A Compiler and Run-time System for Network Programming Languages

Christopher Monsanto
Princeton University

Nate Foster
Cornell University

Rob Harrison *
United States Military Academy

David Walker
Princeton University

Abstract

Software-defined networks (SDNs) are a new implementation 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 enterprise and data center networks are implemented. But to do this, they need novel programming languages to help them craft network-wide algorithms for routing, energy-efficient network management, dynamic access control, traffic monitoring, and other applications.

In this paper, we define a high-level language, called NCore, for expressing packet-forwarding policies and traffic-statistics queries. The language is designed to be simple, expressive, and compositional. We define a formal semantics for NCore and show how to compile it to a distributed switch-controller architecture. To ensure that a majority of packets are processed efficiently on switches, we develop a new compilation technique called *reactive specialization* that generalizes, improves on, and automates the simple (but inefficient) manual techniques commonly used to program SDNs. Reactive specialization and the other compilation techniques we develop are highly generic, assuming only that the packet-matching capabilities available on switches satisfy some basic algebraic laws. This generality makes our technology applicable to all current switches we are aware of, including switches that implement the popular OpenFlow protocol.

Overall, this paper delivers a design for a high-level network programming language; a novel, general-purpose compilation algorithm based on reactive specialization; a run-time system based on a SDN architecture; the first formal semantics and proofs of correctness in this domain; and an implementation and evaluation that demonstrates the benefits over the current state-of-the-art.

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 the statistics gathered from the data plane and the results computed using the device's specialized algorithms, the control plane installs or uninstalls new 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, six companies, 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 [20]. The foundation's proposal is extraordinarily simple: *eliminate the control plane from network devices*. Rather than baking specific control software into each device, support a standard protocol that allows a separate, general-purpose machine called a *controller* 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 Facebook and 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, a number of commercial switch vendors support OpenFlow [17], a concrete realization of the switch-controller protocol required for implementing SDNs, and many 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, and host mobility [8, 12–14, 19, 27].

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*, and 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 in-

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

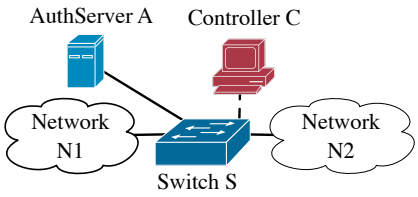

Figure 1. Example topology.

frastructure used to implement them. We can have major impact, making future networks more secure, reliable, and efficient.

As a step toward carrying out this agenda, we propose a high-level programming language called NCore, the *Network Core Programming Language*, for expressing packet-forwarding policies and statistics queries. NCore has an intuitive syntax based on familiar set-theoretic operations, which allows programmers to construct (and reason about!) rich policies in a natural way. NCore's primitives include packet predicates, which are relatively easy to compile to conventional switch hardware, as well as black-box functions, which may depend on the state of the controller. Such functions make it possible to describe complicated, dynamic policies such as authentication and load balancing. Although they are not supported natively on any standard switches, black-box functions can be implemented in SDNs with help from the controller.

The NCore compiler analyzes programs and automatically divides them into two pieces: one that runs on the switches and another that runs on the controller. The main goal of the compiler is to arrange for as much packet processing as possible to occur on the switches because packets redirected to the controller are slowed down by several orders of magnitude. However, as suggested above, not all NCore program parts can be compiled into static packet-processing rules for switches—some, like authentication and load balancing, are fundamentally dynamic. To cope with these programs, we propose a new technique, called *reactive specialization*, that dynamically refines the NCore program using the traffic seen at run time. The additional information obtained from this refinement allows the run-time system to generate and install new, specialized packet-processing rules on switches. Hence, over time, more and more rules are added to the switches and less and less traffic is diverted to the controller.

Our strategy is inspired by the idiom commonly used for SDN applications today [12, 13, 28], in which an event-driven program manually installs a rule to handle future traffic every time a packet is diverted to the controller. However, we improve on it by (1) 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, (2) synthesizing efficient forwarding rules that exploit the capabilities of modern switches (*e.g.*, wildcard rules implemented using TCAMs), and (3) providing high-level abstractions that obviate the need for programmers to deal with the low-level details of individual switches.

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:

- We define a denotational semantics for NCore (Section 3) and model the interaction between the NCore run-time system and the network in a process calculus style (Section 4). Despite the huge industrial and academic momentum surrounding SDNs such as OpenFlow, no one has formalized the low-level details of how a controller system interacts with switches yet.
- We develop novel algorithms for compiling network programs, including *algebraic classifier generation* and *reactive specialization* (Section 5).
- We prove key correctness theorems concerning our compiler and run-time system (Section 6), establishing simulation relations between our low-level, distributed implementation strategy and our high-level NCore semantics. We also prove important *progress* and *quiescence* theorems, which show that our implementation successfully moves computation off of the controller and onto switches.
- We describe a prototype implementation and an evaluation on some simple benchmarks demonstrating the practical utility of our framework (Section 7).

2. SDN and NCore Overview

This section presents additional background on SDN and NCore, using a running example to illustrate the main ideas. For concreteness, we focus on the OpenFlow SDN architecture [21] (eliding and taking liberties with some of its inessential details). Note however, that our compiler is designed to be general, and 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 the 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* on each switch comprising a list of *rules* that can be used to process packets directly on the switch, without having to send them to the controller.

Each forwarding rule has a *pattern* that identifies a set of packets, a list of *actions* that specifies how packets matching the pattern should be processed, a *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 each of the actions listed in the rule to the packet. OpenFlow supports two kinds of actions: `modify`(h, n), which sets the value of header field h to n, and `output`(l), which forwards the packet to an adjacent network location l. A *location* l is either the name of a switch, network, or host, or the special location `controller`.[1]

NCore Example. Suppose that we want to build a security application that implements in-network authentication for the network 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 A or in N_2, and from A and N_2 back to N_1 (although not from N_2 to A). We will assume that once a host is authenticated, it is authenticated for all time.

This policy can be described succinctly in NCore 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 auth_bb
    --> [Forward (Network 2)])
\/ ((InPort <: Network 1) /\  neg (inspect auth_bb)
    --> [Forward (AuthServer A)])
\/ ((InPort <: AuthServer A) \/ (InPort <: Network 3)
    --> [Forward (Network 1)])
```

Intuitively, this policy can be read as a function from triples consisting of the controller state, a switch, and a packet, to sets of actions. Formally, it is structured as the disjunction of three clauses each consisting of a *predicate* describing a set of packets (at a particular switch and controller state) and a set of *actions*, written concretely as a list. The first states that packets arriving from the internal network (`InPort <: Network 1`) should be forwarded to the external network (`Forward (Network 2)`) if `auth_bb` holds in the current state. The `inspect` keyword wraps a user-defined predicate, in this instance, `auth_bb`, which tests whether the host listed as the source in the packet header is authenticated. Similarly, the second clause states that packets from the internal network should be forwarded to the authentication server if `auth_bb` does not hold. The final clause states that packets from the authentication server and external network should be forwarded to the internal network.

Besides specifying the forwarding behavior of the network, administrators often need to monitor it for security purposes, or for making decisions about load balancing, energy efficiency, and billing. To keep things simple, in this example we will monitor the traffic from the authentication server, aggregating the results by destination host, as well as the non-web traffic (*i.e.*, not port 80) from internal hosts to the external network, aggregated by source host. We can define a query in NCore that monitors these traffic statistics, using *keys* to keep track of each result. For the first, we introduce a key `auth` and assign it the indexed type Key[{dstmac}], which indicates that only the dstmac field from each packet should be included in the result. Likewise, for the second query, we call the key `traf` and associate it with the type Key[{srcmac}]. Using these keys, the query itself can be defined as follows:

```
   (auth <-- (InPort <: AuthServer A))
\/ (traf <-- (InPort <: Network 1) /\ neg (DstPort <: 80))
```

The overall program consists of the forwarding policy and the query.

Unfortunately, although these programs are easy to write, they are far from easy to implement—at least, not in an efficient way. Compiling these programs to an efficient distributed implementation on top of an OpenFlow-like architecture requires solving two challenging problems.

Challenge 1: Dynamic program partitioning. OpenFlow's architecture is asymmetric—the controller and the switches have vastly different capabilities. To implement programs efficiently, the compiler must maximize the amount of processing done on the fast (but computationally limited) switches and minimize the amount done on the slow (but omnipotent) controller. In general, it cannot simply push the entire computation onto the switches because the switch hardware is fundamentally limited—it can match bit patterns, maintain counters, and execute modify/forward actions, but it cannot directly evaluate functions like `auth_bb` or collect statistics like the total amount of non-web traffic per host (without several billion rules). Hence, a compiler and run-time system for an SDN must solve the fundamental problem of determining what parts of the overall computation to execute where.

In general, as programs execute, they gain information that can be exploited to push work onto the switches. Thus, although the complete space of possible packets is huge, the actual set of packets that will appear on any given network is far smaller, and those packets are also highly correlated with each other—a packet from one host is likely to be followed by many more packets from the same host. Thus, although it would be impractical for the controller to push 4.3 billion rules out to the switches in advance (one for each possible IP address) to implement a fine-grained query such as the one that counts per-host non-web traffic, it is feasible for the controller to react to new packets from a particular host and dynamically generate a new classifier that tabulates statistics for that host using the counters on the switch. Similarly, although the controller cannot evaluate an arbitrary function like `auth_bb` on every possible packet in advance, when a packet arrives at the controller, it can evaluate the function on that particular packet, and, if the result is known to be fixed (*i.e. invariant*) on some set of similar packets, then it can generate a new classifier that handles future packets in that set directly on the switch.

The idea of dynamically reacting to traffic seen in the network is not new—it is used in all SDN applications (*e.g.*, [12, 13, 28]). However, in current systems, the programs that dynamically react to packets are crafted manually by programmers. For example, in NOX [11], a popular OpenFlow controller platform, programs are written in an event-driven style and explicitly react to the arrival of packets at the controller by running imperative routines that install or uninstall individual rules on switches. In contrast, examining the NCore example presented above, the reader will notice that there are no explicit calls to install or uninstall rules; there is not even a separate notion of switch code and controller code. There is just one high-level, centralized, declarative specification of the policy and queries. Hence, the compiler and run-time system must figure how to distribute the computation across the network efficiently, incrementally, and adaptively over time.

Solution 1: Reactive Specialization. *Reactive specialization* is an implementation strategy for NCore that automates the construction of the kinds of reactive programs that SDN programmers currently write by hand. This frees programmers from having to worry about the low-level details of when to install or uninstall rules on switches and—by construction—prevents them from making mistakes in doing so.

When a packet arrives at the controller, the NCore run-time automatically refines the program, using the packet as a guide. For example, returning to the authentication program, if a packet `p` from source MAC `m` arrives at the controller from an authenticated host, then `auth_bb` holds, and, moreover, will continue to hold in the future. Using this information, the compiler transforms the `inspect auth_bb` predicate into a disjunction of two predicates:

```
(inspect auth_bb) \/ (SrcMac <: MAC m)
```

After applying this program transformation, the compiler has sufficient information to generate and install new rules on the switch that handle all future traffic involving `m`.

This program transformation is enabled by the observation that each host goes through two distinct phases: unauthenticated and authenticated. When the host is unauthenticated, the forwarding policy can only be determined by the controller because it might change (and the switches do not have the computational facilities necessary to recognize the change). But when it is authenticated, the effective forwarding policy for that host suddenly becomes *consistent* and *determinate*—no further state-based decisions are needed and all future packets can be forwarded directly by switches to their intended destination. Thus, the controller can safely install rules that do this.

Challenge 2: Switch rule selection. A second challenge that makes compilation difficult is the large gap between the languages used to express network policies and switch-level classifiers. With that in mind, consider the following policy:[2]

[2] We use standard conventions when writing IP prefixes in our code—*e.g.*, `10.0.0.0/8` denotes the network of IP addresses whose first octet is `10`.

```
(SrcAddr <: IP "10.0.0.0/8") //
(SrcAddr <: IP "10.0.0.1" \/ DstPort <: Port 80)
--> [Forward (Switch 1)]
```

It states that all packets from sources in the network identified by `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 `//` operator denotes set-theoretic difference). 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 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 classifier, which has the highest priority rule placed first, implements this policy:

```
SrcAddr 10.0.0.1   : []
DstPort 80         : []
SrcAddr 10.0.0.0/8 : [Forward 1]
```

Next consider a similar policy:

```
(SrcAddr <: IP "10.2.0.0/16") //
(SrcAddr <: IP "10.2.0.1" \/ DstPort <: Port 22)
--> [Forward (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 : [Forward 2]
```

Now suppose that we then 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 compiled 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 : [Forward 2]
SrcAddr 10.2.0.1                : [Forward 1]
SrcAddr 10.2.0.0/16, DstPort 22 : [Forward 1]
SrcAddr 10.2.0.0/16             : [Forward 1,Forward 2]
SrcAddr 10.0.0.1                : []
SrcAddr 10.0.0.0/8,  DstPort 80 : []
SrcAddr 10.0.0.0/8              : [Forward 1]
```

Dealing with these complexities often leads SDN programmers to use *exact-match* rules—*i.e.*, rules that specify a value for *every single header field*, without using larger rules whose patterns contain wildcards like `10.0.0.0/8` or leave some header fields unconstrained. As interactions between exact-match rules are easy to reason about, composing policies is straightforward. But, exact-match rules, which match narrow bands of traffic are far less efficient than wildcard rules. The results of our experiments, presented in Section 7, highlight the magnitude of the difference.

Solution 2: Algebraic classifier generation. Our solution to the second challenge is a general algorithm that automates the task of generating classifiers. Given an NCore program, this algorithm constructs a low-level classifier that implements the program as fully and efficiently as possible. The classifier is efficient because the algorithm automatically selects and overlays wildcard rules whenever possible. However, because of the presence of black box functions and unimplementable queries, some of the rules will send packets to the controller for further processing instead of sending them directly to their destinations.

Importantly, our classifier generation algorithm interacts harmoniously with reactive specialization. This is critical because at run time, the output of the latter is supplied as the input to the former. Thus, the classifier generation algorithm must be powerful enough to extract new information from the program produced by reactive specialization and generate new classifiers that send fewer packets back to the controller. The theoretical analysis presented in Section 6, especially the *progress* and *quiescence* theorems, demonstrates that our algorithms interoperate harmoniously.

3. A Core Calculus for Network Programming

This section defines the syntax and semantics of NCore, a core calculus for high-level network programming. The calculus has three major components: *predicates*, which describe sets of packets; *policies*, which associate high-level *actions* with the predicates that trigger them; and *queries*, which describe various statistics about the packets flowing through the network. 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 of such objects, the overlined and capitalized version ($\overline{P}$) for multisets of objects, and vector notation ($\vec{p}$) for sequences of objects. To convert a given set (P) to the corresponding multiset, we simply place a line over the metavariable ($\overline{P}$). We will use a name written in a sans-serif font to the left of the grammar (Packet) as the type of the objects and the name enclosed in curly braces ({Packet}) as the type of sets of objects. Many of our types T are indexed by a set of headers H. We write such types $\mathsf{T}[H]$. We write finite maps using the notation $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ and we lookup elements of a map m using function application $m(x_i)$. We write multisets as $\{\!|x_1, \ldots, x_n|\!\}$ and take the union of two multisets M_1 and M_2 using the notation $M_1 \uplus M_2$. We use the usual operations $\cup$, $\cap$, $\neg$ and $\setminus$ for the operations of union, intersection, complement and difference on sets. 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.

Network Values. For simplicity, we only model a single kind of location in our theory, switches s. Packets p are the basic values processed by programs. A *complete packet* has ℓ header fields, h_1 to h_ℓ. We represent a packet as a finite map from these headers to bitstrings b and 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. A *subpacket* sp is like a packet, but is only defined on a subset H of the possible headers. We write $p[H]$ for the subpacket obtained by restricting p to the headers in H and $\mathsf{Packet}[H]$ for the type of subpackets defined on H. A transmission t represents a packet in flight and is represented using the constructor $\mathbb{T}$.

Predicates. Informally, a predicate defines the conditions under which actions or queries are executed. Formally, a predicate e defines a set of *snapshots* x consisting of a controller state Σ, switch s, and packet p. The state and switch components are essential for modeling predicates defined in terms of black-box functions, such as the `auth_bb` function from Section 2. 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 uninteresting.

Figure 3 defines the semantics of predicates. Basic predicates have the form $h : \vec{w}$, where $\vec{w}$ is a vector of wildcard bits. A bit b *matches* a wildcard bit w, written $b \sqsubseteq w$, if w is equal to b or

Network Values

$$\begin{array}{rrcl} \textsf{Switch} & s & & \\ \textsf{Header} & h & & \\ \textsf{Bit} & b & ::= & 1 \mid 0 \\ \textsf{Packet} & p & ::= & \{h_1 \mapsto \vec{b}_1, \ldots, h_\ell \mapsto \vec{b}_\ell\} \\ \textsf{Packet[H]} & sp & ::= & \{h_1 \mapsto \vec{b}_1, \ldots, h_k \mapsto \vec{b}_k\} \\ \textsf{Transmission} & t & ::= & \mathbb{T}\,(s \mid p) \end{array}$$

Predicates

$$\begin{array}{rrcl} \textsf{Snapshot} & x & ::= & (\Sigma, s, p) \\ \textsf{WildcardBit} & w & ::= & 1 \mid 0 \mid ? \\ \textsf{Inspector} & i & \in & \textsf{Snapshot}[H] \to \textsf{Bool} \\ \textsf{Predicate} & e & ::= & h : \vec{w} \mid \textsf{switch}\ s \mid \textsf{inspect}\ i \mid e_1 \cap e_2 \mid \neg e \end{array}$$

Policies

$$\begin{array}{rrcl} \textsf{Action} & a & ::= & \textsf{forward}\ s \mid \textsf{modify}\ h \leftarrow \vec{b}\ \textsf{in}\ A \\ \{\textsf{Action}\} & A & ::= & \{a_1, \ldots, a_n\} \\ \textsf{Policy} & \tau & ::= & e \to A \mid \tau_1 \cap \tau_2 \mid \neg\tau \end{array}$$

Queries

$$\begin{array}{rrcl} \textsf{Key} & k & \in & \textsf{Key}[H] \\ \{\textsf{Key}\} & K & ::= & \{k_1, \ldots, k_n\} \\ \textsf{PreState} & \sigma & ::= & \{(s_1, k_1) \mapsto \mathit{SP}_1, \ldots, (s_n, k_n) \mapsto \mathit{SP}_n\} \\ \textsf{State} & \Sigma & ::= & \{(s_1, k_1) \mapsto \overline{\mathit{SP}}_1, \ldots, (s_n, k_n) \mapsto \overline{\mathit{SP}}_n\} \\ \textsf{Query} & \phi & ::= & K \leftarrow e \mid \phi_1 \cap \phi_2 \mid \neg\phi \end{array}$$

Programs

$$\textsf{Program}\quad \pi ::= (\tau, \phi)$$

Figure 2. NCore syntax.

?. A vector of bits $\vec{b}$ *matches* a vector of wildcard bits $\vec{w}$, written $\vec{b} \sqsubseteq \vec{w}$, if each bit in $\vec{b}$ matches the corresponding wildcard bit in $\vec{w}$. A packet p matches a primitive predicate $h : \vec{w}$ if the h header of p (that is, $p(h)$) matches $\vec{w}$. Another basic predicate, switch s, denotes snapshots x containing s, with arbitrary state and packet components.

Black-box predicates, written inspect i, use an arbitrary function i to determine whether a snapshot x matches the predicate. The behavior of inspectors is constrained in two essential ways, which provides the compiler with sufficient information to generate switch-level rules. First, if i has type $\textsf{Snapshot}[H] \to \textsf{Bool}$, then the inspector may only examine the headers H of the packet contained in x. Second, each inspector has a (programmer supplied) *invariance oracle* that determines if the behavior of the inspector on a future snapshot with the same packet and switch will be the same as for x. More formally, an inspector i is *invariant on* (Σ, s, p), written $\textsf{invariant}\,((\Sigma, s, p), i)$, if for all Σ', we have $i(\Sigma \uplus \Sigma', s, p) = i(\Sigma, s, p)$, where $\Sigma \uplus \Sigma'$ is multiset union applied pointwise to the underlying multisets contained within each state. As an example, the invariance oracle for the `auth_bb` inspector defined in Section 2 can be defined by the following function in Haskell notation:

```
\(state,s,p) ->
    is_auth state p || (dstaddr p) == authsrv_addr
```

This oracle promises that the `auth_bb` function will return the same answer on all packets coming from authorized hosts or going to the authentication server. Together, the inspector type and invariance oracle allow the compiler to generate effective switch-level rules even though the inspector function itself cannot be analyzed.

$[\![e]\!] \in \{\textsf{Snapshot}\}$

$$\begin{aligned} [\![h : \vec{w}]\!] &= \{(\Sigma, s, p) \mid p(h) \sqsubseteq \vec{w}\} \\ [\![\textsf{switch}\ s']\!] &= \{(\Sigma, s, p) \mid s' = s\} \\ [\![\textsf{inspect}\ i]\!] &= \{(\Sigma, s, p) \mid i(\Sigma, s, p)\} \\ [\![e_1 \cap e_2]\!] &= [\![e_1]\!] \cap [\![e_2]\!] \\ [\![\neg e]\!] &= \neg\,[\![e]\!] \end{aligned}$$

$[\![A]\!] \in \textsf{Packet} \to \{\textsf{Transmission}\}$

$$[\![\{a_1, \ldots, a_n\}]\!]\,(p) = T_1 \cup \cdots \cup T_n$$

$$\text{where } T_i = \begin{cases} \{\mathbb{T}\,(s \mid p)\} & \text{if } a_i = \textsf{forward}\ s \\ [\![A]\!]\,(p[h \mapsto \vec{b}]) & \text{if } a_i = \textsf{modify}\ h \leftarrow \vec{b}\ \textsf{in}\ A \end{cases}$$

$[\![\tau]\!] \in \textsf{Snapshot} \to \{\textsf{Transmission}\}$

$$\begin{aligned} [\![e \to A]\!]\,(\Sigma, s, p) &= \begin{cases} [\![A]\!]\,(p) & \text{if } (\Sigma, s, p) \in [\![e]\!] \\ \varnothing & \text{otherwise} \end{cases} \\ [\![\tau_1 \cap \tau_2]\!]\,(\Sigma, s, p) &= [\![\tau_1]\!]\,(\Sigma, s, p) \cap [\![\tau_2]\!]\,(\Sigma, s, p) \\ [\![\neg\tau]\!]\,(\Sigma, s, p) &= \neg\,[\![\tau]\!]\,(\Sigma, s, p) \end{aligned}$$

$[\![K]\!] \in \textsf{Snapshot} \to \textsf{PreState}$

$$[\![\{k_1, \ldots, k_n\}]\!]\,(\Sigma, s, p) = \{(s, k_i) \mapsto \{p[H_i]\} \mid 1 \le i \le n\} \quad \text{where } k_i \in \textsf{Key}[H_i]$$

$[\![\phi]\!] \in \textsf{Snapshot} \to \textsf{PreState}$

$$\begin{aligned} [\![K \leftarrow e]\!]\,(x) &= \begin{cases} [\![K]\!]\,(x) & \text{if } x \in [\![e]\!] \\ \varnothing & \text{otherwise} \end{cases} \\ [\![\phi_1 \cap \phi_2]\!]\,(x) &= [\![\phi_2]\!]\,(x) \cap [\![\phi_2]\!]\,(x) \\ [\![\neg\phi]\!]\,(x) &= \neg\,[\![\phi]\!]\,(x) \end{aligned}$$

Figure 3. NCore semantics.

Complex predicates are built up from basic ones using the intersection and complement operators, which are interpreted set theoretically. Although this predicate language is minimal, it is straightforward to define additional, familiar operators—*e.g.*, True can be defined as $h : ?\cdots?$ (with k question marks) for a specific k-width field h, False as $\neg\textsf{True}$, and $e_1 \cup e_2$ as $\neg(\neg e_1 \cap \neg e_2)$.

Policies. Policies τ specify how packets should be forwarded through the network. Basic policies, written $e \to A$, associate a set of actions A with a predicate e. If (Σ, s, p) matches e, where Σ is the current controller state, then this policy states that actions A should be applied to p at s. An action may either be a simple forwarding action $\textsf{forward}\ s'$ that sends the packet to another switch s', or a modification action $(\textsf{modify}\ h \leftarrow \vec{b}\ \textsf{in}\ a)$, which overwrites the value in header h with the vector $\vec{b}$ and executes the actions A with respect to the modified packet. As with predicates, we build complex policies by combining simple policies using intersection and negation. Although negation is not useful on its own, as it can cause the machine to send every possible packet to every possible switch, we include it in the calculus for reasons of symmetry with predicates, and likewise can use it to encode useful operators on policies such as union and difference. Figure 3 defines the semantics of policies formally, as a function from snapshots x to sets of transmissions T.

Note that although the policy language is syntactically simple, it is remarkably expressive. In particular, inspectors, which are arbitrary functions that possibly depend upon state, are a powerful tool that can be used to express a wide range of policies. For example, it is easy to define policies that implement randomized and round-robin load balancing, fine-grained access control, and many standard routing protocols.

Queries. A query ϕ *selects* a set of packets, *projects* out certain specified fields of those packets, yielding subpackets, and *aggregates* the subpackets under a specified set of *keys*. Basic queries $(K \leftarrow e)$ select packets that match e and associate them under each key in K. The projection is governed by the type of the key—if k belongs to the set K and k has type $\mathsf{Key}[H]$, then $p[H]$ is stored under k. In definitions, we write $k \in \mathsf{Key}[H]$ to indicate the type of k and $K \in \mathsf{KeySet}[H]$ where H is the union of the sets of headers H_i associated with each key $k_i \in K$. Figure 3 defines the semantics of queries as a function from snapshots to *pre-states*, which are maps from key-switch pairs to sets of subpackets. (*States*, used later to aggregate the results of queries, are maps from key-switch pairs to *multisets* of subpackets.) We interpret the operators $\cap$ and $\neg$ on pre-states point-wise.

Programs. An NCore program π consists of a pair of a policy τ and query ϕ. The semantics of a program on a given snapshot x is defined in the obvious way: $[\![(\tau,\phi)]\!](x) = ([\![\tau]\!](x), [\![\phi]\!](x))$.

To understand how such a program executes over time, we define two abstract machines. Both machines forwards packets according to τ and update the controller state using ϕ, but they differ in how often the switches synchronize with the controller. The *synchronous machine* defines an idealized, synchronous implementation of an NCore program. It evaluates the query and policy on each packet immediately and instantaneously. 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. The *asynchronous machine* defines a more realistic implementation. Like the first machine, it is *policy-compliant*—it always forwards packets according to the program policy—but it updates the state asynchronously rather than in lockstep with every packet processed. 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, a reasonable implementation should be policy compliant, but users should not expect perfect synchrony because the cost of implementing it would be prohibitive. In practice, synchronization typically happens at periodic, timed intervals (modulo the variance in the latency of communication) but for simplicity, we do not model time explicitly in our formal system.

Figure 4 defines both reference machines formally. The state of the synchronous machine includes the NCore program π, the state Σ, and a multiset $\overline{T}$ of pending transmissions. At each step, it removes a transmission $\overline{T}$, processes it using the policy and query, and updates the machine state and transmissions with the new state and transmissions generated by the program. The state of the asynchronous machine includes the program π, state Σ, and two multisets of transmissions: $\overline{T}_1$, which represents transmissions waiting to be processed by the policy, and $\overline{T}_2$, which represents transmissions that have been processed by the policy but not yet by the query. The first inference rule for the second machine takes a transmission from $\overline{T}_1$, processes it using the policy, and places it in $\overline{T}_2$; the second rule takes a transmission in $\overline{T}_2$ and processes it using the query.

Before completing this section, we comment on one limitation of the language: the exclusive use of *permanent* invariance. There are policies 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 the period, but a rigorous investigation of this topic is beyond the scope of this paper.

$$\boxed{M_{\text{sync}} \xrightarrow{o} M'_{\text{sync}}} \quad \mathsf{SyncState}\ M_{\text{sync}} ::= (\pi, \Sigma, \overline{T})$$

$$\frac{\pi = (\tau,\phi) \qquad [\![\tau]\!](\Sigma, s, p) = T' \qquad [\![\phi]\!](\Sigma, s, p) = \sigma'}{(\pi, \Sigma, \overline{T} \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}) \xrightarrow{(s,p)} (\pi, \Sigma \uplus \overline{\sigma}', \overline{T} \uplus \overline{T}')}$$

$$\boxed{M_{\text{async}} \xrightarrow{o} M'_{\text{async}}} \quad \mathsf{AsyncState}\ M_{\text{async}} ::= (\pi, \Sigma, \overline{T}_1, \overline{T}_2)$$

$$\frac{\pi = (\tau,\phi) \qquad [\![\tau]\!](\Sigma, s, p) = T'}{(\pi, \Sigma, \overline{T}_1 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}, \overline{T}_2) \xrightarrow{(s,p)} (\pi, \Sigma, \overline{T}_1 \uplus \overline{T}', \overline{T}_2 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\})}$$

$$\frac{\pi = (\tau,\phi) \qquad [\![\phi]\!](\Sigma, s, p) = \sigma}{(\pi, \Sigma, \overline{T}_1, \overline{T}_2 \uplus \{\!|\, \mathbb{T}(s \mid p) \,|\!\}) \to (\pi, \Sigma \uplus \overline{\sigma}, \overline{T}_1, \overline{T}_2)}$$

Figure 4. Reference machines.

4. The Run-time System

The previous section defined the syntax and semantics of the high-level network programming language NCore. However, it did not discuss how to implement the semantics on a software-defined network—*i.e.*, on a distributed set of switches managed by a controller. In this section, we give an operational semantics to the NCore run-time system and the underlying network devices. This operational semantics explains the basic interactions between the controller and the switches. The next section will explain how to compile NCore programs and will use this low-level distributed system as the target of the compilation.

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 action γ. We represent classifiers as sequences as opposed to sets primarily to model the features of modern switch hardware, including priorities (rules on the left have higher priority than rules on the right). Note that classifiers are more restricted than NCore policies—*e.g.*, individual rules cannot express negations or unions directly.

The *pattern* (z) component of a rule recognizes a set of packets, and hence is similar to (but less general than) a predicate in NCore. We write $p \sqsubseteq z$ when pattern z matches packet p. We assume that patterns form a lattice with elements higher in the lattice matching more packets and elements lower in the lattice matching fewer packets. We write $z_1 \sqsubseteq z_2$ when z_1 is lower than (or equal to) another element in the lattice, thereby matching a subset (or equal set) of packets. Operators $\sqcap$ and $\sqcup$ implement meets and joins in the pattern lattice. The top element ($\top$) matches all packets; the bottom element ($\bot$) matches none. In addition to $\top$ and $\bot$ and the pattern lattice contains a number of other elements. For example, OpenFlow switches support prefix pattern matching on source and destination IP addresses (*i.e.*, patterns of the form $0110*$) but only exact or wildcard matching on other most other headers. Other switches support alternative extended patterns, such as ranges with the form $[n_1, n_2]$. We do not specify the exact features available,

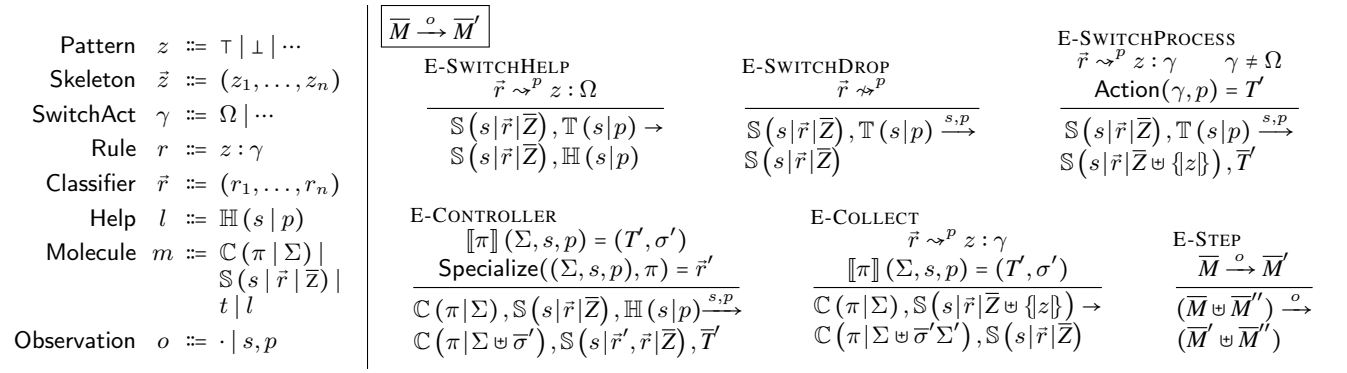

Figure 5. The run-time system.

though we require that patterns have exact meets, in the sense that if $z \sqsubseteq z_1$ and $z \sqsubseteq z_2$ then $z \sqsubseteq z_1 \sqcap z_2$.

As with patterns, we leave the *actions* γ supported by the switches abstract, as different switches support different kinds of actions. However, actions must include Ω, which sends packets to the controller. Moreover, although we do not require it formally, we expect that switches will support simple forward/modify actions. Many switches also support more sophisticated actions—*e.g.*, DevoFlow switches [18] can make conditional forwarding decisions based on whether their ports are up or down. Because switch actions γ are held abstract, we assume a function $\mathsf{Action}(\gamma, p)$ that executes an action γ on p and generates a set of transmissions T'. Typically T' will be a singleton set with a single transmission containing the packet being forwarded. However, if the policy dictates that the packet should be forwarded to multiple switches, T' will be bigger.

Given a packet p and a classifier $\vec{r}$, we match the packet against the classifer by finding the first rule whose pattern matches the packet. We write $\vec{r} \rightsquigarrow^p z : \gamma$ for the matching judgment and $\vec{r} \not\rightsquigarrow^p$ if p does not match any pattern in $\vec{r}$. More formally, we define classifier matching as follows; note that it selects the highest priority (leftmost) matching rule:

$$\frac{p \not\sqsubseteq z_1 \quad \cdots \quad p \not\sqsubseteq z_{i-1} \quad p \sqsubseteq z_i}{(z_1 : \gamma_1, \ldots, z_{i-1} : \gamma_{i-1}, z_i : \gamma_i, \ldots, z_n : \gamma_n) \rightsquigarrow^p z_i : \gamma_i}$$

The Molecular Machine. Now we are ready to define the runtime system itself. We formalize its operational semantics as a *molecular machine*, in the style of Berry and Boudol's chemical abstract machine [2]. The machine's components, called *molecules*, are given on the left side of Figure 5.

The molecule $\mathbb{C}(\pi \mid \Sigma)$ represents the controller machine running the NCore program π in state Σ. The molecule $\mathbb{S}(s \mid \vec{r} \mid \overline{Z})$ represents switch s with packet classifier $\vec{r}$ and local switch state $\overline{Z}$. 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 and elegance, we represent these integers in unary (using a multiset of patterns) in our formal system. A transmission molecule $\mathbb{T}(s \mid p)$ represents a packet p en route to switch 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 machine's operational semantics is defined on the right side of Figure 5. To lighten the notation, in this figure, we use the notation m_1, m_2 as shorthand for $\{\!|m_1, m_2|\!\}$. Each operational rule may optionally be labelled with an *observation* o, which records when transmissions are processed.

The first three rules, E-SwitchProcess, E-SwitchHelp and E-SwitchDrop, model the work done by switches to process packets. There are three possibilities: either (1) the packet matches a rule with a non-controller action and the switch executes the action, (2) the packet matches a rule with a controller action and the switch generates a help molecule, or (3) the packet does not match any rule and is dropped.

The rule E-Controller models the work done by the controller to process help molecules. It first interprets the packet using its NCore program, generating new transmissions T' and new state Σ'. It then uses the additional information contained in the packet to construct new *reactive rules* $\vec{r}'$ for the switch that generated the help molecule. These new rules typically allow the switch to process future packets locally, in the fast path, without requesting further help from the controller. We keep the definitions of the specific algorithms used to specialize the program and generate reactive rules abstract for now; they will be described in the next section.

The rule E-Collect models the work done by the controller to collect statistics about a particular switch-level rule. First, the switch picks a pattern z in the switch state to send to the controller. Next, the controller performs one step of processing using the query ϕ.[3] Because switches do not store actual packets in their state, only patterns, the controller must construct a packet p that matches z to evaluate the query. We use the judgement $\vec{r} \rightsquigarrow^p z : \gamma$ to represent fabrication of an appropriate packet and then appeal to the semantics of queries in the ordinary way. Finally, the controller stores the query result in its state. This rule places a subtle, but crucial constraint on classifiers: because we do not know which packet p' originally matched z on the switch, for correctness, the results of query evaluation must be the same no matter what packet p the controller choses. To put it another way, if the rules placed on the switch are too broad and match too many different kinds of packets, then the results produced by E-Collect will be wrong.

5. The NCore Compiler

The NCore compiler performs two distinct tasks:

- *Classifier Generation:* given an NCore program, it builds a collection of classifiers, one for each switch in the network.
- *Reactive Specialization:* given a packet not handled by the current classifier installed on a switch, it generates additional rules that allow the switch to handle future packets with similar header fields efficiently, without consulting the controller.

[3] In practice, the controller would collect *all* of the statistics for a given rule at once; this can be modeled in our formal system by evaluating multiple E-Collect steps in sequence.

$$\boxed{\mathcal{S}(s,e) = \vec{z}}$$

$$\begin{aligned}
\mathcal{S}(s, h:\vec{w}) &= \mathcal{O}^{\mathsf{compile}}(h,\vec{w}), \top \\
\mathcal{S}(s, \mathsf{inspect}\ i) &= \top \\
\mathcal{S}(s, \mathsf{switch}\ s') &= \top \\
\mathcal{S}(s, e_1 \cap e_2) &= \mathcal{S}(s,e_1) \times \mathcal{S}(s,e_2) \\
\mathcal{S}(s, \neg e) &= \mathcal{S}(s,e)
\end{aligned}$$

$$\boxed{\mathcal{S}(s,\tau) = \vec{z}}$$

$$\begin{aligned}
\mathcal{S}(s, e \to A) &= \mathcal{S}(s,e) \\
\mathcal{S}(s, \tau_1 \cap \tau_2) &= \mathcal{S}(s,\tau_1) \times \mathcal{S}(s,\tau_2) \\
\mathcal{S}(s, \neg\tau) &= \mathcal{S}(s,\tau)
\end{aligned}$$

$$\boxed{\mathcal{S}(s,\phi) = \vec{z}}$$

$$\begin{aligned}
\mathcal{S}(s, K \leftarrow e) &= \mathcal{S}(s,e) \\
\mathcal{S}(s, \phi_1 \cap \phi_2) &= \mathcal{S}(s,\phi_1) \times \mathcal{S}(s,\phi_2) \\
\mathcal{S}(s, \neg\phi) &= \mathcal{S}(s,\phi)
\end{aligned}$$

$$\boxed{\mathcal{S}(s,\pi) = \vec{z}} \qquad \mathcal{S}(s,(\tau,\phi)) = \mathcal{S}(s,\tau) \times \mathcal{S}(s,\phi)$$

Figure 6. Skeleton Generation.

This section describes the key algorithms that implement these tasks. Due to space constraints, we focus on predicate compilation and, to a lesser extent, policy compilation. We discuss formal properties of the compiler in Section 6. Many of the definitions in this section are parameterized on abstract functions called *oracles*. Parameterizing the compiler in this way makes it independent of the details of the rapidly evolving SDN specifications.

5.1 Classifier Generation

The NCore classifier generator works in two phases. The first phase, *skeleton generation*, builds a sequence of switch patterns that form the backbone of the final classifier. The second, *action identification*, attaches a switch action γ to each pattern in the skeleton, using *Consistent Determinate Action Analysis* (CDAA) to select the action for each rule.

Skeleton Generation. A *skeleton* is half of a classifier that comprises a sequence of patterns. Intuitively, a "good" skeleton for a given policy is one where every pattern has a consistent and determinate action in the policy. Given a skeleton with this property, we can easily build a classifier by simply attaching the appropriate action to each pattern. If, however, the skeleton contains patterns for which the policy does not have a consistent and determinate action, then we must associate those patterns with the controller action Ω, which causes packets to be diverted to the controller. This intuition about what makes a good skeleton cuts to the heart of why many OpenFlow programmers write programs using exact-match rules: Analyzing a classifier to identify consistent, determinate actions is easily done by hand when the classifier only contains exact-match rules. But when the classifier contains wildcard rules, the analysis becomes much more difficult. Thus, like register allocation, this analysis is better suited for computer processing than human processing.

Figure 6 presents the skeleton generation algorithm, which is formalized as a switch-indexed function $\mathcal{S}(s,\cdot)$ from NCore programs to skeletons for s. One of the invariants of the algorithm is that it always generates a *complete skeleton*—one that matches all packets. Within this complete skeleton, the algorithm attempts to produce a pattern that matches the same set of packets as the predicate being compiled (relative to the patterns appearing before it in the classifier), and another set of patterns whose domain matches the complement of the predicate. While the algorithm attempts to generate skeletons with these properties, it does not always succeed, for two fundamental reasons: (1) it cannot analyze the decisions made by black-box inspectors—their results are indeterminate, and (2) some predicates cannot be expressed practically (*i.e.*, without exponential blowup) in terms of switch patterns. For instance, building a good skeleton for a query that collects statistics on a per-source-IP (or per-destination-MAC, per-TCP port, *etc.*) basis would require generating distinct patterns for each of the approximately 4.3 billion IP addresses, which is clearly impractical. Nevertheless, in many common cases it successfully generates a good skeleton.

The first definition in Figure 6 generates skeletons for primitive predicates $h : \vec{w}$ by invoking an oracle $\mathcal{O}^{\mathsf{compile}}$. To handle cases where the switch hardware is more limited than these primitives,[4] we allow the skeleton to *overapproximate* the semantics of the primitive. However, when the skeleton is an overapproximation, the pattern in question will be associated with the Ω action in the final classifer (the controller will resolve the pattern precisely). Note that the skeletons generated for primitives are terminated by $\top$. This establishes both properties mentioned above—completeness and the existence of complementary patterns.

The rule for compiling an inspector simply returns the $\top$ pattern—it is impossible to analyze the properties of the black-box function to generate an accurate skeleton, so the rule overapproximates. Later in this section, we will see how reactive specialization allows us to overcome the initial ineffectiveness of this pattern.

The rule for compiling switch predicates is identical to the rule for inspectors, but the rationale is entirely different. If the switch name s' equals s, then the predicate $\mathsf{switch}\ s'$ matches all packets on s so the rule returns the $\top$ pattern. If the two switch names are not equal, then the predicate matches no packets. The skeleton generation algorithm still returns $\top$—the pattern for the predicate itself would be $\bot$, with $\top$ as the complement, but $\bot$ may always be omitted as it does not match any packets.

The rule for intersections uses a cross-product operator, defined as follows, where $\bigoplus_{i=1}^{m}$ denotes the sequence with elements indexed by i ranging from 1 to m:

$$(z_1,\ldots,z_m) \times \left(z'_1,\ldots,z'_n\right) = \bigoplus_{i=1}^{m}\bigoplus_{i=1}^{n} z_i \sqcap z'_j$$

One can think of the cross product as the "intersection" of two skeletons. We illustrate this property with an example, if the two patterns being combined are wildcard strings, the cross-product of the skeletons $(1?,\ ??)$ and $(10,\ 01,\ ??)$ is calculated as follows:

$$\begin{aligned}
& (1? \sqcap 10,\ 1? \sqcap 01,\ 1? \sqcap ??,\ ?? \sqcap 10,\ ?? \sqcap 01,\ ?? \sqcap ??) \\
= \ & (10,\ \bot,\ 1?,\ 10,\ 01,\ ??) \\
\equiv \ & (10,\ 1?,\ 01,\ ??)\,.
\end{aligned}$$

Consider the relationship between the pattern $1?$ in the first skeleton and the pattern $??$ in the second skeleton. The set of wildcard strings that simultaneously matches $1?$ and $??$ in the context of their respective skeletons is $\{11\}$ (note that 10 is matched in the second skeleton before one can reach $??$). Likewise, the meet of $1?$ and $??$ in the context of the cross product matches the same set. This property ensures that intersection preserves the "good" structure in skeletons.

Finally, the skeleton for a negated predicate $\neg e$ is equal to the skeleton for e. This follows from the invariant that a generated skeleton must always be complete, and the goal that a skeleton should represent the semantics of both the predicate itself and its complement.

The cases for policies, queries, and programs are mostly straightforward. The skeleton generation algorithm ignores actions and keys but generate skeletons for the embedded predicates and surrounding set-theoretic operators.

[4] This is not uncommon—although bitwise patterns are convenient for programmers, they are not supported in many cases on OpenFlow.

$$\boxed{\mathcal{A}(s,\pi,\vec{z}) = \vec{r}}$$

$$\mathcal{A}(s,\pi,\cdot) = \cdot$$

$$\mathcal{A}(s,\pi,(\vec{z},z)) = \begin{cases} \mathcal{A}(s,\pi,\vec{z}), z : \mathcal{O}^{\mathsf{switch}}(A) \\ \quad \text{if } s \mid \vec{z} \mid z \vdash \pi \rightsquigarrow (A,K) \\ \quad \text{and } K \in \mathsf{KeySet}[H] \\ \quad \text{and } H \subseteq \mathsf{headers}(z) \\ \mathcal{A}(s,\pi,\vec{z}), z : \Omega \\ \quad \text{otherwise} \end{cases}$$

Figure 7. Action identification.

Action Identification. In the second phase of classifier generation, the compiler attaches an action to each pattern in the skeleton. Most of the work needed to do this is encapsulated in the algorithm for *Consistent, Determinate Action Analysis* (CDAA), which we represent using the following judgement:

$$s \mid \vec{z} \mid z \vdash \pi \rightsquigarrow (A,K).$$

If the above holds, then the procedure has determined that for every packet p that matches z in the context of $\vec{z}$, the program π associates the sets of actions A and keys K with p on switch s.

The CDAA is partial—for example, it cannot analyze black-box inspectors. Due to space limitations, we defer the full definition of the analysis, which is mostly straightforward, to the Appendix, along with the proofs of its formal properties. The key properties we have proven are (1) soundness, and (2) that the CDAA is sufficiently complete to ensure that reactive specialization makes progress; this is the main lemma used in the next section to prove the quiescence theorem.

One other ingredient we need is a switch action oracle ($\mathcal{O}^{\mathsf{switch}}$) that translates high-level actions into switch-level actions. For most actions, the translation is direct and obvious. For instance, it is typically the identity function for simple forwarding actions.

The action identification procedure is defined in Figure 7. It considers each pattern in the skeleton, and either (1) uses CDAA to find a consistent determinate action to associate with the pattern, checks that the pattern is sufficiently fine-grained for the queries, and creates an action using the switch action oracle, or (2) concludes that either a consistent determinate action does not exist (or cannot be found yet) or the pattern is insufficiently fine-grained for the queries, and associates the controller action with the pattern. To ensure correct implementation of queries we assume a function that computes a set of headers $\mathsf{headers}(z)$ from a pattern z. The key property of this function is that if $h \in \mathsf{headers}(z)$, we know all packets $p \sqsubseteq z$ have the same value in header h.

5.2 Reactive Specialization

The classifier generation algorithm described in the preceding section builds classifiers that can be installed on switches. But as we saw, it has some serious limitations. Many policies are dynamic and use black-box inspectors that cannot be analyzed. Even for purely static policies, a set of rules that matches all packets on the switches can be large—much larger than the actual space of flows that will be inhabited in a given network. To deal with these situations, we define *reactive specialization*, a powerful generalization of the simple, reactive, microflow-based strategy implemented manually by OpenFlow programmers. We define reactive specialization as the composition of three basic operations: (1) *program refinement*, which expands the program relative to a new packet witnessed by the controller, (2) *regeneration*, which translates the expanded program to a classifier using the tools of the previous section, and (3) *reactive pruning*, which extracts new rules out of the regenerated classifier.

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

$$\mathcal{R}((\Sigma,s,p),h:\vec{w}) = h:\vec{w} \cup h : \mathcal{O}^{\mathsf{refine}}(h,\vec{w},p)$$

$$\mathcal{R}(x,\mathsf{switch}\ s) = \mathsf{switch}\ s$$

$$\mathcal{R}(x,\mathsf{inspect}\ i) = \begin{cases} \mathsf{inspect}\ i \cup e' & \text{if } \mathsf{invariant}(x,i) \wedge i(x) \\ \mathsf{inspect}\ i - e' & \text{if } \mathsf{invariant}(x,i) \wedge \neg i(x) \\ \mathsf{inspect}\ i & \text{if } \neg\, \mathsf{invariant}(x,i) \end{cases}$$

$$\text{where } i : \mathsf{Snapshot}[H] \to \mathsf{Bool}$$

$$\text{and } e' = \mathsf{similar}(x,H)$$

$$\mathcal{R}(x,e_1 \cap e_2) = \mathcal{R}(x,e_1) \cap \mathcal{R}(x,e_2)$$

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

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

$$\mathcal{R}(x,e \to A) = (\mathcal{R}(x,e)) \to A$$

$$\mathcal{R}(x,\tau_1 \cap \tau_2) = \mathcal{R}(x,\tau_1) \cap \mathcal{R}(x,\tau_2)$$

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

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

$$\mathcal{R}(x,K \leftarrow e) = K \leftarrow (e' \cup (e' \cap \mathsf{similar}(x,H)))$$

$$\text{where } K \in \mathsf{KeySet}[H]$$

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

$$\mathcal{R}(x,\phi_1 \cap \phi_2) = \mathcal{R}(x,\phi_1) \cap \mathcal{R}(x,\phi_2)$$

$$\mathcal{R}(x,\neg\phi) = \neg\, \mathcal{R}(x,\phi)$$

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

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

Figure 8. NCore refinement.

Program Refinement. When the controller receives a new packet that a switch could not handle, it interprets the program with respect to the packet, switch, and its current state. The idea in program refinement is to use that information to augment the program with additional structural information so that we can build a specialized classifier that handles future similar packets on the switch. Figure 8 defines the refinement function. The key invariant for this program transformation is that the semantics of the old and new programs are identical. However, syntactically, the new program will typically have a different structure, as the transformation uses the packet to unfold inspectors and queries. This makes skeleton generation more precise and the recompiled program more effective.

The rules for refining a predicate appear at the top of Figure 8. The first rule uses the refinement oracle $\mathcal{O}^{\mathsf{refine}}$ to refine basic predicates. Unlike the compilation oracle, which may *overapproximate* the predicate, the refinement oracle *underapproximates* it, so that the rest of the compilation infrastructure will be able to generate an implementable switch-level rule that matches the given packet. Importantly, 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 pattern. In many other cases, however, if the switch has prefix matching or wildcards, the refinement oracle will produce a rule that matches many more packets.

The second rule refines the switch predicate. Because the switch predicate already reveals the maximum amount of information, it cannot be refined further.

The rule for inspectors is the most interesting. Recall that if an inspector i is invariant with respect to a snapshot x (which includes the switch, packet, and controller state) then it will return the same boolean decision on similar packets in every future controller state. A packet is *similar* to another when it agrees on the headers H that the given inspector I examines. In such a case it is safe to augment the inspector with a predicate that is independent of the controller state, but matches similar packets in the same way as the inspector. The function $\mathsf{similar}(x, H)$ generates a predicate that matches packets similar to the packet in x on headers H:

$$\begin{aligned}\mathsf{similar}((\Sigma, s, p), H) &= \mathsf{switch}\ s \cap \textstyle\bigcap_i e_i \\ &\quad \text{where } e_i = h_i : p(h_i) \text{ for all } h_i \in H\end{aligned}$$

In the first clause for inspectors, if the inspector depends upon the headers H (as indicated by its indexed type), is invariant in the current state, and evaluates to true on the current snapshot (Σ, s, p), then it may be safely refined by taking the union of the inspector and the similarity predicate $\mathsf{similar}((\Sigma, s, p), H)$. The second inspector clause is similar except the inspector does *not* evaluate to true and hence the inspector predicate may be refined by subtracting out the similarity predicate. Finally, the third rule is used when the inspector is not invariant. In this case, no sound refinement exists—the boolean value returned by the inspector may change as the controller state changes—and so packets must still be diverted to the controller until the inspector stabilizes.

The other interesting element of refinement appears in the query translation. In particular, in the first rule, for queries $K \leftarrow e$, the algorithm expands the predicate e into e', identifies the header set H relevant to K and generates the similarity predicate e''. The query predicate refinement is the union of e' (whose semantics, by induction, is identical to e) and the intersection of e' and e''. The intersection identifies those packets similar to x that also satisfy the query predicate, and ensures that refinement preserves semantics.

Reactive pruning. In general, after the program 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 simple 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 redundant rules not generated via specialization with respect to p), or (3) overlap with a rule we removed earlier (to preserve the semantics of the rules).

5.3 Tying it all together

We now have the technical definitions needed to complete the definition of the NCore compiler and run-time system. In particular, we can now define (1) $\mathsf{GenClassifier}(\pi, s)$, the classifier generator used to build the initial switch classifiers, and (2) $\mathsf{Specialize}(x, \pi)$, the reactive specializer used by rule E-CONTROLLER to incrementally install new switch rules.

$$\frac{\vec{z} = \mathcal{S}(s, \pi) \quad \vec{r} = \mathcal{A}(s, \pi, \vec{z})}{\mathsf{GenClassifier}(\pi, s) = \vec{r}} \qquad \frac{\pi' = \mathcal{R}(x, \pi) \quad \vec{z} = \mathcal{S}(s, \pi') \quad \vec{r} = \mathcal{A}(s, \pi', \vec{z}) \quad \vec{r}' = \mathsf{prune}(\vec{r}, p)}{\mathsf{Specialize}(x, \pi) = \vec{r}'}$$

Finally, to initialize the molecular machine for NCore program π and switches s_1 to s_n, we use the following functions:

$$\begin{aligned}\mathsf{Init}(s, \pi) &= \mathbb{S}(s \mid \mathsf{GenClassifier}(s, \pi) \mid \varnothing) \\ \mathsf{Init}(\{s_1, \ldots, s_n\}, \pi) &= \mathbb{C}(\pi \mid \varnothing), \mathsf{Init}(s_1, \pi), \ldots, \mathsf{Init}(s_n, \pi)\end{aligned}$$

6. Formal Properties

This section presents the essential elements of our two central theoretical results: (1) a proof of *functional correctness* for NCore, 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 To demonstrate the correctness of the NCore 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 NCore molecular machine and the molecular machine simulates the synchronous reference machine. We state these properties formally below; the specific relations used to prove the simulations are given in the Appendix.

Theorem 1 (Asynchronous weak simulation)**.** The asynchronous machine $(\pi, \varnothing, \overline{T}, \varnothing)$ weakly simulates the molecular machine $(\mathsf{Init}(S, \pi), \overline{T})$.

Theorem 2 (Synchronous weak simulation)**.** The molecular machine $(\mathsf{Init}(S, \pi), \overline{T})$ weakly simulates the synchronous machine $(\pi, \varnothing, \overline{T})$.

As usual with simulation arguments, the key insight needed to complete the proof is finding the appropriate relations between the machines. The soundness of CDAA also plays a critical role.

Quiescence The quiescence theorem demonstrates that the NCore compiler effectively moves work off of the controller and onto switches, even when the program is expressed in terms of black-box inspectors. Formally, quiescence states that if all of the inspectors in the program are invariant, then the NCore 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. Given a multiset of runtime system molecules $\overline{M}$, the *help multiset* of $\overline{M}$, written $\mathsf{Help}(\overline{M})$, is the multiset of help molecules in $\overline{M}$. The *controller set* of $\overline{M}$, written $\mathsf{controllerPackets}(\overline{M})$ is the set of switch-packet pairs (s, p) that generate a help molecule when p is processed using the classifier currently installed on s. The first lemma, *controller set monotonicity*, states that the set of packets that require processing on the controller never increases:

Lemma 1 (Controller set monotonicity)**.** If $(\mathsf{Init}(S, \pi), \overline{T}) \rightarrow^* \overline{M}' \xrightarrow{o} \overline{M}''$ then $\mathsf{controllerPackets}(\overline{M}'') \subseteq \mathsf{controllerPackets}(\overline{M}')$.

As usual, the notation $\overline{M} \rightarrow^* \overline{M}'$ denotes the reflexive, transitive closure of the single step judgement, ignoring observations.

Next we introduce two properties a program π must have to ensure quiescence: it must be *realizable* and *fully invariant*. Intuitively, π is realizable if it does not use features that cannnot be implemented on switches—*e.g.*, in OpenFlow, matching on the payloads of packets or modifying multiple copies of a packet in different ways. Formally, we split realizability into two simpler properties: fully matchable and fully actionable. A program π is *fully matchable* if for every basic predicate $h : \vec{w}$ appearing in it and every packet p, if $\mathcal{O}^{\mathsf{refine}}(h, w, p) = w'$ and $\mathcal{O}^{\mathsf{compile}}(h, w') = z$, then for every snapshot (Σ, s, p'), we have $p' \sqsubseteq z$ implies $(\Sigma, s, p') \in [\![h : w]\!]$. A program π is *fully actionable*, if, for all switches s, skeletons $(\vec{z}, z)$, and programs π', if $[\![\pi]\!] = [\![\pi']\!]$ and $s \mid \vec{z} \mid z \vdash \pi' \rightsquigarrow (A, K)$, then $\mathcal{O}^{\mathsf{switch}}(A) \neq \Omega$. Finally, a program π is *fully invariant* on Σ if, for every switch s, packet p and predicate $\mathsf{inspect}\ i$ appearing in it we have $\mathsf{invariant}((\Sigma, s, p), i)$.

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. The full proof depends deeply on several completeness properties of CDAA; the Appendix contains further details.

Lemma 2 (Controller set progress)**.** For every realizable program π, if $(\mathsf{Init}(S,\pi),\overline{T}) \rightarrow^* (\mathbb{C}(\pi \mid \Sigma),\overline{M}') \xrightarrow{s,p} \overline{M}''$, such that π is fully invariant on Σ and $\mathsf{Help}(\overline{M}'') \subset \mathsf{Help}(\overline{M}')$, then $\mathsf{controllerPackets}(\overline{M}'') \subset \mathsf{controllerPackets}(\overline{M}')$.

Quiescence follows as a simple corollary from controller set monotonicity and progress, 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 $\overline{M}$ *may quiesce* if there exists a configuration $\overline{M}'$ such that $\overline{M} \rightarrow^* \overline{M}'$ and the rule E-CONTROLLER cannot be used in any derivation in the operational semantics starting from $\overline{M}'$. With this definition in hand, we can state quiescence as follows:

Corollary 1 (Quiescence)**.** For every realizable program π, if $(\mathsf{Init}(S,\pi),\overline{T}) \rightarrow^* (\mathbb{C}(\pi \mid \Sigma),\overline{M}')$ and π is fully invariant on Σ, then $\overline{M}'$ can quiesce.

7. Implementation and Evaluation

We have built a full working implementation of the NCore compiler in Haskell. It closely follows the formal definitions presented in this paper, modulo a few optimizations, which are described below. The core algorithms including skeleton generation, reactive specialization, and action identification are formulated in terms of type classes (*e.g.*, lattices for switch patterns) and oracle functions. This design makes it easy to instantiate the compiler to target 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 based on OpenFlow switches. The first generates the coarse-grained rules from reactive specialization described in this paper. The other, used for comparison, generates exact-match rules, which emulates the microflow-based techniques commonly used by programmers today. The total amount of code needed to define these instances was only about 200 lines of code, compared to approximately 1600 lines for the compiler itself.

Optimizations. The implementation uses a number of heuristic optimizations to avoid the combinatorial blowup that would result from doing skeleton generation and reactive specialization 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 skeleton generation algorithm identifies and removes patterns completely "shadowed" by other patterns and patterns whose effect is "covered" by a larger pattern lower down with the same actions. Finally, it uses a cheap static analysis to characterize the packets that will not be sent to the controller, which avoids invoking the more costly CDAA in many cases. Although these heuristics are simple, in our experience they go a long way toward ensuring reasonable performance.

Evaluation. To evaluate our implementation, we developed an instrumented version of the run-time system that collects statistics about the sizes of the classifiers and the amount of traffic handled on switches (as opposed to being sent 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 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) compilers on the following programs:

- **Static Policy (SP)**: implements the simple static policy described at the end of Section 2. This benchmark measures the (in)efficiency of microflow-based compilation strategies.
- **Static Policy with Query (SPQ)**: forwards packets using the same policy as in the SP test but also collects traffic statistics for each host. Due to the query, 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 compilation.
- **Black-Box Policy with Query (BPQ)**: 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 both queries and black-box functions.

To drive these experiments, we generated packets using *fs* [23], a tool that synthesizes realistic packet traces from several statistical parameters. We ran each experiment on 100K packets in total. The results are shown in Figure **??**. 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 all of the benchmarks. On the SP 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. On the SPQ benchmark, after seeing a packet from each unique host involved in the query, the full compiler generates wildcard rules (via reactive specialization) that handle all future traffic from the host—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 sometimes 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 BPQ experiment demonstrates that the full compiler generates more effective classifiers than the microflow compiler, even in the presence of black-box functions that it cannot analyze directly. Note that a large number of packets must be diverted to the controller in any correct implementation—at the start, the black-box is not invariant for any host. The difference between the two compilers starts to become clear toward the end of the experiment.

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 programmers construct reactive programs manually out of callbacks and explicit, switch-level packet-processing rules. There are numerous examples of network applications built on top of NOX using microflows [12, 13, 28], but relatively few that use wildcard rules (though Wang's load balancer [27] 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 [26] (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 NCore does.

NCore was inspired in part by Frenetic [9], a recently proposed high-level language for OpenFlow networks. NCore borrows from Frenetic the idea that network programs should be split into two parts: one for specification of forwarding policies and one for specification of queries. The major linguistic advance of NCore over Frenetic involves the inclusion of arbitrary functions in its policies and queries. In addition, NCore has a well-defined semantics, a parameterized, lattice-theoretic compilation strategy that uses the full capabilities of the switches, and proofs of correctness and quiescence of its implementation. Frenetic's implementation uses inefficient microflows, is ad hoc, tailored to OpenFlow, and has no theoretical analysis of formal properties.

Both NCore 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 NCore programs written as if the program has an omniscient, centralized view of the entire network. The NCore 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 NCore compiler is to generate efficient packet classifiers. Most previous research in this area (see Taylor [25] for a survey) focuses on static compilation. The NCore compiler generates classifiers in the face of non-static policies, with unknown black box functions, and synthesizes a distributed switch-controller implementation. Bro [22], Snortran [7], Shangri-La [5] and FPL-3E [6] compile rich packet-filtering and monitoring programs, designed to secure networks and detecting intrusions, down to special packet-processing hardware and FPGAs. The main difference between NCore 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 [24], 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, centralized host.

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. Casadod, 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 Comput. Commun. Rev.*, 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] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee. DevoFlow: Cost-effective flow management for high performance enterprise networks. In *HotNets*, pages 1:1–1:6, 2010.

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

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

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

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

[23] 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.

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

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

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

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

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

A. NCore Auxiliary Definitions and Properties

A.1 Elided Skeleton and Classifier Properties

We define packet lookup $\vec{z} \rightsquigarrow^p z$ in skeletons (and its complement failed lookup: $\vec{z} \not\rightsquigarrow^p$) analogously to lookup in classifiers:

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

We define $\mathsf{packets}(\vec{z})$ to be the set of all packets p such that there exists a pattern z where $\vec{z} \rightsquigarrow^p z$. We define the $\mathsf{packets}(\vec{z}, z)$, as $\mathsf{packets}(z) \setminus \mathsf{packets}(\vec{z})$. which captures the packets defined by z with respect to its prefix. Because classifiers are processed left to right, $\mathsf{packets}(\vec{z}, z)$ tells us the set of packets that will actually match z when preceded by $\vec{z}$.

Lemma 3 (Packets matched by cross product)**.** If $\mathsf{packets}(\vec{z}_1, z_1) = P$ and $\mathsf{packets}(\vec{z}_2, z_2) = P'$, then for all $\vec{z}$,

$$\mathsf{packets}((\vec{z}_1 \times (\vec{z}_2, z_2, \vec{z}), z_1 \times \vec{z}_2), z_1 \sqcap z_2) = P \cap P' .$$

Given a classifier $\vec{r}$, we define $\mathsf{packets}(\vec{r})$ to be the set of packets p such that $\vec{r} \rightsquigarrow^p z : \gamma$ for some z and γ. Define $\mathsf{controllerPackets}(\vec{r})$ to be the set of all packets p such that $\vec{r} \rightsquigarrow^p z : \Omega$.

A.2 Oracles

This subsection states required properties of the oracles used directly in compilation. We use the type Wildcard for Vector WildcardBit.

$$\begin{array}{rrcl} \text{Compilation oracle} & \mathcal{O}^{\mathsf{compile}} & \in & \mathsf{Header} \times \mathsf{Wildcard} \to \mathsf{Pattern} \\ \text{Expansion oracle} & \mathcal{O}^{\mathsf{refine}} & \in & \mathsf{Header} \times \mathsf{Wildcard} \times \mathsf{Packet} \to \mathsf{Wildcard} \\ \text{Switch oracle} & \mathcal{O}^{\mathsf{switch}} & \in & \mathsf{Action} \to \mathsf{SwitchAct} \end{array}$$

Definition 1 (Compilation oracle correctness)**.** If $\mathcal{O}^{\mathsf{compile}}(h, w) = z$, then for all $(\Sigma, s, p) \in [\![h : w]\!]$, we have $p \sqsubseteq z$.

Definition 2 (Expansion oracle correctness)**.** If $\mathcal{O}^{\mathsf{refine}}(h, w, p) = w'$, then

- $[\![h : w']\!] \subseteq [\![h : w]\!]$ and
- if there exists a w'' such that $\mathcal{O}^{\mathsf{compile}}(h, w'') = z$ and $p \sqsubseteq z$ only if $(\Sigma, s, p) \in [\![h : w]\!]$, then $w' = w''$.

Definition 3 (Switch oracle correctness)**.** If $\mathcal{O}^{\mathsf{switch}}(A) = \gamma$ and $\gamma \neq \Omega$, then $\mathsf{Action}(\gamma, p) = T'$ if and only if $[\![A]\!](p) = T'$.

A.3 Pruning

There are a variety of different prunes that can be done. Regardless of the prune chosen, it must satisfy the following correctness conditions.

Definition 4 (Prune correctness)**.** Let $\mathsf{prune}(\vec{r}, p) = \vec{r}'$. Then we have that:

- If $\vec{r}$ on switch s is correct with respect to π then $(\vec{r}', \vec{r})$ on s is correct with respect to π,
- $\mathsf{controllerPackets}(\vec{r}') = \varnothing$,
- If $p \in \mathsf{packets}(\vec{r}) \setminus \mathsf{controllerPackets}(\vec{r})$, then $p \in \mathsf{packets}(\vec{r})$.

A.4 Consistent Definitive Action Analysis

Consistent definitive action analysis (CDAA) is an analysis that attempts to determine the consistent set of actions and keys associated with a pattern. To be more specific, it determines whether a predicate e' is *absorbed* or *annihilated* by a pattern z. Notice that there are no rules involving inspectors – the analysis can not determine the actions produced by an inspector. If it is *necessary* for the analysis to understand the semantics of an inspector, it will fail to deduce a consistent action.

The analysis uses one further oracle, the CDAA oracle.

$$\text{CDAA oracle} \quad \mathcal{O}^{\mathsf{CDAA}} \in \mathsf{Pattern} \times \mathsf{Header} \to \mathsf{Wildcard}$$

Definition 5 (Disjoint bitvectors)**.** We say $\vec{w}$ is disjoint from $\vec{w}'$ when there does not exist a bitvector $\vec{b}$ such that $\vec{b} \sqsubseteq \vec{w}$ and $\vec{b} \sqsubseteq \vec{w}'$

Definition 6 (Static analysis oracle correctness)**.** $p \sqsubseteq z$ if and only if for every h, we have $p(h) \sqsubseteq \mathcal{O}^{\mathsf{CDAA}}(z, h)$.

$\boxed{s \mid \vec{z} \mid z \vdash e \text{ absorbed}, s \mid \vec{z} \mid z \vdash e \text{ annihilated}}$

AB-PRSMALLER
$$\frac{\mathcal{O}^{\mathsf{CDAA}}(z,h) = \vec{w}' \qquad \vec{w}' \sqsubseteq \vec{w}}{s \mid \vec{z} \mid z \vdash h : \vec{w} \text{ absorbed}}$$

AN-PRDISJOINT
$$\frac{\mathcal{O}^{\mathsf{CDAA}}(z,h) = \vec{w}' \qquad \vec{w} \text{ and } \vec{w}' \text{ disjoint}}{s \mid \vec{z} \mid z \vdash h : \vec{w} \text{ annihilated}}$$

AN-VOID
$$\frac{\mathcal{O}^{\mathsf{CDAA}}(z,h) = \vec{w}' \qquad \mathcal{O}^{\mathsf{CDAA}}(z',h) = \vec{w}'' \qquad \vec{w} \sqcap \vec{w}' \sqsubseteq \vec{w} \sqcap \vec{w}''}{s \mid \vec{z}, z', \vec{z}' \mid z \vdash h : \vec{w} \text{ annihilated}}$$

AB-PRSWITCH
$$\frac{}{s \mid \vec{z} \mid z \vdash \mathsf{switch}\ s \text{ absorbed}}$$

AN-PRSWITCH
$$\frac{s \neq s'}{s \mid \vec{z} \mid z \vdash \mathsf{switch}\ s' \text{ annihilated}}$$

AB-PRINTERSECT
$$\frac{s \mid \vec{z} \mid z \vdash e_1 \text{ absorbed} \qquad s \mid \vec{z} \mid z \vdash e_2 \text{ absorbed}}{s \mid \vec{z} \mid z \vdash e_1 \cap e_2 \text{ absorbed}}$$

AN-PRINTERSECT-1
$$\frac{s \mid \vec{z} \mid z \vdash e_1 \text{ annihilated}}{s \mid \vec{z} \mid z \vdash e_1 \cap e_2 \text{ annihilated}}$$

AN-PRINTERSECT-2
$$\frac{s \mid \vec{z} \mid z \vdash e_2 \text{ annihilated}}{s \mid \vec{z} \mid z \vdash e_1 \cap e_2 \text{ annihilated}}$$

AB-PRNEG
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ annihilated}}{s \mid \vec{z} \mid z \vdash \neg e \text{ absorbed}}$$

AN-PRNEG
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ absorbed}}{s \mid \vec{z} \mid z \vdash \neg e \text{ annihilated}}$$

$\boxed{s \mid \vec{z} \mid z \vdash \tau \rightsquigarrow A}$

CDAA-POABPR
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ absorbed}}{s \mid \vec{z} \mid z \vdash (e \to A) \rightsquigarrow A}$$

CDAA-POANPR
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ annihilated}}{s \mid \vec{z} \mid z \vdash (e \to A) \rightsquigarrow \varnothing}$$

CDAA-POINTERSECT
$$\frac{s \mid \vec{z} \mid z \vdash \tau \rightsquigarrow A \qquad s \mid \vec{z} \mid z \vdash \tau' \rightsquigarrow A'}{s \mid \vec{z} \mid z \vdash (\tau \cap \tau') \rightsquigarrow A \cap A'}$$

CDAA-POINTERSECTEMPTY-1
$$\frac{s \mid \vec{z} \mid z \vdash \tau \rightsquigarrow \varnothing}{s \mid \vec{z} \mid z \vdash (\tau \cap \tau') \rightsquigarrow \varnothing}$$

CDAA-POINTERSECTEMPTY-2
$$\frac{s \mid \vec{z} \mid z \vdash \tau' \rightsquigarrow \varnothing}{s \mid \vec{z} \mid z \vdash (\tau \cap \tau') \rightsquigarrow \varnothing}$$

CDAA-PONEG
$$\frac{s \mid \vec{z} \mid z \vdash \tau \rightsquigarrow A}{s \mid \vec{z} \mid z \vdash \neg\tau \rightsquigarrow \neg A}$$

$\boxed{s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow K}$

CDAA-QABPR
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ absorbed}}{s \mid \vec{z} \mid z \vdash (K \leftarrow e) \rightsquigarrow K}$$

CDAA-QANPR
$$\frac{s \mid \vec{z} \mid z \vdash e \text{ annihilated}}{s \mid \vec{z} \mid z \vdash (K \leftarrow e) \rightsquigarrow \varnothing}$$

CDAA-QINTERSECT
$$\frac{s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow K \qquad s \mid \vec{z} \mid z \vdash \phi' \rightsquigarrow K'}{s \mid \vec{z} \mid z \vdash (\phi \cap \phi') \rightsquigarrow K \cap K'}$$

CDAA-QINTERSECTEMPTY-1
$$\frac{s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow \varnothing}{s \mid \vec{z} \mid z \vdash (\phi \cap \phi') \rightsquigarrow \varnothing}$$

CDAA-QINTERSECTEMPTY-2
$$\frac{s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow \varnothing}{s \mid \vec{z} \mid z \vdash (\phi \cap \phi') \rightsquigarrow \varnothing}$$

CDAA-QNEG
$$\frac{s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow K}{s \mid \vec{z} \mid z \vdash \neg\phi \rightsquigarrow \neg K}$$

$\boxed{s \mid \vec{z} \mid z \vdash \pi \rightsquigarrow (A, K)}$

CDAA-PROGRAM
$$\frac{s \mid \vec{z} \mid z \vdash \tau \rightsquigarrow A \qquad s \mid \vec{z} \mid z \vdash \phi \rightsquigarrow K}{s \mid \vec{z} \mid z \vdash (\tau, \phi) \rightsquigarrow (A, K)}$$

Lemma 4 (Soundness of CDAA)**.**

- If $s \mid \vec{z} \mid z \vdash e$ absorbed, then for all (Σ, s, p) such that $p \in \mathsf{packets}(\vec{z}, z)$, we have that $(\Sigma, s, p) \in [\![e]\!]$.
- If $s \mid \vec{z} \mid z \vdash e$ annihilated, then for all (Σ, s, p) such that $p \in \mathsf{packets}(\vec{z}, z)$, we have that $(\Sigma, s, p) \notin [\![e]\!]$.

Proof. Induction on the derivation of the hypotheses. □

Theorem 3 (Soundness)**.** If $s \mid \vec{z} \mid z \vdash \pi \rightsquigarrow (A, K)$, then for all (Σ, s, p) such that $p \in \mathsf{packets}(\vec{z}, z)$, we have $[\![\pi]\!](\Sigma, s, p) = ([\![A]\!](\Sigma, s, p), [\![K]\!](\Sigma, s, p))$.

A.5 Elided Correctness Properties

Definition 7 (Classifier correct with respect to policy)**.** We say that classifier $\vec{r}$ on switch s is correct with respect to policy τ if for every p such that $\vec{r} \rightsquigarrow^p z : \gamma$ and $\gamma \neq \Omega$, then for all Σ, we have $\mathsf{Action}(\gamma, p) = T'$ if and only if $[\![\tau]\!](\Sigma, s, p) = T'$.

Definition 8 (Classifier correct with respect to query)**.** A switch classifier $\vec{r}$ on switch s is correct with respect to ϕ if for all z there exists a Σ' such that for all p such that $\vec{r} \rightsquigarrow^p z : \gamma$ and $\gamma \neq \Omega$, then $[\![\phi]\!](\Sigma, s, p) = \Sigma'$.

Definition 9 (Classifier correct with respect to program)**.** A switch classifier $\vec{r}$ on switch s is correct with respect to $\pi = (\tau, \phi)$ if it is correct with respect to τ and correct with respect to ϕ.

Theorem 4 (Correctness of classifier generation)**.** If $\mathsf{GenClassifier}(\pi, s) = \vec{r}$, then $\vec{r}$ on switch s is correct with respect to π.

Proof. Follows from soundness of CDAA and the axioms of the switch oracle. □

Theorem 5 (Expansion correctness)**.** For all x and π, we have that $[\![\mathcal{R}(x, \pi)]\!] = [\![\pi]\!]$.

Proof. By induction on π. □

Using the above lemmas and theorems, one can prove the following relations, establishing our functional correctness theorem.

Definition 10 (Asynchronous simulation relation)**.** Let

$$\overline{M} = \mathbb{C}(\pi \mid \Sigma), \mathbb{S}(s_1 \mid \vec{r}_1 \mid \overline{Z}_1), \ldots, \mathbb{S}(s_n \mid \vec{r}_n \mid \overline{Z}_n), \overline{L}, \overline{T}$$
$$M_{\text{async}} = (\pi', \Sigma', \overline{T}_1, \overline{T}_2).$$

Then $\overline{M} \sim_{\text{async}} M_{\text{async}}$ if

- $[\![\pi]\!] = [\![\pi']\!]$
- For all i, classifier $\vec{r}_i$ on switch s_i is correct with respect to π.
- $\Sigma = \Sigma'$
- There exists a bijection between $\overline{T}_1$ and $\overline{L} \uplus \overline{T}$, where every mapping preserves (s, p).
- There exists a bijection between transmissions $\mathbb{T}(s \mid p) \in \overline{T}_2$ and patterns $z \in \overline{Z}_i$, where $s_i = s$ and $p \sqsubseteq z$.

Definition 11 (Synchronous simulation relation)**.** Let

$$\overline{M} = \mathbb{C}(\pi \mid \Sigma), \mathbb{S}(s_1 \mid \vec{r}_1 \mid \overline{Z}_1), \ldots, \mathbb{S}(s_n \mid \vec{r}_n \mid \overline{Z}_n), \overline{L}, \overline{T}$$
$$M_{\text{sync}} = (\pi', \Sigma', \overline{T}').$$

Then $\overline{M} \sim_{\text{sync}} M_{\text{sync}}$ if

- $[\![\pi]\!] = [\![\pi']\!]$
- For all i, classifier $\vec{r}_i$ on switch s_i is correct with respect to π.
- $\Sigma = \Sigma'$
- There exists a bijection between $\overline{L} \uplus \overline{T}$ and $\overline{T}'$, where every mapping preserves (s, p).
- Each $\overline{Z}_i$ is empty.

A.6 Elided Quiescence Properties

Definition 12 (Ideal packets matched on switch)**.** Given a predicate e, define the set of packets ideally matched on switch s (written $\mathsf{packetsOn}(s, e)$) as all p such that $(\Sigma, s, p) \in [\![e]\!]$ implies $(\Sigma', s, p) \in [\![e]\!]$, for all Σ and Σ'. Analogously, define the set of packets ideally dropped on the switch s (written $\mathsf{nPacketsOn}(s, e)$) as all p such that $(\Sigma, s, p) \in \neg [\![e]\!]$ implies $(\Sigma', s, p) \in \neg [\![e]\!]$, for all Σ and Σ'.

Definition 13 (anti-kernel)**.** Define the anti-kernel with respect to (Σ, s) of a policy τ (resp. a query ϕ), denoted $\mathsf{aker}(\Sigma, s, \tau)$ (resp. $\mathsf{aker}(\Sigma, s, \phi)$) as the set of all packets p such that $[\![\tau]\!](\Sigma, s, p) \neq \varnothing$ (resp. $[\![\phi]\!](\Sigma, s, p) \neq \varnothing$).

The following function predicts how many packets will go to the controller.

$\boxed{\mathsf{Inconsistent}(s, e) = P}$

INCON-PREXACT
$$\frac{\mathcal{O}^{\mathsf{compile}}(h, \vec{w}) = z \qquad [\![h : \vec{w}]\!] = \mathsf{packets}(z)}{\mathsf{Inconsistent}(s, h : \vec{w}) = \varnothing}$$

INCON-PROVERAPPROX
$$\frac{\mathcal{O}^{\mathsf{compile}}(h, \vec{w}) = z \qquad [\![h : \vec{w}]\!] \neq \mathsf{packets}(z)}{\mathsf{Inconsistent}(s, h : \vec{w}) = \mathsf{packets}(z)}$$

INCON-PRWITCH
$$\frac{}{\mathsf{Inconsistent}(s, \mathsf{switch}\ s') = \varnothing}$$

INCON-PRINSPECT
$$\frac{}{\mathsf{Inconsistent}(s, \mathsf{inspect}\ i) = \top}$$

INCON-PRINTERSECT
$$\frac{\mathsf{Inconsistent}(s, e) = P \qquad \mathsf{Inconsistent}(s, e') = P'}{\mathsf{Inconsistent}(s, e \cap e') = (P \cup P') \cap (\mathsf{packetsOn}(s, e) \cup P) \cap (\mathsf{packetsOn}(s, e') \cup P')}$$

INCON-PRNEG
$$\frac{\mathsf{Inconsistent}(s, e) = P}{\mathsf{Inconsistent}(s, \neg e) = P}$$

$\boxed{\mathsf{Inconsistent}(s, \tau) = P}$

INCON-POPR
$$\frac{\mathsf{Inconsistent}(s, e) = P}{\mathsf{Inconsistent}(s, e \to A) = P}$$

INCON-POINTERSECT
$$\frac{\mathsf{Inconsistent}(s, \tau) = P \qquad \mathsf{Inconsistent}(s, \tau') = P'}{\mathsf{Inconsistent}(s, \tau \cap \tau') = (P \cup P') \cap (\mathsf{aker}(\Sigma, s, \tau) \cup P) \cap (\mathsf{aker}(\Sigma, s, \tau') \cup P')}$$

INCON-PONEG
$$\frac{\mathsf{Inconsistent}(s, \tau) = P}{\mathsf{Inconsistent}(s, \neg\tau) = P}$$

$\boxed{\mathsf{Inconsistent}(s, \phi) = P}$

INCON-QPR
$$\frac{\mathsf{Inconsistent}(s, e) = P}{\mathsf{Inconsistent}(s, K \leftarrow e) = P}$$

INCON-QINTERSECT
$$\frac{\mathsf{Inconsistent}(s, \phi) = P \qquad \mathsf{Inconsistent}(s, \phi') = P'}{\mathsf{Inconsistent}(s, \phi \cap \phi') = (P \cup P') \cap (\mathsf{aker}(\Sigma, s, \phi) \cup P) \cap (\mathsf{aker}(\Sigma, s, \phi') \cup P')}$$

INCON-QNEG
$$\frac{\mathsf{Inconsistent}(s, \phi) = P}{\mathsf{Inconsistent}(s, \neg\phi) = P}$$

$\boxed{\mathsf{Inconsistent}(s, F) = P}$

INCON-PROGRAM
$$\frac{\mathsf{Inconsistent}(s, \tau) = P \qquad \mathsf{Inconsistent}(s, \phi) = P'}{\mathsf{Inconsistent}(s, (\tau, \phi)) = (P \cup P') \cap (\mathsf{aker}(\Sigma, s, \tau) \cup P) \cap (\mathsf{aker}(\Sigma, s, \phi) \cup P')}$$

Lemma 5 (Prefix extension)**.**

- If $s \mid \vec{z} \mid z \vdash e$ absorbed and $\vec{z}'$ is a supersequence of $\vec{z}$, then $s \mid \vec{z}' \mid z \vdash e$ absorbed.
- If $s \mid \vec{z} \mid z \vdash e$ annihilated and $\vec{z}'$ is a supersequence of $\vec{z}$, then $s \mid \vec{z}' \mid z \vdash e$ annihilated.

Proof. Induction on the derivation of the hypotheses, ignoring the other members of $\vec{z}'$ aside from $\vec{z}$. □

Lemma 6 (Uniform meet)**.** If $\mathcal{S}(s, e) = (\vec{z}, z, \vec{z}')$, then

- If $s \mid \vec{z} \mid z \vdash e$ absorbed, then $s \mid \vec{z} \times z' \mid z \sqcap z' \vdash e$ absorbed.
- If $s \mid \vec{z} \mid z \vdash e$ annihilated, then $s \mid \vec{z} \times z' \mid z \sqcap z' \vdash e$ annihilated.

Proof. Induction on the derivation of the hypotheses. □

Lemma 7 (Cross product relations)**.** For all $\vec{z}$, if $\mathcal{S}(s, e_1) = (\vec{z}_1, z_1, \vec{z}_1')$ and $\mathcal{S}(s, e_2) = (\vec{z}_2, z_2, \vec{z}_2')$ and $\vec{z}' = \vec{z}_1 \times (\vec{z}_2, z_2, \vec{z}), z_1 \times \vec{z}_2$, then

- If $s \mid \vec{z}_1 \mid z_1 \vdash e_1$ absorbed and $s \mid \vec{z}_2 \mid z_2 \vdash e_2$ absorbed then $s \mid \vec{z}' \mid z_1 \sqcap z_2 \vdash e_1 \cap e_2$ absorbed.
- If $s \mid \vec{z}_1 \mid z_1 \vdash e_1$ annihilated or $s \mid \vec{z}_2 \mid z_2 \vdash e_2$ annihilated then $s \mid \vec{z}' \mid z_1 \sqcap z_2 \vdash e_1 \cap e_2$ annihilated.

Proof. Follows from uniform meet, prefix extension, and Ab-PrIntersect/An-PrIntersect. □

Lemma 8 (Consistent predicate skeleton and absorption/annihilation)**.** If $\mathcal{S}(s, e) = (\vec{z}, z, \vec{z}')$ and $\mathsf{Inconsistent}(s, e) = P$, then, for all Σ,

- if $\mathsf{packets}(\vec{z}, z) \subseteq \mathsf{packetsOn}(s, e) \setminus P$, we have that $s \mid \vec{z} \mid z \vdash e$ absorbed.
- if $\mathsf{packets}(\vec{z}, z) \subseteq \mathsf{nPacketsOn}(s, e) \setminus P$, we have that $s \mid \vec{z} \mid z \vdash e$ annihilated.

Proof. Induction on e, using the cross-product relations and the algebraic properties of $\mathsf{Inconsistent}(s, e)$ to discharge cases in which we don't have useful induction hypotheses. □

Theorem 6 (Consistent skeleton and CDAA)**.** Let $\mathcal{S}(s, \pi) = (\vec{z}, z, \vec{z}')$ and $\mathsf{Inconsistent}(s, \pi) = P$. Then if $\mathsf{packets}(\vec{z}, z)$ disjoint from P, we have that $s \mid \vec{z} \mid z \vdash \pi \rightsquigarrow (A, K)$.

$\boxed{\mathsf{HeadersAvail}(e, h) = P}$

$$\mathsf{HeadersAvail}(h : w, h) = \begin{cases} \mathsf{packets}(z) & \text{if } \mathcal{O}^{\mathsf{compile}}(h, w) = z \text{ and } \mathcal{O}^{\mathsf{CDAA}}(z, h) = \vec{b} \\ \varnothing & \text{otherwise} \end{cases}$$
$$\mathsf{HeadersAvail}(\mathsf{switch}\ s, h) = \varnothing$$
$$\mathsf{HeadersAvail}(\mathsf{inspect}\ i, h) = \varnothing$$
$$\mathsf{HeadersAvail}(e_1 \cap e_2, h) = \mathsf{HeadersAvail}(e_1, h) \cup \mathsf{HeadersAvail}(e_2, h)$$
$$\mathsf{HeadersAvail}(\neg e, h) = \mathsf{HeadersAvail}(e, h)$$

$\boxed{\mathsf{HeadersAvail}(\tau, h) = P}$

$$\mathsf{HeadersAvail}(e \to A, h) = \mathsf{HeadersAvail}(e, h)$$
$$\mathsf{HeadersAvail}(\tau_1 \cap \tau_2, h) = \mathsf{HeadersAvail}(\tau_1, h) \cup \mathsf{HeadersAvail}(\tau_2, h)$$
$$\mathsf{HeadersAvail}(\neg\tau, h) = \mathsf{HeadersAvail}(\tau, h)$$

$\boxed{\mathsf{HeadersAvail}(\phi, h) = P}$

$$\mathsf{HeadersAvail}(K \leftarrow e, h) = \mathsf{HeadersAvail}(e, h)$$
$$\mathsf{HeadersAvail}(\phi_1 \cap \phi_2, h) = \mathsf{HeadersAvail}(\phi_1, h) \cup \mathsf{HeadersAvail}(\phi_2, h)$$
$$\mathsf{HeadersAvail}(\neg\phi, h) = \mathsf{HeadersAvail}(\phi, h)$$

$\boxed{\mathsf{HeadersAvail}(\pi, h) = P}$

$$\mathsf{HeadersAvail}((\tau, \phi), h) = \mathsf{HeadersAvail}(\tau, h) \cup \mathsf{HeadersAvail}(\phi, h)$$

Theorem 7 (Available headers of skeleton)**.** Let $\mathcal{S}(s, \pi) = (\vec{z}, z, \vec{z}')$. Then there exists a $\vec{b}$ for every $p \in \mathsf{packets}(\vec{z}, z)$ such that $p(h) = \vec{b}$ if $p \in \mathsf{HeadersAvail}(\pi, h)$.

Definition 14 (Headers needed on snapshot)**.** Let $p \in \mathsf{HeadersNeeded}(s, \pi, h)$ if for all Σ, we have $[\![\pi]\!](\Sigma, s, p) = (T', \sigma')$ and there exists $(k, sp) \in \sigma'$ such that $h \in \mathsf{dom}(sp)$.

We now define which packets we expect to see at the controller.

Definition 15. We let $p \in \mathsf{ToController}(s, \pi)$ if:

- $p \in \mathsf{Inconsistent}(s, \pi)$ or
- there exists an h and a p such that $p \in \mathsf{HeadersNeeded}(s, \pi, h)$ and $p \notin \mathsf{HeadersAvail}(\pi, h)$.

Theorem 8 (Controller actions bounded)**.** If π is realizable on s, then $\mathcal{S}(s, \pi) = \vec{z}$ and $\mathcal{A}(s, \pi, \vec{z}) = \vec{r}$ then $\mathsf{controllerPackets}(\vec{r}) \subseteq \mathsf{ToController}(s, \pi)$.

Proof. Induction on π. □

Theorem 9 (Expansion monotonic)**.** For all s, x and π, we have that $\mathsf{ToController}(s, \mathcal{R}(x, \pi)) \subseteq \mathsf{ToController}(s, \pi)$.

Proof. Induction on π. □

Theorem 10 (Expansion invariance)**.** For all (Σ, s, p), if π is fully invariant and fully matchable, we have that $p \notin \mathsf{ToController}(s, \mathcal{R}((\Sigma, s, p), \pi))$.

Proof. Induction on π. □