of control systems and higher-level protocols.

Above the FRP layer, we implement our extensible family of DSLs, each member capturing a particular domain. These more specific DSLs will be implemented in terms of Haskell and FRP. This allows us to integrate the components of different domains in the framework of FRP, which provides an expressive and generic language for composing components and controlling their dynamic structure. Previous work by Pembeci *et al.*[8] has demonstrated the effectiveness of using FRP as a system integration language in the area of robotics, and we intend to follow a similar approach to the integration of disparate network control components.

3 OpenFlow

The OpenFlow specification roughly and informally defines an abstract operational semantics of an OpenFlow switch, and defines a network protocol for remotely interacting with OpenFlow switches by sending and receiving OpenFlow-specific messages. The basic architecture consists of OpenFlow switches, a controller, and end hosts. The switches communicate with the controller over a secure TCP connection, typically on a dedicated control network (distinct from the main data network).

OpenFlow switches maintain a *flow table* containing entries consisting of a *match* condition and a list of *forwarding action*s. The match condition can optionally match on most Ethernet, IP, or transport protocol header fields. The forwarding actions include forwarding to specific ports on the switch, flooding the packet, and many other options. When a packet is received by a switch, it searches for a matching entry. If matches are found, the highest priority one is chosen and its actions are executed. If the list of actions is empty, the packet is dropped. If no match is found, the packet is encapsulated and sent to the controller in a format defined by the OpenFlow protocol.

The controller can use the OpenFlow protocol to modify switches' flow tables and command switches to send packets.

4 Functional Reactive Programming

In this section we briefly introduce the key ideas and constructs of the functional reactive programming (FRP) layer of Nettle. The design of this layer is strongly influenced by *Yampa* [6], an FRP-based DSL that we previously designed for use in robotics and animation.

The simplest way to understand this layer is to think of it as a language for expressing electrical circuits. We refer to the wires in a typical circuit diagram as *signals*, and the boxes (that convert one signal into another) as *signal functions*. For example, this very simple circuit has two signals, x and y, and one signal function, *sigfun*:

This is written as a code fragment in Nettle simply as:

$y \leftarrow sigfun \multimap x$

Nettle has many built-in signal functions, including all of the obvious numeric functions, as well as ones for integration and differentiation of signals. Of course one can also define new signal functions. For example, here is a definion for *sigfun* above that simply returns a signal that always takes the sine of one greater than its input:

$sigfun :: SF\ Float\ Float$
$sigfun = \mathbf{proc}\ x \rightarrow \mathbf{do}$
  $y \leftarrow sin \multimap x + 1$
  $returnA \multimap y$

The first line in this program is a type signature that declares that *sigfun* is a signal function that converts continuous values of type *Float* to continuous values of type *Float*.

In Nettle we can use signals and signal functions in this way to program, for example, controllers that alter traffic flow based on signals that measure the volume of traffic on particular links. But in this report we emphasize a different use: we will use signals to represent *streams of control messages* flowing to and from our OpenFlow switches – you can think of each signal (i.e. wire) as being a stream of messages.

In a conventional language, a message- or event-based system might be implemented by some kind of call-back mechanism and a loop that handles messages as they arise, one by one. But in Nettle, it is done much more declaratively, where we think of, and program with, message streams as a whole. For example, the merger of two message streams *ms1* and *ms2* is simply $ms1 .|. ms2$.

A message, of course, carries data, and sometime we need to manipulate the data in each message of a message stream. We can apply a function *fn* to each message in a message stream *ms* by the simple expression $ms \ggg fn$. Sometimes our chore is even simpler: we may want to simply replace each message with a different one, say m, which can be written $ms \mathbin{-\!\!\gg} m$. We will use both of these operators in later examples.

A slightly more complicated operation on event streams we will use later is *mapFilterE*. If *fn* is a function returning optional values, i.e. having type $a \to Maybe\ b$, then *mapFilterE fn ms* applies *fn* to each message in the stream *ms*, but only replaces the messages for which a non-null value is produced by *fn*. In those cases it produces the value given by *fn*.

Nettle elegantly unifies message streams with continually varying signals by representing message streams as continuous signals that are only defined at discrete points in time. More concretely, Nettle represents a discrete signal that periodically carries information of some type α as signals that take on values of the *Event* α datatype, whose values are either *NoEvent* or *Event a* for some $a :: \alpha$. Note that *Event* α is isomorphic to Haskell's *Maybe* α data type.

For example, a signal function that converts a message stream carrying messages of type *M1* into a message stream carrying messages of type *M2* has type *SF* (*Event M1*) (*Event M2*).

Nettle provides several constructs that convert between discrete and continuous signals. One we will use later is $hold :: a \to SF\ (Event\ a)\ a$, that converts a stream of events carrying values of type a into a continuous, piece-wise constant signal of a values. The output signal of *hold a0* "holds" the last a value received on its input line, and starts out as *a0*.

Nettle also includes several *stateful* signal functions, i.e. signal functions whose output signal at some moment in time depends on the values of the input signal at earlier moments in time. One such signal function is $accum :: a \to SF\ (Event\ (a \to a))\ (Event\ a)$. *accum a0* takes as input an event stream carrying state-modifying functions. At each event in its input stream, it applies the state-modifying function carried by the event to the current state, updates the current state with that new value, and outputs an event carrying the updated value.

5 FRP for OpenFlow Control

At the most basic level, Nettle programs are signal functions operating on input types that carry, among other data, messages from OpenFlow switches, and output types that carry, among other things, commands for OpenFlow switches. More concretely, Nettle programs are signal functions having a type of the form *SF i o* for some domain-specific types *i* and *o*.

Admittedly, this representation of a network control program – as a transformation of a timed stream of messages into a timed stream of commands – is fairly low-level, since it says nothing about high-level networking concepts such as routing, load balancing, security, etc. Surprisingly, though, these basic tools – Haskell, FRP, and domain-specific input and output types for the OpenFlow protocol – provide an excellent basis for the exploration and implementation of a variety of network control algorithms and abstractions. At the same time, they do not impose a restrictive high-level language on Nettle controllers, which we feel would be premature. Rather, this collection of tools provides an ideal platform for the construction of higher-level abstractions and for the integration of components written in terms of different abstractions.

In the following sections, we describe how we tailor FRP and Haskell to the domain of OpenFlow network control, and we demonstrate, with some small examples, how one can program controllers in this language.

5.1 Switch Events

OpenFlow switches establish TCP connections with the controller, transmit OpenFlow messages to the controller, and under some circumstances terminate their connections with the controller. In each of these events, a controller may need to take some actions. We represent these events, and indicate to which switch an event pertains, with values of the *SwitchEvent* datatype:

```
data SwitchEvent = SwitchUp      SockAddr
                 | SwitchDown    SockAddr
                 | SwitchMessage SockAddr SCMessage
```

We use values of the type *SockAddr* to identify a switch. The *SockAddr* value associated with a switch is the IP address and transport port number that the controller uses to communicate with that switch.

The *SCMessage* datatype represents the logical content of the OpenFlow messages that a switch may send to the controller, abstract from their binary OpenFlow formats. Figure 2 shows the definition of this datatype. Here we look at one variant of this datatype, representing packet-in messages, in more detail.

Packet-in messages are among the most important switch-to-controller message types. A switch sends a packet-in message when it receives a packet for which it has no matching flow entry in its flow table, or if a matching entry directs the switch to send the packet to the controller. The *PacketInfo* type represents the logical information carried by such a message:

```
data SCMessage
   = SCHello'           OpenFlowVersion
   | SCEchoRequest'     TransactionID [Word8]
   | SCEchoReply'       TransactionID [Word8]
   | PacketIn'          PacketInfo
   | FlowRemoved'       FlowRemoved
   | PortStatus'        PortStatus
   | FeaturesReply'     SwitchFeatures
   | GetConfigReply'    SwitchConfig
   | StatsReply'        StatsReply
   | Err' Error
...
```

Figure 2: Logical representation of switch-to-controller OpenFlow messages

```
data PacketInfo = PacketInfo {
   packetBufferID :: Maybe BufferID,    -- buffer ID, if packet buffered at switch
   packetOrigLen  :: NumBytes,          -- full length of frame received by switch
   packetInPort   :: PortID,            -- port on which frame was received
   packetInReason :: PacketInReason,    -- reason packet is being sent
   packetData     :: [Word8]            -- ethernet frame received by switch
}
```

In addition to being independent from the binary representation of the message, this type abstracts several other low-level details. For example, a packet-in message may or may not include a buffer ID, depending on whether the sending switch buffered the packet or not. In the OpenFlow message format, the absence of a buffer ID is indicated by a -1 value for this field of the message. Our Haskell representation makes this distinction explicit by modeling the bufferID with a Maybe type, which represents an optional value. This design helps to prevent programmer error, by enabling the Haskell type system to force programmers to acknowledge the possibility of a missing buffer ID.

5.2 Switch Commands

Once a switch establishes a connection with the controller, the controller may send *switch commands* to the switch. Nettle provides a language of switch commands, partially shown in Figure 3, for this purpose. For example,

```
deleteFlowRules dhcp switch1
```

deletes all flow rules from *switch1* whose match condition matches any subset of *dhcp* packets. The identifier *switch1* has type *SockAddr*. The identifier *dhcp* is of type *PacketPredicate* and is defined using a small language of predicates, shown in Figure 4. This language of packet predicates provides a convenient method of describing complex families of packets. In this instance, *dhcp* is

defined as:

$$\begin{aligned} ip &= ethFrameTypeIs\ ethTypeIP \\ udp &= ip \wedge transportProtocolIs\ udpCode \\ dhcp &= udp \wedge (senderTransportIs\ dhcpPort \vee \\ &\qquad receiverTransportIs\ dhcpPort) \end{aligned}$$

These definitions say that *ip* packets are those Ethernet frames whose Ethernet frame type is *ethTypeIP*, that *udp* packets are those *ip* packets having a transport protocol with code *udpCode*, and that *dhcp* packets are *udp* packets having either the sender or the receiver transport port equal to *dhcpPort*.

As a second example, we can write

$$addFlowRule\ (priority\ 1)\ (http \Longrightarrow [sendOnPort\ 1, sendOnPort\ 2])\ switch2$$

to insert flow entries on *switch2* at priority level 1 that forwards *http* traffic on ports 1 and 2. In this example we have used the $\Longrightarrow$ infix operator which provides a suggestive syntax, but which does nothing more than pair a *PacketPredicate* with a list of *Action*s. Again, we make use of packet predicates to define *http*:

$$\begin{aligned} http = {} & ethFrameTypeIs\ ethTypeIP \quad \wedge \\ & transportProtocolIs\ ipTypeTcp \wedge \\ & receiverTransportIn\ [80, 443, 8080] \end{aligned}$$

In this definition we use the function $receiverTransportIn :: [TransportPort] \rightarrow PacketPredicate$ to concisely denote a predicate that matches packets whose receiver transport port is among those listed. It is important to note that this function, while defined in the library, is not primitive, and could easily have been defined by a user. Indeed, it is defined as:

$$\begin{aligned} receiverTransportIn\ ps &= anyP\ [receiverTransportIs\ p \mid p \leftarrow ps] \\ anyP &= foldl\ (\vee)\ noPacket \end{aligned}$$

This definition demonstrates the utility of the embedding technique in enabling users to use familiar tools to extend the basic language.

Furthermore, Nettle provides a binary operator $\oplus$ that combines multiple commands into a single composite command that, when executed, will perform both commands. For example, we can combine the two commands above as:

$$\begin{aligned} & deleteFlowRules\ dhcp\ switch1 \oplus \\ & addFlowRule\ (priority\ 1)\ (http \Longrightarrow [sendOnPort\ 1, sendOnPort\ 2])\ switch2 \end{aligned}$$

$$
\begin{array}{rl}
\mathit{requestFeatures} & :: \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\mathit{addFlowRule} & :: \mathit{Priority} \to (\mathit{PacketPredicate}, [\mathit{Action}]) \to \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\mathit{deleteFlowRules} & :: \mathit{PacketPredicate} \to \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\mathit{sendBufferedPacket} & :: \mathit{BufferID} \to \mathit{Maybe\ PortID} \to [\mathit{Action}] \to \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\mathit{sendPacket} & :: \mathit{ByteString} \to \mathit{Maybe\ PortID} \to [\mathit{Action}] \to \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\mathit{setPortSettings} & :: \mathit{PortID} \to \mathit{EthernetAddress} \to \mathit{Map\ PortAttribute\ Bool} \to \mathit{SockAddr} \to \mathit{SwitchCommand} \\
\ldots &
\end{array}
$$

Figure 3: Switch commands

In fact, we make the type of switch commands, *SwitchCommand*, an instance of the *Monoid* type class, which supports the following operations:

```
class Monoid m where
  mempty  :: m                 -- unit w.r.t. mappend
  mappend :: m → m → m         -- associative binary operation
```

We define *mappend* for the *SwitchCommand* type to be synonymous with $\oplus$, and define *noOp* :: *SwitchCommand* as a suggestive synonym for *mempty*.

With these building blocks, we can now apply standard functional programming techniques to extend the language with new commands defined in terms of the basic commands. For example, one can write a function that turns an entire table of flow rules into a flow table update command, making use of the standard function $\mathit{mconcat} :: \mathit{Monoid}\ m \Rightarrow [m] \to m$:

```
addFlowRules :: Priority → [(PacketPredicate, [Action])] → SockAddr → SwitchCommand
addFlowRules priority rules switch =
  mconcat [addFlowRule priority rule switch | rule ← rules]
```

Clearly, composite *SwitchCommand*s that address different switches will require multiple OpenFlow messages to be sent. In fact, even basic *SwitchCommand* values often require multiple OpenFlow messages. One reason for this is that the *PacketPredicate* language is richer than the underlying match constructs provided by OpenFlow. In particular, *PacketPredicate*s support disjunction, whereas OpenFlow matches do not. However, commands involving *PacketPredicate*s can be simulated, often by using multiple commands and flow table rules. For example, the implementation *deleteFlowRules dhcp switch1* sends two delete commands to *switch1*, one for each disjunct of the equivalent disjunctive normal form for *dhcp*, which happens to be:

$$(\mathit{udp} \wedge \mathit{senderTransportIs\ dhcpPort}) \vee (\mathit{udp} \wedge \mathit{receiverTransportIs\ dhcpPort})$$

$(\wedge), (\vee)$ :: $PacketPredicate \to PacketPredicate \to PacketPredicate$
$anyPacket, noPacket$ :: $PacketPredicate$
$inPortIs$ :: $PortID \to PacketPredicate$
$ethSourceIs, ethDestIs$ :: $EthernetAddress \to PacketPredicate$
$ethFrameTypeIs$ :: $EthernetFrameType \to PacketPredicate$
$transportProtocolIs$ :: $TransportProtocol \to PacketPredicate$
$senderTransportIs, receiverTransportIs$ :: $TransportPort \to PacketPredicate$
...

Figure 4: Packet Predicates

5.3 Generic Controllers

While many Nettle programs can be written simply as signal functions with input type *SwitchEvent* and output type *SwitchCommand*, i.e. as *SF SwitchEvent SwitchCommand*, realistic control systems will incorporate input sources other than *SwitchEvent*s and will communicate with or command devices other than switches. For example, some controllers may interact with intrusion detection systems, in addition to controlling OpenFlow switches, in order to proactively implement network security goals. Other controllers may interact with user interface elements to allow operators to interactively operate some part or aspect of the network, for example to toggle between different routing modes on command. Therefore, each Nettle-controlled system may need input and output types tailored to the particular system. At the same time, many controller modules will be relatively generic and will work primarily with OpenFlow switch events and switch commands. We would like to write these components generically, rather than write them for specific input and output types and consequently be forced to rewrite them for each system.

To accomplish this, Nettle structures its input and output types using Haskell's *type classes*. With type classes, Nettle defines functional interfaces that types may support, and controllers can then be written generically by having their input and output types depend on this interface, rather than a specific implementation. Figure 5 gives some of the output type classes provided by Nettle. The output class for types carrying switch commands is *HasSwitchCommands* and it is identical to the switch command language, except for the result types of the functions. The *HasConsoleOutput* type is implemented by types carrying console output commands.

Both type class declarations require that instances implement the *Monoid* type class. This ensures that commands of different type classes can be combined. For example, this enables one to write the following command to both send a packet and to output a message:

$sendBufferedPacket\ bufID\ (port\ 1)\ [flood]\ switch1 \oplus$
$consoleOut\ (\text{“}Sent\ packet\ with\ buffer\ id : \text{“} +\!\!+ show\ bufID)$

Figure 6 shows some of the input classes supported by Nettle. The *HasSwitchEvents* class is implemented by types which may carry (among other things) *SwitchEvent* values. The *HasConsoleInput* type class is implemented by types carrying input from a console window on the controller machine,

```
class Monoid o ⇒ HasSwitchCommands o where
    requestFeatures    :: SockAddr → o
    addFlowRule        :: Priority → (PacketPredicate, [Action]) → SockAddr → o
    deleteFlowRules    :: PacketPredicate → SockAddr → o
    sendBufferedPacket :: BufferID   → Maybe PortID → [Action] → SockAddr → o
    sendPacket         :: ByteString → Maybe PortID → [Action] → SockAddr → o
    setPortSettings    :: PortID → EthernetAddress → Map PortAttribute Bool → SockAddr → o
        ...
class Monoid o ⇒ HasConsoleOutput o where
  consoleOut :: String → o
```

Figure 5: Type classes for output.

```
class HasSwitchEvents i where
  switchEventE :: i → Event SwitchEvent
class HasConsoleInput i where
  consoleInputE :: i → Event String
class HasMessengerInput i where
  messengerMessageE :: i → Event (SockAddr, Message)
```

Figure 6: Type classes for input.

while the *HasMessengerInput* is implemented by types carrying input from a Messenger server. The Messenger server is a simple, text-based TCP server which accepts ASCII messages from networked devices. A controller may use the Messenger server to accept input from other networked devices, such as intrusion detection systems, or authentication systems. A Messenger message includes the text of the message as well as the *SockAddr* of the sender.

In addition, Nettle includes a collection of functions, shown in Figure 7, implemented in terms of the *HasSwitchEvents* type class, and available in every instance of the type class. These functions, effectively turn the input signal into a stream of some particular kind of event. For example, *arr packetInE* has type *HasSwitchEvents i* ⇒ *SF i* (*Event PacketIn*), and is a signal that outputs the stream of packet-in events sent to the controller. The Nettle primitive $arr :: (a \to b) \to SF\ a\ b$ lifts an ordinary function to the signal function level, simply applying the function pointwise on the input signal.

5.4 Running Nettle Programs

To run a Nettle program, one must (1) define application-specific input and output types and make them instances of the relevant type classes, such as *HasSwitchEvents* and *HasSwitchCommands*, (2) define drivers, i.e. IO commands to produce values of the input types and consume values of the output types, and (3) drive the Nettle signal function with these drivers. Currently, we provide several input and output types and I/O drivers of several kinds, such as TCP servers for interacting with OpenFlow switches and devices using the Messenger protocol. Our intention is that Nettle should provide a convenient, declarative way to define these three components, but this part of the

$$
\begin{array}{ll}
\mathit{switchUpE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ \mathit{SockAddr} \\
\mathit{switchDownE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ \mathit{SockAddr} \\
\mathit{switchMessageE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{SCMessage}) \\
\mathit{helloE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{OpenFlowVersion}) \\
\mathit{echoRequestE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{TransactionID}, [\mathit{Word8}]) \\
\mathit{packetInE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{PacketInfo}) \\
\mathit{switchFeatureE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{SwitchFeaturesRecord}) \\
\mathit{portStatusE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{PortStatusRecord}) \\
\mathit{switchErrorE} & :: \mathit{HasSwitchEvents}\ i \Rightarrow i \rightarrow \mathit{Event}\ (\mathit{SockAddr}, \mathit{SwitchErrorRecord})
\end{array}
$$

Figure 7: Functions available in all types implementing *HasSwitchEvents*.

library is still in development and currently users must write a small amount of code to develop custom input and output types and drivers. We therefore omit detailed disussion here and just present one way of driving a Nettle signal function.

The Nettle library defines the types *NettleInput* and *NettleOutput* that implement all the input and output type classes mentioned above, and a signal function driver,

$$\mathit{nettleDriver} :: (\mathit{ControllerServerPort}, \mathit{MessengerServerPort}) \rightarrow \mathit{SF}\ \mathit{NettleInput}\ \mathit{NettleOutput} \rightarrow \mathit{IO}\ ()$$

to drive any controller using these type classes. This driver starts up a TCP server at the *ControllerServerPort* value that communicates with OpenFlow switches, and a TCP server to communicate with devices using the Messenger protocol at the *MessengerServerPort* value. Additionally, it performs the basic interaction needed to follow the OpenFlow protocol, such as negotiating an OpenFlow version and responding to echo requests from switches. Therefore, to run an OpenFlow controller, a user simply defines their main function to invoke *nettleDriver* for an appropriate controller, as in:

$$
\begin{array}{l}
\mathit{main} :: \mathit{IO}\ () \\
\mathit{main} = \mathit{nettleDriver}\ (2525, 9999)\ \mathit{controller} \\
\mathit{controller} = ...
\end{array}
$$

This *main* program can be compiled to an executable with a Haskell compiler, or run interactively using a Haskell interpreter.

The implementation of Nettle, including the TCP servers and parsing and unparsing libraries for OpenFlow messages, is written entirely in Haskell. Implementing the entire library in Haskell provides us with great flexibility to redesign any aspect of our language and implementation.

6 Examples

In this section we demonstrate how controllers can be written in this language with some small example controllers.

The following is a controller that prints every received packet to the console:

```
sf1 :: (HasSwitchEvents i, HasConsoleOutput o) ⇒ SF i (Event o)
sf1 = proc i → do
  returnA ⤙ packetInE i ⋙ λe → consoleOut (show e)
```

The following controller acts as *sf1*, but also explicitly floods every packet:

```
sf2 :: (HasSwitchEvents i, HasConsoleOutput o, HasSwitchCommands o) ⇒ SF i (Event o)
sf2 = proc i → do
  returnA ⤙ packetInE i ⋙ λe → consoleOut (show e) ⊕ sendReceivedPacket e [flood]
```

In this example, we have made use of a Nettle function, *sendReceivedPacket*::*HasSwitchCommands o* ⇒ (*SockAddr*, *PacketInfo*) → [*Action*] → *o*, easily defined in terms of the functions explained above, that returns a command to perform an action on a particular packet received by a switch.

The previous controller never installs flow table entries at the switches, and hence must process each packet in the network. Instead, the following controller installs a flow table entry at each switch to flood all packets, and this is installed whenever a switch connects with the controller:

```
sf3 :: (HasSwitchEvents i, HasConsoleOutput o, HasSwitchCommands o) ⇒ SF i (Event o)
sf3 = proc i → do
  returnA ⤙ helloE i ⋙ λ(sw, _) → addFlowRule (priority 1) (anyPacket ⟹ [flood]) sw
```

This example illustrates that controllers can act proactively to configure the flow tables of switches, rather than reacting to packet arrivals.

We can also define controllers that will provide useful functionality for many other controllers. For example, the following controller deletes all flow table entries when a switch connects with a controller:

```
switchInitializer :: (HasSwitchEvents i, HasSwitchCommands o) ⇒ SF i (Event o)
switchInitializer = proc i → do
  returnA ⤙ helloE i ⋙ λ(sw, _) → deleteFlowRules anyPacket sw
```

Another useful signal function is *hostDirectionTracker*, which is not a complete controller by itself, but provides useful information to other controllers. This signal function maintains a mapping indicating for each host and switch, on which port, if any, the switch most recently received a packet from the host:

```
type HostDirectionMap = Map (SockAddr, EthernetAddress) PortID
hostDirectionTracker :: HasSwitchEvents i ⇒ SF i HostDirectionMap
```

```
hostDirectionTracker =
  proc i →
    hold Map.empty ⋘ accum Map.empty ⤙ packetInE i ⋙ update
  where update (sw, pktInfo) hdMap =
    let inPort      = packetInPort pktInfo
        ethSrcAddr = sourceAddress (enclosedFrame pktInfo)
    in  Map.insert (sw, ethSrcAddr) inPort hdMap
```

Unlike previous examples, this signal function is *stateful*, and it accomplishes this by using the *hold* and *accum* signal functions mentioned in Section 4. The output of this signal function starts off as the empty map. It then holds its value until a packet-in event is received, at which point it outputs the map resulting from applying the update function to the current map, and repeats this process indefinitely. The update function makes use of the Nettle library function *enclosedFrame* :: *PacketInfo* → *EthernetFrame*, which parses the packet data enclosed in a *PacketInfo* value into a structured representation of an Ethernet frame.

Over time, *hostDirectionTracker* will construct a map indicating in which direction a switch should send a packet to a host. This can be used as a component of a so-called *learning switch controller*. Traditionally, a learning switch is an Ethernet switch which initially acts much like an Ethernet bridge, flooding frames received on one port to all other ports. However, a learning switch also maintains a table of Ethernet addresses and ports, such that if (a, p) is in the table, then p is the port at which the switch most recently received a frame from the host with address a. When a switch receives a frame addressed to a, it forwards the frame on port p if (a, p) is in its table at that time, or else floods it on all ports other than the incoming one.

Our implementation of the learning switch essentially involves the parallel composition of two signal functions. One signal function, *tableUpdater*, manages the forwarding tables of the switches, while the other signal function, *packetSender*, handles packets that fail to match any entries at the switches:

```
learningController = proc i → do
  tableCmds ← tableUpdater  ⤙ i
  sendCmds  ← packetSender ⤙ i
  returnA     ⤙ tableCmds ⊕ sendCmds
```

The signal function *tableUpdater* initializes the flow table of any switch connecting with the controller by clearing its flow table. In addition, it tracks host locations using *hostDirectionTracker* signal function defined above, and updates the forwarding tables of the switches by inserting appropriate flow entries whenever two hosts whose locations have been learned attempt to communicate:

```
tableUpdater = proc i → do
  clearCmd   ← switchInitializer       ⤙ i
  hostDirMap ← hostDirectionTracker ⤙ i
```

$$
\begin{aligned}
&\mathit{updateCmd} \;=\; \mathit{mapFilterE}\ (\mathit{updateTableCommands}\ \mathit{hostDirMap})\ (\mathit{packetInE}\ i)\\
&\mathit{returnA} \;\;\longrightarrow\!\!\prec \mathit{clearCmd} \oplus \mathit{updateCmd}
\end{aligned}
$$

The heart of the logic for *tableUpdater* is in the *updateTableCommands* function. This function takes as arguments the current host direction map and a packet-in event, and optionally returns a command to update the flow tables of switches. It does *not* update the flow tables if either the location of the source or destination of the packet is unknown and returns *Nothing* in this case. It is in fact crucial to the correctness of this controller that it not add flow table entries matching only on the destination address of a frame, because if it did so, the entry would match frames whose sender location is unknown. These frames would be forwarded toward the destination rather than be sent to the controller, and hence the controller would not learn about the sender's location. Thus, such a flow table entry would interfere with the "learning"process of the controller.

In the case that the locations of both the sender s and receiver r of a packet are known and their source and destination ports on switch sw are ps and pr, respectively, *updateTableCommands* will return a command that updates the flow table of the sending switch sw with a pair of flow table entries, using this command:

$$
\begin{aligned}
&\mathit{addFlowRules}\ (\mathit{priority}\ 1)\ [\mathit{flowFromTo}\ s\ ps\ r\ pr, \mathit{flowFromTo}\ r\ pr\ s\ ps]\ sw\\
&\quad \textbf{where}\ \mathit{flowFromTo}\ s\ ps\ r\ pr = \mathit{inPortIs}\ ps \wedge \mathit{ethSourceDestAre}\ s\ r \Longrightarrow [\mathit{sendOnPort}\ pr]
\end{aligned}
$$

Here we use a function *ethSourceDestAre* defined as

$$
\mathit{ethSourceDestAre}\ s\ d = \mathit{ethSourceIs}\ s \wedge \mathit{ethDestIs}\ d
$$

Note that the inserted rules match on the expected incoming port of a frame from a source. If the source changes location in the network, connecting to a different switch or port, the previously installed flow entry will no longer match that source's frames and frames from this source will be sent to the controller. This allows the controller to adjust the flow tables for the source's new location. In light of this, the controller removes any existing flow entries for a source-destination pair when adding new entries. Concretely, the command returned by *updateTableCommands* is:

$$
\begin{aligned}
&\mathit{deleteFlowRules}\ (\mathit{ethSourceDestAre}\ s\ r \vee \mathit{ethSourceDestAre}\ r\ s))\ sw \oplus\\
&\mathit{addFlowRules}\ (\mathit{priority}\ 1)\ [\mathit{flowFromTo}\ s\ ps\ r\ pr, \mathit{flowFromTo}\ r\ pr\ s\ ps]\ sw
\end{aligned}
$$

Figure 8 shows the *updateTableCommands* function in its entirety. Note that we make use of the *Maybe* monad, to succinctly express the conditions under which the function returns *Nothing*, that is, whenever the address lookups fail.

The signal function *packetSender* is straightforward. It simply floods every packet for which a packet-in message is received:

$$
\begin{aligned}
&\mathit{packetSender} = \textbf{proc}\ i \rightarrow \textbf{do}\\
&\quad \mathit{returnA} \longrightarrow\!\!\prec \mathit{packetInE}\ i \ggg \lambda e \rightarrow \mathit{sendReceivedPacket}\ e\ [\mathit{flood}]
\end{aligned}
$$

```
updateTableCommands hostDirMap (sw, pktInfo) =
  do let ethFrame = packetInFrame pktInfo
     let s        = sourceAddress ethFrame
     let r        = destAddress ethFrame
     ps           ← Map.lookup (sw, s) hostDirMap
     pr           ← Map.lookup (sw, r) hostDirMap
     return (deleteFlowRules (ethSourceDestAre s r ∨ ethSourceDestAre r s)) sw
            addFlowRules (priority 1) [flowFromTo s ps r pr, flowFromTo r pr s ps] sw)
  where flowFromTo s ps r pr = inPortIs ps ∧ ethSourceDestAre s r ⟹ [sendOnPort pr]
        ethSourceDestAre s d = ethSourceIs s ∧ ethDestIs d
```

Figure 8: The *updateTableCommands* function

This ensures that packets from or to hosts with unknown locations are sent toward their destination. As the *tableUpdater* learns the locations of hosts and installs flow entries in the switches, most frames will be forwarded directly by the switches and will no longer be flooded by the *packetSender*.

7 Related Work

NOX [2] is an open-source library for writing controllers of OpenFlow switches in C++ and Python. NOX aims to provide a convenient method to program controllers, and hides many of the low-level details, such as the binary formats of OpenFlow messages, from controller writers. It provides an event-driven programming model in which modules, implemented as C++ classes, can register to handle events generated by other modules and can generate events themselves.

We have not conducted a thorough investigation of NOX's features, and therefore cannot give a detailed comparison of Nettle and NOX here. Instead, we highlight some of the main similarities and differences.

NOX and Nettle both provide a setting in which to write programs that respond to events from switches and can generate output for switches. In NOX, users respond to events by registering imperative callbacks, while in Nettle users program more declaratively with entire event streams.

As in NOX, Nettle programs expose an event-based interface. In NOX, this is done by registering handlers in a global registry and generating events. In this model it is unclear which components will be interacting, and in what order. In contrast, Nettle programs make their event interface explicit in their input and output types and their interactions can be precisely described by connecting signal functions using the arrow combinators. Nettle programs can broadcast events, just as NOX modules do, but they can also interact in more varied ways.

By using FRP, Nettle also provides continuous quantities, which are essentially a non-event based form of module interaction, in which modules continually interact. NOX does not provide a comparable functionality.

8 Conclusion

In this report we described Nettle, an embedded DSL in Haskell, for dynamically controlling OpenFlow switches and demonstrated several example controllers in this style. Nettle is based on the Yampa language, a language for functional reactive programming.

This layer of Nettle will form a crucial platform for the construction of higher-level languages for expressing particular network concerns, such as dynamic traffic control, security, and business contracts and objectives. In upcoming work, we intend to design DSLs for some of these areas and to utilize Nettle to integrate expressions in different languages to create complete network control applications.

9 Acknowledgements

This research was supported in part by an STTR grant from the Defense Advanced Research Projects Agency. We wish to thank our STTR industrial partner, Galois, Inc. for its support as well. Vijay Ramachandran motivated our initial foray into language design for networking.

References

[1] `http://www.openflowswitch.org/`.

[2] `http://noxrepo.org/wp/`.

[3] M. Caesar and J. Rexford. BGP routing policies in ISP networks. *Network, IEEE*, 19(6):5 – 11, nov.-dec. 2005.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.

[5] P. Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), Dec. 1996.

[6] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Robots, arrows, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University*. Springer Verlag, LNCS 2638, 2003.

[7] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, pages 3–17, Pittsburgh, PA, Aug. 2002.

[8] I. Pembeci, H. Nilsson, and G. Hager. Functional reactive robotics: an exercise in principled integration of domain-specific languages. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 168–179, New York, NY, USA, 2002. ACM.

[9] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[10] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.

[11] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[12] A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. Int'l Conference on Software Engineering*, May 1999.