A Compiler and Run-time System for Network Programming Languages

Christopher Monsanto
Princeton University

Nate Foster
Cornell University

Rob Harrison *
US Military Academy

David Walker
Princeton University

Abstract

Software-defined networks (SDNs) are a new kind of network architecture in which a controller machine manages a distributed collection of switches by instructing them to install or uninstall packet-forwarding rules and report traffic statistics. The recently formed Open Networking Consortium, whose members include Google, Facebook, Microsoft, Verizon, and others, hopes to use this architecture to transform the way that enterprise and data center networks are implemented.

In this paper, we define a high-level, declarative language, called *NetCore*, for expressing packet-forwarding policies on SDNs. NetCore is expressive, compositional, and has a formal semantics. To ensure that a majority of packets are processed efficiently on switches—instead of on the controller—we present new compilation algorithms for NetCore and couple them with a new run-time system that issues rule installation commands and traffic-statistics queries to switches. Together, the compiler and run-time system generate efficient rules whenever possible and outperform the simple, manual techniques commonly used to program SDNs today. In addition, the algorithms we develop are generic, assuming only that the packet-matching capabilities available on switches satisfy some basic algebraic laws.

Overall, this paper delivers a new design for a high-level network programming language; an improved set of compiler algorithms; a new run-time system for SDN architectures; the first formal semantics and proofs of correctness in this domain; and an implementation and evaluation that demonstrates the performance benefits over traditional manual techniques.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design

Keywords Software-defined Networking, OpenFlow, Frenetic, Network programming languages, Domain specific languages

* The views expressed in this paper are those of the authors and do not reflect the official policy or position of the US Military Academy, the Department of the Army, the Department of Defense, or the US Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01. . . $10.00

1. Introduction

A network is a collection of connected devices that route traffic from one place to another. Networks are pervasive: they connect students and faculty on university campuses, they send packets between a variety of mobile devices in modern households, they route search requests and shopping orders through data centers, they tunnel between corporate networks in San Francisco and Helsinki, and they connect the steering wheel to the drive train in your car. Naturally, these networks have different purposes, properties, and requirements. To service these requirements, companies like Cisco, Juniper, and others manufacture a variety of devices including routers (which forward packets based on IP addresses), switches (which forward packets based on MAC addresses), NAT boxes (which translate addresses within a network), firewalls (which squelch forbidden or unwanted traffic), and load balancers (which distribute work among servers), to name a few.

While each of these devices behaves differently, internally they are all built on top of a *data plane* that buffers, forwards, drops, tags, rate limits, and collects statistics about packets at high speed. More complicated devices like routers also have a *control plane* that run algorithms for tracking the topology of the network and computing routes through it. Using statistics gathered from the data plane and the results computed using the device's specialized algorithms, the control plane installs or uninstalls forwarding rules in the data plane. The data plane is built out of fast, special-purpose hardware, capable of forwarding packets at the rate at which they arrive, while the control plane is typically implemented in software.

Remarkably, however, traditional networks appear to be on the verge of a major upheaval. On March 11th, 2011, Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo!, owners of some of the largest networks in the world, announced the formation of the Open Networking Foundation [19]. The foundation's proposal is extraordinarily simple: *eliminate the control plane from network devices*. Instead of baking specific control software into each device, the foundation proposes a standard protocol that a separate, general-purpose machine called a *controller* can use to program and query the data planes of many cooperating devices. By moving the control plane from special-purpose devices onto stock machines, companies like Google will be able to buy cheap, commodity switches, and write controller programs to customize and optimize their networks however they choose.

Networks built on this new architecture, which arose from earlier work on Ethane [4] and 4D [10], are now commonly referred to as *Software-Defined Networks* (SDNs). Already, several commercial switch vendors support OpenFlow [17], a concrete realization of the switch-controller protocol required for implementing SDNs, and researchers have used OpenFlow to develop new network-wide algorithms for server load-balancing, data center routing, energy-efficient network management, virtualization, fine-grained access

control, traffic monitoring, fault tolerance, denial of service detection, host mobility, and many others [8, 12–14, 18, 26].

Now the obvious question is: Why should programming language researchers, and the POPL community in particular, care about these developments? The answer is clear: Some of our most important infrastructure—our networks—will soon be running *an entirely new kind of program*. Using our experience, principles, tools, and algorithms, our community has a unique opportunity to define the languages these programs will be written in and the infrastructure used to implement them. We can have major impact, and help make future networks easier to program, more secure, more reliable, and more efficient.

As a step toward carrying out this agenda, we propose a high-level language called NetCore, the *Network Core Programming Language*, for expressing packet-forwarding policies. NetCore has an intuitive syntax based on familiar set-theoretic operations that allows programmers to construct (and reason about!) rich policies in a natural way. NetCore's primitives for classifying packets include exact-match bit patterns and arbitrary wildcard patterns. It also supports using arbitrary functions to analyze packets and historical traffic patterns. This feature makes it possible to describe complicated, dynamic policies such as authentication and load balancing in a natural way, using ordinary functional programs.

Unfortunately, compiling these rich policies is challenging. On the one hand, the controller machine has the computational power to evaluate arbitrary policies, but the switches do not: they can only implement simple kinds of bit matching rules. On the other hand, directing a packet to the controller for processing incurs orders of magnitude more latency than processing it on a switch. Hence, despite the limited computational power of the switches, it is critical to find ways for them to perform most packet processing.

The NetCore compiler and run-time system surmounts this challenge by analyzing programs and automatically dividing them into two pieces: one that runs on the switches and another that runs on the controller. Moreover, this division of labour does not occur once at compile time; it occurs dynamically and repeatedly. Intuitively, when a packet cannot be handled by a switch, it is redirected to the controller. The controller partially evaluates the packet with respect to the current network policy and dynamically generates new switch-level rules that handle said packet as well as others like it. The new rules are subsequently sent to the switches so similar packets arriving in the future are handled in the network fast path. Over time, more and more rules are added to the switches and less and less traffic is diverted to the controller. We call this iterative strategy *reactive specialization*.

Our strategy is inspired by the idiom commonly used for SDN applications today [12, 13, 27], in which an event-driven program manually installs a rule to handle future traffic every time a packet is diverted to the controller. However, many programs written manually in this style use inefficient exact-match rules rather than wildcard rules, because reasoning about the semantics of overlapping wildcards quickly becomes very complicated—too complicated to do by hand. Hence, our strategy improves on past work by (1) providing high-level abstractions that obviate the need for programmers to deal with the low-level details of individual switches, (2) synthesizing efficient forwarding rules that exploit the capabilities of modern switches including wildcard rules implemented by ternary content-addressable memories (TCAMs), (3) automating the process of dynamically unfolding packet-processing rules on to switches instead of requiring that programmers craft tricky, low-level, event-based programs manually.

To summarize: the central contribution of this paper is a framework for implementing a canonical, high-level network programming language correctly and efficiently. More specifically:

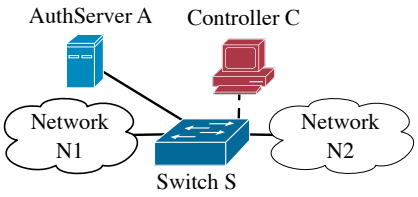


Figure 1. Example topology.

- We define the syntax and semantics for NetCore (Section 3) and model the interaction between the NetCore run-time system and the network in a process calculus style (Section 4). This is the first formal analysis of how a controller platform interacts with switches.
- We develop novel algorithms for compiling network programs and managing controller-switch interactions at run time, including *classifier generation* and *reactive specialization* (Section 5).
- We prove key correctness theorems (Section 6), establishing simulation relations between our low-level, distributed implementation strategy and our high-level NetCore semantics. We also prove an important *quiescence* theorem showing that our implementation successfully relocates computation from the controller onto switches.
- We describe a prototype implementation and an evaluation on some simple benchmarks demonstrating the practical utility of our framework (Section 7).

NetCore arose out of our previous work on Frenetic [9], another high-level network programming language. Frenetic has three main pieces: (1) an SQL-like query language for reading network state, (2) a language for specifying packet-forwarding policies and, (3) a functional reactive "glue" language that processes the results of queries and generates streams of forwarding policies for the network. NetCore replaces Frenetic's language for expressing forwarding policies with a significantly more powerful language that supports processing packets using *arbitrary functions*. In addition, NetCore also contains a minimalist "query language," as its predicates can analyze traffic history.

The main contribution of this paper relative to earlier work on Frenetic is the design of new algorithms for compiling these rich policies and for managing the controller-switch interactions that arise as compiled policies are executed in a network. These algorithms handle NetCore's new policy language, and the core elements of Frenetic's old policy language even better. In particular, the NetCore compiler generates efficient switch classifiers by (1) using *wildcard rules* that process more packets on switches instead of simple *exact-match rules*, and (2) generating rules *proactively* (*i.e.*, in advance of when they are needed), again to process more packets on switches, instead of strictly *reactively* (*i.e.*, on demand). Finally, NetCore has a formal semantics and correctness proofs for its core algorithms, whereas Frenetic had none.

2. NetCore Overview

This section presents additional background on SDNs and NetCore, using examples to illustrate the main ideas. For concreteness, we focus on the OpenFlow SDN architecture [20], but we elide and take liberty with certain inessential details. Our compiler does not assume the specifics of the current OpenFlow platform.

OpenFlow overview. OpenFlow is based on a two-tiered architecture in which a controller manages a collection of subordinate switches. Figure 1 depicts a simple topology with a controller C

managing a single switch S. Packets may either be processed on switches or on the controller, but processing a packet on the controller increases its latency by several orders of magnitude. Hence, to ensure good performance, the controller typically installs a *classifier* consisting of a set of packet-forwarding *rules* on each switch.

Each forwarding rule has a *pattern* that identifies a set of packets, an *action* that specifies how packets matching the pattern should be processed, *counters* that keep track of the number and size of all packets processed using the rule, and an integer *priority*. When a packet arrives at a switch, it is processed in three steps: First, the switch selects a rule whose pattern matches the packet. If it has no matching rules, then it drops the packet, and if it has multiple matching rules, then it picks the one with the highest priority. Second, the switch updates the counters associated with the rule. Finally, the switch applies the action listed in the rule to the packet. In this paper, we are concerned with two kinds of actions: (1) a forwarding action $\{l_1, \ldots, l_k\}$, which forwards the packet to a set of (usually one, sometimes zero, rarely more than one) adjacent network locations l_i, where each l_i may be the name of another switch, network, or host,[1] and (2) a controller action Ω, which forwards the packet to the controller for processing.

NetCore: A simple static forwarding policy. NetCore is a declarative language for specifying high-level packet-forwarding policies. The NetCore compiler and run-time system handle the details of translating these policies to switch-level rules and issuing commands to install the generated rules on switches.

The simplest NetCore policies are specified using a *predicate* e that matches some set of packets and a set S of locations to which those packets should be forwarded. We write these policies $e \to S$. The simplest predicates match bits in a particular packet header field. For example, the predicate `SrcAddr:10.0.0.0/8` specifies that the first octet of the packet's source address must be `10` (using the standard notation for expressing IP prefix patterns). More complex predicates are built by taking the union ($\cup$), intersection ($\cap$), negation ($\neg$), or difference ($\setminus$) of simpler predicates. Analogous set-theoretic operations may be used to compose more complex policies from simpler policies. As an example, consider the following policy.

```
SrcAddr:10.0.0.0/8 ∖ (SrcAddr:10.0.0.1 ∪ DstPort:80)
→ {Switch 1}
```

It states that packets from sources in subnet `10.0.0.0/8` should be forwarded to switch `1`, except for packets coming from `10.0.0.1` or going to a destination on port `80`.

The first challenge in compiling a high-level language such as NetCore to a low-level SDN framework such as OpenFlow arises from the relative lack of expressiveness in the switch packet-matching primitives. For instance, because switches cannot express the difference of two patterns in a single rule, this policy needs to be implemented using three rules installed in a particular prioritized order: one that drops packets from `10.0.0.1`, another that drops all packets going to port `80`, and a final rule that forwards all remaining packets from `10.0.0.0/8` to `Switch 1`. The following switch-level classifier implements this policy. We write these classifiers with the highest priority rule first. Switch-level patterns are on the left, actions are on the right, and a colon separates the two.

```
SrcAddr:10.0.0.1   : {}
DstPort:80         : {}
SrcAddr:10.0.0.0/8 : {Switch 1}
```

Next consider a similar high-level policy to the first:

```
SrcAddr:10.2.0.0/16 ∖ (SrcAddr:10.2.0.1 ∪ DstPort:22)
→ {Switch 2}
```

We can generate a classifier for this policy in the same way:

```
SrcAddr:10.2.0.1    : {}
DstPort:22          : {}
SrcAddr:10.2.0.0/16 : {Switch 2}
```

Now suppose that we want to generate a classifier that implements the union of the two policies. We cannot combine the classifiers in a simple way (*e.g.*, by concatenating or interleaving them) because the rules interact with each other. For example, if we were to simply concatenate the two lists of rules, the rule that drops packets to port `80` would incorrectly shadow the forwarding rule for traffic from `10.2.0.0/16`. Instead, we need to perform a much more complicated translation that produces the following classifier:

```
SrcAddr:10.2.0.1,    DstPort:80 : {},
SrcAddr:10.2.0.0/16, DstPort:80 : {Switch 2}
SrcAddr:10.2.0.1                : {Switch 1}
SrcAddr:10.2.0.0/16, DstPort:22 : {Switch 1}
SrcAddr:10.2.0.0/16             : {Switch 1,Switch 2}
SrcAddr:10.0.0.1                : {}
SrcAddr:10.0.0.0/8,  DstPort:80 : {}
SrcAddr:10.0.0.0/8              : {Switch 1}
```

Dealing with these complexities often leads SDN programmers to use *exact-match* rules—*i.e.*, rules that fully specify every bit in every single header field. Exact-match rules, for instance, do not use wildcard patterns that match many values for a single header field, such as `10.0.0.0/8`, nor do they leave certain header fields completely unconstrained. Our first implementation of Frenetic [9] used exact-match rules exclusively because such rules were far easier for its run-time system to reason about, particularly when it came to composing multiple user policies.

This paper presents new, general-purpose algorithms for synthesizing low-level switch classifiers that use wildcard rules to the extent possible. These new algorithms result in far more efficient system than the one we built in earlier work: in Frenetic's original exact-match architecture, many more packets wound up being sent to the controller (suffering orders of magnitude increase in latency) and many more rules had to be sent to switches. The results of our experiments, presented in Section 7, highlight the magnitude of these differences.

NetCore: Richer predicates and dynamic policies. The policies presented in the previous section were relatively simple—they did nothing besides match bits in header fields and forward packets accordingly. Such static policies can be expressed in Frenetic's simple policy language, though they are not be implemented nearly as efficiently as in the NetCore system. However, many applications demand dynamic policies whose forwarding behavior depends on complex functions of traffic history and other information. And these richer policies cannot be implemented by simply analyzing bits in header fields.

As an example, suppose we want to build a security application that implements in-network authentication for the topology shown in Figure 1. The network N_1 contains a collection of internal hosts, N_2 represents the upstream connection to the Internet, A is the server that handles authentication for hosts in N_1, and all three elements are connected to each other by the switch S. Informally, we want the network to perform routing and access control according to the following policy: Forward packets from unauthenticated hosts in N_1 to A, from authenticated hosts in N_1 to their intended destination in N_2, and from A and N_2 back to N_1 (although not from N_2 to A). This policy can be described succinctly in NetCore as follows.

[1] On real OpenFlow switches, locations are actually integers corresponding to physical ports on the switch; in this paper we model them symbolically.

```
  (InPort:Network 1 ∩ inspect ps auth → {Network 2})
∪(InPort:Network 1 ∩ ¬(inspect ps auth) → {Server A})
∪(InPort:Server  A ∪ InPort:Network 2 → {Network 1})
```

where

```
ps              = InPort:Server A
auth (Σ,s,p)    = any (isAddr p) Σ
isAddr p (_,p') = p.SrcAddr == p'.DstAddr
```

This policy uses an *inspector predicate* to classify traffic from N_1 as authenticated or unauthenticated. An inspector predicate `inspect` e f has two arguments: a filter predicate e over the network traffic history and an (almost) arbitrary boolean-valued function f. The filter predicate generates a *controller state* Σ, which is a collection of traffic statistics, represented abstractly as a multiset of switch-packet pairs (the switch being the place where the packet was processed). The boolean-valued function f receives the controller state as one its arguments and may analyze it as part of its decision-making process.

In the example above, the filter predicate `ps` selects all traffic coming from the authentication server. In this idealized example, we will treat an entity sending a packet `p` as authenticated if the authentication server has ever sent it a packet at any point in the past. The function `auth` takes three arguments: the controller state Σ, the switch `s` that should be handling the packet, and the packet `p` to which the policy applies. Here, the `auth` function tests whether the `SrcAddr` field of the packet `p` being processed is equal to the `DstAddr` of `any` other packet `p'` in the filtered traffic history (and because there is only one switch in this example, `auth` ignores its `s` argument). In other words, it tests whether the authentication server has sent a packet to that sender in the past. If it has, the inspector predicate is satisfied; if not, it is not satisfied. The `auth` function performs these tests using the auxiliary functions `any`, a built-in function that tests whether a boolean function is true of any element of a multiset, and `isAddr`, a user-defined function that tests whether one packet's `DstAddr` is equal to another packet's `SrcAddr`. This inspector is combined with the other set-theoretic operators to implement the overall packet-forwarding policy.

Policies that use inspectors are easy to write because forwarding decisions can be expressed using arbitrary functional programs. These programs can query past traffic history or look up facts they need such as authentication status in a database. On the other hand, these programs never have to manage the low-level details of generating or installing switch-level rules—the run-time system does that tedious, error-prone work for the programmer. Of course, this expressiveness presents an extreme challenge for the compiler and run-time system. While it would be easy to evaluate the results of such policies by sending all packets to the controller, doing so would be totally impractical. We must find a way to implement the policy while processing the majority of traffic on switches.

Our implementation strategy for such policies proceeds as follows. First, we compile the parts of the policy that do not involve inspectors as effectively as we can: The system generates normal forwarding rules when it can, and rules that send packets to the controller otherwise. Next, whenever a packet that cannot be handled by a switch arrives at the controller, the run-time system evaluates the packet against the current policy, which includes inspectors, producing a set of forwarding actions. There are two possibilities:

1. The policy with respect to this packet (and similar ones) is *invariant*. In other words, every subsequent time the system evaluates the policy against this packet, it will return the same set of forwarding actions.
2. The policy with respect to this packet (and similar ones) is *volatile*. In other words, the set of forwarding actions to be applied to this policy may change in the future.

In the first case, the system can install rules on the switch that associate packets similar to the one just processed with the set of forwarding actions just computed. Because the set of computed actions will never change, installing such rules on switches preserves the semantics of the policy. In the second case, the system can not install rules on the switch—the next packet might be forwarded differently, so the system will have to reevaluate it on the controller. In our example, once a host has been authenticated, it stays authenticated—once the `auth` function evaluates to true it will continue to do so, and is therefore *invariant*. Since inferring invariance automatically from an arbitrary program is a difficult problem, we currently ask NetCore programmers to supply invariance information to the compiler in the form of an auxiliary hand-written function. In this simple case, writing the invariance function `auth_inv` is trivial— it is true whenever `auth` is true:

```
auth_inv (Σ,s,p) = auth (Σ,s,p)
```

To effectively generate rules, even in the presence of inspector predicates, the run-time system must be able to determine when inspector returns the same results on one packet as it does on another—*i.e.*, it must be able to calculate the *similar packets* referred to above. Observe that an inspector always returns the same results on two different packets if those packets agree on all header fields that the inspector function examines. Conversely, if the inspector does not examine a particular header field, the value of that field does not affect its result. Hence, when generating a policy after evaluating it against a single packet, the run-time can substitute wildcards for all header fields that the policy does not inspect. Though it is likely possible to infer the set of headers any inspector function examines (at least conservatively), our current implementation assumes that programmers supply this information explicitly.

Overall, these techniques—(1) run-time evaluation of policies against particular packets on the controller, (2) invariance, and (3) specification of header information—collaborate to turn the difficult problem of evaluating policies containing arbitrary functions back in to the simpler problem of compiling static forwarding policies efficiently. We call these techniques *reactive specialization*.

3. A Core Calculus for Network Programming

This section defines the syntax and semantics of NetCore, a core calculus for high-level network programming. The calculus has two major components: *predicates*, which describe sets of packets, and *policies*, which specify where to forward those packets. Figure 2 presents the syntax of these constructs as well as various network values such as headers and packets.

Notation. Throughout this paper, whenever we define syntax, as in the grammar for packets, we will use the grammar non-terminal (p) as a metavariable ranging over the objects being defined, the capitalized version of the non-terminal (P) as a metavariable ranging over sets or multisets of such objects, and vector notation ($\vec{p}$) for sequences of objects.

We describe finite sets using the notation $\{x_1, \ldots, x_k\}$ and combine sets using operations $\cup$, $\cap$, $\neg$, and $\setminus$ (union, intersection, complement, and difference respectively). Typically, we give definitions for intersection and complement and leave union ($S_1 \cup S_2 = \neg(\neg S_1 \cap \neg S_2)$) and difference ($S_1 \setminus S_2 = S_1 \cap \neg S_2$) as derived forms. We also overload $\neg$ and use it to negate booleans; its meaning will be clear from context. We write multisets using the notation $\{\!|x_1, \ldots, x_n|\!\}$ and combine multisets using multiset union $M_1 \uplus M_2$. We write finite maps using the notation $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ and lookup elements of a finite map m using function application $m(x_i)$.

Network values. For simplicity, we only model a single kind of network entity to forward to, switches s. Packets p are the basic

Network Values

$$
\begin{array}{rrcl}
\text{Switch} & s & & \\
\text{Header} & h & & \\
\text{Switch Set} & S & ::= & \{s_1, \ldots, s_n\} \\
\text{Header Set} & H & ::= & \{h_1, \ldots, h_n\} \\
\text{Bit} & b & ::= & 1 \mid 0 \\
\text{Packet} & p & ::= & \{h_1 \mapsto \vec{b}_1, \ldots, h_n \mapsto \vec{b}_n\} \\
\text{State} & \Sigma & ::= & \{\!|(s_1, p_1), \ldots, (s_n, p_n)|\!\}
\end{array}
$$

Language

$$
\begin{array}{rrcl}
\text{Snapshot} & x & ::= & (\Sigma, s, p) \\
\text{Wildcard} & w & ::= & 1 \mid 0 \mid ? \\
\text{Inspector} & f & \in & \mathsf{State}[H_1] \times \mathsf{Switch} \times \mathsf{Packet}[H_2] \to \mathsf{Bool} \\
\text{Predicate} & e & ::= & h : \vec{w} \mid \mathsf{switch}\ s \mid \mathsf{inspect}\ e\ f \mid e_1 \cap e_2 \mid \neg e \\
\text{Policy} & \tau & ::= & e \to S \mid \tau_1 \cap \tau_2 \mid \neg\tau
\end{array}
$$

Figure 2. NetCore syntax.

$$
\begin{aligned}
[\![h : \vec{w}]\!] &= \{(\Sigma, s, p) \mid p(h) \text{ matches } \vec{w}\} \\
[\![\mathsf{switch}\ s']\!] &= \{(\Sigma, s, p) \mid s' = s\} \\
[\![\mathsf{inspect}\ e\ f]\!] &= \{(\Sigma, s, p) \mid f(\Sigma', s, p)\} \\
&\quad \text{where } \Sigma' = \{(s', p') \mid (s', p') \in \Sigma \text{ and } (\Sigma, s', p') \in [\![e]\!]\} \\
[\![e_1 \cap e_2]\!] &= [\![e_1]\!] \cap [\![e_2]\!] \\
[\![\neg e]\!] &= \neg [\![e]\!]
\end{aligned}
$$

$$
[\![\tau]\!](x) = S
$$

$$
\begin{aligned}
[\![e \to S]\!](x) &= \begin{cases} S & \text{if } x \in [\![e]\!] \\ \varnothing & \text{otherwise} \end{cases} \\
[\![\tau_1 \cap \tau_2]\!](x) &= [\![\tau_1]\!](x) \cap [\![\tau_2]\!](x) \\
[\![\neg\tau]\!](x) &= \neg [\![\tau]\!](x)
\end{aligned}
$$

Figure 3. NetCore semantics.

values processed by programs, which we represent as a finite map from headers h to bitstrings $\vec{b}$. We write $p(h)$ for the bitstring associated with the header h in p. We assume all fields have fixed, finite length and therefore the set of complete packets is finite. The *controller state* (Σ) accumulates information about packets that arrive at each switch. We represent controller state as a multiset of switch-packet pairs.

Predicates. Informally, predicates select sets of packets that are of interest in some forwarding policy. Formally, a predicate e denotes a set of *snapshots* x comprising a controller state Σ, a switch s, and a packet p located at s. The state component is essential for modeling predicates that depend upon historical traffic patterns, such as the past load on particular links or packets sent and received from various locations. Figure 3 defines the semantics of predicates. We say that a snapshot x *matches* a predicate e when it belongs to the denotation of e. We sometimes say that a packet p *matches* e, leaving the state and the switch implicit because they are irrelevant or uninteresting. We also say that a bitstring $\vec{b}$ *matches* a wildcard $\vec{w}$ whenever the corresponding bits match. For example, 1111 and 0011 both match the wildcard ??11.

Basic predicates have the form $h : \vec{w}$. A packet p matches $h : \vec{w}$ if $p(h)$, (*i.e.*, the h header of p) matches $\vec{w}$. For example, the predicate `DstPort:1010000` matches all packets with `DstPort` header field equal to 80 (as `1010000` is 80 in binary). Another basic predicate, switch s, matches all packets (in any state) sent to s. More complex predicates are built up from simpler ones using the intersection and complement operators. Additional building blocks such as True, False, $e_1 \cup e_2$, or $e_1 \setminus e_2$ can be implemented as derived forms.

The most interesting component of the language is the *inspector* predicate, inspect e f. The first component of an inspector is a filter predicate e that selects switch-packet pairs matching e from the current state Σ, creating a refined state Σ'. In other words, e acts as a query over the network traffic history. The second component, f, is an (almost) arbitrary Boolean-valued function over Σ' and the switch-packet pair (`s` and `p`) in question. The authentication example defined in the previous section used an inspector. Another example is (`inspect filterWeb cond`) where

```
filterWeb     = DstPort:1010000
cond (Σ,s,p) = cardinality Σ < 10000 ||
                 p.SrcAddr == 10.0.0.1
```

Here, `cardinality` is a function that counts the number of elements in a multiset. This inspector extracts all web traffic (`DstPort` is `1010000`) from the current state. The inspector is satisfied if the total number web packets sent is less than `10000` or the packet `p` comes from a particular sender (`SrcAddr` is `10.0.0.1`).

To make compilation tractable, two additional pieces of information are associated with inspector functions f. The first piece of information comes from the sets of headers mentioned in its indexed type,

$$\mathsf{State}[H_1] \times \mathsf{Switch} \times \mathsf{Packet}[H_2] \to \mathsf{Bool}\,.$$

Such a type restricts f to only examine headers H_1 of packets in the state and headers H_2 in the packet. For instance, the function `cond` above may be assigned a type wherein H_1 is the empty set (as summing packet counts requires looking at no headers) and H_2 is $\{\mathsf{SrcAddr}\}$ (as `cond` only examines the SrcAddr field of its packet argument). The second piece of information associated with f comes from its *invariance oracle*. A function f is *invariant on* (Σ, s, p), written $\mathsf{invariant}\,((\Sigma, s, p), f)$, if for all Σ', we have

$$f(\Sigma \uplus \Sigma', s, p) = f(\Sigma, s, p)\,.$$

Intuitively, a function is invariant on a state when its result does not change, no matter what additional information is added to it. Again, as an example, the `cond` function above is invariant on all snapshots involving packets from `10.0.0.1` as well as all snapshots where `cardinality` $\Sigma \geq$ `10000`—once the total volume of web traffic has crossed the threshold, the function always returns true. In our implementation, the programmer writes invariance oracles by hand as simple Haskell functions.

Together, the header sets in the inspector types, and the invariance oracle, allow the compiler to generate effective switch-level rules even though the inspector function itself cannot be analyzed. However, the language of predicates does have one significant limitation: it depends upon *permanent* invariance of predicates. There are predicates that are invariant for a long time, and hence could have rules installed on switches for that time, but are not permanently invariant. We believe our framework can be extended to handle such semi-permanent invariance properties, having the compiler uninstall rules at the end of a time period, or in response to a network event, but defer an investigation of this topic to future work.

$$\text{Synchronous Machine} \quad M_{\text{sync}} ::= (\tau, \Sigma, T)$$
$$\text{Asynchronous Machine} \quad M_{\text{async}} ::= (\tau, \Sigma, T_1, T_2)$$

$$\boxed{M_{\text{sync}} \xrightarrow{o} M'_{\text{sync}}}$$

$$\frac{[\![\tau]\!](\Sigma, s, p) = S \qquad \mathsf{forward}(S, p) = T'}{(\tau, \Sigma, T \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}) \xrightarrow{s,p} (\tau, \Sigma \uplus \{\!|(s,p)|\!\}, T \uplus T')}$$

$$\boxed{M_{\text{async}} \xrightarrow{o} M'_{\text{async}}}$$

$$\frac{\begin{array}{c}[\![\tau]\!](\Sigma, s, p) = S \qquad \mathsf{forward}(S, p) = T' \\ T_1' = T_1 \uplus T' \qquad T_2' = T_2 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}\end{array}}{(\tau, \Sigma, T_1 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}, T_2) \xrightarrow{s,p} (\tau, \Sigma, T_1', T_2')}$$

$$\frac{}{(\tau, \Sigma, T_1, T_2 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}) \to (\tau, \Sigma \uplus \{\!|(s,p)|\!\}, T_1, T_2)}$$

Figure 4. Reference machines.

Policies. Policies τ specify how packets should be forwarded through the network. Basic policies, written $e \to S$, say that packets matching e should be forwarded to the switches in S. As with predicates, we build complex policies by combining simple policies using intersection and negation. Figure 3 defines the semantics of policies as a function from snapshots x to sets of switches S. Although the policy language is syntactically simple, it is remarkably expressive. In particular, inspectors are a powerful tool that can be used to express a wide range of behaviors including load balancing, fine-grained access control, and many standard routing policies.

Machines. To understand how the network behaves over time, we define two abstract machines. Both machines forward packets according to τ but they differ in how often the switches synchronize traffic statistics with the controller. The *synchronous machine* defines an idealized implementation that, at all times, has perfect information about the traffic sent over the network. Of course, it would be impractical to implement this machine in a real network because, in general, it would require sending every packet to the controller—if any packet were forwarded by a switch there would be a delay between when the packet was forwarded and when the controller state was augmented with information about that packet. The *asynchronous machine* defines a looser, more practical, implementation. Like the first machine, it is *policy-compliant*—it forwards packets according to the policy—but it updates its state asynchronously instead of in lockstep with each packet processed. Hence, it makes no guarantees about what it knows about the network's traffic. While the synchronous machine can be thought of as the best possible policy-compliant machine, the asynchronous machine can be thought of as the worst policy-compliant machine. Any reasonable implementation will sit between the two. In other words, implementations should be policy compliant, but users should not expect perfect synchrony—the cost of implementing it would be prohibitive. In practice, synchronization with switches typically happens at periodic, timed intervals (modulo variances in the latency of communication) but for simplicity, we do not model time explicitly.

Figure 4 defines both reference machines. They use the function $\mathsf{forward}(S, p)$, which generates a multiset of transmissions,

$$\mathsf{forward}(S, p) = \{\!|\, \mathbb{T}(s \mid p) \mid s \in S |\!\}\,.$$

The state of the synchronous machine (M_{sync}) includes the NetCore policy τ, the state Σ, and a multiset T of pending transmissions. At each step, the machine removes a transmission from T, processes it using the policy, updates the machine state, and adds the new transmissions generated by the policy to the multiset of pending transmissions. The state of the asynchronous machine (M_{async}) includes the program τ, state Σ, and two multisets of transmissions: T_1, which represents transmissions waiting to be processed by the policy, and T_2, which represents transmissions that have been processed by the policy but have not yet been added to the state. The first inference rule for the second machine takes a transmission from T_1, processes it using the policy, and places it in T_2, the set of transmissions waiting to be incorporated into Σ; the second rule takes a transmission from T_2 and adds it to Σ.

4. The Run-time System

In this section we discuss how to implement NetCore's semantics on a software-defined network by giving an operational semantics to the NetCore run-time system and the underlying network devices. This operational semantics explains the basic interactions between the controller and the switches.

Switch classifiers. Before we can present the run-time system, we need a concrete representation of the rules that switches use to process packets. A *classifier* $\vec{r}$ is a sequence of rules r, each containing a switch-level pattern z and an action α. While our high-level semantics uses sets, classifiers are represented as sequences to model rule priority: within a classifier, rules on the left have higher priority than rules on the right.

The *pattern* (z) component of a rule recognizes a set of packets, and hence is similar to (but less general than) a predicate in NetCore. We write $p \sqsubseteq z$ when packet m matches pattern z. We hold patterns abstract to model the variety of different matching capabilities in today's switches. For example, OpenFlow switches support prefix pattern matching on source and destination IP addresses (*i.e.*, patterns like `0110*`) but only exact or full unconstrained matching on most other headers. Some switches support various other extended patterns, such as ranges of the form $[n_1, n_2]$.

An action α is either a set of switches S, which forwards packets to each switch in the set, or Ω, which forwards packets to the controller. Most switches support other actions such as modifying header fields, but for simplicity we only model forwarding.

Given a packet p and a classifier $\vec{r}$, we match the packet against the classifier by finding the first rule whose pattern matches the packet. We write $\vec{r} \rightsquigarrow^p z : \alpha$ for the matching judgment. More formally, we define classifier matching as follows; note that it selects the highest priority (leftmost) matching rule:

$$\frac{p \not\sqsubseteq z_1 \qquad \cdots \qquad p \not\sqsubseteq z_{i-1} \qquad p \sqsubseteq z_i}{(z_1 : \alpha_1, \ldots, z_{i-1} : \alpha_{i-1}, z_i : \alpha_i, \ldots, z_n : \alpha_n) \rightsquigarrow^p z_i : \alpha_i}$$

Molecular machine. We formalize the operational semantics of the run-time system as a *molecular machine*, in the style of the chemical abstract machine [2]. The machine's components, called *molecules*, are given on the left side of Figure 5. For simplicity, we assume that packets arriving at a switch may be processed in any order and do not model failures that cause packets to be dropped.

The molecule $\mathbb{C}(\tau \mid \Sigma)$ represents the controller machine running the NetCore policy τ in state Σ. The molecule $\mathbb{S}(s \mid \vec{r} \mid Z)$ represents switch s with packet classifier $\vec{r}$ and local switch state Z. The switch state records the patterns of rules that have been used to match packets but not yet queried and processed by the controller. Real switches use integer counters as state; for simplicity, we represent these counters in unary using a multiset of patterns. A transmission molecule $\mathbb{T}(s \mid p)$ represents a packet p en route to switch

$$
\begin{array}{rrcl}
\text{Pattern} & z & & \\
\text{Switch Action} & \alpha & ::= & S \mid \Omega \\
\text{Rule} & r & ::= & z : \alpha \\
\text{Classifier} & \vec{r} & ::= & (r_1, \ldots, r_n) \\
\text{Switch State} & Z & ::= & \{\!|z_1, \ldots, z_n|\!\} \\
\text{Molecule} & m & ::= & \mathbb{C}(\tau \mid \Sigma) \mid \\
 & & & \mathbb{S}(s \mid \vec{r} \mid Z) \mid \\
 & & & \mathbb{T}(s \mid p) \mid \\
 & & & \mathbb{H}(s \mid p) \\
\text{Machine} & M & ::= & \{\!|m_1, \ldots, m_n|\!\} \\
\text{Observation} & o & ::= & \cdot \mid s, p
\end{array}
$$

$$\boxed{M \xrightarrow{o} M'}$$

E-SwitchProcess

$$\frac{\vec{r} \rightsquigarrow^{p} z : S \qquad \mathsf{forward}(S, p) = T'}{\mathbb{S}(s \mid \vec{r} \mid Z), \mathbb{T}(s \mid p) \xrightarrow{s,p} \mathbb{S}(s \mid \vec{r} \mid Z \uplus \{\!|z|\!\}), T'}$$

E-SwitchHelp

$$\frac{\vec{r} \rightsquigarrow^{p} z : \Omega}{\mathbb{S}(s \mid \vec{r} \mid Z), \mathbb{T}(s \mid p) \to \mathbb{S}(s \mid \vec{r} \mid Z), \mathbb{H}(s \mid p)}$$

E-Controller

$$\frac{[\![\tau]\!](\Sigma, s, p) = S \qquad \mathsf{forward}(S, p) = T' \qquad \mathsf{Specialize}((\Sigma, s, p), \tau) = \vec{r}'}{\mathbb{C}(\tau \mid \Sigma), \mathbb{S}(s \mid \vec{r} \mid Z), \mathbb{H}(s \mid p) \xrightarrow{s,p} \mathbb{C}(\tau \mid \Sigma \uplus \{\!|(s,p)|\!\}), \mathbb{S}\left(s \mid (\vec{r}', \vec{r}) \mid Z\right), T'}$$

E-Collect

$$\frac{\vec{r} \rightsquigarrow^{p} z : \alpha}{\mathbb{C}(\tau \mid \Sigma), \mathbb{S}(s \mid \vec{r} \mid Z \uplus \{\!|z|\!\}) \to \mathbb{C}(\tau \mid \Sigma \uplus \{\!|(s,p)|\!\}), \mathbb{S}(s \mid \vec{r} \mid Z)}$$

E-Step

$$\frac{M \xrightarrow{o} M'}{(M \uplus M'') \xrightarrow{o} (M' \uplus M'')}$$

Figure 5. The run-time system.

s. Finally, a help molecule $\mathbb{H}(s \mid p)$ represents a request issued by switch s to the controller for assistance in processing packet p.

The operational semantics of the molecular machine is defined by the inference rules on the right side of Figure 5. To lighten the notation in this figure, we drop the multiset braces when writing a collection of molecules. In other words, we write $m_1, m_2, \ldots$ instead of $\{\!|m_1, m_2, \ldots|\!\}$. Each operational rule may optionally be labelled with an *observation* o, which records when transmissions are processed. We use observations in Section 6 where we establish equivalences between the molecular machine and the reference machines defined in the last section.

The rules E-SwitchProcess and E-SwitchHelp model the work done by switches to process packets. The former rule is invoked when a packet matches a rule with a non-controller action (a set of switches to forward to). In this case, the switch forwards the packet accordingly and records the rule pattern in its state. The latter rule is invoked when a packet matches a rule with a controller action. In this case, the switch generates a help molecule.

The rule E-Controller models the work done by the controller to process help molecules. The controller interprets the packet using its NetCore policy, generating new transmissions T' to be sent into the network, and adds the packet to its state. In addition, the controller uses the NetCore compiler to generate new rules to process future, similar packets on switches, instead of on the controller. The compiler is accessed through the call to the Specialize function, which generates the new rules for the switch in question. We hold the definition of this function abstract for now; it is defined precisely in the next section.

The rule E-Collect models the work done by the controller to transfer information about the packets that matched a particular switch-level rule from the switch to the controller state. More precisely, it chooses a pattern z from a switch state and then uses the lookup judgement $\vec{r} \rightsquigarrow^{p} z : \alpha$ to synthesize a packet p that might have matched the corresponding rule in the switch classifier. The pair of the packet and the switch are then stored in the controller state as a past transmission that might have occurred.

The interesting part of this transfer is that the controller stores *full* packets whereas switches only store sets of patterns and a pattern only specifies *part* of a packet—perhaps its IP address or VLAN tag, but not the packet itself. Hence the transfer operation must fabricate those parts of the packets that are not specified in the switch pattern, and the system as a whole must be correct no matter how the under-specified parts of a packet are fabricated. This places an important constraint on the compiler: If the rules and their patterns are not specific enough then although one packet may have matched a rule on a switch, a completely different packet may be fabricated and passed back to the controller. Consequently, the controller state will not model past network traffic sufficiently accurately and forwarding policies that depend upon past network traffic will not be implemented correctly.

A second subtle issue with the E-Collect rule is that the patterns of higher-priority rules partially overlap and take precedence over patterns from lower-priority rules. Hence, examining the pattern of a low-priority rule in isolation does not provide sufficient information to synthesize a packet that might have matched that rule. One must take all of the rules of the classifier, and their priority order, in to account when synthesizing a packet that may have matched a pattern. The E-Collect rule does this through the use of the full classifier matching judgement.

Finally, note that the implementation does not actually fabricate all of these packets—in practice, the switch passes integer counters associated with patterns back to the controller. Still, this nondeterministic rule effectively captures a key correctness criterion for the system: The controller program cannot distinguish between any of the packets that might be synthesized by the E-Collect rule and must be correct no matter which one is fabricated. Of course, this is also where the compiler's use of header information comes in to play: the fabricated packets are only different in fields that inspector functions (and other predicates) do not analyze.

5. The NetCore Algorithms

The NetCore system performs two distinct tasks:

- *Classifier Generation:* given a NetCore policy, construct a set of classifiers, one for each switch in the network.

$$
\begin{array}{rrl}
\text{Primitive intermediate form} & u & ::= (h_1 : \vec{w}_1) \wedge \ldots \wedge (h_n : \vec{w}_n) \\
\text{Three-valued boolean} & b & ::= \mathsf{True} \mid \mathsf{Maybe} \mid \mathsf{False} \\
\text{Pattern intermediate form} & \pi & ::= \langle u : z : b : H \rangle \\
\text{Policy intermediate form} & \rho & ::= \langle u : z : S_1, S_2 : H \rangle
\end{array}
$$

$\boxed{\mathcal{I}(s,e) = \vec{\pi}}$

$$
\begin{aligned}
\mathcal{I}(s, h : \vec{w}) &= \langle (h : \vec{w}) : \mathcal{O}(h : \vec{w}) : \mathsf{True} : \varnothing \rangle, \\
&\quad \langle \star : \top : \mathsf{False} : \varnothing \rangle \\
\mathcal{I}\left(s, \mathsf{switch}\ s'\right) &= \begin{cases} \langle \star : \top : \mathsf{True} : \varnothing \rangle & \text{if } s = s' \\ \langle \star : \top : \mathsf{False} : \varnothing \rangle & \text{if } s \neq s' \end{cases} \\
\mathcal{I}(s, \mathsf{inspect}\ e\ f) &= \prod_i \langle u_i : z_i : \mathsf{Maybe} : (H_i \cup H) \rangle \\
&\quad \text{where } f : \mathsf{State}[H] \times \mathsf{Switch} \times \mathsf{Packet}[H'] \to \mathsf{Bool} \\
&\quad \text{and } (\mathcal{I}(s,e))_i = \langle u_i : z_i : b_i : H_i \rangle \\
\mathcal{I}\left(s, e \cap e'\right) &= \prod_i \prod_j \langle u_i \wedge u'_j : z_i \sqcap z'_j : b_i \wedge b'_j : H_i \cup H'_j \rangle \\
&\quad \text{where } (\mathcal{I}(s,e))_i = \langle u_i : z_i : b_i : H_i \rangle \\
&\quad \text{and } \left(\mathcal{I}\left(s,e'\right)\right)_j = \langle u'_j : z'_j : b'_j : H'_j \rangle \\
\mathcal{I}(s, \neg e) &= \prod_i \langle u_i : z_i : \neg b_i : H_i \rangle \\
&\quad \text{where } (\mathcal{I}(s,e))_i = \langle u_i : z_i : b_i : H_i \rangle
\end{aligned}
$$

$\boxed{\mathcal{I}(s,\tau) = \vec{\rho}}$

$$
\begin{aligned}
\mathcal{I}(s, e \to S) &= \prod_i \begin{cases} \langle u_i : z_i : S, S : H_i \rangle & \text{if } b_i = \mathsf{True} \\ \langle u_i : z_i : \varnothing, S : H_i \rangle & \text{if } b_i = \mathsf{Maybe} \\ \langle u_i : z_i : \varnothing, \varnothing : H_i \rangle & \text{if } b_i = \mathsf{False} \end{cases} \\
&\quad \text{where } (\mathcal{I}(s,e))_i = \langle u_i : z_i : b_i : H_i \rangle \\
\mathcal{I}\left(s, \tau \cap \tau'\right) &= \prod_i \prod_j \langle u_i \wedge u'_j : z_i \sqcap z'_j : S'_1, S'_2 : H_i \cup H'_j \rangle \\
&\quad \text{where } (\mathcal{I}(s,\tau))_i = \langle u_i : z_i : S_{1i}, S_{2i} : H_i \rangle \\
&\quad \text{and } \left(\mathcal{I}\left(s,\tau'\right)\right)_j = \langle u'_j : z'_j : S'_{1j}, S'_{2j} : H'_j \rangle \\
&\quad \text{and } S'_1 = S_{1i} \cap S'_{1j} \\
&\quad \text{and } S'_2 = S_{2i} \cap S'_{2j} \\
\mathcal{I}(s, \neg\tau) &= \prod_i \langle u_i : z_i : \neg S_{2i}, \neg S_{1i} : H_i \rangle \\
&\quad \text{where } (\mathcal{I}(s,\tau))_i = \langle u_i : z_i : S_{1i}, S_{2i} : H_i \rangle
\end{aligned}
$$

$\boxed{\mathcal{C}(s,\tau) = \vec{r}}$

$$
\begin{aligned}
\mathcal{C}(s,\tau) &= \prod_i \begin{cases} z_i : S_{1i} & \text{if } S_{1i} = S_{2i} \text{ and } H_i \subseteq \mathsf{headers}(z_i) \\ & \text{and } \mathsf{consistent}(\vec{\rho}, i) \\ z_i : \Omega & \text{otherwise} \end{cases} \\
&\text{where } \mathcal{I}(s,\tau) = \vec{\rho} \\
&\quad \text{and } (\vec{\rho})_i = \langle u_i : z_i : S_{1i}, S_{2i} : H_i \rangle \\
&\quad \text{and } \mathsf{consistent}(\vec{\rho}, i) = \\
&\qquad \forall p.\, \exists j.\, \vec{\rho} \rightsquigarrow^p_z \langle u_i : z_i : S_{1i}, S_{2i} : H_i \rangle \Rightarrow \\
&\qquad\quad \vec{\rho} \rightsquigarrow^p_u \langle u_j : z_j : S_{1i}, S_{2i} : H_j \rangle \\
&\quad \text{and } \mathsf{headers}(z) = \{h \mid p_1 \sqsubseteq z \wedge p_2 \sqsubseteq z \Rightarrow p_1(h) = p_2(h)\}
\end{aligned}
$$

Figure 6. NetCore classifier generation.

- *Reactive Specialization:* given a packet not handled by the current classifier installed on a switch, generate additional rules that allow the switch to handle future packets with similar header fields without consulting the controller.

This section presents the key algorithms that implement these tasks.

5.1 Parameters

The NetCore system is parameterized on several structures: a lattice of switch patterns, and two oracles that map primitive predicates onto switch-level and wildcard patterns respectively. Abstracting some of the low-level details of compilation makes it possible to execute NetCore policies on many diverse kinds of hardware and even use switches with different capabilities in the same network. Formally, we assume that switch patterns form a bounded lattice. A pattern z_1 sits lower than (or equal to) another pattern z_2, written $z_1 \sqsubseteq z_2$, when z_1 matches a subset of the packets matched by z_2. The $\top$ element matches every packet and the $\bot$ element matches none. Abusing notation slightly, we write $p \sqsubseteq z$ to indicate that packet p matches pattern z. To ensure that intersections are compiled correctly, we require that meets be *exact*, in the sense that $p \sqsubseteq z \sqcap z'$ if and only if $p \sqsubseteq z$ and $p \sqsubseteq z'$.

The first oracle, called the *compilation oracle* $\mathcal{O}$, maps primitives $h : \vec{w}$ into the pattern lattice. In many cases, the pattern generated by $\mathcal{O}(h : \vec{w})$ will match the set of packets described by $h : \vec{w}$ exactly, but sometimes this is not possible. For example, OpenFlow switches only support prefix wildcards for IP addresses, so the best approximation of the non-prefix pattern $\mathsf{SrcAddr} : 1?1?$ is $\mathsf{SrcAddr} : 1???$. We give the oracle some flexibility in selecting patterns and only require it to satisfy two conditions: (1) it must return an overapproximation of the primitive and (2) it must be monotonic, in the sense it translates (semantically) larger primitives to larger patterns. Formally, the requirements on compilation oracles are as follows: (1) $(\Sigma, s, p) \in [\![h : w]\!]$ implies $p \sqsubseteq \mathcal{O}(h : w)$ and (2) $[\![h : w]\!] \subseteq [\![h' : w']\!]$ implies $\mathcal{O}(h : w) \sqsubseteq \mathcal{O}(h' : w')$.

The second oracle, called the *refinement oracle* $\mathcal{U}$, takes a primitive $h : \vec{w}$ and a packet p as arguments and produces a pattern $h : \vec{w}'$. Unlike the compilation oracle, which overapproximates predicates, the refinement oracle underapproximates predicates, allowing the compilation infrastructure to generate effective switch-level rules for a subset of the pattern of interest. While there is generally one best overapproximation, there often exist many useful underapproximations; in such cases, we disambiguate by selecting the best underapproximation that matches p. For example, if we were to refine $\mathsf{SrcAddr} : 1?1?$ (which can't be compiled exactly on OpenFlow) with a packet with source address 1111, we would generate the underapproximation $\mathsf{SrcAddr} : 111?$, yet if we refined the same predicate with a packet with source address 1010, we would instead generate $\mathsf{SrcAddr} : 101?$.

5.2 Classifier Generation

Ideally, given a policy, the NetCore compiler would generate a classifier with the same semantics—*i.e.*, one that denotes the same function on packets. But certain NetCore features, such as inspectors and wildcard patterns (when not supported by the underlying hardware), cannot be implemented on switches. So in general, the generated classifier will only approximate the policy, and certain packets will have to be processed on the controller.

The classifier generator works in two phases. In the first phase, it translates high-level policies to an intermediate form containing switch-level patterns and actions, as well as precise semantic information about the policy being compiled. In the second phase, it builds a classifier by attaching actions to patterns, using the semantic information produced in the first phase to determine whether it is safe to attach forwarding actions to a pattern, or whether the special controller action Ω must be used instead.

The grammars at the top of Figure 6 define the syntax for the intermediate forms used in classifier generation. The intermediate form for predicates $\langle u : z : b : H \rangle$ contains four values: an "ideal" pattern u; a switch pattern z; a three-valued boolean b; and a set of headers H. The ideal pattern u represents the pattern we would generate if the pattern lattice supported arbitrary wildcards. Ideal patterns are represented as a conjunction of header and wildcard pairs. We write $\star$ for the unconstrained ideal pattern—*i.e.*, an empty conjunction. The switch pattern z represents the actual pattern generated by the compiler, which is an overapproximation of the ideal pattern in general. The three-valued boolean b indicates whether packets matching the predicate should definitely be accepted (True), rejected (False), or whether there is insufficient information and a definitive answer must be made by the controller (Maybe). To combine three-valued booleans, we extend the standard boolean operators as follows:

$$\mathsf{Maybe} \wedge \mathsf{False} = \mathsf{False} \qquad \mathsf{Maybe} \wedge \mathsf{True} = \mathsf{Maybe}$$
$$\mathsf{Maybe} \wedge \mathsf{Maybe} = \mathsf{Maybe} \qquad \neg\mathsf{Maybe} = \mathsf{Maybe}$$

The set of headers H keeps track of the header fields (within packets in the controller state) that inspector functions may examine. This header information is used to ensure the compiler generates sufficiently fine-grained switch rules so that when information is transferred from the switch to the controller (using the E-COLLECT rule discussed in the previous section), the information is precise enough to guarantee the correctness of the inspectors.

The intermediate form for policies $\langle u : z : S_1, S_2 : H \rangle$ is similar to the form for predicates, but instead of a three-valued boolean, it records lower and upper bounds (S_1 and S_2) on the sets of switches to which a packet might be forwarded. Intuitively, a proper forwarding rule can only be generated when we know exactly which switches to forward packets to (*i.e.*, when S_1 and S_2 are equal). In other cases, the compiler will generate a rule that sends packets to the controller.

Predicate translation. The heart of classifier generation is the function $\mathcal{I}(s, e)$, presented in Figure 6, which takes a predicate e and switch s as arguments and produces a sequence of intermediate predicates that approximate e on s. One of the invariants of the algorithm is that it always generates a *complete* sequence—*i.e.*, intermediate predicates whose patterns collectively match every packet. In addition, the algorithm attempts to produce a sequence whose patterns separate packets into two sets—one with packets that match the predicate being compiled and another with those that do not. However, it does not always succeed in doing so, for two fundamental reasons: (1) the algorithm cannot analyze the decisions made by inspectors—as far as the analysis is concerned, inspectors are black boxes, and (2) certain primitive predicates cannot be expressed precisely using switch patterns. The intermediate predicates contain sufficient information for the compiler to reason about the precision of the rules it generates. We write $(\mathcal{I}(s, e))_i = \langle u_i : z_i : b_i : H_i \rangle$ to indicate that compiling e returns a sequence of intermediate predicates whose i^{th} element is $\langle u_i : z_i : b_i : H_i \rangle$, and

$$\prod_i \langle u_i : z_i : b_i : H_i \rangle$$

to denote the sequence of intermediate predicates out of components $\langle u_i : z_i : b_i : H_i \rangle$ indexed by i.

The first equation at the top of Figure 6 states that the compiler translates primitive predicates $h : \vec{w}$ into two intermediate predicates: $\langle (h : \vec{w}) : \mathcal{O}(h : \vec{w}) : \mathsf{True} : \varnothing \rangle$, which contains the switch pattern produced by the compilation oracle, and $\langle \star : \top : \mathsf{False} : \varnothing \rangle$, which, by using the $\top$ pattern, ensures that the sequence is complete. Like classifiers, these sequences should be interpreted as a prioritized series of rules. Hence the second quadruple only rejects things the first does not match.

The case for switch predicates switch s' has two possible outcomes: If the switch s whose classifier is being compiled is the same as s', then the compiler generates an intermediate form that associates every packet with True. Otherwise, the compiler generates an intermediate form that associates every packet with False.

Intuitively, the classifier generated for inspect e f must satisfy three conditions. First, it should approximate the semantics of f. Because the behavior of f is unknown at compile time, the approximation cannot be exact. Hence, the intermediate predicates generated for the inspector should contain Maybe, indicating that matching packets should be sent to the controller for processing. Second, it should be structured so that it can identify packets matched by the traffic filter predicate e—*i.e.*, the packets that must be present in the controller state to evaluate f. Third, it should also be sufficiently fine-grained to provide information about the set of headers H mentioned in the type of f, which represent the headers of packets in the state that f examines.[2] Hence, the compiler recursively generates a sequence of intermediate predicates from e, and then iterates through it, replacing the three-valued boolean b_i with Maybe and adding H to the set of headers H_i in each $\langle u_i : b_i : z_i : H_i \rangle$.

To generate intermediate forms for an intersection $(e_1 \cap e_2)$, the compiler combines each pair of intermediate predicates generated for e_1 and e_2. The resulting classifier captures intersection in the following sense: if a packet matches z_i in the first intermediate form and matches z'_j in the second intermediate form, it matches the form with pattern $z_i \sqcap z'_j$ in the result, and likewise for u_i and u'_j. Performing this construction naively would result in a combinatorial blowup. However, it is often possible to exploit algebraic properties of patterns to reduce the size of the sequence in practice—see the examples below and also Section 7.

Finally, the case for negated predicates: $\neg e$ iterates through the sequence generated by the compiler for e and negates the three-valued boolean in each intermediate predicate.

Predicate translation examples. To illustrate some of the details of predicate compilation, consider the translation of the inspector-free predicate $(e_1 \cap e_2)$ where e_1 is $(h_1 : 0?)$ and e_2 is $(h_2 : 11)$ and h_1 and h_2 are distinct headers. Assume that switch patterns support wildcards such as $0?$ on h_1. The left and right sides of the intersection generate the following intermediate predicates:

$$\mathcal{I}(s, e_1) = \langle (h_1 : 0?) : (h_1 : 0?) : \mathsf{True} : \varnothing \rangle, \langle \star : \top : \mathsf{False} : \varnothing \rangle$$
$$\mathcal{I}(s, e_2) = \langle (h_2 : 11) : (h_2 : 11) : \mathsf{False} : \varnothing \rangle, \langle \star : \top : \mathsf{True} : \varnothing \rangle$$

Note that the negation in e_2 flips the parts of the intermediate forms designated as True—*i.e.*, it inverts the parts of the sequence that match and do not match the predicate.

Next, consider compilation of the intersection, and note that we simplify the results slightly using identities such as $z \sqcap \top = z$ and $u \wedge \star = u$ and $b \wedge \mathsf{True} = b$.

$$\langle (h_1 : 0? \wedge h_2 : 11) : (h_1 : 0? \sqcap h_2 : 11) : \mathsf{True} : \varnothing \rangle,$$
$$\langle (h_1 : 0?) : (h_1 : 0?) : \mathsf{False} : \varnothing \rangle,$$
$$\langle (h_2 : 11) : (h_2 : 11) : \mathsf{False} : \varnothing \rangle,$$
$$\langle \star : \top : \mathsf{False} : \varnothing \rangle$$

This classifier can then be simplified further, as the last three rules overlap and are associated with the same three-valued boolean:

$$\langle (h_1 : 0? \wedge h_2 : 11) : (h_1 : 0? \sqcap h_2 : 11) : \mathsf{True} : \varnothing \rangle,$$
$$\langle \star : \top : \mathsf{False} : \varnothing \rangle$$

Now suppose instead that the switch only has limited support for wildcards and cannot represent $h_1 : 0?$. In this case, the compilation

[2] Note that the compiler does not add the set H' mentioned in the type of f to H_i. This set is used during specialization to determine "similar" packets.

oracle provides an overapproximation of the pattern, say, $\top$. Hence, the intermediate predicate for e_1 above would be as follows:

$$\mathcal{I}(s, e_1) = \langle (h_1 : 0?) : \top : \mathsf{True} : \varnothing \rangle, \langle \star : \top : \mathsf{False} : \varnothing \rangle$$

For another example, consider compiling a predicate that includes an inspector such as $(\mathsf{inspect}\ (h_1 : 00)\ f) \cap (h_2 : 11)$. In this case, $(h_1 : 00)$ compiles similarly to the simple clauses above:

$$\langle (h_1 : 00) : (h_1 : 00) : \mathsf{True} : \varnothing \rangle, \langle \star : \top : \mathsf{False} : \varnothing \rangle$$

If the set of headers f examines on the state is H, the inspector $\mathsf{inspect}\ (h_1 : 00)\ f$ compiles to the following:

$$\langle (h_1 : 00) : (h_1 : 00) : \mathsf{Maybe} : H \rangle, \langle \star : \top : \mathsf{Maybe} : H \rangle$$

Note that the definitive booleans above have been replaced with Maybe, indicating that the controller will need to determine whether packets match the predicate. However, when we intersect the results of compiling $h_2 : 11$ with the results of compiling the inspector, we obtain the following:

$$\begin{array}{l} \langle (h_1 : 00 \wedge h_2 : 11) : (h_1 : 00 \sqcap h_2 : 11) : \mathsf{Maybe} : H \rangle, \\ \langle (h_2 : 11) : (h_2 : 11) : \mathsf{Maybe} : H \rangle, \\ \langle (h_1 : 00) : (h_1 : 00) : \mathsf{False} : H \rangle, \\ \langle \star : \top : \mathsf{False} : H \rangle \end{array}$$

Importantly, even though the inspector is uncertain (*i.e.*, it has Maybe in each intermediate predicate), the result is not entirely uncertain. Because $b \wedge \mathsf{False}$ is False even when b is Maybe, intersecting inspectors with definitive predicates can resolve uncertainty. Likewise, as $b \vee \mathsf{True}$ is True, compiling the union of a definitive clause with an inspector also eliminates uncertainty. And although the calculus does not represent unions explicitly, its encoding operates as expected—a fact we exploit in reactive specialization.

Policy translation. The function $\mathcal{I}(s, \tau)$, which translates a policy into intermediate form, is similar to the translation for predicates. Figure 6 gives the formal definition of the translation. To translate a basic policy $e \to S$, the compiler first generates a sequence from e, and then attaches a pair of actions representing lower and upper bounds for each rule. There are three cases: If the three-valued boolean b_i is true, it uses S as both the upper and lower bounds. If b_i is false, it uses $\varnothing$ as the bounds. If b_i is Maybe, it uses $\varnothing$ as the lower bound and S as the upper bound, which represents the range of possible actions. The translations of intersected and negated policies are analogous to the cases for predicates.

Classifier construction. The second phase of classifier generation analyzes the intermediate form of the policy and produces a bona fide switch classifier. The $\mathcal{C}(s, \tau)$ function that implements this phase is defined in Figure 6. It first uses $\mathcal{I}(s, \tau)$ to generate a sequence of intermediate policies, and then analyzes each $\langle u_i : S_{1i}, S_{2i} : z_i : H_i \rangle$ to generate a rule. There are two possible outcomes for each intermediate policy in the sequence. First, if (1) the bounds S_{1i} and S_{2i} are tight, (2) z_i is sufficiently fine grained to collect information about all headers in H_i, and (3) we get the same switch bounds (S_{1i}, S_{2i}) regardless of whether we match packets using the ideal primitives or the switch-level patterns, then it is safe for the compiler to throw away the high-level semantic information (bounds and ideal primitives) and emit an effective rule $z_i : S_{1i}$. Otherwise, it generates a rule with the controller action Ω. The formal conditions needed for this analysis are captured by $\mathsf{consistent}(\vec{\rho}, i)$ and $\mathsf{headers}(z)$. The predicate $\mathsf{consistent}(\vec{\rho}, i)$ is satisfied if looking up an arbitrary packet matching the i^{th} switch pattern yields the same switch bounds as looking it up using the ideal pattern (where we extend classifier lookup to sequences of intermediate policies in the obvious way). The function $\mathsf{headers}(z)$ calculates the set of headers constrained fully by z.

Formal properties. For a classifier to be sound, it must satisfy two properties: it must forward packets according to the policy, and its rules must encode enough information about the packets that match them to implement inspectors. The correctness of classifier generation is captured in the following definition and lemma.

Definition 1 (Classifier Soundness)**.** A classifier $\vec{r}$ is sound on switch s with respect to τ if the following two criteria hold:

- *Routing soundness:* for all snapshots (Σ, s, p), if $\vec{r} \rightsquigarrow^{p} S$ then $[\![\tau]\!](\Sigma, s, p) = S$, and
- *Collection soundness:* for all packets p_1 and p_2, if $\vec{r} \rightsquigarrow^{p_1} z : S$ and $\vec{r} \rightsquigarrow^{p_2} z : S$, then for all snapshots (Σ, s', p'),

$$[\![\tau]\!](\Sigma \uplus \{\!|(s, p_1)|\!\}, s', p') = [\![\tau]\!](\Sigma \uplus \{\!|(s, p_2)|\!\}, s', p').$$

Lemma 1 (Classifier Generation Soundness)**.** Classifier $\mathcal{C}(s, \tau)$ is sound on switch s with respect to τ.

Intuitively, routing soundness ensures that the actions computed by looking up rules in the classifier $\vec{r}$ are consistent with τ. Formally, the condition states that if looking up a packet p in $\vec{r}$ on switch s produces a set of switches S, then evaluating τ on snapshots containing s and p also produces S. Note that this condition does not impose any requirements if looking up p in $\vec{r}$ yields Ω, as the packet will be sent to the controller, which will evaluate τ on p directly. Collection soundness ensures that the rules in $\vec{r}$ are sufficiently fine grained so that when the controller collects traffic statistics from switches, the rule patterns contain enough information to implement the policy's inspectors. This is seen in the E-COLLECT rule in the molecular machine (Figure 5), which fabricates packets that match the rule being collected. Collection soundness ensures that fabricated packets are correct. Formally, it requires that τ behave the same on all snapshots in which the state Σ has been extended with arbitrary packets p_1 and p_2 matching a given rule $z : S$. Lemma 1 states that the classifiers generated by the NetCore compiler are sound.

5.3 Reactive Specialization

The algorithm described in the preceding section generates classifiers that can be installed on switches. But as we saw, it has some substantial limitations. Dynamic policies that use inspectors cannot be analyzed. And even for purely static policies, if the switch has poor support for wildcards, the classifier needed to implement the policy may be large—much larger than would be practical to generate. To deal with these situations, we define *reactive specialization*, a powerful generalization of the simple, reactive, strategy implemented manually by OpenFlow programmers. We define reactive specialization using two operations: *program refinement*, which expands the policy relative to a new snapshot witnessed at the controller, and *pruning*, which extracts new, effective rules from the classifier generated from the expanded policy.

Program refinement. When the controller receives a new packet that a switch could not handle, it interprets the policy with respect to the packet, switch, and its current state. The idea in program refinement is to augment the program with additional information gleaned from this packet that can be used to build a specialized classifier that handles similar packets on the switch in the future. Figure 7 defines the refinement function. The key invariant of this program transformation is that the semantics of the old and new policies are identical. However, syntactically, the new program will typically have a different structure, as the transformation uses the packet to unfold primitives and inspectors. This makes compilation more precise and the recompiled program more effective.

The rules for refining a predicate appear at the top of Figure 7. The first rule uses the refinement oracle $\mathcal{U}$ to refine basic predicates. Unlike the compilation oracle, which may *overapproximate*

$\boxed{\mathcal{R}(x, e) = e'}$

$$\mathcal{R}((\Sigma, s, p), h : \vec{w}) = (h : \vec{w}) \cup \mathcal{U}(h : \vec{w}, p)$$
$$\mathcal{R}(x, \mathsf{switch}\ s) = \mathsf{switch}\ s$$
$$\mathcal{R}((\Sigma, s, p), \mathsf{inspect}\ e\ f) =$$
$$\begin{cases} (\mathsf{inspect}\ e'\ f) \cup e'' & \text{if } \mathsf{invariant}(x, f) \wedge f(x) \\ (\mathsf{inspect}\ e'\ f) \setminus e'' & \text{if } \mathsf{invariant}(x, f) \wedge \neg f(x) \\ \mathsf{inspect}\ e'\ f & \text{if } \neg\, \mathsf{invariant}(x, f) \end{cases}$$
$$\text{where } f : \mathsf{State}[H] \times \mathsf{Switch} \times \mathsf{Packet}[H'] \to \mathsf{Bool}$$
$$\text{and } x = (\Sigma, s, p)$$
$$\text{and } e' = \mathcal{R}(x, e) \cup (\mathcal{R}(x, e) \cap \mathsf{similar}(s, p, H))$$
$$\text{and } e'' = \mathsf{similar}(s, p, H')$$
$$\mathcal{R}(x, e_1 \cap e_2) = \mathcal{R}(x, e_1) \cap \mathcal{R}(x, e_2)$$
$$\mathcal{R}(x, \neg e) = \neg\, \mathcal{R}(x, e)$$

$\boxed{\mathcal{R}(x, \tau) = \tau'}$

$$\mathcal{R}(x, e \to S) = \mathcal{R}(x, e) \to S$$
$$\mathcal{R}(x, \tau_1 \cap \tau_2) = \mathcal{R}(x, \tau_1) \cap \mathcal{R}(x, \tau_2)$$
$$\mathcal{R}(x, \neg\tau) = \neg\, \mathcal{R}(x, \tau)$$

$\boxed{\mathsf{Specialize}(x, \tau) = \vec{r}}$

$$\mathsf{Specialize}((\Sigma, s, p), \tau) = \mathsf{prune}(\vec{r}, p)$$
$$\text{where } \vec{r} = \mathcal{C}(s, \mathcal{R}((\Sigma, s, p), \tau))$$

Figure 7. NetCore refinement.

the predicate, the refinement oracle *underapproximates* it, so that the rest of the compilation infrastructure will be able to generate an effective switch-level rule that matches the given packet. Because the new predicate is the union of the old predicate and an underapproximation, the overall semantics is unchanged. In some cases, especially if the switch-supported patterns are weak, the best underapproximation the refinement oracle can generate is an exact-match predicate. In many other cases, however, if the switch supports prefix matching or wildcards, the refinement oracle will produce a predicate that matches many more packets.

The second rule refines switch predicates $\mathsf{switch}\ s$. Because the switch predicate already reveals the maximum amount of information, it cannot be refined further.

The rule for inspectors is the most interesting. It uses a similarity predicate that describes the set of packets sent to the same switch that agree on a set of headers H:

$$\mathsf{similar}(s, p, H) = \mathsf{switch}\ s \cap \bigcap_{h \in H} (h : p(h)) .$$

We first refine the traffic filter predicate e to add additional structure for traffic collection. To ensure that the refined classifier has sufficiently fine-grained rules to collect the packets in the controller state examined by f, we form the union of the refined traffic filter and the similarity predicate $\mathsf{similar}(s, p, H)$, restricted to the refined traffic filter. Next, we add additional information about the inspector's decision on the packet p to the policy. Recall that if f is invariant with respect to a snapshot x (which includes the controller state, switch, and packet) then it will return the same decision on all similar packets in the future. In the first case, if the inspector is invariant and evaluates to true on the current snapshot (Σ, s, p), then we refine it by taking the union of the inspector and the similarity predicate $\mathsf{similar}(s, p, H')$. The second case is similar, except that the inspector does *not* evaluate to true, and hence we refine the inspector by subtracting the similarity predicate. Finally, in the third case, the inspector is not invariant so no sound refinement exists—the decision returned by the inspector may change in the future if the controller state changes. Hence, packets must continue being diverted to the controller until the inspector becomes invariant.

The rules for refining intersection $(e_1 \cap e_2)$ and negation $\neg e$ predicates and policies are all straightforward.

Pruning. In general, after a policy has been refined and recompiled, some of the new rules will be useless—they will not process additional packets on the switch. We prune away these useless rules using a function $\mathsf{prune}(\vec{r}, p)$ that removes rules from $\vec{r}$ that (1) send packets to the controller (adding such rules does not improve the efficiency of the switch), (2) have nothing to do with the packet p (meaning they are irrelevant to specialization with respect to p), or (3) overlap with a rule we removed earlier (to preserve the semantics of the rules).

Putting it all together. We define reactive specialization (the function Specialize at the bottom of Figure 7), by composing refinement, recompilation, and pruning to generate a specialized classifier from a snapshot x and policy τ.

Formal properties. We first establish that specialization, and therefore reactive rule generation, is sound.

Lemma 2 (Specialization Soundness)**.** If $\vec{r}$ is sound on switch s with respect to τ and $\vec{r}' = \mathsf{Specialize}((\Sigma, s, p), \tau)$, then $(\vec{r}', \vec{r})$ is sound on s with respect to τ.

To establish the other properties, we need a way of characterizing the packets that go to the controller. We define the *controller set* of a classifier $\vec{r}$ as follows:

$$\Omega(\vec{r}) = \{p \mid \vec{r} \rightsquigarrow^{p} \Omega\} .$$

The second property we establish is that refinement is *monotonic*. That is, if we append reactive rules to a switch's classifier, the resulting classifier does not send more packets to the controller that the original one. Formally,

Lemma 3 (Specialization Monotonicity)**.** For all policies τ and classifiers $\vec{r}$ and $\vec{r}'$ such that $\vec{r}' = \mathsf{Specialize}((\Sigma, s, p), \tau)$ we have $\Omega((\vec{r}', \vec{r})) \subseteq \Omega(\vec{r})$.

The final property we establish is that under certain assumptions, appending reactive rules to a classifier results in strictly fewer packets going to the controller. To make such a guarantee, we need two conditions: First, the policy τ must be *realizable*—intuitively, it must only use features that can be implemented on switches (*e.g.*, on an OpenFlow switch, the policy must not match on payloads).

Definition 2 (Realizable)**.** A policy τ is *realizable* if, for every subterm $h : \vec{w}$ of τ and $p \in [\![h : w]\!]$, we have $(\Sigma, s, p) \in [\![\mathcal{U}(h : w, p)]\!]$ if and only if $p \sqsubseteq \mathcal{O}(\mathcal{U}(h : w, p))$.

Realizability states that compiling an underapproximation of a high-level predicate with respect to a packet matching the predicate yields a switch-level rule that exactly corresponds to the predicate. Second, all inspectors in the policy must be determinate. We formalize this by extending the notion of invariance to *full invariance*:

Definition 3 (Fully Invariant)**.** A policy τ is *fully invariant* on Σ if for every subterm of τ of the form $\mathsf{inspect}\ e\ f$ and we have $\mathsf{invariant}((\Sigma, s, p), f)$ for all switches s and packets p.

For policies satisfying these conditions, we can guarantee that the packet used to refine and recompile the policy will never be sent to the controller again.

Lemma 4 (Specialization Progress)**.** If τ is realizable and fully invariant on Σ, and $\vec{r}' = \mathsf{Specialize}((\Sigma, s, p), \tau)$, then for any classifier $\vec{r}$, we have $p \notin \Omega((\vec{r}', \vec{r}))$.

6. System-wide Correctness Properties

This section uses the tools developed in the previous section to deliver our two central theoretical results: (1) a proof of *functional correctness* for NetCore, and (2) a proof of *quiescence*, another fundamental theorem which establishes that, when inspectors are invariant, the network eventually reaches a state in which all processing occurs efficiently on its switches.

Functional correctness. Recall in Section 3 we defined two idealized reference machines: the synchronous reference machine, which at all times knows (and has recorded) information about every packet processed in the network, and the asynchronous reference machine, which nondeterministically learns about packets processed in the network. To demonstrate the correctness of the NetCore compiler, we show that it inhabits the space between the asynchronous and synchronous reference machines. More formally, we prove that the asynchronous reference machine simulates the NetCore molecular machine and the molecular machine simulates the synchronous reference machine.

Given a set of switches S and a policy τ, we initialize the molecular machine as follows:

$$\mathsf{Init}(S, \tau) = \{\!|\, \mathbb{C}\,(\tau \mid \varnothing) \,|\!\} \uplus \{\!|\, \mathbb{S}\,(s \mid \mathcal{C}\,(s, \tau) \mid \varnothing) \mid s \in S |\!\}\,.$$

The next theorem establishes the relationship between the reference and molecular machines.

Theorem 1 (Functional Correctness)**.** Given a set of switches S, an initial set of transmissions T such that $\mathbb{T}\,(s \mid p) \in T$ implies $s \in S$, and a molecular machine $M = \mathsf{Init}(S, \tau) \uplus T$, we have:

- The asynchronous machine $(\tau, \varnothing, T, \varnothing)$ weakly simulates M.
- M weakly simulates the synchronous machine $(\tau, \varnothing, T)$.

Proof sketch. We describe the first simulation only; the second is similar. The simulation relation between the asynchronous machine and the molecular machine satisfies the following: (1) each switch's classifier on the molecular machine is sound with respect to τ, (2) there exists an observation-preserving bijection between pending transmissions in the asynchronous machine and transmissions and help molecules in the molecular machine, and (3) there exists an observation–preserving bijection between the processed transmissions in the asynchronous machine and the switch states in the molecular machine. The initial state satisfies these criteria by classifier generation soundness. Now, consider taking a transition. If it forwards a packet, the first bijection is preserved by routing correctness; if it collects a pattern, the second bijection is preserved by collection soundness; finally, if it generates reactive rules, they are sound by specialization soundness. □

Quiescence. The quiescence theorem demonstrates that the NetCore compiler effectively moves work off of the controller and onto switches, even when the program is expressed in terms of patterns the switch cannot implement precisely and inspector functions the compiler cannot analyze. Formally, quiescence states that if all of the inspectors in the program are invariant, then the NetCore compiler will eventually install rules on switches that handle all future traffic—*i.e.*, eventually, the system can reach a configuration where no additional packets need to be sent to the controller.

Before we can state the quiescence theorem precisely, we need a few definitions and supporting lemmas. First, we say that M is derived from policy τ if $(\mathsf{Init}(S, \tau) \uplus T) \rightarrow^* M$, where $\rightarrow^*$ is the reflexive, transitive closure of the single step judgement, ignoring observations. Second, we lift the notion of full invariance to machines M; M is fully invariant if the controller's policy is fully invariant with respect to the controller's state. We also lift the notion of *controller set* on classifiers to machines M:

$$\Omega(M) = \{(s, p) \mid \mathbb{S}\,(s \mid \vec{r} \mid Z) \in M \text{ and } p \in \Omega(\vec{r})\}\,.$$

The first lemma, *controller set monotonicity*, states that the set of packets that require processing on the controller never increases:

Lemma 5 (Controller Set Monotonicity)**.** If M is derived from τ and $M \xrightarrow{o} M'$, then $\Omega(M') \subseteq \Omega(M)$.

Proof. Follows from specialization monotonicity. □

Now we are ready to prove the key lemma needed for quiescence. The *controller set progress* lemma states that if the controller program is realizable and has become fully invariant, then every time the controller processes a help molecule, the controller set becomes strictly smaller. In other words, every help molecule contains enough information (and the compiler is powerful enough to exploit it) for the E-CONTROLLER rule to generate useful new classifier rules.

Lemma 6 (Controller Set Progress)**.** For every realizable policy τ and fully invariant M derived from τ, if $M \xrightarrow{o} M'$ is an instance of E-CONTROLLER then $\Omega(M') \subset \Omega(M)$.

Proof. Follows from specialization progress. □

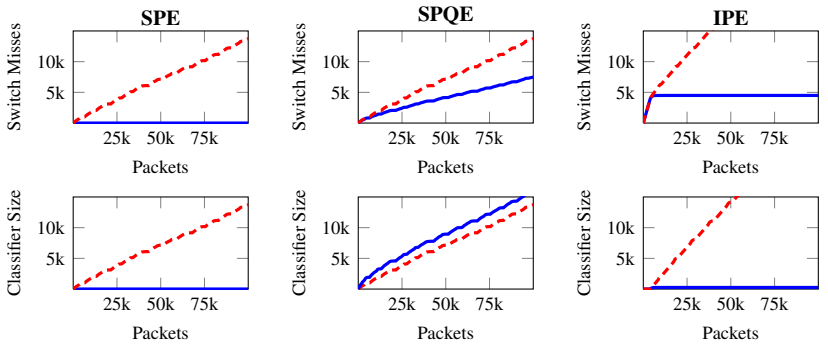
Quiescence follows from these lemmas, as the total number of possible packets is finite. The precise statement of quiescence says that the run-time system *may* (as opposed to *does*) quiesce, because the machine may non-deterministically choose to continue forwarding packets using the switches instead of processing the remaining help molecules. Formally, a machine configuration M *may quiesce* if there exists a configuration M' such that $M \rightarrow^* M'$ and the rule E-CONTROLLER is not used in any derivation in the operational semantics starting from M'. With this definition in hand, we can state quiescence.

Theorem 2 (Quiescence)**.** For every realizable policy τ and fully invariant M derived from τ, we have that M may quiesce.

7. Implementation and Evaluation

We have implemented a prototype NetCore compiler in Haskell using the ideas presented in this paper. The core algorithms are formulated in terms of abstract type classes (*e.g.*, lattices for switch patterns) and oracles. This makes it easy to instantiate the compiler for different switch architectures—one simply has to define instances for a few type classes and provide the appropriate oracles. We have built two back-ends, both targeting OpenFlow switches. The first generates coarse-grained wildcard rules. The other back-end, used for comparison, generates the kind of exact-match rules used in our earlier work on Frenetic [9] and most hand-written NOX applications [11].

Optimizations. The implementation uses a number of heuristic optimizations to avoid the combinatorial blowup that would result from compiling classifiers naively. For example, it applies algebraic rewritings on-the-fly to remove useless patterns and rules and reduce the size of the intermediate patterns and classifiers it needs to manipulate. The compilation algorithms identify and remove patterns completely "shadowed" by other patterns and patterns whose



	SPE	SPQE	IPE
Full Compiler			
Switch Misses	0	7.5k	4.5k
Classifier Size	8	15.9k	0.3k
μFlow Compiler			
Switch Misses	13.7k	13.7k	33.4k
Classifier Size	13.7k	13.7k	29.2k

Figure 8. Experimental results.

effect is "covered" by a larger pattern with lower priority but the same actions. Although these heuristics are simple, they go a long way toward ensuring reasonable performance in our experience.

Evaluation. To evaluate our implementation, we built an instrumented version of the run-time system that collects statistics about the sizes of the classifiers generated by the compiler and the amount of traffic handled on switches (as opposed to the controller). Because space for classifiers is a limited resource on switches, and because the cost of diverting a packet to the controller slows down its processing by orders of magnitude, these metrics quantify some of the most critical performance aspects of the system.

We compared the performance of the "full" (which makes use of all OpenFlow rules, including wildcards) and "μflow" (which only generates exact-match rules, also known as *micro*flow rules) compilers on the following programs:

- **Static Policy Experiment (SPE)**: implements the simple static policy described at the beginning of Section 2. This benchmark measures the (in)efficiency of compilation strategies based on generating exact-match rules.
- **Static Policy with Query Experiment (SPQE)**: forwards packets using the same policy as in SPE but also collects traffic statistics for each host. Due to this collection, this program cannot be directly compiled to a switch classifier—at least, not without expanding all 4.3 billion possible hosts! Thus, this benchmark measures the efficiency of reactive specialization.
- **Inspector Policy Experiment (IPE)**: forwards packets and collects traffic statistics using the authentication application presented in Section 2. This benchmark measures the performance of a more realistic application implemented using inspectors.

To drive these experiments, we generated packets using *fs* [22], a tool that synthesizes realistic packet traces from several statistical parameters. We ran each experiment on 100K packets in total. For the SPE and SPQE benchmarks, we generated traffic with 1024 active hosts sending packets to an external network for 30 seconds each. For the IPE benchmark, we generated traffic with 254 hosts (a class C network) sending traffic to the authentication server and an external network for 30 seconds each. The results of the experiments are shown in Figure 8. The graphs on the top row show the number of packets that "missed" and had to be sent to the controller against the total number of packets processed. Likewise, the graphs on the bottom row show the size of the compiled classifier, in terms of number of rules, versus total packets. The table at the right gives the final results after all 100K packets were processed.

In terms of the proportion of packets processed on switches, the full OpenFlow compiler outperforms the microflow-based compiler on nearly all of the benchmarks. On the SPE benchmark, the full compiler generates a classifier that completely handles the policy, so no packets are sent to the controller. (The line for the full compiler overlaps with the x-axis.) The microflow compiler, of course, diverts a packet to the controller for each distinct microflow, generating 13.7k rules in total. On the SPQE benchmark, the full compiler generates wildcard rules (using reactive specialization) that handle all future traffic from each unique host after seeing a packet from it. These rules handle many more packets than the exact-match rule produced by the microflow compiler. On this benchmark, it is worth noting that the classifiers produced by the full compiler are larger than the ones produced by the microflow compiler, especially initially. This is due to the fact that the full compiler generates multiple rules in response to a single controller packet, attempting to cover a broad space of future similar packets, whereas the microflow compiler predictably generates a single microflow for each controller packet. One can see that the work done by the full compiler pays off in terms of the number of packets that must be diverted to the controller. Moreover, over time, the size of the microflow compiler-generated classifier approaches that of the full compiler. Lastly, the IPE benchmark demonstrates that the full compiler generates more effective classifiers than the microflow compiler, even in the presence of inspector functions that it cannot analyze directly. Note that a large number of packets must be diverted to the controller in any correct implementation—until they authenticate, the inspector is not invariant for any host. However, the full compiler quickly converges to a classifier that processes all traffic directly on the switch.

8. Related Work

Building on ideas first proposed in Ethane [4] and 4D [10], NOX [11] was the first concrete system to popularize what is currently known as software-defined networking. It provides an event-driven interface to OpenFlow [17] and requires that programmers write reactive programs using callbacks and explicit, switch-level packet-processing rules. There are numerous examples of network applications built on top of NOX using microflows [12, 13, 27], but relatively few that use wildcard rules (though Wang's load balancer [26] is a nice example of the latter).

Networking researchers are now actively developing next-generation controller platforms. Some of them, such as Beacon [1] (designed for Java) and Nettle [25] (designed for Haskell) provide elegant OpenFlow interfaces for new programming languages. Others, such as Onix [15], and Maestro [3] improve scalability and fault tolerance through parallelization and distribution. None of

these systems automatically generate reactive protocols or provide formal semantics or correctness guarantees like NetCore does.

Both NetCore and NDLog [16] use high-level languages to program networking infrastructure, but the similarities end there. NDLog programs are written in an explicitly distributed style whereas high-level NetCore programs are written as if the program has an omniscient, centralized view of the entire network. The NetCore implementation automatically partitions work onto a distributed set of switches and synthesizes a reactive communication protocol that simulates the semantics of the high-level language.

Part of the job of the NetCore compiler is to generate efficient packet classifiers. Most previous research in this area (see Taylor [24] for a survey) focuses on static compilation. The NetCore compiler generates classifiers in the face of non-static policies, with unknown inspector functions, and synthesizes a distributed switch-controller implementation. Bro [21], Snortran [7], Shangri-La [5] and FPL-3E [6] compile rich packet-filtering and monitoring programs, designed to secure networks and detect intrusions, down to special packet-processing hardware and FPGAs. The main difference between NetCore and all of these systems is that they are limited to a single device. They do not address the issue of how to program complex, dynamic policies for a collection of interconnected switches and they do not synthesize the distributed communication patterns between the switches and controller.

Active Networking, as in the SwitchWare project [23], shares many high-level goals with Software-Defined Networking, but the implementation strategy is entirely different. The former uses smart switches to interpret programs encapsulated in packets, while the latter uses dumb switches controlled by a remote host.

Acknowledgments

We wish to thank Ken Birman, Mike Freedman, Sharad Malik, Jen Rexford, Cole Schlesinger, and Alec Story and the anonymous POPL reviewers for discussions about this research and comments on drafts of this paper. Our work is supported in part by ONR grants N00014-09-1-0770 and N00014-09-1-0652 and NSF grants CNS-1111698 and CNS-1111520. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR or NSF.

References

[1] Beacon: A java-based OpenFlow control platform., Nov 2010. See `http://www.beaconcontroller.net`.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *POPL*, pages 81–94, 1990.

[3] Z. Cai, A. Cox, and T. Ng. Maestro: A system for scalable OpenFlow control. Technical Report TR10-08, Rice University, Dec 2010.

[4] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *Trans. on Networking.*, 17(4), Aug 2009.

[5] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *PLDI*, Jun 2005.

[6] M. Cristea, C. Zissulescu, E. Deprettere, and H. Bos. FPL-3E: Towards language support for reconfigurable packet processing. In *SAMOS*, pages 201–212, Jul 2005.

[7] S. Egorov and G. Savchuk. *SNORTRAN: An Optimizing Compiler for Snort Rules*. Fidelis Security Systems, 2002.

[8] D. Erickson et al. A demonstration of virtual machine mobility in an OpenFlow network, Aug 2008. Demo at *ACM SIGCOMM*.

[9] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[10] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35:41–54, October 2005.

[11] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[12] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM*.

[13] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, Apr 2010.

[14] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *Hot-ICE*, Mar 2011.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, Oct 2010.

[16] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, 2005.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[18] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control in enterprise networks. In *WREN*, Aug 2009.

[19] The Open Networking Foundation, Mar 2011. See `http://www.opennetworkingfoundation.org/`.

[20] OpenFlow, Nov 2010. See `http://www.openflowswitch.org`.

[21] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, Dec 1999.

[22] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield. Efficient network-wide flow record generation. In *INFOCOM*, pages 2363–2371, 2011.

[23] SwitchWare. `http://www.cis.upenn.edu/~switchware`, 1997.

[24] D. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37:238–275, September 2005.

[25] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, Jan 2011.

[26] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, Mar 2011.

[27] K. Yap, M. Kobayashi, R. Sherwood, T. Huang, M. Chan, N. Handigol, and N. McKeown. OpenRoads: Empowering research in mobile networks. *SIGCOMM CCR*, 40(1):125–126, 2010.