Frenetic: A High-Level Language for OpenFlow Networks

Nate Foster[†] Rob Harrison[*] Matthew L. Meola[*]
Michael J. Freedman[*] Jennifer Rexford[*] David Walker[*]
[†] Cornell University [*] Princeton University

ABSTRACT

Most interfaces for programming network devices are defined at the low level of abstraction supported by the underlying hardware, which leads to complicated programs that are prone to errors. This paper proposes a high-level programming language for OpenFlow networks based on ideas originally developed in the functional programming community. Our language, called Frenetic, includes a rich pattern algebra for classifying packets, a "program like you see every packet" abstraction, and a run-time system that automatically generates the low-level packet-processing rules. We describe the design and implementation of Frenetic, and show how to use it to implement common management tasks.

1. INTRODUCTION

Network administrators must configure network devices to provide services such as routing, load balancing, traffic monitoring, and access control. Unfortunately, most interfaces for programming network devices offer primitive abstractions derived from the capabilities of the underlying hardware. We argue for raising the level of abstraction, drawing on techniques from the programming languages community. In particular, we propose Frenetic, a language with high-level packet-processing operators inspired by previous work on functional reactive programming [5]. Frenetic simplifies the task of programming OpenFlow [9] networks, without compromising flexibility and efficiency.

In an OpenFlow network, a central controller manages switches that support the concept of a flow—i.e., a stream of related packets that are processed in the same way. Every switch maintains a *flow table* containing a set of rules, where each rule includes a *pattern* (the set of packets belonging to the flow), a *priority* (that disambiguates overlapping rules), an *expiration* time, a list of *actions* (to apply to the packets), and *counters* (to measure the traffic). To process an incoming packet, the switch identifies the matching rule with the highest priority, updates the counters of the rule, and applies the actions. If no matching rule is found, the switch forwards the packet to the controller and awaits further instructions.

Most controllers are based on NOX [6], a network operating system for handling events and installing rules. NOX programmers must grapple with several difficult challenges:

Interactions between concurrent modules: Networks often perform multiple tasks, like routing, access control, and monitoring. These functions cannot be performed independently unless they operate on non-overlapping portions of the traffic (as in FlowVisor [14]), since a rule (un)installed by one module could undermine the proper functioning of other modules.

Low-level interface to switch hardware: OpenFlow provides a low-level interface to the switches. Applications must install rules that match on bits in the packet header. Since rules can have wildcards, a packet may match multiple overlapping rules with different priorities. In addition, a high-level policy may translate into multiple low-level rules (e.g., to match on ranges of values, or to support negation).

Two-tiered programming model: The controller only sees packets the switches do not know how to handle. This limits the controller's visibility into the underlying traffic—in essence, application execution is split between the controller and the switches. Applications must avoid installing rules that hide important information from the controller.

Frenetic alleviates these burdens by offering a *programming language* with a high-level filter algebra and a "program like you see every packet" abstraction. While the programmer uses these high-level abstractions, the *run-time system* generates low-level rules, ensures correct execution of multiple modules, and splits execution between the switch and controller to keep packets in the data plane whenever possible. To ease adoption of Frenetic, our language is simply a set of Python libraries. While this paper primarily focuses on the design of the Frenetic language, we also discuss our prototype implementation of the run-time system.

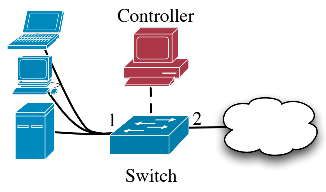

Figure 1: Simple Network Topology

2. OPENFLOW/NOX DIFFICULTIES

This section presents three examples that highlight the difficulties of writing programs for the OpenFlow/NOX platform. We have elided a few details of the platform which are not needed to understand the examples (a complete description is available in the OpenFlow specification [12]).

Let us warm up with a simple program that implements a repeater. We assume that the network has the topology depicted in Figure 1, where a single switch connects a pool of internal hosts on port 1 to a wide-area network on port 2. To implement a repeater, we simply install rules on the switch that forward traffic in both directions.

```
def repeater(switch):
  p1 = {IN_PORT:1}
  p2 = {IN_PORT:2}
  a1 = [output(2)]
  a2 = [output(1)]
  install(switch, p1, a1, DEFAULT)
  install(switch, p2, a2, DEFAULT)
```

The variables `p1` and `p2` are bound to patterns that describe sets of packets while `a1` and `a2` are bound to actions. The `install` function sends a message to instruct the `switch` to apply the actions to packets matching the given patterns at priority level `DEFAULT`. Upon receiving this message, the switch installs a rule in its flow table and begins using this rule to process traffic.

2.1 Interactions Between Concurrent Modules

The first difficulty of writing programs in OpenFlow/NOX is that programs do not compose. Suppose that we want to extend the repeater to monitor the total amount of incoming web traffic. The usual way to implement monitoring in NOX is to install separate rules that handle the traffic that needs to be monitored, and periodically poll the byte and packet counters for those rules to collect the necessary data.[1] The following program uses this strategy to monitor incoming web traffic, printing the total number of bytes every 30 seconds. The `monitor` function installs a rule that matches all incoming packets with TCP source port 80 and issues a query for the counters associated with that rule. Upon receiving the response from the switch, the NOX run-time system invokes `stats_in`, which prints the current byte count to the console, sleeps for 30 seconds, and issues the next query.

[1]Another way is to send every packet to the controller and aggregate the data there. However, this strategy does not scale and increases the latency of processing packets by orders of magnitude.

```
def monitor(switch):
  p = {IN_PORT:2,TP_SRC:80}
  install(switch, p, [], DEFAULT)
  query_stats(switch, pattern)
def stats_in(switch, pattern, stats):
  print stats['bytes']
  sleep(30)
  query_stats(switch, pattern)
```

We would like to compose this program with the repeater to obtain a program that both forwards packets and monitors traffic. Unfortunately, naively composing the programs does *not* behave as expected due to low-level interactions between the rules installed by each program. Because their patterns overlap, when an incoming packet with TCP source port 80 arrives, the switch is free to process the packet using the rule installed by `repeater` or the one installed by `monitor_web`. But either choice leads to incorrect behavior: the `repeater` rule does not update the counters used by the monitoring program, and the `monitor` rule breaks the repeater program as it drops the packet (its list of actions is empty).

To obtain the desired behavior, we have to manually combine the forwarding logic from the first program with the monitoring policy from the second.

```
def repeater_monitor(switch):
  p1 = {IN_PORT:1}
  p2 = {IN_PORT:2}
  p2web = {IN_PORT:2,TP_SRC:80}
  a1 = [output(2)]
  a2 = [output(1)]
  install(switch, p1, a1, DEFAULT)
  install(switch, p2, a2, DEFAULT)
  install(switch, p2web, a2, HIGH)
  query_stats(switch, p2web)
```

Note that performing this combination is non-trivial: the `p2web` rule needs to include the `output(1)` action from the `repeater` program, and must be installed with `HIGH` priority to resolve the overlap with the `p2` rule.

In general, composing OpenFlow/NOX programs requires significant careful, manual effort on the part of the programmer to preserve the semantics of the original programs. This makes it nearly impossible to factor out common pieces of functionality into reusable libraries. It also prevents compositional reasoning about programs.

2.2 Low-Level Programming Interface

Another difficulty in OpenFlow/NOX is the low-level nature of the programming interface, which is derived from the features of the switch hardware rather than being designed for ease-of-use. This interface makes programs unnecessarily complicated, as they must specify low-level details that are irrelevant to the overall behavior of the program. Suppose that we want to extend the repeater further to monitor all incoming web traffic *except* for traffic to an internal server at address `10.0.0.99`. To do this, we need a way to "subtract" patterns, but the patterns supported by switches only express positive constraints. To simulate the difference between two patterns, we must install *two* rules on the switch, disambiguating overlaps using priorities.

```
def repeater_monitor_noserver(switch):
  p1 = {IN_PORT:1}
  p2 = {IN_PORT:2}
  p2web = {IN_PORT:2,TP_SRC:80}
  p2srv = {IN_PORT:2,NW_ADDR:10.0.0.99,TP_SRC:80}
  a1 = [output(2)]
  a2 = [output(1)]
  install(switch, p1, a1, DEFAULT)
  install(switch, p2, a2, DEFAULT)
  install(switch, p2web, a2, MEDIUM)
  install(switch, p2srv, a2, HIGH)
  query_stats(switch, p2web)
```

This program is similar to the previous one, but uses a separate rule to process web traffic destined for the internal server—`p2srv` matches packets going to the internal server, while `p2web` matches all other incoming web packets. The program installs `p2srv` at `HIGH` priority to ensure that the `p2web` rule only processes (and counts!) packets going to hosts other than the internal server.

Describing packets using the low-level patterns supported in OpenFlow/NOX is cumbersome and error-prone. It forces programmers to use multiple rules and priorities to encode patterns that could be easily expressed using natural operations such as negation, difference, and union. It adds unnecessary clutter to programs and further complicates reasoning about programs.

2.3 Two-Tiered System Architecture

Another challenge of programming in OpenFlow/NOX stems from the two-tiered system architecture—the controller program manages the network by installing and uninstalling switch-level rules. The extra level of indirection makes it more complicated to specify the correct processing of packets. Also, the programmer must specify the communication patterns between the controller and switch and deal with tricky issues such as coordinating asynchronous events. Consider extending the repeater to monitor the total amount of incoming traffic by host. Unlike the previous monitoring examples, we cannot install the monitoring rules in advance because we may not know the addresses of each host in the network *a priori*. Instead, the controller must dynamically install rules for the packets seen at run time.

```
def repeater_monitor_hosts(switch):
  p = {IN_PORT:1}
  a = [output(2)]
  install(switch, p, a, DEFAULT)
def packet_in(switch, inport, packet):
  if inport == 2:
     m = srcmac(packet)
     p = {IN_PORT:2,DL_SRC:m}
     a = [output(1)]
     install(switch, p, a, DEFAULT)
     query_stats(switch, p)
```

The `repeater_monitor_hosts` function installs a single rule that forwards outgoing traffic. Initially, incoming packets do not match any flow table entries, so the switch sends them up to the controller. The NOX run-time invokes the `packet_in` function which installs a rule for forwarding incoming packets with the same MAC address and issues a query for the counters associated with that rule. Note that the controller only sees one incoming packet for each host—the rule processes future traffic going to that host directly on the switch.

In essence, OpenFlow/NOX applications are implemented using *two* programs—one on the controller and another on the switch. While essential for efficiency, the two-tiered architecture makes reasoning about applications difficult because the behavior of each program depends on the other—e.g., installing/uninstalling rules changes which packets are sent up to the controller. In addition, the controller program must specify the communication patterns between the two programs and deal with subtle concurrency issues—e.g., if we were to extend the example to monitor traffic in both directions, the controller program would have to issue two queries, one for incoming traffic and another for outgoing traffic, and synchronize the resulting callbacks.

Although OpenFlow/NOX enables the management of networks using arbitrary general-purpose programs, its two-tiered architecture forces programmers to specify the asynchronous and event-driven interaction between the programs running on the controller and the switches in the network. In our experience, these details are a significant distraction and a frequent source of bugs.

3. FRENETIC

This section presents Frenetic, a new language for network programming that provides a number of high-level features addressing each of the issues with the OpenFlow/NOX programming model just described.

3.1 Unified Architecture

Frenetic is based on functional reactive programming (FRP), a model in which programs manipulate streams of values. FRP eliminates the need to write event-driven programs and leads naturally to a unified architecture where programs "see every packet" rather than processing traffic indirectly by manipulating switch-level rules.

To get a taste for FRP, let us reimplement the web monitoring program from the last section (we will extend this program with forwarding later in this section).

```
def monitor_sf():
  return(Filter(inport_p(2) & srcport_p(80)) |o|
         GroupByTime(30) |o|
         SumSizes())
def monitor():
  stats = Apply(Packets(), monitor_sf())
  print_stream(stats)
```

The first declaration defines a *stream function* `monitor_sf` that takes a stream of packets and produces a stream of integers. The stream function `Filter` discards all packets from the input stream that do not represent incoming web traffic. The `GroupByTime` stream function divides the stream of filtered packets into a stream of lists containing the packets in each 30-second window. `SumSizes` computes the total size of all packets in each list. The infix operator `|o|` denotes sequential composition of stream functions. The final result is a stream of integers that represent the amount of incoming web traffic every 30 seconds. The top-level `monitor` function applies `monitor_sf` to `Packets`, which is a stream containing all of the packets flowing through the network, and then uses `print_stream` to print the result to the console.

Note that unlike the OpenFlow/NOX program, which specifies the layout of the rules on the switch as well as the communication needed to retrieve the counters from the switch, Frenetic's unified architecture makes it possible to express this program as a simple, declarative query.

3.2 High-Level Patterns

Frenetic includes a rich pattern algebra which provides an easy way to describe sets of packets. Suppose that we want to change the monitoring program to exclude traffic to the internal server. In Frenetic we can simply take the difference between the pattern describing incoming web traffic and the one describing traffic to the internal web server.

```
def monitor_noserver_sf():
  p1 = inport_p(2) & srcport_p(80)
  p2 = dstip_p("10.0.0.99")
  return (Filter(diff_p(p1,p2)) |o|
          GroupByTime(30) |o|
          SumSizes())
```

The only change in this program compared to the previous version is the pattern passed to `Filter`. The `diff_p` operator computes the difference between patterns. Recall that crafting rules to implement this program in OpenFlow/NOX was challenging—we had to simulate the difference using two rules and priorities.

3.3 Compositional Semantics

Arguably the most important feature of Frenetic is support for composition. Suppose that we want to extend the monitoring program to behave like a repeater. In Frenetic, we specify the forwarding rules and register them with the run-time system.

```
rules = [Rule(inport_p(1), [output(2)]),
         Rule(inport_p(2), [output(1)])]
def repeater_monitor():
   register_static(rules)
   stats = Apply(Packets(), monitor_sf())
   print_stream(stats)
```

The `register_static` function takes a list of `Rule` objects, each containing a high-level pattern and a list of actions, and registers them as the forwarding policy in the Frenetic run-time system. Note that the monitoring portion of the program does not change. The run-time system ensures that there are no harmful interactions between the forwarding and monitoring components.

To illustrate the benefits of composition, let us carry the example a step further and extend it to monitor incoming traffic by host as well. Implementing this program in NOX would be difficult—we cannot run the two smaller programs side-by-side because the rules for monitoring web traffic overlap with the rules for monitoring traffic by host. Thus, we would need to rewrite both programs to ensure that the rules installed on the switch are compatible with both programs—e.g., installing two rules for each host, one for web traffic and another for all other traffic. This would work, but it would require a major effort from the programmer, who would need to understand the low-level implementations of both programs in full detail.

In contrast, the Frenetic program is simple. The following stream function monitors incoming traffic by host.

```
def host_monitor_sf():
  return (Filter(inport_p(2)) |o|
          Group(dstmac_g()) |o|
          RegroupByTime(60) |o|
          SumGroupSizes())
```

It uses `Filter` to discard the outgoing traffic, `Group` to aggregate the top-level stream of packets into a stream of pairs of source MACs and nested streams that contain all packets from that source, `RegroupByTime` to divide the nested streams into 60-second windows, and `SumGroupSizes` to add up the size of the packets in each window. When applied to the stream of packets, it yields a stream of pairs of MAC addresses and integers that represent the total amount of traffic to that host in the preceding 60-second window. The top-level program applies both stream functions to the stream of packets and registers the forwarding rules with the run-time. Despite the slightly different functionality and polling intervals of the two programs, Frenetic allows the programmer to easily compose them without any concerns about undesirable interactions or timing issues.

```
def repeater_monitor_hosts():
   register_static(rules)
   stats1 = Apply(Packets(),monitor_sf())
   stats2 = Apply(Packets(),host_monitor_sf())
   print_stream(Merge(stats1,stats2))
```

Support for composition is one of Frenetic's most important features. Raising the level of abstraction frees programmers from having to worry about low-level details and enables writing programs in a modular style. This represents a major advance over today's NOX, where programs must be written monolithically to avoid harmful interactions between the switch-level rules installed by different program pieces.

4. LEARNING SWITCH

So far, we have focused on small examples that illustrate the features of Frenetic. Our final example is a more substantial program that implements an Ethernet learning switch.

```
def learning_sf():
  return (Group(srcmac_g()) |o|
          Regroup(inport_r()) |o|
          UngroupFirst() |o|
          LoopPre({}, Lift(add_rule)) |o|
          Lift(complete_rules))
def learning():
  rules = Apply(Packet(),learning_sf())
  register_stream(rules)
```

It uses `Group` to aggregate the stream of packets by source MAC address and `Regroup` to split the resulting streams whenever traffic from a source MAC appears at a different switch port (i.e., because the host has moved). We are now left with a stream of streams where each substream contains packets that all share the same source MAC address and ingress switch port. `UngroupFirst` retrieves the first packet from each group, and `LoopPre` builds a dictionary that maps MAC addresses to forwarding rules (the helper `add_rule` inserts a rule into the dictionary). The last operator, `Lift`, converts an ordinary function to a stream function that operates on event streams. The `complete_rules` function extracts the list of rules from the dictionary and adds a catch-all rule that floods packets destined for unknown MAC addresses. The top-level `learning` function registers these rules in the Frenetic run-time. Note that unlike the previous examples, the rules are not static. The `register_stream` function takes a stream of lists of rules and registers them in the run-time.

5. IMPLEMENTATION

Frenetic facilitates describing network programs without having to specify unimportant low-level details concerning the underlying switch hardware. Of course, the need to deal with these details does not go away. The rubber meets the road in the implementation. We have implemented a complete prototype of Frenetic in Python. Figure 2 depicts its architecture, which consists of three pieces: an implementation of the FRP operators, a run-time system, and NOX. The use of NOX is not essential—we borrow its OpenFlow API but could also use a different back-end.

The FRP operators are implemented as a Python library that defines representations for streams and stream functions, as well as implementations of primitives such as `Filter`, `LoopPre`, `SumSizes`, etc. Unlike classic FRP implementations, which support both continuous streams called *behaviors* and discrete streams called *events*, Frenetic focuses exclusively on discrete streams. The pull-based strategy used in most FRP implementations is optimized for behaviors and so is not a good fit for Frenetic. Instead, we use a push-based strategy that propagates values from input to output streams.

Although Frenetic programs "see every packet", a naive implementation that processed every packet on the controller would not scale to networks of realistic size; it is necessary to develop optimizations that move packet processing off the controller and onto the switches. We have developed optimizations that capture some common idioms, but hope

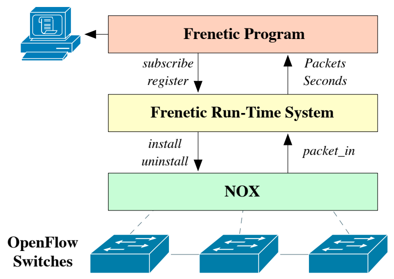

Figure 2: Frenetic Architecture

to discover additional optimizations that will allow Frenetic programs to perform as well as the best hand-written programs. Much like garbage collection, we believe that managing switch-level rules is a tedious task that is best handled in a run-time system.

The interface to our run-time system supports *subscribing* to streams of packets, headers, and statistics, and *registering* packet-forwarding rules. These functions allow the run-time system to determine which packets must go to the controller and which can be processed on the switch. They are designed to be fully compositional—programs can subscribe to multiple, overlapping streams of packets and register forwarding rules for subscribed packets without worrying about harmful low-level interactions. To connect programs to streams we transform programs, replacing groupings of FRP operators with calls to these functions. For example, `Apply(Packets(), Filter(p) |o| sf)`, where `sf` is an arbitrary stream function, becomes `Apply(subscribe(p), sf)`. We currently rewrite programs by hand but are developing an optimizer to do it automatically.

The core of the run-time system is the back-end, which manages the installation and uninstallation of rules as well as all communications between the switches and controller. It generates rules using a simple *reactive* strategy. At the start of the execution of a program, no rules are installed on switches, so all packets are sent up to the controller and passed to the `packet_in` function. Upon receiving a packet, the run-time traverses the lists of subscribers and forwarding rules, propagating the packet to any subscribers and determining the actions specified in the forwarding rule. If there are no subscribers for the packet, the system installs a *microflow* rule on the switch—i.e. a rule whose pattern matches the header fields of the packet exactly—that processes future packets with the same header fields using the packet-forwarding policy registered in the run-time. This rule can be used until the packet-forwarding policy changes. Subscribers to streams of statistics are handled similarly, using the counters associated with the microflow rules.

6. RELATED WORK

Frenetic's stream functions are modeled after functional reactive languages such as Yampa and others [11, 5, 13, 10]. Its push-based implementation is based on FrTime [3] and is similar to self-adjusting computation [2]. The key difference between Frenetic and these languages is in our run-time system, which uses the capabilities of switches to implement the semantics of the operators in the language.

Several other research projects draw on ideas from programming languages to develop new languages for programming networks. The most similar language to Frenetic is Nettle [15], which is also based on FRP. Nettle supports network-wide control and domain-specific languages for different tasks, but lacks Frenetic's support for composition of modules affecting overlapping portions of flowspace. Another related language is NDLog, which has been used to specify and implement routing protocols, overlay networks, and services such as distributed hash tables [8]. NDLog differs from Frenetic in that it is designed for distributed systems (rather than a centralized controller) and is based on logic programming. Also based on logic programming, FML focuses on specifying policies such as access control in OpenFlow networks [7]. Finally, the SNAC OpenFlow controller [1] provides a GUI for specifying access control policies using high-level patterns similar to the ones we have developed for Frenetic. However, SNAC provides a much less general programming environment than Frenetic.

One of the main challenges in the implementation of Frenetic involves splitting work between the (powerful but slow) controller and the (fast but limited) switches. The same idea was used in Gigascope [4], a stream database for monitoring networks. Unlike Frenetic, it only supports querying traffic and cannot be used to control the network itself.

7. CONCLUSIONS AND FUTURE WORK

This paper describes the design and implementation of Frenetic, a new language for programming OpenFlow networks. Frenetic addresses some serious problems with the OpenFlow/NOX platform by providing a high-level, compositional, and unified programming model. It includes a collection of operators for transforming streams of network traffic, and a run-time system that handles all of the details related to installing and uninstalling switch-level rules.

We are currently working to extend Frenetic in several directions. We are developing applications for a variety of tasks including load balancing, authentication and access control, and a framework inspired by FlowVisor [14] for ensuring isolation between programs. We are developing a front-end and an optimizer that will transform programs into a form that can be efficiently implemented on the run-time system. Finally, we are exploring a proactive strategy that generates rules from the registered subscribers and forwarding rules eagerly. We plan to compare the tradeoffs between different rule generation strategies empirically.

Acknowledgments. We wish to thank Minlan Yu for many helpful discussions. Our work is supported by ONR grant N00014-09-1-0770 *Networks Opposing Botnets*. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR.

8. REFERENCES

[1] The SNAC OpenFlow controller. See `http://snacsource.org/`, 2010.
[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.
[3] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, March 2006.
[4] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *ACM SIGMOD*, pages 647–651, 2003.
[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 163–173, June 1997.
[6] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3):105–110, 2008.
[7] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *WREN*, pages 1–10, 2009.
[8] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *CACM*, 52(11):87–95, 2009.
[9] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.
[10] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA*, pages 1–20, 2009.
[11] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Haskell Workshop*, pages 51–64, October 2002.
[12] The OpenFlow Switch Consortium. *OpenFlow Switch Specification*, December 2009.
[13] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL*, January 1999.
[14] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Martin Casado, Guido Appenzeller, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *OSDI*, October 2010.
[15] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *Symposium on Practical Aspects of Declarative Languages (PADL)*, January 2011. To appear.