# FORMAL REASONING IN SOFTWARE-DEFINED NETWORKS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mark Reitblatt

August 2015

FORMAL REASONING IN SOFTWARE-DEFINED NETWORKS

Mark Reitblatt, Ph.D.

Cornell University 2015

This thesis presents an end-to-end approach for building computer networks that can be reasoned about and verified formally. In it, we present a high-level specification language for describing the desired forwarding behavior of networks based on regular expressions over network paths, as well as a tool that automatically verifies network forwarding policies; an approach to building formally verified compilers and runtimes for forwarding policies written in a network programming language that preserve the semantics of the source policy; and a technique for updating network configurations while preserving correctness.

## BIOGRAPHICAL SKETCH

Mark Reitblatt was born, raised, and mostly educated in the great State of Texas. He attended the University of Texas (at Austin), receiving a Bachelors of Science in Pure Mathematics and a Bachelors of Science in Computer Science, with Honors. Then, after a short, but pleasant, stint at Intel, he joined the Computer Science PhD program at Cornell University.

This document is dedicated to all Cornell graduate students.

*"Fight the power. We've got to fight the powers that be."*

—Public Enemy

Please do not print this document unless absolutely necessary.

## ACKNOWLEDGEMENTS

I would like to thank my advisor Nate for his infinite patience and support, Dexter Kozen for teaching me more about theory than I could possibly remember, and Bob Constable for always having an open door and endless enthusiasm for the questions and intellectual wonderings of a junior PhD student. My academic journey would have been deeply impoverished without the (sometimes cruel) tutelage of the inimitable Cornell Systems Lunch, which was (I hope) the site of my progression from truly terrible to acceptably adequate presenter. The crowd changed over the years, but a few faces stand out for their consistently incisive and inspiring insight and questioning: Robbert van Renesse, Fred Schneider[1], and Gün Sirer. I would also like to thank my collaborators for helping "pull me through the hole" on our papers: Marco Canini, Arjun Guha, Kostas Mamouras, Jen Rexford, Cole Schlesinger, Alexandra Silva, and David Walker.

My 5 years in Ithaca were made infinitely more enjoyable by the company and friendship of too many individuals to name here. Key among them was my constant roommate and friend, Eoin O'Mahony, and our merry changing band of roommates and drinking buddies, including but not limited to: Diarmuid Cahalane, Sam Hopkins, Jonathan DiLorenzo, Steffen Smolka, Rahmtin Rotabi, Daniel Freund, and Vasu Raman. Thanks also to my Ithaca Kickball team for giving me an *extra*-Cornell social outlet: Katie Moring, Lesley Middleton, Lynn Vincent, Jamie Schmohe, the Brothers Manning, Jess Gaby, Ian Dunham, Melinda Liptak, Danielle Morgan, and Matt Morgan. I am similarly grateful to my "Vet school friends" for an escape from the overly testosterone drench CS department: Marina Shepelev, Jane Park, Susan Smith, Jenna Goldhaber, Becca Vogel, and Jessica Chandler. And last, but certainly not least, my ever-present labmate, triathlon co-trainer, running mate, and eternal source of entertainment, Hussam Abu-Libdeh. Ithaca was never the same after you left buddy.

Finally, I would like to thank the Texas taxpayers and supporters of the University of

[1]Not to be confused with the unmustachioed lead singer of the B-52's

Texas System. You've created one of the finest public education institutions anywhere in the world, and I would not be here without it.

Work supported in part by the National Science Foundation under grants CNS-1111698 *High-Level Language Support for Trustworthy Networks* and CNS-1413972 *Programmable Inter-Domain Observation and Control*, and the Office of Naval Research under grant N00014-12-1-0757 *Networks Opposing Botnets (NoBot) II*. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reect the views of the National Science Foundation or the Office of Naval Research.

## TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

Computer networks are integral components of modern life. Outages can strand travelers [43], shut down financial markets [72], and bring down the largest cloud provider in the world [95]. Networks need to be reliable which in turn means that network engineers and administrators need to be able to reason about and predict their behavior.

The holy grail of reasoning about computer systems such as networks is *formal verification*: a mathematical proof that a system satisfies a formal specification of its correct behavior. Formal verification has been successfully applied to a spectrum of computer systems, from compilers[59] to operating systems[55] to microprocessor design[73].

Unfortunately, computer networks have largely defied attempts at formal verification. Verification requires building a model of the system and a precise description of the correct behavior. To verify large systems such as a network or a compiler, we decompose it into smaller components that are, individually, easy to model and specify, and that interact with one another in clear, well-defined ways. In contrast, traditional networks are built from complex components whose behavior is defined in terms of its effect on other components. In turn, any of these components may effect the ultimate behavior of the network: how data is forwarded. Therefore, if we want to reason about forwarding behavior then we must model the full system, in all of its complexity.

In traditional networking, the network is abstracted into layers: the *data plane*, which is the hardware layer where data is forwarded, and the *control plane*, which monitors and configures the data plane in response to events in the network (*e.g.* link failures or shift in network traffic). The actual forwarding behavior of the network is determined by the data plane, but its configuration is determined by the control plane, whose behavior is in turn determined by a combination of the network state, the current configuration, and the control plane software. Out of these three factors, the network administrator only directly controls one: the configuration inputs. Thus, if the administrator wants to reason about the behavior of a new configuration, they have to model and reason about the entire layer, including the software[1]. Currently, the only tools available for reasoning about network configurations are either data plane verifiers like VeriFlow [53] or NetPlumber [49] which can check properties of the current data plane configuration, but cannot predict the behavior of the control plane (*e.g.* will a new configuration preserve connectivity?), configuration static analyzers such as rcc [20], which can detect generic configuration errors, but cannot establish correctness (*e.g.* does the configuration allow only traffic from trusted hosts to reach the server?), or control plane verifiers such as Propane[8] that analyze the behavior of routing protocols, but not the resulting dataplane states.

In this thesis, we show how to build software-defined networks that can be formally verified. We show how to build formal specifications of a network's forwarding behavior and prove that the network's forwarding policy, written in a high-level network programming language, satisfies its specification. We then show how to build a compiler and network controller that translates the forwarding policy into a data plane configuration protocol (OpenFlow) and implements the policies runtime requirements in the network. In particular, our compiler and controller have been formally proven correct: the resulting network behavior is *observationally equivalent*[2] to the input network policy. We then show how to update the network policy in such a way that reasoning and verification performed on the old and new policies is preserved by the network while transitioning. Network updates are a common source of network errors and being able to reason about the behavior of a network under

[1]The control plane software itself is highly complex: on the order of 50 million lines of code.
[2]Observational equivalence is precisely defined in Chapter 4

update is essential to a verified system.

## 1.1 Contributions

Verification starts with a specification of correctness. In Chapter 3, we present an expressive, well-defined language for specifying the correct forwarding behavior of networks, precisely and formally describe what it means for a network program (a declarative specification of a specific forwarding configuration) to satisfy its specification, and show how to build a verifier that automatically proves the correctness (or incorrectness) of a program with respect to a given specification.

Verifying a network program has only limited utility if a correct program is translated and implemented incorrectly by a compiler or runtime. Indeed, several works have found correctness bugs in compilers and network controllers that would counter-act the benefits of verification (see *e.g.* [14] [35]). In Chapter 4 we show how to build a formally verified compiler and runtime for network programs that provably preserves the correctness of the input program. We also build a formal model of OpenFlow [71], the most popular SDN protocol, and describe a generic proof technique that can be adapted to verify future implementations.

The techniques described in Chapters 3 and 4 only apply to a single network program, but real network forwarding policies change over time, in response to changes in the network topology, traffic loads, application demands, *etc.*. Chapter 5 we show how to update a network from one forwarding policy to another in such a way that verification performed on the original and final configurations is preserved by the network in the transition.

Each of these developments is demonstrated by a real system that has been implemented and publicly released under an open-source license. Each system is also accompanied by a formal model or semantics that precisely and clearly models its behavior. In addition, all of the theorems and code described in Chapter 4, and most of the theorems in Chapter 5 have been formally verified in the Coq proof assistant.

CHAPTER 2

# BACKGROUND

## 2.1 Historical context

Right now, in 2015, we are at the beginning of a large-scale revolution in the world of networking. For more than 30 years, there has been little fundamental change in the way that we design, build, and maintain networks. If you transported a network administrator from 1985 forward to today, they would recognize the fundamental principles of network design from their own day, even if the names, protocols, and other details have changed. At the same time, the kinds of applications and sheer scale of networks has changed in ways that we could have never dreamt of before the explosion of the internet and internet services.

This traditional networking style was based on a distributed architecture of expensive hardware boxes (called routers or switches, depending upon increasingly irrelevant protocol distinctions) that coordinated and configured themselves through a myriad of distributed protocols. For the purpose of this thesis, the important thing to understand is that every aspect of these networks, from the physical hardware to the software running on them, to the protocols they use to coordinate, was built and controlled wholly by network vendors, not the network owners.

Over the decades, there have been many proposals for new network architectures that solve the problems of traditional networking, and have the flexibility to adapt to new applications and demands. One of the best known proposals, Active Networks [103], turned packets into programs by embedding a full Turing-complete language into packet headers. Switches were transformed into interpreters, and control over forwarding and other network functions was delegated completely to end-hosts.

## 2.2 Software-defined Networking

More recently, Software-defined Networking (henceforth "SDN") has arisen as a serious challenger to the status quo. SDN decouples the packet-processing functions of the data plane from the control plane through a logically centralized-controller[1] that manages the network directly by configuring the packet-handling mechanisms in the underlying switches. The key elements of the design are that the switch configuration protocol is open and standardized, and the essential network management software runs on a commodity server programmed by the network owner. This allows the development of reusable implementations of generic network functions and makes the network almost completely customizable.

SDN has already seen adoption far beyond what Active Networks or any other competing proposal ever saw. Some of the largest technology companies in the world (Google, Microsoft, Facebook, VMware, and more) have either already adopted it, or are developing products around it.

**OpenFlow** In this thesis, we will use the OpenFlow SDN protocol to implement our network programs.

In an OpenFlow network, switches connect to a central controller, which then directly configures their forwarding behavior by programming a *flow table*. A flow table is a list of *match-action* rules, consisting of a *match* pattern that describes packet headers, and a list

[1]The controller is *centralized* is the sense that a single architectural component controls a function, as opposed to being *decentralized* in which many components control their own functions independently. Physically, the controller may be implemented as many co-ordinating distributed controllers.

of *action* rules that dictate how to process matching packet. An example is

| Priority | Pattern | Action |
|---|---|---|
| 2 | $\{\mathbf{dlDst} = \mathit{H1}\}$ | $\{\lvert 10 \rvert\}$ |
| 1 | $\star$ | $\{\lvert\rvert\}$ |

This flow table has two entries: the first one matches packets whose destination MAC address (**dlDst**) is *H1*, and forwards them out port 10. The second rule matches all packets and forwards them on no ports, which drops them. The rules have priorities that disambiguate overlapping rules. Because the first rule has a higher priority, it applies before the second one. Therefore, this flow table forwards packets destined for MAC address *H1* out port 10, and drops all other packets. In the rest of this thesis, we will omit priorities when possible. Rules will be listed in priority order, with high priority to low priority, top to bottom. Note that the action field of a rule is a multiset: if the same port is repeated, then a packet is duplicated and forwarded out the same port multiple times. In addition to forwarding, the action field can modify the value of packet header fields, or send the packet to the controller.

The controller programs flow tables by sending sequences of messages to install rules on the switch. To install the first rule in the above flow table, the controller would send:

$$\mathbf{Add}\ 2\ \{\mathbf{dlDst} = \mathtt{H2}\}\ \{\lvert 10 \rvert\}$$

**Network programming languages** The interfaces exposed by SDN protocols, in particular OpenFlow, are quite low-level, the networking equivalent of assembly. Recent work has proposed a number of high-level languages and language abstractions to simplify the task of developing correct, reliable SDN systems. See [15] and [23] for a survey of current developments in network programming languages.

In this thesis, we focus upon two programming languages in the Frenetic family: NetKAT

[4] and its predecessor NetCore [77]. NetKAT is introduced in this chapter, and NetCore is described before it is used in Chapter 4.

## 2.3 The NetKAT Programming Language

The network programming language NetKAT was developed by Anderson *et al.* [4]. NetKAT is an extension of Kleene algebra with tests (KAT), an algebraic system for program verification that combines Kleene Algebra (KA) with boolean algebra [58]. NetKAT offers a collection of intuitive constructs including: predicates over packets; primitives for modifying packet headers and encoding topologies; iterations; and sequential and parallel composition operators. The semantics is given in terms of a denotational model based on functions from packet histories to sets of packet histories (where a history records a packet's path through the network). In addition to the denotational semantics, NetKAT has a sound and complete equational deductive system.

### 2.3.1 Syntax

NetKAT [4] is based upon Kleene algebra with tests (KAT) [58], a generic equational system for reasoning about partial correctness of programs.

**Kleene Algebra (KA) & Kleene Algebra with Tests (KAT)** A *Kleene algebra* (KA) is an algebraic structure,

$$(K,\ +,\ \cdot,\ \star,\ 0,\ 1)$$

where $K$ is an idempotent semiring under $(+, \cdot, 0, 1)$, and $p^* \cdot q$ is the least solution of the inequality $p \cdot r + q \leq r$, where $p \leq q$ is shorthand for $p + q = q$, and similarly for $q \cdot p^*$. A

$$
\begin{array}{rrll}
\text{Naturals} & n & \in \; 0 \mid 1 \mid 2 \mid \ldots & \\
\text{Fields} & x & ::= \; x_1 \mid \cdots \mid x_k & \\
\text{Packets} & pk & ::= \; \{f_1 = n_1, \cdots, f_k = n_k\} & \\
\text{Histories} & \sigma & ::= \; \langle pk \rangle \mid pk{:}\sigma & \\
\text{Tests} & a & ::= \mathsf{1} & \textit{True} \\
 & & \mid \; \mathsf{0} & \textit{False} \\
 & & \mid \; x = n & \textit{Header test} \\
 & & \mid \; a_1 + a_2 & \textit{Disjunction} \\
 & & \mid \; a_1 \cdot a_2 & \textit{Conjunction} \\
 & & \mid \; \neg a & \textit{Negation} \\
\text{Actions} & p & ::= a & \textit{Test} \\
 & & \mid \; x \leftarrow n & \textit{Modification} \\
 & & \mid \; p_1 + p_2 & \textit{Parallel Composition} \\
 & & \mid \; p_1 \cdot p_2 & \textit{Sequential Composition} \\
 & & \mid \; p^* & \textit{Iteration} \\
 & & \mid \; \mathsf{dup} & \textit{Duplication}
\end{array}
$$

Figure 2.1: NetKAT Syntax.

*Kleene algebra with tests* (KAT) is an algebraic structure,

$$(K,\, B,\, +,\, \cdot,\, \star,\, 0,\, 1,\, \neg)$$

where $\neg$ is a unary operator defined only on $B$, such that

- $(K,\, +,\, \cdot,\, \star,\, 0,\, 1)$ is a Kleene algebra,
- $(B,\, +,\, \cdot,\, \neg\,,\, 0,\, 1)$ is a Boolean algebra, and
- $(B,\, +,\, \cdot,\, 0,\, 1)$ is a subalgebra of $(K,\, +,\, \cdot,\, 0,\, 1)$.

The elements of $B$ and $K$ are called *tests* and *actions.*

The axioms of KA and KAT (both elided here) capture natural conditions such as associativity of $\cdot$; see the original paper by Kozen for a complete listing [58].

**NetKAT** NetKAT [4] extends KAT with network-specific primitives for filtering, modifying, and forwarding packets, along with additional axioms for reasoning about programs built using those primitives. Formally, NetKAT is KAT with atomic actions and tests

$$x \leftarrow n \qquad\qquad x = n \qquad\qquad \mathsf{dup}$$

with the following meanings: The test $x = n$ tests whether field $x$ of the current packet contains the value $n$; the assignment $x \leftarrow n$ assigns the value $n$ to the field $x$ in the current packet; and the action $\mathsf{dup}$ duplicates the last packet in the packet history, which keeps track of the path the packet takes through the network.

For example, the NetKAT expression

$$\mathit{pt} = 5 \cdot \mathit{sw} = 3 \cdot \mathit{dst} \leftarrow 192.168.1.5 \cdot \mathit{pt} \leftarrow 5$$

encodes the command: "For all packets located at port 5 of switch 3, set the destination address to 192.168.1.5 and forward it out on port 5."

### 2.3.2 Semantics

The standard semantics of NetKAT interprets expressions as packet-processing functions. As defined in Figure 2.1, a packet $\pi$ is a record whose fields assign constant values $n$ to fields $x$ and a packet history is a nonempty sequence of packets $\pi_1 {:} \pi_2 {:} \cdots {:} \pi_k$, listed in order of youngest to oldest. Operationally, only the head packet $\pi_1$ exists in the network, but we keep track of the packet's history in the semantics to enable precise reasoning about behavior involving forwarding along different paths.

Formally, a NetKAT term $p$ denotes a function

$$[\![p]\!] : \mathcal{H} \to 2^{\mathcal{H}},$$

where $\mathcal{H}$ is the set of all packet histories. Intuitively, the function $[\![p]\!]$ takes an input packet history $\sigma$ and produces a set of output packet histories $[\![p]\!](\sigma)$, representing all of the packets that result from the forwarding function, and their associated histories.

The semantics of the primitive actions and tests in NetKAT are as follows. For a packet history $\pi{:}\sigma$ with head packet $\pi$,

$$\begin{aligned}
[\![x \leftarrow n]\!](\pi{:}\sigma) &= \{\pi[n/x]{:}\sigma\} \\
[\![x = n]\!](\pi{:}\sigma) &= \begin{cases} \{\pi{:}\sigma\}, & \pi(x) = n \\ \emptyset, & \pi(x) \neq n \end{cases} \\
[\![\mathsf{dup}]\!](\pi{:}\sigma) &= \{\pi{:}\pi{:}\sigma\} \\
[\![\mathsf{1}]\!](\sigma) &= \{\sigma\} \\
[\![\mathsf{0}]\!](\sigma) &= \emptyset.
\end{aligned}$$

where $\pi[n/x]$ denotes the packet $\pi$ with the field $x$ rebound to the value $n$. A test $x = n$ filters out (drops) the packet if the test is not satisfied and passes it through if it is. The $\mathsf{dup}$ construct duplicates the head packet $\pi$, yielding a fresh copy that can be modified by other constructs. Hence, in this standard model, the $\mathsf{dup}$ construct can be used to encode paths through the network, with each occurrence of $\mathsf{dup}$ marking an intermediate hop.

The operations ($+$, $\cdot$, $^*$, $\neg$) are interpreted as follows:

$$\begin{aligned}
[\![p + q]\!](\sigma) &= [\![p]\!](\sigma) \cup [\![q]\!](\sigma) \\
[\![p \cdot q]\!](\sigma) &= \bigcup_{\tau \in [\![p]\!](\sigma)} [\![q]\!](\tau) \\
[\![p^*]\!](\sigma) &= \bigcup_n [\![p^n]\!](\sigma)
\end{aligned}$$

$$[\![\neg a]\!](\sigma) = \begin{cases} \{\sigma\}, & \text{if } [\![a]\!](\sigma) = \emptyset \\ \emptyset, & \text{if } [\![a]\!](\sigma) = \{\sigma\} \end{cases}$$

Note that $+$ behaves like disjunction when applied to tests and like union when applied to actions. Similarly, $\cdot$ behaves like conjunction when applied to tests and like sequential composition when applied to actions. Negation is only ever applied to tests, as is enforced by the syntax of the language.

### 2.3.3 Equational Theory

NetKAT has a sound and complete equational theory, based upon the equational theory of KAT. This means that two NetKAT terms are semantically equivalent (denote the same function) iff there is a proof of equivalence using the NetKAT axioms.

The NetKAT axioms, shown in Fig. 2.2, consist of the axioms for Kleene Algebra (starting with KA-*), the axioms for a Boolean Algebra (starting with BA-*), and NetKAT-specific Packet Algebra axioms (starting with PA-*) describing the interaction between packet modifications and packet tests.

The Packet Algebra axioms say that modifications or filters on disparate fields commute (PA-MOD-MOD-COMM and PA-MOD-FILTER-COMM); filters commute with dup (PA-DUP-FILTER-COMM); modifying a packet field to a specific value and then testing for that same value is the same as only performing the modification, and vice versa (PA-MOD-FILTER and PA-FILTER-MOD); when modifying the same field twice, the second modification "wins" (PA-MOD-MOD); testing the same field for different values is always false (PA-CONTRA); and that the disjunction of all possible tests for a single field is always satisfied (PA-MATCH-ALL).

### 2.3.4 Language Model

We stated that the above axioms are sound and complete for NetKAT. Soundness is easy enough to show using the semantics, but proving completeness requires different tools. Following the standard approach, Anderson *et al.* [4] develop a *language model*, a semantics in which terms are interpreted as sets of "strings" (formally, elements in a monoid). For NetKAT, these are "reduced" strings of the form

$$\alpha p_0 \mathsf{dup} \pi_1 \mathsf{dup} \pi_2 \cdots \pi_{n-1} \mathsf{dup} \pi_n, \quad n \geq 0,$$

where $\alpha$ is a *complete test* $x_1 = n_1 \cdot \ldots \cdot x_k = n_k$, $\pi_i$ is a *complete assignment* $x_1 \leftarrow n_1 \cdot \ldots \cdot x_k \leftarrow n_k$ , and each of the fields is $x_k$ for exactly one $k$. We will write $\mathsf{At}$ for the set of complete tests, and $P$ for the set of complete assignments. The set of reduced strings is $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, where $\mathsf{dup}$ is the singleton set containing $\mathsf{dup}$; $A \cdot B$ denotes string concatenation (lifted to sets of strings)[2]; and $A^*$ denotes $\cup_i A^i$, where $A^{i+1} \triangleq A \cdot A^i$ and $A^0 \triangleq \{\epsilon\}$ for $\epsilon$ the empty string.

Every NetKAT expression is equivalent to a *reduced expression* in which every test is a complete test and every assignment is a complete assignment. The complete tests are the atoms (minimal nonzero elements) of the Boolean algebra generated by the primitive tests. Complete tests and complete assignments are in one-to-one correspondence.

The full language model for NetKAT is defined over the reduced expressions and is shown in Fig. 2.3. The language denoted by a complete test $\alpha$ is the singleton set containing the reduced string $\alpha \cdot \pi_\alpha$, where $\pi_\alpha$ is the complete assignment corresponding to $\alpha$. The language denoted by a complete assignment is the set of reduced string $\alpha \cdot \pi$ for every complete test

[2]Note: string concatenation is different from the guarded concatenation used in the language model. See Fig. 2.3 for guarded concatenation

$\alpha$. The language of $p + q$ is the union of the languages of $p$ and $q$; the language of a sequential composition is the guarded concatenation of the languages; and the language of $p^*$ is the union of all finite iterates of the language of $p^n$. The semantics of dup requires some explanation: dup is supposed to make a head copy of the head packet. Thus, it takes the head packet $\alpha$, and copies it across the place-holder dup as $\pi_\alpha$.

See the original paper on NetKAT [4] for a comprehensive treatment of the language model, including proofs of the claims above.

**Kleene Algebra Axioms**

| | | | |
|---|---|---|---|
| $p + (q + r)$ | $\equiv$ | $(p + q) + r$ | KA-PLUS-ASSOC |
| $p + q$ | $\equiv$ | $q + p$ | KA-PLUS-COMM |
| $p + \mathsf{0}$ | $\equiv$ | $p$ | KA-PLUS-ZERO |
| $p + p$ | $\equiv$ | $p$ | KA-PLUS-IDEM |
| $p \cdot (q \cdot r)$ | $\equiv$ | $(p \cdot q) \cdot r$ | KA-SEQ-ASSOC |
| $\mathsf{1} \cdot p$ | $\equiv$ | $p$ | KA-ONE-SEQ |
| $p \cdot \mathsf{1}$ | $\equiv$ | $p$ | KA-SEQ-ONE |
| $p \cdot (q + r)$ | $\equiv$ | $p \cdot q + p \cdot r$ | KA-SEQ-DIST-L |
| $(p + q) \cdot r$ | $\equiv$ | $p \cdot r + q \cdot r$ | KA-SEQ-DIST-R |
| $\mathsf{0} \cdot p$ | $\equiv$ | $\mathsf{0}$ | KA-ZERO-SEQ |
| $p \cdot \mathsf{0}$ | $\equiv$ | $\mathsf{0}$ | KA-SEQ-ZERO |
| $\mathsf{1} + p \cdot p^*$ | $\equiv$ | $p^*$ | KA-UNROLL-L |
| $q + p \cdot r \leq r$ | $\Longrightarrow$ | $p^* \cdot q \leq r$ | KA-LFP-L |
| $\mathsf{1} + p^* \cdot p$ | $\equiv$ | $p^*$ | KA-UNROLL-R |
| $p + q \cdot r \leq q$ | $\Longrightarrow$ | $p \cdot r^* \leq q$ | KA-LFP-R |

**Additional Boolean Algebra Axioms**

| | |
|---|---|
| $a + (b \cdot c) \equiv (a + b) \cdot (a + c)$ | BA-PLUS-DIST |
| $a + \mathsf{1} \equiv \mathsf{1}$ | BA-PLUS-ONE |
| $a + \neg a \equiv \mathsf{1}$ | BA-EXCL-MID |
| $a \cdot b \equiv b \cdot a$ | BA-SEQ-COMM |
| $a \cdot \neg a \equiv \mathsf{0}$ | BA-CONTRA |
| $a \cdot a \equiv a$ | BA-SEQ-IDEM |

**Packet Algebra Axioms**

| | |
|---|---|
| $f \leftarrow n \cdot f' \leftarrow n' \equiv f' \leftarrow n' \cdot f \leftarrow n, \text{ if } f \neq f'$ | PA-MOD-MOD-COMM |
| $f \leftarrow n \cdot f' = n' \equiv f' = n' \cdot f \leftarrow n, \text{ if } f \neq f'$ | PA-MOD-FILTER-COMM |
| $\mathsf{dup} \cdot f = n \equiv f = n \cdot \mathsf{dup}$ | PA-DUP-FILTER-COMM |
| $f \leftarrow n \cdot f = n \equiv f \leftarrow n$ | PA-MOD-FILTER |
| $f = n \cdot f \leftarrow n \equiv f = n$ | PA-FILTER-MOD |
| $f \leftarrow n \cdot f \leftarrow n' \equiv f \leftarrow n'$ | PA-MOD-MOD |
| $f = n \cdot f = n' \equiv \mathsf{0}, \text{ if } n \neq n'$ | PA-CONTRA |
| $\sum_i f = i \equiv \mathsf{1}$ | PA-MATCH-ALL |

Figure 2.2: NetKAT axioms

**Language model:** $G(p) \subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$

$$\begin{aligned}
G(\alpha) &= \{\alpha \cdot \pi_\alpha\} \\
G(\pi) &= \{\alpha \cdot \pi \mid \alpha \in \mathsf{At}\} \\
G(p + q) &= G(p) \cup G(q) \\
G(p \cdot q) &= G(p) \diamond G(q) \\
G(\mathsf{dup}) &= \{\alpha \cdot \pi_\alpha \cdot \mathsf{dup} \cdot \pi_\alpha \mid \alpha \in \mathsf{At}\} \\
G(p^*) &= \textstyle\bigcup_{n \geq 0} G(p^n)
\end{aligned}$$

**Guarded concatenation**

$$\begin{aligned}
\alpha \cdot p \cdot \pi \diamond \beta \cdot q \cdot \pi' &= \begin{cases} \alpha \cdot p \cdot q \cdot \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases} \\
A \diamond B &= \{p{\cdot}q \mid p \in A,\ q \in B\}
\end{aligned}$$

Figure 2.3: NetKAT language model

CHAPTER 3

# VERIFYING NETWORK PROGRAMS

*"Seek simplicity and distrust it."*

—Alfred North Whitehead

In this chapter, we introduce a novel specification language for network forwarding properties (Pathetic), and show how to build a formal verification tool that automatically analyzes NetKAT policies for correctness with respect to a Pathetic specification.

## 3.1 Introduction

Network configurations have long been a source of serious bugs and misbehavior in real-world networks. Written in arcane, poorly specified (often unspecified) languages, they defied meaningful verification. Before the development of a static configuration analyzer (the router configuration checker *rcc*) for BGP routers, the status quo in practice was run-time testing on operational networks [20]. Even *rcc* was only capable of detecting generic faults in configurations (*e.g.* learning unusable paths) and could not prove functional correctness. More recent verification tools (*e.g.* AntEater [65], HSA [50], or VeriFlow [53]) have focused on stronger correctness properties, but are based on *ad-hoc* foundations, and lack well-defined specification languages. For example, the NetPlumber [49] verifier includes a specification language for describing network paths, but the language itself does not have a semantics and lacks a precise description of what it means for a configuration to satisfy a specification.

This is not just a pedantic, academic point: without a precise semantics that describes the meaning of a specification and what it means to satisfy it, users cannot reason about the specification itself, nor can they have confidence in any purported verification performed against it. Moreover, anyone who builds a verifier based on such a language has no way of knowing whether they have even implemented it correctly.

In this chapter, we take a different approach. We present a specification language (Pathetic) with clear, precise formal semantics, and show how to use it to encode the requirements of an example network program. We then formally describe what it means for a network implementation (in the form of a NetKAT program) to satisfy a Pathetic specification, and show how to design and build a verifier that automatically decides satisfaction.

More specifically, in this chapter, we:

- Define the syntax and semantics of the Pathetic specification language
- Extend NetKAT with new operators (intersection and complement) to enable translation from Pathetic (NetKAT$(-, \cap)$)
- Define a semantics-preserving translation from Pathetic to NetKAT$(-, \cap)$
- Extend the equational and automata theories of NetKAT to NetKAT$(-, \cap)$
- Use the extended theories of NetKAT$(-, \cap)$ to build a decision procedure for Pathetic satisfaction

An early version of the Pathetic language appeared in [92].

## 3.2 Example

We will use the following running example through out the chapter to introduce Pathetic and demonstrate its features.

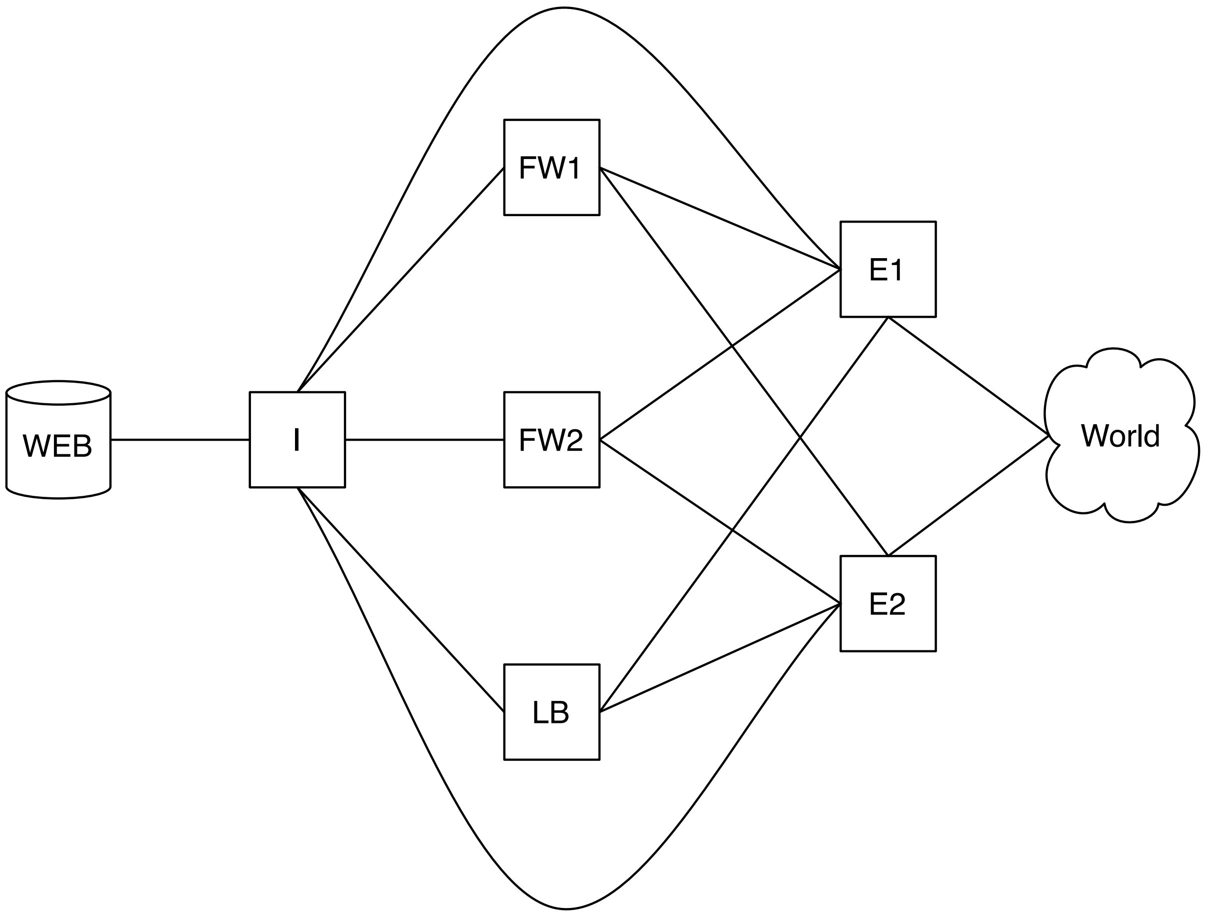


Figure 3.1: Example topology.

Consider the network shown in Figure 3.1. It consists of one attached network, World, a webserver Web, an ingress switch I, two firewalls FW1 and FW2, a load balancer LB, and two egress switches E1 and E2. This network is intended to provide connectivity between Web and World, subject to a security policy and a load balancing policy. The security policy, which we will denote by $\phi_{\mathsf{FW}}$, states that all outbound traffic (traffic originating at Web and destined for World) must traverse a firewall before reaching an egress switch. The load balancing policy, denoted by $\phi_{LB}$, states that all inbound traffic destined for the webserver (traffic originating in World and destined for an address denoted Web), must traverse the load balancer LB before reaching the ingress switch I.

The network is implemented with SDN switches, managed by a single controller $\mathsf{Controller}$ as shown in Figure 3.2[1]. For simplicity, we assume the switches are connected to the controller via a separate control network, independent from the network in the example. Because this chapter deals only with static network configurations and is agnostic to the mechanism used to implements them in the network, the techniques in this chapter are equally applicable when the switches are connected to the controller via the primary data network (so called "inband control").

The network is managed by an application, which receives network events (switches connecting, responses to queries, etc) and sends out network programs written in the network programming language NetKAT. A compiler takes the network programs, converts them into switch rules, and gives them to the controller to implement in the network.

To implement the load-balancing policy, the network application might initially install this NetKAT policy:

$$p_{\mathsf{LB}} \triangleq \quad tpDst = 80 \cdot nwDst = \mathsf{WEB} \cdot \begin{pmatrix} (sw = \mathsf{E1} + sw = \mathsf{E2}) \cdot pt \leftarrow \mathsf{LB} \\ +sw = \mathsf{LB} \cdot pt \leftarrow \mathsf{I} \\ +sw = \mathsf{I} \cdot pt \leftarrow \mathsf{WEB} \end{pmatrix}$$

Informally, this policy should be read as "$p_{\mathsf{LB}}$ is defined to be equal to the policy that matches packets with a destination port of 80 and a destination address of $\mathsf{WEB}$, and if the current switch is $\mathsf{E1}$ or $\mathsf{E2}$ then sends the packet to $\mathsf{LB}$, and if the current switch is $\mathsf{LB}$ then sends the packet to $\mathsf{I}$, and if the current switch is $\mathsf{I}$ then sends the packet to $\mathsf{WEB}$".

We briefly review the syntax and semantics of NetKAT here, but for full details, see

[1]For review of what an SDN network is, see Section 2.2

Chapter 2 or Anderson *et al.* [4]. The policy consists of several terms, composed with the sequential composition operator $\cdot$ and the parallel composition operator $+$. The first term is a predicate (or filter), $\mathit{tpDst} = 80$, that tests the destination port ($\mathit{tpDst}$) of packets, letting them pass through if it is equal to 80, or dropping them if not. This test is sequentially composed with another predicate, $\mathit{nwDst} = \mathsf{WEB}$, that tests the destination IP address for equality with the web server $\mathsf{WEB}$'s IP address. Notice that sequentially composing predicates is equivalent to taking their conjunction. Next, we compose these two predicates with the parallel composition of several terms. The first term in the parallel composition (or union) matches packets that are at either $\mathsf{E1}$ or $\mathsf{E2}$ (parallel composition of predicates is the same as their disjunction), and sends them to the port connected to $\mathsf{LB}$. Similarly, the next terms matches packets on $\mathsf{LB}$ and sends them to the port connected to $\mathsf{I}$. Finally, the last term matches packets that have arrived at $\mathsf{I}$ and sends them to the port connected to $\mathsf{WEB}$.

This policy describes the switch forwarding behavior, but NetKAT programs actually encode the full network behavior, including the topology. Generally, the topology term is provided by the controller and composed with the switch forward term. A snippet of the topology term for our network would be written as follows:

$$\begin{aligned} t \triangleq\ & sw = \mathsf{E1} \cdot pt = \mathsf{FW1} \cdot sw \leftarrow \mathsf{FW1} \cdot pt \leftarrow \mathsf{E1} \\ & + sw = \mathsf{E1} \cdot pt = \mathsf{FW2} \cdot sw \leftarrow \mathsf{FW2} \cdot pt \leftarrow \mathsf{E1} \\ & + sw = \mathsf{E1} \cdot pt = \mathsf{LB} \cdot sw \leftarrow \mathsf{LB} \cdot pt \leftarrow \mathsf{E1} \\ & + sw = \mathsf{E1} \cdot pt = \mathsf{I} \cdot sw \leftarrow \mathsf{I} \cdot pt \leftarrow \mathsf{E1} \\ & \ldots \end{aligned}$$

To construct the term describing the full network, we would compose the switch forwarding term, the term $\mathsf{dup}$, and the topology term, and iterate them using the $^*$ operator:

$$(p_{\mathsf{LB}} \cdot \mathsf{dup} \cdot t)^*$$

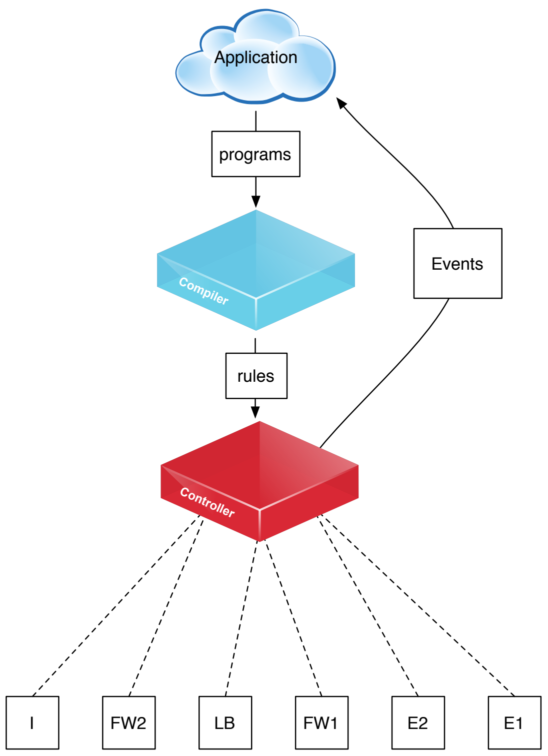


Figure 3.2: Running example control architecture.

The constant `dup` and requires explanation. NetKAT's semantics is given as a function from *histories* (non-empty lists of packet headers) to sets of *histories*. Atomic NetKAT terms (such as header predicates and modifications) work on the head packet of the current history. The operation `dup` duplicates the head packet, putting the new copy on the top of the list. This is how histories remember past values of packets, and how the semantics models paths through the network. In practice, programmers do not program with `dup` directly; it is instead inserted in the appropriate place along with the topology term.

## 3.3 The Pathetic specification language

In this section we introduce Pathetic, a specification language based on regular expressions over network paths with wildcards, and demonstrate its use through the running example introduced in the previous section. While Pathetic is, in one sense, strictly *less* expressive than NetKAT[2], it is in fact specialized to serve a different role. In Pathetic, alternation is interpreted *disjunctively* (*i.e.* choose one of the following), while in NetKAT alternation is interpreted *conjunctively* (*i.e.* perform *all* of the following). Thus, a Pathetic program can specify multiple possible NetKAT implementations. As the examples in this chapter will show, by utilizing the wildcard operator and the iteration operator, a Pathetic program can succinctly describe only the essential details of a path that matter for correctness.

**Load balancing policy** Consider the load balancer requirement from the running example: all web traffic destined for the server $\mathsf{WEB}$ must traverse $\mathsf{LB}$ before reaching the ingress switch $\mathsf{I}$. We can write down this specification in Pathetic as $\phi_{\mathsf{LB}}$:

$$\begin{aligned} \phi_{\mathsf{LB}} &\triangleq \phi_{\mathsf{WEB}} \uplus \phi_{\neg\mathsf{WEB}} \\ &\quad \text{where} \\ \phi_{\mathsf{WEB}} &\triangleq (\mathit{tpDst} = 80 \cdot \mathit{nwDst} = \mathsf{WEB}) \Rightarrow (\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB} \\ \phi_{\neg\mathsf{WEB}} &\triangleq \neg(\mathit{tpDst} = 80 \cdot \mathit{nwDst} = \mathsf{WEB}) \Rightarrow \star^* \end{aligned}$$

The policy $\phi_{\mathsf{LB}}$ is written as the union of two *atomic* Pathetic policies, $\phi_{\mathsf{WEB}}$ and $\phi_{\text{other}}$. Atomic policies such as $\phi_{\mathsf{WEB}}$ consist of a predicate ($\mathit{tpDst} = 80 \cdot \mathit{nwDst} = \mathsf{WEB}$) describing the packets that the policy applies to, and a path expression ($(\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}$) describing

[2]All Pathetic programs are translatable into semantically NetKAT programs, but not vice versa

the valid paths. Predicates are expressed using the syntax of the NetKAT predicate language. ($\mathit{tpDst} = 80 \cdot \mathit{nwDst} = \mathsf{WEB}$) matches web traffic ($\mathit{tpDst} = 80$) that is destined for the web server ($\mathit{nwDst} = \mathsf{WEB}$). The path expression consists of four parts, joined together with the sequence operator ".". In the first part, we use the shorthand $\neg\mathsf{WEB}$ (formally equal to $\star \cap \overline{\mathsf{WEB}}$ where $\star$ is a wildcard denoting any path of length one), which matches all paths of length one other than $[\mathsf{WEB}]$. The policy $P^*$ denotes iteration zero or more times of the path expression $P$, thus $(\neg\mathsf{WEB})^*$ matches paths of any length that do not traverse $\mathsf{WEB}$. After this initial path, the packet must traverse $\mathsf{LB}$ and then is allowed to take any path ($\star^*$) that ends at $\mathsf{WEB}$.

Finally, because Pathetic policies denote total specifications of network behavior (unmatched packets get dropped), we union this with a fall-through policy $\phi_{\neg\mathsf{WEB}}$ that allows non-web traffic to take any path through the network.

**Firewall policy** Similarly, we can write down the policy requiring that all outbound traffic traverse either $\mathsf{FW1}$ or $\mathsf{FW2}$ before leaving the network by decomposing it into three parts. First, all packets ($\mathsf{1}$ denotes the predicate "true") are allowed to take any path that ends inside the network ($\neg(\mathsf{E1}|\mathsf{E2})$):

$$\phi_{internal} \triangleq \mathsf{1} \Rightarrow (\star^*).\neg(\mathsf{E}_1|\mathsf{E}_2)$$

Second, all traffic starting inside the network is allowed to take a path that traverses $\mathsf{FW1}$ or $\mathsf{FW2}$ before leaving the network (via one of the egress switches):

$$\phi_{internal-external} \triangleq \mathsf{1} \Rightarrow (\neg(\mathsf{E}_1|\mathsf{E}_2))^*.(\mathsf{FW1}|\mathsf{FW2}).(\star^*).(\mathsf{E}_1|\mathsf{E}_2)$$

Third, we allow traffic between the egress switches to take arbitrary paths between them:

$$\phi_{external-external} \triangleq \mathsf{1} \Rightarrow (\mathsf{E}_1|\mathsf{E}_2).(\star^*).(\mathsf{E}_1|\mathsf{E}_2)$$

Finally, we combine these policies with the union operator to allow any of the paths to be used:

$$\phi_{\mathsf{FW}} \triangleq \phi_{internal} \uplus \phi_{internal-external} \uplus \phi_{external-external}$$

**Combined example** To enforce both the firewall policy and the previous load balancing policy, we combine them using the intersection operator, to impose the requirements of both policies:

$$\phi_{\mathsf{FW-LB}} \triangleq \phi_{\mathsf{FW}} \mathbin{\cap\!\!\!+} \phi_{\mathsf{LB}}$$

The resulting policy $\phi_{\mathsf{FW-LB}}$ enforces that all outbound traffic traverses the firewall, and that all inbound web traffic traverses the load balancer before arriving at the webserver.

### 3.3.1 Pathetic syntax and semantics

The syntax of Pathetic programs is shown in Figure 3.3a. Atomic Pathetic programs ($a \Rightarrow P$) consist of two parts: a regular expression over network elements describing a set of valid paths ($P$), and a predicate defining the set of packets that the regular expression applies to ($a$). Atomic path regular expressions are either the empty path ($\epsilon$), the empty set of paths ($\emptyset$), a constant path of length one (S for some switch S) or a wildcard path of length 1 ($\star$). Regular expressions can be combined using sequential composition ($P.P'$), non-deterministic choice ($P|P'$), complement ($\overline{P}$), conjunction ($P \cap P'$), or iteration ($P^*$). Compound Pathetic programs are constructed with the union of two programs ($\phi \uplus \phi'$), which denotes the disjunction of the restrictions of $\phi$ and $\phi'$, or intersection ($\phi \mathbin{\cap\!\!\!+} \phi'$), which denotes the conjunction of the restrictions of $\phi$ and $\phi'$.

**Syntax**

| | | | |
|---|---|---|---|
| Path exp. | $P ::=$ | $\epsilon$ | *Empty path* |
| | $\mid$ | $\emptyset$ | *Empty set* |
| | $\mid$ | $S$ | *Explicit switch S* |
| | $\mid$ | $\star$ | *Wildcard hop* |
| | $\mid$ | $\overline{P}$ | *Complement* |
| | $\mid$ | $P.P$ | *Sequencing* |
| | $\mid$ | $P \vert P$ | *Alternation* |
| | $\mid$ | $P \cap P$ | *Intersection* |
| | $\mid$ | $P^*$ | *Iteration* |
| Fields | $f ::=$ | $f_1 \mid \cdots \mid f_k$ | |
| Predicates | $a, b ::=$ | $\mathsf{1}$ | *Identity* |
| | $\mid$ | $\mathsf{0}$ | *Drop* |
| | $\mid$ | $f = n$ | *Test* |
| | $\mid$ | $a + b$ | *Disjunction* |
| | $\mid$ | $a \cdot b$ | *Conjunction* |
| | $\mid$ | $\neg a$ | *Negation* |
| Program | $\phi ::=$ | $a \Rightarrow P$ | *atomic policy* |
| | $\mid$ | $\phi_1 \uplus \phi_2$ | *policy union* |
| | $\mid$ | $\phi_1 \mathbin{⩀} \phi_2$ | *policy intersection* |

(a)

**Path semantics**

$$
\begin{aligned}
[\![P]\!] &\in 2^{\texttt{Sw}^*} \\
[\![\epsilon]\!] &\triangleq \{[]\} \\
[\![\emptyset]\!] &\triangleq \{\} \\
[\![S]\!] &\triangleq \{[S]\} \\
[\![\star]\!] &\triangleq \{[S] \mid S \in \texttt{Sw}\} \\
[\![\overline{P}]\!] &\triangleq 2^{\texttt{Sw}^*} \setminus [\![P]\!] \\
[\![P.P']\!] &\triangleq [\![P]\!] \diamond [\![P']\!] \\
[\![P|P']\!] &\triangleq [\![P]\!] \cup [\![P']\!] \\
[\![P \cap P']\!] &\triangleq [\![P]\!] \cap [\![P']\!] \\
[\![P^*]\!] &\triangleq \textstyle\bigcup_{n \geq 0} F^n
\end{aligned}
$$

where $F^0 = \{[]\}$
and $F^{i+1} = [\![P]\!] \diamond F^i$
and $A \diamond B = \{ab \mid a \in A \wedge b \in B\}$

$$
\begin{aligned}
[\![\phi]\!] &\in \mathcal{PK} \rightarrow 2^{\texttt{Sw}^*} \\
[\![a \Rightarrow P]\!]\ pk &\triangleq \begin{cases} [\![P]\!] & \text{if } [\![a]\!]\ pk \\ \{\} & \text{o.w.} \end{cases} \\
[\![\phi_1 \uplus \phi_2]\!]\ pk &\triangleq [\![\phi_1]\!]\ pk \cup [\![\phi_2]\!]\ pk \\
[\![\phi_1 \mathbin{⩀} \phi_2]\!]\ pk &\triangleq [\![\phi_1]\!]\ pk \cap [\![\phi_2]\!]\ pk
\end{aligned}
$$

(b)

Figure 3.3: Pathetic syntax and semantics.

**Pathetic semantics** The semantics of Pathetic is shown in Figure 3.3b. Pathetic programs denote functions from packets to sets of allowable paths[3]. $\phi \uplus \phi'$ denotes the point-wise union of $\phi$ and $\phi'$, and $\phi \mathbin{⩀} \phi'$ denotes the point-wise intersection.

Let's look at $\phi_{\mathsf{LB}}$ from our running example, and see how it behaves on a web packet destined for $\mathsf{WEB}$:

[3]Notice that this rules out the possibility of packet modifications

$$
\begin{aligned}
[\![\phi_{\mathsf{LB}}]\!]\ pk &= [\![\phi_{\mathsf{WEB}}]\!]\ pk \cup [\![\phi_{\neg\mathsf{WEB}}]\!]\ pk \\
&= \begin{cases} [\![(\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}]\!] & \text{if } [\![(tpDst = 80 \cdot nwDst = \mathsf{WEB})]\!]\ pk \\ [\![\star^*]\!] & \text{o.w.} \end{cases} \\
&= [\![(\neg\mathsf{WEB})^*.\mathsf{LB}.(\star^*).\mathsf{WEB}]\!] \\
&= [\![(\neg\mathsf{WEB})^*]\!] \diamond [\![\mathsf{LB}]\!] \diamond [\![(\star^*)]\!] \diamond [\![\mathsf{WEB}]\!] \\
&= [\![(\star \cap \overline{\mathsf{WEB}})^*]\!] \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \cup_i [\![\star \cap \overline{\mathsf{WEB}}]\!]^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \cup_i \left([\![\star]\!] \cap [\![\overline{\mathsf{WEB}}]\!]\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \cup_i \left(\mathtt{Sw} \cap (\mathtt{Sw}^* \setminus \{[\mathsf{WEB}]\})\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \cup_i \left(\mathtt{Sw} \setminus \{[\mathsf{WEB}]\}\right)^i \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \{P \mid \mathsf{WEB} \notin P\} \diamond \{[\mathsf{LB}]\} \diamond [\![(\star^*)]\!] \diamond \{[\mathsf{WEB}]\} \\
&= \{P{\cdot}[\mathsf{LB}]{\cdot}P'{\cdot}[\mathsf{WEB}] \mid \mathsf{WEB} \notin P\}
\end{aligned}
$$

So, this gives us the set of paths that take any route to LB not through WEB, and then take any route to WEB, which is exactly what we wanted.

### 3.3.2 Relating Pathetic to NetKAT

This section assumes familiarity with the NetKAT semantics language model. For more details, see Chapter 2.

Similar to NetKAT, Pathetic's semantics gives rise to a derived language model. To see

**Language model**

$$
\begin{aligned}
G(P) &\subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \\
G(\epsilon) &\triangleq \{S{\cdot}\pi_S \mid S \in \mathtt{Sw}\} \\
G(\emptyset) &\triangleq \{\} \\
G(S) &\triangleq \{S'{\cdot}\pi_S{\cdot}\mathsf{dup}{\cdot}\pi_S \mid S' \in \mathtt{Sw}\} \\
G(\star) &\triangleq \{S{\cdot}\pi_{S'}{\cdot}\mathsf{dup}{\cdot}\pi_{S'} \mid S, S' \in \mathtt{Sw}\} \\
G(\overline{P}) &\triangleq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(P) \\
G(P.P') &\triangleq G(P) \diamond G(P') \\
G(P|P') &\triangleq G(P) \cup G(P') \\
G(P \cap P') &\triangleq G(P) \cap G(P') \\
G(P^*) &\triangleq \textstyle\bigcup_{n \geq 0} F^n \\
&\text{where } F^0 = \{S{\cdot}S'\} \\
&\text{and } F^{i+1} = G(P) \diamond F^i
\end{aligned}
$$

$$
\begin{aligned}
G(a \Rightarrow P) &\triangleq G(a) \diamond G(P) \\
G(\phi_1 \uplus \phi_2) &\triangleq G(\phi_1) \cup G(\phi_2) \\
G(\phi_1 \Cap \phi_2) &\triangleq G(\phi_1) \cap G(\phi_2)
\end{aligned}
$$

Figure 3.4: Pathetic language model

where the language model comes from, note the isomorphism

$$[\![\phi]\!] \in \mathcal{PK} \to 2^{\mathtt{Sw}^*} \cong \mathcal{PK} \to \mathtt{Sw}^* \to 2 \cong \mathcal{PK} \times \mathtt{Sw}^* \to 2 \cong 2^{\mathcal{PK} \times \mathtt{Sw}^*}$$

.

Recall the language model of NetKAT of reduced strings of the form $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, where $\mathsf{At}$ is the set of complete tests on packets (*i.e.* predicates that match exactly one packet); $P$ is the set of complete assignments on packets (*i.e.* a sequence of modifications on packet headers such that all output packets are the same, regardless of the input packet); $\mathsf{dup}$ is the singleton set containing $\mathsf{dup}$; $A \cdot B$ denotes string concatenation (lifted to sets of strings); and $A^*$ denotes $\cup_i A^i$, where $A^{i+1} \triangleq A \cdot A^i$ and $A^0 \triangleq \{\epsilon\}$ for $\epsilon$ the empty string.

We can define a projection from the language model of NetKAT ($\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$) to $\mathcal{PK} \times \mathtt{Sw}^*$ by noting that the set of packets is isomorphic to the set of complete packet tests ($\mathcal{PK} \cong \mathsf{At}$), projecting out the non-switch header values of each $P$, and dropping $\mathsf{dup}$. Conversely, we can lift an element of $2^{\mathcal{PK} \times \mathtt{Sw}^*}$ to a subset of $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$ by extending each switch $\mathsf{S}$ in a path to a complete assignment (where the switch field in the assignment is set to $\mathsf{S}$), and taking the union over every possible such extension.

To illustrate this construction, consider the case where we have only one header field: *tpDst* (which takes values 0 or 1), and the switch field *pt*. Here, the element of the NetKAT language model $\{(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0)\}$ would get mapped to $\{tpDst = 0 \times (\mathsf{T} \cdot \mathsf{U})\}$. Similarly, this element gets mapped back into the NetKAT model as

$$\begin{aligned}
\{&(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0),\\
&(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 1) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 0),\\
&(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 0) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 1),\\
&(pt = \mathsf{S} \cdot tpDst = 0) \cdot (pt \leftarrow \mathsf{T} \cdot tpDst \leftarrow 1) \cdot \mathsf{dup} \cdot (pt \leftarrow \mathsf{U} \cdot tpDst \leftarrow 1)\}
\end{aligned}$$

The resulting language model for Pathetic is shown in Figure 3.4. Because path expressions only operate on the *sw* header of packets, we elide the other fields, and implicitly perform the extension described above. $S$ denotes the test $sw = \mathsf{S}$, and $\pi_S$ the matching assignment $sw \leftarrow \mathsf{S}$.

Now that Pathetic and NetKAT have a common semantic domain, we can define what it means for a NetKAT program to satisfy a specification. Intuitively, we want this to be true iff every possible path the program forwards packets on is allowed by $\phi$:

**Definition 1.** *$p$ satisfies $\phi$, written $p \vDash \phi$, iff $G(p) \subseteq [\![\phi]\!]$, where $G(p)$ is the language model of NetKAT (see Figure 2.3).*

## 3.4 NetKAT$(-,\cap)$

Rather than develop a one-off verifier for Pathetic and NetKAT, we extend NetKAT to a richer language NetKAT$(-,\cap)$ that we can translate Pathetic into, and then implement an equivalence checker for NetKAT$(-,\cap)$. We then use this equivalence checker to verify satisfaction.

**Equivalence checking** A tool that checks equivalence is a powerful basis for verification. It is well-known in language theory that many useful problems can be reduced to equivalence checking. To convince the reader of the utility of this approach, we outline a couple of examples here.

The Pathetic language is useful for restricting the valid paths a packet can take, but sometimes a simpler specification such as *connectivity* is desired: every host in the network should be able to communicate with every other host. As Foster *et al.* showed in [25], connectivity of a policy $p$ can be specified and verified directly in NetKAT itself with an equivalence checker. Because connectivity does not care about specific paths (only that a path exists), we first replace all instances of $\mathsf{dup}$ in $p$ with $\mathsf{1}$. This gives us a policy with the same connectivity behavior, but where packets "magically appear" at their destination with no record of the path (history) taken through the network. We write this mapping $\Phi(p)$. Then, we check its equivalence against a term encoding the end-to-end forwarding behavior of a connected network:

$$\Phi(p) \quad \equiv \sum_{(sw,sw',pt,pt')} \begin{pmatrix} \mathit{switch} = sw \cdot \mathit{port} = pt \ \cdot \\ \mathit{switch} \leftarrow sw' \cdot \mathit{port} \leftarrow pt' \end{pmatrix}$$

where $(sw, pt)$ and $(sw', pt')$ range over all host-facing ports in the network. One advantage of performing this analysis directly in NetKAT is that both terms are $\mathsf{dup}$-free, which leads to a much more efficient verification than checking the full program (why this is true will become clear in later sections).

Similarly, we can use equivalence checking to implement translation validation and check the correctness of the output of a compiler. The NetKAT compiler translates NetKAT terms into a sequence of OpenFlow forwarding rules for each switch. These rules can be directly encoded back into NetKAT as a cascade of conditional rules:

$$\begin{aligned} c \;=\;& \mathsf{if}\ \mathit{pat_1}\ \mathsf{then}\ \mathit{acts_1}\ \mathsf{else} \\ & \quad \ldots \\ & \mathsf{if}\ \mathit{pat_k}\ \mathsf{then}\ \mathit{acts_k}\ \mathsf{else}\ 0 \end{aligned}$$

where each $pat_i$ is a positive conjunction of tests and each $act_i$ is a sequence of modifications. To verify equivalence, we can check if $p \equiv (c \cdot t)^*$, where $t$ is a term encoding the topology of the network.

For more examples of encodings of NetKAT verification problems that use equivalence checking, see Foster *et al.* [25].

### 3.4.1 NetKAT$(-, \cap)$ syntax and semantics

To translate Pathetic, we have extended NetKAT with complement ($\overline{p}$) and intersection ($p \cap q$), as shown in Figure 3.5a. We call this extended language NetKAT$(-, \cap)$. The semantics (Figure 3.5b) of the two new operators is the obvious interpretation, except for the denotation of complement in the history semantics. Instead of defining $[\![\overline{p}]\!]\ \pi{::}h$ as the complement of $[\![p]\!]\ \pi{::}h$ with respect to the full set of histories, it is instead the complement with respect to all histories with the same past $h$ as $\pi{::}h$, $\mathcal{H}\mid_h$. The reason for this change becomes clear when you consider the relation between the history semantics and the language

| | | | |
|---|---|---|---|
| Fields | $f ::=$ | $f_1 \mid \cdots \mid f_k$ | |
| Predicates | $a, b ::=$ | $\mathsf{1}$ | *Identity* |
| | $\mid$ | $\mathsf{0}$ | *Drop* |
| | $\mid$ | $f = n$ | *Test* |
| | $\mid$ | $a + b$ | *Disjunction* |
| | $\mid$ | $a \cdot b$ | *Conjunction* |
| | $\mid$ | $\neg a$ | *Negation* |
| Policies | $p, q ::=$ | $a$ | *Filter* |
| | $\mid$ | $f \leftarrow n$ | *Modification* |
| | $\mid$ | $\overline{p}$ | *Complement* |
| | $\mid$ | $p \cdot q$ | *Sequence* |
| | $\mid$ | $p + q$ | *Union* |
| | $\mid$ | $p \cap q$ | *Intersection* |
| | $\mid$ | $p^*$ | *Kleene star* |
| | $\mid$ | $\mathsf{dup}$ | *Duplication* |

(a) NetKAT$(-, \cap)$ syntax.

$$
\begin{aligned}
[\![p]\!] &\in \mathcal{H} \to \mathcal{P}(\mathcal{H}) \\
[\![\mathsf{1}]\!]\ h &\triangleq \{h\} \\
[\![\mathsf{0}]\!]\ h &\triangleq \emptyset \\
[\![f = n]\!]\ \pi{::}h &\triangleq \begin{cases} \{\pi{::}h\} & \text{if } \pi.f = n \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h &\triangleq \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ \pi{::}h &\triangleq \{\pi[f \mapsto n]{::}h\} \\
[\![\overline{p}]\!]\ \pi{::}h &\triangleq \mathcal{H} \mid_h \setminus [\![p]\!]\ \pi{::}h \\
[\![p \cdot q]\!]\ h &\triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p + q]\!]\ h &\triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cap q]\!]\ h &\triangleq [\![p]\!]\ h \cap [\![q]\!]\ h \\
[\![p^*]\!]\ h &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ h \\
\text{where } F^0\ h \triangleq \{h\} &\text{ and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\mathsf{dup}]\!]\ (\pi{::}h) &\triangleq \{\pi{::}(\pi{::}h)\}
\end{aligned}
$$

(b) NetKAT$(-, \cap)$ semantics.

Figure 3.5: NetKAT$(-, \cap)$.

model: when you sequentially compose two terms, the second term does not get to "rewrite" history in the language model (it can only extend or discard it).

Once we have defined NetKAT$(-, \cap)$, the translation from Pathetic into NetKAT$(-, \cap)$ (written $(\!|\phi|\!)$ and shown in Figure 3.6) is fairly straightforward.

**Theorem 1** (Equivalence of NetKAT translation). *For every Pathetic program $\phi$, $G(\phi) = G((\!|\phi|\!))$*

**Theorem 2.** *$p \vDash \phi$ iff $(\!|\phi|\!) \cap \overline{p} \equiv \mathsf{0}$.*

**Corollary 1.** *$p \vDash \phi$ iff $p \leq (\!|\phi|\!)$.*

$$\begin{aligned}
(\!|\epsilon|\!) &\triangleq \mathsf{1}\\
(\!|\emptyset|\!) &\triangleq \mathsf{0}\\
(\!|S|\!) &\triangleq sw \leftarrow S \cdot \mathsf{dup}\\
(\!|\star|\!) &\triangleq \textstyle\sum_{S' \in \mathtt{Sw}} sw \leftarrow S' \cdot \mathsf{dup}\\
(\!|\overline{P}|\!) &\triangleq \overline{(\!|P|\!)}\\
(\!|P.P'|\!) &\triangleq (\!|P|\!) \cdot (\!|P'|\!)\\
(\!|(P|P')|\!) &\triangleq (\!|P|\!) + (\!|P'|\!)\\
(\!|(P \cap P')|\!) &\triangleq [\![P]\!] \cap [\![P']\!]\\
(\!|P^*|\!) &\triangleq (\!|P|\!)^*\\
(\!|pr \Rightarrow P|\!) &\triangleq pr \cdot (\!|P|\!)\\
(\!|\phi_1 \uplus \phi_2|\!) &\triangleq (\!|\phi_1|\!) + (\!|\phi_2|\!)\\
(\!|\phi_1 \Cap \phi_2|\!) &\triangleq (\!|\phi_1|\!) \cap (\!|\phi_2|\!)
\end{aligned}$$

Figure 3.6: Translation from Pathetic into NetKAT$(-,\cap)$

**Packet semantics**

$$\begin{aligned}
[\![p]\!]_{-\mathsf{dup}} &\in \mathcal{PK} \to \mathcal{P}(\mathcal{PK})\\
[\![\mathsf{1}]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi\}\\
[\![\mathsf{0}]\!]_{-\mathsf{dup}}\ \pi &\triangleq \emptyset\\
[\![f = n]\!]_{-\mathsf{dup}}\ \pi &\triangleq \begin{cases}\{\pi\} & \text{if } \pi.f = n\\ \emptyset & \text{otherwise}\end{cases}\\
[\![\neg a]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi\} \setminus ([\![a]\!]_{-\mathsf{dup}}\ \pi)\\
[\![f \leftarrow n]\!]_{-\mathsf{dup}}\ \pi &\triangleq \{\pi[f \mapsto n]\}\\
[\![\overline{p}]\!]_{-\mathsf{dup}}\ \pi &\triangleq \mathcal{PK} \setminus [\![p]\!]_{-\mathsf{dup}}\ \pi\\
[\![p \cdot q]\!]_{-\mathsf{dup}}\ \pi &\triangleq ([\![p]\!]_{-\mathsf{dup}} \bullet [\![q]\!]_{-\mathsf{dup}})\ \pi\\
[\![p + q]\!]_{-\mathsf{dup}}\ \pi &\triangleq [\![p]\!]_{-\mathsf{dup}}\ \pi \cup [\![q]\!]_{-\mathsf{dup}}\ \pi\\
[\![p \cap q]\!]_{-\mathsf{dup}}\ \pi &\triangleq [\![p]\!]_{-\mathsf{dup}}\ \pi \cap [\![q]\!]_{-\mathsf{dup}}\ \pi\\
[\![p^*]\!]_{-\mathsf{dup}}\ \pi &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ \pi
\end{aligned}$$

where $F^0\ \pi \triangleq \{\pi\}$ and $F^{i+1}\ \pi \ \triangleq\ ([\![p]\!]_{-\mathsf{dup}} \bullet F^i)\ \pi$

**dup-free Language model**

$$\begin{aligned}
G_{-\mathsf{dup}}(p) &\subseteq \mathsf{At} \cdot P\\
G_{-\mathsf{dup}}(\alpha) &\triangleq \{\alpha{\cdot}\pi_\alpha\}\\
G_{-\mathsf{dup}}(\pi) &\triangleq \{\alpha{\cdot}\pi \mid \alpha \in \mathsf{At}\}\\
G_{-\mathsf{dup}}(\overline{p}) &\triangleq \mathsf{At} \cdot P \setminus G_{-\mathsf{dup}}(p)\\
G_{-\mathsf{dup}}(p \cdot q) &\triangleq G_{-\mathsf{dup}}(p) \diamond G_{-\mathsf{dup}}(q)\\
G_{-\mathsf{dup}}(p + q) &\triangleq G_{-\mathsf{dup}}(p) \cup G_{-\mathsf{dup}}(q)\\
G_{-\mathsf{dup}}(p \cap q) &\triangleq G_{-\mathsf{dup}}(p) \cap G_{-\mathsf{dup}}(q)\\
G_{-\mathsf{dup}}(p^*) &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} G_{-\mathsf{dup}}(p^i)
\end{aligned}$$

Figure 3.7: NetKAT$(-,\cap)$ $\mathsf{dup}$-free semantics and language model.

**Language model**

$$
\begin{aligned}
G(p) &\subseteq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \\
G(\alpha) &\triangleq \{\alpha{\cdot}\pi_\alpha\} \\
G(\pi) &\triangleq \{\alpha{\cdot}\pi \mid \alpha \in \mathsf{At}\} \\
G(\overline{p}) &\triangleq \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(p) \\
G(p \cdot q) &\triangleq G(p) \diamond G(q) \\
G(p + q) &\triangleq G(p) \cup G(q) \\
G(p \cap q) &\triangleq G(p) \cap G(q) \\
G(\mathsf{dup}) &\triangleq \{\alpha{\cdot}\pi_\alpha{\cdot}\mathsf{dup}{\cdot}\pi_\alpha \mid \alpha \in \mathsf{At}\} \\
G(p^*) &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} G(p^i)
\end{aligned}
$$

Figure 3.8: NetKAT($-, \cap$) language model.

### 3.4.2 NetKAT($-, \cap$) equational theory

In this section, we extend the equational theory of NetKAT to NetKAT($-, \cap$), and prove our axioms complete for a restricted subset. The only content of this section that the reader should know to understand the rest of the thesis are the axioms themselves, presented in Figure 3.9. The rest of the section is not essential to understand this chapter, and can safely be skipped by the reader.

The original NetKAT language has a sound and complete equational theory, based on the equational theory of Kleene Algebra with test (KAT) [57]. Because NetKAT($-, \cap$) is a conservative extension of NetKAT, the NetKAT equational theory is sound, but not complete for NetKAT($-, \cap$). Unfortunately, we conjecture that there is no finite extension of the NetKAT axioms that is sound and complete for NetKAT($-, \cap$) (Conjecture 1).

Instead, in this section we give a sound and complete axiomatization of the dup-free fragment of NetKAT($-, \cap$), as shown in Figure 3.9. The semantics of dup-free NetKAT($-, \cap$)

$$
\begin{array}{rcll}
\mathsf{all} & \triangleq & \sum_{\beta} \pi_{\beta} & \\
p \cap q & \equiv & q \cap p & \text{INTER-COMM} \\
p \cap (q \cap r) & \equiv & (q \cap p) \cap r & \text{INTER-ASSOC} \\
p \cap p & \equiv & p & \text{INTER-IDEM} \\
a \cap b & \equiv & a \cdot b & \text{INTER-FILTER} \\
f \leftarrow n \cap a & \equiv & f = n \cdot a & \text{INTER-MOD-FILTER} \\
f \leftarrow n \cap f' \leftarrow n' & \equiv & (f \leftarrow n \cdot f' = n') + (f' \leftarrow n' \cdot f = n) & \text{INTER-MOD-MOD} \\
(p + q) \cap r & \equiv & (p \cap r) + (q \cap r) & \text{INTER-PAR-DIST} \\
(p \cap q) + r & \equiv & (p + r) \cap (q + r) & \text{PAR-INTER-DIST} \\
(a \cdot p) \cap q & \equiv & a \cdot (p \cap q) & \text{INTER-FILTER-DIST-LEFT} \\
(p \cdot a) \cap q & \equiv & (p \cap q) \cdot a & \text{INTER-FILTER-DIST-RIGHT} \\
f \leftarrow n \cdot (p \cap q) & \equiv & (f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q) & \text{INTER-MOD-DIST-LEFT} \\
\overline{f = n} & \equiv & \neg f = n \cdot \sum_{\pi} \pi + \sum_{\alpha} \sum_{\pi \neq \pi_{\alpha}} \alpha \cdot \pi & \text{COMP-FILTER}^{*} \\
\overline{f \leftarrow n} & \equiv & \sum_{\alpha} \sum_{\pi \neq \pi_{\alpha[f \leftarrow n]}} \alpha \cdot \pi & \text{COMP-MOD}^{*} \\
\overline{p + q} & \equiv & \overline{p} \cap \overline{q} & \text{COMP-PAR} \\
\overline{p \cap q} & \equiv & \overline{p} + \overline{q} & \text{COMP-INTER} \\
\overline{p \cdot q} & \equiv & \bigcap_{\gamma} (\overline{p} \cdot \gamma \cdot \mathsf{all} + \pi_{\gamma} \cdot \overline{q}) & \text{COMP-SEQ}^{*}
\end{array}
$$

Figure 3.9: NetKAT$(-, \cap)$ Axioms. Axioms labeled with * are only valid in the $\mathsf{dup}$-free fragment

is given by a semantics over packets, instead of histories, and is shown in Figure 3.7.

For this axiomatization, we use $\alpha, \beta$ as short-hand to denote *complete tests*, a conjunction of header tests $f = n$ such that every header $f$ appears exactly once in a specific order. Similarly, we use $\pi, \gamma$ to denote *complete assignments*, a sequence of header modifications $f \leftarrow n$ such that every header $f$ appears exactly once. We write $\alpha_{\pi}$ to denote the complete test corresponding to $\pi$ (*i.e.* the test that matches exactly the packet containing the header values set by $\pi$), and $\pi_{\alpha}$ to denote the complete assignment corresponding to $\alpha$. Note that because the set of headers and the set of header values are both finite, sums over the set of complete tests and complete assignments are also finite, and thus valid short-hand for the axioms.

Most of the new axioms for complement and intersection (Figure 3.9) are self-explanatory, except for the axiom for the complement of a sequential composition, COMP-SEQ. To understand the axiom, first note that the abbreviation $\mathsf{all}$ is a unit for intersection, and an annihilator for parallel composition in the dup-free fragment. *i.e.* $p \cap \mathsf{all} \equiv p$ and $p + \mathsf{all} \equiv \mathsf{all}$. Thus, the $p \cdot \mathsf{all} + q$ essentially says "if $p$ is non-zero, ignore $q$". Looking at the semantics of the complement of a sequential composition:

$$\begin{aligned}
[\![\overline{p \cdot q}]\!]_{-\mathsf{dup}}\ pk &= \overline{\bigcup_{pk' \in [\![p]\!]_{-\mathsf{dup}}\ pk} [\![q]\!]_{-\mathsf{dup}}\ pk'} \\
&= \bigcap_{pk' \in [\![p]\!]_{-\mathsf{dup}}\ pk} [\![\overline{q}]\!]_{-\mathsf{dup}}\ pk'
\end{aligned}$$

So, this is equivalent to "guessing" each $pk'$ in $[\![p]\!]_{-\mathsf{dup}}\ pk$, computing $[\![\overline{q}]\!]_{-\mathsf{dup}}\ pk'$, ignoring the result if we "guessed wrong" (i.e. $pk' \in [\![\overline{p}]\!]_{-\mathsf{dup}}\ pk$), and taking the intersection over all such guesses.

This trick works for the dup-free fragment because we can filter out the "wrong guesses" by testing against $\overline{p}$. However, it's not at all clear how to lift this trick to the history semantics: because terms ignore all but the last packet in the history, we can't create a term that filters out "wrong guesses in the past".

**Theorem 3** ($\mathsf{dup}$-free Soundness of NetKAT($-, \cap$) Axioms)**.** *For all dup-free policies $p$ and $q$, if $p \equiv q$ is provable from the NetKAT($-, \cap$) axioms, then $[\![p]\!]_{-\mathsf{dup}} = [\![q]\!]_{-\mathsf{dup}}$.*

**Theorem 4** (Soundness of non-complement axioms)**.** *For all policies $p$ and $q$, if*

$$p \equiv q$$

*in the equational theory generated by the NetKAT($-, \cap$) axioms minus* COMP-FILTER*,* COMP-MOD*, and* COMP-SEQ*, then*

$$[\![p]\!] = [\![q]\!]$$

**dup-free completeness** The original proof of NetKAT completeness defined a restricted subset of NetKAT, reduced NetKAT, and showed that every NetKAT term was provably equivalent to a reduced NetKAT term. Next they gave a language model for reduced NetKAT that is isomorphic to the standard (history) semantics. Finally, they defined a normal form for reduced NetKAT and related it to regular sets of guarded string, and showed that every reduced NetKAT policy is provably equivalent to a policy in normal form. Completeness then follows as a corollary of the completeness of KAT.

To prove completeness for NetKAT$(-, \cap)$, we can carry out a parallel development for dup-free NetKAT. We then show that we can translate any term in the dup-free fragment of NetKAT$(-, \cap)$ language into a provably equivalent term in reduced dup-free NetKAT. Completeness then follows as an immediate corollary of the completeness of NetKAT itself.[4]

**Lemma 1.** *Every dup-free NetKAT$(-, \cap)$ policy is provably equivalent to a reduced NetKAT$(-, \cap)$ policy.*

**Theorem 5.** *The axioms for NetKAT$(-, \cap)$ shown in Figure 3.9 plus the NetKAT axioms (minus* PA-DUP-FILTER-COMM*) are complete for the dup-free fragment.*

**Conjecture 1.** *There does not exist any sound, finite equational extension of the NetKAT axioms that is complete for NetKAT$(-, \cap)$.*

[4]At first glance, it might seem that we are missing axioms. For example, we have no axiom explaining how to take the complement of $p^*$. It turns out that such an axiom is unnecessary for the dup-free fragment. Because dup-free NetKAT$(-, \cap)$ is essentially finite, any instance of star can be transformed into an equivalent star-free term.

The axioms and the proof are closely tied to the fact that we are working in the dup-free fragment. To extend the development to the full language would require new axioms for complement, including an axiomatization of $\overline{p^*}$.

## 3.5 NetKAT$(-,\cap)$ automata theory

At this point, we have presented a specification language Pathetic, defined what it means for a NetKAT policy to satisfy a specification, and extended NetKAT to NetKAT$(-,\cap)$ in order to translate Pathetic.

From here, the road map is as follows: (1) we review the theory of NetKAT$(-,\cap)$ coalgebras, which form the foundation for NetKAT$(-,\cap)$ automata; (2) we show how to construct (deterministic) NetKAT$(-,\cap)$ automata from terms; (3) we describe a bisimulation checker for NetKAT$(-,\cap)$ automata. We then appeal to the fact that bisimilarity of NetKAT$(-,\cap)$ automata corresponds to language equivalence, and we are done building an equivalence checker.

### 3.5.1 NetKAT$(-,\cap)$ coalgebra

In this section, we briefly review the coalgebraic theory of NetKAT$(-,\cap)$, and in the next section show how to use it to build to build an automata-theoretic equivalence checker. This section is essentially a recapitulation of the development in Foster *et al.* [25], and we include it for the sake of completeness. Readers familiar with that paper can safely skip reading this section.

In this thesis, we take the coalgebraic view of automata, where an automaton is simply a finite-state coalgebra over a state space S, along with an observation map $S \to 2$ indicating accepting states, and a continuation map $S \times \Sigma \to S$ specifying state transitions. For more details on this approach to representing state-based transition systems, see Rutten [94].

Concretely, a NetKAT coalgebra consists of a state space $S$, along with observation and continuation maps

$$\epsilon_{\alpha\beta} : S \to 2 \qquad\qquad \delta_{\alpha\beta} : S \to S$$

for $\alpha, \beta \in \mathsf{At}$. A deterministic NetKAT automaton is a finite-state NetKAT coalgebra with a start state in $S$. The automaton operates on the (reduced) strings of the language-model, $U = \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$. If the automaton is in state $s$, and sees string $\alpha\pi_\beta$, then it accepts iff $\epsilon_{\alpha\beta}(s)$. If the automaton is in state $s$ and sees string $\alpha{\cdot}\pi_\beta{\cdot}\mathsf{dup}{\cdot}x$, then it transitions to $\delta_{\alpha\beta}(s)$ with residual string $\beta{\cdot}x$.

A reduced string is accepted by the automaton iff it accepts the string from the distinguished start state.

As Foster *et al.* [25] showed, NetKAT's automata theory has an analog to Kleene's theorem for regular expressions:

**Theorem 6** (Kleene's Theorem for NetKAT)**.** *A set of string is $G(p)$ for some NetKAT policy $p$ iff it is the set of strings accepted by some finite NetKAT automaton.*

For the full details of this theorem and its proof, read [25].

### 3.5.2 NetKAT$(-, \cap)$ automata

Because NetKAT automata are closed under intersection and complement, it immediately follows that NetKAT$(-, \cap)$ automata are in fact NetKAT automata. However, this does not mean that we can just reuse the NetKAT automata construction. Foster *et al.* also showed that the size of the minimal deterministic automata $M_p$ for each NetKAT term $p$ is $O(2^l)$, where $l$ is the number of occurences of dup in $p$. It is well-known [101] that enriching regular expressions with complement and intersection causes at least an exponential increase in the size of the minimal automata [31], to doubly-exponential. This lower-bound also applies to our language. Therefore, their construction cannot apply to NetKAT$(-, \cap)$.

Despite this large theoretical lower-bound, we can still hope to build a verifier that is performant in practice. By carefully only exploring the reachable state space, and generating the automata on demand, we can avoid unnecessary work. Moreover, we expect that real-world specifications will not exhibit the pathological structure that causes such an explosion.

**Brzozowski Derivative** The Brzozowski derivative is a standard way of building coalgebras from regular expressions. There is both a *semantic* Brzozowski derivative defined upon subsets of $U$, giving a NetKAT$(-, \cap)$ coalgebra over the state space $2^U$, and a *syntactic* Brzozowski derivative defined over NetKAT$(-, \cap)$ expressions, resulting in a NetKAT$(-, \cap)$ coalgebra over a state space of expressions (shown in Figure 3.10).

## 3.6 Automata Representation

In this section, we show how to build a Pathetic verifier based upon the automata theory in the previous section. We start by outlining the Brzozowski-based construction and representation used by Foster *et al.* for NetKAT verifier, and explain why this representation does not work for NetKAT$(-, \cap)$. We then present our own construction, also based on the Brzozowski derivative.

$$D'_{\alpha\beta}(\pi) = 0 \qquad D'_{\alpha\beta}(b) = 0 \qquad D'_{\alpha\beta}(\mathsf{dup}) = [\alpha = \beta]$$
$$D'_{\alpha\beta}(\overline{p}) = \overline{D'_{\alpha\beta}(p)}$$
$$D'_{\alpha\beta}(p + q) = D'_{\alpha\beta}(p) + D'_{\alpha\beta}(q)$$
$$D'_{\alpha\beta}(p \cdot q) = D'_{\alpha\beta}(p) \cdot q + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D'_{\gamma\beta}(q)$$
$$D'_{\alpha\beta}(p \cap q) = D'_{\alpha\beta}(p) \cap D'_{\alpha\beta}(q)$$
$$D'_{\alpha\beta}(p^*) = D'_{\alpha\beta}(p) \cdot p^* + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D'_{\gamma\beta}(p^*)$$

$$D_{\alpha\beta}(p) = \beta \cdot D'_{\alpha\beta}(p)$$

$$E_{\alpha\beta}(\pi) = [\pi = \pi_\beta] \qquad E_{\alpha\beta}(b) = [\alpha = \beta \leq b] \qquad E_{\alpha\beta}(\mathsf{dup}) = 0$$
$$E_{\alpha\beta}(\overline{p}) = \overline{E_{\alpha\beta}(p)} \qquad E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) + E_{\alpha\beta}(q)$$
$$E_{\alpha\beta}(p \cap q) = E_{\alpha\beta}(p) \cdot E_{\alpha\beta}(q)$$
$$E_{\alpha\beta}(p \cdot q) = \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(q)$$
$$E_{\alpha\beta}(p^*) = [\alpha = \beta] + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(p^*).$$

Figure 3.10: NetKAT$(-, \cap)$ syntactic Brzozowski derivative.

**Foster *et al.*'s automata representation** Looking carefully at the definition of the Brzozowski derivatives, it becomes clear that each of the definitions corresponds to operations on $\mathsf{At} \times \mathsf{At}$ matrices, where $E_{\alpha\beta}(p)$ is the $\alpha, \beta$ entry of the matrix $E(p)$. For example, the definition of $E_{\alpha\beta}(p \cdot q)$ is exactly the definition of matrix multiplication. Moreover, as Foster *et al.* note, for NetKAT, these matrices are highly sparse, and "close" to a diagonal matrix. Consider, for example, $E_{\alpha\beta}(f = n)$. This corresponds to a diagonal binary matrix where entry $\alpha\alpha$ is 1 iff the $f$ value of $\alpha$ is $n$. By using a sparse matrix representation based upon these vectors (and similar vectors corresponding to modifications), they obtain a compact representation that requires much less space than the $|\mathsf{At}|^2$ entries of the full matrix. Note that $\mid \mathsf{At} \mid = \Pi_f \mid v_f \mid$, where $v_f$ is the set of values for header field $f$, *i.e.* exponential in the number of fields.

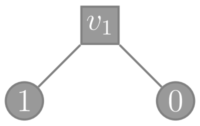
Second, they observe that the size of the state space $S$ can be bounded based upon the structure of the term. They identify a set of subterms, dubbed *spines*, whose size is linear in $l$, the number of occurrences of dup in the original expression, and show that sets of spines suffice as a representation of $S$. Note that the linear bound on the set of spines this leads directly to an exponential bound on the size of $S$: $2^l$.

Unfortunately, neither of these tricks will work for NetKAT$(-, \cap)$. The complement operation of NetKAT$(-, \cap)$ takes a sparse matrix to its complement, which is a dense matrix. This eliminates the benefits of their sparse matrix representation. Similarly, their small set of *spines* no longer suffices to represent the state space. This is immediately obvious because using spines leads to an exponential bound on the size of the state space, but minimal NetKAT$(-, \cap)$ automata may be super-exponentially sized.

### 3.6.1 NetKAT$(-, \cap)$ automata representation

We use a different, novel representation for NetKAT$(-, \cap)$ derivatives that retains the benefits of the sparse representation of Foster *et al.* when possible. We combine an efficient representation of sparse matrices, based on functional decision diagrams, with an algebraic representation of matrix operations. This enables the verifier to exploit sparseness for positive terms (terms without complement), and allows the recognition of simplifying algebraic identities for the full language (*e.g.* $\overline{\overline{p}} \equiv p$, $\overline{p} + \overline{q} \equiv \overline{p \cap q}$).

**NetKAT$(-, \cap)$ E derivative representation** We use functional decision diagrams (**FDD**s), a generalization of binary decision diagrams, to represent $E$ matrices. A binary decision diagram (BDD) represents a boolean valued function on a set of boolean valued variables $(H \to 2) \to 2$, where $H$ is the set of variables. Concretely, a BDD is a directed, acyclic graph where leaf nodes are labeled with boolean values, and interior nodes are labeled with variables and have two outgoing edges, representing the two possible values of the variable. For example,



is a BDD that represents the boolean function $v_1 = 1$ (we draw the true edge on the left, and the false edge on the right).

Similarly, an $(H, V, B)$ **FDD**, is a directed, acyclic graph that represents a function of type $(H \to V) \to B$, where the variables in $H$ are $V$ valued. We replace the boolean variables with boolean tests of equality on mutli-valued variables. Thus, an $(H, V, B)$ has as internal nodes pairs in $(H \times V)$, representing boolean tests on the input, with children nodes representing the path for inputs that satisfy or fail the test, and its leaf nodes are elements of $B$. Just as in BDDs, an **FDD** that represents a highly uniform function may have many identical subgraphs. Reduced **FDD**s use structure sharing to eliminate common sub-**FDD**s, and can provide very compact representations for uniform functions in which large sets of inputs have the same output. In this rest of this chapter, **FDD** means a reduced **FDD**.

If $E$ is an $(H, V, B)$ **FDD**, and $h$ is an element of $(H \to V)$, then we write $[\![E]\!](h)$ to mean the element of $B$ that is output by the function represented by $E$ on the input $h$.

NetKAT **FDD**s (herein just **FDD**s) are $(F, \mathbb{N}, \mathcal{P}(F \rightharpoonup \mathbb{N}))$ **FDD**s. That is, they directly represent functions that take packets as input (represented as finite maps from headers $f$ to

$$\begin{array}{lll} \textbf{FDD} & E, E' ::= \{f \rightharpoonup n\} \\ & \quad\mid\ (f = n)?(E) : (E') \end{array} \qquad \begin{aligned} [\![E]\!] &\in \mathsf{At} \to \mathsf{At} \to 2 \\ [\![\{m_i\}]\!](\alpha)(\beta) &\triangleq \textstyle\sum_{m_i} [(\alpha \circ m_i) = \beta] \\ [\![(f = n)?(E) : (E')]\!] &\triangleq \begin{cases} [\![E]\!](\alpha)(\beta) & \text{if } \alpha[f] = n \\ [\![E']\!](\alpha)(\beta) & \text{o.w.} \end{cases} \end{aligned}$$

Figure 3.11: NetKAT **FDD** syntax and semantics

header values $n$), and output sets of (partial) packets (finite maps that may not have values for all headers). We interpret a NetKAT **FDD** $E$ as a function of type $\alpha \to \beta \to 2$ by $E(\alpha)(\beta) = \exists \beta' \in [\![F]\!](\alpha) \wedge \alpha \circ \beta' = \beta$. For example, the **FDD** $\{[]\}$ corresponds to the E derivative of the diagonal $\mathbf{1}$: $\lambda \alpha, \beta.\alpha = \beta$.

Similarly, the **FDD** representation for $E(f \leftarrow n)$ is just: $\{[f = n]\}$.

**NetKAT FDD operations**

$$\begin{aligned} E_{f \leftarrow n} &\triangleq \{[f \leftarrow n]\} \\ E_{f = n} &\triangleq (f = n)?(\{[]\}) : (\{\}) \\ E_{p+q} &\triangleq E_p \cup E_q \\ E_{p \cap q} &\triangleq E_p \cap E_q \\ E_{p \cdot q} &\triangleq E_p \cdot E_q \\ E_{p^*} &\triangleq \mu E'.E_1 + E_p \cdot E' \end{aligned}$$

The basic operations on **FDD**s are union, intersection, sequential composition, iteration.

The **FDD** representation is an alternative to the sparse matrix representation of Foster *et al.*, and suffers from the same problem when applied to NetKAT$(-, \cap)$. Notice that the **FDD** representation of the term $\mathbf{1}$ is very compact. This representation is optimized for sparse matrices that are close to a diagonal matrix. Consider, by contrast, the representation of the E derivative of term $\overline{1}$. It is equivalent to $\lambda\alpha, \beta.\alpha \neq \beta$. But this function has a very large representation as an **FDD**: it is the full tree where every full path through the tree represents a complete test, and the leaf node on the path corresponding to $\alpha$ is the set $\{\beta \mid \beta \neq \alpha\}$. Even worse, a naive implementation may end up constructing a very large **FDD** as an intermediate state when the final **FDD** is in fact very small. For example, naively constructing the **FDD** for $\overline{\overline{1}} \equiv 1$ would result in an **FDD** equivalent to $\{[]\}$, but construct the **FDD** for $\overline{1}$ as an intermediate.

To avoid the blow-up that comes from complementing **FDD**s, but maintain the benefit of their compactness when possible, we combine **FDD**s with the symbolic representation shown in Figure 3.6.1. Complement-free policies are represented as **FDD**s (smart constructors enforce that the union, intersection, iteration, and sequential composition of **FDD**s are **FDD**s), and policies containing complement are represented as formal terms over **FDD**s. This enables compact representation of positive NetKAT terms while enabling the recognition of simplifying algebraic identities. In this representation, the term $\overline{\overline{1}}$ would be represented exactly as the **FDD** $\{[]\}$, because the symbolic identity $\overline{\overline{p}} \equiv p$ would be recognized and reduced, without computing the **FDD** for $\overline{p}$.

We write $E_p$ to refer to the **FDD** representation of the E derivative of the (complement-free) term $p$.

**NetKAT$(-,\cap)$ derivative representation**

| | | |
|---|---|---|
| E $e, e' ::= E$ | | *Positive* ***FDD*** |
| $\mid$ | $\overline{e}$ | *Complement* |
| $\mid$ | $e + e'$ | *Union* |
| $\mid$ | $e \cap e'$ | *Intersection* |
| $\mid$ | $e \cdot e'$ | *Sequential composition* |
| $\mid$ | $e^*$ | *Kleene star* |

**NetKAT$(-,\cap)$ derivative representation semantics**

$$\begin{aligned}
[\![e]\!] &\in \mathsf{At} \to \mathsf{At} \to 2 \\
[\![E]\!](\alpha)(\beta) &\triangleq [\![E]\!](\alpha)(\beta) \\
[\![\overline{e}]\!](\alpha)(\beta) &\triangleq \overline{[\![e]\!](\alpha)(\beta)} \\
[\![e + e']\!](\alpha)(\beta) &\triangleq [\![e]\!](\alpha)(\beta) + [\![e']\!](\alpha)(\beta) \\
[\![e \cap e']\!](\alpha)(\beta) &\triangleq [\![e]\!](\alpha)(\beta) \cdot [\![e']\!](\alpha)(\beta) \\
[\![e \cdot e']\!](\alpha)(\beta) &\triangleq \textstyle\sum_\gamma [\![e]\!](\alpha)(\gamma) \cdot [\![e']\!](\gamma)(\beta) \\
[\![e^*]\!](\alpha)(\beta) &\triangleq [\alpha = \beta] + \textstyle\sum_\gamma [\![e]\!](\alpha)(\gamma) \cdot [\![e^*]\!](\alpha)(\beta)
\end{aligned}$$

To represent D derivatives, we follow the insight of Foster *et al.* that the D derivatives correspond to basic matrix operations, and use a matrix-like representation. We decompose each D derivative into a sum of (possibly overlapping) single-valued matrices, and represent each matrix as a pair of an E matrix (representing the domain), and a policy (representing the value of the matrix on its domain) (shown in Figure 3.12)[5]. The relationship between this representation and the syntactic Brzozowski derivative is expressed in Lemma 2:

[5]This representation is based upon one proposed by Konstantinos Mamouras in private correspondence.

$$E(p) = E_p \qquad E(b) = E_b \qquad E(\mathsf{dup}) = E_0$$

$$E(\overline{e}) = \overline{E(e)} \qquad E(e_1 + e_2) = E(e_1) + E(e_2)$$

$$E(e_1 \cap e_2) = E(e_1) \cap E(e_2)$$

$$E(e_1 \cdot e_2) = E(e_1) \cdot E(e_2)$$

$$E(e^*) = E(e)^*$$

$$D(p) = \{\} \qquad D(b) = \{\} \qquad D(\mathsf{dup}) = \{(E(1), 1)\}$$

$$D(e_1 + e_2) = D(e_1) \cup D(e_2)$$

$$D(e_1 \cdot e_2) = D(e_1) \cdot e_2 \cup E(e_1) \cdot D(e_2)$$

$$D(e_1 \cap e_2) = \{d_1 \cap d_2 \mid d_1 \in D(e_1), d_2 \in D(e_2)\}$$

$$D(e^*) = E(e^*) \cdot D(e) \cdot e^*$$

$$D(\overline{p}) = \bigcup_{\alpha,\beta} \left\{ (E(\alpha \cdot p_\beta), \bigcap_{(e',d') \in D(p) \wedge e'(\alpha)(\beta)} \overline{d'}) \right\}$$

where

$$D \cdot p \triangleq \{(e, d \cdot p) \mid (e, d) \in D\}$$

$$E \cdot D \triangleq \{(E \cdot e, d) \mid (e, d) \in D\}$$

$$(e, d) \cap (e', d') \triangleq (e \cap e', d \cap d')$$

Figure 3.12: NetKAT$(-, \cap)$ derivative representation.

**Lemma 2.**

$$D_{\alpha,\beta}(p) \equiv \sum_{(e,d) \in D(p)} [e(\alpha)(\beta)] \cdot \beta \cdot d$$

### 3.6.2 NetKAT$(-, \cap)$ equivalence checking

With our automata representation, we can now build an equivalence checker that checks NetKAT$(-, \cap)$ terms for bisimulation. Given two NetKAT$(-, \cap)$ terms $p$ and $q$, we first compare their $E$ matrices for (semantic) equality. If they are not equal, we return false. Otherwise, we calculate the derivatives of both terms and recursively check them for equivalence. Once we've reached every reachable pair of derivatives, the algorithm halts. The proof of termination depends upon finiteness of an extension of the original NetKAT spines to NetKAT$(-, \cap)$, and the closure of these spines under the derivative, which is shown in Appendix A.1.

CHAPTER 4

# CORRECTLY IMPLEMENTING NETWORK PROGRAMS

*"Trust, but verify."*

—Ronald Reagan

In the previous chapter we showed how to verify that a network program correctly implements a specification. In this chapter, we show how to build a system that correctly implements network programs, guaranteeing that the properties of the input program also are preserved by the resulting network itself.

Concretely, this chapter describes the design and implementation of a machine-verified compiler and OpenFlow controller for the NetCore language, a predecessor to NetKAT. Starting from the foundations, we develop a detailed operational model for the OpenFlow SDN platform, and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language (NetCore). We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

The content of this chapter is based upon a joint PLDI paper [35] published with Arjun Guha and Nate Foster in 2013.

## 4.1 Introduction

Bugs in compilers and runtimes are especially pernicious sources of errors. Difficult to track down, their effect can be widespread, potentially affecting every program they touch.

Indeed, a lack of trust in the reliability of complex optimizing compilers and language runtime systems is one potential stumbling block in the adoption of high-level programming languages in the systems domain.

Moreover, recent work has shown that such mistrust would not be entirely misplaced: NICE [14] found a number of runtime bugs in popular SDN controller platforms, and in prior work [35] we found correctness bugs in every network programming language compiler examined.

Fortunately, there is a solution: formal specification and verification of compilers and runtimes. In one study of optimizing C compilers, every single compiler, save one, was found to have bugs that caused wrong-code generation [114]. The one exception was the formally verified compiler from the CompCert project [59][1].

In this chapter, we show how to formally model network programming languages and software-defined networks in the Coq theorem prover. We then show how to use these formal models to build and verify a compiler and network controller that provably preserves the correctness of its input program.

**Architecture** Our system is organized as a verified software stack (Figure 4.1) that translates through the following levels of abstraction:

- **NetCore.** The highest level of abstraction is the NetCore language, proposed in prior work by Monsanto *et al.* [77]. NetCore is a predecessor to the NetKAT language used earlier in this thesis. Unlike NetKAT, NetCore does not directly model the topology of

[1]Initially, the authors of that study found a bug in an unverified component of CompCert. In response, CompCert extended the verified to include that component, and the authors were not able to find any bugs in the newly verified system.

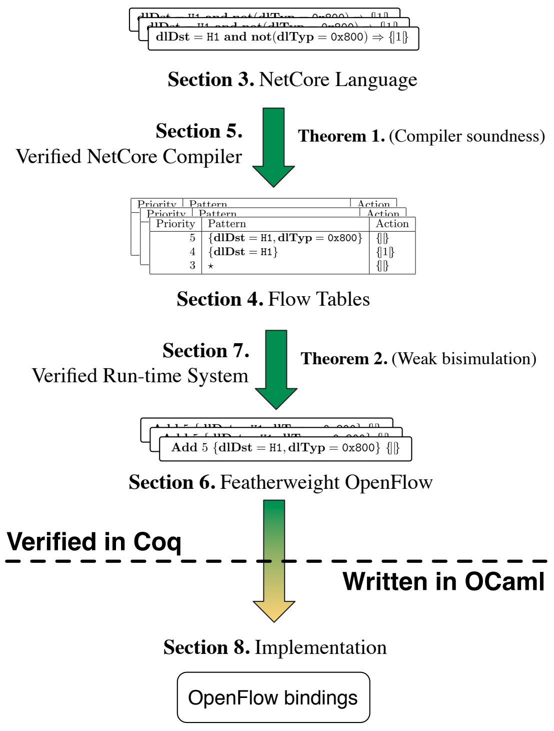

dlDst = H1 and not(dlTyp = 0x800) ⇒ {|1|}
Section 3. NetCore Language
Section 5. Verified NetCore Compiler
Theorem 1. (Compiler soundness)
Priority | Pattern | Action
5 | {dlDst = H1, dlTyp = 0x800} | {||}
4 | {dlDst = H1} | {|1|}
3 | ⋆ | {||}
Section 4. Flow Tables
Section 7. Verified Run-time System
Theorem 2. (Weak bisimulation)
Add 5 {dlDst = H1, dlTyp = 0x800} {||}
Section 6. Featherweight OpenFlow
Verified in Coq
Written in OCaml
Section 8. Implementation
OpenFlow bindings

Figure 4.1: System architecture.

the network, and so is essentially equivalent to dup-free NetKAT. NetCore also lacks the iteration operator of NetKAT, but iteration adds no expressivity to the dup-free fragment, so this is not a significant loss.

- **Flow tables.** The intermediate level of abstraction is *flow tables*, a representation that sits between NetCore programs and switch-level configurations. There are two main differences between NetCore programs and flow tables. First, NetCore programs describe the forwarding behavior of a whole network, while flow tables describe the behavior of a single switch. Second, flow tables process packets using a linear scan through a list of prioritized rules. Hence, to translate operators such as union and negation, the NetCore compiler must generate a sequence of rules that encodes the same semantics. However, because flow table matching uses a lower-level packet representation (as nested frames of Ethernet, IP, TCP, etc. packets), flow tables must satisfy a well-formedness condition to rule out invalid patterns that are inconsistent with this representation.

- **Featherweight OpenFlow.** The lowest level of abstraction is *Featherweight OpenFlow*, a new foundational model we have designed that captures the essential features of SDNs. Featherweight OpenFlow models switches, the controller, the network topology, as well as their internal transitions and interactions in a small-step operational semantics. This semantics is non-deterministic, modeling the asynchrony inherent in networks. To implement a flow table in a Featherweight OpenFlow network, the controller instructs switches to install or uninstall rules as appropriate while dealing with two important issues: First, switches process instructions concurrently with packets flowing through the network, so it must ensure that at all times the rules installed on switches are consistent with the flow table. Second, switches are allowed to buffer instructions and apply them in any order, so it must ensure that the behavior is correct no matter how instructions are reordered through careful use of synchronization primitives.

## 4.2 Overview

To motivate the need for verified SDN controllers, consider a simplified version of the network from our running example, shown in Figure 4.2. This network has only one switch I, one firewall FW, a load balancer LB, a web server WEB, an internal network INTRANET, and an external network WORLD.

Now imagine we want to build an SDN controller that implements the following network policy: block inbound SSH traffic, route inbound HTTP requests through the load-balancer and then to WEB, and allow all other traffic to INTRANET once it has passed through the

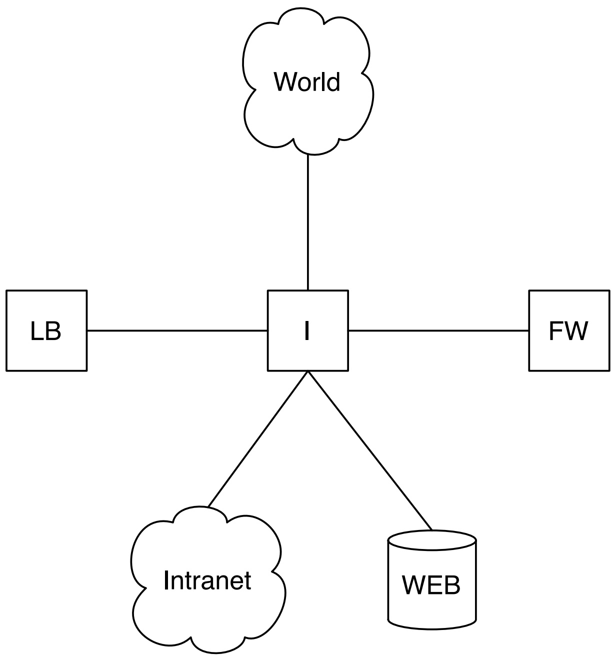


Figure 4.2: Example network topology.

firewall. It is straightforward to formalize this policy as a packet-processing function that maps input packets to (possibly several) output packets: the function drops SSH packets a forwards HTTP packets both to their destination and to the middlebox, and forwards all other packets to the firewall and then their destination.

To implement this function in an SDN, however, we would need to specify several additional low-level details, since switches cannot implement general packet-processing functions directly. First, the controller would need to encode the function as a *flow table*—a set of prioritized forwarding rules. Second, it would need to send the switch a series of control messages to add individual entries from the flow table, incrementally building up the complete table.

More concretely, the controller could first send a message instructing the switch to add a flow table entry that blocks SSH traffic:

$$\textbf{Add } 10 \ \{\textbf{tpDst} = \mathsf{22}\} \ \{\!|\,|\!\}$$

Here 10 is a priority number, $\{\textbf{tpDst} = \mathsf{22}\}$ is a pattern that matches SSH traffic (TCP port 22), and $\{\!|\,|\!\}$ is an empty multiset of ports, which drops packets, as intended. Next, the controller could add an entry to process inbound HTTP requests:

$$\textbf{Add } 9 \ \{\textbf{inPort} = \mathsf{WORLD}, \textbf{dlDst} = \mathit{WEB}, \textbf{tpDst} = \mathsf{80}\} \ \{\!|\mathsf{LB}|\!\}$$

Note that this rule only applies to HTTP (TCP port 80) packets traffic that has not been sent to $\mathsf{LB}$ yet.

The controller can then add another entry to process load-balanced HTTP requests:

$$\textbf{Add } 8 \ \{\textbf{inPort} = \mathsf{LB}, \textbf{dlDst} = \mathit{WEB}, \textbf{tpDst} = \mathsf{80}\} \ \{\!|\mathsf{WEB}|\!\}$$

Finally, the controller could similarly install a pair of entries to forward other packets to their destination, after going through the firewall

$$\textbf{Add } 2 \ \{\textbf{inPort} = \mathsf{WORLD}\} \ \{\!|\mathsf{FW}|\!\}$$
$$\textbf{Add } 1 \ \{\textbf{inPort} = \mathit{FW}\} \ \{\!|\mathsf{INTRANET}|\!\}$$

Note that this rule does not apply to SSH and HTTP traffic, since those packets are handled by the higher-priority rules.

After these control messages have been sent, it would be natural to expect that the network correctly implements the packet-processing function described above. But the situation is actually more complicated: switches have substantial latitude in how they process messages from the controller, and packets may arrive at any time during processing. Establishing that the network correctly implements this function—in particular, that it blocks SSH traffic and load balances HTTP traffic—requires additional reasoning.

**Controller-switch consistency.** Switches process packets and control messages concurrently. In our example, the switch may receive an HTTP request before the flow table entry that handles HTTP packets arrives. In this case, the switch will send the packet to the controller for further processing. Since the controller is a general-purpose machine, it can implement the packet-processing function directly, apply it to the incoming packet, and send the results back to the switch. However, this means that SDN controllers typically have two *different* implementations of the function: one residing at the controller and another on the switches. A key property we verify is that these two implementations are consistent.

**Message reordering.** SDN switches may process control messages in any order, and many switches do, to maximize performance. But unrestricted reordering can cause implementations to violate their intended specifications. For example, if the rule to drop SSH traffic is installed after the final, low-priority rule that forwards all traffic, then SSH traffic will temporarily be forwarded by the low-priority rule, breaking the intended security policy. To ensure that such reorderings do not occur, a controller must carefully insert *barrier messages*, which force the switch to process all outstanding messages. A key property we verify is that controllers use barriers correctly (several unverified controllers ignore this issue).

**Natural patterns.** Another complication is that the patterns presented earlier in this section, such as $\{\textbf{tpDst} = 22\}$, are actually invalid. To match SSH traffic, it is not enough to simply state that the destination port must be 22. The pattern must also specify that the Ethernet frame type must be IP, and the transport protocol must be TCP. Without these additional constraints, switches will interpret the pattern as a wildcard that matches all packets. Several earlier controller platforms did not properly account for this behavior, and had bugs as a result. We develop a semantics for patterns and identify a class of *natural*

$$
\begin{array}{lll}
\text{Packet} & pk & ::= \textbf{Eth}\ \mathit{dlSrc}\ \mathit{dlDst}\ \mathit{dlTyp}\ \mathit{nwPk} \\
\text{Network layer} & \mathit{nwPk} & ::= \textbf{IP}\ \mathit{nwSrc}\ \mathit{nwDst}\ \mathit{nwProto}\ \mathit{tpPk} \\
 & & \quad\ \mid \textbf{Unknown}\ \mathit{payload} \\
\text{Transport layer} & \mathit{tpPk} & ::= \textbf{TCP}\ \mathit{tpSrc}\ \mathit{tpDst}\ \mathit{payload} \\
 & & \quad\ \mid \textbf{Unknown}\ \mathit{payload}
\end{array}
$$

Figure 4.3: Logical packet structure.

*patterns* that are closed under the algebraic operations used by our compiler and flow table optimizer.

**Roadmap.** The rest of this chapter develops techniques for establishing that a given packet-processing function is implemented correctly by an OpenFlow network. More specifically, we tackle the problem of verifying high-level programming abstractions, using NetCore [77] as a concrete instance of a high-level network language. The next section presents NetCore in detail. The following sections describe general and reusable techniques for establishing the correctness of SDN controllers, including NetCore.

## 4.3 NetCore

This section presents the highest layer of our verified stack: the NetCore language. A NetCore program specifies how the switches process packets at each hop through the network. More formally, a program denotes a total function from port-packet pairs to multisets of port-packet pairs. The syntax and semantics of a core NetCore fragment are shown in Fig. 4.4. To save space, we have elided several header fields and operators not used in this chapter.

We can build a NetCore program that implements the example from the previous section by composing several smaller NetCore program fragments. The first fragment forwards traffic

$$
\begin{array}{llcll}
\text{Switch ID} & sw & \in & \mathbb{N} & \\
\text{Port ID} & pt & \in & \mathbb{N} & \\
\text{Headers} & h & ::= & \mathbf{dlSrc} \mid \mathbf{dlDst} & \textit{MAC address} \\
 & & \mid & \mathbf{dlTyp} & \textit{Ethernet frame type} \\
 & & \mid & \mathbf{nwSrc} \mid \mathbf{nwDst} & \textit{IP address} \\
 & & \mid & \mathbf{nwProto} & \textit{IP protocol code} \\
 & & \mid & \mathbf{tpSrc} \mid \mathbf{tpDst} & \textit{transport port} \\
\text{Predicate} & pr & ::= & \star & \textit{wildcard} \\
 & & \mid & h = n & \textit{match header} \\
 & & \mid & \mathbf{on}\ \ sw & \textit{match switch} \\
 & & \mid & \mathbf{at} & \textit{match inport} \\
 & & \mid & \mathbf{not}\ pr & \textit{predicate negation} \\
 & & \mid & pr_1\ \mathbf{and}\ pr_2 & \textit{predicate conjunction} \\
\text{Program} & \phi & ::= & pr \Rightarrow \{\!|pt_1 \cdots pt_n|\!\} & \textit{basic program} \\
 & & \mid & \phi_1 \uplus \phi_2 & \textit{program union} \\
 & & \mid & \mathbf{restrict}\ \phi\ \mathbf{by}\ pr & \textit{program restriction}
\end{array}
$$

$\boxed{[\![pr]\!]\ sw\ pt\ pk}$

$$
\begin{array}{l}
[\![\star]\!]\ sw\ pt\ pk = \mathbf{true} \\
[\![\mathbf{dlSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \textit{dlSrc}\ \_\ \_\ \_) = \textit{dlSrc}{=}n \\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{IP}\ \textit{nwSrc}\ \_\ \_\ \_)) = \textit{nwSrc}{=}n \\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{Unknown}\ \_)) = \mathbf{false} \\
\qquad\qquad \cdots \\
[\![\mathbf{on}\ \ sw']\!]\ sw\ pt\ pk = sw{=}sw' \\
[\![\mathbf{at}\ \ ']\!]\ sw\ pt\ pk = pt{=}pt' \\
[\![\mathbf{not}\ pr]\!]\ sw\ pt\ pk = \neg([\![pr]\!]\ sw\ pt\ pk) \\
[\![pr_1\ \mathbf{and}\ pr_2]\!]\ sw\ pt\ pk = [\![pr_1]\!]\ sw\ pt\ pk \wedge [\![pr_2]\!]\ sw\ pt\ pk
\end{array}
$$

$\boxed{[\![\phi]\!]\ sw\ pt\ pk = \{\!|(pt_1, pk_1) \cdots (pt_n, pk_n)|\!\}}$

$$
\begin{array}{l}
[\![pr \Rightarrow \{\!|pt_1 \cdots pt_n|\!\}]\!]\ sw\ pt\ pk = \\
\quad \mathbf{if}\ [\![pr]\!]\ sw\ pt\ pk\ \mathbf{then}\ \{\!|(pt_1, pk) \cdots (pt_n, pk)|\!\}\ \mathbf{else}\ \{\!||\!\} \\
[\![\phi_1 \uplus \phi_2]\!]\ sw\ pt\ pk = \\
\quad [\![\phi_1]\!]\ sw\ pt\ pk \uplus [\![\phi_2]\!]\ sw\ pt\ pk \\
[\![\mathbf{restrict}\ \phi\ \mathbf{by}\ pr]\!]\ sw\ pt\ pk = \\
\quad \{\!|(pt', pk') \mid (pt', pk') \in [\![pg]\!]\ sw\ pt\ pk \wedge [\![pr]\!]\ sw\ pt\ pk|\!\}
\end{array}
$$

Figure 4.4: NetCore syntax and semantics (extracts).

to $\mathsf{WEB}$:

$$\phi_1 \triangleq \textbf{at } \mathsf{LB} \textbf{ and } \textbf{dlDst}{=}\mathsf{WEB} \Rightarrow \{\!|\mathsf{WEB}|\!\}$$

This basic program consists of a predicate $pr$ and a multiset of actions $\{\!|pt_1 \cdots pt_n|\!\}$. The predicate denotes a set of port-packet pairs, and the actions denote the ports (if any) where those packets should be forwarded on the next hop. In this instance, the predicate denotes the set of all packets whose Ethernet destination (**dlDst**) address has the specified value, and whose inport is $\mathsf{LB}$, and the actions denote a transformation that forwards matching packets to port 1. Note that we represent packets as nested sequences of frames (Ethernet, IP, TCP, etc.) as shown in Fig. 4.3. NetCore provides predicates for matching on well-known header fields as well as logical operators such as **and** and **or**, unlike hardware switches, which only provide prioritized sets of rules.

The next fragment is similar to $\phi_1$, but forwards traffic to $\mathsf{LB}$ instead of $\mathsf{WEB}$:

$$\phi_2 \triangleq \textbf{at } \mathsf{WORLD} \textbf{ and } \textbf{dlDst}{=}\mathit{WEB} \Rightarrow \{\!|\mathsf{LB}|\!\}$$

Using the union operator, we can combine these programs into a single program that implements forwarding for HTTP traffic:

$$\phi_{\mathsf{WEB}} \triangleq \phi_1 \uplus \phi_2$$

Semantically, the $\uplus$ operator produces the (multiset) union of the results produced by each sub-program. Using the restriction operator **restrict by** , we can limit this forwarding policy to web traffic:

$$\textbf{restrict } \phi_{\mathsf{WEB}} \textbf{ by } \textbf{tpDst}{=}22$$

Similarly, we can define the forwarding policy for traffic through the firewall:

$$\begin{aligned}
\phi'_1 &\triangleq \textbf{at } \mathsf{WORLD} \textbf{ and } \textbf{ not } \textbf{dlDst}{=}\mathit{WEB} \Rightarrow \{\!|\mathsf{FW}|\!\} \\
\phi'_2 &\triangleq \textbf{at } \mathsf{FW} \textbf{ and } \textbf{ not } \textbf{dlDst}{=}\mathit{WEB} \Rightarrow \{\!|\mathsf{INTRANET}|\!\} \\
\phi_{\mathsf{FW}} &\triangleq \phi'_1 \uplus \phi'_2
\end{aligned}$$

Finally, we can add the security policy using the **restrict by** operator, which restricts a program by a predicate:

$$\textbf{restrict } (\phi_{\mathsf{WEB}} \uplus \ \phi_{\mathsf{FW}}) \textbf{ by } (\textbf{not } \textbf{tpDst}{=}22)$$

This program is similar the previous one, but drops SSH traffic.

The advantages of using a declarative language such as NetCore should be clear: it provides abstractions that make it easy to establish network-wide properties through compositional reasoning. For example, simply by inspecting the final program and using the denotational semantics (Fig. 4.4), we can easily verify that the network blocks SSH traffic, forwards HTTP traffic to the middlebox, and other forwards traffic to INTRANET. In particular, even though a controller, switches, flow tables, forwarding rules, are all involved in implementing this program, we do not have to reason about them! This is in contrast to lower-level controller platforms, which require programmers to explicitly construct switch-level forwarding rules, issue messages to install those rules on switches, and reason about the asynchronous interactions between switches and controller. Of course, the complexity of the underlying system is not eliminated, but relocated from the programmer to the language implementers. This is an efficient tradeoff: functionality common to many programs can be implemented just once, proved correct, and reused broadly.

$$\begin{array}{lll} \text{Wildcard} & w & ::= n \mid \star \\ \text{Pattern} & pat & ::= \{\mathbf{dlSrc} = w, \mathbf{dlDst} = w, \mathbf{dlTyp} = w, \\ & & \quad \mathbf{nwSrc} = w, \mathbf{nwDst} = w, \mathbf{nwProto} = w, \\ & & \quad \mathbf{tpSrc} = w, \mathbf{tpDst} = w\} \\ \text{Flow table} & FT & \in \{\!|n \times pat \times \{\!|pt|\!\}|\!\} \end{array}$$

$\boxed{[\![FT]\!]\ pt\ pk \ \rightsquigarrow\ \{\!|pt_1 \cdots pt_n|\!\} \times \{\!|pk_1 \cdots pk_m|\!\}}$

$$\frac{\begin{array}{l} \exists (n, pat, \{\!|pt_1 \cdots pt_n|\!\}) \in FT. \\ \quad pk\#pat = \mathbf{true} \\ \forall (n', pat', pts') \in FT.\ n' > n \Rightarrow \\ \quad pk\#pat' = \mathbf{false} \end{array}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!|(pt_1) \cdots (pt_n)|\!\}, \{\!||\!\})} \quad (\textsc{Matched})$$

$$\frac{\forall (n, pat, pts) \in FT \qquad pk\#pat = \mathbf{false}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!||\!\}, \{\!|(pt, pk)|\!\})} \quad (\textsc{Unmatched})$$

$\boxed{pk\#pat}$

$$\begin{array}{l} (\mathbf{Eth}\ dlSrc\ dlDst\ dlTyp\ nwPk)\#pat = \\ \quad dlSrc \sqsubseteq pat.\mathbf{dlSrc}\ \wedge dlDst \sqsubseteq pat.\mathbf{dlDst}\ \wedge \\ \quad dlTyp \sqsubseteq pat.\mathbf{dlTyp}\ \wedge \\ \quad (pat.\mathbf{dlTyp} = \texttt{0x800} \Rightarrow nwPk\#_{nw}pat) \end{array}$$

$\boxed{nwPk\#_{nw}pat}$

$$\begin{array}{l} (\mathbf{IP}\ nwSrc\ nwDst\ nwProto\ tpPk)\#_{nw}pat = \\ \quad nwSrc \sqsubseteq pat.\mathbf{nwSrc} \wedge nwDst \sqsubseteq pat.\mathbf{nwDst}\ \wedge \\ \quad nwProto \sqsubseteq pat.\mathbf{nwProto}\ \wedge \\ \quad (pat.\mathbf{nwProto} = \mathsf{6} \Rightarrow tpPk\#_{tp}pat) \\ (\mathbf{Unknown}\ payload)\#_{nw}pat = \mathbf{true} \end{array}$$

$\boxed{tpPk\#_{tp}pat}$

$$\begin{array}{l} (\mathbf{TCP}\ tpSrc\ tpDst\ payload)\#_{tp}pat = \\ \quad tpSrc \sqsubseteq pat.\mathbf{tpSrc} \wedge tpDst \sqsubseteq pat.\mathbf{tpDst} \\ \mathbf{Unknown}\ payload\#_{tp}pat = \mathbf{true} \end{array}$$

$\boxed{n \sqsubseteq w}$

$$m \sqsubseteq n = m{=}n \qquad n \sqsubseteq \star = \mathbf{true}$$

Figure 4.5: Flow table syntax and semantics.

## 4.4 Flow Tables

The first step toward executing a NetCore program in an SDN involves compiling it to a prioritized set of forwarding rules—a *flow table.* Flow tables are an intermediate representation that play a similar role in NetCore to register transfer language (RTL) in traditional compilers. Flow tables are more primitive than NetCore programs because they lack the logical structure induced by NetCore operators such as union, intersection, negation, and restriction. Also, the patterns used to match packets in flow tables are more restrictive than NetCore predicates. And unlike NetCore programs, which denote total functions, flow tables are partial: switches redirect unmatched packets to the controller.

As defined in Fig. 4.5, a *flow table* consists of a multiset of rules $(n, pat, pts)$ where $n$ is an integer priority, $pat$ is a pattern, and $pts$ is a multiset of ports. A *pattern* is a record that associates each header field to either an integer constant $n$ or the special *wildcard* value $\star$. When writing flow tables, we often elide headers set to $\star$ in patterns as well as priorities when they are clear from context.

**Pattern semantics.** The semantics of patterns is given by the function $pk\#pat$, as defined in Fig. 4.5. This turns out to be subtly complicated, due to the representation of packets as sequences of nested frames—a pattern contains a (possibly wildcarded) field for every header field, but not all packets contain every header field. Some fields only exist in specific frame types (**dlTyp**) or protocols (**nwProto**). For example, only IP packets (**dlTyp** = `0x800`) have IP source and destination addresses. Likewise, TCP (**nwProto** = `6`) and UDP (**nwProto** = `17`) packets have source and destination ports, but ICMP (**nwProto** = `1`) packets do not.

To match on a given field, a pattern must specify values for all other fields it depends on. For example, to match on IP addresses, the pattern must also specify that the Ethernet frame type is IP:

$$\{\mathbf{dlTyp} = \texttt{0x800}, \mathbf{nwSrc} = \texttt{10.0.0.1}\}$$

If the frame type is elided, the value of the dependent header is silently ignored and the pattern is equivalent to a wildcard:

$$\{\mathbf{nwSrc} = \texttt{10.0.0.1}\} \equiv \{\}$$

In effect, patterns not only match packets, but also determine how they are parsed. This behavior, which was ambiguous in early versions of the OpenFlow specification (and later fixed), has lead to real bugs in existing controllers (Section 4.5). Although unintuitive for programmers, this behavior is completely consistent with how packet processing is implemented in modern switch hardware.

**Flow table semantics.** The semantics of flow tables is given by the relation $[\![\cdot]\!]$. The relation has two cases: one for matched packets and another for unmatched packets. Each flow table entry is a tuple containing a priority $n$, pattern $pat$, and a multiset of ports $\{\!|pt_1 \cdots pt_n|\!\}$. Given a packet and its input port, the semantics forwards the packet to all ports in the multiset associated with the highest-priority matching rule in the table. Otherwise, if no matching rule exists, it diverts the packet to the controller. In the formal semantics, the first component of the result pair represents forwarded packets while the second component represents diverted packets. Note that flow table matching is non-deterministic if there are multiple matching entries with the same priority. This has serious implications for a compiler—*e.g.*, naively combining flow tables with overlapping priorities could produce incorrect results. In the NetCore compiler, we avoid this issue by always working with unambiguous and total flow tables.

$\boxed{\mathcal{P} : sw \times pr \to [(pat, \mathbf{bool})]}$

$$
\begin{array}{lll}
\mathcal{P}(sw, \mathbf{dlSrc} = n) & = & [(\{\mathbf{dlSrc} = n\}, \mathbf{true})] \\
\mathcal{P}(sw, \mathbf{nwSrc} = n) & = & [(\{\mathbf{dlTyp} = \mathsf{0x800}, \mathbf{nwSrc} = n\}, \mathbf{true})] \\
 & \cdots & \\
\mathcal{P}(sw, \mathbf{at}\ sw) & = & [(\star, \mathbf{true})] \\
\mathcal{P}(sw, \mathbf{at}\ sw') & = & [(\star, \mathbf{false})] \quad \text{where } sw \neq sw' \\
\mathcal{P}(sw, \mathbf{not}\ pr) & = & [(pat_1, \neg b_1) \cdots (pat_n, \neg b_n), (\star, \mathbf{false})] \\
\multicolumn{3}{l}{\quad \text{where } [(pat_1, b_1) \cdots (pat_n, b_n)] = \mathcal{P}(sw, pr)} \\
\multicolumn{3}{l}{\mathcal{P}(sw, pr\ \mathbf{and}\ pr') =} \\
\multicolumn{3}{l}{\quad [(pat_1 \cap pat'_1, b_1 \wedge b'_1) \cdots (pat_m \cap pat'_n, b_m \wedge b'_n)]} \\
\multicolumn{3}{l}{\qquad \text{where } [(pat_1, b_1) \cdots (pat_m, b_m)] = \mathcal{P}(sw, pr)} \\
\multicolumn{3}{l}{\qquad \text{where } [(pat'_1, b'_1) \cdots (pat'_n, b'_n)] = \mathcal{P}(sw, pr')}
\end{array}
$$

$\boxed{\mathcal{C} : sw \times \phi \to [(pat, pt)]}$

$$
\begin{array}{l}
\mathcal{C}(sw, pr \Rightarrow pt) = [(pat_1, pt_1) \cdots (pat_n, pt_n), (\star, \{\!|\,|\!\})] \\
\quad \text{where } [(pat_1, b_1), \cdots, (pat_n, b_n)] = \mathcal{P}(sw, pr) \\
\quad \text{where } pt_i = pt \text{ if } b_i = \mathbf{true} \\
\quad \text{where } pt_i = \{\!|\,|\!\} \text{ if } b_i = \mathbf{false} \\
\mathcal{C}(sw, \phi \uplus \phi') \quad = \\
\quad [(pat_1 \cap pat'_1, pt_1 \uplus pt'_1), \cdots, (pat_m \cap pat'_n, pt_m \uplus pt'_n)] \mathbin{+\!\!+} \\
\quad [(pat_1, pt_1) \cdots (pat_m, pt_m)] \mathbin{+\!\!+} \\
\quad [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] \\
\qquad \text{where } [(pat_1, pt_1) \cdots (pat_m, pt_m)] = \mathcal{P}(sw, \phi) \\
\qquad \text{where } [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] = \mathcal{P}(sw, \phi')
\end{array}
$$

Figure 4.6: NetCore compilation.

## 4.5 Verified NetCore Compiler

With the syntax and semantics of NetCore and flow tables in place, we now present a verified compiler for NetCore. The compiler takes programs as input and generates a set of flow tables as output, one for every switch. The compilation algorithm is based on previous work [77], but we have verified its implementation in Coq. While building the compiler, we found two serious bugs in the original algorithm related to the handling of (unnatural) patterns in the compiler and flow table optimizer.

The compilation function $\mathcal{C}$, defined in Fig. 4.6, generates a flow table for a given switch $sw$. It uses the auxiliary function $\mathcal{P}$ to compile predicates. The compiler produces a list of pattern-action pairs, but priority numbers are implicit: the pair at the head has highest priority and each successive pair has lower priority.

Because NetCore programs denote total functions, packets not explicitly matched by any predicate are dropped. In contrast, flow tables divert unmatched packets to the controller. The compiler resolves this discrepancy by adding a catch-all rule that drops unmatched packets. For example, we compile the NetCore policy that forwards packets coming from the MAC address $H1$ to port 5 into a flow table with two rules, one that forwards these packets to port 5, and a lower priority rule that matches all (remaining) packets and drops them:

$$\mathcal{C}(sw, \textbf{dlSrc} = \mathit{H1} \Rightarrow \{\!|5|\!\}) = \{\!|(2, \{\textbf{dlSrc} = \mathit{H1}\}, \{\!|5|\!\}), (1, \star, \{\!|\,|\!\})|\!\}$$

The key operator used by the compiler constructs the cross-product of the flow tables provided as input. This operator can be used to compute intersections and unions of flow tables. Note that implementing union in the obvious way—by concatenating flow tables—would be wrong. The cross-product operator performs an element-wise intersection of the input flow tables and then merges their actions. To compile a union, we first use cross-product to build a flow table that represents the intersection, and then concatenate the flow tables for the sub-policies at lower priority. For example, the following NetCore program,

$$\textbf{dlSrc} = \mathit{H1} \Rightarrow \{\!|5|\!\} \uplus \textbf{dlDst} = \mathit{H2} \Rightarrow \{\!|10|\!\}$$

compiles to a flow table:

| Priority | Pattern | Action |
|---|---|---|
| 4 | $\{\textbf{dlSrc} = \mathit{H1}, \textbf{dlDst} = \mathit{H2}\}$ | $\{\!|5, 10|\!\}$ |
| 3 | $\{\textbf{dlSrc} = \mathit{H1}\}$ | $\{\!|5|\!\}$ |
| 2 | $\{\textbf{dlDst} = \mathit{H2}\}$ | $\{\!|10|\!\}$ |
| 1 | $\star$ | $\{\!|\,|\!\}$ |

The first rule matches all packets that match both sub-programs, while the second and third rules match packets only matched by the left and the right programs respectively. The final rule drops all other packets. The compilation of other predicates uses similar manipulations on flow tables.

We have built a large library of flow table manipulation operators in Coq, along with several lemmas that state useful algebraic properties about these operators. With this library, proving the correctness theorem for the NetCore compiler is simple—only about 200 lines of code in Coq.

**Theorem 7** (NetCore Compiler Soundness)**.** *For all NetCore programs $\phi$, switches sw, ports pt, and packets pk we have* $[\![\mathcal{C}(sw,\phi)]\!]\ pt\ pk = [\![\phi]\!]\ sw\ pt\ pk$.

Intuitively, this theorem states that a flow table compiled from a NetCore program for a switch *sw* has the same behavior as the NetCore program evaluated on packets at *sw*.

**Compiler bugs.** In the course of our work, we discovered that several unverified compilers from high-level network programming languages to flow tables suffer from bugs due to subtle pattern semantics. Section 4.4 described inter-field dependencies in patterns. For example, to match packets from IP address `10.0.0.1`, we write

$$\{\textbf{nwSrc} = \texttt{10.0.0.1}, \textbf{dlTyp} = \texttt{0x800}\}$$

and if we omit the **dlTyp** field, the IP address is silently ignored. This unintuitive behavior has led to bugs in PANE [22] and Nettle [109] as well as an unverified version of NetCore [77]. To illustrate, consider the following program:

$$\textbf{nwSrc} = \texttt{10.0.0.1} \Rightarrow \{\!|5|\!\}$$

In NetCore, this program matches all IP packets from `10.0.0.1` and forwards them out port 5. But the original NetCore compiler produced the following flow table for this program:

| Priority | Pattern | Action |
|---:|---|---|
| 2 | $\{$**nwSrc** $=$ `10.0.0.1`$\}$ | $\{\lvert 5 \rvert\}$ |
| 1 | $\star$ | $\{\lvert\rvert\}$ |

In OpenFlow, because the first pattern does not specify **dlTyp** $=$ `0x800`, it is actually equivalent to the all-wildcard pattern and this flow table sends *all* traffic out port 5. Both PANE and Nettle have similar bugs. Nettle has a special case to handle patterns with IP addresses that do not specify **dlTyp** $=$ `0x800`, but it does not correctly handle patterns that specify a transport port number but not the **nwProto** field. PANE suffers from the same bug. Even worse, these invalid patterns lead to further bugs when flow tables are combined and optimized by the compiler.

**Natural patterns.** The verified NetCore compiler does not suffer from the bug above. In our formal development, we require that all patterns manipulated by the compiler be what we call *natural patterns.* A natural pattern has the property that if the pattern specifies the value of a field, then all of that field's dependencies are also met. This rules out patterns such as $\{$**nwSrc** $=$ `10.0.0.1`$\}$, which omits the Ethernet frame type necessary to parse the IP address. Natural patterns are easy to define using dependent types in Coq. Moreover, we can calculate the cross-product of two natural patterns by intersecting fields point-wise. Hence, it is easy to prove that natural patterns are closed under intersection.

**Lemma 3.** *If $pat_1$ and $pat_2$ are natural patterns, then $pat_1 \cap pat_2$ is also a natural pattern.*

Another important property is that all patterns can be expressed as some equivalent natural pattern (where patterns are equivalent if they denote the same set of packets). This property tells us that we do not lose expressiveness by restricting to natural patterns.

**Lemma 4.** *If pat is an arbitrary pattern, then there exists a natural pattern $pat'$, such that $pat \equiv pat'$.*

These lemmas are used extensively in the proofs of correctness for our compiler and flow table optimizer.

**Flow table optimizer.** The basic NetCore compilation algorithm described so far generates flow tables that correctly implement the semantics of the input program. But many flow tables have redundant entries that could be safely removed. For example, a naive compiler might translate the program $(\star \Rightarrow \{\!|5|\!\})$ to the flow table $\{\!|(2, \star, \{\!|5|\!\}), (1, \star, \{\!||\!\})|\!\}$, which is equivalent to $\{\!|(2, \star, \{\!|5|\!\})|\!\}$. Worse, because the compilation rule for union uses a cross-product operator to combine the flow tables computed for sub-programs, the output can be exponentially larger than the input. Without an optimizer, such a naive compiler is essentially useless—*e.g.*, we built an unoptimized implementation of the algorithm in Fig. 4.6 and found that it ran out of memory when compiling a program consisting of just 9 operators.

Our compiler is parameterized on a function $\mathcal{O} : FT \to FT$, that it invokes at each recursive call. Because even simple policies can see a combinatorial explosion during compilation, this inline reduction is necessary. We stipulate that $\mathcal{O}$ must produce equivalent flow tables: $[\![\mathcal{O}(FT)]\!] = [\![FT]\!]$.

We have built an optimizer that eliminates low-priority entries whose patterns are fully subsumed by higher-priority rules and proved that it satisfies the above condition in Coq. Although this optimization is quite simple, it is effective in practice. In addition, earlier attempts to implement this optimization in NetCore had a bug that incorrectly identified certain rules as overlapping which we did not discover until developing this proof. The PANE optimizer also had a bug—it assumed that combining identical actions is always idempotent.

| | |
|---|---|
| Switch | $S ::= \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m)$ |
| Controller | $C ::= \mathbb{C}(\sigma, f_{in}, f_{out})$ |
| Link | $L ::= \mathbb{L}((sw_{src}, pt_{src}), pks, (sw_{dst}, pt_{dst}))$ |
| Link to Controller | $M ::= \mathbb{M}(sw, SMS, CMS)$ |

**Devices**

| | | |
|---|---|---|
| Ports on switch | $pts$ | $\in \{pt\}$ |
| Input/output buffers | $in_p, out_p$ | $\in \{\!\mid (pt, pk) \mid\!\}$ |
| Messages from controller | $in_m$ | $\in \{\!\mid SM \mid\!\}$ |
| Messages to controller | $out_m$ | $\in \{\!\mid CM \mid\!\}$ |

**Switch Components**

| | |
|---|---|
| Controller state | $\sigma$ |
| Controller input relation | $f_{in} \in sw \times CM \times \sigma \rightsquigarrow \sigma$ |
| Controller output relation | $f_{out} \in \sigma \rightsquigarrow sw \times SM \times \sigma$ |

**Controller Components**

| | |
|---|---|
| Queue from controller | $SMS \in [SM_1 \cdots SM_n]$ |
| Queue to controller | $CMS \in [CM_1 \cdots CM_n]$ |

**Controller Link**

| | |
|---|---|
| From controller | $SM ::= \textbf{FlowMod}\ \delta \mid \textbf{PktOut}\ pt\ pk \mid \textbf{BarrierRequest}\ n$ |
| To controller | $CM ::= \textbf{PktIn}\ pt\ pk \mid \textbf{BarrierReply}\ n$ |
| Table update | $\delta ::= \textbf{Add}\ n\ pat\ act \mid \textbf{Del}\ pat$ |

Figure 4.7: Featherweight OpenFlow syntax

Both of these bugs led to incorrect behavior.

## 4.6 Featherweight OpenFlow

The next step towards executing NetCore programs is a controller that configures the switches in the network. To prove that such a controller is correct, we need a model of the network. Unfortunately, the OpenFlow 1.0 specification, consisting of 42 pages of informal prose and C definitions, is not amenable to rigorous proof.

This section presents Featherweight OpenFlow, a detailed operational model that captures the essential features of OpenFlow networks, and yet still fits on a single page. The model elides a number of details such as error codes, counters, packet modification, and advanced configuration options such as the ability to enable and disable ports. But it does include all of the features related to how packets are forwarded and how flow tables are modified. Many existing SDN bug-finding and property-checking tools are based on similar

$$\frac{[\![FT]\!](pt,pk) \rightsquigarrow (\{\!|pt'_1 \cdots pt'_n|\!\}, \{\!|pk'_1 \cdots pk'_m|\!\}) \quad out = \{\!|\mathbf{PktIn}\ \ pt\ pk'_1 \cdots \mathbf{PktIn}\ \ pt\ pk'_m|\!\}}{\begin{array}{l}\mathbb{S}(sw, \_, FT, \{\!|(pt,pk)|\!\} \uplus in_p, out_p, \_, out_m) \\ \xrightarrow{(sw,pt,pk)} \mathbb{S}(sw, \_, FT, in_p, \{\!|(pt'_1,pk) \cdots (pt'_n,pk)|\!\} \uplus out_p, \_, out \uplus out_m)\end{array}} \text{(FWD)}$$

$$\frac{}{\mathbb{S}(sw, \ldots, \{\!|(pt,pk)|\!\} \uplus out_p, \ldots) \mid \mathbb{L}((sw,pt), pks, \_) \longrightarrow \mathbb{S}(sw, \ldots, out_p, \ldots) \mid \mathbb{L}((sw,pt), [pk] +\!\!+ pks, \_)} \text{(WIRE-SEND)}$$

$$\frac{}{\mathbb{L}(\_, pks +\!\!+ [pk], (sw,pt)) \mid \mathbb{S}(sw, \ldots, in_p, \ldots) \longrightarrow \mathbb{L}(\_, pks, (sw,pt)) \mid \mathbb{S}(sw, \ldots, \{\!|(pt,pk)|\!\} \uplus in_p, \ldots)} \text{(WIRE-RECV)}$$

$$\frac{}{\mathbb{S}(\ldots, FT, \ldots, \{\!|\mathbf{FlowMod\ Add}\ m\ pat\ act|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\ldots, FT \uplus \{\!|(m, pat, act)|\!\}, \ldots, in_m, \_)} \text{(ADD)}$$

$$\frac{FT_{rem} = \{\!|(n', pat', act') \mid (n', pat', act') \in FT \text{ and } pat \neq pat'|\!\}}{\mathbb{S}(\ldots, FT, \ldots, \{\!|\mathbf{FlowMod\ Del}\ pat|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\ldots, FT_{rem}, \ldots, in_m, \_)} \text{(DEL)}$$

$$\frac{pt \in pts}{\mathbb{S}(\_, pts, \ldots, out_p, \{\!|\mathbf{PktOut}\ pt\ pk|\!\} \uplus in_m, \_) \longrightarrow \mathbb{S}(\_, pts, \ldots, \{\!|(pt,pk)|\!\} \uplus out_p, in_m, \_)} \text{(PKTOUT)}$$

$$\frac{f_{out}(\sigma) \rightsquigarrow (sw, SM, \sigma')}{\mathbb{C}(\sigma, \_, \_) \mid \mathbb{M}(sw, SMS, \_) \longrightarrow \mathbb{C}(\sigma', \_, \_) \mid \mathbb{M}(sw, [SM] +\!\!+ SMS, \_)} \text{(CTRL-SEND)}$$

$$\frac{f_{in}(sw, \sigma, CM) \rightsquigarrow \sigma'}{\mathbb{C}(\sigma, f_{in}, \_) \mid \mathbb{M}(sw, \_, CMS +\!\!+ [CM]) \longrightarrow \mathbb{C}(\sigma', f_{in}, \_) \mid \mathbb{M}(sw, \_, CMS)} \text{(CTRL-RECV)}$$

$$\frac{SM \neq \mathbf{BarrierRequest}\ n}{\mathbb{M}(sw, SMS +\!\!+ [SM], \_) \mid \mathbb{S}(sw, \ldots, in_m, \_) \longrightarrow \mathbb{M}(sw, SMS, \_) \mid \mathbb{S}(sw, \ldots, \{\!|SM|\!\} \uplus in_m, \_)} \text{(SWITCH-RECV-CTRL)}$$

$$\frac{}{\begin{array}{l}\mathbb{M}(sw, SMS +\!\!+ [\mathbf{BarrierRequest}\ n], \_) \mid \mathbb{S}(sw, \ldots, \{\!||\!\}, out_m) \\ \longrightarrow \mathbb{M}(sw, SMS, \_) \mid \mathbb{S}(sw, \ldots, \{\!||\!\}, \{\!|\mathbf{BarrierReply}\ n|\!\} \uplus out_m)\end{array}} \text{(SWITCH-RECV-BARRIER)}$$

$$\frac{}{\mathbb{S}(sw, \ldots, \{\!|CM|\!\} \uplus out_m) \mid \mathbb{M}(sw, \_, CMS) \longrightarrow \mathbb{S}(sw, \ldots, out_m) \mid \mathbb{M}(sw, \_, [CM] +\!\!+ CMS)} \text{(SWITCH-SEND-CTRL)}$$

$$\frac{Sys_1 \longrightarrow Sys'_1}{Sys_1 \mid Sys_2 \longrightarrow Sys'_1 \mid Sys_2} \text{(CONGRUENCE)}$$

Figure 4.8: Featherweight OpenFlow semantics.

(informal) models [53, 50, 14]. We believe Featherweight OpenFlow could also serve as a foundation for these tools.

### 4.6.1 OpenFlow Semantics

Initially, every switch has an empty flow table that diverts all packets to the controller. Using **FlowMod** messages, the controller can insert new table entries to have the switch process packets itself. A non-trivial program may compile to several thousand flow table entries, but

**FlowMod** messages only add a single entry at a time. In general, many **FlowMod** messages will be needed to fully configure a switch. However, OpenFlow is designed to give switches a lot of latitude to enable efficient processing, often at the expense of programmability and understandability:

- **Pattern semantics.** As discussed in preceding sections, the semantics of flow tables is non-trivial: patterns have implicit dependencies and flow tables can have multiple, overlapping entries. (The OpenFlow specification itself notes that scanning the table to find overlaps is expensive.) Therefore, it is up to the controller to avoid overlaps that introduce non-determinism.
- **Packet reordering.** Switches may reorder packets arbitrarily. For example, switches often have both a "fast path" that uses custom packet-processing hardware and a "slow path" that processes packets using a slower general-purpose CPU.
- **No acknowledgments.** Switches do not acknowledge when **FlowMod** messages are processed, except when errors occur. The controller can explicitly request acknowledgments by sending a *barrier request* after a **FlowMod**. When the switch has processed the **FlowMod** (and all other messages received before the *barrier request*), it responds with a *barrier reply*.
- **Control message reordering.** Switches may process control messages, including **FlowMod** messages, in any order. This is based on the architecture of switches, where the logical flow table is implemented by multiple physical tables working in parallel—each physical table typically only matches headers for one protocol. To process a rule with a pattern such as $\{\mathbf{nwSrc} = \texttt{10.0.0.1}, \mathbf{dlTyp} = \texttt{0x800}\}$, which matches headers across several protocols, several physical tables may need to be reconfigured, which takes longer to process than a simple pattern such as $\{\mathbf{dlDst} = \texttt{H2}\}$.

Figure 4.8 defines the syntax and semantics of Featherweight OpenFlow, which faithfully models all of these behaviors. The rest of this section discusses the key elements of the model in detail.

### 4.6.2 Network Elements

Featherweight OpenFlow has four kinds of elements: switches, controllers, links between switches (carrying data packets), and links between switches and the controller (carrying OpenFlow messages). The semantics is specified using a small-step relation, with elements interacting by passing messages and updating their state non-deterministically.

**Switches.** A switch $\mathbb{S}$ comprises a unique identifier $sw$, a set of ports $pts$, and input and output packet buffers $in_p$ and $out_p$. The buffers are multisets of packets tagged with ports, $(pt, pk)$. In the input buffer, packets are tagged with the port on which they were received. In the output buffer, packets are tagged with the port on which they will be sent out. Since buffers are unordered, switches can process packets in any order. Switches also have a flow table, $FT$, which determines how the switch processes packets. As detailed in Section 4.4, the table is a collection of flow table entries, where each entry has a priority, pattern and, a multiset of output ports. Each switch also has a multiset of messages to and from the controller, $out_m$ and $in_m$. There are three kinds of messages from the controller:

- **PktOut** $pt\ pk$ instructs the switch to emit packet $pk$ on port $pt$.
- **FlowMod** $\delta$ instructs the switch to add or delete entries from its flow table. When $\delta$ is **Add** $n\ pat\ act$, a new entry is created, whereas **Del** $pat$ deletes all entries that match $pat$ exactly. In our model, we assume that flow tables on switches can be arbitrarily

large. This is not the case for hardware switches, where the size of flow tables is often constrained by the amount of silicon used, and varies from switch-to-switch. It would be straightforward to modify our model to bound the size of the table on each switch.

- **BarrierRequest** $n$ forces the switch to process all outstanding messages before replying with a **BarrierReply** $n$ message.

**Controllers.** A controller $\mathbb{C}$ is defined by its local state $\sigma$, an input relation $f_{in}$, and an output relation $f_{out}$. The local state and these relations are application-specific, so Featherweight OpenFlow can be instantiated with any controller whose behavior can be modeled in this way. The $f_{out}$ relation sends a message to a switch while $f_{in}$ receives a message from a switch. Both relations update the state $\sigma$. There are two kinds of messages a switch can send to the controller:

- **PktIn** *pt pk* indicates that packet *pk* was received on *pt* and did not match any entry in the flow table.
- **BarrierReply** $n$ indicates that *sw* has processed all messages up to and including a **BarrierRequest** $n$ sent earlier.

**Data links.** A data link $\mathbb{L}$ is a unidirectional queue of packets between two switch ports. To model bidirectional links we use symmetric unidirectional links. Featherweight OpenFlow does not model packet-loss in links and packet-buffers. It would be easy to extend our model so that packets are lost, for example, with some probability. Without packet loss, a packet traces paths from its source to its destinations (or loops forever). With packet loss, a packet traces a prefix of the complete path given by our model under ideal conditions.

|  |  |
|---|---|
| Location | $loc ::= sw \times pt$ |
| Located packet | $lp ::= loc \times pk$ |
| Topology | $T \in loc \rightharpoonup loc$ |

$$\boxed{\phi, T \vdash \{\!|lp|\!\} \overset{lp}{\Rightarrow} \{\!|lp|\!\}}$$

$$\frac{lps' = \{\!|(T(sw, pt_{out}), pk) \mid (pt_{out}, pk) \in [\![\phi]\!]\ sw\ pt\ pk|\!\}}{\begin{array}{ll}\phi, T \vdash & \{\!|((sw, pt), pk)|\!\} \uplus \{\!|lp_1 \cdots lp_n|\!\} \xrightarrow{(sw,pt,pk)} \\ & lps' \uplus \{\!|lp_1 \cdots lp_n|\!\}\end{array}}$$

Figure 4.9: Network semantics.

**Control links.** A control link $\mathbb{M}$ is a bidirectional link between the switch and the controller that contains a queue of controller messages for the switch and a queue of switch messages headed to the controller. Messages between the controller and the switch are sent and delivered in order, but may be processed in any order.

## 4.7 Verified Run-Time System

So far, we have developed a semantics for NetCore (Section 4.3), a compiler from NetCore to flow tables (Section 4.4), and a low-level semantics for OpenFlow (Section 4.6). To actually execute NetCore programs, we also need to develop a run-time system that installs rules on switches and prove it correct.

### 4.7.1 NetCore Run-Time System

There are many ways to build a controller that implements a NetCore run-time system. A trivial solution is to simply process all packets on the controller. The controller receives input packets as **PktIn** messages, evaluates them using the NetCore semantics, and emits the outputs using **PktOut** messages.

Of course, we can do much better by using the NetCore compiler to actually generate flow tables and install those rules on switches using **FlowMod** messages. For example, given the following program,

$$\textbf{dlDst} = \texttt{H1} \textbf{ and not}(\textbf{dlTyp} = \texttt{0x800}) \Rightarrow \{\!|1|\!\}$$

the compiler might generate the following flow table,

| Priority | Pattern | Action |
|---:|---|---|
| 5 | $\{\textbf{dlDst} = \texttt{H1}, \textbf{dlTyp} = \texttt{0x800}\}$ | $\{\!|\,|\!\}$ |
| 4 | $\{\textbf{dlDst} = \texttt{H1}\}$ | $\{\!|1|\!\}$ |
| 3 | $\star$ | $\{\!|\,|\!\}$ |

and the controller would emit three **FlowMod** messages:

$$\begin{aligned}&\textbf{Add}\ 5\ \{\textbf{dlDst} = \texttt{H1}, \textbf{dlTyp} = \texttt{0x800}\}\ \{\!|\,|\!\}\\ &\textbf{Add}\ 4\ \{\textbf{dlDst} = \texttt{H1}\}\ \{\!|1|\!\}\\ &\textbf{Add}\ 3\ \star\ \{\!|\,|\!\}\end{aligned}$$

However, it would be unsafe to emit just these messages. As discussed in Section 4.6, switches can reorder messages to maximize throughput. This can lead to transient bugs by creating intermediate flow tables that are inconsistent with the intended policy. For example, if the **Add** 3 $\star$ $\{\!|\,|\!\}$ message is processed first, all packets will be dropped. Alternatively, if **Add** 4 $\{\textbf{dlDst} = \texttt{H1}\}$ $\{\!|1|\!\}$ is processed first, traffic that should be dropped will be incorrectly forwarded. Of the six possible permutations, only one has the property that all intermediate states either (i) process packets according to the program, or (ii) send packets to the controller (which can evaluate them using the program). Therefore, to ensure the switch processes the messages in order, the run-time system must intersperse **BarrierRequest** messages between **FlowMod** messages.

**Network semantics.** The semantics of NetCore presented in Section 4.3 defines how a program processes a single packet at a single switch at a time. But Featherweight OpenFlow models the behavior of an entire network of switches with multiple packets in-flight. To reconcile the difference between these two, we need a *network semantics* that models the processing of all packets in the network. In this semantics (Fig. 4.9), the system state is a multiset of in-flight located packets $\{\!|lp|\!\}$. At each step, the system:

1. Removes a located packet $((sw, pt), pk)$ from its state,
2. Processes the packet according to the program to produce a new multiset of located packets,
$$\{\!| lp_1 \cdots lp_n |\!\} = [\![\phi]\!]\ sw\ pt\ pk,$$
3. Transfers these packets to input ports, using the topology, $T(lp_1) \cdots T(lp_n)$, and
4. Adds the transferred packets to the system state.

Note that this approach to constructing a network semantics is not specific to NetCore: any hop-by-hop packet processing function could be used. Below, we refer to any semantics constructed in this way as a *network semantics.*

### 4.7.2 Run-Time System Correctness

Now we are ready to prove the correctness of the NetCore run-time system. However, rather than proving this directly, we instead develop a general framework for establishing controller correctness, and obtain the result for NetCore as a special case.

**Bisimulation equivalence.** The inputs to our framework are: (i) the high-level, hop-by-hop function the network is intended to implement, and (ii) the controller implementation, which is required to satisfy natural safety and liveness conditions. Given these parameters, we construct a *weak bisimulation* between the network semantics of the high-level function and an OpenFlow network instantiated with the controller implementation. This construction handles a number of low-level details once and for all, freeing developers to focus on essential controller correctness properties.

We prove a weak (rather than strong) bisimulation[2] because Featherweight OpenFlow models the mechanics of packet processing in much greater detail than in the network semantics. For example, consider a NetCore program that forwards a packet *pk* from one switch to another, say *S1* to *S2*, in a single step. An equivalent Featherweight OpenFlow implementation would require at least three steps: (i) process *pk* at *S1*, move *pk* from the input buffer to the output buffer, (ii) move *pk* from *S1*'s output buffer to the link to *S2*, and (iii) move *pk* from the link to *S2*'s input buffer. If there were other packets on the link (which is likely!), additional steps would be needed. Moreover, *pk* could take an even more circuitous route if it is redirected to the controller.

The weak bisimulation states that the NetCore and Featherweight OpenFlow are indistinguishable modulo internal steps. Hence, any reasoning about the trajectory of a packet at the NetCore level will be preserved in Featherweight OpenFlow.

**Observations.** To define a weak bisimulation, we need a notion of observation (called an action in the $\pi$-calculus). We say that the NetCore network semantics observes a packet

[2]A weak bisimulation identifies states of two systems as equivalent modulo unobservable (internal) transitions. A strong bisimulation does not allow a system to perform unobservable transitions to catch up. For example, a pipelined processor may split one logical operation into two steps, with an unobservable transition between the processing of the steps. The pipelined processor would be weakly bisimilar to a processor that performed the operation in one step, but not strongly bisimilar.

$(sw, pt, pk)$ when it selects the packet from its state—i.e., just before evaluating it. Likewise, a Featherweight OpenFlow program observes a packet $(sw, pt, pk)$ when it removes $(pt, pk)$ from the input buffer on $sw$ to process it using the FWD rule.

**Bisimulation relation.** Establishing a weak bisimulation requires exhibiting a relation $\approx_{OF}$ between the concrete and abstract states with certain properties. We relate packets located in links and buffers in Featherweight OpenFlow to packets in the abstract network semantics. We elide the full definition of the relation, but describe some of its key characteristics:

- Packets $(pt, pk)$ in input buffers $in_p$ on $sw$ are related to packets $((sw, pt), pk)$ in the abstract state.
- Packets $(pt, pk)$ in output buffers $out_p$ on $sw$ are related to packets located at the other side of the link connected to $pt$.
- Likewise, packets on a data link (or contained in **PktOut** messages) are related to packets located at the other side of the data link (or the link connected to the port in the message).

Intuitively, packets in output buffers have already been processed and observed. The network semantics moves packets to new locations in one step whereas OpenFlow requires several more steps, but we must not be able to observe these intermediate steps. Therefore, after Featherweight OpenFlow observes a concrete packet $pk$ (in the FWD rule), subsequent copies of $pk$ must be related to packets at the ultimate destination.

The structure of the relation is largely straightforward and dictated by the nature of Featherweight OpenFlow. However, a few parts are application specific. In particular, packets at the controller and packets sent to the controller in **PktIn** messages may relate to the state in the network semantics in application-specific ways.

**Abstract semantics.** So far, we have focused on NetCore to build intuitions. But our bisimulation can be obtained for any controller that implements a high-level packet-processing function. We now make this precise with a few additional definitions.

**Definition 2** (Abstract Semantics)**.** *An abstract semantics is defined by the following components:*

1. *An abstract packet-processing function on located packets:*

$$f(lp) = \{\!|\, lp_1 \cdots lp_n \,|\!\}$$

2. *An abstraction function, $c : \sigma \to \{\!|\, lp \,|\!\}$, that identifies the packets the controller has received but not yet processed.*

Note that the type of the NetCore semantics (Fig. 4.9) matches the type of the function above. In addition, because the NetCore controller simply holds the multiset of **PktIn** messages, the abstraction function is trivial. Given such an abstract semantics, we can lift it to a network semantics $\overset{lp}{\Rightarrow}$ as we did for NetCore.

We say that an abstract semantics is *compatible* with a concrete controller implementation, consisting of a type of controller state $\sigma$, and input and output relations $f_{in}$ and $f_{out}$, if the two satisfy the following conditions relating their behavior:

**Definition 3** (Compatibility)**.** *An abstract semantics and controller implementation are* compatible *if:*

1. *The controller ensures that all times packets are either (i) processed by switches in accordance with the packet-processing function or (ii) sent to the controller for processing;*

2. *Whenever the controller receives a packet,*

$$(sw, \textbf{\textit{PktIn}}\ \ pt\ pk, \sigma) \rightsquigarrow \sigma'$$

   *it applies the packet-processing function $f$ to $pk$ to get a multiset of located packets and adds them to its state*

$$c(\sigma') = c(\sigma) \uplus f(pk)$$

3. *Whenever the controller emits a packet,*

$$\sigma \rightsquigarrow (sw, \textbf{\textit{PktOut}}\ \ pt\ pk, \sigma')$$

   *it removes the packet from its state:*

$$c(\sigma') = c(\sigma) \setminus \{\!|(sw, pt, pk)|\!\}$$

4. *The controller eventually processes all packets $(sw, pt, pk)$ in its state $c(\sigma)$ according to the packet-processing function, and*

5. *The controller eventually processes all OpenFlow messages.*

The first property is essential. If it did not hold, switches could process packets contrary to the intended packet-processing relation. Proving it requires reasoning about the messages sent to the switches by the controller. In particular, because switches may reorder messages, barriers must be interspersed appropriately. The second and third properties relate the abstraction function $c$ and the controller implementation. The fourth property requires the controller to correctly process every packet it receives. The fifth property is a liveness condition requiring the controller to eventually process every OpenFlow message. This holds in the absence of failures on the control link and the controller itself.

Given such a semantics, we show that our relation between abstract and Featherweight OpenFlow states and its inverse are weak simulations. This implies that the relation is a weak bisimulation, and thus that the two systems are weakly bisimilar.

**Theorem 8** (Weak Bisimulation)**.** *For all compatible abstract semantics and controller implementations, all Featherweight OpenFlow states $s$ and $s'$, and all abstract states $t$ and $t'$:*

- *If $s \approx_{OF} t$ and $s \xrightarrow{(sw,pt,pk)} s'$, then there exists an abstract network state $t''$ such that $t \overset{(sw,pt,pk)}{\Longrightarrow} t''$ and $s' \approx_{OF} t''$, and*
- *If $s \approx_{OF} t$ and $t \overset{(sw,pt,pk)}{\Longrightarrow} t'$, then there exists a Featherweight OpenFlow state $s''$, and abstract network states $s_i, s_i'$ such that*

$$s \longrightarrow^* s_i \xrightarrow{(sw,pt,pk)} s_i' \longrightarrow^* s''$$

  *and $s'' \approx_{OF} t'$.*

In this theorem, portions of the $\approx_{OF}$ relation are defined in terms of the controller abstraction function, $c$ supplied as a parameter. In addition, the proofs themselves rely on compatibility (Definition 3).

Finally, we instantiate this theorem for the NetCore controller:

**Corollary 2** (NetCore Run-Time Correctness)**.** *The network semantics of NetCore is weakly bisimilar to the concrete semantics of the NetCore controller in Featherweight OpenFlow.*

## 4.8 Implementation and Evaluation

We have built a complete working implementation of the system described in this chapter, including machine-checked proofs of each of the lemmas and theorems. Our implementation is available under an open-source license at the following URL:

`http://frenetic-lang.org`

Our system consists of 12 KLOC of Coq, which we extract to OCaml and link against two unverified components:

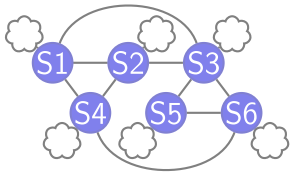
- A library to serialize OpenFlow data types to the OpenFlow wire format. This code is a lightly modified version of the Mirage OpenFlow library [64] (1.4K LOC).
- A module to translate between the full OpenFlow protocol and the fragment used in Featherweight OpenFlow (200 LOC).

We have deployed our NetCore controllers on real hardware and used them to build a number of useful network applications including host discovery, shortest-path routing, spanning tree, access control, and traffic monitoring. Using the union operator, it is easy to compose these modules with others to form larger applications.

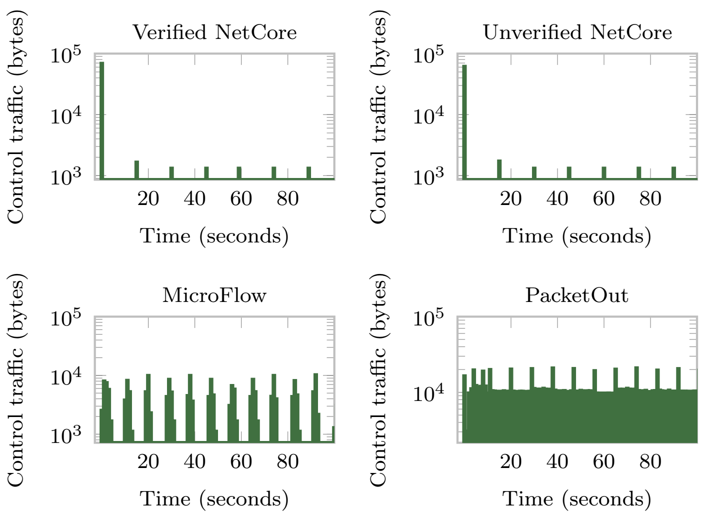
**Controller throughput.** Controller throughput is important for the performance of SDNs. The CBench [104] tool quantifies controller throughput by flooding the controller with **PktIn** messages and measuring the time taken to receive **PktOut** messages in response. This is a somewhat crude metric, but it is still effective, since any controller must respond to **PktIn** messages. We used CBench to compare the throughput of our verified controller with our previous unverified NetCore controller, written in Haskell, and with the popular POX and NOX controllers, written in Python and C++ respectively. To ensure that the experiment tested throughput and not the application running on it, we had each controller execute a trivial program that floods all packets. We ran the experiment on a dual-core 3.3 GHz Intel i3 with 8GB RAM with Ubuntu 12.04 and obtained the results shown in Fig. 4.10 (a).

| Controller | Msgs/sec |
|---|---|
| Unverified NetCore | 26,022 |
| NOX | 16,997 |
| **Verified NetCore** | **9,437** |
| POX | 6,150 |

(a)



(b)



(c)

Figure 4.10: Experiments: (a) controller throughput results; (b) control traffic topology; (c) control traffic results.

Our unverified NetCore controller is significantly faster than our verified controller. We attribute this to (i) a more mature backend that uses an optimized library from Nettle [109] to serialize messages, and (ii) Haskell's superior multicore support, which the controller exploits heavily. However, despite being slower than the original NetCore, the new controller is still fast enough to be useful—indeed, it is faster than the popular POX controller (although POX is not tuned for performance). We plan to optimize our controller to improve its performance in the future.

**Control traffic.** Another key factor that affects SDN performance is the amount of traffic that the controller must handle. This metric measures the effectiveness of the controller at compiling, optimizing, and installing forwarding rules rather than processing packets itself. To properly assess a controller on these points, we need a more substantial application than "flood all packets." Using NetCore, we built an application that computes shortest path forwarding rules as well as a spanning tree for broadcast. We ran this program on the six-switch Waxman topology shown in Fig. 4.10 (b), with two hosts connected to each switch.

In the experiment, every host sent 10 ICMP (ping) broadcast packets along the spanning tree, and received the replies from other hosts along shortest path routes. We used Mininet [37] to simulate the network and collected traffic traces using `tcpdump`. The total amount of network traffic during the experiment was 372 Kb.

We compared our Verified NetCore controller to several others: a (verified) "PacketOut" controller that never installs forwarding rules and processes all packets itself; our previous "Unverified NetCore" controller, written in Haskell; and a reactive "MicroFlow" controller [24] written in Haskell. The results of the experiment are shown in Fig. 4.10 (c). The graphs plot time-series data for every controller, showing the amount of control traffic in each one-second interval. Note that the $y$ axis is on a logarithmic scale.

In the plot for our Verified NetCore controller, there is a large spike in control traffic at the start of the experiment, where the controller sends messages to install the forwarding rules generated from the program. Additional control traffic appears every 15 seconds; these messages implement a simple keep-alive protocol between the controller and switches. The Unverified NetCore controller uses the same compilation and run-time system algorithms as our verified controller, so its plot is nearly identical. The MicroFlow controller installs individual fine-grained rules in response to individual traffic flows rather than proactively compiling complete flow tables. Accordingly, its plot shows that there is much more control traffic than for the two NetCore controllers. The graph shows how traffic spikes when multiple hosts respond simultaneously to an ICMP broadcast. The fourth plot shows the behavior of the PacketOut controller. Because this controller does not install any forwarding rules on the switches, all data traffic flows to the controller and then back into the network.

Although these results are preliminary, we believe they demonstrate that the performance of our verified NetCore controller can be competitive with other controllers. In particular, our verified controller generates the same flow tables and handles a similar amount of traffic as the earlier unverified NetCore controller, which was written in Haskell. Moreover, our system is not tuned for performance. As we optimize and extend our system, we expect that its performance will only improve.

## 4.9 Conclusions

This chapter presented a formal foundation for network reasoning: a detailed model of OpenFlow, formalized in the Coq proof assistant, and a machine-verified compiler and run-time system for the NetCore programming language. The main result is a general framework for establishing controller correctness that reduces the proof obligation to a small number of safety and liveness properties.

CHAPTER 5

# REASONING ABOUT NETWORK UPDATES

*"Nothing endures but change."*

—Heraclitus

In this chapter, we show how to move a network between different configurations in such a way that the network preserves the behavior of the configurations, even while it is in transition. We present network update abstractions, implementations and optimizations of those abstractions, and a formal model of updates in software-defined networks to formally specify our abstractions and prove them correct.

The work in this chapter is based upon a 2012 SIGCOMM paper [93] written with Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker.

## 5.1 Introduction

The techniques in the previous chapters show how to take a network specification and a network program, and build a system that provably satisfies the specification. But real-world networks exist in a constant state of flux. Operators frequently modify routing tables and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. Simply implementing a single network program correctly is not sufficient: we need to update the network program over time in response to changes in the network, and still maintain invariants, even while updates are in progress.

But network updates are difficult to perform correctly: even when planned well in advance

| **Example Application** | **Policy Change** | **Desired Property** | **Practical Implications** |
|---|---|---|---|
| Stateless firewall | Changing access control list | No security holes | Admitting malicious traffic |
| Planned maintenance [28, 91] | Shut down a node/link | No loops/blackholes | Packet/bandwidth loss |
| Traffic engineering [27, 91] | Changing a link weight | No loops/blackholes | Packet/bandwidth loss |
| VM migration [19] | Move server to new location | No loops/blackholes | Packet/bandwidth loss |
| IGP migration [108] | Adding route summarization | No loops/blackholes | Packet/bandwidth loss |
| Traffic monitoring | Changing traffic partitions | Consistent counts | Inaccurate measurements |
| Server load balancing [39, 111] | Changing load distribution | Connection affinity | Broken connections |
| NAT or stateful firewall | Adding/replacing equipment | Connection affinity | Outages, broken connections |

Table 5.1: Example changes to network configuration, and the desired update properties.

they can result in disruptions such as transient outages, lost server connections, unexpected security vulnerabilities, hiccups in VoIP calls, or the death of a player's favorite character in an online game.

To address these problems, researchers have proposed a number of extensions to protocols and operational practices that aim to prevent transient anomalies [29, 28, 46, 91, 108]. However, each of these solutions is limited to a specific protocol (*e.g.*, OSPF and BGP) and a specific set of properties (*e.g.*, freedom from loops and blackholes) and increases the complexity of the system considerably. Hence, in practice, network operators have little help when designing a new protocol or trying to ensure an additional property not covered by existing techniques. A list of example applications and their properties is summarized in Table 5.1.

Instead of relying on point solutions for network updates, this chapter presents foundational principles for designing solutions that are applicable to a wide range of protocols and properties. These solutions come with two parts: (1) an abstract interface that offers strong, precise, and intuitive semantic guarantees, and (2) concrete mechanisms that faithfully implement the semantics specified in the abstract interface. Programmers can use the interface to build robust applications on top of a reliable foundation. The mechanisms, while possibly complex, should be implemented once by experts, tuned and optimized, and used over and over, much like register allocation or garbage collection in a high-level programming language.

Instead of requiring SDN programmers to implement configuration changes using today's low-level interfaces, our high-level, abstract operations allow the programmer to update the configuration of the entire network in one fell swoop. The libraries implementing these abstractions provide strong semantic guarantees about the observable effects of the global updates, and handle all of the details of transitioning between old and new configurations efficiently.

**Abstractions** Our central abstraction is *per-packet consistency*, the guarantee that every packet traversing the network is processed by exactly one consistent global network configuration. When a network update occurs, this guarantee persists: each packet is processed either using the configuration in place prior to the update, or the configuration in place after the update, but never a mixture of the two. Note that this consistency abstraction is *more* powerful than an "atomic" update mechanism that simultaneously updates all switches in the network. Such a simultaneous update could easily catch many packets in flight in the middle of the network, and such packets may wind up traversing a mixture of configurations, causing them to be dropped or sent to the wrong destination. We also introduce *per-flow consistency*, a generalization of per-packet consistency that guarantees all packets in the same flow are processed with the same configuration. This stronger guarantee is needed in applications such as HTTP load balancers, which need to ensure that all packets in the same TCP connection reach the same server replica to avoid breaking connections.

To support these abstractions, we develop several *update mechanisms* that use features commonly available on OpenFlow switches. Our most general mechanism, which enables transition between any two configurations, performs a two-phase update of the rules in the new configuration onto the switches. The other mechanisms are optimizations that achieve better performance under circumstances that arise often in practice. These optimizations transition to new configurations in less time, update fewer switches, or fewer rules.

To analyze our abstractions and mechanisms, we develop a simple, formal model that captures the essential features of OpenFlow networks. This model allows us to define a class of network properties, called *trace properties*, that characterize the paths individual packets take through the network. The model also allows us to prove a remarkable result: if *any* trace property $P$ holds of a network configuration prior to a per-packet consistent update as well as after the update, then $P$ also holds continuously throughout the update process. This illustrates the true power of our abstractions: programmers do not need to specify *which* trace properties our system must maintain during an update, because a per-packet consistent update preserves *all* of them! For example, if the old and new configurations are free from forwarding loops, then the network will be loop-free before, during, and after the update. In addition to the proof sketch included in this chapter, this result has been formally verified in the Coq proof assistant [9].

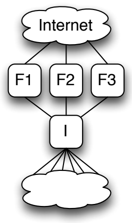
An important and useful corollary of these observations is that it is possible to take any verification tool that checks trace properties of *static* network configurations and transform it into a tool that checks invariance of trace properties as the network configurations evolve *dynamically*—it suffices to check the static policies before and after the update. Indeed, the techniques and systems in the previous chapter all verify trace properties of static configurations, dovetailing perfectly with the developments here.

**Contributions** This chapter makes the following contributions:

- **Update abstractions:** We propose per-packet and per-flow consistency as canonical, general abstractions for specifying network updates (Sections 5.2 and 5.6).
- **Update mechanisms:** We describe OpenFlow-compatible implementation mechanisms and several optimizations tailored to common scenarios (Sections 5.5 and 5.8).
- **Theoretical model:** We develop a mathematical model that captures the essential behavior of SDNs, and we prove that the mechanisms correctly implement the abstractions (Section 5.3). We have formalized the model and proved the main theorems in the Coq proof assistant.
- **Implementation:** We describe a prototype implementation on top of the OpenFlow/NOX platform (Section 5.8).
- **Experiments:** We present results from experiments run on small, but canonical applications that compare the total number of control messages and rule overhead needed to implement updates in each of these applications (Section 5.8).

## 5.2 Example

To illustrate the challenges surrounding network updates, consider an example network with one ingress switch I and three "filtering" switches FW1, FW2, and FW3, each sitting between I and the rest of the Internet, as shown on the left side of Figure 5.1. Several classes of traffic are connected to I: untrustworthy packets from Unknown and Guest hosts, and trustworthy packets from Student and Faculty hosts. At all times, the network should enforce a security policy that denies SSH traffic from untrustworthy hosts, but allows all other traffic to pass through the network unmodified. We assume that any of the filtering switches have the capability to perform the requisite monitoring, blocking, and forwarding.



**Configuration I**

| | Type | Action |
|---|---|---|
| I | $U, G$ | Forward FW1 |
| | $S$ | Forward FW2 |
| | $F$ | Forward FW3 |
| FW1 | $SSH$ | Monitor |
| | $*$ | Allow |
| FW2 | $*$ | Allow |
| FW3 | $*$ | Allow |

→

**Configuration II**

| | Type | Action |
|---|---|---|
| I | $U$ | Forward FW1 |
| | $G$ | Forward FW2 |
| | $S, F$ | Forward FW3 |
| FW1 | $SSH$ | Monitor |
| | $*$ | Allow |
| FW2 | $SSH$ | Monitor |
| | $*$ | Allow |
| FW3 | $*$ | Allow |

Figure 5.1: Access control example.

There are several ways to implement this policy, and depending on the traffic load, one may be better than another. Suppose that initially we configure the switches as shown in the leftmost table in Figure 5.1: switch I sends traffic from U and G hosts to FW1, from S hosts to FW2, and from F hosts to FW3. Switch FW1 monitors (and denies) SSH packets and allows all other packets to pass through, while FW2 and FW3 simply let all packets pass through.

Now, suppose the load shifts, and we need more resources to monitor the untrustworthy traffic. We might reconfigure the network as shown in the table on the right of Figure 5.1, where the task of monitoring traffic from untrustworthy hosts is divided between FW1 and FW2, and all traffic from trustworthy hosts is forwarded to FW3. Because we cannot update the network all at once, the individual switches need to be reconfigured one-by-one. However, if we are not careful, making incremental updates to the individual switches can lead to intermediate configurations that violate the intended security policy. For instance, if we start by updating FW2 to deny SSH traffic, we interfere with traffic sent by trustworthy hosts. If, on the other hand, we start by updating switch I to forward traffic according to the new configuration (sending U traffic to FW1, G traffic to FW2, and S and F traffic to FW3), then SSH packets from untrustworthy hosts will incorrectly be allowed to pass through the network. There is one valid transition plan:

1. Update I to forward S traffic to FW3, while continuing to forward U and G traffic to FW1 and F traffic to FW3.
2. Wait until in-flight packets have been processed by FW2.
3. Update FW2 to deny SSH packets.
4. Update I to forward G traffic to FW2, while continuing to forward U traffic to FW1 and S and F traffic to FW3.

But finding this ordering and verifying that it behaves correctly requires performing intricate reasoning about a sequence of intermediate configurations—something that is tedious and error-prone, even for this simple example. Even worse, in some examples it is impossible to find an ordering that implements the transition simply by adding one part of the new configuration at a time (*e.g.*, if we swap the roles of FW1 and FW3 while enforcing the intended security policy). In general, more powerful update mechanisms are needed.

Any energy the programmer devotes to navigating this space would be better spent in other ways. The tedious job of finding a safe sequence of commands that implement an update should be factored out, optimized, and reused across many applications. This is the main achievement of this chapter. To implement the update using our abstractions, the programmer would simply write:

```
per_packet_update(config2)
```

Here `config2` represents the new global network configuration. The per-packet update library analyzes the configuration and topology and selects a suitable mechanism to implement the update. Note that the programmer does not write any tricky code, does not need to consider how to synchronize switch update commands, and does not need to consider the packets in flight across the network. The `per_packet_update` library handles all of the low-level details, and even attempts to select a mechanism that minimizes the cost of implementing the update.

To implement the update, the library could use the safe, switch-update ordering described above. However, in general, it is not always possible to find such an ordering. Nevertheless, one can always achieve a per-packet consistent update using a two-phase update supported by configuration versioning. Intuitively, this universal update mechanism works by stamping every incoming packet with a version number (*e.g.*, stored in a VLAN tag) and modifying every configuration so that it only processes packets with a set version number. To change from one configuration to the next, it first populates the switches in the middle of the network with new configurations guarded by the next version number. Once that is complete, it enables the new configurations by installing rules at the perimeter of the network that stamp packets with that next version number. Though this general mechanism is somewhat heavyweight, our libraries identify and apply lightweight optimizations.

This short example illustrates some of the challenges that arise when implementing a network update with strong semantic guarantees. However, it also shows that all of these complexities can be hidden from the programmer, leaving only the simplest of interfaces for global network update. We believe this simplicity will lead to a more reliable and secure network infrastructure. The following sections describe our approach in more detail.

## 5.3 The Network Model

This section presents a simple mathematical model of the essential features of SDNs. This model is defined by a relation that describes the fine-grained, step-by-step execution of a

|  |  |
|---|---|
| Bit | $b ::= 0 \mid 1$ |
| Packet | $pk ::= [b_1, \ldots, b_k]$ |
| Port | $p ::= 1 \mid \cdots \mid k \mid \mathit{Drop} \mid \mathit{World}$ |
| Located Pkt | $lp ::= (p, pk)$ |
| Trace | $t ::= [lp_1, \ldots, lp_n]$ |
| Update | $u \in \mathit{LocatedPkt} \rightharpoonup \mathit{LocatedPkt\ list}$ |
| Switch Func. | $S \in \mathit{LocatedPkt} \to \mathit{LocatedPkt\ list}$ |
| Topology Func. | $T \in \mathit{Port} \to \mathit{Port}$ |
| Port Queue | $Q \in \mathit{Port} \to (\mathit{Packet} \times \mathit{Trace})\ \mathit{list}$ |
| Configuration | $C ::= (S, T)$ |
| Network State | $N ::= (Q, C)$ |

(a)

$\boxed{\text{T-PROCESS}}$

if $p$ is any port (1)
and $Q(p) = [(pk_1, t_1), (pk_2, t_2), \ldots, (pk_j, t_j)]$ (2)
and $C = (S, T)$ (3)
and $S(p, pk_1) = [(p'_1, pk'_1), \ldots, (p'_k, pk'_k)]$ (4)
and $T(p'_i) = p''_i$, for $i$ from 1 to $k$ (5)
and $t'_1 = t_1 +\!\!+ [(p, pk_1)]$ (6)
and $Q'_0 = \mathit{override}(Q, p \mapsto [(pk_2, t_2), \ldots, (pk_j, t_j)])$ (7)
and $Q'_1 = \mathit{override}(Q'_0, p''_1 \mapsto Q(p''_1) +\!\!+ [(pk'_1, t'_1)])$

$\vdots$

and $Q'_k = \mathit{override}(Q'_{k-1}, p''_k \mapsto Q(p''_k) +\!\!+ [(pk'_k, t'_1)])$
then $(Q, C) \longrightarrow (Q'_k, C)$ (8)

$\boxed{\text{T-UPDATE}}$

if $S' = \mathit{override}(S, u)$ (9)
then $(Q, (S, T)) \xrightarrow{u} (Q, (S', T))$ (10)

(b)

Figure 5.2: The network model: (a) syntax and (b) semantics.

network. We write the relation using the notation $N \xrightarrow{us}{}^{\star} N'$, where $N$ is the network at the beginning of an execution, $N'$ is the network after some number of steps of execution, and *us* is a list of "observations" that are made during the execution.[1] Intuitively, an observation should be thought of as a message between the controller and the network. In this chapter, we are interested in a single kind of message—a message $u$ that directs a particular switch in the network to update its forwarding table with some new rules. The formal system could easily be augmented with other kinds of observations, such as topology changes or failures. For the sake of brevity, we elide these features in this chapter.

The main purpose of the model is to compute the *traces*, or paths, that a packet takes through a network that is configured in a particular way. These traces in turn define the properties, be they access control or connectivity or others, that a network configuration satisfies. Our end goal is to use this model and the traces it generates to prove that, when we update a network, the properties satisfied by the initial and final configurations are preserved. The rest of this section will make these ideas precise.

**Notation** We use standard notation for types. In particular, the type $T_1 \rightarrow T_2$ denotes the set of total functions that take arguments of type $T_1$ and produce results of type $T_2$, while $T_1 \rightharpoonup T_2$ denotes the set of partial functions from $T_1$ to $T_2$; the type $T_1 \times T_2$ denotes the set of pairs with components of type $T_1$ and $T_2$; and $T$ *list* denotes the set of lists with elements of type $T$.

We also use standard notation to construct tuples: $(x_1, x_2)$ is a pair of items $x_1$ and $x_2$. For lists, we use the notation $[x_1, \ldots, x_n]$ for the list of $n$ elements $x_1$ through $x_n$, $[\,]$ for the empty list, and $xs_1 +\!\!+\ xs_2$ for the concatenation of the two lists $xs_1$ and $xs_2$. Notice that

[1]When a network takes a series of steps and there are no observations (*i.e.,* no updates happen), we omit the list above the arrow, writing $N \longrightarrow^{\star} N'$ instead.

if $x$ is some sort of object, we will typically use $xs$ as the variable for a list of such objects. For example, we use $u$ to represent a single update and $us$ to represent a list of updates.

**Basic Structures** Figure 5.2(a) defines the syntax of the elements of the network model. A *packet* $pk$ is a sequence of bits, where a *bit* $b$ is either 0 or 1. A *port* $p$ represents a location in the network where packets may be waiting to be processed. We distinguish two kinds of ports: ordinary ports numbered uniquely from 1 to $k$, which correspond to the physical input and output ports on switches, and two special ports, *Drop* and *World*. Intuitively, packets queued at the *Drop* port represent packets that have been dropped, while packets queued at the *World* port represent packets that have been forwarded beyond the confines of the network. Each ordinary port will be located on some switch in the network. However, we will leave the mapping from ports to switches unspecified, as it is not needed for our primary analyses.

**Switch and Topology Functions** A network is a packet processor that forwards packets and optionally modifies the contents of those packets on each hop. Following Kazemian *et al.* [50], we model packet processing as the composition of two simpler behaviors: (1) forwarding a packet across a switch and (2) moving packets from one end of a link to the other end. The *switch function* $S$ takes a located packet $lp$ (a pair of a packet and a port) as input and returns a list of located packets as a result. In many applications, a switch function only produces a single located packet, but in applications such as multicast, it may produce several. To drop a packet, a switch function maps the packet to the special *Drop* port. The *topology function* $T$ maps one port to another if the two ports are connected by a link in the network. Given a topology function $T$, we define an ordinary port $p$ to be an *ingress port* if for all other ordinary ports $p'$ we have $T(p') \neq p$. Similarly, we define an ordinary port $p$ to be an *internal port* if it is not an ingress port.

To ensure that switch and topology functions are reasonable, we impose the following conditions:

(1) For all packets $pk$, $S(\mathit{Drop}, pk) = [(\mathit{Drop}, pk)]$ and
$S(\mathit{World}, pk) = [(\mathit{World}, pk)]$;

(2) $T(\mathit{Drop}) = \mathit{Drop}$ and $T(\mathit{World}) = \mathit{World}$; and

(3) For all ports $p$ and packets $pk$
if $S(p, pk) = [(p_1, pk_1), \ldots, (p_k, pk_k)]$ we have $k \geq 1$.

Taken together, the first and second conditions state that once a packet is dropped or forwarded beyond the perimeter of the network, it must stay dropped or beyond the perimeter of the network and never return. If our network forwards a packet out to another network and that other network forwards the packet back to us, we treat the return packet as a "fresh" packet—*i.e.*, we do not explicitly model inter-domain forwarding. The third condition states that applying the forwarding function to a port and a packet must produce at least one packet. This third condition means that the network cannot drop a packet simply by not forwarding it anywhere. Dropping packets occurs by explicitly forwarding a single packet to the *Drop* port. This feature makes it possible to state network properties that require packets either be dropped or not.

**Configurations and Network States** A *trace* $t$ is a list of located packets that keeps track of the hops that a packet takes as it traverses the network. A *port queue* $Q$ is a total function from ports to lists of packet-trace pairs. These port queues record the packets waiting to be processed at each port in the network, along with the full processing history of that packet. Several of our definitions require modifying the state of a port queue. We do this by building a new function that overrides the old queue with a new mapping for one of its ports: $\mathit{override}(Q, p \mapsto l)$ produces a new port queue $Q'$ that maps $p$ to $l$ and like $Q$ otherwise.

$$\begin{aligned} &\mathit{override}(Q, p \mapsto l) = Q' \\ &\text{where } Q'(p') = \begin{cases} l & \text{if } p = p' \\ Q(p') & \text{otherwise} \end{cases} \end{aligned}$$

A *configuration* $C$ comprises a switch function $S$ and a topology function $T$. A *network state* $N$ is a pair $(Q, C)$ containing a port queue $Q$ and configuration $C$.

**Transitions** The formal definition of the network semantics is given by the relations defined in Figure 5.2(b), which describe how the network transitions from one state to the next one. The system has two kinds of transitions: packet-processing transitions and update transitions. In a packet-processing transition, a packet is retrieved from the queue for some port, processed using the switch function $S$ and topology function $T$, and the newly generated packets are enqueued onto the appropriate port queues. More formally, packet-processing transitions are defined by the T-Process case in Figure 5.2(b). Lines 1-8 may be read roughly as follows:

(1) If $p$ is any port,

(2) a list of packets is waiting on $p$,

(3) the configuration $C$ is a pair of a switch function $S$ and topology function $T$,

(4) the switch function $S$ forwards the chosen packet to a single output port, or several ports in the case of multicast, and possibly modifies the packet

(5) the topology function $T$ connects each output port to an input port,

(6) a new trace $t'_1$, which extends the old trace and records the current hop, is generated,

(7) a new queue $Q'_k$ is generated by moving packets across links as specified in steps (4), (5) and (6),

(8) then $(Q, C)$ can step to $(Q'_k, C)$.

In an update transition, the switch forwarding function is updated with new behavior. We represent an *update* $u$ as a partial function from located packets to lists of located packets (*i.e.*, an update is just a "part" of a global (distributed) switch function). To apply an update to a switch function, we overwrite the function using all of the mappings contained in the update. More formally, $\mathit{override}(S, u)$ produces a new function $S'$ that behaves like $u$ on located packets in the domain[2] of $u$, and like $S$ otherwise.

$$\begin{aligned} &\mathit{override}(S, u) = S' \\ &\text{where } S'(p, pk) = \begin{cases} u(p, pk) & \text{if } (p, pk) \in \mathsf{dom}(u) \\ S(p, pk) & \text{otherwise} \end{cases} \end{aligned}$$

Update transitions are defined formally by the T-UPDATE case in Figure 5.2(b). Lines 9-10 may be read as follows: if $S'$ is obtained by applying update $u$ to a switch in the network then network state $(Q, (S, T))$ can step to new network state $(Q, (S', T))$.

**Network Semantics** The overall semantics of a network in our model is defined by allowing the system to take an arbitrary number of steps starting from an *initial state* in which the queues of all internal ports as well as *World* and *Drop* are empty, and the queues of external ports are filled with pairs of packets and the empty trace. The reflexive and transitive closure of the single-step transition relation $N \xrightarrow{us}{}^{\star} N'$ is defined in the usual way, where the sequence of updates recorded in the label above the arrow is obtained by concatenating

[2]Domain of an update is the set of located packets it's defined upon.

all of the updates in the underlying transitions in order.[3] A network *generates* a trace $t$ if and only if there exists an initial state $Q$ such that $(Q, C) \longrightarrow^{\star} (Q', C)$ and $t$ appears in $Q'$. Note that no updates may occur when generating a trace.

**Properties** In general, there are myriad properties a network might satisfy—*e.g.*, access control, connectivity, in-order delivery, quality of service, fault tolerance, to name a few. In this chapter, we will primarily be interested in *trace properties*, which are prefix-closed sets of traces. Trace properties characterize the paths (and the state of the packet at each hop) that an individual packet is allowed to take through the network. Many network properties, including access control, connectivity, routing correctness, loop-freedom, correct VLAN tagging, and waypointing can be expressed using trace properties. For example, loop-freedom can be specified using a set that contain all traces except those in which some ordinary port $p$ appears twice. In contrast, timing properties and relations between multiple packets including quality of service, congestion control, in-order delivery, or flow affinity are not trace properties.

We say that a port queue $Q$ satisfies a trace property $P$ if all of the traces that appear in $Q$ also appear in the set $P$. Similarly, we say that a network configuration $C$ satisfies a trace property $P$ if for all *initial* port queues $Q$ and all (update-free) executions $(Q, C) \longrightarrow^{\star} (Q', C)$, it is the case that $Q'$ satisfies $P$.

[3]The semantics of the network is defined from the perspective of an omniscient observer, so there is an order in which the steps occur.

## 5.4 Per-Packet Abstraction

One reason that network updates are difficult to get right is that they are a form of concurrent programming. Concurrent programming is hard because programmers must consider the interleaving of every operation in every thread and this leads to a combinatorial explosion of possible outcomes—too many outcomes for most programmers to manage. Likewise, when performing a network update, a programmer must consider the interleaving of switch update operations with every kind of packet that might be traversing their network. Again, the number of possibilities explodes.

Per-packet consistent updates reduce the number of scenarios a programmer must consider to just two: for every packet, it is as if the packet flows through the network *completely before* the update occurs, or *completely after* the update occurs.

One might be tempted to think of per-packet consistent updates as “atomic updates”, but they are actually better than that. An atomic update would cause packets in flight to be processed partly according to the configuration in place prior to the update, and partly according to the configuration in place after the update. To understand what happens to those packets (*e.g.*, whether they get dropped), a programmer would have to reason about every possible trace formed by concatenating a prefix generated by the original configuration with a suffix generated by the new configuration.

Intuitively, per-packet consistency states that for a given packet, the traces generated during an update come from the old configurations, or the new configuration, but not a mixture of the two. In the formal definition of per-packet consistency, we introduce an equivalence relation $\sim$ on packets. We extend this equivalence relation to traces by considering two traces to be equivalent if the packets they contain are equivalent according to the $\sim$ relation (similarly, we extend $\sim$ to properties in the obvious way). We then require that all traces generated *during* the update be equivalent to a trace generated by either the initial or final configuration. For the rest of the chapter, when we say that $\sim$ is an equivalence relation on traces, we assume that it has been constructed like this. This specification gives implementations of updates flexibility by allowing some minor, irrelevant differences to appear in traces (where $\sim$ defines the notion of irrelevance precisely). For example, we can define a "version" equivalence relation that relates packets *pk* and *pk'* which differ only in the value of their version tags. This relation will allow us to state that changes to version tags performed by the implementation mechanism for per-packet update are irrelevant. In other words, a per-packet mechanism may perform internal bookkeeping by stamping version tags without violating our technical requirements on the correctness of the mechanism. The precise definition of per-packet consistency is as follows.

**Definition 4** (Per-packet $\sim$-consistent update). *Let $\sim$ be a trace-equivalence relation. An update sequence us is a per-packet $\sim$-consistent update from $C_1$ to $C_2$ if and only if, for all*

- *initial states $Q$,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and traces $t$ in $Q'$,*

*there exists*

- *an initial state $Q_i$,*
- *and either an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$ or an execution $(Q_i, C_2) \longrightarrow^{\star} (Q'', C_2)$,*

*such that $Q''$ contains $t'$, for some trace $t'$ with $t' \sim t$.*

From an implementer's perspective, the operational definition of per-packet consistency given above provides a specification that he or she must meet. However, from a programmer's perspective, there is another, more useful side to per-packet consistency: *per-packet consistent updates preserve every trace property.*

**Definition 5** ($\sim$-property preservation)**.** *Let $C_1$ and $C_2$ be configurations and $\sim$ be a trace-equivalence relation. A sequence us is a $\sim$-property-preserving update from $C_1$ and $C_2$ if and only if, for all*

- *initial states $Q$,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and properties $P$ that are satisfied by $C_1$ and $C_2$ and do not distinguish traces related by $\sim$,*

*we have that $Q'$ satisfies $P$.*

Universal $\sim$-property preservation gives programmers a strong principle they can use to reason about their programs. If programmers check that a trace property such as loop-freedom or access control holds of the network configurations before and after an update, they are guaranteed it holds of every trace generated throughout the update process, even though the series of observations *us* may contain many discrete update steps. Our main theorem states that per-packet consistent updates preserve all properties:

**Theorem 1.** *For all trace-equivalence relations $\sim$, if us is a per-packet $\sim$-consistent update of $C_1$ to $C_2$ then us is a $\sim$-property-preserving update of $C_1$ to $C_2$.*

The proof of the theorem is a relatively straightforward application of our definitions. From a practical perspective, this theorem allows a programmer to get great mileage out of per-packet consistent updates. In particular, since per-packet consistent updates preserve *all* trace properties, the programmers do not have to tell the system *which* specific properties must be invariant in their applications.

From a theoretical perspective, it is also interesting that the converse of the above theorem holds. This gives us a sort of completeness result: if programmers want an update that preserves all properties, they need not search for it outside of the space of per-packet consistent updates—any universal trace-property preserving update is a per-packet consistent update.

**Theorem 2.** *For all trace-equivalence relations $\sim$, if us is a $\sim$-property-preserving update of $C_1$ to $C_2$ then us is a per-packet $\sim$-consistent update of $C_1$ to $C_2$.*

The proof of this theorem proceeds by observing that since *us* preserves all $\sim$-properties, it certainly preserves the following $\sim$-property $P_{\mathsf{or}}$:

$$\begin{aligned}\{t\ |&\text{there exists an initial } Q \text{ and a trace } t'\\ &\text{and } ((Q, C_1) \longrightarrow^{\star} (Q', C_1) \text{ or } (Q, C_2) \longrightarrow^{\star} (Q', C_2)),\\ &\text{and } t \sim t',\\ &\text{and } t' \in Q'\}\end{aligned}$$

By the definition of $P_{\mathsf{or}}$, the update *us* generates no traces that cannot be generated either by the initial configuration $C_1$ or by the final configuration $C_2$. Hence, by definition, *us* is per-packet consistent.

**Formal proof** The network model, and all of the above theorems have been formally specified and proven in the Coq theorem prover.

## 5.5 Per-packet Mechanisms

Depending on the network topology and the specifics of the configurations involved, there may be several ways to implement a per-packet consistent update. However, all of the techniques we have discovered so far, no matter how complicated, can be reduced to two fundamental building blocks: the one-touch update and the unobservable update. For example, our two-phase update mechanism uses unobservable updates to install the new configuration before it is used, and then "unlocks" the new policy by performing a one-touch update on the ingress ports.

**One-touch updates** A one-touch update is an update with the property that no packet can follow a path through the network that reaches an updated (or to-be-updated) part of the switch rule space more than once.

**Definition 6** (One-touch Update). *Let $C_1 = (FT, T)$ be the original network configuration, $us = [u_1, \ldots, u_k]$ an update sequence, and $C_2 = (FT[u_1, \ldots, u_k], T)$ the new configuration, such that the domains of each update $u_1$ to $u_k$ are mutually disjoint. If, for all*

- *initial states $Q$,*
- *and executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*

*there does not exist a trace $t$ in $Q'$ such that*

- *$t$ contains distinct trace elements $(p_1, pk_1)$ and $(p_2, pk_2)$,*
- *and $(p_1, pk_1)$ and $(p_2, pk_2)$ both appear in the domain of update functions $[u_1, \ldots, u_k]$,*

*then $us$ is a one-touch update from $C_1$ to $C_2$.*

**Theorem 9.** *If us is a one-touch update then us is a $\sim$-per-packet consistent update for any $\sim$.*

The proof proceeds by considering the possible traces $t$ generated by an execution $(Q, C_1) \xrightarrow{us}{}^{\star}(Q', C_2)$. There are two cases: (1) There is no element of $t$ that appears in the domain of an update function in $us$, or (2) some element $lp$ of $t$ appears in the domain of an update function in $us$. In case (1), $t$ can also be generated by an execution with no update observations: $(Q, C_1) \longrightarrow^{\star}(Q'', C_1)$, and the definition of per-packet consistency vacuously holds. In case (2), there are two subcases:

(i) $lp$ appears in the trace prior to the update taking place and so $t$ is also generated by $(Q, C_1) \longrightarrow^{\star}(Q'', C_1)$.

(ii) $lp$ appears in the trace after the update has taken place and so $t$ is also generated by $(Q, C_2) \longrightarrow^{\star}(Q'', C_2)$.

The one-touch update mechanism has a few immediate, more specific applications:

- *Loop-free switch updates:* If a switch is not part of a topological loop (either before or after the update), then updating all the ports on that switch is an instance of a one-touch update and is per-packet consistent.
- *Ingress port updates:* An ingress port interfaces exclusively with the external world, so it can not be a part of an internal topological loop and is never on the same trace as any other ingress port. Consequently, any update to ingress ports is a one-touch update and is per-packet consistent. Such updates can be used to change the admission control policy for the network, either by adding or excluding flows.

When one-touch updates are combined with unobservable updates, there are many more possibilities.

**Unobservable updates** An unobservable update is an update that does not change the set of traces generated by a network.

**Definition 7** (Unobservable Update). *Let $C_1 = (FT, T)$ be the original network configuration, $us = [u_1, \ldots, u_k]$ an update sequence, and $C_2 = (FT[u_1, \ldots, u_k], T)$ the new configuration. If, for all*

- *initial states $Q$,*
- *executions $(Q, C_1) \xrightarrow{us}^{\star} (Q', C_2)$,*
- *and traces $t$ in $Q'$,*

*there exists*

- *an initial state $Q_i$,*
- *and an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$,*

*such that the trace $t$ is in $Q''$, then $us$ is an unobservable update from $C_1$ to $C_2$.*

**Theorem 3.** *If $us$ is an unobservable update then $us$ is a per-packet consistent update.*

The proof proceeds by observing that every trace generated during the unobservable update $(Q, C_1) \xrightarrow{us}^{\star} (Q', C_2)$ also appears in the traces generated by $C_1$.

On their own, unobservable updates are useless as they do not change the semantics of packet forwarding. However, they may be combined with other per-packet consistent updates to great effect using the following theorem.

**Theorem 10** (Composition). *If $us_1$ is an unobservable update from $C_1$ to $C_2$ and $us_2$ is a per-packet consistent update from $C_2$ to $C_3$ then $us_1 \mathbin{++} us_2$ is a per-packet consistent update from $C_1$ to $C_3$.*

A simple use of composition arises when one wants to achieve a per-packet consistent update that extends a policy with a completely new path.

- *Path extension:* Consider an initial configuration $C_1$. Suppose $[u_1, u_2, \ldots, u_k]$ updates ports $p_1, p_2, \ldots, p_k$ respectively to lay down a new path through the network with $u_1$ updating the ingress port. Suppose also that the ports updated by $us = [u_2, \ldots, u_k]$ are unreachable in network configuration $C_1$. Hence, $us$ is an unobservable update. Since $[u_1]$ updates an ingress port, it is a one-touch update and also per-packet consistent. By the composition principle, $us \mathbin{++} [u_1]$ is a per-packet consistent update.

Notice that the path update is achieved by first laying down rules on switches 2 to $k$ and then, when that is complete, laying down the rules on switch 1. A well-known (but still common!) bug occurs when programmers attempt to install new forwarding paths but lay down the elements of the path in wrong order [14]. Typically, there is a race condition in which packets traverse the first link and reach the switch 2 before the program has had time to lay down the rules on links 2 to $k$. Then when packets reach switch 2, it does not yet know how to handle them, and a default rule sends the packets to the controller. The controller often becomes confused as it begins to see additional packets that should have already been dealt with by laying down the new rules. The underlying cause of this bug is explained with our model—the programmer intended a per-packet consistent update of the policy with a new path, but failed to implement per-packet consistency correctly. All such bugs are eradicated from network programs if programmers use per-packet consistent updates and never use their own ad hoc update mechanisms.

**Two-phase update** So far, all of our update mechanisms have applied to special cases in which the topology, existing configuration, and/or updates have specific properties. Fortunately, provided there are a few bits in packets that are irrelevant to the network properties a programmer wishes to enforce, and can be used for bookkeeping purposes, we can define a mechanism that handles arbitrary updates using a two-phase update protocol.

Intuitively, the two-phase update works by first installing the new configuration on internal ports, but only enabling the new configuration for packets containing the correct version number. It then updates the ingress ports one-by-one to stamp packets with the new version number. Notice that the updates in the first phase are all unobservable, since before the update, the ingress ports do not stamp packets with the new version number. Hence, since updating ingress ports is per-packet consistent, by the composition principle, the two-phase update is also per-packet consistent.

To define the two-phase update formally, we need a few additional definitions. Let a version-property be a trace property that does not distinguish traces based on the value of version tags. A configuration $C$ is a version-$n$ configuration if $C = (S, T)$ and $S$ modifies packets processed by any ingress port $p_{in}$ so that after passing through $p_{in}$, the packet's version bit is $n$. We assume that the $S$ function does not otherwise modify the version bit of the packet. Two configurations $C$ and $C'$ coincide internally on version-$n$ packets whenever $C = (S, T)$ and $C' = (S', T')$ and for all internal ports $p$, and for all packets $pk$ with version bit set to $n$, we have that $FT(p, pk) = FT'(p, pk)$. Finally, an update $u$ is a refinement of $S$, if for all located packets $lp$ in the domain of $u$, we have that $u(lp) = S(lp)$.

**Definition 8** (Two-phase Update). *Let $C_1 = (S, T)$ be a version-1 configuration and $C_2 =$*

*$(S', T)$ be a version-2 configuration. Assume that $C_1$ and $C_2$ coincide internally on version-1 packets. Let $us = [u^i_1, \ldots, u^i_m, u^e_1, \ldots, u^e_n]$ be an update sequence such that*

- *$S' = override(S, us)$,*
- *each $u^i_j$ and $u^e_k$ is a refinement of $S'$,*
- *$p$ is internal, for each $(p, pk)$ in the domain of $u^i_j$,*
- *and $p$ is an ingress, for each $(p, pk)$ in the domain of $u^e_k$.*

*Then $us$ is a two-phase update from $C_1$ to $C_2$.*

**Theorem 11.** *If $us$ is a two-phase update then $us$ is per-packet consistent.*

The proof simply observes that $us_1 = [u^i_1, \ldots, u^i_m]$ is an unobservable update, and $us_2 = [u^e_1, \ldots, u^e_n]$ is a one-touch update (and therefore per-packet consistent). Hence, by composition, the two-phase update $us_1 +\!\!+\ us_2$ is per-packet consistent.

**Optimized mechanisms** Ideally, update mechanisms should satisfy *update proportionality*, where the cost of installing a new configuration should be proportional to the size of the configuration change. A perfectly proportional update would (un)install just the "delta" between the two configurations. The full two-phase update mechanism that installs the full new policy and then uninstalls the old policy lacks update proportionality. In this section, we describe optimizations that substantially reduce overhead.

Pure extensions and retractions are one important case of updates where a per-packet mechanism achieves perfect proportionality. A pure extension is an update that adds new paths to the current configuration that cannot be reached in the old configuration—*e.g.*, adding a forwarding path for a new host that comes online. Such updates do not require a complete two-phase update, as only the new forwarding rules need to be installed—first at the internal ports and then at the ingresses. The rules are installed using the current version number. A *pure retraction* is the dual of a pure extension in which some paths are removed from the configuration. Again, the paths being removed must be unreachable in the new configuration. Pure retractions can be implemented by updating the ingresses, pausing to wait until packets in flight drain out of the network, and then updating the internal ports.

If paths are not only added or removed but are modified then more powerful optimizations are available. Per-packet consistency requires that the active paths in the network come from either of the configurations. The subset mechanism works by identifying the paths that have been added, removed or changed and then updating the rules along the entire path to use a new version. This optimization is always applicable, but in the degenerate case it devolves into a network-wide two-phase update.

## 5.6 Per-flow Consistency

Per-packet consistency, while simple and powerful, is not always enough. Some applications require a stream of related packets to be handled consistently. For example, a server load-balancer needs all packets from the same TCP connection to reach the same server replica. In this section, we introduce the per-flow consistency abstraction, and discuss mechanisms for per-flow consistent updates.

**Per-flow abstraction** To see the need for per-flow consistency, consider a network where a single switch $\mathsf{S}$ load-balances between two back-end servers $A$ and $B$. Initially, $\mathsf{S}$ directs traffic from IP addresses starting with 0 (*i.e.*, source addresses in 0.0.0.0/1) to $A$ and 1 (*i.e.*, source addresses in 128.0.0.0/1) to $B$. At some time later, we bring two additional servers $C$

and $D$ online, and re-balance the load using a two-bit prefix, directing traffic from addresses starting with 00 to $A$, 01 to $B$, 10 to $C$, and 11 to $D$.

Intuitively, we want to process packets from new TCP connections according to the new configuration. However, all packets in existing flows must go to the same server, where a *flow* is a sequence of packets with related header fields, entering the network at the same port, and not separated by more than $n$ seconds. The particular value of $n$ depends upon the protocol and application. For example, the switch should send packets from a host whose address starts with "11" to $B$, and not to $D$ as the new configuration would dictate, if the packets belong to an ongoing TCP connection. Simply processing individual packets with a single configuration does not guarantee the desired behavior.

*Per-flow consistency* guarantees that all packets in the same flow are handled by the same version of the configuration. Formally, the per-flow abstraction preserves all path properties, as well as all properties that can be expressed in terms of the paths traversed by *sets* of packets belonging to the same flow.

**Per-flow mechanisms** Implementing per-flow consistent updates is much more complicated than per-packet consistency because the system must identify packets that belong to active flows. Below, we discuss three different mechanisms. Our system implements the first of the three; the latter two, while promising, depend upon technology that is not yet available in OpenFlow.

**Switch rules with timeouts:** A simple mechanism can be obtained by combining versioning with rule timeouts, similar to the approach in [111]. The idea is to pre-install the new configuration on the internal switches, leaving the old version in place, as in per-packet consistency. Then, on ingress switches, the controller sets soft timeouts on the rules for the old configuration and installs the new configuration at lower priority. When all flows matching a given rule finish, the rule automatically expires and the rules for the new configuration take effect. When multiple flows match the same rule, the rule may be artificially kept alive even though the "old" flows have all completed. If the rules are too coarse, then they may never die! To ensure rules expire in a timely fashion, the controller can refine the old rules to cover a progressively smaller portion of the flow space. However, "finer" rules require more rules, a potentially scarce commodity. Managing the rules and dynamically refining them over time can be a complex bookkeeping task, especially if the network undergoes a *subsequent* configuration change before the previous one completes. However, this task can be implemented and optimized once in a run-time system, and leveraged over and over again in different applications.

**Wildcard cloning:** An alternative mechanism exploits the wildcard *clone* feature of the DevoFlow extension of OpenFlow [76]. When processing a packet with a *clone* rule, a DevoFlow switch creates a new "microflow" rule that matches the packet header fields exactly. In effect, clone rules cause the switch to maintain a concrete representation of each active flow. This enables a simple update mechanism: first, use clone rules whenever installing configurations; second, to update from old to new, simply replace all old clone rules with the new configuration. Existing flows will continue to be handled by the exact-match rules previously generated by the old clone rules, and new flows will be handled by the new clone rules, which themselves immediately spawn new microflow rules. While this mechanism does not require complicated bookkeeping on the controller, it does require a more complex switch.

**End-host feedback:** The third alternative exploits information readily available on the end hosts, such as servers in a data center. With a small extension, these servers could provide a list of active sockets (identified by the "five tuple" of IP addresses, TCP/UDP ports, and protocol) to the controller. As part of performing an update, the controller would query the local hosts and install high-priority microflow rules that direct each active flow to the assigned server replica. These rules could “timeout” after a period of inactivity, allowing future traffic to “fall through” to the new configuration. Alternatively, the controller could install “permanent” microflow rules, and explicitly remove them when the socket no long exists on the host, obviating the need for any assumptions about the minimum interval time between packets of the same connection.

## 5.7 Update Mechanisms

Ideally, update mechanisms should satisfy *update proportionality*, the where the cost of installing a new configuration should be proportional to the size of the configuration change. A perfectly proportional update would (un)install just the “delta” between the two configurations. The full two-phase update mechanism that installs the full new policy and then uninstalls the old policy lacks update proportionality. In this section, we describe optimizations that substantially reduce overhead.

Pure extensions and retractions are one important case of updates where a per-packet mechanism achieves perfect proportionality. A pure extension is an update that adds new paths to the current configuration that cannot be reached in the old configuration—*e.g.*, adding a forwarding path for a new host that comes online. Such updates do not require a complete two-phase update, as only the new forwarding rules need to be installed—first at the internal ports and then at the ingresses. The rules are installed using the current version number. A *pure retraction* is the dual of a pure extension in which some paths are removed from the configuration. Again, the paths being removed must be unreachable in the new configuration. Pure retractions can be implemented by updating the ingresses, pausing to wait until packets in flight drain out of the network, and then updating the internal ports.

If paths are not only added or removed but are modified then more powerful optimizations than pure extension/retraction are available. Per-packet consistency requires that the active paths in the network come from either of the configurations. There are two different ways to perform an update that maintains this promise. Either you identify the paths that have been added, removed, or changed and you update the entire path to use a new version, or you identify the switches that have changed and you update the entire switch to use a new version. We call the former mechanism the "subset" mechanism and the latter the "island" mechanism. Both of these mechanisms are always safe to apply but require analysis on the configurations. In the degenerate case, these mechanisms devolve to a network-wide two-phase update for all traffic.

**Subset Mechanism** The *subset* mechanism calculates the precise set of forwarding paths affected by an update and only updates the portion of the configuration that implements those paths (using a standard two-phase update). In situations where the set of paths is small compared to the size of the overall configuration, the cost of a subset update is less than a full two-phase update. Unlike pure extensions and retractions, which do not handle cases where existing forwarding paths are modified or where the affected rules are reachable in the new configuration, this mechanism can always be safely applied; in the case where every rule is affected by the update, it simply degenerates to a two-phase update.

The subset mechanism is implemented by computing the set of rules that changed in the new configuration and computing the closure of that set under a certain connectivity relation. We say that rules $r_1$, $r_2$ are connected under $C = (S, T)$ if there are packets $pk$, $pk'$ and ports $p, p'$ such that $r_1((p, pk)) = [..., (p', pk'), ...]$ and $(T(p'), pk') \in \text{dom}(r_2)$. Write

$r_1 \leftarrow C \rightarrow r_2$ if $r_1$ is connected to $r_2$ under $C$ or vice versa.

**Definition 9** (Subset Update)**.** *Let $C = (S, T)$ be a version-*1 *configuration and $C' = (S', T)$ be a version-*2 *configuration. Let $mods_0 = S' - S$, and let mods be the closure of $mods_0$ under the relation $\bullet \leftarrow C' \rightarrow \bullet$. Then mods is a* subset *update from $C$ to $C'$.*

**Theorem 12.** *A subset update from $C = (S, T)$ to $C' = (S', T)$ is a per-packet consistent update from $C$ to $C'$.*

*Proof Sketch:* First show that if a set of rules is closed under the connectivity relation, then there is a well-defined set of traces generated by that set of rules, and that set of traces is equivalent to running the whole configuration over packets in the domain of the set. Then show that $S[mods] = S'$.

**Island Mechanism** The key idea behind the *island* mechanism is to identify a small connected component of the network containing the switches whose configurations changed. We update this "island" of switches to use the new configuration with a new version number. The configurations use a new version when packets are sent between switches in the island and restores the old version when packets leave the island. If more than two versions are active in the network at a time then versions should be implemented with a mechanism that supports a stack of versions. MPLS labels, available in OpenFlow 1.1 are a candidate.

The computation of an *island* update uses a closure computation similar to the *subset* mechanism, except that the relation is over ports instead of rules. Say that two ports $p, p''$ are connected under $C = (S, T)$ via a third port $p'$ if there exists a sequence of rules $r_1, \ldots, r_k, \ldots r_n$ such that $r_1 \in S[p]$, $r_k \in S[p']$, $r_n \in S[p'']$ and for $0 < i < n$, $r_i \leftarrow C \rightarrow r_{i+1}$. Write $p \leftarrow C, p' \rightarrow p''$ if $p, p''$ are connected under $C$ via $p'$.

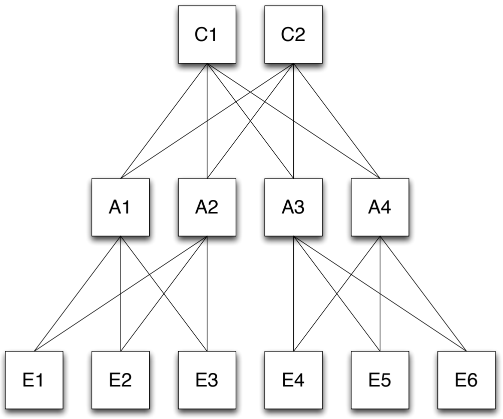


Figure 5.3: Fat tree topology

**Definition 10** (Island Update)**.** *Let $C = (S, T)$ be a version-1 configuration and $C' = (S', T)$ be a version-2 configuration. Let $mods_0 = \{p \mid p \in S' - S\}$. Let mods be the least fixpoint of:*

$$mods = \{p \mid p \in mods_0\} \cup \{p' \mid \exists p', p'' \in mods.\ p' \leftarrow C', p \rightarrow p''\}$$

*Then mods is an* island *update from $C$ to $C'$.*

### 5.7.1 Case Study

To highlight the uses and distinctions between the mechanisms, we work through case studies of networks in a fat tree and a small-world topology.

We show a simple fat tree topology and a snippet of the routing configuration in Figure 5.3. The topology consists of edge switches E1-E6 directly connected to hosts on per-switch subnets, aggregation switches A1-A4, and core switches C1-C2 providing connectivity be-

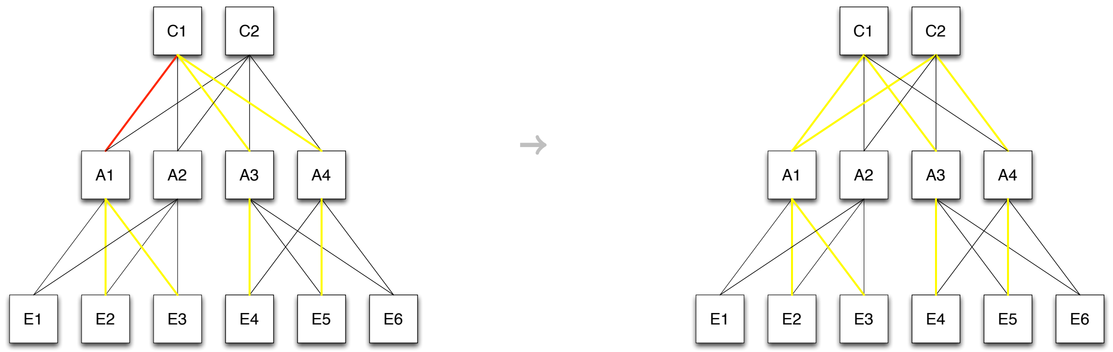


Figure 5.4: Network before and after load balancing

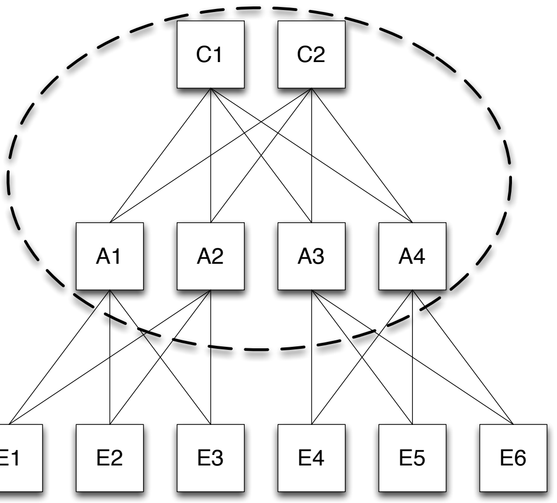


Figure 5.5: Island calculated for maintenance update

tween the two sides of the tree. The controller program runs a simple shortest path routing algorithm and monitors the network to perform load balancing. In addition, the controller takes certain switches down at predetermined times for scheduled maintenance.

**Dynamic Host** When a host comes on or offline at an edge switch, the routes for all the other source-destination pairs remains unchanged. Consider a scenario in which a new host H0 comes online at E1. Routes to and from H0 are installed at each switch, but existing rules and traffic are unaffected. Using a full two-phase update requires updating the configuration of every other switch, a gross violation of proportionality. Instead, the *extension* mechanism installs just the new forwarding rules at the current version number and leaves the existing rules untouched.

**Network Load Balancing** Initially, traffic between the two halves of the network is statically split evenly between C1 and C2, but the dynamic traffic patterns may make this static split unbalanced. Consider what happens if two pairs of hosts start sending heavy traffic flows over C1's link to A1, overloading it as shown in Figure 5.4. The controller program moves to a new configuration that puts the traffic from one of these pairs onto C2 to balance the load. Using a full update would require reinstalling all the rules in the network at the new version, even rules independent of the change. The *subset* mechanism instead updates just the rules involving the affected pair of hosts.

**Switch Maintenance** For scheduled maintenance, the controller installs a configuration that removes traffic from C1 and puts it all onto C2. The *island mechanism* recognizes that the aggregation and core switches form a connected component that contains all switches affected by the update and restricts the update to just that subgraph of the network, shown in Figure 5.5.

| **Application** | **Toplogy** | **Update** | **2PC** | | **Subset** | | |
|---|---|---|---|---|---|---|---|
| | | | *Ops* | *Max Overhead* | *Ops* | *Ops %* | *Max Overhead* |
| Routing | Fat Tree | Hosts | 239830 | 92% | 119003 | 50% | 20% |
| | | Routes | 266234 | 100% | 123929 | 47% | 10% |
| | | Both | 239830 | 92% | 142379 | 59% | 20% |
| | Waxman | Hosts | 273514 | 88% | 136230 | 49% | 66% |
| | | Routes | 299300 | 90% | 116038 | 39% | 9% |
| | | Both | 267434 | 91% | 143503 | 54% | 66% |
| | Small World | Hosts | 320758 | 80% | 158792 | 50% | 30% |
| | | Routes | 326884 | 85% | 134734 | 41% | 23% |
| | | Both | 314670 | 90% | 180121 | 57% | 41% |
| Multicast | Fat Tree | Hosts | 1043 | 100% | 885 | 85% | 100% |
| | | Routes | 1170 | 100% | 634 | 54% | 57% |
| | | Both | 1043 | 100% | 949 | 91% | 100% |
| | Waxman | Hosts | 1037 | 100% | 813 | 78% | 100% |
| | | Routes | 1132 | 85% | 421 | 37% | 50% |
| | | Both | 1005 | 100% | 821 | 82% | 100% |
| | Small World | Hosts | 1133 | 100% | 1133 | 100% | 100% |
| | | Routes | 1114 | 90% | 537 | 48% | 66% |
| | | Both | 1008 | 100% | 1008 | 100% | 100% |

Experimental results comparing two-phase update (2PC) with our subset optimization (Subset). We add or remove hosts and change routes to trigger configuration updates. The *Ops* column measures the number of OpenFlow install operations used in each situation. The Subset portion of the table also has an additional column (Ops %) that tabulates (Subset Ops / 2PC Ops). *Overhead* measures the extra rules concurrently installed on a switch by our update mechanisms. We pessimistically present the maximum of the overheads for all switches in the network – there may be many switches in the network that never suffer that maximum overhead.

Table 5.2: Experimental results.

## 5.8 Implementation and Evaluation

We have built a system called Kinetic that implements the update abstractions introduced in this chapter, and evaluated its performance on small but canonical example applications. This section summarizes the key features of Kinetic and presents experimental results that quantify the cost of implementing network updates in terms of the number of rules added and deleted on each switch.

**Implementation overview** Kinetic is a run-time system that sits on top of the NOX OpenFlow controller [34]. The system comprises several Python classes for representing network configurations and topologies, and a library of update mechanisms. The interface to these mechanisms is through the `per_packet_update` and `per_flow_update` functions. These functions take a new configuration and a network topology, and implement a transition to the new configuration while providing the desired consistency level. Both functions are currently based on the two-phase update mechanism, with the `per_flow_update` function using timeouts to track active flows. In addition to this basic mechanism, we have implemented a number of optimized mechanisms that can be applied under certain conditions—*e.g.*, when the update only affects a fraction of the network or network traffic. The runtime automatically analyzes the new configuration and topology and applies these optimizations when possible to reduce the cost of the update.

As described in Section 5.5, the two-phase update mechanism uses versioning to isolate the old configuration and traffic from the updated configuration. Because Kinetic runs on top of OpenFlow 1.0, we currently use the VLAN field to carry version tags (other options, like MPLS labels, are available in newer versions of OpenFlow). Our algorithms analyze the network topology to determine the ingress and internal ports and perform a two-phase update.

**Experiments** To evaluate the performance of Kinetic, we developed a suite of experiments using the Mininet [37] environment. Because Mininet does not offer performance fidelity or resource isolation between the simulated switches and the controller, we did not measure the time needed to implement an update. However, as a proxy for elapsed time, we counted the total number of install OpenFlow messages needed to implement each update, as well as the number of extra rules (beyond the size of either the old or new configurations) installed on a switch.

To evaluate per-packet consistency, we have implemented two canonical network applications: routing and multicast. The routing application computes the shortest paths between each host in the topology and updates routes as hosts come online or go offline and switches are brought up and taken down for maintenance. The multicast application divides the hosts evenly into two multicast groups and implements IP multicast along a spanning tree that connects all of the hosts in a group. To evaluate the effects of our optimizations, we ran both applications on three different topologies each containing 192 hosts and 48 switches in each of three different scenarios. The topologies were chosen to represent realistic and proposed network topologies found in datacenters (fattree, small-world), enterprises (fattree) and a random topology (waxman). The three scenarios can be divided up into:

1. Dynamic hosts and static routes
2. Static hosts and dynamic routes
3. Dynamic hosts and dynamic routes

In each scenario, we moved between 3 different configurations, changing the network in a well-prescribed manner. In the dynamic host scenario, we randomly selected between $10\% - 20\%$ of the hosts and added or removed them from the network. In the dynamic routes scenario, we randomly selected 20% of the routes in the network, and forced them to re-route as if one of the switches in the route had been removed. For the multicast example, we changed one of the multicast groups each time. Static means that we did not change the host or routes.

To evaluate per-flow updates, we developed a load-balancing application that divides traffic between two server replicas, using a hash computed from the client's IP address.

The update for this experiment involved bringing several new server replicas online and re-balancing the load among all of the servers.

**Results and analysis** Table 5.2 compares the performance of the subset optimization to a full two-phase update. Extension updates are not shown: whenever an extension update is applicable, our subset mechanism performs the same update with the same overhead. The two-phase update has high overhead in all scenarios.

We subject each application to a series of topology changes—adding and dropping hosts and links—reflecting common network events that force the deployment of new network configurations. We measure the number of OpenFlow operations required for the deployment of the new configuration, as well as the overhead of installing extra rules to ensure per-packet consistency. The overhead is the ratio of the number of extra rules installed during the per-packet update of a switch divided by the (maximum) number of rules in the old or new configuration. For example, if the old and new configurations both had 100 rules and during the update the switch had 120 rules installed, that would be a 20% overhead. The *Overhead* column in Table 5.2 presents the maximum overhead of all switches in the network. Two-phase update requires approximately 100% overhead, because it leaves the old configuration on the switch as it installs the new one. Because both configurations may not be precisely the same size, it is not always exactly 100%. In some cases, the new configuration may be much smaller than the old (for example, when routes are diverted away from a switch) and the overhead is much lower than 100%.

The first routing scenario, where hosts are added or removed, demonstrates the potential of our optimizations. When a new host comes online, the application computes routes between it and every other online host. Because the rules for the new routes do not affect traffic between existing hosts, they can be installed without modifying or reinstalling the existing rules. Similarly, when a host goes offline, only the installed rules routing traffic to or from that host need to be uninstalled. This leads to update costs proportional to the number of rules that changed between configurations, as opposed to a full two-phase update, where the cost is proportional to the size of the entire new configuration.

Our optimizations yield fewer improvements for the multicast example, due to the nature of the example: when the spanning tree changes, almost all paths change, triggering an expensive update.

We have not applied our optimizations to the per-flow mechanism, therefore we do not include an optimization evaluation of the load balancing application.

## 5.9 Conclusions and Future Work

Reasoning about concurrency is notoriously difficult, and network software is no exception. To make fundamental progress, the networking field needs simple, general, and reusable abstractions for changing the configuration of the network. Our per-packet and per-flow consistency abstractions allow programmers to focus their attention on the state of the network before and after a configuration change, without worrying about the transition in between. The update abstractions are powerful, in that the programmer does not need to identify the properties that should hold during the transition, since *any* property common to both configurations holds for *any* packet traversing the network during the update. This enables lightweight verification techniques that simply verify the properties of the old and new configurations. In addition, our abstractions are practical, in that efficient and correct update mechanisms exist and are implementable using today's OpenFlow switches. Our implementation and Coq proofs are available at our website www.frenetic-lang.org.

In our ongoing work, we are exploring new mechanisms that make network updates faster and cheaper, by limiting the number of rules or the number of switches affected. In this investigation, our theoretical model is a great asset, enabling us to prove that our proposed optimizations are correct. We also plan to extend our formal model to capture the per-flow consistent update abstraction, and prove the correctness of the per-flow update mechanisms. In addition, we will make our update library available to the community, to enable future OpenFlow applications to leverage these update abstractions. Finally, while per-packet consistency and per-flow consistency are core abstractions with excellent semantic properties, we want to explore other notions of consistency that either perform better (but remain sufficiently strong to provide benefits beyond eventual consistency) or provide even richer guarantees.

CHAPTER 6

# RELATED WORK

## 6.1 General approaches

This thesis proposes a methodology for building reliable networking systems through verification. This is not the first such proposal; see *e.g.*the survey paper [89]. Indeed, a system called *Formally Verifiable Networking* by Wang *et al.* [110] proposed that network protocols be designed and written in a specialized logic programming language called Network Datalog (NDlog) which could then be formally analysed against a formal specification. This is similar in spirit to the verification performed in Chapter 3, though the focus and design is different. NDlog was focused upon building distributed protocols on top of a clean-slate architecture built upon Datalog. It also used the PVS theorem prover [87], which requires users to manually construct proofs of correctness, unlike the fully automated decision procedure in this thesis.

The approach taken in this thesis is to start with network programs written in domain-specific, which are usually generated by a higher-level application written in a general purpose programming language. By performing verification on the output, we essentially "cut off" the need to reason about this higher-level application to assure correctness. An alternative approach is to instead push the reasoning up the stack, and develop higher-level and more expressive primitives for network programming to enable the programmer to reason directly about the top-level program. NDlog is one example of this; another Datalog based language is FlowLog [85],[84]. FlowLog provides a *tierless* programming model in which there is a single unified abstraction shared between the control-plane, data-plane, and controller state. In addition, FlowLog policies can be verified with the Alloy analyzer [44]. Alloy is not a complete procedure; it only performs bounded verification, but the authors of FlowLog found that bounded verification sufficed for almost all of their examples.

The Kinetic system [54][1] approaches the problem of building dynamic network systems with a domain specific language based finite-state machines that react to network events, and a temporal-logic based specification language that is used to analyze the correctness of the finite-state machines.

Verdi, [112] is a system in a similar spirit to this thesis, but focused on the distributed systems domain. Verdi is a formal framework for designing and implementing distributed systems in the Coq theorem prover. It focuses reasoning about the behavior of systems under different failure models, and allows programmers to pick and choose which failure models to adopt.

## 6.2 Formally verified systems

Verification technology has progressed dramatically in the past decades, making it feasible to prove useful theorems about real systems including databases [67], compilers [59], and even whole operating systems [55]. Compilers have been particularly fruitful targets for verification efforts [40]. Most prominently, the CompCert compiler translates programs in a large subset of C to PowerPC, ARM, and x86 executables [59]. The Verified Software Toolchain project provides machine-checked infrastructure for connecting properties obtained by program analysis to guarantees at the machine level [5]. Rocksalt verifies a tool for analyzing machine code against a detailed model of x86 [79]. Another system, Bedrock provides rich Coq libraries for verifying low-level programs [17]. Significant portions of

[1]Note: the system in the paper Chapter 5 is based upon was also named Kinetic. There is no relation between the two.

many other compilers have been formalized and verified, including the LLVM intermediate representation [119], the F* typechecker [102], and an extension of CompCert with garbage collection [69].

The seL4 microkernel [55] project built the first fully verified general purpose OS kernel. They started with a formal specification of full functional correctness, written in Isabelle/HOL [86], built an executable specification (written in Haskell) that provably refined the original specification, and finally built a high-performance implementation in C, which in turn refined the Haskell specification. The final seL4 system achieved similar performance to other L4 micro-kernels, but with a formal guarantee of correctness and reliability.

The CompCert project [59] is another celebrated fully formally verified system. CompCert is a C compiler built in the Coq theorem prover and fully verified for correctness against a formal model of C semantics and the semantics of the x86, ARM, and PowerPC architectures. CompCert includes high-performance, fully verified compiler optimizations, and achieves respectable performance against other, unverified compilers. In one study of compiler bugs, every compiler except the verified CompCert compiler was found to have contain bugs that caused wrong code generation [114][2]. Before CompCert, the so-called "CLInc stack" [11] was a formally verified system consisting of a high-level programming language with a verification engine (Gypsy [32]); a verified compiler for a restricted language [115]; a verified assembler (Piton [78]); a verified multitasking operating system (Kit[10]); and a fully verified microprocessor (the FM8502 [42]).

In a different vein, there have been several projects that built formal models of networks or network protocols (see *e.g.*[74] [113]). One of the most detailed formal networking models

[2]They initially found a bug in an unverified component of CompCert. In response to the bug, the project extended the verification to include that component, and the authors were no longer able to find any bugs in the compiler.

ever built is Bishop *et al.*'s model of the TCP protocol and the Sockets API[13]. They built a fully formal, highly precise model of the TCP protocol and its associated Sockets API in HOL4 [33], and exhaustively validated it against de facto reference implementations.

A portion of the PANE [21] compiler was formalized in Coq, but since the proof did not model several subtleties of flow tables, the compiler still had bugs. Unlike our system, PANE does not model or verify any portion of its run-time system. We used some of the PANE proofs during early development of our system.

## 6.3 Network verification tools

**Specification languages** Before SDN, abstract network specifications independent from implementation were largely limited to firewall policies. See, *e.g.* [7] for an entity-relationship modeling framework for specifying security policies. One particularly notable early work in this area was Guttman's filtering postures [36], which took a global network access control policy and automatically specialized it into local filters whose combination was guaranteed to enforce the global policy.

More recently, the VeriFlow [53] network verification system includes a general API for checking application-specific network invariants. This API is not exactly a specification language (it lacks semantics), and it's not exactly clear what properties it can and can not express. Invariants must be checkable in a sort of incremental manner, where only modified equivalence classes of network rules are given at each step.

The NetPlumber [49] system includes a specification language for relating network flows to allowable paths. This language is based upon regular expressions with wildcards, and is quite similar in spirit to the Pathetic language in this chapter. However, the languages are subtly different: whereas Pathetic is based on regular expressions in the sense of formal language theory, and comes with a formal semantics, NetPlumber's language is based on regular expressions in the sense of the string matching functions found in many programming languages. Arguably, the latter design decision makes them easier to understand and more familiar to the average programmer. They also lack a semantics, making it difficult or impossible for a programmer to reason about the exact meaning of their specification.

ConfigChecker [3] is a system for analyzing firewall and router configurations. It uses specifications of network behavior specified in Computational Tree Logic (CTL), a branching-time temporal logic.

**Network verifiers/static analyzers** One of the first static analyzers to achieve widespread adoption was Feamster and Balakrishnan's routing configuration checker *rcc*[20]. *rcc* was a tool that statically analyzed Border Gateway Protocol (BGP) router configurations for common configuration mistakes such as typos, learning unusable paths, sharing unusable paths, or failing to learn all usable paths. *rcc* was reportedly widely welcomed in industry as an improvement over the status quo of running configurations in small test-beds and then pushing them out to production to detect bugs. *rcc* was limited to detecting generic configuration faults, and was not able to verify full application specific functional correctness.

Recent years have seen an incredible interest in the development of verification tools for the networking domain, largely focused around SDN. Xie et al. introduced techniques for statically analyzing the reachability properties of networks [113]. Some prominent recent domain specific verification engines include: Header Space Analysis [50] and NetPlumber [49], which introduced the idea of including location in the packet metadata, uniformizing packet transformations and topology and Veriflow [53], which functions as a real-time invariant checker sitting between a controller and the network. Zhang and Malik [118] build a SAT-based verification framework that uses a similar network model to Header Space Analysis, but generalizes the model to allow uniform specification of correctness requirements[3]. Unlike the system described in Chapter 3, these tools work at several levels below the programmer, making it difficult to relate the results of verification back to the actual code. For example, if one of these verifiers says that a rule that was just inserted into the network breaks reachability, the programmer would have to determine why that rule was added by walking back through the controller that installed the rule, to the compiler that output the rules to be installed, and connect that back to the input policy, which itself was generated by another piece of code.

The model developed by Kazemian *et. al* [50] was the starting point for the network model in Chapter 5. Since their model only spoke of a single, static configuration, we extended the network semantics to include updates so we could model a network changing dynamically over time. In addition, while their model was used to help *describe* their algorithms, ours was used to help us state and prove various correctness properties of our system.

Finally, Foster *et al.* [25] present an equivalence checker for NetKAT based on very similar foundations as the one in this chapter. The system in this chapter was based on a rewrite of the code base in that paper, and shares a similar architecture and overall bisimulation algorithm. For more specific details on the difference between that paper and this one, see Section 3.6.

[3]Header Space Analysis required ad-hoc extensions of the model to support certain properties. For example, to detect forwarding loops, they extended the packet header to include a list of all visited ports

### 6.3.1 Network debugging

The NICE model checker [14] is a state-space exploration engine that can detect bugs in a full OpenFlow system: switches, controller, and control application. NICE searches for generic network bugs such as forwarding loops or black-holes, and allows the user to program application specific invariants to test. The system uses domain specific exploration strategies to reduce the complexity of the state space and find bugs more quickly. Portions of the Featherweight OpenFlow model in Chapter 4 are inspired by the bugs discovered in NICE.

Sethi *et al.* [96] extend the bounded verification approach of NICE to arbitrarily large numbers of packets with the use of application specific *non-interference* lemmas. The Kuai system [66] further scaled the model checking approach to much larger topologies, and automatically deals with unbounded sets of packets without the need for manual lemmas[4]

A number of projects have focused upon easing the difficulty of debugging network control and dataplanes by extending models of software debuggers to networks as in NDB [38]; automatically generating test packets to detect bugs in data planes [116]; or analyzing snapshots of network state with a SAT solver to detect failures in the dataplane [65].

### 6.3.2 Network verification

VeriCon [6] is a system that verifies the correctness of an SDN program for all topologies with a specific property, and all possible network inputs. Based upon Z3 [18], VeriCon specifies desired network behavior and acceptable topologies in first-order logic, rather than a networking specific language. Unlike most of the other systems here, they assume in-order,

[4]As an interesting coincidence, Kuai is built upon the PReach distributed model checker [12], which the author helped build in between undergrad and graduate school.

atomic installation of switch rules, an assumption that can be enforced in practice, but only with a high performance penalty.

The verification tool in Chapter 3 was originally based upon the NetKAT verifier in [25]. The differences between that tool and the one in this thesis are outlined in detail in the relevant chapter. Stewart [100] built a formal verification system for NetCore (a predecessor to NetKAT) based upon Hoare triples in Coq, and proved the verification tool itself correct with respect to a formal model of NetCore based upon the semantics in Chapter 4.

In a different vein, Dobrescu and Argyraki verify the correctness network dataplanes implemented in the Click software router framework [56]. They use symbolic execution to prove properties of software dataplanes that are routinely verified for hardware dataplanes such as crash-freedom or bounded execution.

## 6.4 Network updates

Updating networks without introducing undesired behavior has long been recognized as an important and difficult problem. Because of the great complexity of networks, even solutions that avoid problems in one protocol level can inadvertently introduce undesired behavior in another, nominally independent, protocol (see especially [105] for a surprising example of this interplay between IGP and BGP). Before the rise of SDN, most approaches focused upon manipulating the inputs to distributed routing protocols so that they would converge to desired configurations without transitioning through "bad states". A full overview of this area would be beyond the scope of this chapter, but a representative sampling can be found through these topics: performing network maintenance without disruption [97] [52] [28], handling topology changes [80] [98] [29], routing protocol migration [51] [108] [106], avoiding disruptions during traffic engineering [26] [27], general techniques for avoiding introducing forwarding loops [30] [99], and general frameworks for updating distributed networking protocols [16] [90]. For a more thorough exploration of the area, see *e.g.* Vanbever's thesis [107].

Consensus Routing [46] seeks to eliminate transient errors, such as disconnectivity, that arise during BGP updates. In particular, Consensus Routing's "stable mode" is similar to our per-packet consistency, though computed in a distributed manner for BGP routes. On the other hand, Consensus Routing only applies to a single protocol (BGP), whereas our work may benefit any protocol or application developed in our framework. The BGP-LP mechanism from [47] is essentially per-packet consistency for BGP.

The dynamic software update problem is related to network update consistency. The problem of upgrading software in a general distributed system is addressed in [2]. The scope of that work differs in that the nodes being updated are general purpose computers, not switches, running general software.

The related problem of maintaining safety while updating firewall configurations has been addressed by Zhang *et. al* [117]. That work formalized a definition of safety similar in spirit to per-packet consistency, but limited to a single device.

### 6.4.1 Alternative abstractions

Since the original publication of the per-packet and per-flow consistency abstractions, there has been a large body of work proposing new abstractions and new mechanisms.

**Customizable properties** One such abstraction preserves specific, application specific properties through a general update specialization procedure. Instead preserving all possible *trace properties* as in per-packet consistency, a programmer can specify exactly the properties they want preserved, and achieve a faster update. The Dionysus system [45] dynamically schedules updates based upon the exact properties to be preserved, as well as the performance characteristics of the switches themselves[5]. However, Dionysus requires a rule dependency calculation that is specific to the general property being preserved, and thus can be automatically applied to different properties. The network synthesis approach [68] views network updates as a distributed programming problem and uses techniques from program synthesis to construct minimal update sequences that preserve desired network invariants specified in temporal logic. Similarly, the Customizeable Consistency Generator [121] analyses the desired invariant and synthesizes a update sequence that satisfies it, or resorts to the consistent update mechanisms outlined in Chapter 5 when no such sequence exists.

**Alternative properties** Consistent updates are guaranteed to preserve all *trace properties*, but not all network properties are *trace properties*. For example, guarantees about congestion, bandwidth or latency are not preserved by consistent updates. The Software-driven Wide Area Network (SWAN) [41] system used SDN to optimzie the network utilization of expensive WAN links. SWAN analyzed the problem of a *congestion-free* update: given a congestion-free network state with known loads and link capacities, compute a sequence of network updates that lead to target congestion-free state such that none of the intermediate states introduce congestion. They show that, in general, no such update may exist, but if $x\%$ of “scratch capacity” is left on each link, then an update sequence can be found of length at most $\lceil \frac{1}{x} \rceil - 1$. Similarly, the zUpdate [60] system provides an abstraction that guarantees

[5]Dionysus found that update times can vary dramatically between different switches. By dynamically adjusting its update schedule to the observed performance of the switches, they were able to achieve significant increases in update performance

a congestion-free, lossless network update.

Per-flow consistency generalizes per-packet consistency by preserving properties on related packets within a single flow. Inter-flow consistency [61] generalizes per-flow consistency by preserving constraints between different flows. For example, inter-flow consistency can guarantee that two specific flows are never colocated on the same link (*e.g.*for fault-tolerance), or that related flows in different directions (*e.g.*the two flows of a single TCP connection) are treated in a consistent manner.

### 6.4.2 Optimized update mechanisms

The update mechanisms described in this thesis may require high-switch rulespace overhead, or long convergence time. A number of optimizations that explore different tradeoffs in time and space have been proposed. If an update is split into several incremental updates, then the maximum rulespace overhead required can be greatly reduced [48]. Similarly, Luo *et al.* observe that by carefully exploiting OpenFlow wildcard rules, rules common to both the old and new configuration can be left on the switch [63]. This optimization is related to the subspace update outlined in Chapter 5, and if combined with that analysis can lead to highly compact updates. ESPRES [88] is an update scheduler, similar in spirit to Dionysus, that reorders, rate-limits, and prioritizes updates to avoid overloading the limited control plane bandwidth found in early generation OpenFlow switches. McGeer [70] describes an algorithm that completely removes the overhead of two-phase update by trading off the time required for the update and possibly introducing latency to packets traveling through the network during update.

The per-flow mechanisms described in this thesis assume that all packets in a single flow arrive at the same ingress switch. Afek *et al.* describe a mechanism [1] that removes this restriction. Similarly, Zhao *et al.* [120] use an optimization approach to minimize the overhead and management required to preserve per-flow consistency using the flow binning scheme. Liu *et al.* [62] provide a heuristic algorithm to solve the per-flow optimization problem.

In a different and novel approach, Mizrahi and Moses [75] propose using precisely synchronized network clocks to schedule and perform network updates with minimal inconsistency windows.

CHAPTER 7

# CONCLUSIONS

*"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."* —Leslie Lamport

To someone who's never built or maintained one, a computer network must seem like the simplest of objects. You draw a network diagram, starting with the nodes you want to communicate, draw a graph fully connecting them, and then you go build it. And in fact, if the only function of your network is connectivity, it is not much more conceptually difficult than that[1]. Unfortunately, most computer networks have requirements beyond simple connectivity. Consider a pre-paid wireless network, such as the ones commonly found in airports or onboard airplanes. The network must provide full connectivity to paid subscribers, limited connectivity to unpaid users, enable new users to create accounts, track and bill traffic usage (but not traffic used for creating/checking accounts!), throttle or restrict traffic for users when their subscription ends, and protect users from one another. And it's not enough just to build the network: maintaining and operating it introduces a whole set of operational requirements such as active monitoring to detect failures, logging to debug connectivity problems, and fine-grained geographic tracking for analytics[2], just to name a few. All of these different requirements are traditionally implemented with their own set of protocols and mechanisms, many of which, by necessity, overlap and interact in complicated ways.

[1]Even in this case, things can quickly become complicated: which connected graph should be chosen? Should wired or wireless links be used? What happens when a link or node fails? How do you add new nodes to the network? What addressing scheme should you use? What nodes need to be able to broadcast to each other?

[2]Which access points are overutilized? Underutilized? How does utilization vary with time of day? *ad infinitum*

And yet, at its core, a network truly is a simple object. Almost all network functionality boils down to looking at a packet and deciding how to modify it and where to forward it. The techniques, languages, and tools presented in this thesis attack the complexity of networking by distilling it down to this simple core.

We began by describing a specification language that describes the desired flow of packets through a network at a high-level of abstraction, but still admits automatic verification of correct implementations. We then showed how to compile network programs into the low-level language of switches in a provably correct way that preserves the original verification promises. Finally, we showed how focusing on the behavior of a single packet through the network leads to a correctness criteria for network updates that provides strong reasoning guarantees about network behavior, even while it is in flux.

# BIBLIOGRAPHY

[1] Yehuda Afek, Anat Bremler-Barr, and Liron Schiff. Ranges and cross-entrance consistency with openflow. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 233–234, New York, NY, USA, 2014. ACM.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 452–476, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] E. Al-Shaer and M.N. Alsaleh. Configchecker: A tool for comprehensive security configuration analytics. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–2, Oct 2011.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[5] Andrew W. Appel. Verified software toolchain. In *ESOP*, 2011.

[6] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Notices*, volume 49, pages 282–293. ACM, 2014.

[7] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 17–31, 1999.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.

[9] Yves Bertot and Pierre Casteran. Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions. In *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.

[10] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, Nov 1989.

[11] William R Bevier, Warren A Hunt Jr, J Strother Moore, and William D Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[12] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, PDMC-HIBI '10, pages 28–36, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Steve Bishop, Matthew Fairbairn, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous specification and validation for tcp/ip and the sockets api.

[14] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[15] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for software-defined networks. *Commun. ACM*, 57(10):86–95, Sep 2014.

[16] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. Pacman: A platform for automated and controlled network operations and configuration management. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 277–288, New York, NY, USA, 2009. ACM.

[17] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[20] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[21] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM '13*, Hong Kong, China, August 2013.

[22] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.

[23] N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, February 2013.

[24] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[25] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 343–355, New York, NY, USA, 2015. ACM.

[26] P. Francois and O. Bonaventure. Avoiding transient loops during IGP convergence in ip networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 237–247 vol. 1, March 2005.

[27] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Trans. on Networking*, Dec 2007.

[28] Pierre Francois, Pierre-Alain Coste, Bruno Decraene, and Olivier Bonaventure. Avoiding disruptions during maintenance operations on BGP sessions. *IEEE Trans. on Network and Service Management*, Dec 2007.

[29] Pierre Francois, Mike Shand, and Olivier Bonaventure. Disruption-free topology reconfiguration in OSPF networks. In *IEEE INFOCOM*, May 2007.

[30] Jing Fu, P. Sjodin, and G. Karlsson. Loop-free updates of forwarding tables. *Network and Service Management, IEEE Transactions on*, 5(1):22–35, March 2008.

[31] Wouter Gelade and Frank Neven. Succinctness of the Complement and Intersection of Regular Expressions. In Susanne Albers and Pascal Weil, editors, *25th International*

*Symposium on Theoretical Aspects of Computer Science*, volume 1 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 325–336, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] D. I. Good and B. A. Wichmann. Mechanical proofs about computer programs [and discussion]. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 312(1522):389–409, 1984.

[33] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, New York, NY, USA, 1993.

[34] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[35] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-Verified Network Controllers . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, WA*, Jun 2013.

[36] J.D. Guttman. Filtering postures: local enforcement for global policies. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 120–129, May 1997.

[37] Nikhil Handigol, Brandon Heller, Vimal Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[38] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60. ACM, 2012.

[39] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM.*

[40] Tony Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, Jan 2003.

[41] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan.

In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

[42] Warren A Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[43] Chris Isidore and Jose Pagliery. United flights resume after computer problem, July 2015. [Online; accessed 17-July-2015].

[44] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis.* MIT Press, 2006.

[45] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.

[46] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The Internet as a distributed system. In *NSDI*, Apr 2008.

[47] Dina Katabi, Nate Kushman, and John Wrocklawski. A Consistency Management Layer for Inter-Domain Routing. Technical Report MIT-CSAIL-TR-2006-006, Cambridge, MA, Jan 2006.

[48] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 49–54, New York, NY, USA, 2013. ACM.

[49] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[51] Eric Keller, Jennifer Rexford, and Jacobus Van Der Merwe. Seamless BGP migration with router grafting. In *Proceedings of the 7th USENIX Conference on Networked*

*Systems Design and Implementation*, NSDI'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[52] Ram Keralapura, Chen-Nee Chuah, and Yueyue Fan. Optimal strategy for graceful network upgrade. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*, INM '06, pages 83–88, New York, NY, USA, 2006. ACM.

[53] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, IL*, Apr 2013.

[54] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, May 2015. USENIX Association.

[55] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, 2009.

[56] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug 2000.

[57] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 14–33. 1996.

[58] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[59] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, Jul 2009.

[60] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 411–422, New York, NY, USA, 2013. ACM.

[61] Weijie Liu, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. Inter-flow consistency: Novel SDN update abstraction for supporting inter-flow constraints. In *Proceedings of the Network and Distributed System Security Symposium*, Network and Distributed System Security Symposium 2015, 2015.

[62] Yujie Liu, Yong Li, Yue Wang, Athanasios V. Vasilakos, and Jian Yuan. Achieving efficient and fast update for multiple flows in software-defined networks. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 77–82, New York, NY, USA, 2014. ACM.

[63] Shouxi Luo, Hongfang Yu, and Lemin Li. Consistency is not easy: How to use two-phase update for wildcard rules? *Communications Letters, IEEE*, 19(3):347–350, March 2015.

[64] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

[65] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[66] Rwitajit Majumdar, Sai Deep Tetali, and Zhen Wang. Kuai: A model checker for software-defined networks. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 163–170. IEEE, 2014.

[67] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Towards a verified relational database management system. In *POPL*, 2010.

[68] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM.

[69] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[70] Rick McGeer. A correct, zero-overhead protocol for network updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 161–162, New York, NY, USA, 2013. ACM.

[71] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[72] CHUCK MIKOLAJCZAK and HERBERT LASH. Network failure interrupts quotes, trade data for NYSE stocks, July 2015. Online; accessed 25-July-2015.

[73] Steven P Miller and Mandayam Srivas. Formal verification of the aamp5 microprocessor: A case study in the industrial use of formal methods. In *Industrial-Strength Formal Specification Techniques, 1995. Proceedings., Workshop on*, pages 2–16. IEEE, 1995.

[74] Saber Mirzaei, Sanaz Bahargam, Richard Skowyra, Assaf Kfoury, and Azer Bestavros. Using Alloy to formally model and reason about an openflow network switch. *Computer Science Department, Boston University, Tech. Rep*, 7, 2013.

[75] Tal Mizrahi and Yoram Moses. Time-based updates in software defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 163–164, New York, NY, USA, 2013. ACM.

[76] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, Aug 2011.

[77] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.

[78] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[79] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[80] J Moy, Padma Pillay-Esnault, and Acee Lindem. Graceful OSPF restart. Technical report, 2003. RFC 3623.

[81] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[82] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[83] Sanjai Narain, Rajesh Talpade, and Gary Levin. Network configuration validation. In Charles R. Kalmanek, Sudip Misra, and Yang (Richard) Yang, editors, *Guide to Reliable Internet Services and Applications*, Computer Communications and Networks, pages 277–316. Springer London, 2010.

[84] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, Seattle, WA, 2014. USENIX Association.

[85] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 79–84, New York, NY, USA, 2013. ACM.

[86] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[87] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

[88] Peter Perešíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. Espres: Easy scheduling and prioritization for SDN. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX.

[89] Junaid Qadir and Osman Hasan. Applying formal methods to networking: Theory, techniques, and applications. *Communications Surveys & Tutorials, IEEE*, 17(1):256–291, 2015.

[90] S. Raza, Y. Zhu, and C-N. Chuah. Graceful network operations. In *IEEE INFOCOM*, Apr 2009.

[91] S. Raza, Y. Zhu, and C-N. Chuah. Graceful network state migrations. *IEEE/ACM Trans. on Networking*, 19(4), Aug 2011.

[92] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault

tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[93] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug 2012.

[94] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, Oct 2000.

[95] Amazon Web Services. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region, April 2011. [Online; accessed 27-June-2015].

[96] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148. IEEE, 2013.

[97] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of multiple OSPF routers. *Networking, IEEE/ACM Transactions on*, 14(3):532–542, June 2006.

[98] Mike Shand and Les Ginsberg. Restart signaling for IS-IS. Technical report, 2008. RFC 5306.

[99] Lei Shi, Jing Fu, and Xiaoming Fu. Loop-free forwarding table updates with minimal link overflow. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC'09, pages 2091–2096, Piscataway, NJ, USA, 2009. IEEE Press.

[100] Gordon Stewart. Computational verification of network programs in Coq. In *Certified Programs and Proofs*, pages 33–49. Springer, 2013.

[101] LJ Stockmeyer. *The complexity of decision procedures in Automata Theory and Logic*. PhD thesis, PhD thesis, MIT, Project MAC Technical Report TR-133, 1974.

[102] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *POPL*, 2012.

[103] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, Oct 2007.

[104] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martín Casado, and Rob Sherwood. On controller performance in software-defined networks. In *HotICE*, 2012.

[105] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure. When the cure is worse than the disease: The impact of graceful IGP operations on BGP. In *INFOCOM, 2013 Proceedings IEEE*, pages 2220–2228, April 2013.

[106] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Lossless migrations of link-state IGPs. *Networking, IEEE/ACM Transactions on*, 20(6):1842–1855, Dec 2012.

[107] Laurent Vanbever. *Methods and Techniques for Disruption-Free Network Reconfiguration*. PhD thesis, University of Louvain, 2012.

[108] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Seamless network-wide IGP migration. In *SIGCOMM*, Aug 2011.

[109] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, Jan 2011.

[110] Anduo Wang, Limin Jia, Changbin Lio, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. In *HotNets*, 2009.

[111] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, Mar 2011.

[112] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *PLDI. ACM*, 2015.

[113] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[114] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[115] W. Young. Verified compilation in micro-Gypsy. In *TAV*, 1989.

[116] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, 2012.

[117] Charles C. Zhang, Marianne Winslett, and Carl A. Gunter. On the safety and efficiency of firewall policy deployment. In *IEEE Symp. on Security and Privacy*, 2007.

[118] Shuyuan Zhang and Sharad Malik. Sat based verification of network data planes. In *Automated Technology for Verification and Analysis*, pages 496–505. Springer, 2013.

[119] Jianzho Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.

[120] Kun Zhao, Qing Li, and Yong Jiang. Flow-level consistent update in SDN based on k-prefix covering. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 1884–1889, Dec 2014.

[121] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2015.

APPENDIX A

# PROOFS

## A.1 Proofs for Chapter 3

We first prove a lemma characterizing the translation of path expressions:

**Lemma 5** (Equivalence of Path expression translation)**.** *For every path expression $P$, $G(P)$ $= G(\llparenthesis P \rrparenthesis)$*

*Proof.* Proof by structural induction on the Pathetic path expression policy $P$.

**Case $\epsilon$**

$$\begin{aligned} G(\epsilon) &= \{\alpha{\cdot}\pi_\alpha \mid \alpha \in \mathsf{At}\} \\ &= G(1) \\ &= G(\llparenthesis \epsilon \rrparenthesis) \end{aligned}$$

**Case $\emptyset$**

$$\begin{aligned} G(\emptyset) &= \emptyset \\ &= G(0) \\ &= G(\llparenthesis \epsilon \rrparenthesis) \end{aligned}$$

**Case S**

$$\begin{aligned} G(S) &= \{\alpha{\cdot}\pi{\cdot}\mathsf{dup}\pi \mid \alpha \in \mathsf{At}, \pi = \pi_\alpha[S/sw]\} \\ &= G(sw \leftarrow S \cdot \mathsf{dup}) \\ &= G(\llparenthesis S \rrparenthesis) \end{aligned}$$

**Case** $\star$

$$\begin{aligned} G(\star) &= \{\alpha{\cdot}\pi{\cdot}\mathsf{dup}\pi \mid \alpha \in \mathsf{At}, \pi = \pi_\alpha[S/sw], S \in \mathtt{Sw}\} \\ &= G(\sum_{S\in\mathtt{Sw}} sw \leftarrow S \cdot \mathsf{dup}) \\ &= G(\llparenthesis\star\rrparenthesis) \end{aligned}$$

**Case** $\overline{P}$

$$\begin{aligned} G(\overline{P}) &= \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(P) \\ &= \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^* \setminus G(\llparenthesis P\rrparenthesis) && \text{By the induction hypothesis} \\ &= G(\overline{\llparenthesis P\rrparenthesis}) \\ &= G(\llparenthesis\overline{P}\rrparenthesis) \end{aligned}$$

**Case** $P.P'$

$$\begin{aligned} G(P.P') &= G(P) \diamond G(P') \\ &= G(\llparenthesis P\rrparenthesis) \diamond G(\llparenthesis P'\rrparenthesis) && \text{By the induction hypothesis} \\ &= G(\llparenthesis P\rrparenthesis \cdot \llparenthesis P'\rrparenthesis) \\ &= G(\llparenthesis P.P'\rrparenthesis) \end{aligned}$$

**Case** $P|P'$

$$\begin{aligned} G(P|P') &= G(P) \cup G(P') \\ &= G(\llparenthesis P\rrparenthesis) \cup G(\llparenthesis P'\rrparenthesis) && \text{By the induction hypothesis} \\ &= G(\llparenthesis P\rrparenthesis + \llparenthesis P'\rrparenthesis) \\ &= G(\llparenthesis P|P'\rrparenthesis) \end{aligned}$$

**Case** $P \cap P'$

$$\begin{aligned} G(P \cap P') &= G(P) \cap G(P') \\ &= G(\llparenthesis P \rrparenthesis) \cap G(\llparenthesis P' \rrparenthesis) && \text{By the induction hypothesis} \\ &= G(\llparenthesis P \rrparenthesis \cap \llparenthesis P' \rrparenthesis) \\ &= G(\llparenthesis P \cap P' \rrparenthesis) \end{aligned}$$

**Case** $P^*$

$$\begin{aligned} G(P^*) &= \bigcup_{i \geq 0} G(P)^i \\ &= \bigcup_{i \geq 0} G(\llparenthesis P \rrparenthesis)^i && \text{By the induction hypothesis} \\ &= G(\llparenthesis P \rrparenthesis^*) \\ &= G(\llparenthesis P^* \rrparenthesis) \end{aligned}$$

□

**Theorem 13** (Theorem 1)**.** *For every Pathetic program $\phi$, $G(\phi) = G(\llparenthesis \phi \rrparenthesis)$*

*Proof.* Proof by induction on $\phi$.

**Case** $a \Rightarrow P$

$$\begin{aligned} G(a \Rightarrow P) &= G(a) \diamond G(P) \\ &= G(a) \diamond G(\llparenthesis P \rrparenthesis) && \text{By Lemma 5} \\ &= G(a \cdot \llparenthesis P \rrparenthesis) \\ &= G(\llparenthesis a \Rightarrow P \rrparenthesis) \end{aligned}$$

**Case** $\phi \uplus \phi'$ Immediate by induction and definition of $(\!|\phi \uplus \phi'|\!)$.

**Case** $\phi \Cap \phi'$ Immediate by induction and definition of $(\!|\phi \Cap \phi'|\!)$.

$\square$

**Theorem 14** (Theorem 2). *$p \vDash \phi$ iff $(\!|\phi|\!) \cap \overline{p} \equiv \mathsf{0}$.*

*Proof.*

$$\begin{aligned}
p \vDash \phi &\iff G(p) \subseteq G(\phi) \\
&\iff \overline{G(p)} \cap G(\phi) = \emptyset \\
&\iff G(\overline{p}) \cap G(\phi) = \emptyset \\
&\iff G(\overline{p}) \cap G((\!|\phi|\!)) = \emptyset \\
&\iff G(\overline{p} \cap G((\!|\phi|\!))) = \emptyset \\
&\iff G(\overline{p} \cap G((\!|\phi|\!))) = G(\mathsf{0}) \\
&\iff \overline{p} \cap G((\!|\phi|\!)) \equiv \mathsf{0}
\end{aligned}$$

$\square$

**Corollary 3** (Corollary 1). *$p \vDash \phi$ iff $p \leq (\!|\phi|\!)$.*

**Theorem 15** (NetKAT$(-, \cap)$ is a conservative extension of NetKAT). *For every complement and intersection-free policy $p$, $[\![p]\!] = [\![p]\!]_{NK}$, where $[\![\cdot]\!]_{NK}$ is the original NetKAT semantics.*

*Proof.* Immediate by induction upon $p$. $\square$

**Theorem 16** (Theorem 3). *For all dup-free policies $p$ and $q$, if $p \equiv q$ is provable by the NetKAT$(-, \cap)$ axioms, then $[\![p]\!]_{-\mathsf{dup}} = [\![q]\!]_{-\mathsf{dup}}$.*

*Proof.* Proof by structural induction on the derivation of $p \equiv q$ with a case analysis on the last rule used. Soundness of the NetKAT axioms follows from Theorem 15.

The soundness of the axioms INTER-*, PAR-INTER-DIST, COMP-PAR, and COMP-INTER follow trivially from the semantics and basic properties of union/intersection.

**INTER-MOD-DIST-LEFT**

$$\begin{aligned}
[\![f \leftarrow n \cdot (p \cap q)]\!]_{-\mathsf{dup}}\ pk &= ([\![f \leftarrow n]\!]_{-\mathsf{dup}} \bullet [\![p \cap q]\!]_{-\mathsf{dup}})\ \ pk \\
&= \bigcup_{pk' \in [\![f \leftarrow n]\!]_{-\mathsf{dup}}\ pk} [\![p \cap q]\!]_{-\mathsf{dup}}\ pk' \\
&= \bigcup_{pk' \in \{pk[n/f]\}} [\![p \cap q]\!]_{-\mathsf{dup}}\ pk' \\
&= [\![p \cap q]\!]_{-\mathsf{dup}}\ pk[n/f] \\
&= [\![p]\!]_{-\mathsf{dup}}\ pk[n/f] \cap [\![q]\!]_{-\mathsf{dup}}\ pk[n/f] \\
&= [\![f \leftarrow n \cdot p]\!]_{-\mathsf{dup}}\ pk \cap [\![f \leftarrow n \cdot q]\!]_{-\mathsf{dup}}\ pk \\
&= [\![(f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q)]\!]_{-\mathsf{dup}}\ pk
\end{aligned}$$

**COMP-FILTER**

$$\begin{aligned}
[\![\overline{f = n}]\!]_{-\mathsf{dup}}\ pk &= \mathcal{PK} \setminus [\![f = n]\!]_{-\mathsf{dup}}\ pk \\
&= \mathcal{PK} \setminus \{pk \mid pk[f] = n\}
\end{aligned}$$

Reasoning by cases:

**If** $pk(f) \neq n$

$[\![\neg f = n \cdot \sum_{\pi} \pi]\!]\ pk = \mathcal{PK}$

**If** $pk(f) = n$

$\llbracket \neg f = n \cdot \sum_\pi \pi \rrbracket\ pk = \emptyset$ and $\llbracket \sum_\alpha \sum_{\pi \neq \pi_\alpha} \alpha \cdot \pi \rrbracket = \mathcal{PK} \setminus \{pk\}$.

**COMP-MOD**

$$\begin{aligned}
\llbracket \sum_\alpha \sum_{\pi \neq \pi_{\alpha[f \leftarrow n]}} \alpha \cdot \pi \rrbracket\ pk &= \bigcup_{\alpha, \pi \neq \pi_{\alpha[f \leftarrow n]}} \llbracket \alpha \cdot \pi \rrbracket\ pk \\
&= \bigcup_{\pi \neq \pi_{\alpha[f \leftarrow n]}} \llbracket \pi \rrbracket\ pk \qquad \text{For the unique } \alpha \text{ such that } \alpha(\pi) \\
&= \mathcal{PK} \setminus \{pk[f/n]\} \\
&= \llbracket \overline{f \leftarrow n} \rrbracket\ pk
\end{aligned}$$

**COMP-SEQ**

$$\begin{aligned}
\llbracket \overline{p \cdot q} \rrbracket_{-\mathsf{dup}}\ pk &= \overline{\llbracket p \cdot q \rrbracket_{-\mathsf{dup}}\ pk} \\
&= \overline{\bigcup_{pk' \in \llbracket p \rrbracket_{-\mathsf{dup}}\ pk} \llbracket q \rrbracket_{-\mathsf{dup}}\ pk'} \\
&= \bigcap_{pk' \in \llbracket p \rrbracket_{-\mathsf{dup}}\ pk} \overline{\llbracket q \rrbracket_{-\mathsf{dup}}\ pk'} \\
&= \bigcap_{pk' \in \llbracket p \rrbracket_{-\mathsf{dup}}\ pk} \llbracket \overline{q} \rrbracket_{-\mathsf{dup}}\ pk' \\
&= \bigcap_{pk'} ([pk' \notin \llbracket p \rrbracket_{-\mathsf{dup}}\ pk] \cdot \mathsf{all} \cup \llbracket \overline{q} \rrbracket_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} \left([pk' \in \overline{\llbracket p \rrbracket_{-\mathsf{dup}}\ pk}] \cdot \mathsf{all} \cup \llbracket \overline{q} \rrbracket_{-\mathsf{dup}}\ pk'\right) \\
&= \bigcap_{pk'} ([pk' \in \llbracket \overline{p} \rrbracket_{-\mathsf{dup}}\ pk] \cdot \mathsf{all} \cup \llbracket \overline{q} \rrbracket_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} (\llbracket \overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} \rrbracket_{-\mathsf{dup}}\ pk \cup \llbracket \overline{q} \rrbracket_{-\mathsf{dup}}\ pk') \\
&= \bigcap_{pk'} (\llbracket \overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} \rrbracket_{-\mathsf{dup}}\ pk \cup \llbracket \pi_{pk'} \cdot \overline{q} \rrbracket_{-\mathsf{dup}}\ pk) \\
&= \bigcap_{pk'} (\llbracket \overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} + \pi_{pk'} \cdot \overline{q} \rrbracket_{-\mathsf{dup}}\ pk)
\end{aligned}$$

$$= [\![\bigcap_{pk'} \overline{p} \cdot \alpha_{pk'} \cdot \mathsf{all} + \pi_{pk'} \cdot \overline{q}]\!]_{-\mathsf{dup}}\ pk$$

□

**Theorem 17** (Theorem 4)**.** *For all policies $p$ and $q$, if*

$$p \equiv q$$

*in the equational theory generated by the NetKAT($-, \cap$) axioms minus* COMP-FILTER*,* COMP-MOD*, and* COMP-SEQ*, then*

$$[\![p]\!] = [\![q]\!]$$

*Proof.* Proof by structural induction on the derivation of $p \equiv q$ with a case analysis on the last rule used. Soundness of the NetKAT axioms follows from Theorem 15.

The soundness of the axioms INTER-* and PAR-INTER-DIST follow trivially from the semantics and basic properties of union/intersection. This leaves only the axiom INTER-MOD-DIST-LEFT.

$$\begin{aligned}
[\![f \leftarrow n \cdot (p \cap q)]\!]\ pk{::}h &= ([\![f \leftarrow n]\!] \bullet [\![p \cap q]\!])\ \ pk{::}h \\
&= \bigcup_{pk'::h' \in [\![f \leftarrow n]\!]\ \ pk::h} [\![p \cap q]\!]\ pk'{::}h' \\
&= \bigcup_{pk'::h' \in \{pk[n/f]::h\}} [\![p \cap q]\!]\ pk'{::}h' \\
&= [\![p \cap q]\!]\ pk[n/f]{::}h \\
&= [\![p]\!]\ pk[n/f]{::}h \cap [\![q]\!]\ pk[n/f]{::}h
\end{aligned}$$

$$= [\![f \leftarrow n \cdot p]\!]\ pk{::}h \cap [\![f \leftarrow n \cdot q]\!]\ pk{::}h$$

$$= [\![(f \leftarrow n \cdot p) \cap (f \leftarrow n \cdot q)]\!]\ pk{::}h$$

$\square$

### A.1.1 Completeness

To prove completeness of the dup-free NetKAT$(-, \cap)$ axioms, we show that every dup-free NetKAT$(-, \cap)$ term is provably equivalent to a dup-free NetKAT term, and then appeal to the completeness of the NetKAT axioms. To make the induction work, we actually show a stronger theorem: every dup-free NetKAT$(-, \cap)$ term is provably equivalent to a dup-free,$^*$-free reduced NetKAT term.

**Lemma 6.** $(\sum_i \alpha_i \cdot \pi_i) \cap (\sum_j \alpha'_j \cdot \pi'_j) \equiv \sum_k \alpha_k \cdot \pi_k$ *such that* $\forall k \exists i, j$ *s.t.* $\alpha_k = \alpha_i = \alpha'_j$ *and* $\pi_k = \pi_i = \pi'_j$.

**Lemma 7.** $\overline{\alpha \cdot \pi} \equiv (\sum_{\alpha' \neq \alpha} \sum_{\pi'} \alpha' \cdot \pi') + \sum'_{\alpha} \sum_{\pi' \neq \pi} \alpha' \cdot \pi'$ *is provable.*

**Lemma 8.** *Every* dup*-free NetKAT$(-, \cap)$ policy is provably equivalent to a sum of reduced* $^*$*-free,*dup*-free NetKAT policies.*

*Proof.* Proof by structural induction upon the policy $p$, with a case analysis on the last syntax rule.

Because the NetKAT$(-, \cap)$ axioms are a conservative extension of the NetKAT axioms, all cases except $p \cap q$,$\overline{p}$, and $p^*$ follow immediately from the induction hypothesis and fact that every dup-free NetKAT policy is provably equivalent to a sum of dup-free reduced NetKAT policies (Lemma 9 in [4]).

**Case** $p \cap q$ By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$, and $q \equiv \sum_j \alpha'_j \cdot \pi'_j$. By Lemma 6, this is provably equal to a term $\sum_k \alpha_k \cdot \pi_k$, which is in the form desired.

**Case** $\overline{p}$ By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$.

$$\begin{aligned}
\overline{p} &\equiv \overline{\sum_i \alpha_i \cdot \pi_i} \\
&\equiv \prod_i \overline{\alpha_i \cdot \pi_i} && \text{By COMP-PAR} \\
&\equiv \prod_i \left( \sum_j \alpha_{i,j} \cdot \pi_{i,j} \right) && \text{By Lemma 7} \\
&\equiv \sum_k \alpha'_k \cdot \pi'_k && \text{For some } \alpha'_k, \pi'_k \text{ by INTER-PAR-DIST and induction}
\end{aligned}$$

**Case** $p^*$ By the induction hypothesis, $p \equiv \sum_i \alpha_i \cdot \pi_i$. Thus, $p^* \equiv (\sum_i \alpha_i \cdot \pi_i)^*$, which is a dup-free NetKAT term. By a theorem of NetKAT (Lemma 9 in [4]), every dup-free NetKAT term is provably equivalent to a dup-free, *-free reduced NetKAT term.

□

**Theorem 18** (Theorem 5). *The axioms for NetKAT($-, \cap$) shown in Figure 3.9 plus the NetKAT axioms (minus* PA-DUP-FILTER-COMM*) are complete for the dup-free fragment.*

*Proof.* Follows directly from Lemma 8 and the proof of NetKAT completeness (Theorem 2 in [4]). □

### A.1.2 NetKAT($-, \cap$) Derivatives

**Lemma 9** (). $E_{\alpha,\beta}(p) \equiv E(p)(\alpha)(\beta)$

*Proof.* Proof by structural induction upon the term $p$.

**Case** $\pi$

$$\begin{aligned} E_{\alpha,\beta}(\pi) &= [\pi = \pi_\beta] \\ &\equiv [\pi \circ \alpha = \pi_\beta] \\ &\equiv E_\pi(\alpha)(\beta) \\ &\equiv E(\pi)(\alpha)(\beta) \end{aligned}$$

**Case** $b$

$$\begin{aligned} E_{\alpha,\beta}(b) &= [\alpha = \beta \leq b] \\ &\equiv E_b(\alpha)(\beta) \\ &\equiv E(b)(\alpha)(\beta) \end{aligned}$$

**Case** $p + q$

$$\begin{aligned} E_{\alpha,\beta}(p+q) &= E_{\alpha,\beta}(p) + E_{\alpha,\beta}(q) \\ &\equiv E(p)(\alpha)(\beta) + E(q)(\alpha)(\beta) \\ &\equiv E(p) + E(q)(\alpha)(\beta) \\ &\equiv E(p+q)(\alpha)(\beta) \end{aligned}$$

**Case** $p \cap q$

$$\begin{aligned} E_{\alpha,\beta}(p \cap q) &= E_{\alpha,\beta}(p) \cdot E_{\alpha,\beta}(q) \\ &\equiv E(p)(\alpha)(\beta) \cdot E(q)(\alpha)(\beta) \\ &\equiv E(p) \cap E(q)(\alpha)(\beta) \\ &\equiv E(p \cap q)(\alpha)(\beta) \end{aligned}$$

**Case** $p \cdot q$

$$\begin{aligned} E_{\alpha,\beta}(p \cdot q) &= \sum_{\gamma} E_{\alpha,\gamma}(p) \cdot E_{\gamma,\beta}(q) \\ &\equiv \sum_{\gamma} E(p)(\alpha)(\gamma) \cdot E(q)(\gamma)(\beta) \\ &\equiv E(p) \cdot E(q)(\alpha)(\beta) \\ &\equiv E(p \cdot q)(\alpha)(\beta) \end{aligned}$$

**Case** $\overline{p}$

$$\begin{aligned} E_{\alpha,\beta}(\overline{p}) &= \overline{E_{\alpha,\beta}(p)} \\ &\equiv \overline{E(p)(\alpha)(\beta)} \\ &\equiv \overline{E(p)}(\alpha)(\beta) \\ &\equiv E(\overline{p})(\alpha)(\beta) \end{aligned}$$

**Case** $p^*$

$$\begin{aligned} E_{\alpha,\beta}(p^*) &= [\alpha = \beta] + \sum_{\gamma} E_{\alpha,\gamma}(p) \cdot E_{\gamma,\beta}(p^*) \\ &\equiv [\alpha = \beta] + \sum_{\gamma} E(p)(\alpha)(\gamma) \cdot E(p^*)(\gamma)(\beta) \\ &\equiv E(p^*)(\alpha)(\beta) \end{aligned}$$

□

**Lemma 10.**

$$D'_{\alpha,\beta}(p) \equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot d$$

*Proof.* Proof by structural induction upon the term $p$.

**Case** $p = f \leftarrow n$ In this case, $D(p)$ is the empty set, which is equivalent to $D'_{\alpha\beta}(f \leftarrow n) = 0$.

**Case** $p = a$ In this case, $D(p)$ is the empty set, which is equivalent to $D'_{\alpha\beta}(a) = 0$.

**Case** $p = \mathsf{dup}$

$$\begin{aligned}
D'_{\alpha\beta}(\mathsf{dup}) &= [\alpha = \beta] \\
&\equiv E_1(\alpha)(\beta) \\
&\equiv E_1(\alpha)(\beta) \cdot 1 \\
&\equiv \sum_{(e,d)\in\{(E(1),1)\}} [e(\alpha)(\beta)] \cdot d \\
&\equiv \sum_{(e,d)\in D(\mathsf{dup})} [e(\alpha)(\beta)] \cdot d
\end{aligned}$$

**Case** $p = e_1 + e_2$

$$\begin{aligned}
D'_{\alpha\beta}(e_1 + e_2) &= D'_{\alpha\beta}(e_1) + D'_{\alpha\beta}(e_2) \\
&\equiv \left(\sum_{(e,d)\in D(e_1)} [e(\alpha)(\beta)] \cdot d\right) + \left(\sum_{(e,d)\in D(e_2)} [e(\alpha)(\beta)] \cdot d\right) && \text{By induction} \\
&\equiv \sum_{(e,d)\in D(e_1)\cup D(e_2)} [e(\alpha)(\beta)] \cdot d \\
&\equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot d
\end{aligned}$$

**Case** $p = q \cdot r$

$$D'_{\alpha\beta}(e_1 \cdot e_2) = D'_{\alpha\beta}(e_1) \cdot e_2 + \sum_{\gamma} E_{\alpha\gamma}(e_1) \cdot D'_{\gamma\beta}(e_2)$$

$$\equiv \left( \sum_{(e,d) \in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) \cdot e_2 + \sum_{\gamma} E_{\alpha\gamma}(e_1) \cdot \left( \sum_{(e',d') \in D(e_2)} [e'(\gamma)(\beta)] \cdot d' \right)$$

By induction

$$\equiv \left( \sum_{(e,d) \in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) \cdot e_2 + \sum_{\gamma} E(e_1)(\alpha)(\gamma) \cdot \left( \sum_{(e',d') \in D(e_2)} [e'(\gamma)(\beta)] \cdot d' \right)$$

By Lemma 9

$$\equiv \left( \sum_{(e,d) \in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) \cdot e_2 + \sum_{\gamma} \left( \sum_{(e',d') \in D(e_2)} E(e_1)(\alpha)(\gamma) \cdot [e'(\gamma)(\beta)] \cdot d' \right)$$

$$\equiv \left( \sum_{(e,d) \in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) \cdot e_2 + \left( \sum_{(e',d') \in D(e_2)} \sum_{\gamma} E(e_1)(\alpha)(\gamma) \cdot [e'(\gamma)(\beta)] \cdot d' \right)$$

$$\equiv \left( \sum_{(e,d) \in D(e_1)} [e(\alpha)(\beta)] \cdot d \right) \cdot e_2 + \left( \sum_{(e',d') \in D(e_2)} [(E(e_1) \cdot e')(\alpha)(\beta)] \cdot d' \right)$$

$$\equiv \left( \sum_{(e,d) \in D(e_1) \cdot e_2} [e(\alpha)(\beta)] \cdot d \right) + \left( \sum_{(e',d') \in E(e_1) \cdot D(e_2)} [e'(\alpha)(\beta)] \cdot d' \right)$$

$$\equiv \sum_{(e,d) \in D(e_1) \cdot e_2 \cup E(e_1) \cdot D(e_2)} [e(\alpha)(\beta)] \cdot d$$

$$\equiv \sum_{(e,d) \in D(p)} [e(\alpha)(\beta)] \cdot d$$

**Case** $p = q \cap r$

$$D'_{\alpha\beta}(q \cap r) = D'_{\alpha\beta}(q) \cap D'_{\alpha\beta}(r)$$

$$\equiv \left( \sum_{(e_1,d_1) \in D(q)} [e_1(\alpha)(\beta)] \cdot d_1 \right) \cap \left( \sum_{(e_2,d_2) \in D(r)} [e_2(\alpha)(\beta)] \cdot d_2 \right)$$

By induction

$$\equiv \sum_{(e_1,d_1) \in D(q), (e_2,d_2) \in D(r)} ([e_1(\alpha)(\beta)] \cdot d_1) \cap ([e_2(\alpha)(\beta)] \cdot d_2)$$

By PAR-INTER-DIST

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot (d_1 \cap d_2)$$

By INTER-FILTER-DIST-LEFT

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot d_1 \cap d_2$$

By INTER-FILTER-DIST-LEFT

$$\equiv \sum_{(e_1,d_1)\in D(q),(e_2,d_2)\in D(r)} ([e_1(\alpha)(\beta)] \cap [e_2(\alpha)(\beta)]) \cdot (d_1 \cap d_2)$$

By INTER-IDEM

$$\equiv \sum_{(e,d)\in\{(e_1,d_1)\cap(e_2,d_2) \mid (e_1,d_1)\in D(q),(e_2,d_2)\in D(r)\}} [e(\alpha)(\beta)] \cdot d$$

$$\equiv \sum_{(e,d)\in D(p\cap q)} [e(\alpha)(\beta)] \cdot d$$

**Case** $p = q^*$

$$\begin{aligned}
D'_{\alpha\beta}(q^*) &\equiv \sum_{\gamma} E_{\alpha,\gamma}(q^*) \cdot D'_{\gamma,\beta}(q) \cdot q^* \\
&\equiv \sum_{\gamma} E_{\alpha,\gamma}(q^*) \cdot \left( \sum_{(e,d)\in D(q)} [e(\gamma)(\beta)] \cdot d \right) q^* \\
&\equiv \sum_{\gamma} E_{\alpha,\gamma}(q^*) \cdot \left( \sum_{(e,d)\in D(q)} [e(\gamma)(\beta)] \cdot d \cdot q^* \right) \\
&\equiv \sum_{\gamma} \sum_{(e,d)\in D(q)} E_{\alpha,\gamma}(q^*) \cdot [e(\gamma)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_{\gamma} \sum_{(e,d)\in D(q)} E(q^*)(\alpha)(\gamma) \cdot [e(\gamma)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_{(e,d)\in D(q)} [(E(q^*) \cdot e)(\alpha)(\beta)] \cdot d \cdot q^* \\
&\equiv \sum_{(e,d)\in D(q)} E(q^*) \cdot [e(\alpha)(\beta)] \cdot d \cdot q^*
\end{aligned}$$

$$\equiv \sum_{(e,d)\in D(q^*)} [e(\alpha)(\beta)] \cdot d$$

**Case** $p = \overline{q}$

$$\begin{aligned}
D'_{\alpha\beta}(\overline{q}) &= \overline{D'_{\alpha\beta}(q)} \\
&\equiv \overline{\sum_{(e,d)\in D(q)} [e(\alpha)(\beta)] \cdot d} \\
&\equiv \prod_{(e,d)\in D(q)} \overline{[e(\alpha)(\beta)] \cdot d} \\
&\equiv \prod_{(e,d)\in D(q) \text{ s.t. } e(\alpha)(\beta)} \overline{d} \\
&\equiv \sum_{(e,d)\in D(\overline{q})} [e(\alpha)(\beta)] \cdot d
\end{aligned}$$

□

**Lemma 11** (Lemma 2)**.**

$$D_{\alpha,\beta}(p) \equiv \sum_{(e,d)\in D(p)} [e(\alpha)(\beta)] \cdot \beta \cdot d$$

*Proof.* Follows directly from previous lemma. □

**NetKAT$(-,\cap)$ spines** To prove that our equivalence checking algorithm terminates, we need to show that there is some finite bound upon the state space of our automata. In the original NetKAT paper, this was shown by demonstrating a finite basis for the state space, called *spines*. We extend their spine construction to NetKAT$(-,\cap)$, and use this extension to show that the set of derivatives of a term is finite, identifying terms up to associativity, commutativity, and idempotency (ACI) of intersection. Unlike NetKAT spines, extended

**NetKAT$(-,\cap)$ extended spines**

$$\begin{aligned}
espine(a) &\triangleq \emptyset\\
espine(f \leftarrow n) &\triangleq \emptyset\\
espine(p+q) &\triangleq espine(p) \cup espine(q)\\
espine(p \cdot q) &\triangleq \{e \cdot q \mid e \in espine(p)\} \cup espine(q)\\
espine(p^*) &\triangleq \{e \cdot p^* \mid e \in espine(p)\}\\
espine(\mathsf{dup}) &\triangleq \{1\}\\
espine(p \cap q) &\triangleq \{p' \cap q' \mid p' \in espine(p) \wedge q' \in espine(q)\}\\
espine(\overline{p}) &\triangleq \cap^*(\{\overline{q} \mid q \in espine(p))
\end{aligned}$$

spines are not proper subterms of the original term. In fact, the size of the set of extended spines of a term is non-elementary in the size of the original term. Because FDDs are a specific representation of the state space, finiteness of spines implies finiteness of FDD based automata, modulo semantic equivalence of FDDs.

The definition of extended spines (shown in Appendix A.1.2) is not quite as elegant as the original spine definition. NetKAT spines were able to succinctly identify terms up to ACI of $+$ by using a set representation. Because we work with two distinct ACI operations ($+$ and $\cap$), a single layer of sets does not suffice. Instead, we use sets to capture ACI of $+$ (as in the original spines), and implicitly work with intersections up to ACI. In the actual implementation, this is also implemented using a set representation.

In particular, we work with the intersection closure of a finite set of terms up to ACI. To make this precise, we can uniquely define the intersection closure of a set $S$ ($\cap^*(S)$) by taking the powerset of the set of terms, and then sorting each subset and taking the formal intersection of each. But this level of formality is not necessary to understand the development.

**Theorem 19.** $\forall(e,d) \in D(p), d \in espine(p)$

*Proof.* Proof by structural induction upon $p$.

**Case** $\pi$

$$\begin{aligned} D(\pi) &= \emptyset \\ &= espine(\pi) \end{aligned}$$

**Case** $b$

$$\begin{aligned} D(b) &= \emptyset \\ &= espine(b) \end{aligned}$$

**Case** $\mathsf{dup}$

$$\begin{aligned} D(\mathsf{dup}) &= \{(E(\mathsf{1}), \mathsf{1})\} \\ \{\mathsf{1}\} &= espine(\mathsf{dup}) \end{aligned}$$

**Case** $p + q$

$$\begin{aligned} D(p+q) &= D(p) \cup D(q) \\ espine(p+q) &= espine(p) \cup espine(q) \end{aligned}$$

The case follows by induction.

**Case** $p \cap q$

$$\begin{aligned} D(p \cap q) &= \{d_1 \cap d_2 \mid d_1 \in D(p), d_2 \in D(q)\} \\ espine(p \cap q) &= \{e_1 \cap e_2 \mid e_1 \in espine(p), e_2 \in espine(q)\} \end{aligned}$$

The case follows by induction.

**Case** $p \cdot q$

$$D(p \cdot q) = \{d \cdot q \mid d \in D(p)\} \cup E(p) \cdot D(q)$$
$$espine(p \cdot q) = \{e \cdot q \mid e \in espine(p)\} \cup espine(q)$$

The case follows by induction.

**Case** $p^*$

$$D(p^*) = \{d \cdot p \mid d \in D(p)\}$$
$$espine(p^*) = \{e \cdot p \mid e \in espine(p)\}$$

The case follows by induction.

**Case** $\overline{p}$

$$D(\overline{p}) = \bigcup_{\alpha,\beta} \left\{ (E(\alpha \cdot p_\beta), \bigcap_{(e',d') \in D(p) \wedge e'(\alpha)(\beta)} \overline{d'}) \right\}$$
$$espine(\overline{p}) = \cap^*(\{\overline{q} \mid q \in espine(p))$$

The case follows by induction.

$\square$

## A.2 Proofs for Chapter 4

The theorems of this chapter have been formally verified in the Coq theorem prover. We include the proof text here. All proofs have been completed in Coq verion 8.4.

The Coq proofs in this appendix were jointly developed with Arjun Guha. The compiler was based on a system originally built by Arjun Guha and Andrew D. Ferguson.

### A.2.1 Bag Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Common.Types.

Require Export Bag.Bag2Defs.

Require Export Bag.Bag2Tactics.

Require Export Bag.Bag2Notations.

Require Bag.Bag2Lemmas.

Module Bag := Bag.Bag2Lemmas.

Arguments to_list _ _ _ : simpl never.

Hint Rewrite
  Bag.unions_nil
  Bag.unions_cons
  Bag.map_union
  Bag.unions_app
  map_app
  Bag.union_assoc
  Bag.from_list_app
  Bag.from_list_cons
  Bag.union_empty_r
  Bag.union_empty_l
  Bag.from_list_nil_is_empty
  Bag.unions_map_union_comm
  Bag.unions_map_union_comm2
  Bag.unions_map_bag
 : bag.
```

### A.2.2 Bag2Defs Library

```
Set Implicit Arguments.

Require Import Coq.Logic.ProofIrrelevance.

Require Import Coq.Relations.Relations.

Require Import Coq.Lists.List.

Require Import Bag.TotalOrder.

Require Import Bag.OrderedLists.

Import ListNotations.

Local Open Scope list_scope.

Record bag {A : Type} (R : relation A) : Type := Bag {
  to_list : list A;
  order : Ordered R to_list
}.

Arguments Bag {A R} to_list order.

Section Definitions.

  Variable A : Type.

  Variable R : relation A.

  Variable Order : TotalOrder R.

  Lemma singleton_ordered : ∀ (x : A), Ordered R [x].

  Proof.
  intros.
  apply Ordered_cons.
  intros.
  simpl in H.
  inversion H.
  apply Ordered_nil.
Qed.

Definition singleton (x : A) := Bag [x] (singleton_ordered x).

Definition union (b1 b2 : bag R) :=
  Bag (union (to_list b1) (to_list b2))
      (union_order_pres Order (order b1) (order b2)).

Definition from_list (lst : list A) :=
  Bag (from_list lst) (from_list_order Order lst).

Lemma unions_order_pres : ∀ (bags : list (bag R)),
  Ordered R (unions (map (@to_list A R) bags)).
Proof with auto.
  intros.
  apply unions_order_pres.
  intros.
  rewrite → in_map_iff in H.
  destruct H as [bag [Heq HIn]].
  destruct bag.
  simpl in Heq.
  subst...
Qed.
```

```
  Definition unions (bags : list (bag R)) :=
    Bag (unions (map (@to_list A R) bags)) (unions_order_pres bags).

  Definition empty := Bag [] (Ordered_nil R).

End Definitions.

Arguments singleton [A R] x.

Arguments union [A R Order] b1 b2.

Arguments from_list [A R Order] lst.

Arguments empty [A R].

Arguments unions [A R Order] bags.
```

### A.2.3 Bag2Lemmas Library

```
Set Implicit Arguments.

Require Import Coq.Logic.ProofIrrelevance.

Require Import Coq.Lists.List.

Require Import Coq.Relations.Relations.

Require Import Bag.TotalOrder.

Require Import Bag.OrderedLists.

Require Import Bag.Bag2Defs.

Require Import Bag.Bag2Notations.

Import ListNotations.

Local Open Scope list_scope.

Local Open Scope list_scope.

Local Open Scope bag_scope.
```

```
Module OL := Bag.OrderedLists.

Section Methods.

  Variable A : Type.

  Variable R : relation A.

  Variable Order : TotalOrder R.

  Lemma ordered_irr: ∀ (b : bag R) (lst : list A) (o : Ordered R lst),
    to_list b = lst →
    b = Bag lst o.

  Proof with auto.
    intros.
    destruct b.
    simpl in *.
    subst.
    assert (o = order).
      apply proof_irrelevance.
    subst...
  Qed.

  Hint Resolve ordered_irr.

  Lemma union_comm : ∀ b1 b2, b1 <+> b2 = b2 <+> b1.

  Proof with auto.
    intros.
    apply ordered_irr.
    simpl.
    apply union_comm...
    destruct b1...
    destruct b2...
  Qed.
  Lemma union_assoc : ∀ x y z, (x <+> y) <+> z = x <+> (y <+> z).
  Proof with auto.
    intros.
    apply ordered_irr.
    simpl.
    symmetry.
    apply union_assoc...
    destruct x...
    destruct y...
    destruct z...
  Qed.
  Lemma union_empty_l : ∀ x, empty <+> x = x.
  Proof with auto.
    intros.
    destruct x.
    apply ordered_irr.
    simpl.
    apply union_nil_l...
  Qed.
  Lemma union_empty_r : ∀ x, x <+> empty = x.
  Proof.
    intros.
    destruct x; auto.
```

```
Qed.

Lemma unions_cons : ∀ (x : bag R) (xs : list (bag R)),
  unions (x :: xs) = x <+> unions xs.
Proof with auto.
  intros.
  apply ordered_irr...
Qed.

Lemma unions_app : ∀ (lst lst0 : list (bag R)),
  unions (lst ++ lst0) = unions lst <+> unions lst0.
Proof with auto.
  intros.
  apply ordered_irr.
  simpl.
  induction lst...
  simpl.
  rewrite → union_nil_l...
  apply unions_order_pres.
  simpl.
  rewrite ← OL.union_assoc...
  rewrite → IHlst...
  destruct a...
  apply unions_order_pres...
  apply unions_order_pres...
Qed.

Lemma pop_union_r : ∀ (b b0 b1: bag R),
  b0 = b1 ↔
  b0 <+> b = b1 <+> b.
Proof with simpl; auto with datatypes.
  split; intros.
  + subst. reflexivity.
  + destruct b,b0,b1.
    rename to_list into lst,to_list0 into lst0,to_list1 into lst1.
    inversion H.
    induction lst...
    simpl in H1.
    inversion order; subst.
    apply insert_eq in H1...
    apply IHlst with (order:=H4) in H1...
    apply ordered_irr. simpl...
    apply union_order_pres...
    apply union_order_pres...
Qed.

Lemma pop_union_l : ∀ (b b0 b1: bag R),
  b0 = b1 ↔
  b <+> b0 = b <+> b1.
Proof with auto.
  intros.
  do 2 rewrite → (union_comm b).
  apply pop_union_r...
Qed.
```

```
Lemma rotate_union : ∀ (b b0 b1 : bag R),
  union b b0 = b1 →
  union b0 b = b1.
Proof.
  intros. subst. apply union_comm.
Qed.

Lemma from_list_cons : ∀ x xs,
  from_list (x :: xs) = ({| x |}) <+> (from_list xs).
Proof with auto.
  intros.
  apply ordered_irr.
  simpl.
  apply from_list_cons.
Qed.

Lemma from_list_app : ∀ lst1 lst2,
  from_list (lst1 ++ lst2) = union (from_list lst1) (from_list lst2).
Proof with auto.
  intros.
  apply ordered_irr.
  simpl.
  apply from_list_app.
Qed.

Lemma from_list_nil_is_empty : from_list nil = empty.
Proof with auto.
  intros.
  apply ordered_irr.
  simpl...
Qed.
Lemma in_union : ∀ (x : A) (b1 b2 : bag R),
  In x (to_list (b1 <+> b2)) ↔
  In x (to_list b1) ∨ In x (to_list b2).
Proof with auto.
  intros.
  split; intros.
  × simpl in H.
    destruct b1.
    destruct b2.
    simpl in H.
    apply In_union in H...
  × destruct H.
    destruct b1; destruct b2; simpl.
    apply In_union...
    apply In_union...
    destruct b1...
    destruct b2...
Qed.
Lemma unions_nil : unions nil = empty.
Proof with auto.
  intros.
  apply ordered_irr...
```

Qed.

Lemma *to_list_nil* : ∀ (*b* : *bag R*), *to_list b* = *nil* → *b* = *empty*.

Proof with auto.

  intros.

  apply *ordered_irr*...

Qed.

Lemma *in_split* : ∀ *x bag*,

  *In x* (*to_list bag*) →

  ∃ *rest*,

    *bag* = (*union* (*singleton x*) *rest*).

Proof with auto with *datatypes*.

  intros.

  destruct *bag*.

  simpl in *H*.

  rename *to_list into lst*.

  induction *lst*...

  + simpl in *H*. inversion *H*.

  + simpl in *H*.

    inversion *order*; subst.

    destruct *H*.

    - subst...

      ∃ (*Bag lst H3*).

      apply *ordered_irr*.

      simpl.

      symmetry.

```
      apply union_cons...
    - apply IHlst with (order := H3) in H.
      destruct H as [rest H].
      ∃ (({|a|}) <+> rest).
      apply ordered_irr.
      simpl.
      assert (Ordered R [x]). { apply Ordered_cons... intros. simpl in H0... inversion
H0. apply Ordered_nil. }
      assert (Ordered R [a]). { apply Ordered_cons... intros. simpl in H1... inversion
H1. apply Ordered_nil. }
      rewrite → OL.union_assoc...
      rewrite → (OL.union_comm Order H0 H1).
      rewrite ← OL.union_assoc...
      unfold union in H.
      simpl in H.
      inversion H.
      rewrite ← H5.
      symmetry.
      apply union_cons...
      destruct rest...
      destruct rest...
Qed.

Lemma in_unions : ∀ (x : A) (lst : list (bag R)),
  In x (to_list (unions lst)) →
  ∃ bag,
    In bag lst ∧ In x (to_list bag).
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl in H.
  apply OL.In_union in H.
  + destruct H.
    - destruct a.
      simpl in H...
    - apply IHlst in H.
      destruct H as [bag HIn].
      destruct HIn...
  + destruct a...
  + apply unions_order_pres.
Qed.
Lemma in_to_from_list : ∀ x lst,
  In x (to_list (from_list lst)) →
  In x lst.
Proof with auto with datatypes.
  intros.
  induction lst...
  simpl in H.
  apply In_insert in H.
  destruct H...
  subst...
```

```
Qed.

Lemma singleton_eq_singleton : ∀ x y lst,
  ({|x|}) = ({|y|}) <+> lst → lst = empty ∧ x = y.
Proof with auto with datatypes.
  intros.
  inversion H.
  rewrite → OL.union_comm in H1.
  simpl in H1.
  destruct lst.
  rename to_list into lst.
  simpl in H1.
  destruct lst...
  + simpl in H1.
    inversion H1.
    subst.
    split...
    apply ordered_irr...
  + simpl in H1.
    destruct (compare y a).
    - inversion H1.
    - inversion H1.
      subst.
      destruct lst...
      simpl in H3...
      inversion H3.
      simpl in H3.
      destruct (compare y a0)...
      inversion H3.
      inversion H3.
 + apply Ordered_cons. intros. inversion H0. apply Ordered_nil.
 + destruct lst...
Qed.

Lemma singleton_union_disjoint : ∀ x y b1 b2,
  ({|x|} <+> b1) = ({|y|} <+> b2) →
  (In x (to_list b2) → False) →
  x = y ∧ b1 = b2.
Proof with auto with datatypes.
  intros.
  assert (In x (to_list ({|y|} <+> b2))) as J.
  { rewrite ← H. apply in_union; simpl... }
  apply in_union in J; simpl in J.
  destruct J as [[J | J] | J].
  + subst.
    apply pop_union_l in H...
  + inversion J.
  + contradiction J.
Qed.

Lemma union_from_ordered : ∀ b1 b2 b3 b4,
  OL.union (to_list b1) (to_list b2) = OL.union (to_list b3) (to_list b4) →
  (b1 <+> b2) = (b3 <+> b4).
  Proof with auto with datatypes.
    intros.
    destruct b1, b2, b3, b4.
    simpl in H.
    unfold union.
    simpl.
    apply ordered_irr...
  Qed.
End Methods.
Section BinaryMethods.
  Variable A B: Type.
  Variable RA : relation A.
  Variable RB : relation B.
  Variable AOrder : TotalOrder RA.
  Variable BOrder : TotalOrder RB.
  Lemma map_union : ∀ (f : A → B) (bag1 bag2 : bag RA),
    from_list (map f (to_list (union bag1 bag2))) =
      (union (from_list (map f (to_list bag1)))
        (from_list (map f (to_list bag2)))).
  Proof with auto.
    intros.
    apply ordered_irr.
    simpl.
    apply map_union...
    destruct bag1...
  destruct bag2...
Qed.

Lemma in_unions_map : ∀ (b : B) (lst: list A) (f : A → bag RB),
 In b (to_list (unions (map f lst))) →
  ∃ (a : A), In a lst ∧ In b (to_list (f a)).
Proof with eauto with datatypes.
  intros.
  induction lst.
  + simpl in H. inversion H.
  + simpl in H.
    apply In_union in H.
    destruct H...
    apply IHlst in H.
    clear IHlst.
    destruct H as [a0 [Ha0In HbIn]].
    ∃ a0...
    destruct (f a)...
    apply unions_order_pres...
Qed.

Lemma unions_map_insert_comm : ∀ (x : A) (xs : list A)
  (f : A → bag RB) ,
  Ordered RA xs →
  unions (map f (insert x xs)) = (f x) <+> unions (map f xs).
Proof with auto.
  intros.
  induction xs...
  apply ordered_irr...
  simpl.
  destruct (compare x a).
  simpl.
  rewrite → unions_cons...
  simpl.
  rewrite → unions_cons...
  rewrite → unions_cons...
  rewrite ← union_assoc.
  assert (f x <+> f a = f a <+> f x) by apply union_comm.
  rewrite → H0.
  rewrite → union_assoc.
  rewrite → IHxs...
  inversion H...
Qed.

Lemma unions_map_union_comm : ∀ (x : A) (xs : bag RA)
  (f : A → bag RB),
  unions (map f (to_list ((singleton x) <+> xs))) =
  (f x) <+> unions (map f (to_list xs)).
Proof with eauto with datatypes.
  intros.
  destruct xs.
  induction to_list...
  simpl.
  apply ordered_irr...
  simpl in *.
  inversion order; subst.
  apply IHto_list in H2; clear IHto_list.
  rewrite → unions_map_insert_comm.
  rewrite → unions_cons.
  rewrite → H2.
  assert (f x <+> f a = f a <+> f x) by apply union_comm.
  rewrite ← union_assoc.
  rewrite ← H.
  rewrite → union_assoc...
  apply union_order_pres...
  apply Ordered_cons...
  intros.
  simpl in H.
  inversion H.
  apply Ordered_nil.
  inversion order...
Qed.

Lemma unions_map_union_comm2 : ∀ (lst0 lst1 : bag RA)
  (f : A → bag RB),
  unions (map f (to_list (lst0 <+> lst1))) =
  unions (map f (to_list lst0)) <+> unions (map f (to_list lst1)).
Proof with eauto with datatypes.
  intros.
  destruct lst1.
  induction to_list...
  simpl.
  apply ordered_irr...
  simpl in *.
  inversion order; subst.
  apply IHto_list in H2; clear IHto_list.
  rewrite → unions_map_insert_comm.
  rewrite → unions_cons.
  rewrite → H2.
  rewrite ← union_assoc.
  rewrite → (union_comm _ (f a)).
  rewrite → union_assoc...
  apply union_order_pres...
  destruct lst0...
  inversion order...
Qed.

Lemma unions_map_bag : ∀ (lst : list A) (f : A → bag RB),
  unions (map f (to_list (from_list lst))) = unions (map f lst).
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl.
  rewrite → unions_cons.
  rewrite → unions_map_insert_comm.
```

```
    simpl in IHlst.
    rewrite → IHlst...
    apply from_list_order.
  Qed.

  Require Import Common.AllDiff.

  Lemma AllDiff_preservation : ∀ (f : A → B) x y lst,
      AllDiff f (to_list (({|x|}) <+> lst)) →
      f x = f y →
      AllDiff f (to_list (({|y|}) <+> lst)).
  Proof with auto with datatypes.
    intros.
    destruct lst.
    rename to_list into lst.
    unfold singleton in *.
    unfold to_list in *.
    simpl in *.
    apply OrderedLists.AllDiff_preservation with (x:=x)...
  Qed.
End BinaryMethods.
```

### A.2.4 Bag2Notations Library

```
Require Import Bag.Bag2Defs.

Reserved Notation "{| x |}" (at level 70, no associativity).

Reserved Notation "{| |}" (at level 70, no associativity).
Reserved Notation "x <+> y" (at level 69, right associativity).

Notation "x <+> y" := (union x y) : bag_scope.

Notation "{| x |}" := (singleton x) : bag_scope.

Notation "{| |}" := (empty) : bag_scope.
```

### A.2.5 Bag2Tactics Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Coq.Relations.Relations.

Require Import Bag.TotalOrder.

Require Import Bag.Bag2Defs.

Require Import Bag.Bag2Lemmas.

Require Import Bag.Bag2Notations.

Local Open Scope list_scope.

Local Open Scope bag_scope.

Ltac bag_perm n :=

  match goal with

    | ⊢ ?bag = ?bag ⇒

      reflexivity

    | ⊢ ?b <+> ?lst = ?b <+> ?lst0 ⇒

      apply pop_union_l;

          bag_perm (pred n)

    | ⊢ ?b <+> ?lst1 = ?lst2 ⇒

      match eval compute in n with
```

```
          | O ⇒ fail "out of time / not equivalent"
          | S _ ⇒
            apply rotate_union;
              repeat rewrite → union_assoc;
                bag_perm (pred n)
      end
  end.
Ltac solve_bag_permutation :=
  bag_perm 100.
Module Examples.
  Variable A : Type.
  Variable R : relation A.
  Variable O : TotalOrder R.
  Variable b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 : bag R.
  Example test_identity : b0 <+> b1 <+> b2 = b0 <+> b1 <+> b2.
  Proof.
    solve_bag_permutation.
  Qed.
  Example test_rotate : b0 <+> b1 <+> b2 = b1 <+> b2 <+> b0.
  Proof.
    solve_bag_permutation.
  Qed.
  Example test3 : b0 <+> b1 <+> b2 = b1 <+> b0 <+> b2.
  Proof.
    solve_bag_permutation.
  Qed.

  Example test4 :
    b3 <+> b0 <+> b5 <+> b1 <+> b4 <+> b2 <+> b6 =
    b1 <+> b4 <+> b5 <+> b6 <+> b3 <+> b0 <+> b2.
  Proof.
    bag_perm 20.
  Qed.

  Example test_termination1 : False → b0 <+> b2 = b1 <+> b2.
  Proof.
    intros.
    try solve [clear H; bag_perm 10].
    inversion H.   Qed.

  Example test_termination2 :
    b3 <+> b0 <+> b5 <+> b1 <+> b4 <+> b2 <+> b6 =
    b1 <+> b4 <+> b5 <+> b6 <+> b3 <+> b0 <+> b2.
  Proof.
    try solve [bag_perm 10].
    bag_perm 20.   Qed.

End Examples.
```

### A.2.6 OrderedLists Library

```
Set Implicit Arguments.

Require Import Coq.Relations.Relations.

Require Import Coq.Lists.List.
Require Import Bag.TotalOrder.

Import ListNotations.

Local Open Scope list_scope.

Inductive Ordered (A : Type) (R : relation A) : list A → Prop :=
| Ordered_nil : Ordered R nil
| Ordered_cons : ∀ x xs,
    (∀ y, In y xs → R x y) →
    Ordered R xs →
    Ordered R (x :: xs).

Section Definitions.

  Variable A : Type.

  Variable R : relation A.

  Variable Order : TotalOrder R.

  Fixpoint insert (x : A) (b : list A) : list A :=
    match b with
      | nil ⇒ [x]
      | y :: ys ⇒
        match compare x y with
          | left _ ⇒ x :: y :: ys
          | right _ ⇒ y :: insert x ys
        end
    end.

  Definition union (b1 b2 : list A) : list A := fold_right insert b1 b2.

  Definition from_list (lst : list A) : list A := fold_right insert nil lst.
  Definition unions (lsts : list (list A)) : list A := fold_right union nil lsts.

End Definitions.

Arguments insert [A R Order] x b.

Arguments union [A R Order] b1 b2.

Arguments from_list [A R Order] lst.

Arguments unions [A R Order] lsts.

Section Lemmas.

  Variable A : Type.

  Variable R : relation A.

  Variable Order : TotalOrder R.

  Hint Constructors Ordered.

  Lemma insert_in : ∀ (x y : A) (b : list A),

    In x (insert y b) →

    x = y ∨ In x b.

  Proof with auto with datatypes.

    intros.

    induction b...

    simpl in H...

    destruct H...

    simpl in H.

    remember (compare y a) as cmp.

    destruct cmp.

    simpl.

    simpl in H.

    destruct H as [H | [H0 | H1]]...
    simpl in H.
    destruct H.
    subst.
    right...
    apply IHb in H.
    destruct H...
  Qed.

  Lemma insert_order_pres : ∀ (x : A) (b : list A),
    Ordered R b →
    Ordered R (insert x b).
  Proof with eauto.
    intros.
    induction b...
    simpl...
    apply Ordered_cons...
    intros.
    inversion H0.
    simpl.
    remember (compare x a) as cmp.
    destruct cmp.
    apply Ordered_cons...
    intros.
    simpl in H0.
    destruct H0.
    subst...
    inversion H.
    subst.
    eapply transitivity...
    apply Ordered_cons...
    intros.
    apply insert_in in H0.
    destruct H0.
    subst...
    inversion H...
    subst...
    inversion H...
  Qed.

  Hint Resolve insert_order_pres.

  Lemma singleton_order : ∀ (x : A),
    Ordered R [x].
  Proof.
    intros. apply Ordered_cons. intros. simpl in H; inversion H.
    apply Ordered_nil.
  Qed.

  Lemma union_order_pres : ∀ (b1 b2 : list A),
    Ordered R b1 →
    Ordered R b2 →
    Ordered R (union b1 b2).
  Proof with auto with datatypes.
    intros.
    induction b2...
    simpl.
    inversion H0; subst...
  Qed.

  Hint Resolve union_order_pres.

  Lemma unions_order_pres : ∀ (lsts : list (list A)),
    (∀ (b : list A), In b lsts → Ordered R b) →
    Ordered R (unions lsts).
  Proof with simpl;auto with datatypes.
    intros.
    induction lsts...
  Qed.

  Hint Resolve unions_order_pres union_order_pres.

  Hint Resolve antisymmetry transitivity.

  Lemma insert_eq_head : ∀ x b,
    Ordered R b →
    (∀ y, In y b → R x y) →
    insert x b = x :: b.
  Proof with eauto with datatypes.
    intros.
    induction b...
    simpl.
    destruct (compare x a)...
    assert (x = a)...
    subst.
    rewrite → IHb...
    inversion H...
  Qed.

  Hint Resolve insert_eq_head.

  Lemma insert_comm : ∀ x y b,
    Ordered R b →
    insert x (insert y b) = insert y (insert x b).
  Proof with eauto with datatypes.
    intros.
    induction H...
    + simpl.
      destruct (compare x y); destruct (compare y x)...
      assert (x = y)...
      subst...
      assert (x = y)...
      subst...
    + simpl.
      remember (compare y x0) as cmp0.
      remember (compare x x0) as cmp1.
      destruct cmp0; destruct cmp1.
      simpl.
      remember (compare x y) as cmp2.
      rewrite ← Heqcmp1.
      rewrite ← Heqcmp0.
      remember (compare y x) as cmp3.
```

```
destruct cmp2;
destruct cmp3...
assert (x = y)...
subst...
assert (x = y)...
subst...
simpl.
rewrite ← Heqcmp0.
rewrite ← Heqcmp1.
remember (compare x y) as cmp2.
destruct cmp2...
assert (x = x0)...
subst.
assert (x0 = y)...
subst.
destruct xs...
simpl.
remember (compare y a) as cmp0.
destruct cmp0...
assert (a = y)...
subst.
rewrite → insert_eq_head...
inversion H0...
simpl.
rewrite ← Heqcmp1...
rewrite ← Heqcmp0...
```

```
    remember (compare y x) as cmp2.
    destruct cmp2...
    assert (x = x0)...
    assert (x0 = y)...
    subst.
    destruct xs...
    simpl.
    remember (compare y a) as cmp0.
    destruct cmp0...
    assert (a = y)...
    subst.
    rewrite → insert_eq_head...
    inversion H0...
    simpl.
    rewrite ← Heqcmp0...
    rewrite ← Heqcmp1...
    rewrite → IHOrdered...
Qed.

Hint Resolve insert_comm.

Lemma union_nil_l : ∀ b,
  Ordered R b →
  union nil b = b.
Proof with auto.
  intros.
  unfold union.
  induction b...
  simpl.
  inversion H.
  subst.
  rewrite → IHb...
Qed.

Hint Resolve union_nil_l.

Lemma union_cons_insert : ∀ a b1 b2,
  Ordered R (a :: b1) →
  Ordered R b2 →
  union (a :: b1) b2 = insert a (union b1 b2).
Proof with auto with datatypes.
  intros.
  generalize dependent b1.
  induction b2; intros.
  simpl.
  symmetry.
  inversion H.
  subst.
  apply insert_eq_head...
  simpl.
  inversion H.
  inversion H0.
  subst.
  rewrite → IHb2...
```

```
Qed.

Lemma insert_nonempty : ∀ x xs,
  insert x xs = nil → False.
Proof with auto with datatypes.
  intros.
  destruct xs...
  simpl in H.
  inversion H.
  simpl in H.
  destruct (compare x a).
  inversion H.
  inversion H.
Qed.

Lemma insert_eq : ∀ x lst0 lst1,
  Ordered R lst0 →
  Ordered R lst1 →
  insert x lst0 = insert x lst1 → lst0 = lst1.
Proof with auto with datatypes.
  intros.
  generalize dependent lst0.
  induction lst1; intros.
  + simpl in H1.
    destruct lst0...
    simpl in H1.
    destruct (compare x a).
    inversion H1.
    inversion H1.
    apply insert_nonempty in H4.
    inversion H4.
  + destruct lst0.
    simpl in H1.
    destruct (compare x a).
    inversion H1.
    inversion H1.
    symmetry in H4. apply insert_nonempty in H4. inversion H4.
    simpl in H1.
    { destruct (compare x a), (compare x a0).
      + inversion H1...
      + inversion H1; subst.
        inversion H1; subst.
        assert (R a x).
        { destruct lst0.
          simpl in H4. inversion H4; subst...
          simpl in H4.
          destruct (compare x a0).
          inversion H4... apply reflexivity.
          inversion H4; subst... }
        assert (a = x).
        { apply antisymmetry... }
        subst.
        symmetry.
```

```
        inversion H; subst.
        apply insert_eq_head...
      + inversion H1; subst; clear H1.
        rewrite → H4.
        inversion H0; subst.
        apply insert_eq_head...
      + inversion H1; subst; clear H1.
        f_equal.
        inversion H0; inversion H; subst.
        apply IHlst1...
    }
Qed.

Hint Resolve union_cons_insert.

Lemma In_insert : ∀ x y b,
  In x (insert y b) ↔
  x = y ∨ In x b.
Proof with eauto with datatypes.
  intros.
  split; intros.
  + induction b...
    simpl in H.
    destruct H...
    simpl in H.
    remember (compare y a) as cmp0.
    destruct cmp0.
    simpl in H.
    destruct H...
    simpl in H.
    destruct H...
    subst.
    right...
    apply IHb in H.
    destruct H...
  + destruct H.
    × subst...
      induction b...
      simpl...
      simpl...
      destruct (compare y a)...
    × induction b...
      simpl...
      simpl.
      destruct (compare y a)...
      simpl in *.
      destruct H...
Qed.
```

```
Lemma union_insert_l_comm : ∀ a b1 b2,
  Ordered R b1 →
  Ordered R b2 →
  union (insert a b1) b2 = insert a (union b1 b2).
```

```
Proof with subst; eauto with datatypes.
  intros.
  induction b1...
  simpl.
  rewrite → union_nil_l...
  unfold union.
  induction b2...
  inversion H0; subst.
  simpl.
  remember (compare a a0) as cmp0.
  destruct cmp0...
  rewrite → IHb2...
  assert (insert a b2 = a :: b2)...
  rewrite → H1.
  simpl.
  remember (compare a0 a) as cmp1.
  destruct cmp1...
  assert (a0 = a)...
  rewrite → insert_eq_head...
  inversion H0...
  rewrite → IHb2...
  rewrite → insert_eq_head...
  intros.
  apply In_insert in H1...
  destruct H1...
  inversion H...
  rewrite → union_cons_insert...
  rewrite → insert_comm...
  remember (compare a a0) as cmp0.
  destruct cmp0...
  simpl.
  rewrite ← Heqcmp0...
  rewrite → insert_comm...
  rewrite → union_cons_insert...
  rewrite → union_cons_insert...
  apply Ordered_cons...
  intros.
  simpl in H1.
  destruct H1...
  simpl.
  rewrite ← Heqcmp0...
  rewrite → union_cons_insert...
  rewrite → IHb1...
  apply Ordered_cons...
  intros...
  apply In_insert in H1.
  destruct H1...
Qed.

Lemma union_insert_r_comm : ∀ a b1 b2,
  Ordered R b1 →
  Ordered R b2 →
  union b1 (insert a b2) = insert a (union b1 b2).
Proof with eauto with datatypes.
  intros.
  induction b1...
  simpl.
  rewrite → union_nil_l...
  rewrite → union_nil_l...
  inversion H. subst.
  rewrite → union_cons_insert...
  rewrite → union_cons_insert...
  rewrite → IHb1...
Qed.
Lemma union_assoc : ∀ b1 b2 b3,
  Ordered R b1 →
  Ordered R b2 →
  Ordered R b3 →
  union b1 (union b2 b3) = union (union b1 b2) b3.
Proof with auto with datatypes.
  intros.
  induction b2...
  + rewrite → union_nil_l...
  + simpl.
    inversion H0.
    subst.
    rewrite → union_insert_l_comm...
    rewrite → union_cons_insert...
    rewrite → union_insert_r_comm...
    rewrite → IHb2...
Qed.

Lemma union_comm : ∀ b1 b2,
  Ordered R b1 →
  Ordered R b2 →
  union b1 b2 = union b2 b1.
Proof with auto with datatypes.
  intros.
  induction b2...
  + simpl...
    rewrite → union_nil_l...
  + simpl...
    rewrite → union_cons_insert...
    rewrite → IHb2...
    inversion H0...
Qed.

Lemma from_list_order : ∀ (lst : list A),
  Ordered R (from_list lst).
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl...
  simpl...
```

```
Qed.

Hint Resolve from_list_order.

Lemma unions_app : ∀ (lst lst0 : list (list A)),
  (∀ (b : list A), In b lst → Ordered R b) →
  (∀ (b : list A), In b lst0 → Ordered R b) →
  unions (lst ++ lst0) = union (unions lst) (unions lst0).
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl.
  rewrite → union_nil_l...
  simpl.
  rewrite ← union_assoc...
  rewrite → IHlst...
Qed.

Lemma from_list_app : ∀ (lst lst0 : list A),
  from_list (lst ++ lst0) = union (from_list lst) (from_list lst0).
Proof with auto.
  intros.
  simpl.
  induction lst.
  simpl.
  rewrite → union_nil_l...
  simpl.
  rewrite → IHlst.
  rewrite → union_insert_l_comm...
Qed.

Lemma In_union : ∀ (x : A) (b1 b2 : list A),
  Ordered R b1 →
  Ordered R b2 →
  (In x (union b1 b2) ↔
   In x b1 ∨ In x b2).
Proof with auto with datatypes.
  intros.
  split; intros.
  + induction b1...
    rewrite → union_nil_l in H1...
    rewrite → union_cons_insert in H1...
    apply In_insert in H1...
    destruct H1.
    × subst...
    × apply IHb1 in H1...
      destruct H1...
      inversion H...
  + destruct H1...
    × induction b2...
      simpl.
      rewrite → In_insert.
      inversion H0; subst.
      apply IHb2 in H5...
      × induction b2...
        simpl in H1. inversion H1.
        simpl.
        apply In_insert.
        simpl in H1.
        destruct H1...
        apply IHb2 in H1...
        inversion H0...
Qed.

Lemma In_unions : ∀ (x : A) lst,
  (∀ (b : list A), In b lst → Ordered R b) →
  In x (unions lst) →
  ∃ elt, In elt lst ∧ In x elt.
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl in H0.
  apply In_union in H0...
  destruct H0...
  apply IHlst in H0.
  destruct H0 as [elt [HeltIn HxIn]]...
  intros.
  apply H...
Qed.

Lemma union_singleton_l : ∀ x xs,
  Ordered R xs →
  insert x xs = union [x] xs.
Proof with auto with datatypes.
  intros.
  simpl...
  rewrite → union_comm...
  apply Ordered_cons...
  intros.
  simpl in H0.
  inversion H0.
Qed.

Lemma from_list_cons : ∀ x xs,
  insert x (from_list xs) = union [x] (from_list xs).
Proof with auto with datatypes.
  intros.
  apply union_singleton_l.
  apply from_list_order.
Qed.

Lemma in_from_list_iff : ∀ x lst, In x lst ↔ In x (from_list lst).
Proof with auto with datatypes.
  split; intros.
  + induction lst...
    simpl in H.
    destruct H.
    - subst. simpl. apply In_insert...
    - simpl. apply In_insert...
  + induction lst...
    simpl in H.
    apply In_insert in H.
    destruct H...
    subst...
Qed.

Lemma from_list_id : ∀ lst,
  Ordered R lst →
  from_list lst = lst.
Proof with auto with datatypes.
  intros.
  induction lst...
  inversion H; subst.
  simpl.
  remember (from_list lst) as lst0.
  destruct lst0...
  + simpl. rewrite → IHlst...
  + simpl.
    destruct (compare a a0).
    rewrite → IHlst...
    assert (a = a0).
    apply antisymmetry...
    apply H2.
    apply in_from_list_iff.
    rewrite ← Heqlst0...
    subst...
    f_equal.
    rewrite ← IHlst...
    assert (Ordered R (a0 :: lst0)).
    { rewrite → Heqlst0... }
    inversion H0...
Qed.

Lemma union_cons : ∀ x xs,
  Ordered R xs →
  (∀ y, In y xs → R x y) →
  union [x] xs = x :: xs.
Proof with auto with datatypes.
  intros.
  induction xs...
  simpl.
  inversion H; subst.
  rewrite → IHxs...
  simpl.
  destruct (compare a x)...
  + assert (a = x).
    apply antisymmetry...
    subst...
  + f_equal...
Qed.
```

```
Lemma in_inserted : ∀ x lst, In x (insert x lst).
Proof with auto with datatypes.
  intros.
  induction lst...
  simpl...
  simpl.
  destruct (compare x a)...
Qed.
Lemma in_inserted_tail : ∀ x y lst,
  In x lst →
  In x (insert y lst).
Proof with auto with datatypes.
  intros.
  induction lst...
  inversion H.
  simpl in H.
  destruct H; subst.
  + simpl.
    destruct (compare y x)...
  + simpl.
    destruct (compare y a)...
Qed.
Lemma in_split : ∀ x lst,
  Ordered R lst →
  In x lst →
```

```
∃ rest,
   Ordered R rest ∧
   lst = union [x] rest.
Proof with auto with datatypes.
  intros.
  induction lst...
  simpl in H0.
  inversion H0.
  simpl in H0.
  destruct H0; subst...
  + ∃ lst.
    inversion H; subst.
    rewrite ← union_singleton_l...
    simpl.
    split...
    symmetry.
    apply insert_eq_head...
  + inversion H; subst.
    apply IHlst in H0...
    destruct H0 as [rest [HOrderedRest Heq]].
    ∃ (union [a] rest).
    split...
    - apply union_order_pres...
      apply Ordered_cons... intros. simpl in H0. inversion H0.
    - rewrite ← insert_eq_head...
      rewrite → Heq.
      rewrite ← union_singleton_l...

      rewrite ← union_singleton_l...

      rewrite ← union_singleton_l...

      apply union_order_pres...

      apply Ordered_cons... intros. simpl in H0. inversion H0.

  Qed.

End Lemmas.

Section BinaryLemmas.

  Variable A : Type.

  Variable R : relation A.

  Variable Order : TotalOrder R.

  Variable B: Type.

  Variable S : relation B.

  Variable P : TotalOrder S.

  Lemma from_list_map_cons : ∀ (f : A → B) x xs,

    Ordered R xs →

    from_list (map f (insert x xs)) =

    from_list (map f (x :: xs)).

  Proof with auto with datatypes.

    intros.

    induction xs...

    simpl.

    destruct (compare x a)...

    simpl.

    rewrite → insert_comm.
  2: apply from_list_order.
  inversion H; subst.
  apply IHxs in H3; clear IHxs.
  rewrite → H3.
  simpl...
Qed.

Lemma map_union : ∀ (f : A → B) (xs ys : list A),
  Ordered R xs →
  Ordered R ys →
  from_list (map f (union xs ys)) =
  union (from_list (map f xs)) (from_list (map f ys)).
Proof with auto with datatypes.
  intros.
  induction ys...
  simpl.
  inversion H0.
  subst.
  apply IHys in H4; clear IHys.
  rewrite → union_insert_r_comm.
  rewrite ← H4.
  rewrite → from_list_map_cons.
  reflexivity.
  apply union_order_pres...
  inversion H0...
  apply from_list_order.
  apply from_list_order.
Qed.

Lemma in_unions_map : ∀ (b : B) (lst: list A) (f : A → list B),
  (∀ x, Ordered S (f x)) →
  In b (unions (map f lst)) →
  ∃ (a : A), In a lst ∧ In b (f a).
Proof with eauto with datatypes.
  intros.
  induction lst...
  simpl in H0.
  inversion H0.
  simpl in H0.
  apply In_union in H0...
  destruct H0...
  apply IHlst in H0; clear IHlst.
  destruct H0 as [a0 [HIna HInb]]...
  apply unions_order_pres.
  intros.
  rewrite → in_map_iff in H1.
  destruct H1 as [a0 [HEq HIn]].
  remember (H a0); clear Heqo.
  rewrite → HEq in o...
Qed.

Section AllDiff.

  Require Import Common.AllDiff.
```

```
Lemma AllDiff_insert : ∀ (f : A → B) x lst,
  Ordered R lst →
  AllDiff f (insert x lst) →
  AllDiff f (x :: lst).
Proof with auto.
  intros.
  induction lst...
  simpl in H0.
  destruct (compare x a)...
  simpl in H0.
  destruct H0 as [H0 H1].
  apply IHlst in H1.
  2: solve [inversion H;trivial].
  simpl in H1.
  destruct H1 as [H1 H2].
  simpl.
  repeat split; intros...
  + destruct H3...
    subst.
    assert (f y ≠ f x).
    { apply H0. apply in_inserted. }
    unfold not; intros.
    unfold not in H3.
    apply H3...
  + apply H0.
    apply in_inserted_tail...
```

```
Qed.

Lemma AllDiff_insert_2 : ∀ (f : A → B) x lst,
  AllDiff f (x :: lst) →
  AllDiff f (insert x lst).
Proof with auto.
  intros.
  induction lst...
  simpl...
  destruct (compare x a)...
  simpl.
  split.
  + intros.
    simpl in H.
    destruct H as [J [J0 J1]]...
    apply In_insert in H0.
    destruct H0; subst.
    - assert (f x ≠ f a).
      apply J...
      unfold not. unfold not in H. intros. symmetry in H0.
      apply H in H0...
    - apply J0...
  + apply IHlst...
    simpl in H.
    destruct H as [J [J0 J1]]...
    simpl...
```

```
Qed.

Lemma AllDiff_preservation : ∀ (f : A → B) x y lst,
   Ordered R lst →
   Ordered R lst →
   AllDiff f (union [x] lst) →
   f x = f y →
   AllDiff f (union [y] lst).
Proof with auto with datatypes.
   intros.
   induction lst...
   + simpl...
   + simpl in H1.
     simpl.
     inversion H; subst.
     remember H1 as X eqn:Y; clear Y.
     apply AllDiff_insert in H1.
     inversion H1; subst...
     apply IHlst in H4...
     clear IHlst.
     rewrite ← union_singleton_l in *...
     apply AllDiff_insert_2.
     simpl. split...
     intros.
     apply In_insert in H7.
     { destruct H7.
```

```
          - subst...
            rewrite ← H2.
            apply H3.
            apply in_inserted.
          - apply H3...
            apply in_inserted_tail... }
        apply union_order_pres...
        apply Ordered_cons... intros. simpl in H3. inversion H3.
        apply Ordered_nil.
    Qed.
  End AllDiff.
End BinaryLemmas.
```

### A.2.7 TotalOrder Library

```
Set Implicit Arguments.
Require Import Coq.Relations.Relations.
Class TotalOrder {A : Type} (R : relation A) := {
  reflexivity : ∀ (x : A), R x x;
  antisymmetry : ∀ (x y : A), R x y → R y x → x = y;
  transitivity : ∀ (x y z : A), R x y → R y z → R x z;
  compare : ∀ (x y : A), { R x y } + { R y x };
  eqdec : ∀ (x y : A), { x = y } + { x ≠ y }
}.
```

```
Require Import Coq.Arith.Le.

Require Import Coq.Arith.Compare_dec.

Require Import Coq.Arith.Peano_dec.

Instance TotalOrder_nat : @TotalOrder nat le.

Proof with auto with arith.

  split...

  intros. eauto with arith.

  intros.

  assert ({ x ≤ y } + {x ≥ y}) as H.

  { apply le_ge_dec. }

  destruct H...

Qed.

Inductive PairOrdering

  {A B : Type}

  (AR : relation A) (BR : relation B) : (A × B) → (A × B) → Prop :=

| MkPairOrdering1 : ∀ a b1 b2,

  BR b1 b2 →

  PairOrdering AR BR (a,b1) (a,b2)

| MkPairOrdering2 : ∀ a1 a2 b1 b2,

  AR a1 a2 →

  a1 ≠ a2 →

  PairOrdering AR BR (a1,b1) (a2,b2).

Inductive SumOrdering {A B : Type}

  (AR : relation A) (BR : relation B) : relation (A + B) :=

| MkSumOrdering1 : ∀ x y,
  SumOrdering AR BR (inl x) (inr y)
| MkSumOrdering2 : ∀ x y,
  AR x y →
  SumOrdering AR BR (inl x) (inl y)
| MkSumOrdering3 : ∀ x y,
  BR x y →
  SumOrdering AR BR (inr x) (inr y).

Section SumOrdering.

  Hint Constructors SumOrdering.

  Variable A B : Type.
  Variable AR : relation A.
  Variable BR : relation B.
  Variable OrdA : TotalOrder AR.
  Variable OrdB : TotalOrder BR.

  Lemma SumOrdering_reflexivity : ∀ x, SumOrdering AR BR x x.
  Proof with eauto.
    destruct x...
    apply MkSumOrdering2...
    apply reflexivity.
    apply MkSumOrdering3...
    apply reflexivity.
  Qed.

  Lemma SumOrdering_antisymmetry : ∀ x y,
    SumOrdering AR BR x y →
    SumOrdering AR BR y x →
    x = y.
  Proof with eauto.
    intros.
    inversion H; subst.
    - inversion H0; subst.
    - inversion H0; subst.
      f_equal.
      apply antisymmetry...
    - inversion H0; subst.
      f_equal.
      apply antisymmetry...
  Qed.
  Lemma SumOrdering_transitivity : ∀ x y z,
    SumOrdering AR BR x y →
    SumOrdering AR BR y z →
    SumOrdering AR BR x z.
  Proof with eauto.
    intros.
    destruct x.
    destruct y.
    destruct z.
    inversion H; inversion H0; subst.
    - apply MkSumOrdering2...
      eapply transitivity...
    - apply MkSumOrdering1...
    - destruct z...
      inversion H0.
    - destruct y.
      inversion H.
      destruct z...
      inversion H0.
      eapply MkSumOrdering3...
      inversion H; subst.
      inversion H0; subst.
      eapply transitivity...
  Qed.

  Lemma SumOrdering_compare : ∀ x y,
    { SumOrdering AR BR x y } + { SumOrdering AR BR y x }.
  Proof with eauto.
    intros.
    destruct x.
    destruct y.
    + assert ({AR a a0} + {AR a0 a})...
      apply compare...
      destruct H...
    + left...
    + destruct y.
      - right...
      - assert ({BR b b0} + {BR b0 b})...
        apply compare...
        destruct H...
    Qed.

    Lemma SumOrdering_eqdec : ∀ (x y : A + B),
      { x = y } + { x ≠ y }.
    Proof with eauto.
      destruct x.
      destruct y.
      decide equality.
      + apply eqdec.
      + apply eqdec.
      + right. unfold not. intros. inversion H.
      + destruct y.
        - right. unfold not. intros. inversion H.
        - destruct (eqdec b b0); subst...
          right. unfold not. intros. inversion H. subst. contradiction n...
    Qed.

    Instance TotalOrder_sum : TotalOrder (SumOrdering AR BR) := {
      reflexivity := SumOrdering_reflexivity;
      antisymmetry := SumOrdering_antisymmetry;
      transitivity := SumOrdering_transitivity;
      compare := SumOrdering_compare;
      eqdec := SumOrdering_eqdec
    }.

  End SumOrdering.

  Section PairOrdering.
```

```
Hint Constructors PairOrdering.

Variable A B : Type.

Variable AR : relation A.

Variable BR : relation B.

Variable OrdA : TotalOrder AR.

Variable OrdB : TotalOrder BR.

Lemma PairOrdering_reflexivity : ∀ x, PairOrdering AR BR x x.

Proof with eauto.

  destruct x...

  apply MkPairOrdering1...

  apply reflexivity.

Qed.

Lemma PairOrdering_antisymmetry : ∀ x y,

  PairOrdering AR BR x y →

  PairOrdering AR BR y x →

  x = y.

Proof with eauto.

  intros.

  inversion H; subst.

  - inversion H0; subst.

    assert (b1 = b2). apply antisymmetry...

    subst...

    contradiction H7...

  - inversion H0; subst.

    contradiction H2...
    assert (a1 = a2). apply antisymmetry...
    subst.
    contradiction H2...
Qed.

Lemma PairOrdering_transitivity : ∀ x y z,
  PairOrdering AR BR x y →
  PairOrdering AR BR y z →
  PairOrdering AR BR x z.
Proof with eauto.
  intros.
  destruct x.
  destruct y.
  destruct z.
  inversion H; inversion H0; subst.
  - apply MkPairOrdering1...
    eapply transitivity...
  - apply MkPairOrdering2...
  - apply MkPairOrdering2...
  - apply MkPairOrdering2...
    eapply transitivity...
    assert ({a = a1 } + { a ≠ a1 }). apply eqdec.
    destruct H1...
    subst.
    assert (a0 = a1). apply antisymmetry...
    subst...
```

```
Qed.

Lemma PairOrdering_compare : ∀ x y,
  { PairOrdering AR BR x y } + { PairOrdering AR BR y x }.
Proof with eauto.
  intros.
  destruct x.
  destruct y.
  assert ({AR a a0} + {AR a0 a})...
  apply compare...
  assert ({BR b b0} + {BR b0 b})...
  apply compare...
  assert ({a = a0 } + { a ≠ a0 }). apply eqdec.
  destruct H1; subst...
  - destruct H0...
  - destruct H...
Qed.

Lemma PairOrdering_eqdec : ∀ (x y : A × B),
  { x = y } + { x ≠ y }.
Proof with eauto.
  destruct x.
  destruct y.
  decide equality.
  apply eqdec.
  apply eqdec.
Qed.
```

```
  Instance TotalOrder_pair : TotalOrder (PairOrdering AR BR) := {
    reflexivity := PairOrdering_reflexivity;
    antisymmetry := PairOrdering_antisymmetry;
    transitivity := PairOrdering_transitivity;
    compare := PairOrdering_compare;
    eqdec := PairOrdering_eqdec
  }.
End PairOrdering.
Existing Instances TotalOrder_pair TotalOrder_sum.
Definition inverse (A B : Type) (f : A → B) (g : B → A) : Prop :=
  ∀ x, g (f x) = x.
Inductive ProjectOrdering (A B : Type) (f : A → B) (BR : relation B) : relation A :=
  | MkProjOrdering : ∀ x y,
    BR (f x) (f y) →
    ProjectOrdering f BR x y.
Hint Constructors ProjectOrdering.
Lemma TotalOrder_Project : ∀ (A B : Type) (f : A → B)
   (g : B → A)
   (RB : relation B)
   (OrdB : TotalOrder RB),
   inverse f g →
   TotalOrder (ProjectOrdering f RB).
Proof with eauto.
  intros.
  destruct OrdB.
  split; intros.
  + eapply MkProjOrdering...
  + unfold inverse in H.
    inversion H0; inversion H1; subst...
    assert (f x = f y) as X.
    apply antisymmetry0...
    rewrite ← H.
    rewrite ← (H x).
    rewrite → X...
  + inversion H0; inversion H1; subst...
  + destruct (compare0 (f x) (f y))...
  + destruct (eqdec0 (f x) (f y))...
    - left.
      rewrite ← H.
      rewrite ← (H x).
      rewrite → e...
    - right.
      unfold not in *.
      intros.
      rewrite → H0 in n.
      apply n.
      reflexivity.
Qed.
```

### A.2.8 Classifier Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Coq.Lists.List.

Require Import Network.NetworkPacket.

Require Import Word.WordInterface.

Require Import Pattern.Pattern.

Local Open Scope list_scope.

Definition Classifier (A : Type) := list (pattern × A) %type.

Fixpoint scan {A : Type} (default : A) (classifier : Classifier A) (pt : portId)
  (pk : packet) :=
  match classifier with
    | nil ⇒ default
    | (pat,a) :: rest ⇒
      match Pattern.match_packet pt pk pat with
        | true ⇒ a
        | false ⇒ scan default rest pt pk
      end
  end.

Definition inter_entry {A : Type} {B : Type} (f : A → A → B)
  (cl : Classifier A) (v : pattern × A) :=
  let (pat, act) := v in
    fold_right (fun (v' : pattern × A) acc ⇒
      let (pat', act') := v' in
```

```
        (Pattern.inter pat pat', f act act') :: acc)
    nil cl.

Definition inter {A : Type} {B : Type} (f : A → A → B) (cl1 cl2 : Classifier A) :=
  fold_right (fun v acc ⇒ inter_entry f cl2 v ++ acc)
  nil cl1.

Definition union {A : Type} (f : A → A → A) (cl1 cl2 : Classifier A) :=
  inter f cl1 cl2 ++ cl1 ++ cl2.
```

Why so baroque? Filtering the tail of the list is not structurally recursive. `Fixpoint`

```
elim_shadowed_helper {A : Type} (prefix : Classifier A)
  (cf : Classifier A) :=
  match cf with
    | nil ⇒ prefix
    | (pat,act) :: cf' ⇒
      match existsb
        (fun (entry : pattern × A) ⇒
          let (pat', act) := entry in
            if Pattern.beq pat pat' then true else false)
        prefix with
        | true ⇒ elim_shadowed_helper prefix cf'
        | false ⇒ elim_shadowed_helper (prefix ++ [(pat,act)]) cf'
      end
  end.

Definition elim_shadowed {A : Type} (cf : Classifier A) :=
  elim_shadowed_helper nil cf.
```

```
Fixpoint prioritize
  {A : Type}
  (prio : nat)
  (lst : Classifier A) : list (nat × pattern ×A) :=
  match lst with
    | nil ⇒ nil
    | (pat, act) :: lst' ⇒ (prio, pat, act) :: (prioritize (pred prio) lst')
  end.
```

### A.2.9 Theory Library

```
Set Implicit Arguments.
Require Import Coq.Classes.Equivalence.
Require Import Coq.Lists.List.
Require Import Coq.Bool.Bool.
Require Import Common.CpdtTactics.
Require Import Common.Types.
Require Import Network.NetworkPacket.
Require Import Pattern.Pattern.
Require Import Classifier.Classifier.
Local Open Scope list_scope.
Local Open Scope equiv_scope.
Section Equivalence.
  Definition Classifier_equiv {A : Type} (cf1 cf2 : Classifier A) :=
    ∀ pt pk def, scan def cf1 pt pk = scan def cf2 pt pk.
  Lemma Classifier_equiv_is_Equivalence : ∀ {A : Type},
    Equivalence (@Classifier_equiv A).
  Proof with auto.
    intros.
    split.
    unfold Reflexive.
    unfold Classifier_equiv...
    unfold Symmetric.
    unfold Classifier_equiv...
    unfold Transitive.
    unfold Classifier_equiv.
    intros.
    rewrite → H...
  Qed.
End Equivalence.
Instance Classifier_Equivalance `(A : Type) :
  Equivalence (@Classifier_equiv A).
Proof.
  apply Classifier_equiv_is_Equivalence.
Qed.
Class ClassifierAction `(A : Type) := {
  action_eqdec : ∀ (x y : A), { x = y } + { x ≠ y };
  zero : A
}.
Definition has_unit {A : Type} {Act : ClassifierAction A}
```

```
  (f : A → A → A) : Prop :=
  (∀ a, f a zero = a) ∧ ∀ a, f zero a = a.

Inductive total (A : Type) : Classifier A → Prop :=
| total_tail : ∀ a cf,
  total (cf ++ [(Pattern.all, a)]).

Section Lemmas.

  Hint Constructors total.

  Variable A B : Type.

  Lemma scan_map_comm : ∀ (f : A → B) (defA : A) (defB : B) cf pt pk,
    total cf →
    scan defB (map (second f) cf) pt pk = f (scan defA cf pt pk).
  Proof with auto.
    intros f defA defB cf pt pk H.
    inversion H.
    generalize dependent cf0.
    induction cf; intros.
    + simpl.
      destruct cf0. simpl in H0. inversion H0.
      rewrite ← app_comm_cons in H0. inversion H0.
    + intros.
      destruct cf0.
      - simpl...
        rewrite → Pattern.all_spec...
      - simpl.
        destruct p.
        simpl.
        destruct (Pattern.match_packet pt pk t)...
        rewrite ← app_comm_cons in H0.
        inversion H0.
        apply IHcf...
        destruct cf0...
        simpl in H3.
        subst.
        rewrite ← app_nil_l...
        subst...
  Qed.

  Lemma scan_elim_unit_tail : ∀ (def : A) pk pt cf pat,
    scan def (cf ++ [(pat, def)]) pt pk = scan def cf pt pk.
  Proof with auto.
    intros.
    induction cf.
    simpl.
    destruct (Pattern.match_packet pt pk pat)...
    destruct a as [pat' a].
    simpl.
    destruct (Pattern.match_packet pt pk pat')...
  Qed.

  Lemma scan_inv : ∀ (def : A) pkt port
    (N1 : Classifier A),
    ((∀ m a, In (m,a) N1 → Pattern.match_packet port pkt m = false) ∧
          scan def N1 port pkt = def) ∨
        (∃ N2 N3, ∃ m : pattern, ∃ a : A,
          N1 = N2 ++ (m,a)::N3 ∧
          Pattern.match_packet port pkt m = true ∧
          scan def N1 port pkt = a ∧
          (∀ (m' : pattern) (a' : A), In (m',a') N2 →
            Pattern.match_packet port pkt m' = false)).
  Proof with intros; simpl; auto with datatypes.
    intros.
    induction N1.
    intros.
    left...
    split...
    contradiction.
    destruct a.
    destruct IHN1.
    remember (Pattern.match_packet port pkt p) as b. destruct b.
    right. ∃ nil. ∃ N1. ∃ p. ∃ a.
    crush. rewrite ← Heqb...
    left. crush. apply H0 in H2. crush. rewrite ← Heqb...
    destruct H as [N2 [N3 [m [a' [Neq [Hov [Ha'eq H]]]]]]].
    remember (Pattern.match_packet port pkt p) as b. destruct b.
    right. ∃ nil. ∃ N1. ∃ p. ∃ a. crush. rewrite ← Heqb...
    right. ∃ ((p,a) :: N2). ∃ N3. ∃ m. ∃ a'.
    crush. rewrite ← Heqb... apply H in H1. crush.
  Qed.
```

```
Hint Unfold union inter inter_entry.

Variable f : A → A → A.

Variable def : A.

Variable cf : Classifier A.

Lemma inter_nil_l : inter f nil cf = nil.

Proof. intros. induction cf; crush. Qed.

Lemma inter_nil_r : inter f cf nil = nil.

Proof. intros. induction cf; crush. Qed.

Hint Resolve inter_nil_l inter_nil_r.

Lemma elim_scan_head : ∀ cf1 cf2 pkt pt,
  (∀ m a, In (m,a) cf1 → Pattern.match_packet pt pkt m = false) →
  scan def (cf1 ++ cf2) pt pkt = scan def cf2 pt pkt.
Proof with simpl; auto with datatypes.
  intros.
  induction cf1...
  destruct a as [m a].
  assert (∀ m a', In (m,a') cf1 → Pattern.match_packet pt pkt m = false).
    intros. apply H with (a0 := a')...
  apply IHcf1 in H0.
  assert (Pattern.match_packet pt pkt m = false).
    assert (In (m,a) ((m,a)::cf1))...
    apply H in H1...
  rewrite → H1...
Qed.

Hint Resolve elim_scan_head.
```

```
Lemma elim_scan_middle : ∀ cf1 cf2 cf3 pkt pt,
  (∀ m (a : A), In (m,a) cf2 → Pattern.match_packet pt pkt m = false) →
  scan def (cf1 ++ cf2 ++ cf3) pt pkt = scan def (cf1 ++ cf3) pt pkt.
Proof.
  intros.
  generalize dependent cf2.
  induction cf1; crush.
Qed.

Hint Resolve elim_scan_middle.

Lemma elim_scan_tail : ∀ cf1 cf2 cf3 pat a pt pk,
  Pattern.match_packet pt pk pat = true →
  scan def (cf1 ++ (pat, a) :: cf2 ++ cf3) pt pk =
  scan def (cf1 ++ (pat, a) :: cf2) pt pk.
Proof with auto.
  intros.
  induction cf1.
  simpl.
  rewrite → H...
  destruct a0 as [pat0 a0].
  simpl.
  remember (Pattern.match_packet pt pk pat0) as b.
  destruct b...
Qed.

Lemma elim_inter_head : ∀ cf1 cf2 pt pkt m a,
  Pattern.match_packet pt pkt m = false →
  scan def
  (fold_right
    (fun (v' : pattern × A) (acc : list (pattern × A)) ⇒
      let (pat', act') := v' in (Pattern.inter m pat', f a act') :: acc)
    nil cf1 ++ cf2) pt pkt = scan def cf2 pt pkt.
Proof with auto.
  intros.
  induction cf1...
  destruct a0 as [p0 a0].
  simpl.
  rewrite → Pattern.is_match_false_inter_l...
Qed.

Hint Resolve elim_inter_head.

Lemma elim_inter_head_aux : ∀ cf1 cf2 pkt pt m (a : A),
  Pattern.match_packet pt pkt m = false →
  scan def (inter_entry f cf1 (m, a) ++ cf2) pt pkt = scan def cf2 pt pkt.
Proof with auto.
  intros.
  induction cf1.
  crush.
  destruct a0 as [p0 a0].
  simpl.
  rewrite → Pattern.is_match_false_inter_l...
Qed.

Lemma inter_empty_aux : ∀ N1 m m0 pkt pt (a a0 : A),
  (∀ m (a : A), In (m,a) N1 → Pattern.match_packet pt pkt m = false) →
  In (m,a) (inter_entry f N1 (m0,a0)) →
  Pattern.match_packet pt pkt m = false.
Proof with auto with datatypes.
  intros.
  induction N1.
  + crush.
  + destruct a1.
    simpl in H0.
    destruct H0.
    - inversion H0; subst; clear H0.
      apply Pattern.no_match_subset_r...
      eapply H...
    - apply IHN1...
      intros. eapply H... simpl. right. exact H1.
Qed.

Lemma inter_empty : ∀ N2 pkt pt,
  (∀ m (a : A), In (m,a) N2 → Pattern.match_packet pt pkt m = false) →
  (∀ N1 m (a : A), In (m,a) (inter f N1 N2) →
    Pattern.match_packet pt pkt m = false).
Proof with auto with datatypes.
  intros N2 pkt pt.
  intros Hlap.
  intros.
  generalize dependent N2.
```

```
    induction N1.
    crush.
    destruct a0.
    intros.
    simpl in H.
    rewrite → in_app_iff in H.
    destruct H.
    apply inter_empty_aux with (N1 := N2) (m0 := p) (a := a) (a0 := a0)...
    apply IHN1 in Hlap...
  Qed.

  Hint Resolve inter_empty scan_inv.
  Hint Rewrite in_app_iff.
End Lemmas.

Section Optimizer.

  Lemma elim_shadowed_equiv : ∀ {A : Type}
    pat1 pat2 act1 act2 (cf1 cf2 cf3 : Classifier A),
    Pattern.equiv pat1 pat2 →
    Classifier_equiv
      (cf1 ++ (pat1,act1) :: cf2 ++ (pat2,act2) :: cf3)
      (cf1 ++ (pat1,act1) :: cf2 ++ cf3).
  Proof with auto.
    intros.
    unfold Classifier_equiv.
    intros.
    remember (Pattern.match_packet pt pk pat1) as Hmatched.
```

```
destruct Hmatched.
assert (scan def (cf1 ++ (pat1,act1) :: cf2 ++ (pat2,act2) :: cf3) pt pk =
  scan def (cf1 ++ (pat1,act1) :: cf2) pt pk).
apply elim_scan_tail...
rewrite → H0.
assert (scan def (cf1 ++ (pat1,act1) :: cf2 ++ cf3) pt pk =
  scan def (cf1 ++ (pat1,act1) :: cf2) pt pk).
apply elim_scan_tail...
rewrite → H1...
assert (false = Pattern.match_packet pt pk pat2) as Hpat2Unmatched.
  rewrite → HeqHmatched.
  unfold equiv in H...
assert ((pat2,act2) :: cf3 = [(pat2,act2)] ++ cf3) as J0 by auto.
rewrite → J0.
assert (cf1 ++ (pat1,act1) :: cf2 ++ [(pat2,act2)] ++ cf3 =
  (cf1 ++ (pat1,act1) :: cf2) ++ [(pat2,act2)] ++ cf3) as J1.
rewrite ← app_assoc...
rewrite → J1.
assert (cf1 ++ (pat1,act1) :: cf2 ++ cf3 =
  (cf1 ++ (pat1,act1) :: cf2) ++ cf3) as J2.
rewrite ← app_assoc...
rewrite → J2.
apply elim_scan_middle.
exact (fun x y ⇒ x).        intros.
inversion H0.
inversion H1.
```

```
  subst...
  inversion H1.
Qed.
Lemma elim_shadowed_helper_ok : ∀ {A : Type}
  (prefix postfix : Classifier A),
  Classifier_equiv
    (prefix ++ postfix) (elim_shadowed_helper prefix postfix).
Proof with auto.
  intros.
  unfold Classifier_equiv.
  generalize dependent prefix.
  induction postfix; intros.
  simpl.
  rewrite → app_nil_r...
  destruct a as [pat act].
  simpl.
  match goal with
    | [ ⊢ context[if ?b then _ else _] ] ⇒ remember b
  end.
  destruct b.
  Focus 2.
  assert ((pat,act) :: postfix = [(pat,act)] ++ postfix) as Hfoo by auto.
  rewrite → Hfoo.
  rewrite → app_assoc.
  apply IHpostfix.
  symmetry in Heqb.
  rewrite → existsb_exists in Heqb.
  destruct Heqb as [[pat' act'] [HIn Heq]].
  assert (scan def (prefix ++ (pat,act) :: postfix) pt pk =
    scan def (prefix ++ postfix) pt pk) as Hit.
  apply In_split in HIn.
  destruct HIn as [l1 [l2 HIn]].
  rewrite → HIn.
  rewrite ← app_assoc.
  rewrite ← app_assoc.
  simpl.
  apply elim_shadowed_equiv.
  remember (Pattern.beq pat pat') as b.
  destruct b.
  symmetry in Heqb.
  apply Pattern.beq_true_spec in Heqb.
  unfold Coq.Classes.Equivalence.equiv in Heqb.
  apply symmetry...
  inversion Heq.
  rewrite → Hit.
  apply IHpostfix.
Qed.
Theorem elim_shadowed_ok : ∀ {A : Type} (cf : Classifier A),
  cf === elim_shadowed cf.
Proof with auto.
```

```
    intros.
    unfold elim_shadowed.
    assert (nil ++ cf = cf) as J0...
    rewrite ← J0.
    apply elim_shadowed_helper_ok.
  Qed.
End Optimizer.
Section Action.
  Variable A : Type.
  Variable A_as_Action : ClassifierAction A.
  Implicit Arguments A.
  Implicit Arguments A_as_Action.
  Definition left_biased (a b : A) :=
    match action_eqdec a zero with
      | left _ ⇒ b
      | right _ ⇒ a
    end.
  Lemma left_biased_has_unit : has_unit left_biased.
  Proof with auto.
    unfold left_biased.
    split; intros.
    remember (action_eqdec a zero) as b.
    destruct b...
    remember (action_eqdec zero zero) as b.
    destruct b...
  contradiction n...
Qed.

Hint Resolve left_biased_has_unit.

Hint Constructors total.

Lemma inter_entry_app : ∀ cf1 cf2 m (a : A) (f : A → A → A),
  inter_entry f (cf1 ++ cf2) (m,a) =
  inter_entry f cf1 (m,a) ++ inter_entry f cf2 (m,a).
Proof with auto.
  intros.
  induction cf1...
  destruct a0.
  simpl. f_equal...
Qed.

Lemma inter_entry_andb_true : ∀ cf pat pt pk b,
  true = Pattern.match_packet pt pk pat →
  scan b (inter_entry andb cf (pat, true)) pt pk = scan b cf pt pk.
Proof with auto with datatypes.
  intros.
  induction cf...
  destruct a.
  simpl...
  remember (Pattern.match_packet pt pk p) as b1.
  destruct b1...
  + rewrite → Pattern.is_match_true_inter...
  + rewrite → Pattern.no_match_subset_r...
```

```
Qed.

Lemma scan_app_compose : ∀ pt pk lst1 lst2,
  scan false (lst1 ++ lst2) pt pk = scan (scan false lst2 pt pk) lst1 pt pk.
Proof with auto with datatypes.
  intros.
  induction lst1...
  destruct a.
  simpl.
  rewrite → IHlst1...
Qed.

Lemma scan_full_false : ∀ lst pat pt pk,
  scan false (inter_entry andb lst (pat, false)) pt pk = false.
Proof with auto with datatypes.
  intros.
  induction lst...
  destruct a.
  simpl.
  clear b.
  remember (Pattern.match_packet pt pk (Pattern.inter pat p)) as b.
  destruct b...
Qed.

 Lemma scan_bool_flatten : ∀ b cf2 pt pk,
  scan (b && scan false cf2 pt pk) cf2 pt pk = scan false cf2 pt pk.
 Proof with auto with datatypes.
   intros.
  destruct b...
  simpl.
  rewrite ← scan_app_compose.
  induction cf2...
  destruct a.
  simpl.
  remember (Pattern.match_packet pt pk p) as b0.
  destruct b0...
  assert (cf2 ++ (p,b) :: cf2 = cf2 ++ [(p,b)] ++ cf2) as X...
  rewrite → X.
  rewrite → elim_scan_middle...
  intros.
  simpl in H.
  destruct H.
  + inversion H; subst; clear H...
  + inversion H.
Qed.

Lemma inter_entry_andb_false : ∀ lst pat b pt pk,
  scan false lst pt pk = true →
  Pattern.match_packet pt pk pat = true →
  scan b (inter_entry andb lst (pat, false)) pt pk = false.
Proof with auto with datatypes.
  intros.
  induction lst...
  simpl in H.
```

```
    inversion H.
    destruct a.
    simpl.
    remember (Pattern.match_packet pt pk p) as b1.
    destruct b1.
    rewrite → Pattern.is_match_true_inter...
    rewrite → Pattern.no_match_subset_r...
    assert (fold_right
              (fun (v' : pattern × bool) (acc : list (Pattern.t × bool)) ⇒
                 let (pat', _) := v' in (Pattern.inter pat pat', false) :: acc) nil
              lst = inter_entry andb lst (pat, false)) as X...
    rewrite → X; clear X.
    apply IHlst.
    simpl in H.
    rewrite ← Heqb1 in H...
  Qed.
Lemma inter_comm_bool_range : ∀ (cf1 cf2 : Classifier bool)
  (pt : portId) (pk : packet),
  @scan bool false (inter andb cf1 cf2) pt pk = andb (scan false cf1 pt pk) (scan false
cf2 pt pk).
Proof with auto with datatypes.
  intros.
  induction cf1.
  + simpl...
  + destruct a.
```

```
    simpl.
    remember (Pattern.match_packet pt pk p) as matched.
    destruct matched.
    - { assert (fold_right
        (fun (v' : pattern × bool) (acc : list (Pattern.t × bool)) ⇒
          let (pat', act') := v' in (Pattern.inter p pat', b && act') :: acc)
        nil cf2 = inter_entry andb cf2 (p,b)) as X.
        { simpl. reflexivity. }
        rewrite → X. clear X.
        rewrite → scan_app_compose.
        rewrite → IHcf1.
        destruct b.
        + rewrite → inter_entry_andb_true.
          simpl.
          apply scan_bool_flatten.
          symmetry...
        + rewrite → andb_false_l.
          remember (scan false cf1 pt pk && scan false cf2 pt pk) as b.
          destruct b.
          - symmetry in Heqb.
            rewrite → andb_true_iff in Heqb.
            destruct Heqb.
            rewrite → inter_entry_andb_false...
          - apply scan_full_false. }
    - rewrite → elim_inter_head...
Qed.
```

```
  Lemma union_scan_comm : ∀ (f : A → A → A) pt pk cf1 cf2,
    has_unit f →
    scan zero (union f cf1 cf2) pt pk =
    f (scan zero cf1 pt pk) (scan zero cf2 pt pk).
  Proof with simpl; eauto with datatypes.
    intros f pt pk cf1 cf2 H.
    remember H as Hwb.
    destruct H as [H H0].
    induction cf1.
    rewrite → H0...
    unfold union.
    destruct a as [m a].
    remember (Pattern.match_packet pt pk m).
    remember (scan_inv zero pk pt (inter f ((m, a) :: cf1) cf2 ++ ((m, a) :: cf1) ++ cf2))
as H1. clear HeqH1.
    destruct H1.
    destruct H1 as [H1 H2].
    rewrite → H2.
    assert (Pattern.match_packet pt pk m = false) as HnotA.
    apply H1 with (a0 := a)...
    simpl in H2.
    rewrite ← app_assoc in H2.
    rewrite → elim_inter_head in H2...
    assert ((m,a) :: cf1 ++ cf2 = [(m,a)] ++ cf1 ++ cf2) as Hcf. auto.
    rewrite → Hcf in H2.
    rewrite → elim_scan_middle in H2.
rewrite → HnotA.
rewrite ← IHcf1.
unfold union...
exact f.
intros. inversion H3. inversion H4. subst... inversion H4.
destruct H1 as [cf3 [cf4 [m0 [a0 [H1 [H2 [H3 H4]]]]]]].
destruct b.
clear IHcf1.
rewrite ← app_comm_cons.
remember (scan_inv zero pk pt cf2) as Hinv. clear HeqHinv.
destruct Hinv as [[H5 H6]|Hinv].
rewrite → H6.
assert (∀ m' (a' : A), In (m',a') (inter f ((m, a) :: cf1) cf2) →
  Pattern.match_packet pt pk m' = false) as H7.
apply inter_empty; auto.
assert (scan zero (inter f ((m, a) :: cf1) cf2 ++
  (m, a) :: cf1 ++ cf2) pt pk =
scan zero ((m,a) :: cf1 ++ cf2) pt pk) as HelimHd.
apply elim_scan_head; auto.
rewrite → HelimHd.
assert ((m,a) :: cf1 ++ cf2 =
  nil ++ (m,a) :: cf1 ++ cf2) as HNilHd by auto.
rewrite → HNilHd.
rewrite → elim_scan_tail.
rewrite → app_nil_l.
rewrite → H.
```

```
reflexivity.
auto.
destruct Hinv as [N2' [N3' [m' [a' [Heq' [Hlap' [Hscan' Hlap2']]]]]]].
match goal with
  | [ ⊢ ?X = ?Y ] ⇒ remember Y as RHS end.
assert (RHS = f a a') as HRHS.
rewrite → HeqRHS.
simpl.
rewrite ← Heqb.
rewrite → Hscan'...
simpl.
match goal with
  | [ ⊢ context[fold_right ?f ?acc ?lst]] ⇒ remember (fold_right f acc lst) as F
end.
rewrite ← app_assoc.
remember (inter f cf1 cf2 ++ (m,a) :: cf1 ++ cf2) as Trash.
assert (∀ m5 (a5 : A),
  In (m5,a5) (fold_right
    (fun (v' : pattern × A) (acc : list (pattern × A)) ⇒
      let (pat', act') := v' in (Pattern.inter m pat', f a act') :: acc)
    nil N2') →
  Pattern.match_packet pt pk m5 = false) as HOMG.
match goal with
  | [ ⊢ context[fold_right ?f ?acc ?lst]] ⇒ remember (fold_right f acc lst) as F1
end.
assert (F1 = inter f [(m,a)] N2') as HF1.
```

```
simpl. rewrite → app_nil_r. rewrite → HeqF1...
rewrite → HF1.
apply inter_empty; auto.
rewrite → Heq' in HeqF.
assert (F = inter_entry f (N2' ++ (m',a') :: N3') (m,a)).
rewrite → HeqF. simpl. auto.
assert ( (fold_right
  (fun (v' : pattern × A) (acc : list (pattern × A)) ⇒
    let (pat', act') := v' in (Pattern.inter m pat', f a act') :: acc)
  nil N2') = inter_entry f N2' (m,a)). simpl. auto.
rewrite → H6 in HOMG.
rewrite → H5.
rewrite → inter_entry_app.
rewrite ← app_assoc.
rewrite → elim_scan_head.
simpl.
rewrite → Pattern.is_match_true_inter...
auto.
assert ((m,a) :: cf1 = [(m,a)] ++ cf1) as Hsimpl. auto.
rewrite → Hsimpl.
rewrite ← app_assoc.
rewrite → elim_scan_middle with (cf2 := [(m,a)]).
rewrite ← Hsimpl. clear Hsimpl.
simpl.
rewrite ← app_assoc.
```

```
    rewrite → elim_inter_head.
    rewrite ← Heqb.
    unfold union in IHcf1.
    trivial.
    auto.
    exact f.
    intros. inversion H5. inversion H6. subst... inversion H6.
  Qed.

  Lemma prefix_equivalence : ∀ cf1 cf2 pt pk,
    scan unit cf1 pt pk = scan unit (cf1 ++ cf2) pt pk ∨
    scan unit cf1 pt pk = unit.
  Proof with auto.
    intros cf1 cf2 pt pk.
    induction cf1.
    right...
    destruct a as [pat a].
    simpl.
    remember (Pattern.match_packet pt pk pat) as b.
    destruct b.
    left...
    exact IHcf1.
  Qed.
End Action.
```

### A.2.10 AllDiff Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Coq.Lists.List.

Local Open Scope list_scope.

Fixpoint AllDiff (A B : Type) (f : A → B) (lst : list A) : Prop :=
  match lst with
    | nil ⇒ True
    | x :: xs ⇒ (∀ (y : A), In y xs → f x ≠ f y) ∧ AllDiff f xs
  end.

Lemma AllDiff_uniq : ∀
  (A B : Type) (f : A → B) (lst : list A),
  AllDiff f lst →
  ∀ (x y : A),
    In x lst →
    In y lst →
    f x = f y →
    x = y.
Proof with auto with datatypes.
  intros.
  induction lst.
  inversion H0.
  simpl in H.
  destruct H as [HDiffHd HDiffTl].
  simpl in H0.
  simpl in H1.
  destruct H0 as [H0 | H0]; destruct H1 as [H1 | H1]; subst...
  apply HDiffHd in H1.
  contradiction.
  apply HDiffHd in H0.
  symmetry in H2.
  contradiction.
Qed.

Lemma map_eq_inj : ∀ (A B : Type) (f : A → B) (lst1 lst2 : list A) (x : A),
  map f lst1 = map f lst2 →
  In x lst1 →
  ∃ (y : A),
    In y lst2 ∧ f x = f y.
Proof with auto with datatypes.
  intros.
  generalize dependent lst1.
  induction lst2; intros.
  simpl in H. destruct lst1. inversion H0. simpl in H. inversion H.
  destruct lst1.
  simpl in H. inversion H0.
  simpl in H.
  inversion H.
  simpl in H0.
  destruct H0.
  subst. ∃ a...
  apply IHlst2 in H0...
  destruct H0 as [y0 [HIn2 HEq]].
  ∃ y0...
Qed.

Lemma AllDiff_preservation : ∀
  (A B : Type) (f : A → B) (lst1 lst2 : list A),
  AllDiff f lst1 →
  map f lst1 = map f lst2 →
  AllDiff f lst2.
Proof with auto with datatypes.
  intros.
  generalize dependent lst1.
  induction lst2; intros.
  simpl...
  destruct lst1.
  simpl in H0. inversion H0.
  simpl in H0.
  inversion H0.
  simpl in H.
  destruct H as [HDiffHd HDiffTl].
  simpl.
  split.
  intros.
  apply map_eq_inj with (f := f) (lst2 := lst1) in H...
  destruct H as [y0 [HIn2 HEqyy0]].
  apply HDiffHd in HIn2.
  rewrite → HEqyy0.
  rewrite ← H2.
  trivial.
  apply IHlst2 in H3...
Qed.
```

### A.2.11 Bisimulation Library

```
Set Implicit Arguments.
Require Import Common.Types.
Require Import Coq.Lists.List.
Local Open Scope list_scope.
Definition relation (A B : Type) := A → B → Prop.
Definition inverse_relation (A B : Type) (R : relation A B) : relation B A :=
  fun (b : B) (a : A) ⇒ R a b.
Definition step (A Ob : Type) := A → option Ob → A → Prop.
Inductive multistep (A Ob : Type) (step : step A Ob)
  : A → list Ob → A → Prop :=
  | multistep_nil : ∀ a, multistep step a nil a
  | multistep_tau : ∀ a a0 a1 obs,
     step a None a0 →
     multistep step a0 obs a1 →
     multistep step a obs a1
  | multistep_obs : ∀ a a0 a1 ob obs,
     step a (Some ob) a0 →
     multistep step a0 obs a1 →
     multistep step a (ob :: obs) a1.

Hint Constructors multistep.

Lemma multistep_app : ∀ (A Ob : Type)
  (step : step A Ob)
  (s1 s2 s3 : A)
  (obs1 obs2 obs3: list Ob),
  multistep step s1 obs1 s2 →
  multistep step s2 obs2 s3 →
  obs3 = obs1 ++ obs2 →
  multistep step s1 obs3 s3.
Proof with auto.
  intros.
  generalize dependent obs3.
  induction H; intros...
  simpl in H1. subst...
  apply IHmultistep with (obs3 := obs3) in H0.
  apply multistep_tau with (a0 := a0)...
  trivial.
  simpl in H2.
  rewrite → H2.
  apply multistep_obs with (a0 := a0)...
Qed.
```

```
Definition list_of_option {A : Type} (opt : option A) : list A :=
  match opt with
    | Some a ⇒ [a]
    | None ⇒ nil
  end.

Definition weak_simulation
  (S T Ob : Type)
  (step_S : step S Ob)
  (step_T : step T Ob)
  (R : relation S T) :=
  ∀ (s : S) (t : T),
    R s t →
    ∀ (s' : S) (obs : option Ob),
      step_S s obs s' →
      ∃ (t' : T),
        R s' t' ∧
        multistep step_T t (list_of_option obs) t'.

Definition weak_bisimulation
  (S T A : Type)
  (step_S : step S A)
  (step_T : step T A)
  (R : relation S T) :=
  weak_simulation step_S step_T R ∧
  weak_simulation step_T step_S (inverse_relation R).
```

### A.2.12 Monad Library

```
Set Implicit Arguments.

Reserved Notation "x <- M ; K" (at level 60, right associativity).

Module Type MONAD.

  Parameter m : Type → Type.

  Parameter bind : ∀ {A B : Type}, m A → (A → m B) → m B.

  Parameter ret : ∀ {A : Type}, A → m A.

End MONAD.
```

### A.2.13 Types Library

```
Set Implicit Arguments.

Require Import Arith.Peano_dec.

Require Import Coq.Lists.List.

Open Local Scope list_scope.

Notation "[ a ; .. ; b ]" := (a :: .. (b :: nil) ..) : list_scope.

Definition Eqdec (A : Type) :=
  ∀ (x y : A), { x = y } + { x ≠ y }.

Definition second {A B C: Type} (f : B → C) (pair : (A × B)) :=
  match pair with
    | (a,b) ⇒ (a, (f b))
  end.

Class Eq (a : Type) := {
  eqdec : ∀ (x y : a), { x = y } + { ¬ x = y }
}.
Definition beqdec { A : Type } { E : Eq A } (x y : A) := match eqdec x y with
  | left _ ⇒ true
  | right _ ⇒ false
end.
Instance Eq_nat : Eq nat := {
  eqdec := eq_nat_dec
}.
Lemma option_eq : ∀ { a : Type } { E : Eq a } (x y : option a), { x = y } + { ¬ x = y }.
Proof.
  decide equality. apply eqdec.
Defined.
Instance Eq_option `(a : Type, E : Eq a) : Eq (option a) := {
  eqdec := option_eq
}.
Lemma pair_eq : ∀ { A B : Type } { EqA : Eq A } { EqB : Eq B }
                      (x y : A × B),
                      { x = y } + { ¬ x = y }.
Proof.
  decide equality. apply eqdec. apply eqdec.
Defined.
Instance Eq_pair `(A : Type, B : Type, EA : Eq A, EB : Eq B) : Eq (A × B) := {
  eqdec := pair_eq
}.
```

```
Lemma list_eq : ∀ (A : Type) (E : Eq A) (lst1 lst2 : list A),
  { lst1 = lst2 } + { lst1 ≠ lst2 }.
Proof with auto.
  intros.
  decide equality.
  apply eqdec.
Qed.
Extract Constant list_eq ⇒ "(fun _x y -> x = y)".
Instance Eq_list `(A : Type, E : Eq A) : Eq (list A) := {
  eqdec := list_eq E
}.
Reserved Notation "x == y" (at level 70, no associativity).
Notation "x == y" := (beqdec x y) : of_scope.
Section List.
Definition concat_map {A B : Type} (f : A → list B) (lst : list A) : list B :=
  fold_right (fun a bs ⇒ f a ++ bs) nil lst.
Lemma concat_map_app : ∀ {A B : Type} (f : A → list B) (l1 l2 : list A),
  concat_map f (l1 ++ l2) = (concat_map f l1) ++ (concat_map f l2).
Proof with auto.
  intros.
  induction l1.
  simpl...
  simpl.
  rewrite ← app_assoc.
  rewrite → IHl1...
```

```
Qed.

Definition filter_map_body {A B : Type} (f : A → option B) a bs := match f a with
  | Some b ⇒ b :: bs
  | None ⇒ bs
end.

Definition filter_map {A B : Type} (f : A → option B) (lst : list A) :=
  fold_right (filter_map_body f) nil lst.

Lemma filter_map_app : ∀ (A B : Type) (f : A → option B) (lst1 lst2 : list A),
  filter_map f (lst1 ++ lst2) = filter_map f lst1 ++ filter_map f lst2.
Proof with auto.
  intros.
  induction lst1...
  simpl.
  rewrite → IHlst1.
  unfold filter_map_body.
  destruct (f a)...
Qed.

Definition intersperse {A : Type} (v : A) (lst : list A) : list A :=
  fold_right (fun x xs ⇒ x :: v :: xs) nil lst.

Lemma nil_cons_false : ∀ {A : Type} (x : A) (xs : list A),
  nil = xs ++ [x] → False.
Proof with auto.
  intros.
  destruct xs; simpl in H; inversion H.
Qed.
```

```
Hint Resolve nil_cons_false : datatypes.

Lemma cons_tail : ∀ {A : Type} (x y : A) (l1 l2 : list A),
  l1 ++ [x] = l2 ++ [y] → l1 = l2 ∧ x = y.
Proof with auto with datatypes.
  intros A x y l1.
  induction l1; intros...
Qed.

Hint Resolve cons_tail : datatypes.

End List.
```

### A.2.14 Utilities Library

```
Require Export Common.CpdtTactics.

Notation "[ ]" := nil : list_scope.
Require Import Lists.List.
Notation "[ a ; .. ; b ]" := (a :: .. (b :: []) ..) : list_scope.
Require Import Bool.Bool.

Lemma app_nil :
  ∀ A (ls : list A),
    (app ls []) = ls.
Proof.
  crush.
Qed.

Hint Rewrite app_nil.
```

```
Lemma app_cons :
  ∀ A (e : A) l,
    [e] ++ l = e :: l.
Proof.
  crush.
Qed.

Hint Rewrite app_nil.

Lemma hd_error_app :
  ∀ A (e : A) l1 l2,
    hd_error l1 = value e →
    hd_error (l1 ++ l2) = value e.
Proof.
  induction l1; crush. inversion H.
Qed.

Hint Rewrite hd_error_app.

Fixpoint last_error {A : Type} (l : list A) :=
  match l with
    | [] ⇒ error
    | [a] ⇒ value a
    | a::l' ⇒ last_error l'
  end.

Lemma last_error_error_is_nil :
  ∀ A (l : list A),
    last_error l = error → l = [].
Proof.
```

```
  induction l; crush.
  destruct l. inversion H.
  apply IHl in H. inversion H.
Qed.

Lemma last_error_if_nil :
  ∀ A (l : list A),
    last_error l = error ↔ l = [].
Proof.
  intros.
  split; crush. apply last_error_error_is_nil. assumption.
Qed.

Lemma last_error_non_nil :
  ∀ A a (l : list A),
    last_error l = value a → l ≠ [].
Proof.
  red in ⊢ ×.
  intros.
  rewrite ← last_error_if_nil in H0. rewrite H in H0. inversion H0.
Qed.

Lemma last_error_app :
  ∀ A (e : A) l1 l2,
    last_error l2 = value e →
    last_error (l1 ++ l2) = value e.
Proof.
  induction l1; crush. apply IHl1 in H. apply last_error_non_nil in H. crush.
  destruct (l1 ++ l2); crush.
Qed.

Definition override {A : Type} {B : Type} (f : A → B) (g : A → option B) (a' : A) :=
  match g a' with
    | None ⇒ f a'
    | Some b' ⇒ b'
  end.

Fixpoint override_list {A : Type} {B : Type} (f : A → B) (us : list (A → option B)) :=
  match us with
    | [] ⇒ f
    | g :: gs ⇒ override (override_list f gs) g
  end.

Fixpoint snoc {A} (elem : A) lst :=
  match lst with
    | [] ⇒ [ elem ]
    | x :: lst ⇒ x :: snoc elem lst
  end.

Lemma app_snoc :
  ∀ A (l1 : list A) (e : A) l2,
    app l1 (snoc e l2) = snoc e (app l1 l2).
Proof.
  intros. induction l1; crush.
Qed.

Lemma override_list_override {A : Type} {B : Type} :
  ∀ (S : A → B) u us,
```

$\quad\quad$ *override_list* (*override* $S$ $u$) *us* = *override_list* $S$ (*snoc* $u$ *us*).

`Proof`.

$\quad$ `intros`. `induction` *us*; *crush*.

`Qed`.

`Lemma` *override_empty* :

$\quad$ $\forall$ $A$ $B$ ($F$ : $A \rightarrow B$),

$\quad\quad$ *override_list* $F$ [] = $F$.

`Proof`.

$\quad$ *crush*.

`Qed`.

`Hint Rewrite` *override_empty*.

`Lemma` *snoc_not_nil* :

$\quad$ $\forall$ $A$ ($t$ : $A$) *tr*,

$\quad\quad$ *snoc* $t$ *tr* $\neq$ [].

`Proof`.

$\quad$ `induction` *tr*; *crush*.

`Qed`.

`Lemma` *snoc_singleton* :

$\quad$ $\forall$ $A$ ($t$ : $A$) *t'* *tr*,

$\quad\quad$ *snoc* $t$ *tr* = [*t'*] $\rightarrow$ *tr* = [] $\wedge$ $t$ = *t'*.

`Proof`.

$\quad$ `induction` *tr*; *crush*; `apply` *snoc_not_nil* `in` $H$; *crush*.

`Qed`.

`Lemma` *override_list_app* :

$\quad$ $\forall$ $A$ $B$ ($f$ : $A \rightarrow B$) *f'* *f''* *a1* *a2*,

```
    f' = override_list f a1 →
    f'' = override_list f' a2 →
    f'' = override_list f (a2 ++ a1).
Proof.
  crush. induction a2; crush.
Qed.

Lemma app_is_nil :
  ∀ A (l1 : list A) l2,
    [] = l1 ++ l2 → l1 = [] ∧ l2 = [].
Proof.
  crush. induction l1; induction l2; crush. symmetry in H; apply app_eq_nil in H;
crush.
Qed.

Lemma in_snoc :
  ∀ A a1 a2 (l1 : list A),
    In a1 l1 → In a1 (snoc a2 l1).
Proof.
  intros. induction l1; crush.
Qed.

Lemma in_snoc2 :
  ∀ A a1 (l1 : list A),
    In a1 (snoc a1 l1).
Proof.
  intros. induction l1; crush.
Qed.
```

```
Lemma in_snoc3 :
  ∀ A a2 a1 (l1 : list A),
    a1 ≠ a2 →
    In a1 (snoc a2 l1) →
    In a1 l1.
Proof.
  induction l1; crush.
Qed.

Lemma snoc_app:
  ∀ A (a1 : A) l1,
    snoc a1 l1 = l1 ++ [a1].
Proof.
  induction l1; crush.
Qed.

Lemma snoc_cons :
  ∀ A (a1 : A) a2 l1,
    a1 :: snoc a2 l1 = snoc a2 (a1 :: l1).
Proof.
  induction l1; crush.
Qed.

Lemma snoc_inj :
  ∀ A (a1 : A) l1 a2 l2,
    snoc a1 l1 = snoc a2 l2 →
    a1 = a2 ∧ l1 = l2.
Proof.
```

```
  induction l1 ; induction l2; crush. symmetry in H. apply snoc_not_nil in H. contradiction.
  symmetry in H. apply snoc_not_nil in H. contradiction.
  apply snoc_not_nil in H. contradiction.
  apply snoc_not_nil in H. contradiction.
  specialize (IHl1 a2 l2). intuition.
  specialize (IHl1 a2 l2). crush.
Qed.

Hint Rewrite rev_involutive.

Lemma snoc_rev :
  ∀ A (a : A) ls,
    snoc a ls = rev (a :: rev ls).
Proof.
  crush. induction ls; crush.
Qed.

Lemma snoc_inv :
  ∀ A (ls : list A),
    ls = [] ∨ ∃ a, ∃ b, ls = snoc a b.
Proof.
  crush. induction ls; crush.
  right. ∃ a, []. crush.
  right. rewrite snoc_cons. ∃ x, (a :: x0). reflexivity.
Qed.

Lemma snoc_induction :
  ∀ A (P : list A → Prop),
    (P []) →
    (∀ a ls, P ls → P (snoc a ls)) →
    ∀ ls, P ls.
Proof.
  intros. apply rev_ind. assumption.
  intros. rewrite ← snoc_app. crush.
Qed.

Lemma snoc_double_induction :
  ∀ A (P : list A → list A → Prop),
    (P [] []) →
    (∀ a l1 l2, P l1 l2 → P (snoc a l1) l2) →
    (∀ a l1 l2, P l1 l2 → P l1 (snoc a l2)) →
    ∀ l1 l2, P l1 l2.
Proof.
  destruct l1 using snoc_induction; destruct l2 using snoc_induction; crush.
Qed.

Ltac snoc_nil_tac :=
  match goal with
    | [ H : snoc _ _ = [] ⊢ _ ] ⇒ apply snoc_not_nil in H; contradiction
    | [ H : [] = snoc _ _ ⊢ _ ] ⇒ symmetry in H; apply snoc_not_nil in H; contradiction
  end.

Ltac snoc_singleton_tac :=
  match goal with
    | [ H : snoc _ ?A = [_] ⊢ _ ] ⇒
      match goal with
        | [ H' : ?A = [] ⊢ _ ] ⇒ fail 2
        |_ ⇒ apply snoc_singleton in H
      end
    | [ H : [_] = snoc _ _ ⊢ _ ] ⇒ symmetry in H; snoc_singleton_tac
  end.

Ltac snoc_inj_tac :=
  match goal with
    | [ H : snoc _ _ = snoc _ _ ⊢ _ ] ⇒ apply snoc_inj in H; subst
  end.

Ltac in_snoc_tac :=
  match goal with
    | [ H : In ?A ?B ⊢ In ?A (snoc _ ?B) ] ⇒ apply in_snoc; assumption
  end.

Ltac remember_clear H H' :=
  remember H as H';
    match goal with [ H'' : H' = H ⊢ _ ] ⇒ clear H'' end.

Ltac snoc_tac' :=
  match goal with
    | [ ⊢ [] ≠ snoc _ _ ] ⇒ intuition; snoc_nil_tac
    | [ ⊢ snoc _ _ ≠ [] ] ⇒ intuition; snoc_nil_tac
  end.

Ltac snoc_tac :=
  snoc_nil_tac || snoc_tac' || snoc_inj_tac || snoc_singleton_tac || in_snoc_tac.

Ltac nil_cons_tac :=
  match goal with
    | [ H : [] = _ :: _ ⊢ _ ] ⇒ discriminate
    | [ H : _ :: _ = [] ⊢ _ ] ⇒ discriminate
  end.

Ltac util_crush :=
  repeat (snoc_tac || crush || nil_cons_tac).

Require Import Coq.Classes.Equivalence.
Require Import Coq.Classes.EquivDec.
Require Import Coq.Logic.Decidable.

Hint Extern 0 (?x === ?x) ⇒ reflexivity.
Hint Extern 1 (_ === _) ⇒ (symmetry; trivial; fail).
Hint Extern 1 (_ =/= _) ⇒ (symmetry; trivial; fail).

Lemma equiv_reflexive' : ∀ {A} `{EqDec A} (x : A),
  x === x.
Proof.
  intros. apply equiv_reflexive.
Qed.

Lemma equiv_symmetric' : ∀ {A} `{EqDec A} (x y : A),
  x === y →
  y === x.
Proof.
  intros. apply equiv_symmetric; assumption.
Qed.

Lemma equiv_transitive' : ∀ {A} `{EqDec A} (x y z : A),
  x === y →
  y === z →
  x === z.
```

```
Proof.
  intros. eapply @equiv_transitive; eassumption.
Qed.

Lemma equiv_decidable : ∀ {A} `{EqDec A} (x y : A),
  decidable (x === y).
Proof.
  intros. unfold decidable. destruct (x == y); auto.
Defined.

Class EqDec_eq (A : Type) :=
eq_dec : ∀ (x y : A), {x = y} + {x ≠ y}.

Instance EqDec_eq_of_EqDec {A} `(@EqDec A eq eq_equivalence) : EqDec_eq A.
Proof.
  trivial.
Defined.

Notation " x == y " := (eq_dec (x :>) (y :>)) (no associativity, at level 70) :
equiv_scope.

Definition equiv_decb' {A} `{EqDec_eq A} (x y : A) : bool :=
  if x == y then true else false.

Definition nequiv_decb' {A} `{EqDec_eq A} (x y : A) : bool :=
  negb (equiv_decb' x y).

Infix "==b" := equiv_decb' (no associativity, at level 70).

Infix "<>b" := nequiv_decb' (no associativity, at level 70).

Lemma eq_option_dec : ∀ {A} `{EqDec_eq A} (x y : option A),
  {x = y} + {x ≠ y}.
```

```
Proof.
  repeat decide equality.
Qed.

Lemma eq_prod_dec : ∀ {A B} `{EqDec_eq A} `{EqDec_eq B} (x : A×B) (y : A×B),
  {x = y} + {x ≠ y}.
Proof.
  repeat decide equality.
Qed.

Instance EqDec_of_prod_EqDec {A B} `{EqDec_eq A} `{EqDec_eq B} : EqDec (A×B) eq
:= eq_prod_dec.

Definition updateFun {A : Type} {B : Type} `{eqA : EqDec_eq A} (f : A → B) (a : A)
(b : B) (a' : A) :=
  if a ==b a' then b else f a'.

Notation "f  a |-> b  " :=
  (updateFun f a b) (at level 0).

Lemma update_fun_same_arg {A : Type} {B : Type} `{eqA : EqDec_eq A} :
  ∀ (f : A → B) a (b : B),
    updateFun f a b a = b.
Proof.
  intros. unfold updateFun. unfold equiv_decb'. destruct eq_dec; crush.
Qed.

Lemma update_fun_diff_arg {A : Type} {B : Type} `{eqA : EqDec_eq A} :
  ∀ f a (b : B) a',
    a≠a'→ updateFun f a b a' = f a'.
```

```
Proof.
  crush. unfold updateFun. case_eq (a ==b a'); crush. unfold equiv_decb' in H0.
destruct eq_dec; crush.
Qed.

Definition append_function {A : Type} {B : Type} `{eqA : EqDec_eq A} (f : A → list B)
(a : A) (b : B) (a' : A) :=
  if a ==b a' then b :: f a' else f a'.

Definition extend {A : Type} {B : Type} (f : A → list B) (g : A → list B) (a : A) :=
  f a ++ g a.

Definition extend2 {A : Type} {B : Type} {C : Type} (f : A → list B×list C) (g : A →
list B×list C) (a : A) :=
  (fst (f a) ++ fst (g a), snd (f a) ++ snd (g a)).

Fixpoint mem {A} `{EqDec_eq A} (a : A) (l : list A) :=
  match l with
    | [] ⇒ false
    | b :: m ⇒ if a ==b b then true else mem a m
  end.
```

### A.2.15 Extract Library

```
Require Import PArith.BinPos.
Require Import NArith.BinNat.
Require Import OpenFlow.OpenFlow0x01Types.
Require Import NetCore.NetCoreController.
Require Import Pattern.PatternInterface.
```

Require Import *FwOF.FwOFExtractableController*.

Require Import *Extraction.OCaml*.

Cd "../../ocaml/extracted".

Recursive Extraction *Library NetCoreController*.

Recursive Extraction *Library PatternInterface*.

Recursive Extraction *Library FwOFExtractableController*.

Recursive Extraction *Library OpenFlowTypes*.

### A.2.16 OCaml Library

Extraction *Language Ocaml*.

Require Import *Coq.Lists.List*.

Require Import *PArith.BinPos*.

Require Import *NArith.BinNat*.

Require Import *Common.Types*.

Require Import *ExtrOcamlBasic*.

Require Import *ExtrOcamlString*.

Require Import *ExtrOcamlNatInt*.

Extraction *Blacklist String List*.

Extract *Constant destruct_list* ⇒

"fun _-> failwith ""destruct_list axiom""".

Extract *Constant exists_last* ⇒

"fun _-> failwith ""exists_last axiom""".

Extract *Constant nth_in_or_default* ⇒

"fun ___-> failwith ""nth_in_or_default axiom""".

Extract Inductive *comparison* ⇒ "int" [ "0" "(-1)" "1" ].

### A.2.17 FwOFBisimulation Library

Set Implicit Arguments.

Require Import *Coq.Classes.Equivalence.*

Require Import *Common.Bisimulation.*

Require Import *FwOF.FwOFSignatures.*

Require *FwOF.FwOFRelationDefinitions.*

Require *FwOF.FwOFWellFormedness.*

Require *FwOF.FwOFWeakSimulation1.*

Require *FwOF.FwOFWeakSimulation2.*

Local Open Scope *equiv_scope*.

Module *Make* (*AtomsAndController* : *ATOMS_AND_CONTROLLER*).

  Module *RelationDefinitions* := *FwOF.FwOFRelationDefinitions.Make* (*AtomsAndController*).

  Module *Relation* := *FwOF.FwOFWellFormedness.Make* (*RelationDefinitions*).

  Module *WeakSim1* := *FwOF.FwOFWeakSimulation1.Make* (*RelationDefinitions*).

  Module *WeakSim2* := *FwOF.FwOFWeakSimulation2.Make* (*Relation*).

  Import *Relation.*

  Import *RelationDefinitions.*

  Theorem *fwof_abst_weak_bisim* :

    *weak_bisimulation concreteStep abstractStep bisim_relation.*

  Proof.

```
    unfold weak_bisimulation.
    split.
    exact WeakSim1.weak_sim_1.
    exact WeakSim2.weak_sim_2.
  Qed.
End Make.
```

### A.2.18 FwOFExtractableController Library

```
Set Implicit Arguments.
Require Import Coq.Lists.List.
Require Import Common.Types.
Require Import FwOF.FwOFExtractableSignatures.
Require OpenFlow.OpenFlow0x01Types.
Require Network.NetworkPacket.
Require NetCore.NetCoreEval.
Require Import Pattern.Pattern.
Local Open Scope list_scope.
Module Type POLICY.
  Require Import OpenFlow.OpenFlow0x01Types.
  Require Import Network.NetworkPacket.
  Parameter abst_func : switchId → portId → (packet × bufferId) → list (portId × (packet
× bufferId)).
End POLICY.
```

```
Module MakeAtoms (Policy : POLICY) <: EXTRACTABLE_ATOMS.
  Definition switchId := OpenFlow.OpenFlow0x01Types.switchId.
  Definition portId := Network.NetworkPacket.portId.
  Definition packet := (Network.NetworkPacket.packet × OpenFlow.OpenFlow0x01Types.bufferId)
%type.
  Definition flowTable :=
    list (nat × pattern × list (NetCore.NetCoreEval.act)).
  Inductive fm : Type :=
  | AddFlow : nat → pattern → list (NetCore.NetCoreEval.act) → fm.
  Definition flowMod := fm.
  Inductive fromController : Type :=
  | PacketOut : portId → packet → fromController
  | BarrierRequest : nat → fromController
  | FlowMod : flowMod → fromController.
  Inductive fromSwitch : Type :=
  | PacketIn : portId → packet → fromSwitch
  | BarrierReply : nat → fromSwitch.
  Definition abst_func := Policy.abst_func.
End MakeAtoms.
Module MakeController (Atoms_ : EXTRACTABLE_ATOMS) <: EXTRACTABLE_CONTROLLER.
  Import Atoms_.
  Record switchState := SwitchState {
    theSwId : switchId;
    pendingCtrlMsgs : list fromController
  }.

  Record srcDst := SrcDst {
    pkSw : switchId;
    srcPt : portId;
    srcPk : packet;
    dstPt : portId;
    dstPk : packet
  }.

  Record state := State {
    pktsToSend : list srcDst;
    switchStates : list switchState
  }.

  Definition mkPktOuts_body sw srcPt srcPk ptpk :=
    match ptpk with
      | (dstPt,dstPk) ⇒ SrcDst sw srcPt srcPk dstPt dstPk
    end.

  Definition mkPktOuts (sw : switchId) (srcPt : portId) (srcPk : packet) :=
    map (mkPktOuts_body sw srcPt srcPk)
      (abst_func sw srcPt srcPk).

  Definition controller := state.

  Fixpoint send_queued (swsts : list switchState) : option (list switchState × switchId ×
fromController) :=
    match swsts with
      | nil ⇒ None
      | (SwitchState sw (msg :: msgs)) :: ss ⇒
```

```
        Some (SwitchState sw msgs :: ss, sw, msg)
      | (SwitchState sw nil) :: ss ⇒
        match send_queued ss with
          | None ⇒ None
          | Some (ss', sw',msg) ⇒ Some (SwitchState sw nil :: ss', sw', msg)
        end
    end.
Fixpoint send (st : state) : option (state × switchId × fromController) :=
    match st with
      | State ((SrcDst sw _ _ pt pk) :: pks) sws ⇒
        Some (State pks sws, sw, PacketOut pt pk)
      | State nil ss ⇒
        match send_queued ss with
          | None ⇒ None
          | Some (ss', sw, msg) ⇒ Some (State nil ss', sw, msg)
        end
    end.
Fixpoint recv (st : state) (sw : switchId) (msg : fromSwitch) :=
    match msg with
      | BarrierReply _ ⇒ st
      | PacketIn pt pk ⇒
        match st with
          | State pktOuts ss ⇒
              State (mkPktOuts sw pt pk ++ pktOuts) ss
        end
    end.

  Module Atoms := Atoms_.

End MakeController.
```

### A.2.19 FwOFExtractableSignatures Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Import ListNotations.

Local Open Scope list_scope.

Module Type EXTRACTABLE_ATOMS.

  Parameter packet : Type.

  Parameter switchId : Type.

  Parameter portId : Type.

  Parameter flowTable : Type.

  Parameter flowMod : Type.

  Inductive fromController : Type :=
  | PacketOut : portId → packet → fromController
  | BarrierRequest : nat → fromController
  | FlowMod : flowMod → fromController.

  Inductive fromSwitch : Type :=
  | PacketIn : portId → packet → fromSwitch
  | BarrierReply : nat → fromSwitch.

  Parameter abst_func : switchId → portId → packet → list (portId × packet).
```

```
End EXTRACTABLE_ATOMS.

Module Type EXTRACTABLE_CONTROLLER.

  Declare Module Atoms : EXTRACTABLE_ATOMS.

  Import Atoms.

  Parameter controller : Type.

  Parameter send : controller → option (controller × switchId × fromController).

  Parameter recv : controller → switchId → fromSwitch → controller.

End EXTRACTABLE_CONTROLLER.
```

### A.2.20 FwOFMachine Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Bag.TotalOrder.

Require Import Bag.Bag2.

Require Import Common.Types.

Require Import FwOF.FwOFSignatures.

Local Open Scope list_scope.

Local Open Scope equiv_scope.

Local Open Scope bag_scope.

Module Make (Atoms_ : ATOMS) <: MACHINE.

  Module Atoms := Atoms_.

  Import Atoms.

  Existing Instances TotalOrder_packet TotalOrder_switchId TotalOrder_portId
  TotalOrder_flowTable TotalOrder_flowMod TotalOrder_fromSwitch
  TotalOrder_fromController.

Record switch := Switch {
  swId : switchId;
  pts : list portId;
  tbl : flowTable;
  inp : bag (PairOrdering portId_le packet_le);
  outp : bag (PairOrdering portId_le packet_le);
  ctrlm : bag fromController_le;
  switchm : bag fromSwitch_le
}.

Inductive switch_le : switch → switch → Prop :=
| SwitchLe : ∀ sw1 sw2,
    switchId_le (swId sw1) (swId sw2) →
    switch_le sw1 sw2.

Axiom Instance TotalOrder_switch : TotalOrder switch_le.

Record dataLink := DataLink {
  src : switchId × portId;
  pks : list packet;
  dst : switchId × portId
}.

Record openFlowLink := OpenFlowLink {
  of_to : switchId;
  of_switchm : list fromSwitch;
  of_ctrlm : list fromController
}.

Definition observation := (switchId × portId × packet) %type.

Reserved Notation "SwitchStep[ sw ; obs ; sw0 ]"
  (at level 70, no associativity).
Reserved Notation "ControllerOpenFlow[ c ; l ; obs ; c0 ; l0 ]"
  (at level 70, no associativity).
Reserved Notation "TopoStep[ sw ; link ; obs ; sw0 ; link0 ]"
  (at level 70, no associativity).
Reserved Notation "SwitchOpenFlow[ s ; l ; obs ; s0 ; l0 ]"
  (at level 70, no associativity).

Inductive NotBarrierRequest : fromController → Prop :=
| PacketOut_NotBarrierRequest : ∀ pt pk,
    NotBarrierRequest (PacketOut pt pk)
| FlowMod_NotBarrierRequest : ∀ fm,
    NotBarrierRequest (FlowMod fm).
```

Devices of the same type do not interact in a single step. Therefore, we never have to permute the lists below. If we instead had just one list of all devices, we would have to worry about permuting the list or define symmetric step-rules.

```
Record state := State {
  switches : bag switch_le;
  links : list dataLink;
  ofLinks : list openFlowLink;
  ctrl : controller
}.

Inductive step : state → option observation → state → Prop :=
| PktProcess : ∀ swId pts tbl pt pk inp outp ctrlm switchm outp'
                                   pksToCtrl,
  process_packet tbl pt pk = (outp', pksToCtrl) →
  SwitchStep[
    Switch swId pts tbl ({|(pt,pk)|} <+> inp) outp ctrlm switchm;
    Some (swId,pt,pk);
    Switch swId pts tbl inp (from_list outp' <+> outp)
      ctrlm (from_list (map (PacketIn pt) pksToCtrl) <+> switchm)
  ]
| ModifyFlowTable : ∀ swId pts tbl inp outp fm ctrlm switchm,
  SwitchStep[
    Switch swId pts tbl inp outp ({|FlowMod fm|} <+> ctrlm) switchm;
    None;
    Switch swId pts (modify_flow_table fm tbl) inp outp ctrlm switchm
  ]
| SendPacketOut : ∀ pt pts swId tbl inp outp pk ctrlm switchm,
  SwitchStep[
    Switch swId pts tbl inp outp ({|PacketOut pt pk|} <+> ctrlm) switchm;
    None;
    Switch swId pts tbl inp ({| (pt,pk) |} <+> outp) ctrlm switchm
  ]
| SendDataLink : ∀ swId pts tbl inp pt pk outp ctrlm switchm pks dst,
  TopoStep[
    Switch swId pts tbl inp ({|(pt,pk)|} <+> outp) ctrlm switchm;
    DataLink (swId,pt) pks dst;
    None;
```

```
      Switch swId pts tbl inp outp ctrlm switchm;
      DataLink (swId,pt) (pk :: pks) dst
    ]
| RecvDataLink : ∀ swId pts tbl inp outp ctrlm switchm src pks pk pt,
    TopoStep[
      Switch swId pts tbl inp outp ctrlm switchm;
      DataLink src (pks ++ [pk]) (swId,pt);
      None;
      Switch swId pts tbl ({|(pt,pk)|} <+> inp) outp ctrlm switchm;
      DataLink src pks (swId,pt)
    ]
| Step_controller : ∀ sws links ofLinks ctrl ctrl',
    controller_step ctrl ctrl' →
    step (State sws links ofLinks ctrl)
           None
           (State sws links ofLinks ctrl')
| ControllerRecv : ∀ ctrl msg ctrl' swId fromSwitch fromCtrl,
    controller_recv ctrl swId msg ctrl' →
    ControllerOpenFlow[
      ctrl;
      OpenFlowLink swId (fromSwitch ++ [msg]) fromCtrl;
      None;
      ctrl';
      OpenFlowLink swId fromSwitch fromCtrl
    ]
| ControllerSend : ∀ ctrl msg ctrl' swId fromSwitch fromCtrl,
    controller_send ctrl ctrl' swId msg →
    ControllerOpenFlow[
      ctrl ;
      (OpenFlowLink swId fromSwitch fromCtrl);
      None;
      ctrl';
      (OpenFlowLink swId fromSwitch (msg :: fromCtrl)) ]
| SendToController : ∀ swId pts tbl inp outp ctrlm msg switchm fromSwitch
      fromCtrl,
    SwitchOpenFlow[
      Switch swId pts tbl inp outp ctrlm ({| msg |} <+> switchm);
      OpenFlowLink swId fromSwitch fromCtrl;
      None;
      Switch swId pts tbl inp outp ctrlm switchm;
      OpenFlowLink swId (msg :: fromSwitch) fromCtrl
    ]
| RecvBarrier : ∀ swId pts tbl inp outp switchm fromSwitch fromCtrl
      xid,
    SwitchOpenFlow[
      Switch swId pts tbl inp outp empty switchm;
      OpenFlowLink swId fromSwitch (fromCtrl ++ [BarrierRequest xid]);
      None;
      Switch swId pts tbl inp outp empty
             ({| BarrierReply xid |} <+> switchm);
      OpenFlowLink swId fromSwitch fromCtrl
    ]
```

```
| RecvFromController : ∀ swId pts tbl inp outp ctrlm switchm
    fromSwitch fromCtrl msg,
  NotBarrierRequest msg →
  SwitchOpenFlow[
    Switch swId pts tbl inp outp ctrlm switchm;
    OpenFlowLink swId fromSwitch (fromCtrl ++ [msg]);
    None;
    Switch swId pts tbl inp outp ({| msg |} <+> ctrlm) switchm;
    OpenFlowLink swId fromSwitch fromCtrl
  ]
    where
"ControllerOpenFlow[ c ; l ; obs ; c0 ; l0 ]" :=
  (∀ sws links ofLinks ofLinks',
    step (State sws links (ofLinks ++ l :: ofLinks') c)
         obs
         (State sws links (ofLinks ++ l0 :: ofLinks') c0))
  and
"TopoStep[ sw ; link ; obs ; sw0 ; link0 ]" :=
  (∀ sws links links0 ofLinks ctrl,
    step
    (State (({|sw|}) <+> sws) (links ++ link :: links0) ofLinks ctrl)
    obs
    (State (({|sw0|}) <+> sws) (links ++ link0 :: links0) ofLinks ctrl))
  and
"SwitchStep[ sw ; obs ; sw0 ]" :=
  (∀ sws links ofLinks ctrl,
    step
      (State (({|sw|}) <+> sws) links ofLinks ctrl)
      obs
      (State (({|sw0|}) <+> sws) links ofLinks ctrl))
  and
"SwitchOpenFlow[ sw ; of ; obs ; sw0 ; of0 ]" :=
  (∀ sws links ofLinks ofLinks0 ctrl,
    step
      (State (({|sw|}) <+> sws) links (ofLinks ++ of :: ofLinks0) ctrl)
      obs
      (State (({|sw0|}) <+> sws) links (ofLinks ++ of0 :: ofLinks0) ctrl)).

Definition swPtPks : Type :=
  bag (PairOrdering (PairOrdering switchId_le portId_le)
                    packet_le).

Definition abst_state := swPtPks.

Definition transfer (sw : switchId) (ptpk : portId × packet) :=
  match ptpk with
    | (pt,pk) ⇒
      match topo (sw,pt) with
        | Some (sw',pt') ⇒
          @singleton _
             (PairOrdering
                (PairOrdering switchId_le portId_le) packet_le)
             (sw',pt',pk)
        | None ⇒ {| |}
      end
  end.
Definition select_packet_out (sw : switchId) (msg : fromController) :=
  match msg with
    | PacketOut pt pk ⇒ transfer sw (pt,pk)
    | _ ⇒ {| |}
  end.
Definition select_packet_in (sw : switchId) (msg : fromSwitch) :=
  match msg with
    | PacketIn pt pk ⇒ unions (map (transfer sw) (abst_func sw pt pk))
    | _ ⇒ {| |}
  end.
Definition FlowTableSafe (sw : switchId) (tbl : flowTable) : Prop :=
  ∀ pt pk forwardedPkts packetIns,
    process_packet tbl pt pk = (forwardedPkts, packetIns) →
    unions (map (transfer sw) forwardedPkts) <+>
    unions (map (select_packet_in sw) (map (PacketIn pt) packetIns)) =
    unions (map (transfer sw) (abst_func sw pt pk)).
Inductive NotFlowMod : fromController → Prop :=
| NotFlowMod_BarrierRequest : ∀ n, NotFlowMod (BarrierRequest n)
| NotFlowMod_PacketOut : ∀ pt pk, NotFlowMod (PacketOut pt pk).
Inductive FlowModSafe : switchId → flowTable → bag fromController_le → Prop :=
| NoFlowModsInBuffer : ∀ swId tbl ctrlm,
    (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
    FlowTableSafe swId tbl →
```

```
      FlowModSafe swId tbl ctrlm
  | OneFlowModsInBuffer : ∀ swId tbl ctrlm f,
      (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
      FlowTableSafe swId tbl →
      FlowTableSafe swId (modify_flow_table f tbl) →
      FlowModSafe swId tbl (({|FlowMod f|}) <+> ctrlm).

  Definition FlowTablesSafe (sws : bag switch_le) : Prop :=
    ∀ swId pts tbl inp outp ctrlm switchm,
      In (Switch swId pts tbl inp outp ctrlm switchm) (to_list sws) →
      FlowModSafe swId tbl ctrlm.

  Definition SwitchesHaveOpenFlowLinks (sws : bag switch_le) ofLinks :=
    ∀ sw,
      In sw (to_list sws) →
      ∃ ofLink,
        In ofLink ofLinks ∧
        swId sw = of_to ofLink.

End Make.
```

### A.2.21 FwOFNetworkAtoms Library

```
Set Implicit Arguments.

Require Import Bag.TotalOrder.

Require Import Coq.Lists.List.

Require Import Coq.Relations.Relations.

Require Import FwOF.FwOFSignatures.
Require Import Classifier.Classifier.

Require OpenFlow.OpenFlow0x01Types.

Require Network.NetworkPacket.

Require Network.PacketTotalOrder.

Require Import NetCore.NetCoreEval.

Require Import Pattern.Pattern.

Require Import Word.WordTheory.

Import ListNotations.

Local Open Scope list_scope.

Module NetworkAtoms <: NETWORK_ATOMS.

  Definition packet := (Network.NetworkPacket.packet × OpenFlow.OpenFlow0x01Types.bufferId)
% type.

  Definition switchId := OpenFlow.OpenFlow0x01Types.switchId.

  Definition portId := Network.NetworkPacket.portId.

  Definition flowTable :=

    list (nat × pattern × list (NetCore.NetCoreEval.act)).

  Inductive fm : Type :=

    | AddFlow : nat → pattern → list (NetCore.NetCoreEval.act) → fm.

  Definition flowMod := fm.

  Inductive fromController : Type :=

  | PacketOut : portId → packet → fromController

  | BarrierRequest : nat → fromController

  | FlowMod : flowMod → fromController.

  Inductive fromSwitch : Type :=

  | PacketIn : portId → packet → fromSwitch
  | BarrierReply : nat → fromSwitch.

  Definition strip_prio (x : nat × pattern × list (NetCore.NetCoreEval.act)) :=
    match x with
      | (prio,pat,act) ⇒ (pat,Some act)
    end.

  Require Import Common.Types.

  Definition eval_act (pt : portId) (pk : packet) (act : act) :=
    match act with


      | Forward _ (OpenFlow.OpenFlow0x01Types.PhysicalPort pt') ⇒ [(pt',pk)]


      | _ ⇒ nil
    end.
```

Produces a list of packets to forward out of ports, and a list of packets to send to the controller.

```
Definition process_packet (tbl : flowTable) (pt : portId) (pk : packet) :=
  match pk with
    | (actualPk, buf) ⇒
        match scan None (map strip_prio tbl) pt actualPk with
          | None ⇒ (nil, [pk])
          | Some acts ⇒ (concat_map (eval_act pt pk) acts, nil)
        end
  end.

Definition modify_flow_table (fm : flowMod) (ft : flowTable) :=
  match fm with
    | AddFlow prio pat act ⇒
```

```
      (prio,pat,act) :: ft
  end.

Section TotalOrderings.

  Definition proj_fromController msg :=
    match msg with
      | PacketOut pt pk ⇒ inl (pt, pk)
      | BarrierRequest n ⇒ inr (inl n)
      | FlowMod f ⇒ inr (inr f)
    end.

  Definition inj_fromController sum :=
    match sum with
      | inl (pt, pk) ⇒ PacketOut pt pk
      | inr (inl n ) ⇒ BarrierRequest n
      | inr (inr f) ⇒ FlowMod f
    end.

  Definition proj_fromSwitch msg :=
    match msg with
      | PacketIn pt pk ⇒ inl (pt, pk)
      | BarrierReply n ⇒ inr n
    end.

  Definition inj_fromSwitch sum :=
    match sum with
      | inl (pt, pk) ⇒ PacketIn pt pk
      | inr n ⇒ BarrierReply n
    end.

  End TotalOrderings.

  Definition packet_le := PairOrdering Network.PacketTotalOrder.packet_le Word32.le.

  Definition switchId_le := Word64.le.

  Definition portId_le := Word16.le.

  Parameter flowTable_le : Relation_Definitions.relation flowTable.

  Parameter flowMod_le : Relation_Definitions.relation flowMod.

  Definition fromSwitch_le :=
    ProjectOrdering proj_fromSwitch (SumOrdering (PairOrdering portId_le packet_le) le).

  Definition fromController_le :=
    ProjectOrdering proj_fromController (SumOrdering (PairOrdering portId_le packet_le)
(SumOrdering le flowMod_le)).

  Instance TotalOrder_packet : TotalOrder packet_le.
  Proof. apply TotalOrder_pair; auto. exact Network.PacketTotalOrder.TotalOrder_packet.
Qed.

  Definition TotalOrder_switchId := Word64.TotalOrder.

  Definition TotalOrder_portId := Word16.TotalOrder.

  Instance TotalOrder_flowMod : TotalOrder flowMod_le.
  Admitted.

  Instance TotalOrder_flowTable : TotalOrder flowTable_le.
  Admitted.

  Instance TotalOrder_fromController : TotalOrder fromController_le.
  Proof with auto.
    apply TotalOrder_Project with (g := inj_fromController)...
    + apply TotalOrder_sum.
      apply TotalOrder_pair.
      apply TotalOrder_portId.
      apply TotalOrder_packet.
      apply TotalOrder_sum.
      apply TotalOrder_nat.
      apply TotalOrder_flowMod.
    + unfold inverse.
      intros.
      destruct x...
  Qed.

  Instance TotalOrder_fromSwitch : TotalOrder fromSwitch_le.
  Proof with auto.
    apply TotalOrder_Project with (g := inj_fromSwitch)...
    + apply TotalOrder_sum.
      apply TotalOrder_pair.
      apply TotalOrder_portId.
      apply TotalOrder_packet.
      apply TotalOrder_nat.
    + unfold inverse.
      intros.
      destruct x...
  Qed.

End NetworkAtoms.
```

### A.2.22 FwOFRelationDefinitions Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Coq.Relations.Relations.

Require Import Common.Types.

Require Import Bag.TotalOrder.

Require Import Bag.Bag2.

Require Import Common.AllDiff.

Require Import Common.Bisimulation.

Require Import FwOF.FwOFSignatures.

Local Open Scope list_scope.

Local Open Scope equiv_scope.

Local Open Scope bag_scope.
```

This is a really trivial functor. RELATION_DEFINITIONS is just a bunch of definitions. `Module` *Make* (`Import` *AtomsAndController* : *ATOMS_AND_CONTROLLER*) <: *RELATION_DEFINITIONS*.

```
  Import AtomsAndController.

  Import Machine.

  Import Atoms.

  Definition affixSwitch (sw : switchId) (ptpk : portId × packet) :=
    match ptpk with
      | (pt,pk) ⇒ (sw,pt,pk)
    end.
```

```
Definition ConsistentDataLinks (links : list dataLink) : Prop :=
  ∀ (lnk : dataLink),
    In lnk links →
    topo (src lnk) = Some (dst lnk).
Definition LinkHasSrc (sws : bag switch_le) (link : dataLink) : Prop :=
  ∃ switch,
    In switch (to_list sws) ∧
    fst (src link) = swId switch ∧
    In (snd (src link)) (pts switch).
Definition LinkHasDst (sws : bag switch_le) (link : dataLink) : Prop :=
  ∃ switch,
    In switch (to_list sws) ∧
    fst (dst link) = swId switch ∧
    In (snd (dst link)) (pts switch).
Definition LinksHaveSrc (sws : bag switch_le) (links : list dataLink) :=
  ∀ link, In link links → LinkHasSrc sws link.
Definition LinksHaveDst (sws : bag switch_le) (links : list dataLink) :=
  ∀ link, In link links → LinkHasDst sws link.
Definition UniqSwIds (sws : bag switch_le) := AllDiff swId (to_list sws).
Definition ofLinkHasSw (sws : bag switch_le) (ofLink : openFlowLink) :=
  ∃ sw,
    In sw (to_list sws) ∧
    of_to ofLink = swId sw.
Definition OFLinksHaveSw (sws : bag switch_le) (ofLinks : list openFlowLink) :=
  ∀ ofLink, In ofLink ofLinks → ofLinkHasSw sws ofLink.
```

```
Definition DevicesFromTopo (devs : state) :=
  ∀ swId0 swId1 pt0 pt1,
    Some (swId0,pt0) = topo (swId1,pt1) →
    ∃ sw0 sw1 lnk,

      In sw0 (to_list (switches devs)) ∧
      In sw1 (to_list (switches devs)) ∧
      In lnk (links devs) ∧
      swId sw0 = swId0 ∧
      swId sw1 = swId1 ∧
      src lnk = (swId1,pt1) ∧
      dst lnk = (swId0, pt0).
Definition NoBarriersInCtrlm (sws : bag switch_le) :=
  ∀ sw,
    In sw (to_list sws) →
    ∀ m,
      In m (to_list (ctrlm sw)) →
      NotBarrierRequest m.
Record concreteState := ConcreteState {
  devices : state;
  concreteState_flowTableSafety : FlowTablesSafe (switches devices);
  concreteState_consistentDataLinks : ConsistentDataLinks (links devices);
  linksHaveSrc : LinksHaveSrc (switches devices) (links devices);
  linksHaveDst : LinksHaveDst (switches devices) (links devices);
  uniqSwIds : UniqSwIds (switches devices);
  ctrlP : P (switches devices) (ofLinks devices) (ctrl devices);
  uniqOfLinkIds : AllDiff of_to (ofLinks devices);
  ofLinksHaveSw : OFLinksHaveSw (switches devices) (ofLinks devices);
  devicesFromTopo : DevicesFromTopo devices;
  swsHaveOFLinks : SwitchesHaveOpenFlowLinks (switches devices) (ofLinks devices);
  noBarriersInCtrlm : NoBarriersInCtrlm (switches devices)
}.

Implicit Arguments ConcreteState [].

Definition concreteStep (st : concreteState) (obs : option observation)
  (st0 : concreteState) :=
  step (devices st) obs (devices st0).

Inductive abstractStep : abst_state → option observation → abst_state →
  Prop :=
| AbstractStep : ∀ sw pt pk lps,
  abstractStep
    ({| (sw,pt,pk) |} <+> lps)
    (Some (sw,pt,pk))
    (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps).

Definition relate_switch (sw : switch) : abst_state :=
  match sw with
    | Switch swId _ tbl inp outp ctrlm switchm ⇒
      from_list (map (affixSwitch swId) (to_list inp)) <+>
      unions (map (transfer swId) (to_list outp)) <+>
      unions (map (select_packet_out swId) (to_list ctrlm)) <+>
      unions (map (select_packet_in swId) (to_list switchm))
    end.

  Definition relate_dataLink (link : dataLink) : abst_state :=
    match link with
      | DataLink _ pks (sw,pt) ⇒
         from_list (map (fun pk ⇒ (sw,pt,pk)) pks)
    end.

  Definition relate_openFlowLink (link : openFlowLink) : abst_state :=
    match link with
      | OpenFlowLink sw switchm ctrlm ⇒
         unions (map (select_packet_out sw) ctrlm) <+>
         unions (map (select_packet_in sw) switchm)
    end.

  Definition relate (st : state) : abst_state :=
    unions (map relate_switch (to_list (switches st))) <+>
    unions (map relate_dataLink (links st)) <+>
    unions (map relate_openFlowLink (ofLinks st)) <+>
    relate_controller (ctrl st).

  Definition bisim_relation : relation concreteState abst_state :=
    fun (st : concreteState) (ast : abst_state) ⇒
      ast = (relate (devices st)).

  Module AtomsAndController := AtomsAndController.

End Make.
```

### A.2.23 FwOFSafeWire Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Common.Types.

Require Import Common.Bisimulation.

Require Import Bag.TotalOrder.

Require Import Bag.Bag2.

Require Import FwOF.FwOFSignatures.

Require Import Common.Bisimulation.

Require Import Common.AllDiff.

Require FwOF.FwOFMachine.

Require FwOF.FwOFSimpleController.

Local Open Scope list_scope.

Local Open Scope bag_scope.

Module Make (Import Machine : MACHINE).

  Import Atoms.

  Inductive NotPacketOut : fromController → Prop :=

  | BarrierRequest_NotPacketOut : ∀ xid,

                                        NotPacketOut (BarrierRequest xid)

  | FlowMod_NotPacketOut : ∀ fm,

                                  NotPacketOut (FlowMod fm).

  Hint Constructors NotPacketOut NotFlowMod.

  Inductive Alternating : bool → list fromController → Prop :=

  | Alternating_Nil : ∀ b, Alternating b nil
  | Alternating_PacketOut :
      ∀ b pt pk lst,
        Alternating b lst →
        Alternating b (PacketOut pt pk :: lst)
  | Alternating_FlowMod :
      ∀ f lst,
        Alternating true lst →
        Alternating false (FlowMod f :: lst)
  | Alternating_BarrierRequest :
      ∀ b n lst,
        Alternating false lst →
        Alternating b (BarrierRequest n :: lst).

  Inductive Approximating : switchId → flowTable → list fromController → Prop :=
  | Approximating_Nil :
      ∀ sw tbl, Approximating sw tbl nil
  | Approximating_FlowMod :
      ∀ sw f tbl lst,
        FlowTableSafe sw (modify_flow_table f tbl) →
        Approximating sw (modify_flow_table f tbl) lst →
        Approximating sw tbl (lst ++ [FlowMod f])
  | Approximating_PacketOut :
      ∀ sw pt pk tbl lst,
        Approximating sw tbl lst →
        Approximating sw tbl (lst ++ [PacketOut pt pk])
  | Approximating_BarrierRequest :
      ∀ sw n tbl lst,
        Approximating sw tbl lst →
        Approximating sw tbl (lst ++ [BarrierRequest n]).
  Inductive Barriered : switchId → list fromController → flowTable → bag fromCon-
  troller_le → Prop :=
  | Barriered_NoFlowMods :
      ∀ swId lst ctrlm tbl,
        (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
        Alternating false lst →
        Approximating swId tbl lst →
        FlowTableSafe swId tbl →
        Barriered swId lst tbl ctrlm
  | Barriered_OneFlowMod :
      ∀ swId lst ctrlm f tbl,
        (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
        Alternating false (lst ++ [FlowMod f]) →
        Approximating swId tbl (lst ++ [FlowMod f]) →
        FlowTableSafe swId tbl →
        Barriered swId lst tbl (({|FlowMod f|}) <+> ctrlm).
  Hint Constructors Alternating Approximating.
  Lemma alternating_pop : ∀ b x xs,
                                          Alternating b (xs ++ [x]) → Alternating b xs.
  Proof with auto with datatypes.
    intros b x xs H.
    generalize dependent b.
    induction xs; intros...
    simpl in H.
    inversion H...
  Qed.

  Lemma alternating_fm_fm_false :
    ∀ b lst f f0,
      Alternating b ((lst ++ [FlowMod f]) ++ [FlowMod f0]) →
      False.
  Proof with eauto with datatypes.
    intros b lst f f0 H.
    generalize dependent b.
    induction lst; intros...
    + simpl in H.
      inversion H; subst...
      inversion H2.
    + simpl in H.
      inversion H; subst...
  Qed.

  Lemma approximating_pop_FlowMod :
    ∀ sw tbl lst f,
      Approximating sw tbl (lst ++ [FlowMod f]) →
      Approximating sw (modify_flow_table f tbl) lst.
  Proof with auto with datatypes.
    intros.
    inversion H; subst.
    + destruct lst; simpl in H3; inversion H3.
    + apply cons_tail in H0. destruct H0. inversion H2; subst...
    + apply cons_tail in H0. destruct H0. inversion H1...
    + apply cons_tail in H0. destruct H0. inversion H1...
  Qed.

  Lemma approximating_pop_BarrierRequest :
    ∀ sw tbl lst n,
      Approximating sw tbl (lst ++ [BarrierRequest n]) →
      Approximating sw tbl lst.
  Proof with auto with datatypes.
    intros.
    inversion H; subst.
    + destruct lst; simpl in H3; inversion H3.
    + apply cons_tail in H0. destruct H0. inversion H2.
    + apply cons_tail in H0. destruct H0. inversion H1.
    + apply cons_tail in H0. destruct H0. inversion H1; subst...
  Qed.

  Lemma approximating_pop_PacketOut :
    ∀ sw tbl lst pt pk,
      Approximating sw tbl (lst ++ [PacketOut pt pk]) →
      Approximating sw tbl lst.
  Proof with auto with datatypes.
    intros.
    inversion H; subst.
    + destruct lst; simpl in H3; inversion H3.
    + apply cons_tail in H0. destruct H0. inversion H2.
    + apply cons_tail in H0. destruct H0. inversion H1; subst...
    + apply cons_tail in H0. destruct H0. inversion H1.
  Qed.
  Lemma approximating_pop_FlowMod_safe :
    ∀ sw tbl lst f,
      Approximating sw tbl (lst ++ [FlowMod f]) →
      FlowTableSafe sw (modify_flow_table f tbl).
  Proof with auto with datatypes.
    intros sw tbl lst f H.
    inversion H; subst.
    + destruct lst; simpl in H3; inversion H3.
    + apply cons_tail in H0. destruct H0. inversion H2; subst...
    + apply cons_tail in H0. destruct H0. inversion H1...
    + apply cons_tail in H0. destruct H0. inversion H1...
  Qed.
  Lemma Barriered_entails_FlowModSafe :
    ∀ swId lst tbl ctrlm,
      Barriered swId lst tbl ctrlm →
      FlowModSafe swId tbl ctrlm.
  Proof with eauto with datatypes.
    intros.
    inversion H; subst...
    + eapply NoFlowModsInBuffer...
    + eapply OneFlowModsInBuffer...
```

```
    inversion H2; subst...
  - destruct lst; simpl in H7; inversion H7.
  - apply cons_tail in H4.
    destruct H4; subst.
    inversion H6; subst...
  - apply cons_tail in H4.
    destruct H4.
    inversion H5.
  - apply cons_tail in H4.
    destruct H4.
    inversion H5.
Qed.

Lemma barriered_pop_BarrierRequest :
  ∀ swId xid lst tbl ctrlm,
    Barriered swId (lst ++ [BarrierRequest xid]) tbl ctrlm →
    (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
    Barriered swId lst tbl ctrlm.
Proof with eauto with datatypes.
  intros.
  rename H0 into X.
  inversion H; subst.
  + apply Barriered_NoFlowMods...
    apply alternating_pop in H1...
    apply approximating_pop_BarrierRequest in H2...
  + assert (NotFlowMod (FlowMod f)).
```

```
    apply X.
    apply Bag.in_union; simpl...
    inversion H4.
Qed.

Lemma alternating_splice_PacketOut :
  ∀ b lst1 pt pk lst2,
    Alternating b (lst1 ++ PacketOut pt pk :: lst2) ↔
    Alternating b (lst1 ++ lst2).
Proof with auto with datatypes.
  intros b lst1 pt pk lst2.
  split.
  + intros H.
    generalize dependent b.
    induction lst1; intros; simpl in *; inversion H...
  + intros H.
    generalize dependent b.
    induction lst1; intros...
  - simpl in *...
              - simpl in *.
    inversion H; subst...
Qed.

Hint Resolve alternating_pop approximating_pop_PacketOut approximating_pop_FlowMod
     approximating_pop_BarrierRequest.

Lemma approximating_splice_PacketOut :
  ∀ sw tbl lst1 pt pk lst2,
    Approximating sw tbl (lst1 ++ PacketOut pt pk :: lst2) ↔
    Approximating sw tbl (lst1 ++ lst2).
Proof with eauto with datatypes.
  intros sw tbl lst1 pt pk lst2.
  split.
  + intros H.
    generalize dependent tbl.
    induction lst2 using rev_ind; intros.
  - rewrite → app_nil_r...
  - rewrite → app_comm_cons in H.
    rewrite → app_assoc in H.
    rewrite → app_assoc.
    destruct x...
    assert (FlowTableSafe sw (modify_flow_table f tbl0)).
    { eapply approximating_pop_FlowMod_safe... }
    eauto.
    + intros H.
      generalize dependent tbl.
      induction lst2 using rev_ind; intros.
  - rewrite → app_nil_r in H...
  - rewrite → app_comm_cons.
    rewrite → app_assoc.
    rewrite → app_assoc in H.
    destruct x...
    assert (FlowTableSafe sw (modify_flow_table f tbl0)) as X...
    { eapply approximating_pop_FlowMod_safe... }
```

```
      Grab Existential Variables.
    exact 0.
Qed.

Hint Resolve alternating_splice_PacketOut approximating_splice_PacketOut.

Lemma barriered_pop_PacketOut :
  ∀ sw pt pk lst tbl ctrlm,
    Barriered sw (lst ++ [PacketOut pt pk]) tbl ctrlm →
    Barriered sw lst tbl (({|Atoms.PacketOut pt pk|}) <+> ctrlm).
Proof with eauto with datatypes.
  intros sw pt pk lst tbl ctrlm H.
  inversion H; subst.
  + apply Barriered_NoFlowMods...
  - intros.
    apply Bag.in_union in H4; simpl in H4.
    destruct H4 as [[H4 | H4] | H4]; subst...
    inversion H4.
    + rewrite ← Bag.union_assoc.
      rewrite ← (Bag.union_comm _ ({|FlowMod f|})).
      rewrite → Bag.union_assoc.
      apply Barriered_OneFlowMod...
  - intros.
    apply Bag.in_union in H4; simpl in H4.
    destruct H4 as [[H4 | H4] | H4]; subst...
    inversion H4.
  - rewrite ← app_assoc in H1.
```

```
    simpl in H1.
    apply alternating_splice_PacketOut in H1...
  - rewrite ← app_assoc in H2.
    simpl in H2.
    apply approximating_splice_PacketOut in H2...
    Grab Existential Variables. exact 0.
Qed.

Lemma barriered_splice_PacketOut :
  ∀ sw lst1 pt pk lst2 tbl ctrlm,
    Barriered sw (lst1 ++ lst2) tbl ctrlm →
    Barriered sw (lst1 ++ PacketOut pt pk :: lst2) tbl ctrlm.
Proof with eauto with datatypes.
  intros sw lst1 pt pk lst2 tbl ctrlm H.
  inversion H; subst.
  + eapply Barriered_NoFlowMods...
  - eapply alternating_splice_PacketOut...
  - eapply approximating_splice_PacketOut...
    + eapply Barriered_OneFlowMod...
  - rewrite ← app_assoc.
    rewrite ← app_comm_cons.
    apply alternating_splice_PacketOut...
    rewrite → app_assoc...
  - rewrite ← app_assoc.
    rewrite ← app_comm_cons.
    apply approximating_splice_PacketOut...
    rewrite → app_assoc...
Qed.

Lemma barriered_process_PacketOut :
  ∀ sw lst tbl pt pk ctrlm,
    Barriered sw lst tbl (({|PacketOut pt pk|}) <+> ctrlm) →
    Barriered sw lst tbl ctrlm.
Proof with eauto with datatypes.
  intros.
  inversion H; subst.
  + eapply Barriered_NoFlowMods...
    intros. apply H0. apply Bag.in_union...
  + apply Bag.union_from_ordered in H0.
    assert (In (FlowMod f) (to_list ctrlm0)) as J.
    { assert (In (FlowMod f) (to_list (({|PacketOut pt pk|}) <+> ctrlm0))) as J.
      rewrite ← H0. apply Bag.in_union; simpl...
      apply Bag.in_union in J. simpl in J.
      destruct J as [[J|J]|J]...
      + inversion J.
      + inversion J. }
    eapply Bag.in_split in J.
    destruct J as [ctrlm2 HEq].
    rewrite → HEq.
    eapply Barriered_OneFlowMod...
    intros.
    subst.
```

```
      rewrite ← Bag.union_assoc in H0.
      rewrite → (Bag.union_comm _ ({|PacketOut pt pk|})) in H0.
      rewrite → Bag.union_assoc in H0.
      apply Bag.pop_union_l in H0.
      subst.
      eapply H1.
      apply Bag.in_union...
  Qed.
  Lemma barriered_pop_FlowMod : ∀ sw f tbl lst ctrlm,
                                          (∀ x, In x (to_list ctrlm) → NotFlowMod x) →
                                          Barriered sw (lst ++ [FlowMod f]) tbl ctrlm →
                                          Barriered sw lst tbl (({|FlowMod f|}) <+> ctrlm).
  Proof with eauto with datatypes.
    intros sw f tbl lst ctrlm H H0.
    inversion H0; subst.
    + apply Barriered_OneFlowMod...
    + assert (NotFlowMod (FlowMod f0)) as X.
      apply H. apply Bag.in_union; simpl...
      inversion X.
  Qed.
End Make.
```

### A.2.24 FwOFSignatures Library

```
Set Implicit Arguments.
```

```
Require Import Coq.Lists.List.

Require Import Coq.Relations.Relations.

Require Import Common.Types.

Require Import Bag.TotalOrder.

Require Import Bag.Bag2.

Require Import Common.AllDiff.

Require Import Common.Bisimulation.

Local Open Scope list_scope.

Local Open Scope equiv_scope.

Local Open Scope bag_scope.

Module Type NETWORK_ATOMS.

  Parameter packet : Type.

  Parameter switchId : Type.

  Parameter portId : Type.

  Parameter flowTable : Type.

  Parameter flowMod : Type.

  Inductive fromController : Type :=

  | PacketOut : portId → packet → fromController

  | BarrierRequest : nat → fromController

  | FlowMod : flowMod → fromController.

  Inductive fromSwitch : Type :=

  | PacketIn : portId → packet → fromSwitch

  | BarrierReply : nat → fromSwitch.
```

Produces a list of packets to forward out of ports, and a list of packets to send to the

controller.
```
Parameter process_packet : flowTable → portId → packet →
  list (portId × packet) × list packet.

  Parameter modify_flow_table : flowMod → flowTable → flowTable.

  Parameter packet_le : Relation_Definitions.relation packet.
  Parameter switchId_le : Relation_Definitions.relation switchId.
  Parameter portId_le : Relation_Definitions.relation portId.
  Parameter flowTable_le : Relation_Definitions.relation flowTable.
  Parameter flowMod_le : Relation_Definitions.relation flowMod.
  Parameter fromSwitch_le : Relation_Definitions.relation fromSwitch.
  Parameter fromController_le : Relation_Definitions.relation fromController.

  Declare Instance TotalOrder_packet : TotalOrder packet_le.
  Declare Instance TotalOrder_switchId : TotalOrder switchId_le.
  Declare Instance TotalOrder_portId : TotalOrder portId_le.
  Declare Instance TotalOrder_flowTable : TotalOrder flowTable_le.
  Declare Instance TotalOrder_flowMod : TotalOrder flowMod_le.
  Declare Instance TotalOrder_fromSwitch : TotalOrder fromSwitch_le.
  Declare Instance TotalOrder_fromController : TotalOrder fromController_le.

End NETWORK_ATOMS.

Module Type NETWORK_AND_POLICY <: NETWORK_ATOMS.

  Include Type NETWORK_ATOMS.

  Parameter topo : switchId × portId → option (switchId × portId).
  Parameter abst_func : switchId → portId → packet → list (portId × packet).

End NETWORK_AND_POLICY.
```

Elements of a Featherweight OpenFlow model. `Module Type ATOMS <: NET-`

```
WORK_AND_POLICY.
  Include NETWORK_AND_POLICY.
  Parameter controller : Type.
  Parameter controller_recv : controller → switchId → fromSwitch →
    controller → Prop.
  Parameter controller_step : controller → controller → Prop.
  Parameter controller_send : controller → controller → switchId →
    fromController → Prop.
End ATOMS.
Module Type MACHINE.
  Declare Module Atoms : ATOMS.
  Import Atoms.
  Existing Instances TotalOrder_packet TotalOrder_switchId TotalOrder_portId
    TotalOrder_flowTable TotalOrder_flowMod TotalOrder_fromSwitch
    TotalOrder_fromController.
  Record switch := Switch {
    swId : switchId;
    pts : list portId;
    tbl : flowTable;
    inp : bag (PairOrdering portId_le packet_le);
    outp : bag (PairOrdering portId_le packet_le);
    ctrlm : bag fromController_le;
    switchm : bag fromSwitch_le
  }.
```

```
Inductive switch_le : switch → switch → Prop :=
| SwitchLe : ∀ sw1 sw2,
    switchId_le (swId sw1) (swId sw2) →
    switch_le sw1 sw2.

Declare Instance TotalOrder_switch : TotalOrder switch_le.

Record dataLink := DataLink {
  src : switchId × portId;
  pks : list packet;
  dst : switchId × portId
}.

Record openFlowLink := OpenFlowLink {
  of_to : switchId;
  of_switchm : list fromSwitch;
  of_ctrlm : list fromController
}.

Definition observation := (switchId × portId × packet) %type.

Reserved Notation "SwitchStep[ sw ; obs ; sw0 ]"
  (at level 70, no associativity).

Reserved Notation "ControllerOpenFlow[ c ; l ; obs ; c0 ; l0 ]"
  (at level 70, no associativity).

Reserved Notation "TopoStep[ sw ; link ; obs ; sw0 ; link0 ]"
  (at level 70, no associativity).

Reserved Notation "SwitchOpenFlow[ s ; l ; obs ; s0 ; l0 ]"
  (at level 70, no associativity).

Inductive NotBarrierRequest : fromController → Prop :=
| PacketOut_NotBarrierRequest : ∀ pt pk,
    NotBarrierRequest (PacketOut pt pk)
| FlowMod_NotBarrierRequest : ∀ fm,
    NotBarrierRequest (FlowMod fm).
```

Devices of the same type do not interact in a single step. Therefore, we never have to permute the lists below. If we instead had just one list of all devices, we would have to worry about permuting the list or define symmetric step-rules.

```
Record state := State {
  switches : bag switch_le;
  links : list dataLink;
  ofLinks : list openFlowLink;
  ctrl : controller
}.

Inductive step : state → option observation → state → Prop :=
| PktProcess : ∀ swId pts tbl pt pk inp outp ctrlm switchm outp'
                            pksToCtrl,
  process_packet tbl pt pk = (outp', pksToCtrl) →
  SwitchStep[
    Switch swId pts tbl ({|(pt,pk)|} <+> inp) outp ctrlm switchm;
    Some (swId,pt,pk);
    Switch swId pts tbl inp (from_list outp' <+> outp)
      ctrlm (from_list (map (PacketIn pt) pksToCtrl) <+> switchm)
  ]
| ModifyFlowTable : ∀ swId pts tbl inp outp fm ctrlm switchm,
  SwitchStep[
    Switch swId pts tbl inp outp ({|FlowMod fm|} <+> ctrlm) switchm;
```

*None*;

*Switch swId pts* (*modify_flow_table fm tbl*) *inp outp ctrlm switchm*

]

We add the packet to the output-buffer, even if its port is invalid. Packets with invalid ports will simply accumulate in the output buffer, since the SendDataLink rule only pulls out packets with valid ports. This is reasonable for now. The right fix is to add support for OpenFlow errors. | *SendPacketOut* : ∀ *pt pts swId tbl inp outp pk ctrlm switchm*,

*SwitchStep*[

*Switch swId pts tbl inp outp* ({|*PacketOut pt pk*|} <+> *ctrlm*) *switchm*;

*None*;

*Switch swId pts tbl inp* ({| (*pt*,*pk*) |} <+> *outp*) *ctrlm switchm*

]

| *SendDataLink* : ∀ *swId pts tbl inp pt pk outp ctrlm switchm pks dst*,

*TopoStep*[

*Switch swId pts tbl inp* ({|(*pt*,*pk*)|} <+> *outp*) *ctrlm switchm*;

*DataLink* (*swId*,*pt*) *pks dst*;

*None*;

*Switch swId pts tbl inp outp ctrlm switchm*;

*DataLink* (*swId*,*pt*) (*pk* :: *pks*) *dst*

]

| *RecvDataLink* : ∀ *swId pts tbl inp outp ctrlm switchm src pks pk pt*,

*TopoStep*[

*Switch swId pts tbl inp outp ctrlm switchm*;

*DataLink src* (*pks* ++ [*pk*]) (*swId*,*pt*);

*None*;

*Switch swId pts tbl* ({|(*pt*,*pk*)|} <+> *inp*) *outp ctrlm switchm*;

```
    DataLink src pks (swId,pt)
  ]
| Step_controller : ∀ sws links ofLinks ctrl ctrl',
  controller_step ctrl ctrl' →
  step (State sws links ofLinks ctrl)
       None
       (State sws links ofLinks ctrl')
| ControllerRecv : ∀ ctrl msg ctrl' swId fromSwitch fromCtrl,
  controller_recv ctrl swId msg ctrl' →
  ControllerOpenFlow[
    ctrl;
    OpenFlowLink swId (fromSwitch ++ [msg]) fromCtrl;
    None;
    ctrl';
    OpenFlowLink swId fromSwitch fromCtrl
  ]
| ControllerSend : ∀ ctrl msg ctrl' swId fromSwitch fromCtrl,
  controller_send ctrl ctrl' swId msg →
  ControllerOpenFlow[
    ctrl ;
    (OpenFlowLink swId fromSwitch fromCtrl);
    None;
    ctrl';
    (OpenFlowLink swId fromSwitch (msg :: fromCtrl)) ]
| SendToController : ∀ swId pts tbl inp outp ctrlm msg switchm fromSwitch
    fromCtrl,
  SwitchOpenFlow[
    Switch swId pts tbl inp outp ctrlm ({| msg |} <+> switchm);
    OpenFlowLink swId fromSwitch fromCtrl;
    None;
    Switch swId pts tbl inp outp ctrlm switchm;
    OpenFlowLink swId (msg :: fromSwitch) fromCtrl
  ]
| RecvBarrier : ∀ swId pts tbl inp outp switchm fromSwitch fromCtrl
    xid,
  SwitchOpenFlow[
    Switch swId pts tbl inp outp empty switchm;
    OpenFlowLink swId fromSwitch (fromCtrl ++ [BarrierRequest xid]);
    None;
    Switch swId pts tbl inp outp empty
           ({| BarrierReply xid |} <+> switchm);
    OpenFlowLink swId fromSwitch fromCtrl
  ]
| RecvFromController : ∀ swId pts tbl inp outp ctrlm switchm
    fromSwitch fromCtrl msg,
  NotBarrierRequest msg →
  SwitchOpenFlow[
    Switch swId pts tbl inp outp ctrlm switchm;
    OpenFlowLink swId fromSwitch (fromCtrl ++ [msg]);
    None;
    Switch swId pts tbl inp outp ({| msg |} <+> ctrlm) switchm;
    OpenFlowLink swId fromSwitch fromCtrl
  ]
    where
"ControllerOpenFlow[ c ; l ; obs ; c0 ; l0 ]" :=
  (∀ sws links ofLinks ofLinks',
    step (State sws links (ofLinks ++ l :: ofLinks') c)
         obs
         (State sws links (ofLinks ++ l0 :: ofLinks') c0))
  and
"TopoStep[ sw ; link ; obs ; sw0 ; link0 ]" :=
  (∀ sws links links0 ofLinks ctrl,
    step
    (State (({|sw|}) <+> sws) (links ++ link :: links0) ofLinks ctrl)
    obs
    (State (({|sw0|}) <+> sws) (links ++ link0 :: links0) ofLinks ctrl))
  and
"SwitchStep[ sw ; obs ; sw0 ]" :=
  (∀ sws links ofLinks ctrl,
    step
      (State (({|sw|}) <+> sws) links ofLinks ctrl)
      obs
      (State (({|sw0|}) <+> sws) links ofLinks ctrl))
  and
"SwitchOpenFlow[ sw ; of ; obs ; sw0 ; of0 ]" :=
  (∀ sws links ofLinks ofLinks0 ctrl,
    step
      (State (({|sw|}) <+> sws) links (ofLinks ++ of :: ofLinks0) ctrl)
    obs
    (State (({|sw0|}) <+> sws) links (ofLinks ++ of0 :: ofLinks0) ctrl)).

Definition swPtPks : Type :=
  bag (PairOrdering (PairOrdering switchId_le portId_le)
                    packet_le).

Definition abst_state := swPtPks.

Definition transfer (sw : switchId) (ptpk : portId × packet) :=
  match ptpk with
    | (pt,pk) ⇒
      match topo (sw,pt) with
        | Some (sw',pt') ⇒
          @singleton _
             (PairOrdering
                (PairOrdering switchId_le portId_le) packet_le)
             (sw',pt',pk)
        | None ⇒ {| |}
      end
  end.

Definition select_packet_out (sw : switchId) (msg : fromController) :=
  match msg with
    | PacketOut pt pk ⇒ transfer sw (pt,pk)
    | _ ⇒ {| |}
  end.

Definition select_packet_in (sw : switchId) (msg : fromSwitch) :=
  match msg with
```

```
    | PacketIn pt pk ⇒ unions (map (transfer sw) (abst_func sw pt pk))
    | _ ⇒ {| |}
  end.

Definition FlowTableSafe (sw : switchId) (tbl : flowTable) : Prop :=
  ∀ pt pk forwardedPkts packetIns,
    process_packet tbl pt pk = (forwardedPkts, packetIns) →
    unions (map (transfer sw) forwardedPkts) <+>
    unions (map (select_packet_in sw) (map (PacketIn pt) packetIns)) =
    unions (map (transfer sw) (abst_func sw pt pk)).

Inductive NotFlowMod : fromController → Prop :=
| NotFlowMod_BarrierRequest : ∀ n, NotFlowMod (BarrierRequest n)
| NotFlowMod_PacketOut : ∀ pt pk, NotFlowMod (PacketOut pt pk).

Inductive FlowModSafe : switchId → flowTable → bag fromController_le → Prop :=
| NoFlowModsInBuffer : ∀ swId tbl ctrlm,
    (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
    FlowTableSafe swId tbl →
    FlowModSafe swId tbl ctrlm
| OneFlowModsInBuffer : ∀ swId tbl ctrlm f,
    (∀ msg, In msg (to_list ctrlm) → NotFlowMod msg) →
    FlowTableSafe swId tbl →
    FlowTableSafe swId (modify_flow_table f tbl) →
    FlowModSafe swId tbl (({|FlowMod f|}) <+> ctrlm).

Definition FlowTablesSafe (sws : bag switch_le) : Prop :=
  ∀ swId pts tbl inp outp ctrlm switchm,
    In (Switch swId pts tbl inp outp ctrlm switchm) (to_list sws) →
```

```
      FlowModSafe swId tbl ctrlm.

  Definition SwitchesHaveOpenFlowLinks (sws : bag switch_le) ofLinks :=
    ∀ sw,
      In sw (to_list sws) →
      ∃ ofLink,
        In ofLink ofLinks ∧
        swId sw = of_to ofLink.

End MACHINE.

Module Type ATOMS_AND_CONTROLLER.

  Declare Module Machine : MACHINE.

  Import Machine.

  Import Atoms.

  Parameter relate_controller : controller → swPtPks.

  Parameter ControllerRemembersPackets :
    ∀ (ctrl ctrl' : controller),
      controller_step ctrl ctrl' →
      relate_controller ctrl = relate_controller ctrl'.

  Parameter P : bag switch_le → list openFlowLink → controller → Prop.

  Parameter P_entails_FlowTablesSafe : ∀ sws ofLinks ctrl,
    P sws ofLinks ctrl →
    SwitchesHaveOpenFlowLinks sws ofLinks →
    FlowTablesSafe sws.

  Parameter step_preserves_P : ∀ sws0 sws1 links0 links1 ofLinks0 ofLinks1
    ctrl0 ctrl1 obs,
  AllDiff of_to ofLinks0 →
  AllDiff swId (to_list sws0) →
  step (State sws0 links0 ofLinks0 ctrl0)
       obs
       (State sws1 links1 ofLinks1 ctrl1) →
  P sws0 ofLinks0 ctrl0 →
  P sws1 ofLinks1 ctrl1.

Parameter ControllerSendForgetsPackets : ∀ ctrl ctrl' sw msg,
  controller_send ctrl ctrl' sw msg →
  relate_controller ctrl = select_packet_out sw msg <+>
  relate_controller ctrl'.

Parameter ControllerRecvRemembersPackets : ∀ ctrl ctrl' sw msg,
  controller_recv ctrl sw msg ctrl' →
  relate_controller ctrl' = select_packet_in sw msg <+>
  (relate_controller ctrl).
```

If (*sw*,*pt*,*pk*) is a packet in the controller's abstract state, then the controller will eventually emit the packet.

```
Parameter ControllerLiveness : ∀ sw pt pk ctrl0 sws0 links0

                                    ofLinks0,
  In (sw,pt,pk) (to_list (relate_controller ctrl0)) →
  ∃ ofLinks10 ofLinks11 ctrl1 swTo ptTo switchmLst ctrlmLst,
    (multistep
       step (State sws0 links0 ofLinks0 ctrl0) nil
       (State sws0 links0
              (ofLinks10 ++
                    (OpenFlowLink swTo switchmLst
                     (PacketOut ptTo pk :: ctrlmLst)) ::
                    ofLinks11)
                  ctrl1)) ∧
        select_packet_out swTo (PacketOut ptTo pk) = ({|(sw,pt,pk)|}).
```

If $m$ is a message from the switch to the controller, then the controller will eventually consume $m$, adding its packet-content to its state. `Parameter` *ControllerRecvLiveness* :

```
∀ sws0 links0 ofLinks0 sw switchm0 m
    ctrlm0 ofLinks1 ctrl0,
     ∃ ctrl1,
      (multistep
         step
         (State
            sws0 links0
            (ofLinks0 ++ (OpenFlowLink sw (switchm0 ++ [m]) ctrlm0) :: ofLinks1)
            ctrl0)
         nil
         (State
            sws0 links0
            (ofLinks0 ++ (OpenFlowLink sw switchm0 ctrlm0) :: ofLinks1)
            ctrl1)) ∧
       ∃ (lps : swPtPks),
         (select_packet_in sw m) <+> lps = relate_controller ctrl1.
End ATOMS_AND_CONTROLLER.
Module Type RELATION_DEFINITIONS.
```

```
Declare Module AtomsAndController : ATOMS_AND_CONTROLLER.

Import AtomsAndController.

Import Machine.

Import Atoms.

Definition affixSwitch (sw : switchId) (ptpk : portId × packet) :=
  match ptpk with
    | (pt,pk) ⇒ (sw,pt,pk)
  end.

Definition ConsistentDataLinks (links : list dataLink) : Prop :=
  ∀ (lnk : dataLink),
    In lnk links →
    topo (src lnk) = Some (dst lnk).

Definition LinkHasSrc (sws : bag switch_le) (link : dataLink) : Prop :=
  ∃ switch,
    In switch (to_list sws) ∧
    fst (src link) = swId switch ∧
    In (snd (src link)) (pts switch).

Definition LinkHasDst (sws : bag switch_le) (link : dataLink) : Prop :=
  ∃ switch,
    In switch (to_list sws) ∧
    fst (dst link) = swId switch ∧
    In (snd (dst link)) (pts switch).

Definition LinksHaveSrc (sws : bag switch_le) (links : list dataLink) :=
  ∀ link, In link links → LinkHasSrc sws link.

Definition LinksHaveDst (sws : bag switch_le) (links : list dataLink) :=
  ∀ link, In link links → LinkHasDst sws link.

Definition UniqSwIds (sws : bag switch_le) := AllDiff swId (to_list sws).

Definition ofLinkHasSw (sws : bag switch_le) (ofLink : openFlowLink) :=
  ∃ sw,
    In sw (to_list sws) ∧
    of_to ofLink = swId sw.

Definition OFLinksHaveSw (sws : bag switch_le) (ofLinks : list openFlowLink) :=
  ∀ ofLink, In ofLink ofLinks → ofLinkHasSw sws ofLink.

Definition DevicesFromTopo (devs : state) :=
  ∀ swId0 swId1 pt0 pt1,
    Some (swId0,pt0) = topo (swId1,pt1) →
    ∃ sw0 sw1 lnk,


      In sw0 (to_list (switches devs)) ∧
      In sw1 (to_list (switches devs)) ∧
      In lnk (links devs) ∧
      swId sw0 = swId0 ∧
      swId sw1 = swId1 ∧
      src lnk = (swId1,pt1) ∧
      dst lnk = (swId0, pt0).

Definition NoBarriersInCtrlm (sws : bag switch_le) :=
  ∀ sw,
    In sw (to_list sws) →
    ∀ m,
      In m (to_list (ctrlm sw)) →
```

```
        NotBarrierRequest m.

Record concreteState := ConcreteState {
  devices : state;
  concreteState_flowTableSafety : FlowTablesSafe (switches devices);
  concreteState_consistentDataLinks : ConsistentDataLinks (links devices);
  linksHaveSrc : LinksHaveSrc (switches devices) (links devices);
  linksHaveDst : LinksHaveDst (switches devices) (links devices);
  uniqSwIds : UniqSwIds (switches devices);
  ctrlP : P (switches devices) (ofLinks devices) (ctrl devices);
  uniqOfLinkIds : AllDiff of_to (ofLinks devices);
  ofLinksHaveSw : OFLinksHaveSw (switches devices) (ofLinks devices);
  devicesFromTopo : DevicesFromTopo devices;
  swsHaveOFLinks : SwitchesHaveOpenFlowLinks (switches devices) (ofLinks devices);
  noBarriersInCtrlm : NoBarriersInCtrlm (switches devices)
}.

Implicit Arguments ConcreteState [].

Definition concreteStep (st : concreteState) (obs : option observation)
  (st0 : concreteState) :=
  step (devices st) obs (devices st0).

Inductive abstractStep : abst_state → option observation → abst_state →
  Prop :=
| AbstractStep : ∀ sw pt pk lps,
  abstractStep
    ({| (sw,pt,pk) |} <+> lps)
    (Some (sw,pt,pk))
    (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps).

Definition relate_switch (sw : switch) : abst_state :=
  match sw with
    | Switch swId _ tbl inp outp ctrlm switchm ⇒
      from_list (map (affixSwitch swId) (to_list inp)) <+>
      unions (map (transfer swId) (to_list outp)) <+>
      unions (map (select_packet_out swId) (to_list ctrlm)) <+>
      unions (map (select_packet_in swId) (to_list switchm))
  end.

Definition relate_dataLink (link : dataLink) : abst_state :=
  match link with
    | DataLink _ pks (sw,pt) ⇒
      from_list (map (fun pk ⇒ (sw,pt,pk)) pks)
  end.

Definition relate_openFlowLink (link : openFlowLink) : abst_state :=
  match link with
    | OpenFlowLink sw switchm ctrlm ⇒
      unions (map (select_packet_out sw) ctrlm) <+>
      unions (map (select_packet_in sw) switchm)
  end.

Definition relate (st : state) : abst_state :=
  unions (map relate_switch (to_list (switches st))) <+>
  unions (map relate_dataLink (links st)) <+>
  unions (map relate_openFlowLink (ofLinks st)) <+>
  relate_controller (ctrl st).
  Definition bisim_relation : relation concreteState abst_state :=
    fun (st : concreteState) (ast : abst_state) ⇒
      ast = (relate (devices st)).
End RELATION_DEFINITIONS.
Module Type RELATION.
  Declare Module RelationDefinitions : RELATION_DEFINITIONS.
  Import RelationDefinitions.
  Import AtomsAndController.
  Import Machine.
  Import Atoms.
  Parameter simpl_multistep : ∀ (st1 : concreteState) (devs2 : state) obs,
    multistep step (devices st1) obs devs2 →
    ∃ (st2 : concreteState),
      devices st2 = devs2 ∧
      multistep concreteStep st1 obs st2.
  Parameter simpl_weak_sim : ∀ st1 devs2 sw pt pk lps,
    multistep step (devices st1) [(sw,pt,pk)] devs2 →
    relate (devices st1) = ({| (sw,pt,pk) |} <+> lps) →
    abstractStep
      ({| (sw,pt,pk) |} <+> lps)
      (Some (sw,pt,pk))
      (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps) →
   ∃ st2 : concreteState,
     inverse_relation
       bisim_relation
```

```
        (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps)
        st2 ∧
      multistep concreteStep st1 [(sw,pt,pk)] st2.
End RELATION.

Module Type WEAK_SIM_1.
  Declare Module Relation : RELATION.
  Import Relation.
  Import RelationDefinitions.
  Import AtomsAndController.
  Import Machine.
  Import Atoms.
  Parameter weak_sim_1 : weak_simulation concreteStep abstractStep bisim_relation.
End WEAK_SIM_1.

Module Type WEAK_SIM_2.
  Declare Module Relation : RELATION.
  Import Relation.
  Import RelationDefinitions.
  Import AtomsAndController.
  Import Machine.
  Import Atoms.
  Parameter weak_sim_2 :
    weak_simulation abstractStep concreteStep (inverse_relation bisim_relation).
End WEAK_SIM_2.
```

### A.2.25 FwOFSimpleController Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Common.Types.

Require Import Common.Bisimulation.

Require Import FwOF.FwOFSignatures.

Require Import Common.Bisimulation.

Require Import Common.AllDiff.

Require FwOF.FwOFExtractableController.

Local Open Scope list_scope.

Module Make (NetAndPol : NETWORK_AND_POLICY) <: ATOMS.
  Include NetAndPol.

  Module Import ExtractableController := FwOF.FwOFExtractableController.MakeController
(NetAndPol).

  Definition controller := controller.

  Inductive Recv : controller → switchId → fromSwitch → controller → Prop :=
  | RecvBarrierReply : ∀ st swId n,
    Recv st swId (BarrierReply n) st
  | RecvPacketIn : ∀ swsts pksToSend sw pt pk,
    Recv (State pksToSend swsts)
         sw (PacketIn pt pk)
         (State (mkPktOuts sw pt pk ++ pksToSend) swsts).

  Inductive Send : state → state → switchId → fromController → Prop :=
  | SendPacketOut : ∀ swsts srcPt srcPk dstPt dstPk sw lps,
    Send (State ((SrcDst sw srcPt srcPk dstPt dstPk)::lps) swsts)
         (State lps swsts)
         sw
         (PacketOut dstPt dstPk)
  | SendMessage : ∀ sw stsws stsws' msg msgs,
      Send
        (State nil
               (stsws ++ (SwitchState sw (msg::msgs)) :: stsws'))
        (State nil
               (stsws ++ (SwitchState sw msgs) :: stsws'))
        sw
        msg.

  Inductive Step : state → state → Prop := .

  Definition controller_recv := Recv.

  Definition controller_step := Step.

  Definition controller_send := Send.

  Hint Constructors Send Recv.

  Lemma Send_cons : ∀ s ss1 ss2 sw msg,
    Send (State nil ss1) (State nil ss2) sw msg →
    Send (State nil (s::ss1)) (State nil (s::ss2)) sw msg.
  Proof with auto with datatypes.
    intros.
    inversion H; subst.
    do 2 rewrite → app_comm_cons...
  Qed.
```

```
Lemma send_queued_compat : ∀ ss1 ss2 sw msg,
  send_queued ss1 = Some (ss2, sw, msg) →
  Send (State nil ss1) (State nil ss2) sw msg.
Proof with auto with datatypes.
  intros.
  generalize dependent ss2.
  generalize dependent sw.
  generalize dependent msg.
  induction ss1; intros...
  simpl in H. inversion H.
  simpl in H.
  destruct a.
  + destruct pendingCtrlMsgs0.
    - remember (send_queued ss1) as rest.
      destruct rest.
      × destruct p.
        destruct p.
        remember (IHss1 f s l eq_refl) as J eqn:X; clear X.
        inversion H; subst.
        apply Send_cons...
      × inversion H.
    - inversion H; subst.
      assert (SwitchState sw (msg::pendingCtrlMsgs0) :: ss1 =
                nil ++ SwitchState sw (msg::pendingCtrlMsgs0) :: ss1) as X...
      rewrite → X; clear X.
      assert (SwitchState sw pendingCtrlMsgs0 :: ss1 =
              nil ++ SwitchState sw pendingCtrlMsgs0 :: ss1) as X...
      rewrite → X; clear X...
Qed.

Lemma send_compat : ∀ st1 st2 sw msg,
  send st1 = Some (st2, sw, msg) →
  Send st1 st2 sw msg.
Proof with auto with datatypes.
  intros.
  destruct st1.
  simpl in H.
  destruct pktsToSend0.
  + remember (send_queued switchStates0) as J.
    destruct J.
    - destruct p as [[ss2 sw2] msg2].
      symmetry in HeqJ; apply send_queued_compat in HeqJ.
      inversion H; subst...
    - inversion H.
  + destruct s.
    inversion H; subst...
Qed.

Lemma recv_compat : ∀ st1 sw msg st2,
  recv st1 sw msg = st2 →
  Recv st1 sw msg st2.
Proof with auto with datatypes.
  intros.
```

```
    destruct msg...
    + destruct st1.
      simpl in H.
      subst...
    + destruct st1.
      simpl in H.
      subst...
  Qed.
End Make.
```

### A.2.26 FwOFSimpleControllerLemmas Library

```
Set Implicit Arguments.
Require Import Coq.Lists.List.
Require Import Common.Types.
Require Import Common.Bisimulation.
Require Import Bag.TotalOrder.
Require Import Bag.Bag2.
Require Import FwOF.FwOFSignatures.
Require Import Common.Bisimulation.
Require Import Common.AllDiff.
Require FwOF.FwOFMachine.
Require FwOF.FwOFSimpleController.
Require FwOF.FwOFSafeWire.
Local Open Scope list_scope.
Local Open Scope bag_scope.

Module MakeController (NetAndPol : NETWORK_AND_POLICY) <: ATOMS_AND_CONTROLLER.
  Module Import Atoms := FwOF.FwOFSimpleController.Make (NetAndPol).
  Module Import Machine := FwOF.FwOFMachine.Make (Atoms).

  Module Import SafeWire := FwOF.FwOFSafeWire.Make (Machine).
  Import ExtractableController.

  Hint Resolve alternating_pop Barriered_entails_FlowModSafe approximating_pop_FlowMod.

  Inductive Invariant : bag switch_le → list openFlowLink → controller → Prop :=
  | MkP : ∀ sws ofLinks swsts pktOuts,
      AllDiff theSwId swsts →
      (∀ sw swId0 switchmLst ctrlmLst,
         In sw (to_list sws) →
         In (OpenFlowLink swId0 switchmLst ctrlmLst) ofLinks →
         swId sw = swId0 →
         ∃ pendingMsgs,
           In (SwitchState swId0 pendingMsgs) swsts ∧
           (∀ msg, In msg pendingMsgs → NotPacketOut msg) ∧
           Barriered swId0 (rev pendingMsgs ++ ctrlmLst) (tbl sw) (ctrlm sw)) →
      Invariant sws ofLinks (State pktOuts swsts).

  Hint Constructors Invariant.

  Lemma P_entails_FlowTablesSafe : ∀ sws ofLinks ctrl,
    Invariant sws ofLinks ctrl →
    SwitchesHaveOpenFlowLinks sws ofLinks →
    FlowTablesSafe sws.
  Proof with eauto with datatypes.
    intros.
    unfold FlowTablesSafe.
    intros.
    inversion H; subst.
    edestruct H0 as [lnk [HIn HIdEq]]...
    simpl...
    destruct lnk.
    simpl in HIdEq. subst...
    rename of_to0 into swId0.
    inversion H; subst...
    edestruct H3 as [pendingMsgs [J [J0 J1]]]...
  Qed.

  Lemma controller_recv_pres_P : ∀ sws ofLinks0 ofLinks1
    ctrl0 ctrl1 swId msg switchm ctrlm,
    Recv ctrl0 swId msg ctrl1 →
    Invariant sws
              (ofLinks0 ++ (OpenFlowLink swId (switchm ++ [msg]) ctrlm) :: ofLinks1)
              ctrl0 →
    Invariant sws
              (ofLinks0 ++ (OpenFlowLink swId switchm ctrlm) :: ofLinks1)
              ctrl1.
  Proof with eauto with datatypes.
    intros.
    inversion H; subst.
    + inversion H0; subst.
```

```
    eapply MkP...
    intros.
    apply in_app_iff in H4. simpl in H4.
    destruct H4 as [H4 | [H4 | H4]]...
    inversion H4; subst; clear H4.
    edestruct H2 as [msgs [HInSw [HNotPktOuts HBarriered]]]...
  + inversion H0; subst.
    eapply MkP...
    intros.
    eapply in_app_iff in H2; simpl in H2.
    destruct H2 as [H2 | [H2 | H2]]...
    inversion H2; subst; clear H2...
Qed.

Lemma controller_send_pres_P : ∀ sws ofLinks0 ofLinks1
  ctrl0 ctrl1 swId0 msg switchm ctrlm,
  Send ctrl0 ctrl1 swId0 msg →
  AllDiff of_to (ofLinks0 ++ (OpenFlowLink swId0 switchm ctrlm) :: ofLinks1) →
  AllDiff swId (to_list sws) →
  Invariant sws
            (ofLinks0 ++ (OpenFlowLink swId0 switchm ctrlm) :: ofLinks1)
            ctrl0 →
  Invariant sws
            (ofLinks0 ++ (OpenFlowLink swId0 switchm (msg :: ctrlm)) :: ofLinks1)
            ctrl1.
Proof with eauto with datatypes.
```

```
intros sws ofLinks0 ofLinks1 ctrl0 ctrl1 swId0 msg switchm0 ctrlm0 H Hdiff HdiffSws H0.
inversion H; subst.
+ inversion H0; subst.
  eapply MkP...
  intros.
  apply in_app_iff in H2; simpl in H2.
  destruct H2 as [H2 | [H2 | H2]]...
  inversion H2; subst.
  edestruct H6 as [pendingMsgs [HIn [HNotPktOuts HBarriered]]]...
  ∃ pendingMsgs.
  split...
  split...
  apply barriered_splice_PacketOut...
+ inversion H0; subst.
  eapply MkP.
  eapply AllDiff_preservation...
  do 2 rewrite → map_app...
  intros.
  destruct (eqdec swId1 swId0)...
  - destruct sw; simpl in *; subst...
    assert (AllDiff of_to (ofLinks0 ++ OpenFlowLink swId0 switchm0 (msg::ctrlm0) :: ofLinks1)) as Hdiff0.
    { eapply AllDiff_preservation... do 2 rewrite → map_app... }
    assert (OpenFlowLink swId0 switchmLst ctrlmLst = OpenFlowLink swId0 switchm0 (msg::ctrlm0)).
    { eapply AllDiff_uniq... }
    inversion H3; subst; clear H3.
    edestruct H6 as [pendingMsgs [HIn [HNotPktOuts HBarriered]]]...
    assert (msg :: msgs = pendingMsgs) as X.
    { assert (SwitchState swId0 (msg::msgs) = SwitchState swId0 pendingMsgs).
      eapply AllDiff_uniq...
      inversion H3; subst... }
    inversion X; subst.
    simpl in *.
    ∃ msgs...
    split...
    split...
    rewrite ← app_assoc in HBarriered.
    simpl in HBarriered...
  - destruct sw; subst; simpl in *.
    apply in_app_iff in H2; simpl in H2.
    { destruct H2 as [H2|[H2|H2]].
      + edestruct H6 with (swId1 := swId2) as [pendingMsgs [HIn [HNotPktOuts
HBarriered]]]...
        simpl in *.
        ∃ pendingMsgs...
        split...
        apply in_app_iff in HIn; simpl in HIn.
        destruct HIn as [HIn|[HIn|HIn]]...
        inversion HIn; subst.
        contradiction n...
```

```
          + inversion H2; subst.
            contradiction n...
          + edestruct H6 with (swId1 := swId2) as [pendingMsgs [HIn [HNotPktOuts
HBarriered]]]...
            simpl in *.
            ∃ pendingMsgs...
            split...
            apply in_app_iff in HIn; simpl in HIn.
            destruct HIn as [HIn|[HIn|HIn]]...
            inversion HIn; subst.
            contradiction n... }
  Qed.

  Lemma controller_step_pres_P : ∀ sws ofLinks ctrl0 ctrl1,
    Step ctrl0 ctrl1 →
    Invariant sws ofLinks ctrl0 →
    Invariant sws ofLinks ctrl1.
  Proof.
    intros.
    inversion H.
  Qed.

  Local Open Scope bag_scope.
  Hint Constructors NotFlowMod.

  Lemma Invariant_ofLink_vary : ∀ sws swId switchm0 switchm1 ctrlm
                                         ofLinks0 ofLinks1 ctrl,
    Invariant sws
```

```
          (ofLinks0 ++ OpenFlowLink swId switchm0 ctrlm :: ofLinks1)
          ctrl →
  Invariant sws
          (ofLinks0 ++ OpenFlowLink swId switchm1 ctrlm :: ofLinks1)
          ctrl.
Proof with eauto with datatypes.
  intros.
  inversion H; subst.
  eapply MkP...
  intros.
  apply in_app_iff in H3; simpl in H2.
  destruct H3 as [H3 | [H3 | H3]]...
  destruct sw. simpl in *. subst.
  inversion H3; clear H3; subst...
  edestruct H1 as [pendingMsgs0 [HIn [HNotPktOuts HBarriered]]]...
Qed.

Lemma Invariant_sw_vary : ∀ swId pts tbl inp outp cms sms sws
                              ofLinks ctrl inp' outp' sms',
  Invariant (({|Switch swId pts tbl inp outp cms sms|}) <+> sws)
          ofLinks
          ctrl →
  Invariant (({|Switch swId pts tbl inp' outp' cms sms'|}) <+> sws)
          ofLinks
          ctrl.
Proof with eauto with datatypes.
```

```
  intros.
  inversion H; subst.
  eapply MkP...
  intros.
  apply Bag.in_union in H2; simpl in H2.
  destruct H2 as [[H2 | H2] | H2]...
  - subst. simpl in *.
    remember (Switch swId0 pts0 tbl0 inp0 outp0 cms sms) as sw.
    assert (swId0 = swId sw) as X.
    { subst. simpl... }
    edestruct H1 as [msgs [HIn [HNotPktOuts HBarriered]]]...
    { apply Bag.in_union. left. simpl... }
    ∃ msgs...
    split...
    split...
    destruct sw; subst; simpl in *.
    inversion Heqsw; subst...
  - inversion H2.
  - destruct sw. simpl in *; subst.
    edestruct H1 as [msgs [HIn [HNotPktOuts HBarriered]]]...
    { apply Bag.in_union. right... }
    simpl...
    ∃ msgs...
Qed.

Lemma step_preserves_P : ∀ sws0 sws1 links0 links1 ofLinks0 ofLinks1
```

```
    ctrl0 ctrl1 obs,
    AllDiff of_to ofLinks0 →
    AllDiff swId (to_list sws0) →
    step (Machine.State sws0 links0 ofLinks0 ctrl0)
         obs
         (Machine.State sws1 links1 ofLinks1 ctrl1) →
    Invariant sws0 ofLinks0 ctrl0 →
    Invariant sws1 ofLinks1 ctrl1.
  Proof with eauto with datatypes.
    intros sws0 sws1 links0 links1 ofLinks0 ofLinks1 ctrl0 ctrl1 obs HdiffOfLinks HdiffSws H H0.
    destruct ctrl1.
    inversion H0; subst.
    rename H0 into HInvariant.
    inversion H; subst.
    + eapply Invariant_sw_vary...
    + eapply MkP...
      intros.
      apply Bag.in_union in H0; simpl in H0.
      destruct H0 as [[H0 | H0] | H0]...
      - subst; simpl in *.
        edestruct H2 as [msgs [HIn [HNotPktOuts HBarriered]]]...
        apply Bag.in_union. simpl. left. left. reflexivity. simpl...
        simpl in *.
        inversion HBarriered; subst.
        × assert (NotFlowMod (FlowMod fm)) as X.
      apply H0.
      apply Bag.in_union; simpl...
      inversion X.
    × ∃ msgs...
      split...
      split...
      apply Bag.union_from_ordered in H0.
      assert (FlowMod fm = FlowMod f ∧ ctrlm0 = ctrlm1) as HEq.
      { eapply Bag.singleton_union_disjoint.
        symmetry...
        intros.
        assert (NotFlowMod (FlowMod fm)) as X...
        inversion X. }
      destruct HEq as [HEq HEq0].
      inversion HEq; subst.
      eapply Barriered_NoFlowMods...
      apply approximating_pop_FlowMod_safe in H6...
  - inversion H0.
  - subst; simpl in *.
    edestruct H2 as [msgs [HIn [HNotPktOuts HBarriered]]]...
    apply Bag.in_union...
+
  eapply MkP...
  intros.
  destruct sw.
  simpl in *.
  subst.
  apply Bag.in_union in H0. simpl in H0.
  destruct H0 as [[H0|H0]|H0]...
  - subst. simpl in *.
    inversion H0; subst...
    edestruct H2 as [msgs [HIn [HNotPktOuts HBarriered]]]...
    simpl.
    apply Bag.in_union. left. simpl. left. reflexivity.
    simpl...
    eexists msgs...
    split...
    split...
    simpl in HBarriered...
    eapply barriered_process_PacketOut...
- inversion H0.
- edestruct H2 as [msgs [HIn [HNotPktOuts HBarriered]]]...
  { apply Bag.in_union. right... }
  simpl...
  ∃ msgs...
+ eapply Invariant_sw_vary...
+ eapply Invariant_sw_vary...
+ eapply controller_step_pres_P...
+ eapply controller_recv_pres_P...
+ eapply controller_send_pres_P...
+ eapply Invariant_sw_vary...
  eapply Invariant_ofLink_vary...
```

```
+ eapply MkP...
  intros.
  destruct sw; subst; simpl in *.
  destruct (TotalOrder.eqdec swId0 swId2).
  - subst.
    assert (OpenFlowLink swId2 switchmLst ctrlmLst =
                OpenFlowLink swId2 fromSwitch0 fromCtrl) as X.
    { assert (AllDiff of_to (ofLinks2 ++ OpenFlowLink swId2 fromSwitch0 fromCtrl
:: ofLinks3)).
      eapply AllDiff_preservation...
      do 2 rewrite → map_app...
      eapply AllDiff_uniq... }
    inversion X; clear H2; subst; clear X.
    assert (Switch swId2 pts0 tbl0 inp0 outp0 ({||}) (({|BarrierReply xid|})<+>switchm0)
=
                Switch swId2 pts1 tbl1 inp1 outp1 ctrlm0 switchm1) as X.
    { assert (AllDiff swId (to_list (({|Switch swId2 pts0 tbl0 inp0 outp0 ({||}) ({|Bar-
rierReply xid|} <+> switchm0)|}) <+> sws))).
      { eapply Bag.AllDiff_preservation... }
      clear HdiffSws.
      eapply AllDiff_uniq...
      apply Bag.in_union; simpl... }
    inversion X; clear H0; subst; clear X.
    inversion HInvariant; subst.
    edestruct H7 as [msgs [HIn [HNotPktOuts HBarriered]]]...
    { apply Bag.in_union. left. simpl. left. reflexivity. }
```

```
      { simpl... }
      simpl in *.
      ∃ msgs...
      split...
      split...
      eapply barriered_pop_BarrierRequest...
      rewrite → app_assoc in HBarriered...
      intros.
      simpl in H0.
      inversion H0.
    - apply Bag.in_union in H0; simpl in H0.
      destruct H0 as [[H0|H0]|H0].
      × inversion H0; subst. contradiction n...
      × inversion H0.
      × edestruct H2 with (swId1:=swId2) as [msgs [HIn [HNotPktOuts HBarriered]]]...
        apply Bag.in_union...
        apply in_app_iff in H3; simpl in H3.
        destruct H3 as [H3|[H3|H3]]...
        inversion H3; subst; contradiction n...
        simpl...
        ∃ msgs...
+ eapply MkP...
  intros.
  destruct sw; subst; simpl in *.
  destruct (TotalOrder.eqdec swId0 swId2).
  - subst.
```

```
assert (OpenFlowLink swId2 switchmLst ctrlmLst =
        OpenFlowLink swId2 fromSwitch0 fromCtrl) as X.
{ assert (AllDiff of_to (ofLinks2 ++ OpenFlowLink swId2 fromSwitch0 fromCtrl
:: ofLinks3)).
  eapply AllDiff_preservation...
  do 2 rewrite → map_app...
  eapply AllDiff_uniq... }
inversion X; clear H2; subst; clear X.
assert (Switch swId2 pts1 tbl1 inp1 outp1 ctrlm1 switchm1 =
        Switch swId2 pts0 tbl0 inp0 outp0 ({|msg|} <+> ctrlm0) switchm0) as
X.
{ assert (AllDiff swId (to_list (({|Switch swId2 pts0 tbl0 inp0 outp0 (({|msg|})<+>
ctrlm0) switchm0|}) <+> sws))).
  { eapply Bag.AllDiff_preservation... }
  clear HdiffSws.
  eapply AllDiff_uniq...
  apply Bag.in_union; simpl... }
inversion X; clear H0; subst; clear X.
inversion HInvariant; subst.
edestruct H8 as [msgs [HIn [HNotPktOuts HBarriered]]]...
{ apply Bag.in_union. left. simpl. left. reflexivity. }
{ simpl... }
simpl in *.
∃ msgs...
split...
split...
```

```
    destruct msg.
    × eapply barriered_pop_PacketOut...
      rewrite → app_assoc in HBarriered...
    × inversion H7.
    × { inversion HBarriered; subst.
        + eapply barriered_pop_FlowMod...
          rewrite ← app_assoc...
        + clear H HInvariant H1. move H2 after HBarriered.
          rewrite → app_assoc in H2.
          apply alternating_fm_fm_false in H2.
          inversion H2.
      }
  - intros.
    apply Bag.in_union in H0; simpl in H0.
    destruct H0 as [[H0|H0]|H0].
    × inversion H0; subst. contradiction n...
    × inversion H0.
    × edestruct H2 with (swId1:=swId2) as [msgs [HIn [HNotPktOuts HBarriered]]]...
      apply Bag.in_union...
      apply in_app_iff in H3; simpl in H3.
      destruct H3 as [H3|[H3|H3]]...
      inversion H3; subst; contradiction n...
      simpl...
      ∃ msgs...
Qed.
```

```
Definition relate_helper (sd : srcDst) : swPtPks :=
  match topo (pkSw sd,dstPt sd) with
    | None ⇒ {| |}
    | Some (sw',pt') ⇒ {| (sw',pt',dstPk sd) |}
  end.

Definition relate_controller (st : controller) :=
  unions (map relate_helper (pktsToSend st)).

Lemma ControllerRemembersPackets :
  ∀ (ctrl ctrl' : controller),
    controller_step ctrl ctrl' →
    relate_controller ctrl = relate_controller ctrl'.
Proof with auto.
  intros. inversion H.
Qed.

Lemma ControllerSendForgetsPackets : ∀ ctrl ctrl' sw msg,
  controller_send ctrl ctrl' sw msg →
  relate_controller ctrl = select_packet_out sw msg <+>
  relate_controller ctrl'.
Proof with auto.
  intros.
  inversion H; subst.
  + unfold relate_controller.
    simpl.
    unfold relate_helper.
    simpl.
```

```
      rewrite → Bag.unions_cons.
      reflexivity.
    + simpl.
      unfold relate_controller.
      simpl.
      { destruct msg.
        - idtac "TODO(arjun): Cannot pre-emit packetouts (need P here).".
          admit.
        - simpl. rewrite → Bag.union_empty_l...
        - simpl. rewrite → Bag.union_empty_l... }
Qed.

Lemma like_transfer : ∀ srcPt srcPk sw ptpk,
  relate_helper (mkPktOuts_body sw srcPt srcPk ptpk) =
  transfer sw ptpk.
Proof with auto.
  intros.
  unfold mkPktOuts_body.
  unfold relate_helper.
  unfold transfer.
  destruct ptpk.
  simpl.
  reflexivity.
Qed.

Lemma like_transfer_abs : ∀ sw pt pk lst,
  map
```

```
    (fun x : portId × packet ⇒ relate_helper (mkPktOuts_body sw pt pk x))
    lst =
  map (transfer sw) lst.
Proof with auto.
  intros.
  induction lst...
  simpl.
  rewrite → like_transfer.
  rewrite → IHlst.
  reflexivity.
Qed.

Lemma ControllerRecvRemembersPackets : ∀ ctrl ctrl' sw msg,
  controller_recv ctrl sw msg ctrl' →
  relate_controller ctrl' = select_packet_in sw msg <+>
  (relate_controller ctrl).
Proof with auto.
  intros.
  inversion H; subst.
  unfold relate_controller.
  simpl.
  rewrite → Bag.union_empty_l...
  unfold relate_controller.
  simpl.
  rewrite → map_app.
  rewrite → Bag.unions_app.
  apply Bag.pop_union_r.
  unfold mkPktOuts.
  rewrite → map_map.
  rewrite → like_transfer_abs...
Qed.

Definition P := Invariant.

Axiom ControllerLiveness : ∀ sw pt pk ctrl0 sws0 links0
                                                      ofLinks0,
  In (sw,pt,pk) (to_list (relate_controller ctrl0)) →
  ∃ ofLinks10 ofLinks11 ctrl1 swTo ptTo switchmLst ctrlmLst,
    (multistep
       step (Machine.State sws0 links0 ofLinks0 ctrl0) nil
       (Machine.State sws0 links0
              (ofLinks10 ++
               (OpenFlowLink swTo switchmLst
                (PacketOut ptTo pk :: ctrlmLst)) ::
               ofLinks11)
              ctrl1)) ∧
    select_packet_out swTo (PacketOut ptTo pk) = ({|(sw,pt,pk)|}).

Lemma ControllerRecvLiveness : ∀ sws0 links0 ofLinks0 sw switchm0 m
  ctrlm0 ofLinks1 ctrl0,
   ∃ ctrl1,
    (multistep
       step
       (Machine.State
            sws0 links0
            (ofLinks0 ++ (OpenFlowLink sw (switchm0 ++ [m]) ctrlm0) :: ofLinks1)
            ctrl0)
         nil
         (Machine.State
            sws0 links0
            (ofLinks0 ++ (OpenFlowLink sw switchm0 ctrlm0) :: ofLinks1)
            ctrl1)) ∧
      ∃ (lps : swPtPks),
         (select_packet_in sw m) <+> lps = relate_controller ctrl1.
Proof with eauto with datatypes.
  intros.
  destruct ctrl0.
  destruct m.
  + eexists.
    split.
    eapply multistep_tau.
    apply ControllerRecv.
    apply RecvPacketIn.
    apply multistep_nil.
    ∃ (relate_controller (State pktsToSend0 switchStates0)).
    simpl.
    unfold relate_controller.
    simpl.
    autorewrite with bag using simpl.
    unfold mkPktOuts.
```

```
      rewrite → map_map.
      rewrite → like_transfer_abs...
    + eexists.
      split.
      eapply multistep_tau.
      apply ControllerRecv.
      apply RecvBarrierReply.
      apply multistep_nil.
      simpl.
      eexists.
      rewrite → Bag.union_empty_l.
      reflexivity.
  Qed.
End MakeController.
```

### A.2.27 FwOFWeakSimulation1 Library

```
Set Implicit Arguments.
Require Import Coq.Lists.List.
Require Import Coq.Classes.Equivalence.
Require Import Coq.Structures.Equalities.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Setoids.Setoid.
Require Import Common.Types.
Require Import Common.Bisimulation.
Require Import Bag.Bag2.

Require Import FwOF.FwOFSignatures.

Local Open Scope list_scope.

Local Open Scope equiv_scope.

Local Open Scope bag_scope.

Arguments to_list _ _ _ : simpl never.

Module Make (Import RelationDefinitions : RELATION_DEFINITIONS).

  Import AtomsAndController.

  Import Machine.

  Import Atoms.

  Theorem weak_sim_1 :

    weak_simulation concreteStep abstractStep bisim_relation.

  Proof with auto with datatypes.

    unfold weak_simulation.

    intros.

    unfold bisim_relation in H.

    unfold relate in H.

    destruct s. simpl in *.

    unfold concreteStep in H0.

    destruct s'.

    { inversion H0; subst; simpl in *; subst; simpl in *.

      + idtac "Proving weak_sim_1 (Case 2 of 12) ...".

        autorewrite with bag using simpl.

        match goal with

          | [ ⊢ context[({|(swId0,pt,pk)|}) <+> ?t1] ] ⇒ remember t1
        end.
        ∃ (unions (map (transfer swId0) (abst_func swId0 pt pk)) <+> b).
        split.
        unfold bisim_relation.
        unfold relate.
        rewrite → Heqb.
        assert (FlowTableSafe swId0 tbl0) as J.
        { assert (FlowModSafe swId0 tbl0 ctrlm0) as J.
          refine (concreteState_flowTableSafety1
                      swId0 pts0 tbl0 inp0 (from_list outp' <+> outp0) ctrlm0
                      (from_list (map (PacketIn pt) pksToCtrl) <+> switchm0) _)...
          rewrite → Bag.in_union; simpl...
          inversion J... }
        unfold FlowTableSafe in J.
        pose (J0 := J pt pk outp' pksToCtrl H1).
        subst.
        rewrite ← J0.
        autorewrite with bag using simpl.
        bag_perm 100.
        apply multistep_obs with
        (a0 := (unions (map (transfer swId0) (abst_func swId0 pt pk)) <+> b)).
        apply AbstractStep.
        subst.
        apply multistep_nil.
      + idtac "Proving weak_sim_1 (Case 3 of 12) ...".
        autorewrite with bag using simpl.
```

```
match goal with
  | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
end.
∃ t.
split.
unfold bisim_relation.
unfold relate.
subst.
simpl.
autorewrite with bag using simpl.
bag_perm 100.
apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 4 of 12) ...".
autorewrite with bag using simpl.
match goal with
  | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
end.
∃ t.
split.
unfold bisim_relation.
unfold relate.
subst.
simpl.
autorewrite with bag using simpl.
bag_perm 100.
apply multistep_nil.
```

```
+ idtac "Proving weak_sim_1 (Case 5 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  destruct dst0.
  rename concreteState_consistentDataLinks0 into X.
  simpl in X.
  assert (In (DataLink (swId0,pt) pks0 (s,p))
                 (links0 ++ (DataLink (swId0,pt) pks0 (s,p)) :: links1)) as J...
  apply X in J.
  simpl in J.
  rewrite → J.
  autorewrite with bag using simpl.
  bag_perm 100.
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 6 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
```

```
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  bag_perm 100.
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 7 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  rewrite → (ControllerRemembersPackets H1)...
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 8 of 12) ...".
```

```
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  rewrite → (ControllerRecvRemembersPackets H1).
  bag_perm 100.
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 9 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  rewrite → (ControllerSendForgetsPackets H1).
  bag_perm 100.
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 10 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
  subst.
  simpl.
  autorewrite with bag using simpl.
  bag_perm 100.
  apply multistep_nil.
+ idtac "Proving weak_sim_1 (Case 11 of 12) ...".
  autorewrite with bag using simpl.
  match goal with
    | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
  end.
  ∃ t.
  split.
  unfold bisim_relation.
  unfold relate.
```

```
        subst.
        simpl.
        autorewrite with bag using simpl.
        bag_perm 100.
        apply multistep_nil.
      + idtac "Proving weak_sim_1 (Case 12 of 12) ...".
        autorewrite with bag using simpl.
        match goal with
          | [ ⊢ context [multistep _ ?X _ _] ] ⇒ remember X as t
        end.
        ∃ t.
        split.
        unfold bisim_relation.
        unfold relate.
        subst.
        simpl.
        autorewrite with bag using simpl.
        bag_perm 100.
        apply multistep_nil.
    }
  Qed.
End Make.
```

### A.2.28 FwOFWeakSimulation2 Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Common.Types.

Require Import Common.Bisimulation.

Require Import Common.AllDiff.

Require Import Bag.Bag2.

Require Import FwOF.FwOFSignatures.

Require Import Bag.TotalOrder.

Local Open Scope list_scope.

Local Open Scope bag_scope.

Module Make (Import Relation : RELATION).

  Import RelationDefinitions.

  Import AtomsAndController.

  Import Machine.

  Import Atoms.

  Lemma DrainWire : ∀ sws (swId : switchId) pts tbl inp outp
    ctrlm switchm links src pks0 pks swId pt links0 ofLinks ctrl,
     multistep step
      (State
        ({|Switch swId pts tbl inp outp ctrlm switchm|} <+> sws)
        (links ++ (DataLink src (pks0 ++ pks) (swId,pt)) :: links0)
        ofLinks ctrl)
      nil
    (State
      ({|Switch swId pts tbl (from_list (map (fun pk ⇒ (pt,pk)) pks) <+> inp) outp
ctrlm switchm|} <+> sws)
      (links ++ (DataLink src pks0 (swId,pt)) :: links0)
      ofLinks ctrl).
  Proof with auto.
    intros.
    generalize dependent inp0.
    generalize dependent pks1.
    induction pks1 using rev_ind.
    + intros.
      simpl in *.
      rewrite → app_nil_r.
      rewrite → Bag.from_list_nil_is_empty.
      rewrite → Bag.union_empty_l.
      apply multistep_nil.
    + intros.
      rewrite → (app_assoc pks0 pks1).
      rewrite → map_app.
      eapply multistep_tau.
      apply RecvDataLink.
      eapply multistep_app with
      (s2 := State ({|Switch swId1 pts0 tbl0 (from_list (map (fun pk ⇒ (pt,pk)) pks1)
<+> (({|(pt,x)|}) <+> inp0)) outp0 ctrlm0 switchm0|} <+> sws)
                        (links0 ++ DataLink src0 pks0 (swId1,pt) :: links1)
                        ofLinks0
```

```
                    ctrl0).
      apply (IHpks1 ( ({| (pt, x) |}) <+> inp0)).
      autorewrite with bag using simpl.
      apply multistep_nil.
      simpl...
  Qed.

Lemma ObserveFromOutp : ∀ pktOuts pktIns pk pt0 pt1
  swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
  swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1
  sws pks links0 links1 ofLinks0 ctrl0,
  (pktOuts, pktIns) = process_packet tbl1 pt1 pk →
  Some (swId1,pt1) = topo (swId0,pt0) →
  multistep step
    (State
      (({|Switch swId0 pts0 tbl0 inp0 ({|(pt0,pk)|} <+> outp0)
                ctrlm0 switchm0|}) <+>
       ({|Switch swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1|}) <+>
       sws)
      (links0 ++ (DataLink (swId0,pt0) pks (swId1,pt1)) :: links1)
      ofLinks0 ctrl0)
    [(swId1,pt1,pk)]
    (State
      (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+>
       ({|Switch swId1 pts1 tbl1
                (from_list (map (fun pk ⇒ (pt1,pk)) pks) <+> inp1)
```

```
                    (from_list pktOuts <+> outp1)
                    ctrlm1
                    (from_list (map (PacketIn pt1) pktIns) <+> switchm1)|}) <+>
         sws)
        (links0 ++ (DataLink (swId0,pt0) nil (swId1,pt1)) :: links1)
        ofLinks0 ctrl0).
Proof with simpl;eauto with datatypes.
  intros.
  eapply multistep_tau.
  apply SendDataLink.
  rewrite ← Bag.union_assoc.
  rewrite → (Bag.union_comm _ ({|Switch swId0 pts0 tbl0 inp0 outp0
                                              ctrlm0 switchm0|})).
  rewrite → Bag.union_assoc.
  eapply multistep_app with (obs2 := [(swId1,pt1,pk)]).
  apply (DrainWire
    (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+> sws)
    swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1
    links0 (swId0,pt0) [pk] pks0 swId1 pt1 links1 ofLinks0 ctrl0).
  assert ([pk] = nil ++ [pk]) as X... rewrite → X. clear X.
  eapply multistep_tau.
  apply RecvDataLink.
  eapply multistep_obs.
  apply PktProcess.
  instantiate (1 := pktIns).
  instantiate (1 := pktOuts).
  symmetry...
  match goal with
  | [ ⊢ multistep step _ nil ?Y ] ⇒ remember Y as S2
  end.
  subst.
  assert (∀ (b1 b2 b3 : bag switch_le), b1 <+> b2 <+> b3 = b2 <+> b1 <+> b3).
  { intros. bag_perm 100. }
  rewrite → H1.
  apply multistep_nil.
  trivial.
Qed.

Lemma ObserveFromOutp_same_switch : ∀ pktOuts pktIns pk pt0 pt1
  swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
  sws pks links0 links1 ofLinks0 ctrl0,
  (pktOuts, pktIns) = process_packet tbl0 pt1 pk →
  Some (swId0,pt1) = topo (swId0,pt0) →
  multistep step
    (State
      (({|Switch swId0 pts0 tbl0 inp0 ({|(pt0,pk)|} <+> outp0)
                  ctrlm0 switchm0|}) <+>
        sws)
      (links0 ++ (DataLink (swId0,pt0) pks (swId0,pt1)) :: links1)
      ofLinks0 ctrl0)
    [(swId0,pt1,pk)]
    (State
      (({|Switch swId0 pts0 tbl0
                  (from_list (map (fun pk ⇒ (pt1,pk)) pks) <+> inp0)
                  (from_list pktOuts <+> outp0)
                  ctrlm0
                  (from_list (map (PacketIn pt1) pktIns) <+> switchm0)|}) <+>
       sws)
      (links0 ++ (DataLink (swId0,pt0) nil (swId0,pt1)) :: links1)
      ofLinks0 ctrl0).
Proof with simpl;eauto with datatypes.
  intros.
  eapply multistep_tau.
  apply SendDataLink.
  eapply multistep_app with (obs2 := [(swId0,pt1,pk)]).
  apply (DrainWire
           sws
           swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
           links0 (swId0,pt0) [pk] pks0 swId0 pt1 links1 ofLinks0 ctrl0).
  rewrite ← (app_nil_l [pk]).
  eapply multistep_tau.
  apply RecvDataLink.
  eapply multistep_obs.
  eapply PktProcess...
  apply multistep_nil.
  trivial.
Qed.
```

```
Lemma DrainFromControllerBag : ∀ swId0 pts0 tbl0 inp0 outp0 ctrlm0
  switchm0 sws0 links0 ofLinks0 ctrl0,
  (∀ x, In x (to_list ctrlm0) → NotBarrierRequest x) →
  ∃ tbl1 outp1,
    multistep step
      (State
        (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+>
         sws0)
        links0 ofLinks0 ctrl0)
      nil
      (State
        (({|Switch swId0 pts0 tbl1 inp0 outp1 ({||}) switchm0|}) <+>
         sws0)
        links0 ofLinks0 ctrl0).
Proof with simpl;eauto with datatypes.
  intros swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0 sws0 links0 ofLinks0
         ctrl0 HNotBarrier.
  destruct ctrlm0.
  rename to_list into ctrlm0.
  generalize dependent tbl0.
  generalize dependent outp0.
  induction ctrlm0; intros.
  + ∃ tbl0.
    ∃ outp0.
    subst...
    assert (Bag nil order = empty). apply Bag.ordered_irr. simpl...
```

```
      rewrite → H.
      apply multistep_nil.
    + destruct a.
      - inversion order; subst.
        assert (∀ x, In x (to_list (Bag ctrlm0 H2)) → NotBarrierRequest x) as Y.
        { intros. apply HNotBarrier. simpl. right... }
        destruct (IHctrlm0 H2 Y ({|(p, p0)|} <+> outp0) tbl0)
          as [tbl1 [outp1 Hstep]].
        ∃ tbl1. ∃ outp1.
        assert (Bag (PacketOut p p0 :: ctrlm0) order = from_list (PacketOut p p0 ::
ctrlm0)).
        { apply Bag.ordered_irr... unfold to_list.
          simpl. rewrite → OrderedLists.insert_eq_head...
          f_equal. symmetry. apply OrderedLists.from_list_id...
          apply OrderedLists.from_list_order. intros. apply H1.
          rewrite → OrderedLists.in_from_list_iff... }
        rewrite → Bag.from_list_cons in H.
        eapply multistep_tau.
        rewrite → H.
        apply SendPacketOut.
        assert (Bag ctrlm0 H2 = from_list ctrlm0).
        { apply Bag.ordered_irr...
          unfold to_list.
          symmetry.
          apply OrderedLists.from_list_id... }
        rewrite → H0 in *.
      apply Hstep.
    - assert (NotBarrierRequest (BarrierRequest n)) as contra.
      { apply HNotBarrier... }
      inversion contra.
    - inversion order; subst.
      assert (∀ x, In x (to_list (Bag ctrlm0 H2)) → NotBarrierRequest x) as Y.
      { intros. apply HNotBarrier. simpl. right... }
      destruct (IHctrlm0 H2 Y outp0 (modify_flow_table f tbl0)) as [tbl1 [outp1 Hstep]].
      ∃ tbl1.
      ∃ outp1.
      assert (Bag (FlowMod f :: ctrlm0) order = (({|FlowMod f|}) <+> (Bag ctrlm0
H2))).
      { apply Bag.ordered_irr.
        unfold to_list. simpl.
        symmetry. apply OrderedLists.union_cons... }
      rewrite → H.
      eapply multistep_tau.
      apply ModifyFlowTable.
      apply Hstep.
Qed.

Lemma DrainFromController : ∀ swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
  sws0 links0 ofLinks0 lstSwitchm0 lstCtrlm0 lstCtrlm1 ofLinks1 ctrl0,
  (∀ x, In x (to_list ctrlm0) → NotBarrierRequest x) →
  ∃ tbl0' ctrlm0' outp0' switchm1,
    multistep step
    (State
        (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+>
          sws0)
        links0
        (ofLinks0 ++
          (OpenFlowLink swId0 lstSwitchm0 (lstCtrlm0 ++ lstCtrlm1)) ::
          ofLinks1)
        ctrl0)
    nil
    (State
        (({|Switch swId0 pts0 tbl0' inp0 outp0' ctrlm0' switchm1|}) <+>
          sws0)
        links0
        (ofLinks0 ++ (OpenFlowLink swId0 lstSwitchm0 lstCtrlm0) ::
          ofLinks1)
        ctrl0).
Proof with simpl;eauto with datatypes.
  intros swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0 sws0 links0 ofLinks0
          lstSwitchm0 lstCtrlm0 lstCtrlm1 ofLinks1 ctrl0 HNotBarrier.
  generalize dependent tbl0.
  generalize dependent ctrlm0.
  generalize dependent outp0.
  generalize dependent switchm0.
  induction lstCtrlm1 using rev_ind; intros.
  ∃ tbl0.
  ∃ ctrlm0.
```

```
    ∃ outp0.
    ∃ switchm0.
    rewrite → app_nil_r.
    apply multistep_nil.
    destruct x.
    + assert (∀ x, In x (to_list ({|PacketOut p p0|} <+> ctrlm0)) → NotBarrierRequest
x) as Y.
        { intros.
          apply Bag.in_union in H; simpl in H.
          destruct H...
          destruct H. subst; apply PacketOut_NotBarrierRequest. inversion H. }
      destruct (IHlstCtrlm1 switchm0 outp0 (({|PacketOut p p0|}) <+> ctrlm0) Y
tbl0)
          as [tbl1 [ctrlm1 [outp1 [switchm1 Hstep]]]].
      ∃ tbl1.
      ∃ ctrlm1.
      ∃ outp1.
      ∃ switchm1.
      eapply multistep_tau.
      rewrite → app_assoc.
      apply RecvFromController.
      apply PacketOut_NotBarrierRequest.
      exact Hstep.
    + destruct
        (DrainFromControllerBag swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0 sws0
                              links0
```

(*ofLinks0* ++
  (*OpenFlowLink swId0 lstSwitchm0*
                (*lstCtrlm0* ++ *lstCtrlm1* ++ [*BarrierRequest n*])) ::
  *ofLinks1*)
*ctrl0*)

as [*tbl1* [*outp1 Hdrain*]]...

assert (∀ *x*, *In x* (*to_list* (@*empty fromController fromController_le*)) → *NotBarrierRequest x*) as *Y*.

{ intros. simpl in *H*. inversion *H*. }

destruct (*IHlstCtrlm1* (({|*BarrierReply n*|}) <+> *switchm0*) *outp1 empty Y tbl1*)
  as [*tbl2* [*ctrlm2* [*outp2* [*switchm2 Hstep2*]]]].

∃ *tbl2*.

∃ *ctrlm2*.

∃ *outp2*.

∃ *switchm2*.

eapply *multistep_app*.

apply *Hdrain*.

eapply *multistep_tau*.

rewrite → *app_assoc*.

apply *RecvBarrier*.

apply *Hstep2*.

trivial.

+ assert (∀ *x*, *In x* (*to_list* ({|*FlowMod f*|} <+> *ctrlm0*)) → *NotBarrierRequest x*) as *Y*.

{ intros.

apply *Bag.in_union* in *H*; simpl in *H*.

```
          destruct H...
          destruct H. subst; apply FlowMod_NotBarrierRequest. inversion H. }
      destruct (IHlstCtrlm1 switchm0 outp0 (({|FlowMod f|}) <+> ctrlm0) Y tbl0)
        as [tbl1 [ctrlm1 [outp1 [switchm1 Hstep]]]].
       ∃ tbl1.
       ∃ ctrlm1.
       ∃ outp1.
       ∃ switchm1.
       eapply multistep_tau.
       rewrite → app_assoc.
       apply RecvFromController.
       apply FlowMod_NotBarrierRequest.
       exact Hstep.
Qed.

Lemma ObserveFromController : ∀
  (srcSw dstSw : switchId)
  dstPt pk pktOuts pktIns srcPt
  pts0 tbl0 inp0 outp0 ctrlm0 switchm0
  pts1 tbl1 inp1 outp1 ctrlm1 switchm1
  sws0 links0 pks0 links1
  ofLinks0 lstSwitchm0 lstCtrlm0 lstCtrlm1 ofLinks1
  ctrl0,
  (pktOuts, pktIns) = process_packet tbl1 dstPt pk →
  Some (dstSw,dstPt) = topo (srcSw, srcPt) →
  (∀ x, In x (to_list ctrlm0) → NotBarrierRequest x) →
```

```
∃ tbl0' switchm0' ctrlm0' outp0',
  multistep step
    (State
       (({|Switch srcSw pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+>
        ({|Switch dstSw pts1 tbl1 inp1 outp1 ctrlm1 switchm1|}) <+>
        sws0)
       (links0 ++ (DataLink (srcSw,srcPt) pks0 (dstSw,dstPt)) :: links1)
       (ofLinks0 ++
        (OpenFlowLink srcSw lstSwitchm0
                       (lstCtrlm0 ++ (PacketOut srcPt pk) :: lstCtrlm1)) ::
        ofLinks1)
       ctrl0)
    [(dstSw,dstPt,pk)]
    (State
       (({|Switch srcSw pts0 tbl0' inp0 outp0'
                ctrlm0' switchm0'|}) <+>
        ({|Switch dstSw pts1 tbl1
                (from_list (map (fun pk ⇒ (dstPt,pk)) pks0) <+> inp1)
                (from_list pktOuts <+> outp1)
                ctrlm1
                (from_list (map (PacketIn dstPt) pktIns) <+>
                 switchm1)|}) <+>
        sws0)
       (links0 ++ (DataLink (srcSw,srcPt) nil (dstSw,dstPt)) :: links1)
       (ofLinks0 ++
        (OpenFlowLink srcSw lstSwitchm0 lstCtrlm0) ::
              ofLinks1)
            ctrl0).
Proof with simpl;eauto with datatypes.
  intros.
  destruct (DrainFromController srcSw pts0 tbl0 inp0 outp0 ctrlm0 switchm0
                  (({|Switch dstSw pts1 tbl1 inp1 outp1 ctrlm1 switchm1|}) <+>
                   sws0)
                  (links0 ++ (DataLink (srcSw,srcPt) pks0 (dstSw,dstPt)) :: links1)
                  ofLinks0
                  lstSwitchm0
                  (lstCtrlm0 ++ [PacketOut srcPt pk])
                  lstCtrlm1
                  ofLinks1
                  ctrl0) as [tbl01 [ctrlm01 [outp01 [switchm01 Hdrain]]]]...
  ∃ tbl01.
  ∃ switchm01.
  ∃ ctrlm01.
  ∃ outp01.
  eapply multistep_app.
  assert (lstCtrlm0 ++ PacketOut srcPt pk :: lstCtrlm1 =
            (lstCtrlm0 ++ [PacketOut srcPt pk]) ++ lstCtrlm1) as X.
    rewrite ← app_assoc...
   rewrite → X. clear X.
  exact Hdrain.
  clear Hdrain.
  eapply multistep_tau.
  apply RecvFromController.
  apply PacketOut_NotBarrierRequest.
  eapply multistep_tau.
  apply SendPacketOut.
  eapply ObserveFromOutp...
  trivial.
Qed.

Lemma EasyObservePacketOut : ∀ sw pt srcSw p switches0 links0 ofLinks01
  of_switchm0 lstCtrlm0 pk lstCtrlm1 ofLinks02 ctrl0
  (linksHaveSrc0 : LinksHaveSrc switches0 links0)
  (linksHaveDst0 : LinksHaveDst switches0 links0)
  (devicesFromTopo0 : DevicesFromTopo (State switches0 links0
                 (ofLinks01 ++
                  (OpenFlowLink srcSw of_switchm0
                               (lstCtrlm0 ++ PacketOut p pk :: lstCtrlm1)) ::
                  ofLinks02)
                 ctrl0))
  (Htopo : Some (sw,pt) = topo (srcSw,p)),
  NoBarriersInCtrlm switches0 →
  ∃ state1,
    multistep
      step
      (State switches0 links0
                 (ofLinks01 ++
                  (OpenFlowLink srcSw of_switchm0
                                (lstCtrlm0 ++ PacketOut p pk :: lstCtrlm1)) ::
                  ofLinks02)
                 ctrl0)
          [(sw,pt,pk)]
          state1.
Proof with simpl;eauto with datatypes.
  intros.
  rename H into HNoBarriers.
  assert (∃ pks, In (DataLink (srcSw,p) pks (sw,pt)) links0) as X.
  {
    unfold DevicesFromTopo in devicesFromTopo0.
    apply devicesFromTopo0 in Htopo.
    destruct Htopo as [sw0 [sw1 [lnk [_ [_ [Hlnk [_ [_ [HIdEq0 HIdEq1]]]]]]]]].
    simpl in Hlnk.
    destruct lnk.
    subst.
    simpl in *.
    rewrite → HIdEq0 in Hlnk.
    rewrite → HIdEq1 in Hlnk.
    ∃ pks0... }
  destruct X as [pks Hlink]. apply in_split in Hlink.
  destruct Hlink as [links01 [links02 Hlink]]. subst.
  assert (LinkHasSrc switches0 (DataLink (srcSw,p) pks (sw,pt))) as X.
    apply linksHaveSrc0...
  unfold LinkHasSrc in X.
```

```
simpl in X.
destruct X as [switch0 [HMemSw0 [HSwId0Eq HPtsIn0]]].
assert (LinkHasDst switches0 (DataLink (srcSw,p) pks (sw,pt))) as X.
  apply linksHaveDst0...
unfold LinkHasDst in X.
simpl in X.
destruct X as [switch1 [HMemSw1 [HSwId1Eq HPtsIn1]]].
destruct switch0.
destruct switch1.
subst.
simpl in *.
remember (process_packet tbl1 pt pk) as X eqn:Hprocess.
destruct X as [pktOuts pktIns].
apply Bag.in_split with (Order := TotalOrder_switch) in HMemSw0.
destruct HMemSw0 as [sws0 HMemSw0].
subst.
apply Bag.in_split with (Order := TotalOrder_switch) in HMemSw1.
destruct HMemSw1 as [sws1 HMemSw0].
subst.
destruct (TotalOrder.eqdec swId0 swId1) as [HEq | HNeq].
+ subst.
  destruct (DrainFromController swId1 pts0 tbl0 inp0 outp0 ctrlm0 switchm0 sws0
                                              (links01 ++ DataLink (swId1,p) pks (swId1,pt) ::
links02)
                                              ofLinks01 of_switchm0 (lstCtrlm0 ++ [PacketOut
p pk])
                                    lstCtrlm1 ofLinks02 ctrl0) as
        [tbl2 [ctrlm2 [outp2 [switchm2 Hstep1]]]].
    { unfold NoBarriersInCtrlm in HNoBarriers.
      remember (Switch swId1 pts0 tbl0 inp0 outp0 ctrlm0 switchm0) as sw.
      assert (ctrlm0 = ctrlm sw) as HEq.
      { subst... }
      rewrite → HEq.
      refine (HNoBarriers _ _).
      apply Bag.in_union; simpl... }
    rewrite ← app_assoc in Hstep1.
    simpl in Hstep1.
    remember (process_packet tbl2 pt pk) as J.
    destruct J.
    symmetry in HeqJ.
    eexists.
    eapply multistep_app.
    exact Hstep1.
    eapply multistep_tau.
    apply RecvFromController.
    { apply PacketOut_NotBarrierRequest. }
    eapply multistep_tau.
    apply SendPacketOut.
    eapply multistep_tau.
    apply SendDataLink.
    eapply multistep_app.
```

```
      { assert (pk :: pks = [pk] ++ pks) as J. auto.
        rewrite → J.
        apply DrainWire... }
      eapply multistep_tau.
      { assert ([pk] = nil ++ [pk]) as J.
        { auto. }
        rewrite → J.
        apply RecvDataLink. }
      eapply multistep_obs.
      eapply PktProcess...
      eapply multistep_nil.
      reflexivity.
      reflexivity.
    + assert (In (Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0) (to_list sws1)) as
J.
      { assert (In (Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0)
                     (to_list (({|Switch swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1|})
<+> sws1))).
        { rewrite ← HMemSw0. apply Bag.in_union; simpl... }
        apply Bag.in_union in H. simpl in H.
        destruct H as [[H|H]|H]...
        + inversion H. subst. contradiction HNeq...
        + inversion H. }
      apply Bag.in_split with (Order:=TotalOrder_switch) in J.
      destruct J as [otherSws Heq].
      rewrite → Heq in HMemSw0.
```

```
        assert (In (Switch swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1) (to_list sws0)) as
J.
        { assert (In (Switch swId1 pts1 tbl1 inp1 outp1 ctrlm1 switchm1)
                         (to_list (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|})
<+> sws0))).
          { rewrite -> HMemSw0. apply Bag.in_union; simpl... }
          apply Bag.in_union in H. simpl in H.
          destruct H as [[H|H]|H]...
          + inversion H. subst. contradiction HNeq...
          + inversion H. }
        apply Bag.in_split with (Order:=TotalOrder_switch) in J.
        destruct J as [otherSws0 Heq0].
        rewrite -> Heq0 in HMemSw0.
        move HMemSw0 after Heq0.
        do 2 rewrite <- Bag.union_assoc in HMemSw0.
        rewrite <- (Bag.union_comm _ ({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}))
in HMemSw0.
        do 2 rewrite -> Bag.union_assoc in HMemSw0.
        apply Bag.pop_union_l in HMemSw0.
        apply Bag.pop_union_l in HMemSw0.
        subst.
      destruct (@ObserveFromController swId0 swId1 pt pk pktOuts pktIns p
                 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
                 pts1 tbl1 inp1 outp1 ctrlm1 switchm1
                 otherSws
                 links01 pks links02
                 ofLinks01 of_switchm0 lstCtrlm0 lstCtrlm1 ofLinks02
                 ctrl0 Hprocess Htopo) as
          [ tbl2 [switchm2 [ctrlm2 [outp2 Hstep]]]].
      { intros.
        unfold NoBarriersInCtrlm in HNoBarriers.
        eapply HNoBarriers.
        apply Bag.in_union. left. simpl. left. reflexivity.
        simpl... }
      eexists. exact Hstep.
    Qed.

    Lemma DrainToController : ∀ sws0 links0 ofLinks00 swId0 switchm0
      switchm1 ctrlm0 ofLinks01 ctrl0,
      ∃ sws1 links1 ofLinks10 ofLinks11 ctrl1,
        multistep step
          (State
             sws0
             links0
             (ofLinks00 ++
                 (OpenFlowLink swId0 (switchm0 ++ switchm1) ctrlm0) ::
                 ofLinks01)
             ctrl0)
          nil
          (State
             sws1
```

```
          links1
          (ofLinks10 ++
           (OpenFlowLink swId0 switchm0 ctrlm0) ::
           ofLinks11)
          ctrl1).
Proof with simpl;eauto with datatypes.
  intros.
  generalize dependent ctrl0.
  induction switchm1 using rev_ind; intros.
  ∃ sws0. ∃ links0. ∃ ofLinks00. ∃ ofLinks01.
  ∃ ctrl0. rewrite → app_nil_r. apply multistep_nil.
  destruct (ControllerRecvLiveness sws0 links0 ofLinks00 swId0
                                      (switchm0 ++ switchm1) x
                                      ctrlm0 ofLinks01 ctrl0)
           as [ctrl1 [Hstep1 _]].
  destruct (IHswitchm1 ctrl1) as
       [sws1 [links1 [ofLinks10 [ofLinks11 [ctrl2 Hstep2]]]]].
  ∃ sws1. ∃ links1. ∃ ofLinks10. ∃ ofLinks11.
  ∃ ctrl2.
  eapply multistep_app.
    rewrite → app_assoc.
    apply Hstep1.
    apply Hstep2.
  trivial.
Qed.
```

```
  Instance PtPk_TotalOrder : TotalOrder (PairOrdering portId_le packet_le).
  Proof. apply TotalOrder_pair; eauto. exact TotalOrder_portId. exact TotalOrder_packet.
Defined.
  Theorem weak_sim_2 :
    weak_simulation abstractStep concreteStep (inverse_relation bisim_relation).
  Proof with simpl;eauto with datatypes.
    unfold weak_simulation.
    intros.
    unfold inverse_relation in H.
    unfold bisim_relation in H.
    unfold relate in H.
    inversion H0; subst.
    simpl.
    remember (devices t) as devices0.
    destruct devices0.
    simpl in *.
    assert (In (sw,pt,pk) (to_list (({|(sw,pt,pk)|}) <+> lps))) as J.
    { apply Bag.in_union... }
    rewrite → H1 in J.
    repeat rewrite → Bag.in_union in J.
    destruct J as [HMemSwitch | [ HMemLink | [HMemOFLink | HMemCtrl ]]].
    apply Bag.in_unions in HMemSwitch.
    destruct HMemSwitch as [switch_abst [XIn XMem]].
    apply in_map_iff in XIn.
    destruct XIn as [switch [XRel XIn]].
```

```
subst.
destruct switch.
simpl in XMem.
repeat rewrite → Bag.in_union in XMem.
destruct XMem as [HMemInp | [HMemOutp | [HMemCtrlm | HMemSwitchm]]].
{ apply Bag.in_to_from_list in HMemInp.
  apply in_map_iff in HMemInp.
  destruct HMemInp as [[pt0 pk0] [Haffix HMemInp]].
  simpl in Haffix. inversion Haffix. subst. clear Haffix.
  apply Bag.in_split with (Order := PtPk_TotalOrder) in HMemInp.
  destruct HMemInp as [inp HEqInps].
  subst.
  apply Bag.in_split with (Order := TotalOrder_switch) in XIn.
  destruct XIn as [sws0 XIn].
  remember (process_packet tbl0 pt pk) as ToCtrl eqn:Hprocess.
  destruct ToCtrl as (outp',inPkts).
  eapply simpl_weak_sim.
  rewrite ← Heqdevices0.
  rewrite → XIn.
  apply multistep_obs with (a0 := State (({|Switch sw pts0 tbl0 inp (from_list outp'
<+> outp0) ctrlm0
                                                  (from_list (map (PacketIn pt) inPkts) <+>
switchm0) |}) <+> sws0)
                                                  links0 ofLinks0 ctrl0).
  apply PktProcess...
```

```
    eapply multistep_nil.
    rewrite ← Heqdevices0.
    unfold relate.
    simpl...
    rewrite → H1... }
```

```
  Instance SwPtPk_TotalOrder : TotalOrder (PairOrdering (PairOrdering switchId_le
portId_le) packet_le).
  Proof.
    apply TotalOrder_pair.
    apply TotalOrder_pair.
    exact TotalOrder_switchId.
    exact TotalOrder_portId.
    exact TotalOrder_packet.
  Defined.
```

```
    { apply Bag.in_unions in HMemOutp.
      destruct HMemOutp as [lps0 [HIn HMemOutp]].
      apply in_map_iff in HIn.
```

```
destruct HIn as [[srcPt srcPk] [HTransfer HIn]].
rename swId0 into srcSw.
simpl in HTransfer.
remember (topo (srcSw,srcPt)) as Htopo.
destruct Htopo.
+ destruct p as [dstSw dstPt].
  subst.
  simpl in HMemOutp.
  destruct HMemOutp. 2: inversion H.
  symmetry in H. inversion H. subst. clear H.
  rename srcPk into pk.
  assert (∃ pks, In (DataLink (srcSw,srcPt) pks (dstSw,dstPt)) links0) as X.
  {
    destruct t.
    unfold DevicesFromTopo in devicesFromTopo0.
    apply devicesFromTopo0 in HeqHtopo.
    destruct HeqHtopo as [sw0 [sw1 [lnk [_ [_ [Hlnk [_ [_ [HIdEq0 HIdEq1]]]]]]]]].
    simpl in Hlnk.
    destruct lnk.
    subst.
    simpl in *.
    rewrite → HIdEq0 in Hlnk.
    rewrite → HIdEq1 in Hlnk.
    ∃ pks0...
    rewrite ← Heqdevices0 in Hlnk.
    simpl in Hlnk... }
```

```
destruct X as [pks Hlink].
apply in_split in Hlink.
destruct Hlink as [links01 [links02 Hlink]].
assert
  (LinkHasDst
     switches0
     (DataLink (srcSw,srcPt) pks (dstSw,dstPt))) as J0.
{ destruct t.
  simpl in Heqdevices0.
  rewrite ← Heqdevices0 in *.
  subst.
  apply linksHaveDst0... }
unfold LinkHasDst in J0.
destruct J0 as [switch2 [HSw2In [HSw2IdEq HSw2PtsIn]]].
destruct switch2.
simpl in *.
symmetry in HSw2IdEq.
subst.
apply Bag.in_split with (Order := TotalOrder_switch) in XIn.
destruct XIn as [sws XXX].
subst.
apply Bag.in_union with (Order:=TotalOrder_switch) in HSw2In.
destruct HSw2In.
- simpl in H.
  destruct H; inversion H.
  subst; clear H.
  apply Bag.in_split with (Order:=PtPk_TotalOrder) in HIn.
  destruct HIn as [outp0' HEq0].
  subst.
  remember (process_packet tbl1 dstPt pk) as X eqn:Hprocess.
  destruct X as [outp1' pktIns].
  eapply simpl_weak_sim.
  rewrite ← Heqdevices0...
  eapply ObserveFromOutp_same_switch...
  rewrite ← Heqdevices0...
  apply AbstractStep.
- { apply Bag.in_split with (Order := TotalOrder_switch) in H.
    destruct H as [sws0 XXX].
    subst.
    apply Bag.in_split with (Order:=PtPk_TotalOrder) in HIn.
    destruct HIn as [outp0' HEq0].
    subst.
    rename outp0' into outp0.
    remember (process_packet tbl1 dstPt pk) as X eqn:Hprocess.
    destruct X as [outp1' pktIns].
    eapply simpl_weak_sim.
    rewrite ← Heqdevices0.
    eapply ObserveFromOutp...
    rewrite ← Heqdevices0.
    unfold relate.
    rewrite → H1.
    autorewrite with bag using simpl.
```

```
        reflexivity.
        apply AbstractStep. }
   + subst.
     unfold to_list in HMemOutp.
     simpl in HMemOutp.
     inversion HMemOutp. }
```

```
{ apply Bag.in_unions in HMemCtrlm.
   destruct HMemCtrlm as [ctrlmBag [HIn HMemCtrlm]].
   apply in_map_iff in HIn.
   destruct HIn as [ctrlm [Htopo HInMsg]].
   subst.
   destruct ctrlm.
   2: solve [ simpl in HMemCtrlm; inversion HMemCtrlm ].          2: solve [ simpl
in HMemCtrlm; inversion HMemCtrlm ].          simpl in HMemCtrlm.
   remember (topo (swId0,p)) as Htopo.
   destruct Htopo.
   2: solve [ simpl in HMemCtrlm; inversion HMemCtrlm ].
   destruct p1.
   simpl in HMemCtrlm.
   destruct HMemCtrlm. 2: solve [inversion H].
   inversion H. subst. clear H.
   apply Bag.in_split with (Order:=TotalOrder_fromController) in HInMsg.
```

```
destruct HInMsg as [ctrlm0' XX].
subst. rename ctrlm0' into ctrlm0.
assert (∃ pks, In (DataLink (swId0,p) pks (sw,pt)) links0) as X.
{
  destruct t.
  unfold DevicesFromTopo in devicesFromTopo0.
  apply devicesFromTopo0 in HeqHtopo.
  destruct HeqHtopo as [sw0 [sw1 [lnk [_ [_ [Hlnk [_ [_ [HIdEq0 HIdEq1]]]]]]]]].
  simpl in Hlnk.
  destruct lnk.
  subst.
  simpl in *.
  rewrite → HIdEq0 in Hlnk.
  rewrite → HIdEq1 in Hlnk.
  rewrite ← Heqdevices0 in *.
  simpl in *.
  ∃ pks0... }
destruct X as [pks Hlink].
apply in_split in Hlink.
destruct Hlink as [links01 [links02 Hlink]].
subst.
assert
  (LinkHasDst
     switches0
     (DataLink (swId0,p) pks (sw,pt))) as J0.
{ destruct t.
```

```
  simpl in *.
  rewrite ← Heqdevices0 in *.
  apply linksHaveDst0... }
unfold LinkHasDst in J0.
destruct J0 as [switch2 [HSw2In [HSw2IdEq HSw2PtsIn]]].
destruct switch2.
simpl in *.
subst.
apply Bag.in_split with (Order:=TotalOrder_switch) in XIn.
destruct XIn as [sws0 HEq1].
subst.
apply Bag.in_union in HSw2In.
destruct HSw2In.
+ simpl in H.
  destruct H; inversion H; subst; clear H.
  remember (process_packet tbl1 pt pk) as X eqn:Hprocess.
  destruct X as [outp1' pktIns].
  eapply simpl_weak_sim.
  rewrite ← Heqdevices0.
  eapply multistep_tau.
  apply SendPacketOut.
  eapply ObserveFromOutp_same_switch...
  rewrite ← Heqdevices0...
  apply AbstractStep.
+ apply Bag.in_split with (Order:=TotalOrder_switch) in H.
  destruct H as [sws XXX].
```

```
    subst.
    remember (process_packet tbl1 pt pk) as X eqn:Hprocess.
    destruct X as [outp1' pktIns].
    eapply simpl_weak_sim.
    rewrite ← Heqdevices0.
    eapply multistep_tau.
    apply SendPacketOut.
    eapply ObserveFromOutp...
    rewrite ← Heqdevices0...
    apply AbstractStep... }





{ apply Bag.in_unions in HMemSwitchm.
  destruct HMemSwitchm as [lps0 [HIn HMem]].
  apply in_map_iff in HIn.
  destruct HIn as [switchm [HEq HIn]].
  subst.
  destruct switchm.
  2: simpl in HMem; inversion HMem.          simpl in HMem.
  apply Bag.in_unions_map in HMem.
  destruct HMem as [[srcPt srcPk] [HInAbst HInTransfer]].
  simpl in HInTransfer.
  remember (topo (swId0, srcPt)) as Htopo.
```

```
destruct Htopo.
+ destruct p1.
  simpl in HInTransfer.
  destruct HInTransfer.
  2: inversion H.
  inversion H; subst; clear H.
  assert (∃ switchm0l ctrlm0l,
              In (OpenFlowLink swId0 switchm0l ctrlm0l) ofLinks0) as X.
  { destruct t.
    unfold SwitchesHaveOpenFlowLinks in swsHaveOFLinks0.
    simpl in swsHaveOFLinks0.
    simpl in Heqdevices0.
    assert (switches0 = switches devices0).
    { rewrite ← Heqdevices0... }
    rewrite ← H in swsHaveOFLinks0.
    apply swsHaveOFLinks0 in XIn.
    destruct XIn as [ofLink [HOFLinkIn HIdEq]].
    clear H.
    destruct ofLink.
    simpl in HIdEq...
    subst... }
  destruct X as [switchm0l [ctrlm0l HOfLink]].
  apply in_split in HOfLink.
  destruct HOfLink as [ofLinks00 [ofLinks01 HOFLink]].
  subst.
  apply Bag.in_split with (Order:=TotalOrder_switch) in XIn.
```

```
destruct XIn as [sws XX].
apply Bag.in_split with (Order:=TotalOrder_fromSwitch) in HIn.
destruct HIn as [switchm0' XX'].
subst.
rename switchm0' into switchm0.
destruct (DrainToController
            (({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|}) <+> sws)
            links0 ofLinks00 swId0 [PacketIn p p0] switchm0l ctrlm0l
            ofLinks01 ctrl0)
  as [sws1 [links1 [ofLinks10 [ofLinks11 [ctrl1 Hstep2]]]]].
destruct (ControllerRecvLiveness sws1 links1 ofLinks10 swId0 nil
                                  (PacketIn p p0)
                                  ctrlm0l ofLinks11 ctrl1)
  as [ctrl2 [Hstep3 [lps' HInCtrl]]].
simpl in HInCtrl.
apply in_split in HInAbst.
destruct HInAbst as [pre [post HInAbst]].
rewrite → HInAbst in HInCtrl.
rewrite → map_app in HInCtrl.
rewrite → Bag.unions_app in HInCtrl.
simpl in HInCtrl.
rewrite → Bag.unions_cons in HInCtrl.
rewrite ← HeqHtopo in HInCtrl.
assert (In (sw,pt,pk) (to_list (relate_controller ctrl2))) as HIn.
{ rewrite ← HInCtrl.
```

```
          apply Bag.in_union.
          left.
          apply Bag.in_union.
          right.
          apply Bag.in_union.
          left.
          simpl... }
        destruct (ControllerLiveness
                    sw pt pk ctrl2 sws1 links1
                    (ofLinks10 ++ (OpenFlowLink swId0 nil ctrlm0l) :: ofLinks11)
                    HIn)
          as [ofLinks20 [ofLinks21 [ctrl3 [swTo [ptTo [switchmLst [ctrlmLst
             [Hstep4 HPktEq]]]]]]]].
        simpl in HPktEq.
        remember (topo (swTo,ptTo)) as X.
        destruct X.
        destruct p1.
        inversion HPktEq. subst. clear HPktEq.
        assert (multistep step (devices t) nil (State sws1 links1 (ofLinks20 ++ (Open-
FlowLink swTo switchmLst (PacketOut ptTo pk :: ctrlmLst)) :: ofLinks21) ctrl3)).
        { eapply multistep_tau.
          rewrite ← Heqdevices0.
          eapply SendToController.
          eapply multistep_app.
          simpl in Hstep2.
          eapply Hstep2.
```

```
  eapply multistep_app.
  eapply Hstep3.
  eapply Hstep4.
  instantiate (1 := nil)...
  reflexivity. }
remember H as H' eqn:X; clear X.
apply simpl_multistep in H'.
destruct H' as [st1 [Hdevices1 HPreStep]].
destruct (@EasyObservePacketOut sw pt swTo ptTo sws1 links1 ofLinks20
                                switchmLst nil pk ctrlmLst
                                ofLinks21 ctrl3) as [stateN stepN]...
{ destruct st1. simpl in *. subst. simpl in *... }
{ destruct st1. simpl in *. subst. simpl in *... }
{ destruct st1. simpl in *. subst. simpl in *... }
{ destruct st1. simpl in *. subst. simpl in *... }
eapply simpl_weak_sim.
eapply multistep_tau.
rewrite ← Heqdevices0.
eapply SendToController.
eapply multistep_app.
simpl in Hstep2.
eapply Hstep2.
eapply multistep_app.
eapply Hstep3.
eapply multistep_app.
eapply Hstep4.
```

```
    exact stepN.
    instantiate (1 := [(sw,pt,pk)])...
    instantiate (1 := [(sw,pt,pk)])...
    reflexivity.
    rewrite ← Heqdevices0.
    rewrite → H1.
    unfold relate.
    autorewrite with bag using simpl.
    reflexivity.
    apply AbstractStep...
    inversion HPktEq.
  + simpl in HInTransfer.
    inversion HInTransfer. }



{ apply Bag.in_unions_map in HMemLink.
  destruct HMemLink as [link [HIn HMemLink]].
  destruct link.
  simpl in HMemLink.
  destruct dst0 as [sw0 pt0].
  apply Bag.in_to_from_list in HMemLink.
  rewrite → in_map_iff in HMemLink.
  destruct HMemLink as [pk0 [HEq HMemLink]].
  inversion HEq. subst. clear HEq.
```

```
apply in_split in HMemLink.
destruct HMemLink as [pks01 [pks02 HMemLink]].
subst.
assert
  (LinkHasDst
     switches0 (DataLink src0 (pks01 ++ pk :: pks02) (sw,pt))) as J0.
{ destruct t.
  simpl in *.
  rewrite ← Heqdevices0 in *.
  apply linksHaveDst0... }
unfold LinkHasDst in J0.
destruct J0 as [switch2 [HSw2In [HSw2IdEq HSw2PtsIn]]].
destruct switch2.
simpl in *.
subst.
apply Bag.in_split with (Order:=TotalOrder_switch) in HSw2In.
destruct HSw2In as [sws HSw2In].
remember (process_packet tbl0 pt pk) as X eqn:Hprocess.
destruct X as [pktOuts pktIns].
apply in_split in HIn.
destruct HIn as [links01 [links02 HIn]].
subst.
eapply simpl_weak_sim.
rewrite ← Heqdevices0.
eapply multistep_app with (obs2 := [(swId0,pt,pk)]).
```

```
  assert ((pks01 ++ [pk]) ++ pks02 = pks01 ++ pk :: pks02) as X.
  { rewrite ← app_assoc... }
  rewrite ← X.
  apply (DrainWire sws swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0
                    links01 src0 (pks01 ++ [pk]) pks02 swId0 pt
                    links02 ofLinks0 ctrl0).
  eapply multistep_tau.
  eapply RecvDataLink.
  eapply multistep_obs.
  eapply PktProcess...
  eapply multistep_nil.
  reflexivity.
  rewrite ← Heqdevices0.
  rewrite → H1.
  reflexivity.
  apply AbstractStep. }



apply Bag.in_unions_map in HMemOFLink.
destruct HMemOFLink as [link0 [HIn HMem]].
destruct link0.
simpl in HMem.
apply Bag.in_union in HMem.
destruct HMem as [HMemCtrlm | HMemSwitchm].
```

```
{ apply in_split in HIn.
  destruct HIn as [ofLinks01 [ofLinks02 HIn]]. subst.
  apply Bag.in_unions_map in HMemCtrlm.
  destruct HMemCtrlm as [msg [MemCtrlm HPk]].
  destruct msg.
  2: solve [ simpl in HPk; inversion HPk ].          2: solve [ simpl in HPk;
inversion HPk ].          simpl in HPk.
  remember (topo (of_to0,p)) as Htopo.
  rename of_to0 into srcSw.
  destruct Htopo.
  2: solve [ simpl in HPk; inversion HPk ].
  destruct p1.
  simpl in HPk.
  destruct HPk.
  2: solve[inversion H].
  inversion H. subst. clear H.
  apply in_split in MemCtrlm.
  destruct MemCtrlm as [lstCtrlm0 [lstCtrlm1 HMemCtrlm]].
  subst.
  destruct (@EasyObservePacketOut sw pt srcSw p switches0 links0 ofLinks01
                                   of_switchm0 lstCtrlm0 pk lstCtrlm1
                                   ofLinks02 ctrl0) as [stateN stepN]...
{ destruct t. simpl in *. rewrite ← Heqdevices0 in *. auto. }
{ destruct t. simpl in *. rewrite ← Heqdevices0 in *. auto. }
{ destruct t. simpl in *. rewrite ← Heqdevices0 in *. auto. }
{ destruct t. simpl in *. rewrite ← Heqdevices0 in *. auto. }
apply simpl_weak_sim with (devs2 := stateN)...
rewrite ← Heqdevices0...
rewrite → H1.
unfold relate.
simpl.
rewrite ← Heqdevices0...
apply AbstractStep. }




{ apply in_split in HIn.
  destruct HIn as [ofLinks00 [ofLinks01 HIn]]. subst.
  apply Bag.in_unions_map in HMemSwitchm.
  destruct HMemSwitchm as [msg [HmsgIn HPk]].
  apply in_split in HmsgIn.
  destruct HmsgIn as [switchm0 [switchm1 HmsgIn]]. subst.
  destruct msg.
  2: solve [ simpl in HPk; inversion HPk ].        destruct (DrainToController
switches0 links0 ofLinks00 of_to0
                                        (switchm0 ++ [PacketIn p p0]) switchm1 of_ctrlm0
                                        ofLinks01 ctrl0)
  as [sws1 [links1 [ofLinks10 [ofLinks11 [ctrl1 Hstep1]]]]].
  match goal with
```

```
    | [ H : multistep step ?s1 nil ?s2 ⊢ _ ] ⇒
    remember s1 as S1; remember s2 as S2
end.
assert (switchm0 ++ PacketIn p p0 :: switchm1 =
        (switchm0 ++ [PacketIn p p0]) ++ switchm1) as X.
rewrite ← app_assoc...
rewrite → X in *. clear X.
rewrite ← HeqS1 in Heqdevices0.
assert (multistep step (devices t) nil S2) as X...
{ rewrite ← Heqdevices0... }
apply simpl_multistep in X.
destruct X as [st2 [Heqdevices2 HconcreteStep1]].
destruct (ControllerRecvLiveness sws1 links1 ofLinks10 of_to0 switchm0
                                  (PacketIn p p0)
                                  of_ctrlm0 ofLinks11 ctrl1)
          as [ctrl2 [Hstep2 [lps' HInCtrl]]].
simpl in HInCtrl.
simpl in HPk.
assert (In (sw,pt,pk) (to_list (relate_controller ctrl2))) as HMem2.
{ rewrite ← HInCtrl.
  rewrite → Bag.in_union. left.
  simpl.
  exact HPk. }
destruct (ControllerLiveness
          sw pt pk ctrl2 sws1 links1
          (ofLinks10 ++ (OpenFlowLink of_to0 switchm0 of_ctrlm0) :: ofLinks11)
        HMem2)
    as [ofLinks20 [ofLinks21 [ctrl3 [swTo [ptTo [switchmLst [ctrlmLst
        [Hstep3 HPktEq]]]]]]]].
simpl in HPktEq.
remember (topo (swTo, ptTo)) as X eqn:Htopo.
destruct X.
destruct p1. inversion HPktEq. subst. clear HPktEq.
destruct (@EasyObservePacketOut sw pt swTo ptTo sws1 links1 ofLinks20
                                switchmLst nil pk ctrlmLst
                                ofLinks21 ctrl3) as [stateN stepN]...
{ destruct st2. simpl in *. subst. simpl in *. auto. }
{ destruct st2. simpl in *. subst. simpl in *. auto. }
{ destruct st2. simpl in *. subst. simpl in *. auto. }
{ destruct st2. simpl in *. subst. simpl in *. auto. }
apply simpl_weak_sim with (devs2 := stateN).
eapply multistep_app with (obs2 := [(sw,pt,pk)]).
apply Hstep1. clear Hstep1.
eapply multistep_app with (obs2 := [(sw,pt,pk)]).
apply Hstep2. clear Hstep2.
eapply multistep_app with (obs2 := [(sw,pt,pk)]).
apply Hstep3.
apply stepN.
reflexivity.
reflexivity.
reflexivity.
rewrite → H1.
```

```
unfold relate.
rewrite → HeqS1.
autorewrite with bag using simpl.
reflexivity.
apply AbstractStep.
simpl in HPktEq.
inversion HPktEq. }




{ simpl in *.
  destruct
    (ControllerLiveness sw pt pk ctrl0 switches0 links0 ofLinks0 HMemCtrl)
    as [ofLinks10 [ofLinks11 [ctrl1 [swTo [ptTo [switchmLst
          [ctrlmLst [Hstep Hrel]]]]]]]].
  simpl in Hrel.
  remember (topo (swTo,ptTo)) as X eqn:Htopo.
  destruct X.
  destruct p.
  inversion Hrel. subst. clear Hrel.
  rename swTo into srcSw. rename ptTo into p.
  destruct (@EasyObservePacketOut sw pt srcSw p switches0 links0 ofLinks10
                                  switchmLst nil pk ctrlmLst
                                  ofLinks11 ctrl1) as [stateN stepN]...
    { destruct t. simpl in *. subst. simpl in *. auto. }
```

```
        { destruct t. simpl in *. subst. simpl in *. auto. }
        { destruct t. simpl in *. subst. simpl in *. auto. }
        { destruct t. simpl in *. subst. simpl in *. auto. }
        apply simpl_weak_sim with (devs2 := stateN)...
        rewrite ← Heqdevices0.
        eapply multistep_app...
        rewrite → H1.
        rewrite ← Heqdevices0.
        unfold relate.
        simpl.
        autorewrite with bag using simpl.
        trivial.
        apply AbstractStep.
        inversion Hrel. }
Qed.
End Make.
```

### A.2.29 FwOFWellFormedness Library

```
Set Implicit Arguments.
Require Import Coq.Lists.List.
Require Import Common.Types.
Require Import Common.Bisimulation.
Require Import Common.AllDiff.
Require Import Bag.Bag2.
Require Import FwOF.FwOFSignatures.

Require FwOF.FwOFWellFormednessLemmas.

Local Open Scope list_scope.

Local Open Scope bag_scope.

Module Make (Import RelationDefinitions : RELATION_DEFINITIONS) <: RELATION.

  Module RelationDefinitions := RelationDefinitions.

  Import AtomsAndController.

  Import Machine.

  Import Atoms.

  Module Import Lemmas := FwOF.FwOFWellFormednessLemmas.Make (RelationDefini-
tions).

  Hint Resolve OfLinksHaveSrc_pres1 OfLinksHaveSrc_pres2 OfLinksHaveSrc_pres3.

  Hint Unfold UniqSwIds.

  Hint Resolve P_entails_FlowTablesSafe step_preserves_P.

  Hint Resolve DevicesFromTopo_pres0 DevicesFromTopo_pres1 SwitchesHaveOpenFlowLinks_pres0
       SwitchesHaveOpenFlowLinks_pres1.

  Hint Resolve LinksHaveSrc_untouched.

  Hint Resolve LinksHaveDst_untouched.

  Hint Resolve FlowTablesSafe_untouched.

  Hint Resolve UniqSwIds_pres.

  Hint Resolve FlowTablesSafe_PacketOut.

  Hint Resolve AllDiff_preservation.

  Hint Resolve NoBarriersInCtrlm_preservation.

  Ltac sauto := solve[eauto with datatypes].

  Lemma simpl_step : ∀ (st1 st2 : state) obs
  (tblsOk1 : FlowTablesSafe (switches st1))
  (linksTopoOk1 : ConsistentDataLinks (links st1))
  (haveSrc1 : LinksHaveSrc (switches st1) (links st1))
  (haveDst1 : LinksHaveDst (switches st1) (links st1))
  (uniqSwIds1 : UniqSwIds (switches st1))
  (P0 : P (switches st1) (ofLinks st1) (ctrl st1))
  (uniqOfLinkIds1 : AllDiff of_to (ofLinks st1))
  (ofLinksHaveSw1 : OFLinksHaveSw (switches st1) (ofLinks st1))
  (devsFromTopo1 : DevicesFromTopo st1)
  (swsHaveOFLinks1 : SwitchesHaveOpenFlowLinks (switches st1) (ofLinks st1))
  (noBarriersInCtrlm1 : NoBarriersInCtrlm (switches st1)),
  step st1 obs st2 →
  ∃ tblsOk2 linksTopoOk2 haveSrc2 haveDst2 uniqSwIds2 P1
        uniqOfLinkIds2 ofLinksHaveSw2 devsFromTopo2 swsHaveOFLinks2
        noBarriersInCtrlm2,
    concreteStep
      (ConcreteState st1 tblsOk1 linksTopoOk1 haveSrc1 haveDst1 uniqSwIds1
                     P0 uniqOfLinkIds1 ofLinksHaveSw1 devsFromTopo1
                     swsHaveOFLinks1 noBarriersInCtrlm1)
      obs
      (ConcreteState st2 tblsOk2 linksTopoOk2 haveSrc2 haveDst2 uniqSwIds2
                     P1 uniqOfLinkIds2 ofLinksHaveSw2 devsFromTopo2
                     swsHaveOFLinks2 noBarriersInCtrlm2).
Proof with simpl;eauto with datatypes.
  intros.
  unfold concreteStep.
```

```
simpl.
{ inversion H; subst; simpl in *.
  + eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    eexists. sauto.
    sauto.
  + eexists.
    unfold FlowTablesSafe.
    intros.
    apply Bag.in_union in H0. simpl in H0.
    { destruct H0 as [[HIn | HContra] | HIn];
      [idtac | solve[inversion HContra] | idtac].
      - inversion HIn; subst; clear HIn.
        assert (FlowModSafe swId1 tbl0 (({|FlowMod fm|}) <+> ctrlm1)) as J.
        { unfold FlowTablesSafe in tblsOk1.
          eapply tblsOk1.
          apply Bag.in_union. left. simpl... }
        inversion J; subst.
      × assert (NotFlowMod (FlowMod fm)) as Hcontra.
        { apply H0. apply Bag.in_union; simpl... }
        inversion Hcontra.
      × assert (FlowMod fm = FlowMod f ∧ ctrlm1 = ctrlm0) as HEq.
        { eapply Bag.singleton_union_disjoint.
          apply Bag.union_from_ordered in H0...
          intros.
          assert (NotFlowMod (FlowMod fm)) as X...
          inversion X. }
        destruct HEq as [HEq HEq0].
        inversion HEq; subst...
        apply NoFlowModsInBuffer...
    - unfold FlowTablesSafe in tblsOk1.
      eapply tblsOk1.
      apply Bag.in_union... }
eexists...
eexists...
eexists...
eexists...
eexists...
eexists...
eexists...
eexists. sauto.
eexists. sauto.
eexists.
{ unfold NoBarriersInCtrlm in *.
```

```
          intros.
          apply Bag.in_union in H0; simpl in H0.
          destruct H0 as [[H0|H0]|H0].
          + refine (noBarriersInCtrlm1 (Switch swId0 pts0 tbl0 inp0 outp0 (({|FlowMod
fm|}) <+> ctrlm0) switchm0) _ _ _).
            - apply Bag.in_union. simpl...
            - apply Bag.in_union.
              rewrite ← H0 in H1.
              simpl in H1.
              right...
          + inversion H0.
          + refine (noBarriersInCtrlm1 sw _ _ _)...
            apply Bag.in_union... }
        sauto.
      + eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists. sauto.
        eexists.
        { unfold NoBarriersInCtrlm in *.
```

```
            intros.
            apply Bag.in_union in H0; simpl in H0.
            destruct H0 as [[H0|H0]|H0].
            + refine (noBarriersInCtrlm1 (Switch swId0 pts0 tbl0 inp0 outp0 (({|Pack-
etOut pt pk|}) <+> ctrlm0) switchm0) _ _ _).
              - apply Bag.in_union. simpl...
              - apply Bag.in_union.
                rewrite ← H0 in H1.
                simpl in H1.
                right...
            + inversion H0.
            + refine (noBarriersInCtrlm1 sw _ _ _)...
              apply Bag.in_union... }
          sauto.
        + eexists. sauto.
          ∃ (LinkTopoOK_inv pks0 (pk::pks0) linksTopoOk1).
          ∃ (LinksHaveSrc_inv pks0 (pk::pks0) (LinksHaveSrc_untouched haveSrc1)).
          ∃ (LinksHaveDst_inv pks0 (pk::pks0) (LinksHaveDst_untouched haveDst1)).
          eexists. sauto.
          eexists. sauto.
          eexists. sauto.
          eexists. sauto.
          eexists. sauto.
          eexists. sauto.
          eexists. sauto.
          sauto.
```

```
+ eexists. sauto.
  ∃ (LinkTopoOK_inv (pks0 ++ [pk]) pks0 linksTopoOk1).
  ∃
    (LinksHaveSrc_inv (pks0 ++ [pk]) pks0 (LinksHaveSrc_untouched haveSrc1)).
  ∃
    (LinksHaveDst_inv (pks0 ++ [pk]) pks0 (LinksHaveDst_untouched haveDst1)).
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
```

```
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { eapply AllDiff_preservation.
    exact uniqOfLinkIds1.
    do 2 rewrite → map_app... }
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { eapply AllDiff_preservation.
    exact uniqOfLinkIds1.
    do 2 rewrite → map_app... }
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { eapply AllDiff_preservation.
    exact uniqOfLinkIds1.
    do 2 rewrite → map_app... }
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { eapply AllDiff_preservation.
    exact uniqOfLinkIds1.
    do 2 rewrite → map_app... }
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  sauto.
+ eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { eapply AllDiff_preservation.
    exact uniqOfLinkIds1.
    do 2 rewrite → map_app... }
  eexists. sauto.
  eexists. sauto.
  eexists. sauto.
  eexists.
  { unfold NoBarriersInCtrlm in *.
    intros.
    apply Bag.in_union in H1; simpl in H1.
    destruct H1 as [[H1|H1]|H1].
```

```
          + rewrite ← H1 in H2.
            simpl in H2.
            apply Bag.in_union in H2. simpl in H2.
            destruct H2 as [[H2|H2]|H2].
            × subst. exact H0.
            × inversion H2.
            × refine (noBarriersInCtrlm1 (Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0
switchm0) _ _ _).
              apply Bag.in_union. simpl...
              simpl...
          + inversion H1.
          + refine (noBarriersInCtrlm1 sw _ _ _)...
            apply Bag.in_union... }
        sauto.
    Grab Existential Variables.
    eauto. eauto. eauto. eauto. eauto. eauto. eauto.
    eauto. eauto. eauto. eauto. eauto. eauto. eauto.
    eauto. eauto. eauto. eauto. eauto. eauto. eauto.
    eauto. eauto. eauto. eauto. eauto. eauto. eauto.
    eauto. eauto. eauto. eauto. eauto. eauto. eauto.
    eauto. eauto.
    }
  Qed.

  Lemma simpl_multistep' : ∀ (st1 st2 : state) obs
    (tblsOk1 : FlowTablesSafe (switches st1))
  (linksTopoOk1 : ConsistentDataLinks (links st1))
  (haveSrc1 : LinksHaveSrc (switches st1) (links st1))
  (haveDst1 : LinksHaveDst (switches st1) (links st1))
  (uniqSwIds1 : UniqSwIds (switches st1))
  (P1 : P (switches st1) (ofLinks st1) (ctrl st1))
  (uniqOfLinkIds1 : AllDiff of_to (ofLinks st1))
  (ofLinksHaveSw1 : OFLinksHaveSw (switches st1) (ofLinks st1))
  (devsFromTopo1 : DevicesFromTopo st1)
  (swsHaveOFLinks1 : SwitchesHaveOpenFlowLinks (switches st1) (ofLinks st1))
  (noBarriersInCtrlm1 : NoBarriersInCtrlm (switches st1)),
  multistep step st1 obs st2 →
  ∃ tblsOk2 linksTopoOk2 haveSrc2 haveDst2 uniqSwIds2 P2
        uniqOfLinkIds2 ofLinksHaveSw2 devsFromTopo2 swsHaveOFLinks2
        noBarriersInCtrlm2,
    multistep concreteStep
              (ConcreteState st1 tblsOk1 linksTopoOk1 haveSrc1 haveDst1
                             uniqSwIds1 P1 uniqOfLinkIds1
                             ofLinksHaveSw1 devsFromTopo1 swsHaveOFLinks1
                             noBarriersInCtrlm1)
              obs
              (ConcreteState st2 tblsOk2 linksTopoOk2 haveSrc2 haveDst2
                             uniqSwIds2 P2 uniqOfLinkIds2
                             ofLinksHaveSw2 devsFromTopo2 swsHaveOFLinks2
                             noBarriersInCtrlm2).
Proof with eauto with datatypes.
  intros.
```

```
induction H.
+ solve [ eauto 13 ].
+ destruct (simpl_step tblsOk1 linksTopoOk1 haveSrc1 haveDst1
                       uniqSwIds1 P1 uniqOfLinkIds1 ofLinksHaveSw1
                       devsFromTopo1 swsHaveOFLinks1
                       noBarriersInCtrlm1
                       H)
    as [tblsOk2 [linksTopoOk2 [haveSrc2 [haveDst2 [uniqSwIds2
       [P2 [uniqOfLinkIds2 [ofLinksHaveSw2
       [devsFromTopo2 [swsHaveOFLinks2
       [noBarriersInCtrlm2 step]]]]]]]]]]].
  destruct (IHmultistep tblsOk2 linksTopoOk2 haveSrc2 haveDst2
                        uniqSwIds2 P2 uniqOfLinkIds2 ofLinksHaveSw2
                        devsFromTopo2 swsHaveOFLinks2 noBarriersInCtrlm2)
    as [tblsOk3 [linksTopoOk3 [haveSrc3 [haveDst3
             [uniqSwIds3 [PN [uniqOfLinkIdsN [ofLinksHaveSwN
       [devsFromTopoN [swsHaveOFLinksN [noBarriersInCtrlmN stepN]]]]]]]]]]].
  solve [ eauto 13 ].
+ destruct (simpl_step tblsOk1 linksTopoOk1 haveSrc1 haveDst1
                       uniqSwIds1 P1 uniqOfLinkIds1 ofLinksHaveSw1
                       devsFromTopo1 swsHaveOFLinks1
                       noBarriersInCtrlm1
                       H)
    as [tblsOk2 [linksTopoOk2 [haveSrc2 [haveDst2 [uniqSwIds2
       [P2 [uniqOfLinkIds2 [ofLinksHaveSw2
       [devsFromTopo2 [swsHaveOFLinks2
```

```
        [noBarriersInCtrlm2 step]]]]]]]]]]].
    destruct (IHmultistep tblsOk2 linksTopoOk2 haveSrc2 haveDst2
                        uniqSwIds2 P2 uniqOfLinkIds2 ofLinksHaveSw2
                        devsFromTopo2 swsHaveOFLinks2 noBarriersInCtrlm2)
     as [tblsOk3 [linksTopoOk3 [haveSrc3 [haveDst3
            [uniqSwIds3 [PN [uniqOfLinkIdsN [ofLinksHaveSwN
        [devsFromTopoN [swsHaveOFLinksN [noBarriersInCtrlmN stepN]]]]]]]]]]].
    solve [ eauto 13 ].
Qed.

Lemma simpl_multistep : ∀ (st1 : concreteState) (devs2 : state) obs,
  multistep step (devices st1) obs devs2 →
  ∃ (st2 : concreteState),
    devices st2 = devs2 ∧
    multistep concreteStep st1 obs st2.
Proof with simpl;auto.
  intros.
  destruct st1.
  destruct (simpl_multistep' concreteState_flowTableSafety0
    concreteState_consistentDataLinks0 linksHaveSrc0 linksHaveDst0
    uniqSwIds0 ctrlP0 uniqOfLinkIds0 ofLinksHaveSw0 devicesFromTopo0
    swsHaveOFLinks0 noBarriersInCtrlm0 H) as [v0 [v1 [v2 [v3 [v4 [v5 [v6 [v7 [v8 [v9
[v10 Hstep]]]]]]]]]]].
  ∃ (ConcreteState devs2 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10)...
Qed.

Lemma relate_step_simpl_tau : ∀ st1 st2,
  concreteStep st1 None st2 →
  relate (devices st1) = relate (devices st2).
Proof with eauto with datatypes.
  intros.
  inversion H; subst.
  idtac "Proving relate_step_simpl_tau (Case 2 of 11)...".
  destruct st1. destruct st2. subst. unfold relate. simpl.
  autorewrite with bag using simpl.
  bag_perm 100.
  idtac "Proving relate_step_simpl_tau (Case 3 of 11)...".
  destruct st1. destruct st2. subst. unfold relate. simpl.
  autorewrite with bag using simpl.
  bag_perm 100.
  idtac "Proving relate_step_simpl_tau (Case 4 of 11)...".
  destruct st1. destruct st2. subst. unfold relate. simpl.
  autorewrite with bag using simpl.
  destruct dst0.
  rewrite → Bag.from_list_cons.
  assert (topo (src (DataLink (swId0, pt) pks0 (s,p))) =
          Some (dst (DataLink (swId0, pt) pks0 (s,p)))) as Jtopo.
  {
    unfold ConsistentDataLinks in concreteState_consistentDataLinks0.
    apply concreteState_consistentDataLinks0.
      simpl in H1.
      rewrite ← H1.
      simpl... }
```

```
simpl in Jtopo.
rewrite → Jtopo.
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 5 of 11)...".
destruct st1. destruct st2. subst. unfold relate. simpl.
autorewrite with bag using simpl.
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 6 of 11)...".
unfold relate.
simpl.
rewrite → (ControllerRemembersPackets H2).
reflexivity.
idtac "Proving relate_step_simpl_tau (Case 7 of 11)...".
unfold relate.
simpl.
repeat rewrite → map_app.
simpl.
repeat rewrite → unions_app.
autorewrite with bag using simpl.
rewrite → (ControllerRecvRemembersPackets H2).
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 8 of 11)...".
unfold relate.
simpl.
repeat rewrite → map_app.
simpl.
```

```
repeat rewrite → unions_app.
autorewrite with bag using simpl.
rewrite → (ControllerSendForgetsPackets H2).
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 9 of 11)...".
unfold relate.
simpl.
repeat rewrite → map_app.
simpl.
repeat rewrite → unions_app.
autorewrite with bag using simpl.
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 10 of 11)...".
unfold relate.
simpl.
repeat rewrite → map_app.
simpl.
repeat rewrite → unions_app.
autorewrite with bag using simpl.
bag_perm 100.
idtac "Proving relate_step_simpl_tau (Case 11 of 11)...".
unfold relate.
simpl.
repeat rewrite → map_app.
simpl.
repeat rewrite → unions_app.
```

```
  autorewrite with bag using simpl.
  bag_perm 100.
Qed.

Lemma relate_multistep_simpl_tau : ∀ st1 st2,
  multistep concreteStep st1 nil st2 →
  relate (devices st1) = relate (devices st2).
Proof with eauto.
  intros.
  remember nil.
  induction H...
  + apply relate_step_simpl_tau in H.
    rewrite → H...
  + inversion Heql.
Qed.

Lemma relate_step_simpl_obs : ∀ sw pt pk lps st1 st2,
  relate (devices st1) = ({| (sw,pt,pk) |} <+> lps) →
  concreteStep st1 (Some (sw,pt,pk)) st2 →
  relate (devices st2) =
    (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps).
Proof with eauto with datatypes.
  intros.
  inversion H0.
  destruct st1.
  destruct st2.
  destruct devices0.
  destruct devices1.
  subst.
  simpl in *.
  inversion H1; subst; clear H1.
  inversion H6; subst; clear H6.
  assert (FlowTableSafe sw tbl0) as Z.
  { assert (FlowModSafe sw tbl0 ctrlm0) as Z.
    { unfold FlowTablesSafe in concreteState_flowTableSafety0.
      eapply concreteState_flowTableSafety0...
      apply Bag.in_union; simpl... }
    unfold FlowTableSafe in Z.
    inversion Z... }
  remember (Z pt pk outp' pksToCtrl H3) as Y eqn:X. clear X Z.
  rewrite ← Y. clear Y.
  unfold relate in *.
  simpl in *.
  autorewrite with bag using simpl.
  apply (Bag.pop_union_r _ ({|(sw,pt,pk)|})).
  repeat rewrite → Bag.union_assoc.
  rewrite → (Bag.union_comm _ lps).
  rewrite ← H.
  simpl.
  autorewrite with bag using simpl.
  bag_perm 100.
Qed.
```

```
Lemma relate_multistep_simpl_obs : ∀ sw pt pk lps st1 st2,
  relate (devices st1) = ({| (sw,pt,pk) |} <+> lps) →
  multistep concreteStep st1 [(sw,pt,pk)] st2 →
  relate (devices st2) =
    (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps).
Proof with eauto.
  intros.
  remember [(sw,pt,pk)] as obs.
  induction H0; subst.
  inversion Heqobs.
  apply IHmultistep...
  apply relate_step_simpl_tau in H0.
  symmetry in H0.
  rewrite → H0...
  destruct obs; inversion Heqobs.
  subst.
  clear Heqobs.
  apply relate_multistep_simpl_tau in H1.
  apply relate_step_simpl_obs with (lps := lps) in H0 .
  rewrite ← H0.
  symmetry...
  trivial.
Qed.

Lemma simpl_weak_sim : ∀ st1 devs2 sw pt pk lps,
  multistep step (devices st1) [(sw,pt,pk)] devs2 →
```

```
    relate (devices st1) = ({| (sw,pt,pk) |} <+> lps) →
    abstractStep
      ({| (sw,pt,pk) |} <+> lps)
      (Some (sw,pt,pk))
      (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps) →
   ∃ st2 : concreteState,
     inverse_relation
       bisim_relation
       (unions (map (transfer sw) (abst_func sw pt pk)) <+> lps)
       st2 ∧
     multistep concreteStep st1 [(sw,pt,pk)] st2.
  Proof with eauto.
    intros.
    destruct (simpl_multistep st1 H) as [st2 [Heq Hmultistep]].
    assert (relate (devices st1) = ({| (sw,pt,pk) |} <+> lps)) as Hrel.
      subst. simpl...
    ∃ st2.
    split.
    unfold inverse_relation.
    unfold bisim_relation.
    symmetry...
    exact (relate_multistep_simpl_obs Hrel Hmultistep).
    trivial.
  Qed.
End Make.
```

### A.2.30 FwOFWellFormednessLemmas Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import Common.Types.

Require Import Common.Bisimulation.

Require Import Common.AllDiff.

Require Import Bag.Bag2.

Require Import FwOF.FwOFSignatures.

Local Open Scope list_scope.

Local Open Scope bag_scope.

Module Make (Import RelationDefinitions : RELATION_DEFINITIONS).

  Import AtomsAndController.

  Import Machine.

  Import Atoms.

  Lemma LinksHaveSrc_untouched : ∀
    {swId tbl pts sws links
    inp outp ctrlm switchm tbl' inp' outp' ctrlm' switchm' },
    LinksHaveSrc
      ({| Switch swId pts tbl inp outp ctrlm switchm |} <+> sws) links →
    LinksHaveSrc
      ({| Switch swId pts tbl' inp' outp' ctrlm' switchm' |} <+> sws)
      links.
  Proof with auto.
    intros.
    unfold LinksHaveSrc in *.
    intros.
    apply H in H0. clear H.
    unfold LinkHasSrc in *.
    destruct H0 as [sw [HMem [HEq HIn]]].
    simpl in HMem.
    rewrite → Bag.in_union in HMem.
    destruct HMem.
    + destruct sw.
      simpl in *.
      destruct H.
      inversion H.
      subst.
      eexists.
      split.
      apply Bag.in_union. left. simpl. left. reflexivity.
      simpl...
      inversion H.
    + ∃ sw.
      split...
      simpl...
      apply Bag.in_union...
  Qed.
  Lemma LinksHaveDst_untouched : ∀
    {swId tbl pts sws links
    inp outp ctrlm switchm tbl' inp' outp' ctrlm' switchm' },
    LinksHaveDst
      ({| Switch swId pts tbl inp outp ctrlm switchm |} <+> sws) links →
    LinksHaveDst
      ({| Switch swId pts tbl' inp' outp' ctrlm' switchm' |} <+> sws)
      links.
  Proof with auto.
    intros.
    unfold LinksHaveDst in *.
    intros.
    apply H in H0. clear H.
    unfold LinkHasDst in *.
    destruct H0 as [sw [HMem [HEq HIn]]].
    simpl in HMem.
    rewrite → Bag.in_union in HMem.
    destruct HMem.
    + destruct sw.
      simpl in *.
      destruct H.
      inversion H.
      subst.
      eexists.
      split.
      apply Bag.in_union. left. simpl. left. reflexivity.
      simpl...
      inversion H.
    + ∃ sw.
      split...
      simpl...
      apply Bag.in_union...
  Qed.

   Lemma LinkTopoOK_inv : ∀ {links links0 src dst} pks pks',
     ConsistentDataLinks (links ++ (DataLink src pks dst) :: links0) →
     ConsistentDataLinks (links ++ (DataLink src pks' dst) :: links0).
   Proof with auto with datatypes.
     intros.
     unfold ConsistentDataLinks in *.
     intros.
     apply in_app_iff in H0.
     simpl in H0.
     destruct H0 as [H0 | [H0 | H0]]...
     pose (lnk' := (DataLink src0 pks0 dst0)).
     remember (H lnk').
     assert (In lnk' (links0 ++ lnk' :: links1))...
     apply e in H1.
     simpl in H1.
     inversion H0.
     simpl...
   Qed.

  Lemma FlowTablesSafe_untouched : ∀ {sws swId pts tbl inp inp'
```

```
  outp outp' ctrlm switchm switchm' },
  FlowTablesSafe
    ({|Switch swId pts tbl inp outp ctrlm switchm|} <+> sws) →
  FlowTablesSafe
    ({|Switch swId pts tbl inp' outp' ctrlm switchm'|} <+> sws).
Proof with eauto.
  intros.
  unfold FlowTablesSafe in *.
  intros.
  simpl in H0.
  apply Bag.in_union in H0; simpl in H0.
  destruct H0 as [[H0 | H0] | H0].
  + inversion H0; subst; clear H0.
    eapply H...
    apply Bag.in_union; simpl...
  + inversion H0.
  + eapply H.
    apply Bag.in_union...
Qed.
Lemma FlowModSafe_PacketOut : ∀ swId tbl pt pk ctrlm,
  FlowModSafe swId tbl (({|PacketOut pt pk|}) <+> ctrlm) →
  FlowModSafe swId tbl ctrlm.
Proof with eauto.
  intros.
  inversion H; subst.
  + apply NoFlowModsInBuffer...
    intros. apply H0. apply Bag.in_union...
  + remember (Bag2Lemmas.union_from_ordered _ _ _ _ _ H0) as J0 eqn:X; clear X.
    clear H0.
    assert (In (FlowMod f) (to_list ctrlm0)) as X.
    { assert (In (FlowMod f) (to_list (({|PacketOut pt pk|}) <+> ctrlm0))).
      { rewrite ← J0.
        rewrite → Bag.in_union; simpl... }
      rewrite → Bag.in_union in H0.
      simpl in H0; destruct H0 as [[H0|H0]|H0]; solve [auto;inversion H0]. }
    apply Bag.in_split with (Order:=TotalOrder_fromController) in X.
    destruct X as [rest Heq].
    subst.
    eapply OneFlowModsInBuffer...
    intros.
    rewrite ← Bag.union_assoc in J0.
    rewrite → (Bag.union_comm _ ({|PacketOut pt pk|})) in J0.
    rewrite → Bag.union_assoc in J0.
    apply Bag.pop_union_l in J0.
    subst.
    apply H1...
    apply Bag.in_union...
Qed.

Lemma FlowTablesSafe_PacketOut : ∀ sws swId pts tbl inp inp'
  outp outp' ctrlm switchm switchm' pt pk,
  FlowTablesSafe
    ({|Switch swId pts tbl inp outp (({|PacketOut pt pk|}) <+> ctrlm) switchm|} <+>
sws) →
  FlowTablesSafe
    ({|Switch swId pts tbl inp' outp' ctrlm switchm'|} <+> sws).
Proof with eauto.
  intros.
  unfold FlowTablesSafe in *.
  intros.
  simpl in H0.
  apply Bag.in_union in H0; simpl in H0.
  destruct H0 as [[H0 | H0] | H0].
  + inversion H0; subst; clear H0.
    assert (FlowModSafe swId1 tbl1 (({|PacketOut pt pk|} <+> ctrlm1))) as X.
    { eapply H...
      apply Bag.in_union; simpl... }
    eapply FlowModSafe_PacketOut...
  + inversion H0.
  + eapply H. apply Bag.in_union. right...
Qed.

Lemma LinksHaveSrc_inv : ∀ {sws links links0 src dst} pks pks',
  LinksHaveSrc sws (links ++ (DataLink src pks dst) :: links0) →
  LinksHaveSrc sws (links ++ (DataLink src pks' dst) :: links0).
Proof with auto with datatypes.
  intros.
  unfold LinksHaveSrc in *.
  intros.
  apply in_app_iff in H0.
  simpl in H0.
  destruct H0 as [H0 | [H0 | H0]]; subst...
  destruct (H (DataLink src0 pks0 dst0))...
  destruct H0 as [HMem [HEq HIn]].
  simpl in *.
  unfold LinkHasSrc.
  ∃ x...
Qed.

Lemma LinksHaveDst_inv : ∀ {sws links links0 src dst} pks pks',
  LinksHaveDst sws (links ++ (DataLink src pks dst) :: links0) →
  LinksHaveDst sws (links ++ (DataLink src pks' dst) :: links0).
Proof with auto with datatypes.
  intros.
  unfold LinksHaveDst in *.
  intros.
  apply in_app_iff in H0.
  simpl in H0.
  destruct H0 as [H0 | [H0 | H0]]; subst...
  destruct (H (DataLink src0 pks0 dst0))...
  destruct H0 as [HMem [HEq HIn]].
  simpl in *.
  unfold LinkHasDst.
  ∃ x...
Qed.

Lemma UniqSwIds_pres : ∀ {sws swId pts tbl inp outp ctrlm switchm
  pts' tbl' inp' outp' ctrlm' switchm'},
  UniqSwIds ({|Switch swId pts tbl inp outp ctrlm switchm|} <+> sws) →
  UniqSwIds ({|Switch swId pts' tbl' inp' outp' ctrlm' switchm'|} <+> sws).
Proof with auto with datatypes.
  intros.
  unfold UniqSwIds in *.
  apply Bag.AllDiff_preservation with (x := Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0
switchm0)...
Qed.

Lemma OfLinksHaveSrc_pres1 : ∀ { sws swId pts1 tbl1 inp1 outp1 ctrlm1
  switchm1 pts2 tbl2 inp2 outp2 ctrlm2 switchm2 switchmLst1 ctrlmLst1
  switchmLst2 ctrlmLst2 ofLinks1 ofLinks2 },
  OFLinksHaveSw
    ({|Switch swId pts1 tbl1 inp1 outp1 ctrlm1 switchm1|} <+> sws)
    (ofLinks1 ++ OpenFlowLink swId switchmLst1 ctrlmLst1 :: ofLinks2) →
  OFLinksHaveSw
    ({|Switch swId pts2 tbl2 inp2 outp2 ctrlm2 switchm2|} <+> sws)
    (ofLinks1 ++ OpenFlowLink swId switchmLst2 ctrlmLst2 :: ofLinks2).
Proof with auto with datatypes.
  intros.
  unfold OFLinksHaveSw in *.
  intros.
```

```
destruct ofLink.
unfold ofLinkHasSw in *.
apply in_app_iff in H0; simpl in H0; destruct H0 as [H0 | [H0 | H0]]; subst.
+ destruct H with (ofLink := OpenFlowLink of_to0 of_switchm0 of_ctrlm0)
    as [sw [HMem HEq]]...
  assert ({ of_to0 = swId0 } + { of_to0 ≠ swId0 }) as J by (apply TotalOrder.eqdec).
  destruct J; subst.
  - ∃ (Switch swId0 pts2 tbl2 inp2 outp2 ctrlm2 switchm2).
    split...
    apply Bag.in_union.
    left.
    simpl...
  - ∃ sw.
    destruct sw.
    simpl in HEq.
    subst.
    split...
    apply Bag.in_union.
    right.
    apply Bag.in_union in HMem.
    destruct HMem.
    × simpl in H1.
      destruct H1.
      inversion H1.
      subst.
      contradiction n...
      inversion H1.
     × trivial.
+ inversion H0; subst; clear H0.
  ∃ (Switch of_to0 pts2 tbl2 inp2 outp2 ctrlm2 switchm2).
  split...
  apply Bag.in_union.
  left.
  simpl...
+ destruct H with (ofLink := OpenFlowLink of_to0 of_switchm0 of_ctrlm0)
    as [sw [HMem HEq]]...
  assert ({ of_to0 = swId0 } + { of_to0 ≠ swId0 }) as J by (apply TotalOrder.eqdec).
  destruct J; subst.
  - ∃ (Switch swId0 pts2 tbl2 inp2 outp2 ctrlm2 switchm2).
    split... apply Bag.in_union. left. simpl...
  - ∃ sw.
    destruct sw.
    simpl in HEq.
    subst.
    split...
    apply Bag.in_union.
    right.
    apply Bag.in_union in HMem.
    destruct HMem...
    simpl in H1.
    destruct H1.
    × inversion H1.
```

```
        subst.
        contradiction n...
      × inversion H1.
Qed.

Lemma OfLinksHaveSrc_pres2 : ∀ { sws swId pts1 tbl1 inp1 outp1 ctrlm1
  switchm1 pts2 tbl2 inp2 outp2 ctrlm2 switchm2 ofLinks },
  OFLinksHaveSw
    ({|Switch swId pts1 tbl1 inp1 outp1 ctrlm1 switchm1|} <+> sws)
    ofLinks →
  OFLinksHaveSw
    ({|Switch swId pts2 tbl2 inp2 outp2 ctrlm2 switchm2|} <+> sws)
    ofLinks.
Proof with auto with datatypes.
  intros.
  unfold OFLinksHaveSw in *.
  intros.
  destruct ofLink.
  apply H in H0.
  clear H.
  unfold ofLinkHasSw in *.
  destruct H0 as [sw [HMem HEq]].
  simpl in HEq.
  destruct sw.
  assert ({ of_to0 = swId0 } + { of_to0 ≠ swId0 }) as J by (apply TotalOrder.eqdec).
  destruct J; subst.
  simpl in *. subst.
  ∃ (Switch swId0 pts2 tbl2 inp2 outp2 ctrlm2 switchm2).
  split...
  + apply Bag.in_union.
    left.
    simpl...
  + ∃ (Switch swId1 pts0 tbl0 inp0 outp0 ctrlm0 switchm0).
    apply Bag.in_union in HMem.
    destruct HMem.
    - simpl in H.
      destruct H.
      × inversion H.
        subst.
        simpl in n.
        contradiction n...
      × inversion H.
    - split...
      apply Bag.in_union...
Qed.

Lemma OfLinksHaveSrc_pres3 : ∀ { sws swId switchmLst1 ctrlmLst1
  switchmLst2 ctrlmLst2 ofLinks1 ofLinks2 },
  OFLinksHaveSw sws
    (ofLinks1 ++ OpenFlowLink swId switchmLst1 ctrlmLst1 :: ofLinks2) →
  OFLinksHaveSw sws
    (ofLinks1 ++ OpenFlowLink swId switchmLst2 ctrlmLst2 :: ofLinks2).
```

```
Proof with auto with datatypes.
  intros.
  unfold OFLinksHaveSw in *.
  intros.
  destruct ofLink.
  apply in_app_iff in H0; simpl in H0; destruct H0 as [H0 | [H0 | H0]]; subst...
  inversion H0; subst; clear H0.
  unfold ofLinkHasSw in *.
  destruct (H (OpenFlowLink of_to0 switchmLst1 ctrlmLst1)) as [sw [HMem HEq]]...
  ∃ sw...
Qed.

Hint Resolve OfLinksHaveSrc_pres1 OfLinksHaveSrc_pres2 OfLinksHaveSrc_pres3.

Hint Unfold UniqSwIds.
Hint Resolve P_entails_FlowTablesSafe.

Section DevicesFromTopo.

  Hint Unfold DevicesFromTopo.

  Lemma DevicesFromTopo_pres0 : ∀ swId0 pts0 tbl0 inp0 outp0 ctrlm0
    switchm0 pts1 tbl1 inp1 outp1 ctrlm1 switchm1 sws links0 ofLinks0
    ctrl0,
    DevicesFromTopo
      (State ({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|} <+> sws)
              links0
              ofLinks0
              ctrl0) →
    DevicesFromTopo
      (State ({|Switch swId0 pts1 tbl1 inp1 outp1 ctrlm1 switchm1|} <+> sws)
              links0
              ofLinks0
              ctrl0).
  Proof with simpl; eauto.
    intros.
    unfold DevicesFromTopo in *.
    intros.
    apply H in H0.
    clear H.
    destruct H0 as [sw0 [sw1 [lnk [HMemSw0 [HMemSw1 [HInLnk [HSw0Eq [HSw1Eq
                    [HLnkSrcEq HLnkDstEq]]]]]]]]].
    destruct lnk.
    simpl in *.
    destruct sw0.
    destruct sw1.
    simpl in *.
    subst.
    apply Bag.in_union in HMemSw0.
    apply Bag.in_union in HMemSw1.
    destruct HMemSw0; destruct HMemSw1.
    + eexists. eexists. eexists.
      simpl in H; simpl in H0.
      destruct H0; destruct H...
      inversion H; inversion H0...
      subst...
    subst...
    split...
    apply Bag.in_union.
    left...
    split...
    apply Bag.in_union. left...
    split...
    inversion H.
    inversion H0.
    inversion H.
  + do 3 eexists.
    simpl in H.
    destruct H. 2: solve[inversion H].
    inversion H. subst.
    repeat split.
    apply Bag.in_union. left. simpl. left. reflexivity.
    apply Bag.in_union. right. exact H0.
    exact HInLnk.
    trivial.
    trivial.
    trivial.
    trivial.
  + do 3 eexists.
    simpl in H0.
    destruct H0. 2: solve[inversion H0].
    inversion H0; subst; clear H0.
```

```
    repeat split...
    apply Bag.in_union. right. exact H.
    apply Bag.in_union. left. simpl. left. reflexivity.
    trivial.
    trivial.
  + do 3 eexists.
    repeat split...
    apply Bag.in_union. right. exact H.
    apply Bag.in_union. right. exact H0.
    trivial.
    trivial.
Qed.

Lemma DevicesFromTopo_pres1 : ∀ sws0 links0 links1 src
  dst pks0 pks1 ofLinks0 ctrl0,
  DevicesFromTopo
    (State sws0
           (links0 ++ DataLink src pks0 dst :: links1)
           ofLinks0
           ctrl0) →
  DevicesFromTopo
    (State sws0
           (links0 ++ DataLink src pks1 dst :: links1)
           ofLinks0
           ctrl0).
Proof with simpl; eauto with datatypes.
```

```
intros.
unfold DevicesFromTopo in *.
intros.
apply H in H0.
clear H.
destruct H0 as [sw0 [sw1 [lnk [HMemSw0 [HMemSw1 [HInLnk [HSw0Eq [HSw1Eq
                [HLnkSrcEq HLnkDstEq]]]]]]]]].
destruct lnk.
simpl in *.
destruct sw0.
destruct sw1.
simpl in *.
subst.
rewrite → in_app_iff in HInLnk. simpl in HInLnk.
destruct HInLnk as [HInLnk | [HInLnk | HInLnk]].
+ repeat eexists.
  exact HMemSw0.
  exact HMemSw1.
  instantiate (1 := DataLink (swId1,pt1) pks2 (swId0,pt0)).
  auto with datatypes.
  simpl...
  simpl...
  simpl...
  simpl...
+ repeat eexists.
  exact HMemSw0.
```

```
      exact HMemSw1.
      instantiate (1 := DataLink (swId1,pt1) pks1 (swId0,pt0)).
      inversion HInLnk. subst.
      auto with datatypes.
      simpl...
      simpl...
      simpl...
      simpl...
    + repeat eexists.
      exact HMemSw0.
      exact HMemSw1.
      instantiate (1 := DataLink (swId1,pt1) pks2 (swId0,pt0)).
      auto with datatypes.
      simpl...
      simpl...
      simpl...
      simpl...
  Qed.
End DevicesFromTopo.
Section SwitchesHaveOpenFlowLinks.
  Hint Unfold SwitchesHaveOpenFlowLinks.
  Lemma SwitchesHaveOpenFlowLinks_pres0 : ∀ swId0 pts0 tbl0 inp0 outp0
    ctrlm0 switchm0 pts1 tbl1 inp1 outp1 ctrlm1 switchm1 sws ofLinks0,
    SwitchesHaveOpenFlowLinks
      ({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|} <+> sws)
    ofLinks0 →
  SwitchesHaveOpenFlowLinks
    ({|Switch swId0 pts1 tbl1 inp1 outp1 ctrlm1 switchm1|} <+> sws)
           ofLinks0.
Proof with simpl; eauto.
  intros.
  unfold SwitchesHaveOpenFlowLinks in *.
  intros.
  simpl in *.
  apply Bag.in_union in H0.
  simpl in H0.
  destruct H0 as [[H0 | H0] | H0].
  + destruct sw.
    inversion H0; subst; clear H0.
    edestruct H as [lnk [HIn HEq]]...
    apply Bag.in_union. simpl. left. left. reflexivity.
    destruct lnk.
    simpl in *.
    subst.
    eexists...
  + inversion H0.
  + edestruct H as [lnk [HIn HEq]]...
    apply Bag.in_union. right...
Qed.
Lemma SwitchesHaveOpenFlowLinks_pres1 : ∀ sws0 ofLinks0
```

```
    ofLinks1 swId switchm0 ctrlm0 switchm1 ctrlm1,
    SwitchesHaveOpenFlowLinks
      sws0
      (ofLinks0 ++ OpenFlowLink swId switchm0 ctrlm0 :: ofLinks1) →
    SwitchesHaveOpenFlowLinks
      sws0
      (ofLinks0 ++ OpenFlowLink swId switchm1 ctrlm1 :: ofLinks1).
    Proof with eauto with datatypes.
      unfold SwitchesHaveOpenFlowLinks.
      intros.
      simpl in *.
      apply H in H0.
      clear H.
      destruct H0 as [ofLink [HIn HIdEq]].
      apply in_app_iff in HIn.
      simpl in HIn.
      destruct HIn as [HIn | [HIn | HIn]].
      + eexists...
      + ∃ (OpenFlowLink swId0 switchm1 ctrlm1).
        split...
        subst.
        simpl in *...
      + eexists...
    Qed.
End SwitchesHaveOpenFlowLinks.
```

```
  Section NoBarriersInCtrlm.

  Lemma NoBarriersInCtrlm_preservation : ∀ swId0 pts0 tbl0 inp0 outp0
      ctrlm0 switchm0 tbl1 inp1 outp1 switchm1 sws,
    NoBarriersInCtrlm ({|Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0|} <+> sws)
→
    NoBarriersInCtrlm ({|Switch swId0 pts0 tbl1 inp1 outp1 ctrlm0 switchm1|} <+>
sws).
  Proof with eauto with datatypes.
    unfold NoBarriersInCtrlm.
    intros.
    apply Bag.in_union in H0; simpl in H0.
    destruct H0 as [[H0|H0]|H0].
    + subst.
      refine (H (Switch swId0 pts0 tbl0 inp0 outp0 ctrlm0 switchm0) _ m _)...
      apply Bag.in_union. simpl...
    + inversion H0.
    + refine (H sw _ m _)...
      apply Bag.in_union...
  Qed.

  End NoBarriersInCtrlm.

End Make.
```

### A.2.31 NetCoreCompiler Library

```
Require Import Coq.Lists.List.
Require Import Coq.Bool.Bool.

Require Import NetCore.NetCoreEval.

Require Import Common.Types.

Require Import Classifier.Classifier.

Require Import Word.WordInterface.

Require Import Pattern.Pattern.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import Network.NetworkPacket.

Set Implicit Arguments.

Import ListNotations.

Fixpoint compile_pred (opt : Classifier bool → Classifier bool)
         (pr : pred) (sw : switchId) : Classifier bool :=
  match pr with
    | PrHdr pat ⇒ [(pat, true)]
    | PrOnSwitch sw' ⇒
      match Word64.eq_dec sw sw' with
        | left _ ⇒ [(Pattern.all, true)]
        | right _ ⇒ []
      end
    | PrOr pr1 pr2 ⇒ opt (union orb (compile_pred opt pr1 sw)
                                    (compile_pred opt pr2 sw))
    | PrAnd pr1 pr2 ⇒ opt (inter andb (compile_pred opt pr1 sw)
                                      (compile_pred opt pr2 sw))
    | PrNot pr' ⇒
      opt (map (second negb)
                (compile_pred opt pr' sw ++ [(Pattern.all, false)]))
    | PrAll ⇒ [(Pattern.all, true)]
    | PrNone ⇒ []
  end.

Definition apply_act (a : list act) (b : bool) :=
  match b with
    | true ⇒ a
    | false ⇒ nil
  end.

Fixpoint compile_pol
  (opt : ∀ (A : Type), Classifier A → Classifier A)
  (p : pol) (sw : switchId) : Classifier (list act) :=
  match p with
    | PoAtom pr act ⇒
      opt _ (map (second (apply_act act))
                 (compile_pred (opt bool) pr sw ++ [(Pattern.all, false)]))
    | PoUnion pol1 pol2 ⇒
      opt _ (union (@app act)
                   (compile_pol opt pol1 sw)
                   (compile_pol opt pol2 sw))
  end.

Fixpoint strip_empty_rules (A : Type) (cf : Classifier A) : Classifier A :=
  match cf with
    | nil ⇒ nil
    | (pat, acts) :: cf ⇒
```

```
      if Pattern.is_empty pat
      then strip_empty_rules cf
      else (pat, acts) :: strip_empty_rules cf
  end.

Definition no_opt (A : Type) := @Datatypes.id (Classifier A).

Definition compile_no_opt := compile_pol no_opt.

Definition compile_opt :=
  compile_pol ((fun A x ⇒ @strip_empty_rules A (@elim_shadowed A x))).
```

### A.2.32 NetCoreController Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Common.Monad.

Require Import Word.WordInterface.

Require Import Network.NetworkPacket.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import Pattern.Pattern.

Require Import Classifier.Classifier.

Require Import NetCore.NetCoreEval.

Require Import NetCore.NetCoreCompiler.

Require Import OpenFlow.ControllerInterface.

Local Open Scope list_scope.

Section Prioritize.
```

```
  Fixpoint prio_rec {A : Type} (prio : Word16.t) (lst : Classifier A) :=
    match lst with
      | nil ⇒ nil
      | (pat, act) :: rest ⇒
         (prio, pat, act) :: (prio_rec (Word16.pred prio) rest)
    end.
  Definition prioritize {A : Type} (lst : Classifier A) :=
    prio_rec Word16.max_value lst.
End Prioritize.
Section PacketIn.
  Definition packetIn_to_in (sw : switchId) (pktIn : packetIn) :=
    InPkt sw (packetInPort pktIn) (packetInPacket pktIn)
      (packetInBufferId pktIn).
End PacketIn.
Section ToFlowMod.
  Definition maybe_openflow0x01_modification {A : Type} (newVal : option A)
                (mkModify : A → OpenFlow0x01Types.action) : actionSequence :=
    match newVal with
      | None ⇒ nil
      | Some v ⇒ [mkModify v]
    end.
  Definition modification_to_openflow0x01 (mods : modification) : actionSequence :=
    match mods with
      | Modification dlSrc dlDst dlVlan dlVlanPcp
```

```
                    nwSrc nwDst nwTos
                    tpSrc tpDst ⇒
      maybe_openflow0x01_modification dlSrc SetDlSrc ++
      maybe_openflow0x01_modification dlDst SetDlDst ++
      maybe_openflow0x01_modification (withVlanNone dlVlan) SetDlVlan ++
      maybe_openflow0x01_modification dlVlanPcp SetDlVlanPcp ++
      maybe_openflow0x01_modification nwSrc SetNwSrc ++
      maybe_openflow0x01_modification nwDst SetNwDst ++
      maybe_openflow0x01_modification nwTos SetNwTos ++
      maybe_openflow0x01_modification tpSrc SetTpSrc ++
      maybe_openflow0x01_modification tpDst SetTpDst
  end.

Definition translate_action (in_port : option portId) (act : act) : actionSequence :=
  match act with
    | Forward mods (PhysicalPort pp) ⇒
      modification_to_openflow0x01 mods ++
      [match in_port with
         | None ⇒ Output (PhysicalPort pp)
         | Some pp' ⇒ match Word16.eq_dec pp' pp with
                        | left _ ⇒ Output InPort
                        | right _ ⇒ Output (PhysicalPort pp)
                      end
       end]
    | Forward mods p ⇒ modification_to_openflow0x01 mods ++ [Output p]
    | ActGetPkt x ⇒ [Output (Controller Word16.max_value)]
  end.

Definition to_flow_mod (prio : priority) (pat : pattern) (act : list act)
              (isfls : Pattern.is_empty pat = false) :=
  let ofMatch := Pattern.to_match isfls in
  FlowMod AddFlow
            ofMatch
            prio
            (concat_map (translate_action (matchInPort ofMatch)) act)
            Word64.zero
            Permanent
            Permanent
            false
            None
            None
            false.

Definition flow_mods_of_classifier lst :=
  List.fold_right
    (fun (ppa : priority × pattern × list act)
          (lst : list flowMod) ⇒
       match ppa with
         | (prio,pat,act) ⇒
           (match (Pattern.is_empty pat) as b
                  return (Pattern.is_empty pat = b → list flowMod) with
              | true ⇒ fun _ ⇒ lst
              | false ⇒ fun H ⇒ (to_flow_mod prio act H) :: lst
```

```
                    end) eq_refl
                end)
            nil
            (prioritize lst).
  Definition delete_all_flows :=
    FlowMod DeleteFlow

                (Pattern.to_match Pattern.all_is_not_empty)
                Word16.zero
                nil
                Word64.zero
                Permanent
                Permanent
                false
                None
                None
                false.
End ToFlowMod.
Record ncstate := State {
  policy : pol;
  switches : list switchId
}.
Module Type NETCORE_MONAD <: CONTROLLER_MONAD.
  Include MONAD.
```

These functions are from CONTROLLER_MONAD, with the *state* parameter specialized to *ncstate*.

```
Definition state := ncstate.
  Parameter get : m state.
  Parameter put : state → m unit.
  Parameter send : switchId → xid → message → m unit.
  Parameter recv : m event.
  Parameter forever : m unit → m unit.
```

These functions are NetCore-specific.

```
Parameter handle_get_packet : id → switchId
→ portId → packet → m unit.
End NETCORE_MONAD.
Module Make (Import Monad : NETCORE_MONAD).
  Local Notation "x <- M ; K" := (bind M (fun x ⇒ K)).
  Fixpoint sequence (lst : list (m unit)) : m unit :=
    match lst with
      | nil ⇒ ret tt
      | cmd :: lst' ⇒
        bind cmd (fun _ ⇒ sequence lst')
    end.
  Definition config_commands (pol: pol) (swId : switchId) :=
    sequence
      (List.map
         (fun fm ⇒ send swId Word32.zero (FlowModMsg fm))
         (delete_all_flows
            :: (flow_mods_of_classifier (compile_opt pol swId)))).
```

```
Definition set_policy (pol : pol) :=
  st ← get;
  let switch_list := switches st in
  _ ← put (State pol switch_list);
  _ ← sequence (List.map (config_commands pol) switch_list);
  ret tt.

Definition handle_switch_disconnected (swId : switchId) :=
  st ← get;
  let switch_list :=
      List.filter
        (fun swId' ⇒ match Word64.eq_dec swId swId' with
                         | left _ ⇒ false
                         | right _ ⇒ true
                       end)
        (switches st) in
  _ ← put (State (policy st) switch_list);
  ret tt.
```

I'm assuming disconnected and connected are interleaved. OCaml should provide that guarantee.

```
Definition handle_switch_connected (swId : switchId) :=
  st ← get;
  _ ← put (State (policy st) (swId :: (switches st)));
  _ ← config_commands (policy st) swId;
  ret tt.

Definition send_output (out : output) :=
  match out with
```

```
      | OutNothing ⇒ ret tt
      | OutPkt swId pp pkt bufOrBytes ⇒
          send swId Word32.zero
                (PacketOutMsg (PacketOut bufOrBytes None [Output pp]))
      | OutGetPkt x switchId portId packet ⇒
        handle_get_packet x switchId portId packet
    end.

  Definition handle_packet_in (swId : switchId) (pk : packetIn) :=
    st ← get;
    let outs := classify (policy st) (packetIn_to_in swId pk) in
    sequence (List.map send_output outs).

  Definition handle_event evt :=
    match evt with
      | SwitchDisconnected swId ⇒ handle_switch_disconnected swId
      | SwitchConnected swId ⇒ handle_switch_connected swId
      | SwitchMessage swId xid (PacketInMsg pktIn) ⇒
        handle_packet_in swId pktIn
      | SwitchMessage swId xid msg ⇒ ret tt
    end.

  Definition main := forever (evt ← recv; handle_event evt).

End Make.
```

### A.2.33 NetCoreEval Library

The module NetCore is defined in OCaml, which is why this is called NetCore semantics.

```
Set Implicit Arguments.

Require Import Coq.Classes.Equivalence.

Require Import Coq.Lists.List.

Require Import Coq.Bool.Bool.

Require Import Common.Utilities.

Require Import Common.Types.

Require Import Word.WordInterface.

Require Import Classifier.Classifier.

Require Import Network.NetworkPacket.

Require Import Pattern.Pattern.

Require Import OpenFlow.OpenFlow0x01Types.

Local Open Scope list_scope.

Inductive id : Type := MkId : nat → id.

Record modification : Type := Modification {
  modifyDlSrc : option dlAddr;
  modifyDlDst : option dlAddr;
  modifyDlVlan : option (option dlVlan);
  modifyDlVlanPcp : option dlVlanPcp;
  modifyNwSrc : option nwAddr;
  modifyNwDst : option nwAddr;
  modifyNwTos : option nwTos;
  modifyTpSrc : option tpPort;
  modifyTpDst : option tpPort
}.
Definition unmodified : modification :=
 Modification None None None None None None None None None.
Inductive act : Type :=
| Forward : modification → pseudoPort → act
| ActGetPkt : id → act.
Inductive pred : Type :=
| PrHdr : pattern → pred
| PrOnSwitch : switchId → pred
| PrOr : pred → pred → pred
| PrAnd : pred → pred → pred
| PrNot : pred → pred
| PrAll : pred
| PrNone : pred.
Inductive pol : Type :=
| PoAtom : pred → list act → pol
| PoUnion : pol → pol → pol.
Inductive input : Type :=
| InPkt : switchId → portId → packet → option bufferId → input.
Inductive output : Type :=
| OutPkt : switchId → pseudoPort → packet → bufferId + bytes → output
| OutGetPkt : id → switchId → portId → packet → output
| OutNothing : output.
Fixpoint match_pred (pr : pred) (sw : switchId) (pt : portId) (pk : packet) :=
  match pr with
    | PrHdr pat ⇒ Pattern.match_packet pt pk pat
    | PrOnSwitch sw' ⇒ match Word64.eq_dec sw sw' with
                                  | left _ ⇒ true
                                  | right _ ⇒ false
                                end
    | PrOr p1 p2 ⇒ orb (match_pred p1 sw pt pk) (match_pred p2 sw pt pk)
    | PrAnd p1 p2 ⇒ andb (match_pred p1 sw pt pk) (match_pred p2 sw pt pk)
    | PrNot p' ⇒ negb (match_pred p' sw pt pk)
    | PrAll ⇒ true
    | PrNone ⇒ false
  end.

Parameter serialize_pkt : packet → bytes.

Extract Constant serialize_pkt ⇒ "Packet_Parser.serialize_packet".

Definition maybe_modify {A : Type} (newVal : option A)
              (modifier : packet → A → packet) (pk : packet) : packet :=
  match newVal with
    | None ⇒ pk
    | Some v ⇒ modifier pk v
  end.

Definition withVlanNone maybeVlan :=
  match maybeVlan with
    | None ⇒ None
    | Some None ⇒ Some VLAN_NONE
    | Some (Some n) ⇒ Some n
  end.

Section Modification.

Local Notation "f $ x" := (f x) (at level 51, right associativity).

Definition modify_pkt (mods : modification) (pk : packet) :=
  match mods with
    | Modification dlSrc dlDst dlVlan dlVlanPcp
                   nwSrc nwDst nwTos
                   tpSrc tpDst ⇒
      maybe_modify dlSrc setDlSrc $
      maybe_modify dlDst setDlDst $
      maybe_modify (withVlanNone dlVlan) setDlVlan $
      maybe_modify dlVlanPcp setDlVlanPcp $
      maybe_modify nwSrc setNwSrc $
      maybe_modify nwDst setNwDst $
      maybe_modify nwTos setNwTos $
      maybe_modify tpSrc setTpSrc $
      maybe_modify tpDst setTpDst pk
  end.

End Modification.

Definition eval_action (inp : input) (act : act) : output :=
  match (act, inp) with
    | (Forward mods pp, InPkt sw _ pk buf) ⇒
      OutPkt sw pp (modify_pkt mods pk)
             (match buf with
                | Some b ⇒ inl b
```

```
                     | None ⇒ inr (serialize_pkt (modify_pkt mods pk))
                   end)
    | (ActGetPkt x, InPkt sw pt pk buf) ⇒ OutGetPkt x sw pt pk
  end.

Fixpoint classify (p : pol) (inp : input) :=
  match p with
    | PoAtom pr actions ⇒
      match inp with
        | InPkt sw pt pk buf ⇒
          if match_pred pr sw pt pk then
            map (eval_action inp) actions
          else nil
      end
    | PoUnion p1 p2 ⇒ classify p1 inp ++ classify p2 inp
  end.
```

### A.2.34 NetCoreTheorems Library

```
Set Implicit Arguments.

Require Import Coq.Classes.Equivalence.
Require Import Coq.Lists.List.
Require Import Coq.Bool.Bool.

Require Import Common.Types.
Require Import Common.CpdtTactics.
Require Import Word.WordInterface.
Require Import Classifier.Classifier.

Require Import Classifier.Theory.

Require Import Network.NetworkPacket.

Require Import Pattern.Pattern.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import NetCore.NetCoreEval.

Require Import NetCore.NetCoreCompiler.

Require Import NetCore.Verifiable.

Local Open Scope list_scope.

Instance bool_as_Action : ClassifierAction bool :=
  {
    zero := false;
    action_eqdec := bool_dec
  }.

Hint Resolve zero.

Definition Equiv_Preserving (f : ∀ A, Classifier A → Classifier A) :=
  ∀ (A : Type) (EA : ClassifierAction A) (pt : portId) (pk : packet) (cf : Classifier A),
    scan zero (f A cf) pt pk = scan zero cf pt pk.

Hint Unfold Equiv_Preserving.

Theorem compile_pred_correct :
  ∀ pr sw pt pk opt,
    Equiv_Preserving opt →
    match_pred pr sw pt pk = scan false (compile_pred (opt bool) pr sw) pt pk.

Proof with auto.
  intros.
```

```
assert (∀ cf pt pk, scan false (opt bool cf) pt pk = scan false cf pt pk) as Heqp.
unfold Equiv_Preserving in H.
intros.
assert (false = zero)...
rewrite → H0.
rewrite → H...
clear H.
induction pr.
simpl.
remember (Pattern.match_packet pt pk p).
destruct b. trivial. trivial.
simpl.
remember (Word64.eq_dec sw s) as b.
destruct b.
simpl.
rewrite → Pattern.all_spec...
simpl...
assert (false = zero) as J...
rewrite → J in *.
simpl.
rewrite → Heqp.
rewrite → union_scan_comm.
rewrite → IHpr1.
rewrite → IHpr2.
trivial.
unfold has_unit.
```

```
rewrite ← J.
split; intros.
destruct a...
destruct a...
{ simpl.
  rewrite → IHpr1.
  rewrite → IHpr2.
  rewrite → Heqp...
  rewrite → inter_comm_bool_range... }


  simpl.
rewrite → Heqp.
rewrite → scan_map_comm with (defA := false)...
remember (scan_inv false pk pt (compile_pred (opt bool) pr sw)) as Inv.
clear HeqInv.
destruct Inv as [[H H0]| H].
rewrite → H0 in IHpr.
rewrite → IHpr.
rewrite → elim_scan_head...
simpl.
rewrite → Pattern.all_spec...
destruct H as [cf2 [cf3 [pat [a [H [H0 [H1 H2]]]]]]].
rewrite → H.
rewrite ← app_assoc.
rewrite ← app_comm_cons.
rewrite → elim_scan_tail...
```

```
  unfold pattern in *.
  rewrite ← H.
  f_equal...
  apply total_tail...
  simpl.
  rewrite → Pattern.all_spec...
  simpl...
Qed.

Lemma A_eqdec : ∀ (a1 a2 : list act), { a1 = a2 } + { a1 ≠ a2 }.
Proof. repeat decide equality. Defined.

Instance A_as_Action : ClassifierAction (list act) :=
  {
    zero := @nil act;
    action_eqdec := A_eqdec
  }.

Lemma compile_pol_correct :
  ∀ opt po sw pt pk bufid,
    Vf_pol po →
    Equiv_Preserving opt →
    classify po (InPkt sw pt pk bufid) =
    map (eval_action (InPkt sw pt pk bufid))
         (scan nil (compile_pol opt po sw) pt pk).
Proof with auto.
  intros.
  rename H into HVfPol.
rename H0 into Heqp.
induction po.
simpl.
assert (∀ cf pt pk, scan nil (opt (list act) cf) pt pk = scan nil cf pt pk) as J0...
intros.
unfold Equiv_Preserving in Heqp.
assert (nil = zero)...
rewrite → H.
rewrite → Heqp...
rewrite → J0.
rewrite → scan_map_comm with (defA := false)...
rewrite → scan_elim_unit_tail.
assert (match_pred p sw pt pk = scan false (compile_pred (opt bool) p sw) pt pk).
apply compile_pred_correct...
rewrite → H.
destruct (scan false (compile_pred (opt bool) p sw) pt pk)...
apply total_tail.
simpl.
assert (nil = zero) as J...
rewrite → J in *.
rewrite → Heqp.
rewrite → union_scan_comm.
rewrite → IHpo1.
rewrite → IHpo2.
rewrite → map_app.
```

```
  trivial.
  inversion HVfPol...
  inversion HVfPol...
  split... intros. rewrite ← J. apply app_nil_r.
Qed.

Local Open Scope equiv_scope.

Lemma Equiv_Preserving_elim_shadowed : Equiv_Preserving (@elim_shadowed).
Proof.
  unfold Equiv_Preserving.
  intros.
  remember (elim_shadowed_ok cf) as H.
  clear HeqH.
  unfold equiv in H.
  unfold Classifier_equiv in H.
  rewrite → H.
  trivial.
Qed.

Lemma Equiv_Preserving_id : Equiv_Preserving no_opt.
Proof.
  unfold Equiv_Preserving.
  intros.
  unfold Datatypes.id.
  reflexivity.
Qed.

Lemma Equiv_Preserving_composes :
  ∀ f g,
    Equiv_Preserving f →
    Equiv_Preserving g →
    Equiv_Preserving (fun A x ⇒ g A (f A x)).
Proof.
  intros.
  unfold Equiv_Preserving in *.
  intros.
  specialize H0 with A EA pt pk (f A cf).
  rewrite H0.
  apply H.
Qed.

Lemma scan_pat_none :
  ∀ A (def : A) cf pt pk a pat,
    Pattern.is_empty pat = true →
    scan def ((pat, a) :: cf) pt pk = scan def cf pt pk.
Proof with auto.
  intros.
  simpl.
  rewrite → Pattern.match_packet_spec.
  rewrite → Pattern.is_empty_true_r...
Qed.

Lemma Equiv_Preserving_strip_empty : Equiv_Preserving strip_empty_rules.
Proof with auto.
  unfold Equiv_Preserving.
  intros.
  induction cf; auto.
  destruct a.
  simpl.
  remember (Pattern.is_empty p) as b.
  destruct b.
  rewrite → Pattern.match_packet_spec.
  rewrite → Pattern.is_empty_true_r...
  simpl.
  destruct (Pattern.match_packet pt pk p)...
Qed.

Lemma compile_no_opt_ok :
  ∀ po sw pt pk bufid,
     Vf_pol po →
     classify po (InPkt sw pt pk bufid) =
     map (eval_action (InPkt sw pt pk bufid))
          (scan nil (compile_no_opt po sw) pt pk).
Proof.
  intros.
  unfold compile_no_opt.
  apply compile_pol_correct.
  trivial.
  apply Equiv_Preserving_id.
Qed.

Lemma compile_opt_ok :
  ∀ po sw pt pk bufid,
    Vf_pol po →
    classify po (InPkt sw pt pk bufid) =
    map (eval_action (InPkt sw pt pk bufid))
        (scan nil (compile_opt po sw) pt pk).
Proof.
  intros.
  unfold compile_no_opt.
  apply compile_pol_correct.
  trivial.
  apply Equiv_Preserving_composes.
  apply Equiv_Preserving_elim_shadowed.
  apply Equiv_Preserving_strip_empty.
Qed.

Definition SemanticsPreserving opt := Equiv_Preserving opt.

Definition netcore_eval pol sw pt pk bufId :=
  classify pol (InPkt sw pt pk bufId).

Definition flowtable_eval ft sw pt pk (bufId : option bufferId) :=
  map (eval_action (InPkt sw pt pk bufId)) (scan nil ft pt pk).

Definition compile := compile_pol.

Definition compose {A B C : Type} (f : B → C) (g : A → B) x := f (g x).

Theorem compile_correct :
  ∀ pol sw pt pk bufId,
    Vf_pol pol →
    netcore_eval pol sw pt pk bufId =
    flowtable_eval (compile_opt pol sw) sw pt pk bufId.
Proof.
  unfold compile.
  unfold SemanticsPreserving.
  unfold netcore_eval.
  unfold flowtable_eval.
  intros.
  apply compile_pol_correct.
  trivial.
  apply Equiv_Preserving_composes.
  apply Equiv_Preserving_elim_shadowed.
  apply Equiv_Preserving_strip_empty.
Qed.
```

### A.2.35 Verifiable Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import NetCore.NetCoreEval.

Inductive Vf_act : act → Prop :=
  | Vf_FwdUnmodifiedPhysicalPort : ∀ pt,
      Vf_act (Forward unmodified (PhysicalPort pt)).

Inductive Vf_pol : pol → Prop :=
  | Vf_PoAtom : ∀ pr acts,
    (∀ act, In act acts → Vf_act act) →
    Vf_pol (PoAtom pr acts)
  | Vf_PoUnion : ∀ pol1 pol2,
    Vf_pol pol1 →
    Vf_pol pol2 →
    Vf_pol (PoUnion pol1 pol2).
```

### A.2.36 NetKAT Library

```
Set Implicit Arguments.

Require Import kat normalisation rewriting kat_tac.

Require Import rel comparisons.

Require Import Packet.

Local Open Scope bool_scope.

Inductive pred : Type :=
| pr_true
| pr_false
| pr_and : pred → pred → pred
| pr_or : pred → pred → pred
| pr_not : pred → pred
| pr_test : hdr → val → pred.

Inductive pol : Type :=
| po_id : pol
| po_drop : pol
| po_sum : pol → pol → pol
| po_seq : pol → pol → pol
| po_star : pol → pol
| po_pred : pred → pol
| po_upd : hdr → val → pol
| po_obs : pol.

Definition test0 h v : dset trace :=
  fun tr ⇒ test h v (head tr).

Fixpoint eval_pred (pr : pred) : dset trace :=
  match pr with
    | pr_and pr1 pr2 ⇒ eval_pred pr1 \cap eval_pred pr2
    | pr_or pr1 pr2 ⇒ eval_pred pr1 \cup eval_pred pr2
    | pr_true ⇒ top
    | pr_false ⇒ bot
    | pr_not pr' ⇒ ! (eval_pred pr')
    | pr_test h v ⇒ test0 h v
  end.

Definition upd0 h v : rel trace trace :=
  fun t1 t2 ⇒ replace_head (upd h v (head t1)) t1 = t2.

Definition obs0 : rel trace trace :=
  fun t1 t2 ⇒ tr_cons (head t1) t1 = t2.

Fixpoint eval_pol (po : pol) : rel trace trace :=
  match po with
    | po_id ⇒ 1
    | po_drop ⇒ 0
    | po_sum e1 e2 ⇒ eval_pol e1 + eval_pol e2
    | po_seq e1 e2 ⇒ eval_pol e1 × eval_pol e2
    | po_star e' ⇒ (eval_pol e')^*
    | po_pred pr ⇒ [eval_pred pr]
    | po_upd h v ⇒ upd0 h v
    | po_obs ⇒ obs0
  end.

Coercion po_pred : pred >-> pol.

Reserved Notation "h ~:= n" (at level 48, no associativity).
Reserved Notation "h ^:= n" (at level 48, no associativity).
Reserved Notation "h =? n" (at level 48, no associativity).
Reserved Notation "x ; y" (at level 50, left associativity).

Module KatNotation.
  Notation "h =? n" := (pr_test h n) : kat_scope.
  Notation "h ~:= n" := (po_upd h n) : kat_scope.
  Notation "x + y" := (po_sum x y) : kat_scope.
  Notation "x ; y" := (po_seq x y) : kat_scope.
  Notation "x ^*" := (po_star x) : kat_scope.
  Notation "#t" := pr_true : kat_scope.
  Notation "#f" := pr_false : kat_scope.
  Notation "x && y" := (pr_and x y) : kat_scope.
  Notation "x || y" := (pr_or x y) : kat_scope.
  Notation "p ~ q" := (eval_pol p == eval_pol q) (at level 80) : kat_scope.
  Notation "~ p" := (pr_not p).

  Definition dup := po_obs.
End KatNotation.
```

```
Module NOTATION.

  Notation "h =? n" := (test0 h n) : netcore_scope.

  Notation "h ~:= n" := (upd0 h n) : netcore_scope.

  Notation "x + y" := (x + y) : netcore_scope.

  Notation "x ; y" := (x × y) : netcore_scope.

  Notation "p ~ q" :=

    (eval_pol p == eval_pol q) (at level 80) : netcore_scope.

  Notation "~ p" := (pr_not p).

  Definition dup := po_obs.

End NOTATION.

Section DomainEquations.

  Variable h h1 h2 : hdr.

  Variable m n : val.

  Import Notation.

  Local Open Scope netcore_scope.

  Hint Unfold rel_dot rel_inj test0 obs0 upd0.

  Lemma upd_compress : (h~:=m) × (h~:=n) == (h~:=n).

  Proof with auto.

    simpl. intros. autounfold. split; intros.

    + destruct H.

      destruct (head a) as [[[sw pt] src] dst].

      subst.

      autorewrite with pkt using simpl...

      rewrite → upd_upd_compress...

    + destruct (head a) as [[[sw pt] src] dst].
      subst. eexists. reflexivity.
      autorewrite with pkt using simpl...
      rewrite → upd_upd_compress...
  Qed.
```

Lemma upd_comm : $h1 \neq h2 \rightarrow$ h1~:=m; h2~:=n == h2~:=n; h1~:=m.

```
Proof with auto.
  simpl. intros. autounfold. split; intros.
  + destruct H0.
    subst.
    destruct (head a) as [[[sw pt] src] dst].
    unfold not in H.
    destruct h1; destruct h2;
    try solve [contradiction H; trivial |
                 destruct a; eexists; simpl; eauto].
  + intros.
    destruct H0.
    subst.
    destruct (head a) as [[[sw pt] src] dst].
    unfold not in H.
    destruct h1; destruct h2;
    try solve [contradiction H; trivial |
                 destruct a; eexists; simpl; eauto].
Qed.
```

Lemma upd_test_compress : h~:=n; [h=?n] == h~:=n.

```
Proof with eauto.
```

```
  simpl. intros. autounfold. split; intros.
  + destruct H. destruct H0. subst.
    trivial.
  + subst. eexists. reflexivity.
    unfold rel_inj.
    split...
    autorewrite with pkt using simpl.
    rewrite → test_upd_true...
Qed.

Lemma upd_test_comm : h1 ≠ h2 → h1~:=m; [h2=?n] == [h2=?n]; h1~:=m.
Proof with eauto.
  simpl. intros. autounfold. split; intros.
  + destruct H0. destruct H1. subst.
    autorewrite with pkt in H2 using (simpl in H2). subst.
    rewrite → test_upd_ignore in H2...
  + destruct H0. destruct H0. subst.
    eexists. reflexivity.
    split...
    autorewrite with pkt using simpl.
    rewrite → test_upd_ignore...
Qed.

Lemma test_test_zero : m ≠ n → [h=?m]; [h=?n] == bot.
Proof with eauto.
  simpl. intros. autounfold. split; intros.
  + destruct H0 as [a1 [H0 H1] [H2 H3]].
```

```
      subst.
      assert (m = n).
      { eapply test_true_diff... }
      subst...
    + inversion H0.
  Qed.

  Lemma upd_test_zero : m ≠ n → h~:=n; [h=?m] == bot.
  Proof with eauto.
    simpl. intros. autounfold. split; intros.
    destruct H0.
    + destruct H1. subst.
      remember (test h m (head a)) as b.
      destruct b.
      - subst.
        autorewrite with pkt in H2 using (simpl in H2)...
        rewrite → test_upd_0 in H2...
        inversion H2.
      - subst.
        autorewrite with pkt in H2 using (simpl in H2)...
        rewrite → test_upd_0 in H2...
        inversion H2.
    + inversion H0.
  Qed.

End DomainEquations.

Ltac kat_simpl :=
  unfold eval_pol; unfold eval_pred; fold eval_pred; fold eval_pol.
```

### A.2.37 Packet Library

```
Set Implicit Arguments.

Require Import Coq.Setoids.Setoid.

Require Import Coq.Arith.EqNat.

Local Open Scope bool_scope.

Definition pk : Set := (nat × nat × nat × nat) %type.

Inductive trace : Set :=

| tr_single : pk → trace

| tr_cons : pk → trace → trace.

Inductive hdr :=

| sw : hdr

| pt : hdr

| src : hdr

| dst : hdr.

Definition val := nat.

Definition upd (h : hdr) (v : val) (p : pk) : pk :=

  match (h, p) with

    | (sw, (_, v2, v3, v4)) ⇒ (v, v2, v3, v4)

    | (pt, (v1, _, v3, v4)) ⇒ (v1, v, v3, v4)

    | (src, (v1, v2, _, v4)) ⇒ (v1, v2, v, v4)

    | (dst, (v1, v2, v3, _)) ⇒ (v1, v2, v3, v)

  end.
```

```
Definition test (h : hdr) (v : val) (p : pk) : bool :=
  match (h, p) with
    | (sw, (v1 , v2 , v3 , v4)) ⇒ beq_nat v v1
    | (pt, (v1 , v2 , v3 , v4)) ⇒ beq_nat v v2
    | (src, (v1 , v2 , v3 , v4)) ⇒ beq_nat v v3
    | (dst, (v1 , v2 , v3 , v4)) ⇒ beq_nat v v4
  end.

Definition head (tr : trace) : pk :=
  match tr with
    | tr_single pk ⇒ pk
    | tr_cons pk _ ⇒ pk
  end.

Definition replace_head (pk : pk) (tr : trace) : trace :=
  match tr with
    | tr_single _ ⇒ tr_single pk
    | tr_cons _ tr' ⇒ tr_cons pk tr'
  end.

Axiom hdr_eqdec : ∀ (h1 h2 : hdr), { h1 = h2 } + { h1 ≠ h2 }.
Axiom val_eqdec : ∀ (v1 v2 : val), { v1 = v2 } + { v1 ≠ v2 }.

Create HintDb pkt.

Lemma head_replace_head : ∀ pk a, head (replace_head pk a) = pk.
Proof with auto.
  intros.
  destruct a...
Qed.
```

```
Lemma replace_head2 :
  ∀ pk1 pk2 a,
    replace_head pk1 (replace_head pk2 a) = replace_head pk1 a.
Proof with auto.
  intros.
  destruct a...
Qed.

Hint Rewrite head_replace_head replace_head2 : pkt.

Lemma test_upd_true : ∀ h n pk, test h n (upd h n pk) = true.
Proof with auto.
  intros.
  destruct pk0 as [[[sw pt] src] dst].
  unfold test.
  unfold upd.
  destruct h; auto; rewrite ← beq_nat_refl...
Qed.

Lemma test_upd_ignore :
  ∀ h1 h2 m n pk,
    h1 ≠ h2 →
    test h2 n (upd h1 m pk) = test h2 n pk.
Proof with auto.
  intros.
  destruct pk0 as [[[sw pt] src] dst].
  unfold not in H.
  destruct h1; destruct h2; try solve[contradiction H; auto];
  unfold upd;
  unfold test...
Qed.

Lemma test_upd_0 :
  ∀ h m n pk,
    m ≠ n →
    test h m (upd h n pk) = false.
Proof with auto.
  intros.
  destruct pk0 as [[[sw pt] src] dst].
  destruct h; unfold upd; unfold test;
  rewrite → beq_nat_false_iff...
Qed.

Lemma test_true_diff :
  ∀ h m n pk,
    true = test h m pk →
    true = test h n pk →
    m = n.
Proof with auto.
  intros.
  destruct pk0 as [[[sw pt] src] dst].
  unfold test in *.
  symmetry in H.
  symmetry in H0.
  destruct h; rewrite → beq_nat_true_iff in *; subst...
```

```
Qed.

Lemma upd_upd_compress :
  ∀ h m n pk,
    upd h m (upd h n pk) = upd h m pk.
Proof with auto.
  intros.
  destruct pk0 as [[[sw pt] src] dst].
  unfold upd.
  destruct h; simpl...
Qed.
```

### A.2.38 Network Library

```
Require Import Common.Utilities.
Require Import Coq.Lists.List.
Require Import Classes.EquivDec.
Require Import OpenFlow.Types.
Section Network.
  Definition host := nat.


  Inductive link :=
  | Link : Switch → Port → link.
  Definition host_map := host → link.
  Definition graph := list (link × link).
  Definition topology := (host_map, graph).
  Definition path := list (link).

  Definition graph_search := link → link → graph → option path.

  Parameter G : graph.

  Parameter H : host_map.

  Parameter H_inj : ∀ H1 H2, H H1 = H H2 → H1 = H2.

  Parameter H_unique_ports : ∀ sw p h,

    ¬ In (H h, Link sw p) G ∧ ¬ In (Link sw p, H h) G.

  Lemma eq_host_dec : ∀ h1 h2 : host, {h1=h2} + {h1≠h2}.

  Proof.

    repeat decide equality.

  Qed.

  Lemma eq_link_dec : ∀ l1 l2 : link, {l1=l2} + {l1≠l2}.

  Proof.

    repeat decide equality.

  Qed.

  Program Instance link_eq_eqdec : EqDec link eq := eq_link_dec.

  Program Instance host_eq_eqdec : EqDec host eq := eq_host_dec.

  Inductive Legal_path : path → link → link → graph → Prop :=

  | single_path_legal : ∀ s port1 port2 g,

    Legal_path [(Link s port1) ; (Link s port2)] (Link s port1) (Link s port2) g

  | trans_path_legal : ∀ a b c d g p p',

    In (b, c) g →

    Legal_path p a b g →

    Legal_path p' c d g →

    Legal_path (p ++ p') a d g.
  Lemma Legal_path_non_empty :
    ∀ p n n' g,
      Legal_path p n n' g → p ≠ [].
  Proof.
    red in ⊢ ×.
    intros.
    induction H0; util_crush; apply app_eq_nil in H1; intuition.
  Qed.
  Definition reachable host1 host2 g := ∃ p, Legal_path p (H host1) (H host2) g.
  Inductive loop_free_path : path → Prop :=
  | empty_loop_free_path : loop_free_path []
  | unique_loop_free_path : ∀ (n : link) (p : path),
    loop_free_path p →
    not (In n p) →
    loop_free_path (n :: p).
End Network.
```

### A.2.39 NetworkPacket Library

```
Set Implicit Arguments.
Require Import Coq.Structures.Equalities.
Require Import NArith.BinNat.
Require Import Common.Types.
Require Import Word.WordInterface.
Local Open Scope list_scope.
Local Open Scope N_scope.

Parameter bytes : Type.

Extract Constant bytes ⇒ "Cstruct.t".

Section Constants.

  Definition Const_0x800 := @Word16.Mk 2048 eq_refl.

  Definition Const_0x806 := @Word16.Mk 2054 eq_refl.

  Definition Const_0x6 := @Word8.Mk 6 eq_refl.

  Definition Const_0x7 := @Word8.Mk 7 eq_refl.

  Definition Const_0x1 := @Word8.Mk 1 eq_refl.

End Constants.

Extract Constant Const_0x800 ⇒ "0x800".

Extract Constant Const_0x806 ⇒ "0x806".

Extract Constant Const_0x6 ⇒ "0x6".

Extract Constant Const_0x7 ⇒ "0x7".

Extract Constant Const_0x1 ⇒ "0x1".

Definition portId := Word16.t.

Definition dlAddr := Word48.t.

Definition dlTyp := Word16.t.

Definition dlVlan := Word16.t.

Definition dlVlanPcp := Word8.t. Definition nwAddr := Word32.t.

Definition nwProto := Word8.t.

Definition nwTos := Word8.t.  6 bits  Definition tpPort := Word16.t.

Unset Elimination Schemes.

Record tcp : Type := Tcp {
  tcpSrc : tpPort;
  tcpDst : tpPort;
  tcpSeq : Word32.t;
  tcpAck : Word32.t;
  tcpOffset : Word8.t;
  tcpFlags : Word16.t; nine lower bits      tcpWindow : Word16.t;
  tcpChksum : Word8.t;
  tcpUrgent : Word8.t;
  tcpPayload : bytes
}.
Record icmp : Type := Icmp {
  icmpType : Word8.t;
  icmpCode : Word8.t;
  icmpChksum : Word16.t;
  icmpPayload : bytes
}.
Inductive tpPkt : Type :=
  | TpTCP : tcp → tpPkt
  | TpICMP : icmp → tpPkt
  | TpUnparsable : nwProto → bytes → tpPkt.
Record ip : Type := IP {
  pktIPVhl : Word8.t;
  pktIPTos : nwTos;
  pktIPLen : Word16.t;
  pktIPIdent : Word16.t;
  pktIPFlags : Word8.t;
  pktIPFrag : Word16.t;  13 bits    pktIPTtl : Word8.t;
  pktIPProto : nwProto;
  pktIPChksum : Word16.t;
  pktIPSrc : nwAddr;
  pktIPDst : nwAddr;
  pktTpHeader : tpPkt
}.
Inductive arp : Type :=
  | ARPQuery : dlAddr → nwAddr → nwAddr → arp
  | ARPReply : dlAddr → nwAddr → dlAddr → nwAddr → arp.
Inductive nw : Type :=
  | NwIP : ip → nw
  | NwARP : arp → nw
  | NwUnparsable : dlTyp → bytes → nw.
Record packet : Type := Packet {
  pktDlSrc : dlAddr;
  pktDlDst : dlAddr;
  pktDlTyp : dlTyp;
  pktDlVlan : dlVlan;
  pktDlVlanPcp : dlVlanPcp;
  pktNwHeader : nw
}.
Section Accessors.
```

These accessors return zero if a field does not exist.

```
Definition pktNwSrc pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒ pktIPSrc ip
        | NwARP (ARPQuery _ ip _) ⇒ ip
        | NwARP (ARPReply _ ip _ _) ⇒ ip
        | NwUnparsable _ _ ⇒ Word32.zero
      end
  end.
Definition pktNwDst pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒ pktIPDst ip
        | NwARP (ARPQuery _ _ ip) ⇒ ip
        | NwARP (ARPReply _ _ _ ip) ⇒ ip
        | NwUnparsable _ _ ⇒ Word32.zero
      end
  end.
Definition pktNwProto pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒ pktIPProto ip
        | NwARP (ARPQuery _ _ _) ⇒ Word8.zero
        | NwARP (ARPReply _ _ _ _) ⇒ Word8.zero
        | NwUnparsable _ _ ⇒ Word8.zero
      end
  end.

Definition pktNwTos pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒ pktIPTos ip
        | NwARP (ARPQuery _ _ _) ⇒ Word8.zero
        | NwARP (ARPReply _ _ _ _) ⇒ Word8.zero
        | NwUnparsable _ _ ⇒ Word8.zero
      end
  end.

Definition pktTpSrc pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒
          match pktTpHeader ip with
            | TpTCP frag ⇒ tcpSrc frag
            | TpICMP _ ⇒ Word16.zero
            | TpUnparsable _ _ ⇒ Word16.zero
          end
        | NwARP (ARPQuery _ _ _) ⇒ Word16.zero
        | NwARP (ARPReply _ _ _ _) ⇒ Word16.zero
        | NwUnparsable _ _ ⇒ Word16.zero
      end
  end.
Definition pktTpDst pk :=
  match pk with
    | {| pktNwHeader := hdr |} ⇒
      match hdr with
        | NwIP ip ⇒
          match pktTpHeader ip with
            | TpTCP frag ⇒ tcpDst frag
            | TpICMP _ ⇒ Word16.zero
            | TpUnparsable _ _ ⇒ Word16.zero
          end
        | NwARP (ARPQuery _ _ _) ⇒ Word16.zero
        | NwARP (ARPReply _ _ _ _) ⇒ Word16.zero
        | NwUnparsable _ _ ⇒ Word16.zero
      end
  end.
End Accessors.
Section Setters.
```

These fail silently if the field does not exist.

```
Definition setDlSrc pk dlSrc :=
  match pk with
    | Packet _ dlDst dlTyp dlVlan dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
  end.

Definition setDlDst pk dlDst :=
  match pk with
    | Packet dlSrc _ dlTyp dlVlan dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
  end.

Definition setDlVlan pk dlVlan :=
  match pk with
    | Packet dlSrc dlDst dlTyp _ dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
  end.

Definition setDlVlanPcp pk dlVlanPcp :=
  match pk with
    | Packet dlSrc dlDst dlTyp dlVlan _ nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
  end.

Definition nw_setNwSrc (typ : dlTyp) (nwPkt : nw) src : nw :=
  match nwPkt with
    | NwIP (IP vhl tos len ident flags frag ttl proto chksum _ dst tp) ⇒
      NwIP (@IP vhl tos len ident flags frag ttl proto chksum src dst tp)
    | NwARP arp ⇒ NwARP arp
    | NwUnparsable typ b ⇒ NwUnparsable typ b
  end.

Definition nw_setNwDst (typ : dlTyp)(nwPkt : nw) dst : nw :=
  match nwPkt with
    | NwIP (IP vhl tos len ident flags frag ttl proto chksum src _ tp) ⇒
      NwIP (@IP vhl tos len ident flags frag ttl proto chksum src dst tp)
    | NwARP arp ⇒ NwARP arp
    | NwUnparsable typ b ⇒ NwUnparsable typ b
  end.

Definition nw_setNwTos (typ : dlTyp) (nwPkt : nw) tos :=
  match nwPkt with
    | NwIP (IP vhl _ len ident flags frag ttl proto chksum src dst tp) ⇒
      NwIP (@IP vhl tos len ident flags frag ttl proto chksum src dst tp)
    | NwARP arp ⇒ NwARP arp
    | NwUnparsable typ b ⇒ NwUnparsable typ b
  end.

Definition setNwSrc pk nwSrc :=
  match pk with
    | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (nw_setNwSrc dlTyp nw nwSrc)
  end.

Definition setNwDst pk nwDst :=
  match pk with
    | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (nw_setNwDst dlTyp nw nwDst)
  end.
```

```
Definition setNwTos pk nwTos :=
  match pk with
    | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
      @Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (nw_setNwTos dlTyp nw nwTos)
  end.

Definition tp_setTpSrc tp src :=
  match tp with
    | TpTCP (Tcp _ dst seq ack off flags win chksum urgent payload) ⇒
      TpTCP (Tcp src dst seq ack off flags win chksum urgent payload)
    | TpICMP icmp ⇒ TpICMP icmp
    | TpUnparsable proto payload ⇒ TpUnparsable proto payload
  end.

Definition tp_setTpDst tp dst :=
  match tp with
    | TpTCP (Tcp src _ seq ack off flags win chksum urgent payload) ⇒
      TpTCP (Tcp src dst seq ack off flags win chksum urgent payload)
    | TpICMP icmp ⇒ TpICMP icmp
    | TpUnparsable proto payload ⇒ TpUnparsable proto payload
  end.

Definition nw_setTpSrc nwPkt tpSrc :=
  match nwPkt with
    | NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp) ⇒
      NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst
                 (tp_setTpSrc tp tpSrc))
    | NwARP arp ⇒ NwARP arp
```

```
      | NwUnparsable typ b ⇒ NwUnparsable typ b
    end.
  Definition nw_setTpDst nwPkt tpDst :=
    match nwPkt with
      | NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst
                  (tp_setTpDst tp tpDst))
      | NwARP arp ⇒ NwARP arp
      | NwUnparsable typ b ⇒ NwUnparsable typ b
    end.
  Definition setTpSrc pk tpSrc :=
    match pk with
      | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (nw_setTpSrc nw tpSrc)
    end.
  Definition setTpDst pk nwDst :=
    match pk with
      | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (nw_setTpDst nw nwDst)
    end.
End Setters.
```

### A.2.40 PacketTotalOrder Library

```
Set Implicit Arguments.
```

```
Require Import Coq.Structures.Equalities.
Require Import NArith.BinNat.
Require Import Bag.TotalOrder.
Require Import Word.WordTheory.
Require Import Network.NetworkPacket.

Local Open Scope list_scope.
Local Open Scope N_scope.

Parameter bytes_le : bytes → bytes → Prop.
Parameter Instance TotalOrder_bytes : TotalOrder bytes_le.

Definition proj_tcp r :=
  match r with
    | Tcp src dst seq ack off flags win chk urg payload ⇒
      (src,dst,seq,ack,off,flags,win,chk,urg,payload)
  end.

Definition inj_tcp tup :=
  match tup with
    | (src,dst,seq,ack,off,flags,win,chk,urg,payload) ⇒
      Tcp src dst seq ack off flags win chk urg payload
  end.

Local Notation "x ** y" := (PairOrdering x y) (at level 71, left associativity).
Local Notation "x +++ y" := (SumOrdering x y) (at level 70, right associativity).

Definition tcp_le :=
  (Word16.le ** Word16.le ** Word32.le ** Word32.le ** Word8.le **
     Word16.le ** Word16.le ** Word8.le ** Word8.le ** bytes_le).

Hint Resolve TotalOrder_sum TotalOrder_pair TotalOrder_bytes
```

```
  Word16.TotalOrder Word32.TotalOrder Word8.TotalOrder Word48.TotalOrder.
Instance TotalOrder_tcp : TotalOrder (ProjectOrdering proj_tcp tcp_le).
Proof.
  apply TotalOrder_Project with (g:=inj_tcp).
  + unfold tcp_le. auto 20.
  + unfold inverse. destruct x; auto.
Qed.
Definition proj_icmp r :=
  match r with
    | Icmp typ code chksum payload ⇒ (typ, code, chksum, payload)
  end.
Definition inj_icmp tup :=
  match tup with
    | (typ, code, chksum, payload) ⇒ Icmp typ code chksum payload
  end.
Definition icmp_le := Word8.le ** Word8.le ** Word16.le ** bytes_le.
Instance TotalOrder_icmp : TotalOrder (ProjectOrdering proj_icmp icmp_le).
Proof.
  apply TotalOrder_Project with (g:=inj_icmp).
  + unfold icmp_le. auto 20.
  + unfold inverse. destruct x; auto.
Qed.
Definition proj_tpPkt r :=
  match r with
    | TpTCP tcp ⇒ inl (proj_tcp tcp)
    | TpICMP icmp ⇒ inr (inl (proj_icmp icmp))
    | TpUnparsable proto bytes ⇒ inr (inr (proto,bytes))
  end.

Definition inj_tpPkt tup :=
  match tup with
    | inl tcp ⇒ TpTCP (inj_tcp tcp)
    | inr (inl icmp) ⇒ TpICMP (inj_icmp icmp)
    | inr (inr (proto,bytes)) ⇒ TpUnparsable proto bytes
  end.

Definition tpPkt_le := tcp_le +++ icmp_le +++ (Word8.le ** bytes_le).

Instance TotalOrder_tpPkt : TotalOrder (ProjectOrdering proj_tpPkt tpPkt_le).
Proof.
  apply TotalOrder_Project with (g:=inj_tpPkt).
  + unfold tpPkt_le. unfold tcp_le. unfold icmp_le.
    auto 20.
  + unfold inverse.
    destruct x; auto.
    destruct t; auto.
    destruct i; auto.
Qed.

Definition proj_ip r :=
  match r with
    | IP vhl tos len ident flags frag ttl proto chksum src dst tp ⇒
        (vhl, tos, len, ident, flags, frag, ttl, proto, chksum, src, dst, proj_tpPkt tp)
  end.
```

```
Definition inj_ip tup :=
  match tup with
    | (vhl, tos, len, ident, flags, frag, ttl, proto, chksum, src, dst, tp) ⇒
      IP vhl tos len ident flags frag ttl proto chksum src dst (inj_tpPkt tp)
  end.

Definition ip_le :=
  Word8.le ** Word8.le ** Word16.le ** Word16.le ** Word8.le ** Word16.le ** Word8.le
** Word8.le **
  Word16.le ** Word32.le ** Word32.le ** tpPkt_le.

Lemma inverse_ip : inverse proj_ip inj_ip.
Proof with auto.
  unfold inverse.
    destruct x; auto.
    simpl.
    destruct pktTpHeader; auto.      destruct t; auto.
    destruct i; auto.
Qed.

Instance TotalOrder_ip : TotalOrder (ProjectOrdering proj_ip ip_le).
Proof.
  apply TotalOrder_Project with (g:=inj_ip).
  + unfold ip_le. unfold tpPkt_le. unfold tcp_le. unfold icmp_le.
    auto 20.
  + exact inverse_ip.
Qed.

Definition proj_arp x :=
  match x with
    | ARPQuery x y z ⇒ inl (x,y,z)
    | ARPReply w x y z ⇒ inr (w,x,y,z)
  end.

Definition inj_arp x :=
  match x with
    | inl (x,y,z) ⇒ ARPQuery x y z
    | inr (w,x,y,z) ⇒ ARPReply w x y z
  end.

Definition arp_le := (Word48.le ** Word32.le ** Word32.le) +++ (Word48.le ** Word32.le
** Word48.le ** Word32.le).

Lemma inverse_arp : inverse proj_arp inj_arp.
Proof. unfold inverse. destruct x; auto. Qed.

Instance TotalOrder_arp : TotalOrder (ProjectOrdering proj_arp arp_le).
Proof.
  apply TotalOrder_Project with (g:=inj_arp).
  + unfold arp_le. auto 20.
  + exact inverse_arp.
Qed.

Definition proj_nw x :=
  match x with
    | NwIP ip ⇒ inl ip
    | NwARP arp ⇒ inr (inl arp)
    | NwUnparsable typ bytes ⇒ inr (inr (typ, bytes))
  end.
```

```
Definition inj_nw x :=
  match x with
    | inl ip ⇒ NwIP ip
    | inr (inl arp) ⇒ NwARP arp
    | inr (inr (typ, bytes)) ⇒ NwUnparsable typ bytes
  end.

Definition nw_le := ProjectOrdering proj_ip ip_le +++ (ProjectOrdering proj_arp arp_le
+++ (Word16.le ** bytes_le)).

Definition inverse_nw : inverse proj_nw inj_nw.
Proof. unfold inverse. destruct x; auto. Qed.

Instance TotalOrder_nw : TotalOrder (ProjectOrdering proj_nw nw_le).
Proof.
  apply TotalOrder_Project with (g:=inj_nw).
  + unfold nw_le. repeat apply TotalOrder_sum. apply TotalOrder_ip. apply To-
talOrder_arp. auto.
  + exact inverse_nw.
Qed.

Definition proj_packet x :=
  match x with
    | Packet src dst typ vlan pcp nw ⇒
       (src,dst,typ,vlan,pcp,nw)
  end.

Definition inj_packet x :=
  match x with
    | (src,dst,typ,vlan,pcp,nw) ⇒
    Packet src dst typ vlan pcp nw
  end.

Lemma inverse_packet : inverse proj_packet inj_packet.

Proof. unfold inverse. destruct x; auto. Qed.

Definition packet_le := ProjectOrdering proj_packet (Word48.le ** Word48.le ** Word16.le
** Word16.le ** Word8.le ** (ProjectOrdering proj_nw nw_le)).

Instance TotalOrder_packet : TotalOrder packet_le.

Proof.
  apply TotalOrder_Project with (g:=inj_packet).
  + unfold packet_le. repeat apply TotalOrder_pair; auto. apply TotalOrder_nw.
  + exact inverse_packet.
Qed.
```

### A.2.41 ControllerInterface Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Common.Monad.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import Network.NetworkPacket.

Inductive event : Type :=
  | SwitchConnected : switchId → event
  | SwitchDisconnected : switchId → event
  | SwitchMessage : switchId → xid → message → event.
```

Using a thin trusted shim, written in Haskell, we drive verified controllers that match this signature. `Module Type` *CONTROLLER_MONAD* <: *MONAD*.

```
  Include MONAD.

  Parameter state : Type.

  Parameter get : m state.

  Parameter put : state → m unit.

  Parameter send : switchId → xid → message → m unit.

  Parameter recv : m event.
```

Must be defined in OCaml `Parameter` *forever* : *m unit* → *m unit*.

```
End CONTROLLER_MONAD.
```

### A.2.42 FlowTable Library

```
Set Implicit Arguments.

Require Import Coq.Lists.List.

Require Import OpenFlow.OpenFlow0x01Types.

Require Import Network.NetworkPacket.

Require Import Word.WordInterface.

Import ListNotations.

Open Scope list_scope.

Record flowTableRule := Rule {
  priority : Word16.t;
  pattern : of_match;
  actions : actionSequence
}.

Definition flowTable := list flowTableRule.
```

### A.2.43 OpenFlow0x01Types Library

```
Set Implicit Arguments.

Require Import Coq.Structures.Equalities.

Require Import Word.WordInterface.

Require Import Network.NetworkPacket.

Definition VLAN_NONE : dlVlan := @Word16.Mk 65535 eq_refl.

Extract Constant VLAN_NONE ⇒ "65535".

Record of_match : Type := Match {
  matchDlSrc : option dlAddr;
  matchDlDst : option dlAddr;
  matchDlTyp : option dlTyp;
  matchDlVlan : option dlVlan;
  matchDlVlanPcp : option dlVlanPcp;
  matchNwSrc : option nwAddr;
  matchNwDst : option nwAddr;
  matchNwProto : option nwProto;
  matchNwTos : option nwTos;
  matchTpSrc : option tpPort;
  matchTpDst : option tpPort;
  matchInPort : option portId
}.
```

```
Record capabilities : Type := Capabilities {
  flow_stats: bool;
  table_stats: bool;
  port_stats: bool;
  stp: bool;
  ip_reasm: bool;
  queue_stats: bool;
  arp_match_ip: bool
}.
Record actions : Type := Actions {
  output: bool;
  set_vlan_id: bool;
  set_vlan_pcp: bool;
  strip_vlan: bool;
  set_dl_src: bool;
  set_dl_dst: bool;
  set_nw_src: bool;
  set_nw_dst: bool;
  set_nw_tos: bool;
  set_tp_src: bool;
  set_tp_dst: bool;
  enqueue: bool;
  vendor: bool
}.
Record features : Type := Features {
  switch_id : Word64.t;
  num_buffers: Word32.t;
  num_tables: Word8.t;
  supported_capabilities: capabilities;
  supported_actions: actions

}.
Inductive flowModCommand : Type :=
| AddFlow : flowModCommand
| ModFlow : flowModCommand
| ModStrictFlow : flowModCommand
| DeleteFlow : flowModCommand
| DeleteStrictFlow : flowModCommand.
Definition switchId := Word64.t.
Definition priority := Word16.t.
Definition bufferId := Word32.t.
Inductive pseudoPort : Type :=
| PhysicalPort : portId → pseudoPort
| InPort : pseudoPort
| Flood : pseudoPort
| AllPorts : pseudoPort
| Controller : Word16.t → pseudoPort.
Inductive action : Type :=
| Output : pseudoPort → action
| SetDlVlan : dlVlan → action
| SetDlVlanPcp : dlVlanPcp → action
| StripVlan : action
| SetDlSrc : dlAddr → action
| SetDlDst : dlAddr → action
| SetNwSrc : nwAddr → action
| SetNwDst : nwAddr → action
| SetNwTos : nwTos → action
| SetTpSrc : tpPort → action
| SetTpDst : tpPort → action.

Definition actionSequence := list action.

Inductive timeout : Type :=
| Permanent : timeout
| ExpiresAfter : ∀ (n : Word16.t),
    Word16.to_nat n > Word16.to_nat Word16.zero →
    timeout.

Record flowMod := FlowMod {
  mfModCmd : flowModCommand;
  mfMatch : of_match;
  mfPriority : priority;
  mfActions : actionSequence;
  mfCookie : Word64.t;
  mfIdleTimeOut : timeout;
  mfHardTimeOut : timeout;
  mfNotifyWhenRemoved : bool;
  mfApplyToPacket : option bufferId;
  mfOutPort : option pseudoPort;
  mfCheckOverlap : bool
}.

Inductive packetInReason : Type :=
| NoMatch : packetInReason
| ExplicitSend : packetInReason.

Record packetIn : Type := PacketIn {
  packetInBufferId : option bufferId;
  packetInTotalLen : Word16.t;
  packetInPort : portId;
  packetInReason_ : packetInReason;
  packetInPacket : packet
}.

Definition xid : Type := Word32.t.

Inductive packetOut : Type := PacketOut {
  pktOutBufOrBytes : bufferId + bytes;
  pktOutPortId : option portId;
  pktOutActions : actionSequence
}.

Inductive message : Type :=
| Hello : bytes → message
| EchoRequest : bytes → message
| EchoReply : bytes → message
| FeaturesRequest : message
| FeaturesReply : features → message
| FlowModMsg : flowMod → message
| PacketInMsg : packetIn → message
| PacketOutMsg : packetOut → message
| BarrierRequest : message
| BarrierReply : message.
```

### A.2.44 OpenFlowSemantics Library

```
Set Implicit Arguments.
Require Import Coq.Lists.List.
Require Import Common.Types.
Require Import Word.WordInterface.
Require Import Network.NetworkPacket.
Require Import OpenFlow.OpenFlow0x01Types.
Require Import OpenFlow.FlowTable.
Import ListNotations.
Local Open Scope list_scope.
Local Open Scope bool_scope.
Section Actions.
  Definition setVlan dlVlan pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp _ dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
    end.
  Definition setVlanPriority dlVlanPcp pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp dlVlan _ nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
    end.
  Definition stripVlanHeader pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp dlVlan _ nw ⇒
        Packet dlSrc dlDst dlTyp VLAN_NONE Word8.zero nw
    end.
  Definition setEthSrcAddr dlSrc pkt :=
    match pkt with
      | Packet _ dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
    end.
  Definition setEthDstAddr dlDst pkt :=
    match pkt with
      | Packet dlSrc _ dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw
    end.
  Definition setIPSrcAddr_nw src pkt :=
    match pkt with
      | NwUnparsable pf data ⇒
        NwUnparsable pf data
      | NwIP (IP vhl tos len ident flags frag ttl proto chksum _ dst tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp)
      | NwARP (ARPQuery sha _ tpa) ⇒
        NwARP (ARPQuery sha src tpa)
      | NwARP (ARPReply sha _ tha tpa) ⇒
        NwARP (ARPReply sha src tha tpa)
    end.
  Definition setIPSrcAddr nwSrc pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlTyp dlVlanPcp (setIPSrcAddr_nw nwSrc nw)
    end.
  Definition setIPDstAddr_nw dst pkt :=
    match pkt with
      | NwUnparsable pf data ⇒
        NwUnparsable pf data
      | NwIP (IP vhl tos len ident flags frag ttl proto chksum src _ tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp)
      | NwARP (ARPQuery sha spa _) ⇒
        NwARP (ARPQuery sha spa dst)
      | NwARP (ARPReply sha spa tha _) ⇒
        NwARP (ARPReply sha spa tha dst)
    end.
  Definition setIPDstAddr nwDst pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlTyp dlVlanPcp (setIPDstAddr_nw nwDst nw)
    end.

  Definition setIPToS_nw tos pkt :=
    match pkt with
      | NwUnparsable pf data ⇒
        NwUnparsable pf data
      | NwIP (IP vhl _ len ident flags frag ttl proto chksum src dst tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp)
      | NwARP (ARPQuery dlSrc nwSrc nwDst) ⇒
        NwARP (ARPQuery dlSrc nwSrc nwDst)
      | NwARP (ARPReply dlSrc nwSrc dlDst nwDst) ⇒
        NwARP (ARPReply dlSrc nwSrc dlDst nwDst)
    end.

  Definition setIPToS nwToS pkt :=
    match pkt with
      | Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlTyp dlVlan dlVlanPcp (setIPToS_nw nwToS nw)
    end.

  Definition setTransportSrcPort_tp tpSrc pkt :=
    match pkt with
      | TpTCP (Tcp _ dst seq ack off flags win chksum urgent payload) ⇒
        TpTCP (Tcp tpSrc dst seq ack off flags win chksum urgent payload)
      | TpICMP icmp ⇒ TpICMP icmp
      | TpUnparsable proto data ⇒ TpUnparsable proto data
    end.

  Definition setTransportSrcPort_nw tpSrc pkt :=
    match pkt with
      | NwUnparsable pf data ⇒
        NwUnparsable pf data
      | NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst
                    (setTransportSrcPort_tp tpSrc tp))
      | NwARP arp ⇒
        NwARP arp
    end.
  Definition setTransportSrcPort tpSrc pkt :=
    match pkt with
      | Packet dlSrc dlDst typ dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst typ dlVlan dlVlanPcp
        (setTransportSrcPort_nw tpSrc nw)
    end.
  Definition setTransportDstPort_tp tpDst pkt :=
    match pkt with
      | TpTCP (Tcp src _ seq ack off flags win chksum urgent payload) ⇒
        TpTCP (Tcp src tpDst seq ack off flags win chksum urgent payload)
      | TpICMP icmp ⇒ TpICMP icmp
      | TpUnparsable proto data ⇒ TpUnparsable proto data
    end.
  Definition setTransportDstPort_nw tpDst pkt :=
    match pkt with
      | NwUnparsable pf data ⇒ NwUnparsable pf data
```

```
    | NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst tp) ⇒
      NwIP (IP vhl tos len ident flags frag ttl proto chksum src dst
                (setTransportDstPort_tp tpDst tp))
    | NwARP arp ⇒ NwARP arp
  end.

Definition setTransportDstPort tpDst pkt :=
  match pkt with
    | Packet dlSrc dlDst proto dlVlan dlVlanPcp nw ⇒
      Packet dlSrc dlDst proto dlVlan dlVlanPcp
      (setTransportSrcPort_nw tpDst nw)
  end.

Definition apply_action (pt : portId) (pk : packet) (act : action) :=
  match act with
    | Output pp ⇒ inl pp
    | SetDlVlan vlan ⇒ inr (setVlan vlan pk)
    | StripVlan ⇒ inr (stripVlanHeader pk)
    | SetDlVlanPcp prio ⇒ inr (setVlanPriority prio pk)
    | SetDlSrc addr ⇒ inr (setEthSrcAddr addr pk)
    | SetDlDst addr ⇒ inr (setEthDstAddr addr pk)
    | SetNwSrc ip ⇒ inr (setIPSrcAddr ip pk)
    | SetNwDst ip ⇒ inr (setIPDstAddr ip pk)
    | SetNwTos tos ⇒ inr (setIPToS tos pk)
    | SetTpSrc pt ⇒ inr (setTransportSrcPort pt pk)
    | SetTpDst pt ⇒ inr (setTransportDstPort pt pk)
  end.
```

```
Fixpoint apply_actionSequence (pt : portId) (pk : packet)
  (acts : actionSequence) : list (pseudoPort × packet) :=
  match acts with
    | nil ⇒ nil
    | act :: acts' ⇒
      match apply_action pt pk act with
        | inl pp ⇒ (pp,pk) :: apply_actionSequence pt pk acts'
        | inr pk' ⇒ apply_actionSequence pt pk' acts'
      end
  end.
End Actions.
Section Match.
Definition match_opt {A : Type} (eq_dec : Eqdec A) (x : A) (v : option A) :=
  match v with
    | None ⇒ true
    | Some y ⇒ if eq_dec x y then true else false
  end.
Definition match_tp pk (mat : of_match) :=
  match mat with
    | Match _ _ _ _ _ _ _ _ _ _ mTpSrc mTpDst _ ⇒
      match pk with
        | TpTCP (Tcp tpSrc tpDst _ _ _ _ _ _ _ _ _) ⇒
          match_opt Word16.eq_dec tpSrc mTpSrc &&
          match_opt Word16.eq_dec tpDst mTpDst
        | TpICMP (Icmp typ code _ _) ⇒
```

```
          true



        | TpUnparsable _ _ ⇒
          true
      end
  end.

Definition match_nw pk (mat : of_match) :=
  match mat with
    | Match _ _ _ _ _ mNwSrc mNwDst mNwProto mNwTos _ _ _ ⇒
      match pk with
        | NwIP (IP _ nwTos _ _ _ _ _ nwProto _ nwSrc nwDst tpPkt) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst &&
          match_opt Word8.eq_dec nwTos mNwTos &&
          match mNwProto with
            | None ⇒ true
            | Some nwProto' ⇒
              if Word8.eq_dec nwProto nwProto' then
                match_tp tpPkt mat
              else
                false
          end
        | NwARP (ARPQuery _ nwSrc nwDst) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst
        | NwARP (ARPReply _ nwSrc _ nwDst) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst
        | NwUnparsable _ _ ⇒ true
      end
    end.
  Definition match_ethFrame (pk : packet) (pt : portId) (mat : of_match) :=
    match mat with
      | Match mDlSrc mDlDst mDlTyp mDlVlan mDlVlanPcp mNwSrc mNwDst mNw-
Proto
      mNwTos mTpSrc mTpDst mInPort ⇒
      match_opt Word16.eq_dec pt mInPort &&
      match pk with
        | Packet pkDlSrc pkDlDst pkDlTyp pkDlVlan pkDlVlanPcp pkNwFrame ⇒
          match_opt Word48.eq_dec pkDlSrc mDlSrc &&
          match_opt Word48.eq_dec pkDlDst mDlDst &&
          match_opt Word16.eq_dec pkDlVlan mDlVlan &&
          match_opt Word8.eq_dec pkDlVlanPcp mDlVlanPcp &&
          match mDlTyp with
            | None ⇒ true
            | Some dlTyp' ⇒
              if Word16.eq_dec pkDlTyp dlTyp' then
                match_nw pkNwFrame mat
                else
                false
            end
        end
    end.
End Match.

Section FlowTable.
  Require Import Word.WordTheory.
  Definition gt16 (x y : Word16.Word.t) : Prop :=
    Word16.le x y → False.
  Inductive Match : flowTable → packet → portId → option actionSequence → Prop :=
  | Matched : ∀ tbl1 prio pat act tbl2 pt pk,
    match_ethFrame pk pt pat = true →
    (∀ rule,
        In rule (tbl1 ++ tbl2) →
        gt16 (priority rule) prio →
        match_ethFrame pk pt (pattern rule) = false) →
    Match (tbl1 ++ Rule prio pat act :: tbl2) pk pt (Some act)
  | Unmatched : ∀ tbl pt pk,
    (∀ rule,
        In rule tbl →
        match_ethFrame pk pt (pattern rule) = false) →
    Match tbl pk pt None.
End FlowTable.
```

### A.2.45 ControllerInterface0x04 Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Common.Monad.

Require Import OpenFlow13.OpenFlow0x01Types.

Require Import Network.NetworkPacket.

Inductive event : Type :=
  | SwitchConnected : switchId → event
  | SwitchDisconnected : switchId → event
  | SwitchMessage : switchId → xid → message → event.
```

Using a thin trusted shim, written in Haskell, we drive verified controllers that match this signature. `Module Type CONTROLLER_MONAD <: MONAD.`

```
  Include MONAD.

  Parameter state : Type.

  Parameter get : m state.

  Parameter put : state → m unit.

  Parameter send : switchId → xid → message → m unit.

  Parameter recv : m event.
```

Must be defined in OCaml `Parameter forever : m unit → m unit.`

```
End CONTROLLER_MONAD.
```

### A.2.46 MessagesDef Library

```
Set Implicit Arguments.

Require Import Coq.Structures.Equalities.

Require Import Word.WordInterface.

Require Import Network.NetworkPacket.

Definition VLAN_NONE : dlVlan := @Word16.Mk 65535 eq_refl.

Extract Constant VLAN_NONE ⇒ "65535".

Record of_match : Type := Match {
  matchDlSrc : option dlAddr;
  matchDlDst : option dlAddr;
  matchDlTyp : option dlTyp;
  matchDlVlan : option dlVlan;
  matchDlVlanPcp : option dlVlanPcp;
  matchNwSrc : option nwAddr;
  matchNwDst : option nwAddr;
  matchNwProto : option nwProto;
  matchNwTos : option nwTos;
  matchTpSrc : option tpPort;
  matchTpDst : option tpPort;
  matchInPort : option portId
}.

Record capabilities : Type := Capabilities {
  flow_stats: bool;
  table_stats: bool;
  port_stats: bool;
  stp: bool;
  ip_reasm: bool;
  queue_stats: bool;
  arp_match_ip: bool
}.
Record actions : Type := Actions {
  output: bool;
  set_vlan_id: bool;
  set_vlan_pcp: bool;
  strip_vlan: bool;
  set_dl_src: bool;
  set_dl_dst: bool;
  set_nw_src: bool;
  set_nw_dst: bool;
  set_nw_tos: bool;
  set_tp_src: bool;
  set_tp_dst: bool;
  enqueue: bool;
  vendor: bool
}.
Record features : Type := Features {
  switch_id : Word64.t;
  num_buffers: Word32.t;
  num_tables: Word8.t;
  supported_capabilities: capabilities;
  supported_actions: actions

}.
Inductive flowModCommand : Type :=
| AddFlow : flowModCommand
| ModFlow : flowModCommand
| ModStrictFlow : flowModCommand
| DeleteFlow : flowModCommand
| DeleteStrictFlow : flowModCommand.
Definition priority := Word16.t.
Definition bufferId := Word32.t.
Definition groupId := Word32.t.
Inductive pseudoPort : Type :=
| PhysicalPort : portId → pseudoPort
| InPort : pseudoPort
| Flood : pseudoPort
| AllPorts : pseudoPort
| Controller : Word16.t → pseudoPort.
Inductive action : Type :=
| Output : pseudoPort → action
| Group : groupId → action
| SetDlVlan : dlVlan → action
| SetDlVlanPcp : dlVlanPcp → action
| StripVlan : action
| SetDlSrc : dlAddr → action
| SetDlDst : dlAddr → action
| SetNwSrc : nwAddr → action
| SetNwDst : nwAddr → action
| SetNwTos : nwTos → action
| SetTpSrc : tpPort → action
| SetTpDst : tpPort → action.

Definition actionSequence := list action.

Record bucket := Bucket {
  weight : Word16.t;
  watch_port : portId;
  watch_group : groupId;
  bucket_actions : list action
}.

Definition bucketSequence := list bucket.

Inductive groupType : Type :=
| All : groupType
| Select : groupType
| Indirect : groupType
| FastFailover : groupType.

Inductive groupModCommand : Type :=
| AddGroup : groupModCommand
| DelGroup : groupModCommand.

Record groupMod := GroupMod {
  mgModCmd : groupModCommand;
  mgType : groupType;
  mgId : groupId;
  mgBuckets : bucketSequence
}.
Inductive timeout : Type :=
| Permanent : timeout
| ExpiresAfter : ∀ (n : Word16.t),
    Word16.to_nat n > Word16.to_nat Word16.zero →
    timeout.
Record flowMod := FlowMod {
  mfModCmd : flowModCommand;
  mfMatch : of_match;
  mfPriority : priority;
  mfActions : actionSequence;
  mfCookie : Word64.t;
  mfIdleTimeOut : timeout;
  mfHardTimeOut : timeout;
  mfNotifyWhenRemoved : bool;
  mfApplyToPacket : option bufferId;
  mfOutPort : option pseudoPort;
  mfOutGroup : option groupId;
  mfCheckOverlap : bool
}.
Inductive packetInReason : Type :=
| NoMatch : packetInReason
| ExplicitSend : packetInReason.

Record packetIn : Type := PacketIn {
  packetInBufferId : option bufferId;
  packetInTotalLen : Word16.t;
  packetInPort : portId;
  packetInReason_ : packetInReason;
  packetInPacket : packet
}.

Definition xid : Type := Word32.t.

Inductive message : Type :=
| Hello : bytes → message
| EchoRequest : bytes → message
| EchoReply : bytes → message
| FeaturesRequest : message
| FeaturesReply : features → message
| FlowModMsg : flowMod → message
| GroupModMsg : groupMod → message
| PacketInMsg : packetIn → message.
```

### A.2.47 OpenFlow0x04Semantics Library

```
Set Implicit Arguments.

Require Import Common.Types.

Require Import Word.WordInterface.

Require Import Network.NetworkPacket.
Require Import OpenFlow13.OpenFlow0x04Types.

Local Open Scope list_scope.

Local Open Scope bool_scope.

Section Actions.

  Definition setVlan dlVlan pkt :=
    match pkt with
      | Packet dlSrc dlDst _ _ dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlVlanPcp nw
    end.

  Definition setVlanPriority dlVlanPcp pkt :=
    match pkt with
      | Packet dlSrc dlDst _ dlVlan _ nw ⇒
        Packet dlSrc dlDst dlVlan dlVlanPcp nw
    end.

  Definition stripVlanHeader pkt :=
    match pkt with
      | Packet dlSrc dlDst _ dlVlan _ nw ⇒
        Packet dlSrc dlDst VLAN_NONE Word8.zero nw
    end.

  Definition setEthSrcAddr dlSrc pkt :=
    match pkt with
      | Packet _ dlDst _ dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlVlanPcp nw
    end.

  Definition setEthDstAddr dlDst pkt :=
    match pkt with
      | Packet dlSrc _ _ dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlVlanPcp nw
    end.

  Definition setIPSrcAddr_nw (ethTyp : dlTyp) (src : nwAddr)
    (pkt : nw ethTyp) : nw ethTyp :=
    match pkt with
      | NwUnparsable pf data ⇒
        NwUnparsable pf data
      | NwIP (IP vhl tos len ident flags frag ttl _ chksum _ dst tp) ⇒
        NwIP (IP vhl tos len ident flags frag ttl chksum src dst tp)
      | NwARP (ARPQuery sha _ tpa) ⇒
        NwARP (ARPQuery sha src tpa)
      | NwARP (ARPReply sha _ tha tpa) ⇒
        NwARP (ARPReply sha src tha tpa)
    end.

  Definition setIPSrcAddr nwSrc pkt :=
    match pkt with
      | Packet dlSrc dlDst _ dlVlan dlVlanPcp nw ⇒
        Packet dlSrc dlDst dlVlan dlVlanPcp (setIPSrcAddr_nw nwSrc nw)
    end.

  Definition setIPDstAddr_nw (ethTyp : dlTyp) (dst : nwAddr)
    (pkt : nw ethTyp) : nw ethTyp :=
    match pkt with
      | NwUnparsable pf data ⇒
```

```
      NwUnparsable pf data
    | NwIP (IP vhl tos len ident flags frag ttl _ chksum src _ tp) ⇒
      NwIP (IP vhl tos len ident flags frag ttl chksum src dst tp)
    | NwARP (ARPQuery sha spa _) ⇒
      NwARP (ARPQuery sha spa dst)
    | NwARP (ARPReply sha spa tha _) ⇒
      NwARP (ARPReply sha spa tha dst)
  end.

Definition setIPDstAddr nwDst pkt :=
  match pkt with
    | Packet dlSrc dlDst _ dlVlan dlVlanPcp nw ⇒
      Packet dlSrc dlDst dlVlan dlVlanPcp (setIPDstAddr_nw nwDst nw)
  end.

Definition setIPToS_nw (ethTyp : dlTyp) (tos : nwTos)
  (pkt : nw ethTyp) : nw ethTyp :=
  match pkt with
    | NwUnparsable pf data ⇒
      NwUnparsable pf data
    | NwIP (IP vhl _ len ident flags frag ttl _ chksum src dst tp) ⇒
      NwIP (IP vhl tos len ident flags frag ttl chksum src dst tp)
    | NwARP (ARPQuery dlSrc nwSrc nwDst) ⇒
      NwARP (ARPQuery dlSrc nwSrc nwDst)
    | NwARP (ARPReply dlSrc nwSrc dlDst nwDst) ⇒
      NwARP (ARPReply dlSrc nwSrc dlDst nwDst)
  end.
```

```
Definition setIPToS nwToS pkt :=
  match pkt with
    | Packet dlSrc dlDst _ dlVlan dlVlanPcp nw ⇒
       Packet dlSrc dlDst dlVlan dlVlanPcp (setIPToS_nw nwToS nw)
  end.

Definition setTransportSrcPort_tp (proto : nwProto)
  (tpSrc : tpPort) (pkt : tpPkt proto) : tpPkt proto :=
  match pkt with
    | TpTCP (Tcp _ dst seq ack off flags win payload) ⇒
       TpTCP (Tcp tpSrc dst seq ack off flags win payload)
    | TpICMP icmp ⇒ TpICMP icmp
    | TpUnparsable proto data ⇒ TpUnparsable proto data
  end.

Definition setTransportSrcPort_nw (ethTyp : dlTyp)
  (tpSrc : tpPort) (pkt : nw ethTyp) : nw ethTyp :=
  match pkt with
    | NwUnparsable pf data ⇒
       NwUnparsable pf data
    | NwIP (IP vhl tos len ident flags frag ttl _ chksum src dst tp) ⇒
       NwIP (IP vhl tos len ident flags frag ttl chksum src dst
                  (setTransportSrcPort_tp tpSrc tp))
    | NwARP arp ⇒
       NwARP arp
  end.

Definition setTransportSrcPort tpSrc pkt :=
  match pkt with
    | Packet dlSrc dlDst _ dlVlan dlVlanPcp nw ⇒
      Packet dlSrc dlDst dlVlan dlVlanPcp
      (setTransportSrcPort_nw tpSrc nw)
  end.
Definition setTransportDstPort_tp (proto : nwProto)
  (tpDst : tpPort) (pkt : tpPkt proto) : tpPkt proto :=
  match pkt with
    | TpTCP (Tcp src _ seq ack off flags win payload) ⇒
      TpTCP (Tcp src tpDst seq ack off flags win payload)
    | TpICMP icmp ⇒ TpICMP icmp
    | TpUnparsable proto data ⇒ TpUnparsable proto data
  end.
Definition setTransportDstPort_nw (ethTyp : dlTyp)
  (tpDst : tpPort) (pkt : nw ethTyp) : nw ethTyp :=
  match pkt with
    | NwUnparsable pf data ⇒ NwUnparsable pf data
    | NwIP (IP vhl tos len ident flags frag ttl _ chksum src dst tp) ⇒
      NwIP (IP vhl tos len ident flags frag ttl chksum src dst
                (setTransportDstPort_tp tpDst tp))
    | NwARP arp ⇒ NwARP arp
  end.
Definition setTransportDstPort tpDst pkt :=
  match pkt with
    | Packet dlSrc dlDst _ dlVlan dlVlanPcp nw ⇒
    Packet dlSrc dlDst dlVlan dlVlanPcp
    (setTransportSrcPort_nw tpDst nw)
  end.
Inductive action_result : Type :=
| arPort : pseudoPort → action_result
| arPkt : packet → action_result
| arGroup : groupId → action_result.
Definition apply_action (pt : portId) (pk : packet) (act : action) :=
  match act with
    | Output pp ⇒ arPort pp
    | Group gid ⇒ arGroup gid
    | SetDlVlan vlan ⇒ arPkt (setVlan vlan pk)
    | StripVlan ⇒ arPkt (stripVlanHeader pk)
    | SetDlVlanPcp prio ⇒ arPkt (setVlanPriority prio pk)
    | SetDlSrc addr ⇒ arPkt (setEthSrcAddr addr pk)
    | SetDlDst addr ⇒ arPkt (setEthDstAddr addr pk)
    | SetNwSrc ip ⇒ arPkt (setIPSrcAddr ip pk)
    | SetNwDst ip ⇒ arPkt (setIPDstAddr ip pk)
    | SetNwTos tos ⇒ arPkt (setIPToS tos pk)
    | SetTpSrc pt ⇒ arPkt (setTransportSrcPort pt pk)
    | SetTpDst pt ⇒ arPkt (setTransportDstPort pt pk)
  end.
Fixpoint apply_actionSequence (pt : portId) (pk : packet)
  (acts : actionSequence) : list ((pseudoPort + groupId) × packet) :=
  match acts with
```

```
    | nil ⇒ nil
    | act :: acts' ⇒
      match apply_action pt pk act with
        | arPort pp ⇒ (inl pp,pk) :: apply_actionSequence pt pk acts'
        | arGroup gid ⇒ (inr gid,pk) :: apply_actionSequence pt pk acts'
        | arPkt pk' ⇒ apply_actionSequence pt pk' acts'
      end
  end.
End Actions.
Section Match.
 Definition match_opt {A : Type} (eq_dec : Eqdec A) (x : A) (v : option A) :=
  match v with
    | None ⇒ true
    | Some y ⇒ if eq_dec x y then true else false
  end.
 Definition match_tp (nwProto : nwProto) (pk : tpPkt nwProto)
  (mat : of_match) :=
  match mat with
    | Match _ _ _ _ _ _ _ _ _ _ mTpSrc mTpDst _ ⇒
      match pk with
        | TpTCP (Tcp tpSrc tpDst _ _ _ _ _ _) ⇒
          match_opt Word16.eq_dec tpSrc mTpSrc &&
          match_opt Word16.eq_dec tpDst mTpDst
        | TpICMP (Icmp typ code _ _) ⇒
          true
        | TpUnparsable _ _ ⇒
          true
      end
    end.

Definition match_nw (ethTyp : dlTyp) (pk : nw ethTyp)
  (mat : of_match) :=
  match mat with
    | Match _ _ _ _ _ mNwSrc mNwDst mNwProto mNwTos _ _ _ ⇒
      match pk with
        | NwIP (IP _ nwTos _ _ _ _ _ nwProto _ nwSrc nwDst tpPkt) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst &&
          match_opt Word8.eq_dec nwTos mNwTos &&
          match mNwProto with
            | None ⇒ true
            | Some nwProto' ⇒
              if Word8.eq_dec nwProto nwProto' then
                match_tp tpPkt mat
              else
                false
          end
        | NwARP (ARPQuery _ nwSrc nwDst) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst
        | NwARP (ARPReply _ nwSrc _ nwDst) ⇒
          match_opt Word32.eq_dec nwSrc mNwSrc &&
          match_opt Word32.eq_dec nwDst mNwDst
        | NwUnparsable _ _ ⇒ true
      end
    end.

  Definition match_ethFrame (pk : packet) (pt : portId) (mat : of_match) :=
    match mat with
      | Match mDlSrc mDlDst mDlTyp mDlVlan mDlVlanPcp mNwSrc mNwDst mNw-
Proto
        mNwTos mTpSrc mTpDst mInPort ⇒
        match_opt Word16.eq_dec pt mInPort &&
        match pk with
          | Packet pkDlSrc pkDlDst pkDlTyp pkDlVlan pkDlVlanPcp pkNwFrame ⇒
            match_opt Word48.eq_dec pkDlSrc mDlSrc &&
            match_opt Word48.eq_dec pkDlDst mDlDst &&
            match_opt Word16.eq_dec pkDlVlan mDlVlan &&
            match_opt Word8.eq_dec pkDlVlanPcp mDlVlanPcp &&
            match mDlTyp with
              | None ⇒ true
              | Some dlTyp' ⇒
                if Word16.eq_dec pkDlTyp dlTyp' then
                  match_nw pkNwFrame mat
                  else
```

```
                    false
                end
            end
        end.
End Match.
```

### A.2.48 OpenFlowTypes Library

```
Set Implicit Arguments.
Require Import Coq.Structures.Equalities.
Require Import Word.WordInterface.
Require Import Network.NetworkPacket.
Definition VLAN_NONE : dlVlan := @Word16.Mk 65535 eq_refl.
Extract Constant VLAN_NONE ⇒ "65535".
Record mask A := Mask {
  m_value : A;
  m_mask : option A
}.
Definition xid := Word32.t.
Definition val_to_mask {A} (v : A) := Mask v None.
Definition switchId := Word64.t.
Definition groupId := Word32.t.
Definition portId := Word32.t.
Definition tableId := Word8.t.
Definition bufferId := Word32.t.
```

See Table 11 of the specification

```
Inductive oxm : Type :=
  | OxmInPort : portId → oxm
  | OxmInPhyPort : portId → oxm
  | OxmMetadata : mask Word64.t → oxm
  | OxmEthType : Word16.t → oxm
  | OxmEthDst : mask Word48.t → oxm
  | OxmEthSrc : mask Word48.t → oxm
  | OxmVlanVId : mask Word12.t → oxm
  | OxmVlanPcp : Word8.t → oxm
  | OxmIPProto : Word8.t → oxm
  | OxmIPDscp : Word8.t → oxm
  | OxmIPEcn : Word8.t → oxm
  | OxmIP4Src : mask Word32.t → oxm
  | OxmIP4Dst : mask Word32.t → oxm
  | OxmTCPSrc : mask Word16.t → oxm
  | OxmTCPDst : mask Word16.t → oxm
  | OxmARPOp : Word16.t → oxm
  | OxmARPSpa : mask Word32.t → oxm
  | OxmARPTpa : mask Word32.t → oxm
  | OxmARPSha : mask Word48.t → oxm
  | OxmARPTha : mask Word48.t → oxm
  | OxmICMPType : Word8.t → oxm
  | OxmICMPCode : Word8.t → oxm
  | OxmMPLSLabel : Word32.t → oxm
```

| *OxmMPLSTc* : *Word8.t* → *oxm*

| *OxmTunnelId* : *mask* *Word64.t* → *oxm*.

Hard-codes OFPMT_OXM as the match type, since OFPMT_STANDARD is deprecated.

`Definition` *oxmMatch* := *list* *oxm*.

`Inductive` *pseudoPort* : `Type` :=

| *PhysicalPort* : *portId* → *pseudoPort*

| *InPort* : *pseudoPort*

| *Flood* : *pseudoPort*

| *AllPorts* : *pseudoPort*

| *Controller* : *Word16.t* → *pseudoPort*

| *Any* : *pseudoPort*.

`Inductive` *action* : `Type` :=

| *Output* : *pseudoPort* → *action*

| *Group* : *groupId* → *action*

| *PopVlan* : *action*

| *PushVlan* : *action*

| *PopMpls* : *action*

| *PushMpls* : *action*

| *SetField* : *oxm* → *action*.

`Definition` *actionSequence* := *list* *action*.

`Inductive` *instruction* : `Type` :=

| *GotoTable* : *tableId* → *instruction*

| *ApplyActions* : *actionSequence* → *instruction*

| *WriteActions* : *actionSequence* → *instruction*.

```
Record bucket := Bucket {
  bu_weight : Word16.t;
  bu_watch_port : option portId;
  bu_watch_group : option groupId;
  bu_actions : actionSequence
}.

Inductive groupType : Type :=
  | All : groupType
  | Select : groupType
  | Indirect : groupType
  | FF : groupType.

Inductive groupMod : Type :=
  | AddGroup : groupType → groupId → list bucket → groupMod
  | DeleteGroup : groupType → groupId → groupMod.

Inductive timeout : Type :=
| Permanent : timeout
| ExpiresAfter : ∀ (n : Word16.t),
    Word16.to_nat n > Word16.to_nat Word16.zero →
    timeout.

Inductive flowModCommand : Type :=
| AddFlow : flowModCommand
| ModFlow : flowModCommand
| ModStrictFlow : flowModCommand
| DeleteFlow : flowModCommand
| DeleteStrictFlow : flowModCommand.
```

```
Record flowModFlags : Type := FlowModFlags {
  fmf_send_flow_rem : bool;
  fmf_check_overlap : bool;
  fmf_reset_counts : bool;
  fmf_no_pkt_counts : bool;
  fmf_no_byt_counts : bool
}.

Record flowMod := FlowMod {
  mfCookie : mask Word64.t;
  mfTable_id : tableId;
  mfCommand : flowModCommand;
  mfIdle_timeout : timeout;
  mfHard_timeout : timeout;
  mfPriority : Word16.t;
  mfBuffer_id : option bufferId;
  mfOut_port : option pseudoPort;
  mfOut_group : option groupId;
  mfFlags : flowModFlags;
  mfOfp_match : oxmMatch;
  mfInstructions : list instruction
}.

Inductive packetInReason : Type :=
| NoMatch : packetInReason
| ExplicitSend : packetInReason.

Record packetIn : Type := PacketIn {
  pi_buffer_id : option Word32.t;
  pi_total_len : Word16.t;
  pi_reason : packetInReason;
  pi_table_id : tableId;
  pi_cookie : Word64.t;
  pi_ofp_match : oxmMatch;
  pi_pkt : option packet
}.
Record capabilities : Type := Capabilities {
  flow_stats : bool;
  table_stats : bool;
  port_stats : bool;
  group_stats : bool;
  ip_reasm : bool;
  queue_stats : bool;
  port_blocked : bool
}.
Record features : Type := Features {
  datapath_id : Word64.t;
  num_buffers : Word32.t;
  num_tables : Word8.t;
  aux_id : Word8.t;
  supported_capabilities : capabilities
}.
Inductive packetOut : Type := PacketOut {
  po_buffer_id : option bufferId;
  po_in_port : pseudoPort;
  po_actions : actionSequence;
  po_pkt : option packet
}.

Inductive message : Type :=
  | Hello : message
  | EchoRequest : bytes → message
  | EchoReply : bytes → message
  | FeaturesRequest : message
  | FeaturesReply : features → message
  | FlowModMsg : flowMod → message
  | GroupModMsg : groupMod → message
  | PacketInMsg : packetIn → message
  | PacketOutMsg : packetOut → message
  | BarrierRequest : message
  | BarrierReply : message.
```

### A.2.49 Pattern Library

```
Set Implicit Arguments.
Require Import Pattern.PatternImplDef.
Require Import Pattern.PatternImplTheory.
Require Import Pattern.PatternInterface.
Require Import Coq.Lists.List.
Require Import Wildcard.Wildcard.

Require Import Network.NetworkPacket.

Require Import Coq.Classes.Equivalence.

Local Open Scope equiv_scope.

Module Pattern : PATTERN.

  Record pat := Pat {

    raw : pattern;

    valid : ValidPattern raw

  }.

  Definition t := pat.

  Definition beq (p1 p2 : t) :=

    match eq_dec (raw p1) (raw p2) with

      | left _ ⇒ true

      | right _ ⇒ false

    end.

  Definition inter (p1 p2 : t) :=

    Pat (inter_preserves_valid (valid p1) (valid p2)).

  Lemma all_is_Valid : ValidPattern all.

  Proof.

    apply ValidPat_any.

  Qed.

  Definition all : t := Pat all_is_Valid.

  Lemma empty_is_valid : ValidPattern empty.

  Proof.
    apply ValidPat_None.
    reflexivity.
  Qed.

  Definition empty : t := Pat empty_is_valid.

  Definition exact_pattern pk pt : t :=
    Pat (exact_is_valid pt pk).

  Definition is_empty pat : bool := is_empty (raw pat).

  Definition match_packet pt pk pat : bool :=
    match_packet pt pk (raw pat).

  Definition is_exact pat : bool := is_exact (raw pat).

  Definition to_match pat (H : is_empty pat = false) :=
    to_match (raw pat) H.

  Section Constructors.

    Definition inPort pt : t :=
      @Pat
        (Pattern
           WildcardAll
           WildcardAll
           WildcardAll
           WildcardAll
           WildcardAll
           WildcardAll
           WildcardAll
           WildcardAll
```

```
        WildcardAll
        WildcardAll
        WildcardAll
        (WildcardExact pt))
    (ValidPat_any _ _ _ _ _ _ _).
Definition dlSrc dlAddr : t :=
  @Pat
    (Pattern
      (WildcardExact dlAddr)
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll)
    (ValidPat_any (WildcardExact dlAddr) _ _ _ _ _).
Definition dlDst dlAddr : t :=
  @Pat
    (Pattern
        WildcardAll
      (WildcardExact dlAddr)
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll)
    (ValidPat_any _ (WildcardExact dlAddr) _ _ _ _).

Definition dlTyp typ : t :=
  @Pat
    (Pattern
       WildcardAll
       WildcardAll
      (WildcardExact typ)
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
       WildcardAll
```

```
        WildcardAll
        WildcardAll)
    (ValidPat_any _ _ (WildcardExact typ) _ _ _).

Definition dlVlan vlan : t :=
  @Pat
    (Pattern
        WildcardAll
        WildcardAll
        WildcardAll
        (WildcardExact vlan)
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll)
    (ValidPat_any _ _ _ (WildcardExact vlan) _ _).

Definition dlVlanPcp pcp : t :=
  @Pat
    (Pattern
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        (WildcardExact pcp)
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll
        WildcardAll)
    (ValidPat_any _ _ _ _ (WildcardExact pcp) _).

Definition ipSrc addr : t :=
  @Pat
    (Pattern
      WildcardAll
      WildcardAll
      (WildcardExact Const_0x800)
      WildcardAll
      WildcardAll
      (WildcardExact addr)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll)
    (ValidPat_IP_any _ _ _ _ (WildcardExact addr) _ _ _ _).
Definition ipDst addr : t :=
  @Pat
    (Pattern
      WildcardAll
      WildcardAll
      (WildcardExact Const_0x800)
      WildcardAll
      WildcardAll
      WildcardAll
      (WildcardExact addr)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll)
    (ValidPat_IP_any _ _ _ _ _ (WildcardExact addr) _ _ _).
Definition ipProto proto : t :=
  @Pat
    (Pattern
      WildcardAll
      WildcardAll
      (WildcardExact Const_0x800)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      (WildcardExact proto)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll)
    (ValidPat_IP_any _ _ _ _ _ _ _ _ (WildcardExact proto)).

Definition tpSrcPort proto (H : In proto SupportedNwProto) tpPort : t :=
  @Pat
    (Pattern
      WildcardAll
      WildcardAll
      (WildcardExact Const_0x800)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      (WildcardExact proto)
      WildcardAll
      (WildcardExact tpPort)
      WildcardAll
      WildcardAll)
    (@ValidPat_TCPUDP _ _ _ _ _ _ _ (WildcardExact tpPort) _ _ _ H).

Definition tpDstPort proto (H : In proto SupportedNwProto) tpPort : t :=
  @Pat
    (Pattern
      WildcardAll
      WildcardAll
      (WildcardExact Const_0x800)
      WildcardAll
      WildcardAll
      WildcardAll
      WildcardAll
      (WildcardExact proto)
      WildcardAll
      WildcardAll
      (WildcardExact tpPort)
      WildcardAll)
    (@ValidPat_TCPUDP _ _ _ _ _ _ _ _ (WildcardExact tpPort) _ _ H).

Lemma TCP_is_supported : In Const_0x6 SupportedNwProto.
Proof with auto with datatypes.
  unfold SupportedNwProto...
Qed.

Lemma UDP_is_supported : In Const_0x7 SupportedNwProto.
Proof with auto with datatypes.
  unfold SupportedNwProto...
Qed.

Definition tcpSrcPort := tpSrcPort TCP_is_supported.

Definition tcpDstPort := tpDstPort TCP_is_supported.
```

```
    Definition udpSrcPort := tpSrcPort UDP_is_supported.

    Definition udpDstPort := tpDstPort UDP_is_supported.

  End Constructors.

    Definition equiv (pat1 pat2 : t) : Prop :=

      ∀ pt pk,

        match_packet pt pk pat1 = match_packet pt pk pat2.

    Lemma equiv_is_Equivalence : Equivalence equiv.

    Proof with auto.

      unfold equiv.

      unfold match_packet.

      split.

      unfold Reflexive...

      unfold Symmetric...

      unfold Transitive...

      intros.

      rewrite → H...

    Qed.

Instance Pattern_Equivalence : Equivalence equiv.

  apply equiv_is_Equivalence.

Qed.

Section Lemmas.

  Lemma inter_comm : ∀ (p p0 : pat), equiv (inter p p0) (inter p0 p).

  Proof with auto.

    unfold equiv.

    unfold match_packet.
  unfold inter.
  intros.
  simpl.
  rewrite → inter_comm...
Qed.

Lemma inter_assoc : ∀ (p p' p'' : pat),
  equiv (inter p (inter p' p'')) (inter (inter p p') p'').
Proof with auto.
  unfold equiv.
  unfold match_packet.
  unfold inter.
  intros.
  simpl.
  rewrite → inter_assoc...
Qed.

Hint Unfold inter is_empty is_exact equiv match_packet.

Lemma is_empty_false_distr_l : ∀ x y,
  is_empty (inter x y) = false →
  is_empty x = false .
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_empty_false_distr_l...
Qed.

Lemma is_empty_false_distr_r : ∀ x y,
  is_empty (inter x y) = false →
  is_empty y = false.
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_empty_false_distr_r...
Qed.

Lemma is_empty_true_l : ∀ x y,
  is_empty x = true →
  is_empty (inter x y) = true.
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_empty_true_l...
Qed.

Lemma is_empty_true_r : ∀ x y,
  is_empty y = true →
  is_empty (inter x y) = true.
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_empty_true_r...
Qed.

Lemma is_match_false_inter_l :
  ∀ pt (pkt : packet) pat1 pat2,
    match_packet pt pkt pat1 = false →
    match_packet pt pkt (inter pat1 pat2) = false.
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_match_false_inter_l...
Qed.

Lemma is_match_false_inter_r :
  ∀ pt (pkt : packet) pat1 pat2,
    match_packet pt pkt pat2 = false →
    match_packet pt pkt (inter pat1 pat2) = false.
Proof with eauto.
  intros.
  autounfold in *.
  eapply is_match_false_inter_r...
Qed.

Lemma no_match_subset_r : ∀ k n t t',
  match_packet n k t' = false →
  match_packet n k (inter t t') = false.
Proof with eauto.
  intros.
  autounfold in *.
  eapply no_match_subset_r...
Qed.

Lemma exact_match_inter : ∀ x y,
  is_exact x = true →
  is_empty (inter x y) = false →
  equiv (inter x y) x.
Proof with eauto.
  intros.
  unfold equiv.
  unfold match_packet.
  intros.
  destruct x.
  destruct y.
  unfold is_exact in *.
  unfold inter in *.
  unfold is_empty in *.
  simpl in H.
  simpl in H0.
  pose (J := PatternImplTheory.exact_match_inter _ _ H H0).
  simpl.
  rewrite → J...
Qed.

Lemma all_spec : ∀ pt pk,
  match_packet pt pk all = true.
Proof with auto.
  unfold all.
  unfold match_packet.
  simpl.
  exact all_spec.
Qed.

Lemma all_is_not_empty : is_empty all = false.
Proof.
  reflexivity.
Qed.

Lemma exact_match_is_exact : ∀ pk pt,
  is_exact (exact_pattern pk pt) = true.
Proof with auto.
  unfold exact_pattern.
  unfold is_exact.
  intros.
  apply exact_match_is_exact.
Qed.

Lemma exact_intersect : ∀ k n t,
  match_packet k n t = true →
  equiv (inter (exact_pattern n k) t) (exact_pattern n k).
Proof with auto.
  unfold equiv.
  unfold exact_pattern.
  unfold match_packet.
  unfold inter.
  intros.
  simpl.
  pose (J := exact_intersect k n (raw t0) H).
  rewrite → J...
Qed.

Lemma is_match_true_inter : ∀ pat1 pat2 pt pk,
  match_packet pt pk pat1 = true →
  match_packet pt pk pat2 = true →
  match_packet pt pk (inter pat1 pat2) = true.
Proof with auto.
  intros.
  unfold match_packet in *.
  unfold inter.
  simpl.
  rewrite → is_match_true_inter...
Qed.

Lemma beq_true_spec : ∀ p p',
  beq p p' = true →
  equiv p p'.
Proof with auto.
  intros.
  unfold equiv.
  unfold match_packet.
  destruct p.
  destruct p'.
  unfold beq in H.
  simpl in H.
  destruct (eq_dec raw0 raw1); subst...
```

```
    inversion H.
  Qed.

  Lemma match_packet_spec : ∀ pt pk pat,
    match_packet pt pk pat =
    negb (is_empty (inter (exact_pattern pk pt) pat)).
  Proof.
    intros.
    destruct pat0.
    unfold match_packet.
    unfold is_empty.
    unfold inter.
    unfold exact_pattern.
    unfold PatternImplDef.match_packet.
    unfold raw.
    reflexivity.
  Qed.

End Lemmas.

End Pattern.

Definition pattern := Pattern.t.
```

### A.2.50 PatternImplDef Library

```
Set Implicit Arguments.

Require Import Coq.Arith.EqNat.

Require Import NPeano.
Require Import Arith.Peano_dec.
Require Import Bool.Bool.
Require Import Coq.Classes.Equivalence.
Require Import Coq.Lists.List.
Require Import OpenFlow.OpenFlow0x01Types.
Require Import Common.Types.
Require Import Word.WordInterface.
Require Import Network.NetworkPacket.
Require Import Wildcard.Wildcard.
Local Open Scope bool_scope.
Local Open Scope list_scope.
Record pattern : Type := Pattern {
  ptrnDlSrc : Wildcard dlAddr;
  ptrnDlDst : Wildcard dlAddr;
  ptrnDlType : Wildcard dlTyp;
  ptrnDlVlan : Wildcard dlVlan;
  ptrnDlVlanPcp : Wildcard dlVlanPcp;
  ptrnNwSrc : Wildcard nwAddr;
  ptrnNwDst : Wildcard nwAddr;
  ptrnNwProto : Wildcard nwProto;
  ptrnNwTos : Wildcard nwTos;
  ptrnTpSrc : Wildcard tpPort;
  ptrnTpDst : Wildcard tpPort;
  ptrnInPort : Wildcard portId
}.
```

```
Lemma eq_dec : ∀ (x y : pattern), { x = y } + { x ≠ y }.
Proof.
  decide equality;
    try solve [ apply (Wildcard.eq_dec Word16.eq_dec) |
      apply (Wildcard.eq_dec Word32.eq_dec) |
        apply (Wildcard.eq_dec Word8.eq_dec) |
          apply (Wildcard.eq_dec Word48.eq_dec) ].
Defined.

Definition Wildcard_of_option { a : Type } (def : a) (v : option a) :=
  WildcardExact (match v with
                       | None ⇒ def
                       | Some v ⇒ v
                     end).

Definition all :=
  Pattern
    WildcardAll WildcardAll WildcardAll WildcardAll WildcardAll
    WildcardAll WildcardAll WildcardAll WildcardAll WildcardAll
    WildcardAll WildcardAll.

Definition empty :=
  Pattern WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone
  WildcardNone.
```

Note that we do not have a unique representation for empty patterns! `Definition`

```
is_empty pat : bool :=
  match pat with
    | Pattern dlSrc dlDst typ vlan pcp nwSrc nwDst nwProto nwTos tpSrc tpDst
      inPort ⇒
      Wildcard.is_empty inPort ||
        Wildcard.is_empty dlSrc ||
          Wildcard.is_empty dlDst ||
            Wildcard.is_empty vlan ||
              Wildcard.is_empty pcp ||
                Wildcard.is_empty typ ||
                  Wildcard.is_empty nwSrc ||
                    Wildcard.is_empty nwDst ||
                      Wildcard.is_empty nwTos ||
                        Wildcard.is_empty nwProto ||
                          Wildcard.is_empty tpSrc ||
                            Wildcard.is_empty tpDst
  end.
Lemma is_empty_neq_None : ∀ {A : Type} (w : Wildcard A),
  Wildcard.is_empty w = false → w ≠ WildcardNone.
Proof.
  unfold not.
  intros.
  destruct w.
  inversion H0.
  inversion H0.
  simpl in H.
  inversion H.
Qed.

Hint Resolve is_empty_neq_None.

Lemma is_empty_dlSrc : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                              nwProto nwTos tpSrc tpDst inPort) = false →
  dlSrc ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_dlDst : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                    nwProto nwTos tpSrc tpDst inPort) = false →
  dlDst ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_dlTyp : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                    nwProto nwTos tpSrc tpDst inPort) = false →
  dlTyp ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_dlVlan : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                    nwProto nwTos tpSrc tpDst inPort) = false →
  dlVlan ≠ WildcardNone.
Proof with auto.
```

```
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_dlVlanPcp : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                           nwProto nwTos tpSrc tpDst inPort) = false →
  dlVlanPcp ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_nwSrc : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                           nwProto nwTos tpSrc tpDst inPort) = false →
  nwSrc ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_nwDst : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                    nwProto nwTos tpSrc tpDst inPort) = false →
  nwDst ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_nwProto : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                    nwProto nwTos tpSrc tpDst inPort) = false →
  nwProto ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_nwTos : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
```

```
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                          nwProto nwTos tpSrc tpDst inPort) = false →
  nwTos ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_tpSrc : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                          nwProto nwTos tpSrc tpDst inPort) = false →
  tpSrc ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_tpDst : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                          nwProto nwTos tpSrc tpDst inPort) = false →
  tpDst ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma is_empty_inPort : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  is_empty (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                           nwProto nwTos tpSrc tpDst inPort) = false →
  inPort ≠ WildcardNone.
Proof with auto.
  intros.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H)...
Qed.

Lemma to_match : ∀ pat (H : is_empty pat = false), of_match.
Proof.
  intros.
  destruct pat.
  exact (Match
    (Wildcard.to_option (is_empty_dlSrc _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_dlDst _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_dlTyp _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_dlVlan _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_dlVlanPcp _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_nwSrc _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_nwDst _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_nwProto _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_nwTos _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_tpSrc _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_tpDst _ _ _ _ _ _ _ _ _ _ _ H))
    (Wildcard.to_option (is_empty_inPort _ _ _ _ _ _ _ _ _ _ _ H))).
 Defined.

Definition inter p p' :=
  let dlSrc := Wildcard.inter Word48.eq_dec (ptrnDlSrc p)
    (ptrnDlSrc p') in
  let dlDst := Wildcard.inter Word48.eq_dec (ptrnDlDst p)
    (ptrnDlDst p') in
  let dlType := Wildcard.inter Word16.eq_dec (ptrnDlType p) (ptrnDlType p') in
  let dlVlan := Wildcard.inter Word16.eq_dec (ptrnDlVlan p) (ptrnDlVlan p') in
  let dlVlanPcp := Wildcard.inter Word8.eq_dec (ptrnDlVlanPcp p)
    (ptrnDlVlanPcp p') in
   let nwSrc := Wildcard.inter Word32.eq_dec (ptrnNwSrc p) (ptrnNwSrc p') in
   let nwDst := Wildcard.inter Word32.eq_dec (ptrnNwDst p) (ptrnNwDst p') in
   let nwProto := Wildcard.inter Word8.eq_dec (ptrnNwProto p)
     (ptrnNwProto p') in
   let nwTos := Wildcard.inter Word8.eq_dec (ptrnNwTos p) (ptrnNwTos p') in
```

```
    let tpSrc := Wildcard.inter Word16.eq_dec (ptrnTpSrc p) (ptrnTpSrc p') in
    let tpDst := Wildcard.inter Word16.eq_dec (ptrnTpDst p) (ptrnTpDst p') in
    let inPort := Wildcard.inter Word16.eq_dec (ptrnInPort p) (ptrnInPort p') in
      Pattern dlSrc dlDst dlType dlVlan dlVlanPcp
        nwSrc nwDst nwProto nwTos
        tpSrc tpDst
        inPort.

Definition exact_pattern (pk : packet) (pt : Word16.t) :=
  Pattern
  (WildcardExact (pktDlSrc pk))
  (WildcardExact (pktDlDst pk))
  (WildcardExact (pktDlTyp pk))
  (WildcardExact (pktDlVlan pk))
  (WildcardExact (pktDlVlanPcp pk))
  (WildcardExact (pktNwSrc pk))
  (WildcardExact (pktNwDst pk))
  (WildcardExact (pktNwProto pk))
  (WildcardExact (pktNwTos pk))
  (WildcardExact (pktTpSrc pk))
  (WildcardExact (pktTpDst pk))
  (WildcardExact pt).

Definition match_packet (pt : Word16.t) (pk : packet) pat :=
  negb (is_empty (inter (exact_pattern pk pt) pat)).

Definition is_exact pat :=
  match pat with
  | Pattern dlSrc dlDst typ vlan pcp nwSrc nwDst nwProto nwTos tpSrc tpDst
    inPort ⇒
    Wildcard.is_exact inPort &&
    Wildcard.is_exact dlSrc &&
    Wildcard.is_exact dlDst &&
    Wildcard.is_exact typ &&
    Wildcard.is_exact vlan &&
    Wildcard.is_exact pcp &&
    Wildcard.is_exact nwSrc &&
    Wildcard.is_exact nwDst &&
    Wildcard.is_exact nwProto &&
    Wildcard.is_exact nwTos &&
    Wildcard.is_exact tpSrc &&
    Wildcard.is_exact tpDst
  end.

Definition SupportedNwProto :=
  [ Const_0x6;
    Const_0x7 ].

Definition SupportedDlTyp :=
  [ Const_0x800; Const_0x806 ].

Inductive ValidPattern : pattern → Prop :=
| ValidPat_TCPUDP : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
    nwTos tpSrc tpDst inPort nwProto,
    In nwProto SupportedNwProto →
  ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
    dlVlan dlVlanPcp
    nwSrc nwDst (WildcardExact nwProto)
    nwTos tpSrc tpDst inPort)
| ValidPat_ARP : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
    inPort,
  ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x806)
    dlVlan dlVlanPcp
    nwSrc nwDst (WildcardExact Word8.zero)
    (WildcardExact Word8.zero)
    (WildcardExact Word16.zero)
    (WildcardExact Word16.zero)
    inPort)
| ValidPat_IP_generic : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
      nwTos inPort nwProto,
  ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
    dlVlan dlVlanPcp
    nwSrc nwDst nwProto
    nwTos
    (WildcardExact Word16.zero)
    (WildcardExact Word16.zero)
    inPort)
| ValidPat_generic : ∀ dlSrc dlDst dlVlan dlVlanPcp
    inPort frameTyp,
  ValidPattern (Pattern dlSrc dlDst (WildcardExact frameTyp)
    dlVlan dlVlanPcp
    (WildcardExact Word32.zero)
    (WildcardExact Word32.zero)
    (WildcardExact Word8.zero)
    (WildcardExact Word8.zero)
    (WildcardExact Word16.zero)
    (WildcardExact Word16.zero)
    inPort)
| ValidPat_any : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp inPort,
    ValidPattern
      (Pattern dlSrc
               dlDst
               dlTyp
               dlVlan
               dlVlanPcp
               WildcardAll
               WildcardAll
               WildcardAll
               WildcardAll
               WildcardAll
               WildcardAll
               inPort)
| ValidPat_IP_any : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
      nwTos inPort nwProto,
   ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
    dlVlan dlVlanPcp
    nwSrc nwDst nwProto
    nwTos
```

```
      WildcardAll
      WildcardAll
      inPort)
| ValidPat_None : ∀ pat,
      is_empty pat = true →
   ValidPattern pat.
```

### A.2.51 PatternImplTheory Library

```
Set Implicit Arguments.
Require Import Coq.Arith.EqNat.
Require Import NPeano.
Require Import Arith.Peano_dec.
Require Import Bool.Bool.
Require Import Coq.Classes.Equivalence.
Require Import Lists.List.
Require Import Word.WordInterface.
Require Import Network.NetworkPacket.
Require Import Common.Types.
Require Import Pattern.PatternImplDef.
Require Import Wildcard.Wildcard.
Require Import Wildcard.Theory.
Require Import OpenFlow.OpenFlowSemantics.
Open Scope bool_scope.
Open Scope list_scope.
Open Scope equiv_scope.

Create HintDb pattern.

Lemma IP_ARP_frametyp_neq : Const_0x800 ≠ Const_0x806.

Proof with auto.

  assert ({ Const_0x800 = Const_0x806 } + { Const_0x800 ≠ Const_0x806 }).

  apply Word16.eq_dec.

  unfold Const_0x800 in *.

  unfold Const_0x806 in *.

  destruct H.

  inversion e.

  trivial.

Qed.

Hint Unfold inter empty is_empty.

Hint Resolve Word8.eq_dec Word16.eq_dec Word32.eq_dec Word48.eq_dec.

Lemma inter_comm : ∀ p p', inter p p' = inter p' p.

Proof with auto.

  intros.

  destruct p.

  destruct p'.

  unfold inter.

  simpl.

  rewrite → (inter_comm _ ptrnDlSrc0).

  rewrite → (inter_comm _ ptrnDlDst0).

  rewrite → (inter_comm _ ptrnDlType0).

  rewrite → (inter_comm _ ptrnDlVlan0).
  rewrite → (inter_comm _ ptrnDlVlanPcp0).
  rewrite → (inter_comm _ ptrnNwSrc0).
  rewrite → (inter_comm _ ptrnNwDst0).
  rewrite → (inter_comm _ ptrnNwTos0).
  rewrite → (inter_comm _ ptrnTpSrc0).
  rewrite → (inter_comm _ ptrnTpDst0).
  rewrite → (inter_comm _ ptrnInPort0).
  rewrite → (inter_comm _ ptrnNwProto0).
  reflexivity.
Qed.

Lemma inter_assoc : ∀ p p' p'',
  inter p (inter p' p'') = inter (inter p p') p''.
Proof with simpl; auto.
  intros.
  unfold inter.
  simpl.
  repeat rewrite → inter_assoc...
Qed.

Lemma is_empty_false_distr_l : ∀ x y,
  is_empty (inter x y) = false →
  is_empty x = false .
Proof with simpl; eauto.
  intros.
  unfold inter in H.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H).
  unfold is_empty.
  destruct x.
  destruct y.
  simpl in *.
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  reflexivity.
Qed.

Lemma is_empty_false_distr_r : ∀ x y,
  is_empty (inter x y) = false →
  is_empty y = false .
Proof.
  intros.
  rewrite → inter_comm in H.
  eapply is_empty_false_distr_l.
  exact H.
Qed.

Lemma is_empty_true_l : ∀ x y,
  is_empty x = true →
  is_empty (inter x y) = true.
Proof with auto.
  intros.
  destruct x.
  destruct y.
  simpl in *.
  repeat rewrite → orb_true_iff in H.
  repeat rewrite → or_assoc in H.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  repeat (destruct H; auto 13 with wildcard).
Qed.

Lemma is_empty_true_r : ∀ x y,
  is_empty y = true →
  is_empty (inter x y) = true.
Proof with auto.
  intros.
  rewrite inter_comm.
  apply is_empty_true_l...
```

```
Qed.

Lemma is_match_false_inter_l :
  ∀ (pt : portId) (pkt : packet) pat1 pat2,
    match_packet pt pkt pat1 = false →
    match_packet pt pkt (inter pat1 pat2) = false.
Proof with auto.
  intros.
  unfold match_packet in *.
  rewrite → negb_false_iff in H.
  rewrite → negb_false_iff.
  rewrite → inter_assoc.
  apply is_empty_true_l...
Qed.

Lemma is_match_false_inter_r :
  ∀ (pt : portId) (pkt : packet) pat1 pat2,
    match_packet pt pkt pat2 = false →
    match_packet pt pkt (inter pat1 pat2) = false.
Proof with auto.
  intros.
  unfold match_packet in *.
  rewrite → negb_false_iff in H.
  rewrite → negb_false_iff.
  rewrite inter_comm with (p := pat1).
  rewrite → inter_assoc.
  apply is_empty_true_l...
```

```
Qed.

Lemma no_match_subset_r : ∀ k n t t',
  match_packet n k t' = false →
  match_packet n k (inter t t') = false.
Proof with auto.
  intros.
  rewrite → inter_comm.
  apply is_match_false_inter_l...
Qed.

Lemma exact_match_inter : ∀ x y,
  is_exact x = true →
  is_empty (inter x y) = false →
  inter x y = x.
Proof with auto.
  intros.
  destruct x. destruct y. simpl in *.
  repeat rewrite → andb_true_iff in H.
  do 11 (destruct H).
  repeat rewrite → orb_false_iff in H0.
  do 11 (destruct H0).
  unfold inter.
  unfold inter.
  simpl.
  repeat rewrite → exact_match_inter_l...
Qed.
```

Lemma *all_spec* : ∀ *pt pk*,

*match_packet pt pk all* = *true*.

Proof with auto.

intros.

unfold *match_packet*.

rewrite → *negb_true_iff*.

unfold *all*.

simpl.

reflexivity.

Qed.

Lemma *exact_match_is_exact* : ∀ *pk pt*,

*is_exact* (*exact_pattern pk pt*) = *true*.

Proof with auto.

intros.

unfold *exact_pattern*.

unfold *is_exact*.

unfold *Wildcard.is_exact*.

simpl.

unfold *Wildcard_of_option*.

simpl.

destruct (*pktTpSrc pk*); destruct (*pktTpDst pk*)...

Qed.

Lemma *exact_intersect* : ∀ *k n t*,

*match_packet k n t* = *true* →

*inter* (*exact_pattern n k*) *t* = *exact_pattern n k*.

```
Proof with auto.
  intros.
  unfold match_packet in H.
  rewrite → negb_true_iff in H.
  apply exact_match_inter...
Qed.

Lemma is_match_true_inter : ∀ pat1 pat2 pt pk,
  match_packet pt pk pat1 = true →
  match_packet pt pk pat2 = true →
  match_packet pt pk (inter pat1 pat2) = true.
Proof with auto.
  intros.
  apply exact_intersect in H.
  apply exact_intersect in H0.
  unfold match_packet.
  rewrite → negb_true_iff.
  rewrite → inter_assoc.
  rewrite → H.
  rewrite → H0.
  unfold exact_pattern.
  unfold is_empty.
  simpl.
  reflexivity.
Qed.

Hint Rewrite inter_assoc : pattern.
Hint Resolve is_empty_false_distr_l is_empty_false_distr_r : pattern.

Hint Resolve is_empty_true_l is_empty_true_r : pattern.

Hint Resolve is_match_false_inter_l : pattern.

Hint Resolve all_spec : pattern.

Hint Resolve exact_match_is_exact : pattern.

Hint Resolve exact_intersect : pattern.

Hint Resolve is_match_true_inter : pattern.

Hint Resolve is_empty_true_l is_empty_true_r.

Hint Constructors ValidPattern.

Lemma exact_is_valid : ∀ pt pk, ValidPattern (exact_pattern pk pt).

Proof with auto with datatypes.

  intros.

  unfold exact_pattern.

  destruct pk; simpl.

  destruct pktNwHeader; simpl...

  destruct i; simpl.

  destruct pktTpHeader; simpl...

Admitted.

Lemma pres0 : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst

  nwProto nwProto' nwTos tpSrc tpDst inPort,

  nwProto ≠ nwProto' →

  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst

    (Wildcard.inter Word8.eq_dec

      (WildcardExact nwProto)

      (WildcardExact nwProto'))
    nwTos tpSrc tpDst inPort).
Proof with auto with wildcard.
  intros.
  apply ValidPat_None.
  simpl.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  rewrite → inter_exact_neq...
  simpl.
  auto 13.
Qed.

Hint Immediate pres0.

Lemma pres1 : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwProto' nwTos tpSrc tpDst inPort,
  In nwProto SupportedNwProto →
  ¬ In nwProto' SupportedNwProto →
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
    (Wildcard.inter Word8.eq_dec
      (WildcardExact nwProto)
      (WildcardExact nwProto'))
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word8.eq_dec nwProto nwProto').
  destruct X; subst...
```

```
Qed.

Hint Immediate pres1.

Lemma pres1' : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwProto' nwTos tpSrc tpDst inPort,
  ¬ In nwProto SupportedNwProto →
  In nwProto' SupportedNwProto →
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
    (Wildcard.inter Word8.eq_dec
      (WildcardExact nwProto)
      (WildcardExact nwProto'))
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word8.eq_dec nwProto nwProto').
  destruct X; subst...
Qed.

Hint Immediate pres1'.

Lemma pres2 : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  dlTyp ≠ dlTyp' →
  ValidPattern (Pattern dlSrc dlDst
    (Wildcard.inter Word16.eq_dec
      (WildcardExact dlTyp)
      (WildcardExact dlTyp'))
    dlVlan dlVlanPcp nwSrc nwDst nwProto
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply ValidPat_None.
  simpl.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  rewrite → inter_exact_neq; auto 13.
Qed.

Hint Immediate pres2.

Lemma pres3 : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  In dlTyp SupportedDlTyp →
  ¬ In dlTyp' SupportedDlTyp →
  ValidPattern (Pattern dlSrc dlDst
    (Wildcard.inter Word16.eq_dec
      (WildcardExact dlTyp)
      (WildcardExact dlTyp'))
    dlVlan dlVlanPcp nwSrc nwDst nwProto
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word16.eq_dec dlTyp dlTyp').
  destruct X; subst...
Qed.
```

```
Hint Resolve pres3.

Lemma pres3' : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  ¬ In dlTyp SupportedDlTyp →
  In dlTyp' SupportedDlTyp →
  ValidPattern (Pattern dlSrc dlDst
    (Wildcard.inter Word16.eq_dec
      (WildcardExact dlTyp)
      (WildcardExact dlTyp'))
    dlVlan dlVlanPcp nwSrc nwDst nwProto
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word16.eq_dec dlTyp dlTyp').
  destruct X; subst...
Qed.

Hint Resolve pres3'.

Lemma pres4 : In Const_0x800 SupportedDlTyp.
Proof with auto with datatypes.
  intros.
  unfold SupportedDlTyp...
Qed.

Hint Resolve pres4.

Lemma pres4' : In Const_0x806 SupportedDlTyp.
Proof with auto with datatypes.
```

```
  intros.
  unfold SupportedDlTyp...
Qed.

Hint Resolve pres4'.

Lemma pres5 : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  ValidPattern (Pattern dlSrc dlDst
    (Wildcard.inter Word16.eq_dec
      (WildcardExact Const_0x800)
      (WildcardExact Const_0x806))
    dlVlan dlVlanPcp nwSrc nwDst nwProto
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply pres2.
  exact IP_ARP_frametyp_neq.
Qed.

Hint Immediate pres5.

Lemma pres5' : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  ValidPattern (Pattern dlSrc dlDst
    (Wildcard.inter Word16.eq_dec
      (WildcardExact Const_0x806)
      (WildcardExact Const_0x800))
    dlVlan dlVlanPcp nwSrc nwDst nwProto
    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply pres2.
  unfold not.
  intros.
  symmetry in H.
  apply IP_ARP_frametyp_neq...
Qed.

Hint Immediate pres5'.

Lemma dlTyp_None_Valid : ∀ dlSrc dlDst dlVlan dlVlanPcp
  nwSrc nwDst nwProto nwTos tpSrc tpDst inPort,
  ValidPattern (Pattern dlSrc dlDst WildcardNone dlVlan dlVlanPcp
                          nwSrc nwDst nwProto nwTos
                          tpSrc tpDst inPort).
Proof with auto with bool.
  intros.
  apply ValidPat_None.
  unfold is_empty.
  simpl.
  rewrite → orb_true_r...
Qed.

Lemma nwProto_None_Valid : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp
  nwSrc nwDst nwTos tpSrc tpDst inPort,
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp
                                nwSrc nwDst WildcardNone nwTos
                                tpSrc tpDst inPort).
Proof with auto with bool.
  intros.
  apply ValidPat_None.
  unfold is_empty...
Qed.

Lemma tpSrc_None_Valid : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp
  nwSrc nwDst nwProto nwTos tpDst inPort,
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp
                                nwSrc nwDst nwProto nwTos
                                WildcardNone tpDst inPort).
Proof with auto with bool.
  intros.
  apply ValidPat_None.
  unfold is_empty...
Qed.

Lemma tpDst_None_Valid : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp
  nwSrc nwDst nwProto nwTos tpSrc inPort,
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp
                                nwSrc nwDst nwProto nwTos
                                tpSrc WildcardNone inPort).
Proof with auto with bool.
  intros.
  apply ValidPat_None.
  unfold is_empty...
Qed.

Hint Immediate dlTyp_None_Valid nwProto_None_Valid tpSrc_None_Valid
  tpDst_None_Valid.

Lemma empty_valid_l : ∀ pat pat',
  is_empty pat = true →
  ValidPattern (inter pat pat').
Proof with auto.
  intros.
  apply ValidPat_None.
  apply is_empty_true_l...
Qed.

Lemma empty_valid_r : ∀ pat pat',
  is_empty pat' = true →
  ValidPattern (inter pat pat').
Proof with auto.
  intros.
  apply ValidPat_None.
  apply is_empty_true_r...
Qed.

Ltac inter_solve :=
  unfold inter; simpl; autorewrite with wildcard using auto.

Lemma dlTyp_inter_exact_r : ∀ dlSrc dlDst dlTyp k dlVlan dlVlanPcp
  nwSrc nwDst nwProto nwTos tpSrc tpDst inPort,
  ValidPattern
```

```
    (Pattern dlSrc dlDst (WildcardExact k)
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort) →
  ValidPattern
    (Pattern dlSrc dlDst WildcardNone
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort) →
  ValidPattern
    (Pattern dlSrc dlDst
      (Wildcard.inter Word16.eq_dec dlTyp (WildcardExact k))
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  destruct (is_exact_split_r Word16.eq_dec dlTyp k).
  assert (Wildcard.inter Word16.eq_dec dlTyp (WildcardExact k) =
          WildcardExact k) as J...
  rewrite → J...
  assert (Wildcard.inter Word16.eq_dec dlTyp (WildcardExact k) =
           WildcardNone) as J...
  rewrite → J...
Qed.

Lemma dlTyp_inter_exact_l : ∀ dlSrc dlDst dlTyp k dlVlan dlVlanPcp
  nwSrc nwDst nwProto nwTos tpSrc tpDst inPort,
  ValidPattern
    (Pattern dlSrc dlDst (WildcardExact k)
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort) →
  ValidPattern
```

```
    (Pattern dlSrc dlDst WildcardNone
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort) →
  ValidPattern
    (Pattern dlSrc dlDst
      (Wildcard.inter Word16.eq_dec (WildcardExact k) dlTyp)
      dlVlan dlVlanPcp nwSrc nwDst nwProto nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  destruct (is_exact_split_l Word16.eq_dec k dlTyp).
  assert (Wildcard.inter Word16.eq_dec (WildcardExact k) dlTyp =
          WildcardExact k) as J...
  rewrite → J...
  assert (Wildcard.inter Word16.eq_dec (WildcardExact k) dlTyp =
           WildcardNone) as J...
  rewrite → J...
Qed.

Hint Resolve dlTyp_inter_exact_l dlTyp_inter_exact_r.

Axiom zero_not_supportedProto : In Word8.zero SupportedNwProto → False.

Hint Resolve zero_not_supportedProto.

Lemma inter_preserves_valid : ∀ pat1 pat2,
  ValidPattern pat1 →
  ValidPattern pat2 →
  ValidPattern (inter pat1 pat2).
Proof with auto.
  intros pat1 pat2 H H0.
```

```
destruct H; destruct H0;
  try solve [ auto using empty_valid_l, empty_valid_r |
                inter_solve; auto ].
pose (X := Word8.eq_dec nwProto nwProto0); destruct X; subst; inter_solve.
inter_solve.
pose (J0 := is_exact_split_r Word16.eq_dec tpSrc Word16.zero).
pose (J1 := is_exact_split_r Word16.eq_dec tpDst Word16.zero).
destruct J0. destruct J1.
rewrite → H0.
rewrite → H1...
rewrite → H1...
rewrite → H0...
inter_solve.
apply dlTyp_inter_exact_l...
remember (Word8.eq_dec nwProto Word8.zero) as J0.
destruct J0; subst...
inter_solve.
destruct (is_exact_split_l Word8.eq_dec nwProto nwProto0).
assert (Wildcard.inter Word8.eq_dec (WildcardExact nwProto)
  nwProto0 = WildcardExact nwProto)...
rewrite → H1...
assert (Wildcard.inter Word8.eq_dec (WildcardExact nwProto)
  nwProto0 = WildcardNone)...
rewrite → H1...
inter_solve.
```

```
pose (J := is_exact_split_r Word8.eq_dec nwProto nwProto0).
destruct J.
inter_solve.
assert (Wildcard.inter Word8.eq_dec nwProto
                    (WildcardExact nwProto0) = WildcardExact nwProto0)...
rewrite → H1.
pose (J0 := is_exact_split_l Word16.eq_dec Word16.zero tpSrc).
pose (J1 := is_exact_split_l Word16.eq_dec Word16.zero tpDst).
destruct J0.
destruct J1.
rewrite → H2.
rewrite → H3...
rewrite → H3...
rewrite → H2...
inter_solve.
assert (Wildcard.inter Word8.eq_dec nwProto
                    (WildcardExact nwProto0) = WildcardNone)...
rewrite → H1...
inter_solve.
apply dlTyp_inter_exact_r...
remember (Word8.eq_dec Word8.zero nwProto) as J0.
destruct J0; subst...
inter_solve.
destruct (is_exact_split_r Word8.eq_dec nwProto nwProto0).
assert (Wildcard.inter Word8.eq_dec nwProto (WildcardExact nwProto0)
      = WildcardExact nwProto0)...
    rewrite → H1...
    assert (Wildcard.inter Word8.eq_dec nwProto (WildcardExact nwProto0)
      = WildcardNone)...
    rewrite → H1...
Qed.

Section Equivalence.

  Definition Pattern_equiv (pat1 pat2 : pattern) : Prop :=
    ∀ pt pk,
      match_packet pt pk pat1 = match_packet pt pk pat2.

  Hint Unfold Pattern_equiv.

  Lemma Pattern_equiv_is_Equivalence : Equivalence Pattern_equiv.
  Proof with auto.
    split.
    unfold Reflexive...
    unfold Symmetric...
    unfold Transitive.
      unfold Pattern_equiv.
      intros.
      rewrite → H...
    Qed.

End Equivalence.

Instance Pattern_Equivalance : Equivalence Pattern_equiv.
apply Pattern_equiv_is_Equivalence.
Qed.
```

```
Lemma match_opt_const_equiv : ∀ (A : Type)
      (eq_dec : Eqdec A) (val : A) (opt : A),
   negb (Wildcard.is_empty
            (Wildcard.inter eq_dec (WildcardExact val) (WildcardExact opt))) =
   if eq_dec val opt then true else false.
Proof with auto.
   intros.
   unfold Wildcard.inter.
   destruct (eq_dec opt val); subst...
   destruct (eq_dec val val)...
   destruct (eq_dec val opt); subst...
Qed.

Lemma trans : ∀ (A : Type) (eq : Eqdec A) (x : A)
   (w : Wildcard A) (H : w ≠ WildcardNone),
   match_opt eq x (Wildcard.to_option H) =
   negb (Wildcard.is_empty (Wildcard.inter eq (WildcardExact x) w)).
Proof with auto.
   intros.
   destruct w.
   simpl.
   remember (eq x a) as b.
   destruct b.
   subst...
   rewrite → inter_exact_eq.
   unfold Wildcard.is_empty...
```

```
  rewrite → inter_exact_neq...
  simpl...
  contradiction H...
Qed.

  Lemma icmp_tpSrc : ∀ pat,
    ValidPattern pat →
    is_empty pat = false →
    ptrnNwProto pat = WildcardExact Const_0x1 →
    ptrnTpSrc pat = WildcardAll ∨ ptrnTpSrc pat = WildcardExact Word16.zero.
  Proof with auto.
    intros.
    destruct pat.
    simpl in *.
    subst.
    inversion H; subst...
    unfold SupportedNwProto in H2.
    destruct H2. inversion H1. destruct H1. inversion H1. inversion H1.
    simpl in H1.
    simpl in H0.
    rewrite → H0 in H1.
    inversion H1.
  Qed.

  Lemma icmp_tpDst : ∀ pat,
    ValidPattern pat →
    is_empty pat = false →
    ptrnNwProto pat = WildcardExact Const_0x1 →
    ptrnTpDst pat = WildcardAll ∨ ptrnTpDst pat = WildcardExact Word16.zero.
  Proof with auto.
    intros.
    destruct pat.
    simpl in *.
    subst.
    inversion H; subst...
    unfold SupportedNwProto in H2.
    destruct H2. inversion H1. destruct H1. inversion H1. inversion H1.
    simpl in H1.
    simpl in H0.
    rewrite → H0 in H1.
    inversion H1.
  Qed.
Theorem match_equiv : ∀ pt pk pat (Hempty : is_empty pat = false),
  ValidPattern pat →
  match_ethFrame pk pt (to_match pat Hempty) =
  match_packet pt pk pat.
Proof with auto.
  intros.
  destruct pat.
  destruct pk.
  unfold match_packet.
  inversion H.
```

```
subst.
simpl.
repeat rewrite → negb_orb.
destruct (Word16.eq_dec pktDlTyp Const_0x800).
destruct pktNwHeader.
destruct i.
destruct (Word8.eq_dec pktIPProto nwProto).
destruct pktTpHeader.
destruct t.
subst.
repeat rewrite → trans.
repeat rewrite → andb_assoc.
simpl.
repeat rewrite → inter_exact_eq.
simpl.
rewrite → andb_true_r.
rewrite → andb_true_r.
reflexivity.
destruct i.
subst.
repeat rewrite → trans.
repeat rewrite → andb_assoc.
simpl.
repeat rewrite → inter_exact_eq.
simpl.
rewrite → andb_true_r.
```

```
  rewrite → andb_true_r.
  rewrite → andb_true_r.
  assert (
   negb
     (Wildcard.is_empty
        (Wildcard.inter Word16.eq_dec (WildcardExact Word16.zero) ptrnTpSrc)) =
     true).
    destruct (icmp_tpSrc H Hempty)...
Admitted.
```

### A.2.52 PatternInterface Library

```
Set Implicit Arguments.

Require Import Coq.Classes.Equivalence.
Require Import WordInterface.
Require Import Network.NetworkPacket.
Require Import OpenFlow.OpenFlow0x01Types.

Local Open Scope equiv_scope.

Module Type PATTERN.

  Parameter t : Type.

  Parameter inter : t → t → t.

  Parameter all : t.

  Parameter empty : t.

  Parameter exact_pattern : packet → portId → t.
```

Parameter *is_empty* : $t \to$ *bool*.

Parameter *match_packet* : *portId* $\to$ *packet* $\to t \to$ *bool*.

Parameter *is_exact* : $t \to$ *bool*.

Parameter *to_match* : $\forall$ *x*, *is_empty* $x =$ *false* $\to$ *of_match*.

Parameter *beq* : $t \to t \to$ *bool*.

Constructors that produce valid patterns.

Parameter *dlSrc* : *dlAddr* $\to t$.

Parameter *dlDst* : *dlAddr* $\to t$.

Parameter *dlTyp* : *dlTyp* $\to t$.

TODO(arjun): Only the 12 lower bits matter. If higher-order bits are non-zero, we might calculate incorrect intersections here too. Parameter *dlVlan* : *dlVlan* $\to t$.

Parameter *dlVlanPcp* : *dlVlanPcp* $\to t$.

Parameter *ipSrc* : *nwAddr* $\to t$.

Parameter *ipDst* : *nwAddr* $\to t$.

Parameter *ipProto* : *nwProto* $\to t$.

Parameter *inPort* : *portId* $\to t$.

Parameter *tcpSrcPort* : *tpPort* $\to t$.

Parameter *tcpDstPort* : *tpPort* $\to t$.

Parameter *udpSrcPort* : *tpPort* $\to t$.

Parameter *udpDstPort* : *tpPort* $\to t$.

Pattern equivalence

```
Definition equiv (pat1 pat2 : t) : Prop :=
  ∀ pt pk,
    match_packet pt pk pat1 = match_packet pt pk pat2.

Parameter equiv_is_Equivalence : Equivalence equiv.

Instance Pattern_Equivalence : Equivalence equiv.
  apply equiv_is_Equivalence.
Qed.

Parameter beq_true_spec : ∀ p p',
  beq p p' = true →
  p === p'.

Parameter inter_comm : ∀ p p', inter p p' === inter p' p.

Parameter inter_assoc : ∀ p p' p'',
  inter p (inter p' p'') === inter (inter p p') p''.

Parameter is_empty_false_distr_l : ∀ x y,
  is_empty (inter x y) = false →
  is_empty x = false .

Parameter is_empty_false_distr_r : ∀ x y,
  is_empty (inter x y) = false →
  is_empty y = false .

Parameter is_empty_true_l : ∀ x y,
  is_empty x = true →
  is_empty (inter x y) = true.

Parameter is_empty_true_r : ∀ x y,
  is_empty y = true →
  is_empty (inter x y) = true.

Parameter is_match_false_inter_l :
  ∀ (pt : portId) (pkt : packet) pat1 pat2,
    match_packet pt pkt pat1 = false →
    match_packet pt pkt (inter pat1 pat2) = false.

Parameter no_match_subset_r : ∀ k n t t',
  match_packet n k t' = false →
  match_packet n k (inter t t') = false.

Parameter exact_match_inter : ∀ x y,
  is_exact x = true →
  is_empty (inter x y) = false →
  inter x y === x.

Parameter all_spec : ∀ pt pk,
  match_packet pt pk all = true.

Parameter exact_match_is_exact : ∀ pk pt,
  is_exact (exact_pattern pk pt) = true.

Parameter exact_intersect : ∀ k n t,
  match_packet k n t = true →
  inter (exact_pattern n k) t === exact_pattern n k.

Parameter is_match_true_inter : ∀ pat1 pat2 pt pk,
  match_packet pt pk pat1 = true →
  match_packet pt pk pat2 = true →
  match_packet pt pk (inter pat1 pat2) = true.

Parameter match_packet_spec : ∀ pt pk pat,
```

*match_packet pt pk pat* =

*negb* (*is_empty* (*inter* (*exact_pattern pk pt*) *pat*)).

`Parameter` *all_is_not_empty* : *is_empty all* = *false*.

`End` *PATTERN*.

### A.2.53 Theory Library

`Set Implicit Arguments`.

`Require Import` *Coq.Arith.EqNat*.

`Require Import` *NPeano*.

`Require Import` *Arith.Peano_dec*.

`Require Import` *Bool.Bool*.

`Require Import` *Coq.Classes.Equivalence*.

`Require Import` *Lists.List*.

`Require Import` *Word.WordInterface*.

`Require Import` *Network.NetworkPacket*.

`Require Import` *Common.Types*.

`Require Import` *Pattern.Pattern*.

`Require Import` *Pattern.Valid*.

`Require Import` *Wildcard.Wildcard*.

`Require Import` *Wildcard.Theory*.

`Open Scope` *bool_scope*.

`Open Scope` *list_scope*.

`Open Scope` *equiv_scope*.

*Create HintDb* `pattern`.

```
Lemma IP_ARP_frametyp_neq : Const_0x800 ≠ Const_0x806.
Proof with auto.
  assert ({ Const_0x800 = Const_0x806 } + { Const_0x800 ≠ Const_0x806 }).
  apply Word16.eq_dec.
  unfold Const_0x800 in *.
  unfold Const_0x806 in *.
  destruct H.
  inversion e.
  trivial.
Qed.

Module PatMatchable.
  Definition t := pattern.

  Definition inter := Pattern.inter.
  Definition empty := Pattern.empty.
  Definition is_empty := Pattern.is_empty.
  Definition is_exact := Pattern.is_exact.

  Hint Unfold inter empty is_empty
    Pattern.inter Pattern.empty Pattern.is_empty.

  Hint Resolve Word8.eq_dec Word16.eq_dec Word32.eq_dec Word48.eq_dec.

  Lemma inter_comm : ∀ p p', inter p p' = inter p' p.
  Proof with auto.
    intros.
    destruct p.
    destruct p'.
    unfold inter.
    unfold Pattern.inter.
    simpl.
    rewrite → (inter_comm _ ptrnDlSrc0).
    rewrite → (inter_comm _ ptrnDlDst0).
    rewrite → (inter_comm _ ptrnDlType0).
    rewrite → (inter_comm _ ptrnDlVlan0).
    rewrite → (inter_comm _ ptrnDlVlanPcp0).
    rewrite → (inter_comm _ ptrnNwSrc0).
    rewrite → (inter_comm _ ptrnNwDst0).
    rewrite → (inter_comm _ ptrnNwTos0).
    rewrite → (inter_comm _ ptrnTpSrc0).
    rewrite → (inter_comm _ ptrnTpDst0).
    rewrite → (inter_comm _ ptrnInPort0).
    rewrite → (inter_comm _ ptrnNwProto0).
    reflexivity.
  Qed.

  Lemma inter_assoc : ∀ p p' p'',
    inter p (inter p' p'') = inter (inter p p') p''.
  Proof with simpl; auto.
    intros.
    unfold inter.
    unfold Pattern.inter.
    simpl.
    repeat rewrite → inter_assoc...
  Qed.
```

```
Lemma is_empty_false_distr_l : ∀ x y,
  is_empty (inter x y) = false →
  is_empty x = false .
Proof with simpl; eauto.
  intros.
  unfold inter in H.
  unfold Pattern.inter in H.
  simpl in H.
  repeat rewrite → orb_false_iff in H.
  do 11 (destruct H).
  unfold is_empty.
  unfold Pattern.is_empty.
  destruct x.
  destruct y.
  simpl in *.
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  erewrite → is_empty_false_distr_l; [ idtac | eauto ].
  reflexivity.
Qed.

Lemma is_empty_false_distr_r : ∀ x y,
  is_empty (inter x y) = false →
  is_empty y = false .
Proof.
  intros.
  rewrite → inter_comm in H.
  eapply is_empty_false_distr_l.
  exact H.
Qed.

Lemma is_empty_true_l : ∀ x y,
  is_empty x = true →
  is_empty (inter x y) = true.
Proof with auto.
  intros.
  destruct x.
  destruct y.
  simpl in *.
  repeat rewrite → orb_true_iff in H.
  repeat rewrite → or_assoc in H.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  repeat (destruct H; auto 13 with wildcard).
```

```
Qed.

Lemma is_empty_true_r : ∀ x y,
  is_empty y = true →
  is_empty (inter x y) = true.
Proof with auto.
  intros.
  rewrite inter_comm.
  apply is_empty_true_l...
Qed.

Lemma is_match_false_inter_l :
  ∀ (pt : portId) (pkt : packet) pat1 pat2,
    Pattern.match_packet pt pkt pat1 = false →
    Pattern.match_packet pt pkt (inter pat1 pat2) = false.
Proof with auto.
  intros.
  unfold Pattern.match_packet in *.
  rewrite → negb_false_iff in H.
  rewrite → negb_false_iff.
  rewrite → inter_assoc.
  apply is_empty_true_l...
Qed.

Lemma no_match_subset_r : ∀ k n t t',
  Pattern.match_packet n k t' = false →
  Pattern.match_packet n k (inter t t') = false.
Proof with auto.
```

```
  intros.
  rewrite → inter_comm.
  apply is_match_false_inter_l...
Qed.

Lemma exact_match_inter : ∀ x y,
  is_exact x = true →
  is_empty (inter x y) = false →
  inter x y = x.
Proof with auto.
  intros.
  destruct x. destruct y. simpl in *.
  repeat rewrite → andb_true_iff in H.
  do 11 (destruct H).
  repeat rewrite → orb_false_iff in H0.
  do 11 (destruct H0).
  unfold inter.
  unfold Pattern.inter.
  simpl.
  repeat rewrite → exact_match_inter_l...
Qed.

Lemma all_spec : ∀ pt pk,
  Pattern.match_packet pt pk Pattern.all = true.
Proof with auto.
  intros.
  unfold Pattern.match_packet.
  rewrite → negb_true_iff.
  unfold Pattern.all.
  destruct pk.
  unfold Pattern.exact_pattern.
  unfold Pattern.inter.
  unfold Pattern.Wildcard_of_option in *.
  destruct pktTpSrc; destruct pktTpDst; simpl...
Qed.

Lemma exact_match_is_exact : ∀ pk pt,
  Pattern.is_exact (Pattern.exact_pattern pk pt) = true.
Proof with auto.
  intros.
  unfold Pattern.exact_pattern.
  unfold Pattern.is_exact.
  unfold Wildcard.is_exact.
  simpl.
  unfold Pattern.Wildcard_of_option.
  simpl.
  destruct (pktTpSrc pk); destruct (pktTpDst pk)...
Qed.

Lemma exact_intersect : ∀ k n t,
  Pattern.match_packet k n t = true →
  Pattern.inter (Pattern.exact_pattern n k) t = Pattern.exact_pattern n k.
Proof with auto.
  intros.
  unfold Pattern.match_packet in H.
  rewrite → negb_true_iff in H.
  apply exact_match_inter...
Qed.

Lemma is_match_true_inter : ∀ pat1 pat2 pt pk,
  Pattern.match_packet pt pk pat1 = true →
  Pattern.match_packet pt pk pat2 = true →
  Pattern.match_packet pt pk (Pattern.inter pat1 pat2) = true.
Proof with auto.
  intros.
  apply exact_intersect in H.
  apply exact_intersect in H0.
  unfold Pattern.match_packet.
  rewrite → negb_true_iff.
  rewrite → inter_assoc.
  unfold inter.
  rewrite → H.
  rewrite → H0.
  unfold Pattern.exact_pattern.
  unfold Pattern.is_empty.
  simpl.
  unfold Pattern.Wildcard_of_option.
  destruct (pktTpSrc pk); destruct (pktTpDst pk)...
Qed.

Hint Rewrite inter_assoc : pattern.
Hint Resolve is_empty_false_distr_l is_empty_false_distr_r : pattern.

Hint Resolve is_empty_true_l is_empty_true_r : pattern.

Hint Resolve is_match_false_inter_l : pattern.

Hint Resolve all_spec : pattern.

Hint Resolve exact_match_is_exact : pattern.

Hint Resolve exact_intersect : pattern.

Hint Resolve is_match_true_inter : pattern.

Hint Resolve is_empty_true_l is_empty_true_r.

Hint Constructors ValidPattern.

Lemma pres0 : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwProto' nwTos tpSrc tpDst inPort,
  nwProto ≠ nwProto' →
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                         (Wildcard.inter Word8.eq_dec
                              (WildcardExact nwProto)
                              (WildcardExact nwProto'))
                         nwTos tpSrc tpDst inPort).
Proof with auto with wildcard.
  intros.
  apply ValidPat_None.
  simpl.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  rewrite → inter_exact_neq...
  simpl.
  auto 13.
Qed.

Hint Immediate pres0.

Lemma pres1 : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwProto' nwTos tpSrc tpDst inPort,
  In nwProto SupportedNwProto →
  ¬ In nwProto' SupportedNwProto →
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                        (Wildcard.inter Word8.eq_dec
                            (WildcardExact nwProto)
                            (WildcardExact nwProto'))
                        nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word8.eq_dec nwProto nwProto').
  destruct X; subst...
Qed.

Hint Immediate pres1.

Lemma pres1' : ∀ dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwProto' nwTos tpSrc tpDst inPort,
  ¬ In nwProto SupportedNwProto →
  In nwProto' SupportedNwProto →
  ValidPattern (Pattern dlSrc dlDst dlTyp dlVlan dlVlanPcp nwSrc nwDst
                        (Wildcard.inter Word8.eq_dec
                            (WildcardExact nwProto)
```

```
                    (WildcardExact nwProto'))
                  nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word8.eq_dec nwProto nwProto').
  destruct X; subst...
Qed.

Hint Immediate pres1'.

Lemma pres2 : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  dlTyp ≠ dlTyp' →
  ValidPattern (Pattern dlSrc dlDst
                (Wildcard.inter Word16.eq_dec
                  (WildcardExact dlTyp)
                  (WildcardExact dlTyp'))
                dlVlan dlVlanPcp nwSrc nwDst nwProto
                nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply ValidPat_None.
  simpl.
  repeat rewrite → orb_true_iff.
  repeat rewrite → or_assoc.
  rewrite → inter_exact_neq; auto 13.
Qed.
```

```
Hint Immediate pres2.

Lemma pres3 : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  In dlTyp SupportedDlTyp →
  ¬ In dlTyp' SupportedDlTyp →
  ValidPattern (Pattern dlSrc dlDst
                (Wildcard.inter Word16.eq_dec
                  (WildcardExact dlTyp)
                  (WildcardExact dlTyp'))
                dlVlan dlVlanPcp nwSrc nwDst nwProto
                nwTos tpSrc tpDst inPort).
  Proof with auto.
    intros.
    pose (X := Word16.eq_dec dlTyp dlTyp').
    destruct X; subst...
  Qed.
  Hint Resolve pres3.

Lemma pres3' : ∀ dlSrc dlDst dlTyp dlTyp' dlVlan dlVlanPcp nwSrc nwDst
  nwProto nwTos tpSrc tpDst inPort,
  ¬ In dlTyp SupportedDlTyp →
  In dlTyp' SupportedDlTyp →
  ValidPattern (Pattern dlSrc dlDst
                (Wildcard.inter Word16.eq_dec
                  (WildcardExact dlTyp)
                  (WildcardExact dlTyp'))
```

```
                    dlVlan dlVlanPcp nwSrc nwDst nwProto
                    nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  pose (X := Word16.eq_dec dlTyp dlTyp').
  destruct X; subst...
Qed.

Hint Resolve pres3'.

Lemma pres4 : In Const_0x800 SupportedDlTyp.
Proof with auto with datatypes.
  intros.
  unfold SupportedDlTyp...
Qed.

Hint Resolve pres4.

Lemma pres4' : In Const_0x806 SupportedDlTyp.
Proof with auto with datatypes.
  intros.
  unfold SupportedDlTyp...
Qed.

Hint Resolve pres4'.

Lemma pres5 : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                          nwProto nwTos tpSrc tpDst inPort,
ValidPattern (Pattern dlSrc dlDst
                    (Wildcard.inter Word16.eq_dec
                      (WildcardExact Const_0x800)
```

```
                         (WildcardExact Const_0x806))
                       dlVlan dlVlanPcp nwSrc nwDst nwProto
                       nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply pres2.
  exact IP_ARP_frametyp_neq.
Qed.

Hint Immediate pres5.

Lemma pres5' : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                             nwProto nwTos tpSrc tpDst inPort,
 ValidPattern (Pattern dlSrc dlDst
                       (Wildcard.inter Word16.eq_dec
                         (WildcardExact Const_0x806)
                         (WildcardExact Const_0x800))
                       dlVlan dlVlanPcp nwSrc nwDst nwProto
                       nwTos tpSrc tpDst inPort).
Proof with auto.
  intros.
  apply pres2.
  unfold not.
  intros.
  symmetry in H.
  apply IP_ARP_frametyp_neq...
Qed.
```

```
  Hint Immediate pres5'.

  Lemma empty_valid_l : ∀ pat pat',
    is_empty pat = true →
    ValidPattern (Pattern.inter pat pat').
  Proof with auto.
    intros.
    apply ValidPat_None.
    apply is_empty_true_l...
  Qed.

  Lemma empty_valid_r : ∀ pat pat',
    is_empty pat' = true →
    ValidPattern (Pattern.inter pat pat').
  Proof with auto.
    intros.
    apply ValidPat_None.
    apply is_empty_true_r...
  Qed.

Ltac inter_solve :=
  unfold Pattern.inter; simpl; autorewrite with wildcard using auto.

Lemma inter_preserves_valid : ∀ pat1 pat2,
  ValidPattern pat1 →
  ValidPattern pat2 →
  ValidPattern (Pattern.inter pat1 pat2).
Proof with auto.
  intros pat1 pat2 H H0.
```

```
destruct H; destruct H0;
  try solve [ auto using empty_valid_l, empty_valid_r ].
pose (X := Word8.eq_dec nwProto nwProto0); destruct X; subst; inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
pose (X := Word8.eq_dec nwProto nwProto0); destruct X; subst; inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
inter_solve.
```

```
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    pose (X := Word16.eq_dec frameTyp frameTyp0);
      destruct X; subst; inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
    inter_solve.
  Qed.

End PatMatchable.

Section Equivalence.

  Inductive Pattern_equiv : pattern → pattern → Prop :=
  | Pattern_equiv_match : ∀ pat1 pat2,
    (∀ pt pk,
      Pattern.match_packet pt pk pat1 = Pattern.match_packet pt pk pat2) →
    Pattern_equiv pat1 pat2.

  Hint Constructors Pattern_equiv.

  Lemma Pattern_equiv_is_Equivalence : Equivalence Pattern_equiv.

  Proof with auto.
```

```
    split.
    unfold Reflexive...
    unfold Symmetric. intros. inversion H...
    unfold Transitive. intros. inversion H. inversion H0. subst.
    split. intros. rewrite → H1...
  Qed.
End Equivalence.
Instance Pattern_Equivalance : Equivalence Pattern_equiv.
  apply Pattern_equiv_is_Equivalence.
Qed.
```

### A.2.54 Valid Pattern Library

```
Set Implicit Arguments.
Require Import Coq.Arith.EqNat.
Require Import NPeano.
Require Import Arith.Peano_dec.
Require Import Bool.Bool.
Require Import Coq.Classes.Equivalence.
Require Import Lists.List.
Require Import PArith.BinPos.
Require Import Word.WordInterface.
Require Import Network.NetworkPacket.
Require Import Common.Types.
Require Import Pattern.Pattern.
Require Import Wildcard.Wildcard.

Require Import Wildcard.Theory.

Open Scope bool_scope.

Open Scope list_scope.

Open Scope equiv_scope.

Open Scope positive_scope.

Definition SupportedNwProto :=
  [ Const_0x6;
    Const_0x7;
    Const_0x1 ].

Definition SupportedDlTyp :=
    [ Const_0x800; Const_0x806 ].
```

Based on the flow chart on Page 8 of OpenFlow 1.0 specification. In a ValidPattern, all exact-match fields are used to match packets.

```
Inductive ValidPattern : pattern → Prop :=
| ValidPat_TCPUDP : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                                       nwTos tpSrc tpDst inPort nwProto,
    In nwProto SupportedNwProto →
    ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
                               dlVlan dlVlanPcp
                               nwSrc nwDst (WildcardExact nwProto)
                               nwTos tpSrc tpDst inPort)
| ValidPat_ARP : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                             inPort,
    ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x806)
                              dlVlan dlVlanPcp
                              nwSrc nwDst WildcardAll
                              WildcardAll WildcardAll WildcardAll inPort)
| ValidPat_IP_other : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                               nwTos inPort nwProto,
    ¬ In nwProto SupportedNwProto →
    ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
                              dlVlan dlVlanPcp
                              nwSrc nwDst (WildcardExact nwProto)
                              nwTos WildcardAll WildcardAll inPort)
| ValidPat_IP_NoNwProto : ∀ dlSrc dlDst dlVlan dlVlanPcp nwSrc nwDst
                               nwTos inPort,
    ValidPattern (Pattern dlSrc dlDst (WildcardExact Const_0x800)
                              dlVlan dlVlanPcp
                              nwSrc nwDst WildcardAll
                              nwTos WildcardAll WildcardAll inPort)
| ValidPat_OtherFrameTyp : ∀ dlSrc dlDst dlVlan dlVlanPcp
                                   inPort frameTyp,
    ¬ In frameTyp SupportedDlTyp →
    ValidPattern (Pattern dlSrc dlDst (WildcardExact frameTyp)
                              dlVlan dlVlanPcp
                              WildcardAll WildcardAll WildcardAll
                              WildcardAll WildcardAll WildcardAll inPort)
| ValidPat_AnyFrameTyp : ∀ dlSrc dlDst dlVlan dlVlanPcp
                                   inPort,
    ValidPattern (Pattern dlSrc dlDst WildcardAll
                                   dlVlan dlVlanPcp
                                   WildcardAll WildcardAll WildcardAll
                                   WildcardAll WildcardAll WildcardAll inPort)
| ValidPat_None : ∀ pat,
   Pattern.is_empty pat = true →
   ValidPattern pat.
```

### A.2.55 Theory Library

```
Set Implicit Arguments.

Require Import Coq.Structures.Equalities.

Require Import Common.Types.

Require Import Wildcard.Wildcard.

Import Wildcard.

Create HintDb wildcard.

Section Lemmas.

  Variable A : Type.

  Variable eq_dec : (∀ (x y : A), { x = y } + { x ≠ y }).

  Hint Unfold inter inter is_empty is_exact.

  Hint Constructors Wildcard.

  Ltac destruct_eq x y :=
    destruct (eq_dec x y).

  Lemma inter_all_r : ∀ x, inter eq_dec x WildcardAll = x.
```

```
Proof with auto with wildcard.
  intros.
  destruct x...
Qed.

Lemma inter_all_l : ∀ x, inter eq_dec WildcardAll x = x.
Proof with auto with wildcard.
  intros.
  destruct x...
Qed.

Lemma inter_exact_eq : ∀ v,
  inter eq_dec (WildcardExact v) (WildcardExact v) = WildcardExact v.
Proof with auto.
  intros.
  autounfold.
  destruct_eq v v...
  unfold not in n.
  contradiction (n eq_refl).
Qed.

Lemma inter_exact_neq : ∀ v v',
  v ≠ v' →
  inter eq_dec (WildcardExact v) (WildcardExact v') = WildcardNone.
Proof with auto.
  intros.
  autounfold.
  destruct_eq v v'...
  contradiction.
Qed.

Lemma inter_none_r : ∀ x,
  inter eq_dec x WildcardNone = WildcardNone.
Proof.
  intros.
  destruct x; auto.
Qed.

Lemma inter_none_l : ∀ y,
  inter eq_dec WildcardNone y = WildcardNone.
Proof.
  intros.
  destruct y; auto.
Qed.

Hint Rewrite inter_all_l inter_all_r.
Hint Rewrite inter_exact_neq inter_exact_eq.
Hint Rewrite inter_none_l inter_none_r.

Lemma inter_comm : ∀ x y,
  inter eq_dec x y = inter eq_dec y x.
Proof with auto.
  intros.
  destruct x; destruct y...
  destruct_eq a a0; subst;
    autorewrite with core using (subst;auto)...
Qed.
```

```
Lemma inter_assoc : ∀ x y z,
  inter eq_dec x (inter eq_dec y z) = inter eq_dec (inter eq_dec x y) z.
Proof with auto.
  intros.
  destruct x; destruct y; destruct z...
  destruct_eq a a0; destruct_eq a a1; destruct_eq a0 a1; subst;
    autorewrite with core using (subst;auto)...
  destruct_eq a a0; subst; autorewrite with core using (subst;auto)...
  destruct_eq a a0; subst; autorewrite with core using (subst;auto)...
  destruct_eq a a0; subst; autorewrite with core using (subst;auto)...
  destruct_eq a a0; subst; autorewrite with core using (subst;auto)...
Qed.

Definition is_empty_false_distr_l : ∀ x y,
  is_empty (inter eq_dec x y) = false →
  is_empty x = false.
Proof with auto.
  intros.
  destruct x; destruct y...
Qed.

Definition is_empty_false_distr_r : ∀ x y,
  is_empty (inter eq_dec x y) = false →
  is_empty y = false.
Proof with auto.
  intros.
  destruct x; destruct y...
```

```
Qed.

Lemma exact_match_inter_l : ∀ x y,
  is_exact x = true →
  is_empty (inter eq_dec x y) = false →
  inter eq_dec x y = x.
Proof with auto.
  intros.
  destruct x; simpl in H; try solve [ inversion H ].
  destruct y; simpl in H0; try solve [ inversion H0 ].
  autounfold in *.
  remember (eq_dec a a0) as H1.
  destruct H1...
  inversion H0.
  intuition.
Qed.

Lemma exact_match_inter_r : ∀ x y,
  is_exact y = true →
  is_empty (inter eq_dec x y) = false →
  inter eq_dec x y = y.
Proof with auto.
  intros.
  rewrite → inter_comm in *...
  apply exact_match_inter_l...
Qed.

Lemma is_empty_true_l : ∀ x y,
  is_empty x = true →
  is_empty (inter eq_dec x y) = true.
Proof with auto with wildcard.
  intros.
  destruct x...
  inversion H.
  inversion H.
  autorewrite with core using (subst;auto)...
Qed.

Lemma is_empty_true_r : ∀ x y,
  is_empty y = true →
  is_empty (inter eq_dec x y) = true.
Proof with auto.
  intros.
  rewrite → inter_comm.
  apply is_empty_true_l...
Qed.

Lemma is_exact_split_l :
  ∀ f (x : A) (y : Wildcard A),
    inter f (WildcardExact x) y = WildcardExact x ∨
    inter f (WildcardExact x) y = WildcardNone.
Proof with auto.
  intros.
  destruct y...
  unfold inter.
```

```
    destruct (f x a)...
  Qed.

  Lemma is_exact_split_r :
    ∀ f (y : Wildcard A) (x : A),
      inter f y (WildcardExact x) = WildcardExact x ∨
      inter f y (WildcardExact x) = WildcardNone.
  Proof with auto.
    intros.
    destruct y...
    unfold inter.
    destruct (f a x)...
    subst...
  Qed.

End Lemmas.

Hint Rewrite inter_all_l inter_all_r : wildcard.

Hint Rewrite inter_exact_eq : wildcard.

Hint Rewrite inter_none_l inter_none_r : wildcard.

Hint Rewrite inter_assoc : wildcard.

Hint Rewrite is_empty_false_distr_l is_empty_false_distr_r : wildcard.

Hint Resolve exact_match_inter_l exact_match_inter_r : wildcard.

Hint Resolve inter_exact_neq is_empty_true_l is_empty_true_r : wildcard.
```

### A.2.56 Wildcard Library

```
Set Implicit Arguments.
```

```
Require Import Common.Types.
```

We use Wildcards to build Patterns. When a pattern field is not present, we set it to *WildcardExact* 0. An alternative design is to use have a *WildcardNotPresent* variant. However, this complicates the definition of *ValidPattern*. We have to state that most fields are not *WildcardNotPresent*. `Inductive` *Wildcard* ($A$ : `Type`) : `Type` :=

```
| WildcardExact : A → Wildcard A
| WildcardAll : Wildcard A
| WildcardNone : Wildcard A.

Implicit Arguments WildcardAll [[A]].
Implicit Arguments WildcardNone [[A]].

Module Wildcard.

  Definition inter {A : Type} (eqdec : Eqdec A) (x y : Wildcard A) :=
    match (x, y) with
      | (_, WildcardNone) ⇒ WildcardNone
      | (WildcardNone, _) ⇒ WildcardNone
      | (WildcardAll, _) ⇒ y
      | (_, WildcardAll) ⇒ x
      | (WildcardExact m, WildcardExact n) ⇒
        if eqdec m n then WildcardExact m else WildcardNone
    end.

  Definition is_all {A : Type} (w : Wildcard A) :=
    match w with
      | WildcardAll ⇒ true
      | _ ⇒ false
    end.

  Definition is_empty {A : Type} (w : Wildcard A) :=
    match w with
      | WildcardNone ⇒ true
      | _ ⇒ false
    end.

  Definition is_exact {A : Type} (w : Wildcard A) :=
    match w with
      | WildcardExact _ ⇒ true
      | _ ⇒ false
    end.

  Lemma eq_dec : ∀ {A : Type} (eqdec : Eqdec A) (x y : Wildcard A),
    { x = y } + { x ≠ y }.
  Proof with auto.
    decide equality.
  Defined.

  Definition to_option (A : Type) (w : Wildcard A) :=
    match w as w0 return (w0 ≠ WildcardNone → option A) with
      | WildcardExact a ⇒ fun _ ⇒ Some a
      | WildcardAll ⇒ fun _ ⇒ None
      | WildcardNone ⇒ fun not_null ⇒ False_rect _ (not_null eq_refl)
    end.

End Wildcard.
```

### A.2.57 WordInterface Library

Set Implicit Arguments.

Require Import *Coq.Logic.ProofIrrelevance*.

Require Import *Coq.Structures.Equalities*.

Require Import *PArith.BinPos*.

Require Import *NArith.BinNat*.

Local Open Scope *N_scope*.

Module Type *WORD* <: *MiniDecidableType*.

Opaque representation of the word  Parameter $t$ : Type.

bit-width  Parameter *width* : *positive*.

Parameter *eq_dec* : $\forall$ ($m$ $n$ : $t$), { $m = n$ } + { $m \neq n$ }.

Parameter *zero* : $t$.

End *WORD*.

Unset *Elimination Schemes*.

Module Type *WIDTH*.

Parameter *width* : *positive*.

End *WIDTH*.

Module Type *MAKEWORD*.

Local Open Scope *N_scope*.

Parameter *width* : *positive*.

Inductive *Word* : Type :=

```
  | Mk : ∀ (v : N), v < 2 ^ N.pos width → Word.

  Definition t := Word.

End MAKEWORD.

Module MakeWord (Width : WIDTH).

  Local Open Scope N_scope.

  Definition width := Width.width.

  Inductive Word : Type :=
  | Mk : ∀ (v : N), v < 2 ^ N.pos width → Word.

  Definition t := Word.

  Definition zero : t := @Mk 0 eq_refl.

End MakeWord.

Module Width8 <: WIDTH.

  Definition width := 8 %positive.

End Width8.

Module Width12 <: WIDTH.

  Definition width := 12 %positive.

End Width12.

Module Width16 <: WIDTH.

  Definition width := 16 %positive.

End Width16.

Module Width32 <: WIDTH.

  Definition width := 32 %positive.
```

```
End Width32.

Module Width48 <: WIDTH.

  Definition width := 48 %positive.

End Width48.

Module Width64 <: WIDTH.

  Definition width := 64 %positive.

End Width64.
```

Semantically, this module is equivalent to:

Module Word8 := MakeWord (Width8).

However, the more elaborate definition below allows us to extract words of different widths to different OCaml types. `Module Word8 <: WORD.`

```
  Module M := MakeWord (Width8).
  Include M.
  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

End Word8.

Module Word12 <: WORD.

  Module M := MakeWord (Width12).
  Include M.
  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

End Word12.

Module Word16 <: WORD.

  Module M := MakeWord (Width16).
  Include M.

  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

  Definition to_nat (w : Word) : nat :=

    match w with

      | Mk n _ ⇒ N.to_nat n

    end.

  Definition max_value : t := @Mk 65535 eq_refl.

  Axiom pred : Word → Word.

End Word16.

Module Word32 <: WORD.

  Module M := MakeWord (Width32).

  Include M.

  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

End Word32.

Module Word48 <: WORD.

  Module M := MakeWord (Width48).

  Include M.

  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

End Word48.

Module Word64 <: WORD.

  Module M := MakeWord (Width64).

  Include M.

  Parameter eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.

End Word64.
```

`Extract` *Constant* *Word16.pred* ⇒ "(fun n -> if n = 0 then 0 else n - 1)". `Extract` *Constant* *Word16.max_value* ⇒ "65535".

`Extract Inductive` *Word8.Word* ⇒ "int" [ "" ].

`Extract Inductive` *Word12.Word* ⇒ "int" [ "" ].

`Extract Inductive` *Word16.Word* ⇒ "int" [ "" ].

`Extract Inductive` *Word32.Word* ⇒ "int32" [ "" ].

`Extract Inductive` *Word48.Word* ⇒ "int64" [ "" ].

`Extract Inductive` *Word64.Word* ⇒ "int64" [ "" ].

`Extract` *Constant* *Word8.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word12.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word16.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word32.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word48.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word64.eq_dec* ⇒ "(=)".

`Extract` *Constant* *Word8.zero* ⇒ "0".

`Extract` *Constant* *Word12.zero* ⇒ "0".

`Extract` *Constant* *Word16.zero* ⇒ "0".

`Extract` *Constant* *Word32.zero* ⇒ "Int32.zero".

`Extract` *Constant* *Word48.zero* ⇒ "Int64.zero".

`Extract` *Constant* *Word64.zero* ⇒ "Int64.zero".

`Extract` *Constant* *Word16.to_nat* ⇒ "(fun x -> x)".

### A.2.58 WordTheory Library

```
Set Implicit Arguments.
```

```
Require Import Coq.Logic.ProofIrrelevance.
Require Import Coq.Structures.Equalities.
Require Import PArith.BinPos.
Require Import NArith.BinNat.
Require Import Bag.TotalOrder.
Require Import Word.WordInterface.

Local Open Scope N_scope.

Module Make (Import W : MAKEWORD).

  Module Export Word := W.

  Lemma eq_dec : ∀ (m n : t), { m = n } + { m ≠ n }.
  Proof.
    intros.
    destruct m, n.
    assert ({v = v0} + {v ≠ v0}) as J.
    { apply N.eq_dec. }
    destruct J; subst.
    + left.
      f_equal.
      apply proof_irrelevance.
    + right.
      unfold not.
      intros.
      inversion H; subst.
      contradiction n.
      reflexivity.
```

```
Qed.

Definition le (x y : t) : Prop :=
  match (x, y) with
    | (Mk m _, Mk n _) ⇒ m ≤ n
  end.

Lemma le_reflexivity : ∀ (x : t), le x x.
Proof with auto.
  intros.
  destruct x...
  apply N.le_refl.
Qed.

Lemma le_antisymmetry : ∀ (x y : t), le x y → le y x → x = y.
Proof with eauto.
  intros.
  destruct x, y.
  unfold le in *.
  apply N.lt_eq_cases in H.
  apply N.lt_eq_cases in H0.
  destruct H, H0...
  + assert (v < v). eapply N.lt_trans...
    apply N.lt_irrefl in H1. inversion H1.
  + subst...
    assert (l = l0). apply proof_irrelevance.
    subst...
  + subst...
```

```
    assert (l = l0). apply proof_irrelevance.
    subst...
  + clear H0; subst.
    assert (l = l0). apply proof_irrelevance.
    subst...
Qed.
Lemma le_transitivity : ∀ (x y z : t), le x y → le y z → le x z.
Proof with eauto.
  intros.
  destruct x, y, z.
  unfold le in *.
  eapply N.le_trans...
Qed.
Lemma le_compare : ∀ (x y : t), { le x y } + { le y x }.
Proof with eauto.
  intros.
  destruct x, y.
  unfold le in *.
  remember (N.compare v v0) as cmp eqn:J.
  symmetry in J.
  destruct cmp.
  + apply N.compare_eq_iff in J.
    subst.
    left...
    apply N.lt_eq_cases...
    + rewrite → N.compare_lt_iff in J.
      left.
      apply N.lt_eq_cases...
    + rewrite → N.compare_gt_iff in J.
      right.
      apply N.lt_eq_cases...
  Qed.

  Instance TotalOrder : TotalOrder le := {
    reflexivity := le_reflexivity;
    antisymmetry := le_antisymmetry;
    transitivity := le_transitivity;
    compare := le_compare;
    eqdec := eq_dec
  }.

End Make.

Module Word8 := Make (Word8).
Module Word12 := Make (Word12).
Module Word16 := Make (Word16).
Module Word32 := Make (Word32).
Module Word48 := Make (Word48).
Module Word64 := Make (Word64).

Existing Instances Word8.TotalOrder Word16.TotalOrder Word12.TotalOrder
  Word32.TotalOrder Word48.TotalOrder Word64.TotalOrder.
```

## A.3 Proofs for Chapter 5

The theorems of this chapter have been formally verified in the Coq theorem prover. We include the proof text here. All proofs have been completed in Coq verion 8.4.

### A.3.1 Packet Library

```
Require Import Bool.Bool.

Require Import Classes.EquivDec.

Require Import Network.

Section packet.

  Program Instance port_eq_eqdec : EqDec port eq := eq_port_dec.

  Program Instance host_eq_eqdec : EqDec host eq := eq_host_dec.

  Inductive packet : Type :=

  | Packet : host → host → option nat → nat → packet.

  Inductive packet_field :=

  | Src : packet_field

  | Dst : packet_field

  | Ver : packet_field.

  Definition set_field pk fld (val : nat) :=

    match pk with

      Packet src dst ver data ⇒

      match fld with

        | Src ⇒ Packet val dst ver data

        | Dst ⇒ Packet src val ver data
        | Ver ⇒ Packet src dst (Some val) data
      end
    end.
Definition strip_vlan pk :=
  match pk with
    | Packet src dst _ data ⇒ Packet src dst None data
  end.
Lemma eq_packet_dec : ∀ (p1 p2 : packet), {p1 = p2} + {p1 ≠ p2}.
Proof.
  repeat decide equality.
Defined.
Definition packet_src pk :=
  match pk with
    | Packet src _ _ _ ⇒ src
  end.
Definition packet_dst pk :=
  match pk with
    | Packet _ dst _ _ ⇒ dst
  end.
Inductive pattern : Type :=
| src_patt : host → pattern
| dst_patt : host → pattern
| inport_patt : port → pattern
| and_patt : pattern → pattern → pattern.
Fixpoint packet_in_pattern pk port patt : bool :=
  match patt with
    | src_patt src ⇒ packet_src pk ==b src
    | dst_patt dst ⇒ packet_dst pk ==b dst
    | inport_patt port' ⇒ port ==b port'
    | and_patt patt1 patt2 ⇒ packet_in_pattern pk port patt1 && packet_in_pattern pk port
patt2
  end.
End packet.
```

### A.3.2 OpenFlow Library

```
Inductive action : Type :=
| Controller : port → action
| Forward : port → action
| Mod : packet_field → nat → action
| Strip_vlan : action.
Definition classifier := list (pattern × list action).
Inductive command : Type :=
| Update : classifier → command
| Send : packet → list action → command.
Inductive controller_event : Type :=
| Packet_in : packet → switch → port → controller_event.
Inductive network_event : Type :=
| Packet_recv : packet → link → network_event
| Packet_send : packet → link → network_event.
```

Definition trace := **list network_event**.

Notation " s # n " :=

  (Link $s$ $n$) (at level 0).

Let *packetQueue* := **list packet**.

Let *emptyPacketQueue* : packetQueue := [].

Let *eventQueue* := **list controller_event**.

Let *emptyEventQueue* : eventQueue := [].

Let *commandQueue* := **list command**.

Let *emptyCommandQueue* : commandQueue := [].

Definition switchState := **switch** → classifier.

Definition emptySwitchState : switchState := fun _ ⇒ [].

Definition linkState := **link** → (packetQueue×packetQueue).

Definition emptyLinkState : linkState := fun _ ⇒ (emptyPacketQueue, emptyPacketQueue).

Definition topology := **link** → **option link**.

Definition commandQueues := **switch** → commandQueue.

Definition emptyCommandQueues : commandQueues := fun _ ⇒ emptyCommandQueue.

Definition emptyClassifier : classifier := [].

Module Type OPENFLOW_STATES.

  Parameter *CState* : Type.

  Parameter *controllerProgramStep* : *CState* → eventQueue → *CState* → eventQueue → commandQueues → Prop.

  Parameter *HState* : Type.

  Parameter *emptyHState* : *HState*.

  Parameter *hostProgramRecvStep* : *HState* → **packet** → *HState* → Prop.

```
  Parameter hostProgramSendStep : HState → HState → packet → Prop.
End OPENFLOW_STATES.

Module OPENFLOW_SEMANTICS (M : OPENFLOW_STATES).

  Import M.

  Parameter T : topology.

  Definition hostState := host → HState.

  Definition emptyHostState : hostState := fun _ ⇒ emptyHState.

  Inductive system : Type :=
  | System : CState → eventQueue → commandQueues → switchState → hostState →
linkState → system.

  Notation " cs , eq , cq , ss , hs , ls " :=
    (System cs eq cq ss hs ls).

  Inductive interpretClassifier : classifier → packet → port → list action → Prop :=
  | MatchClassifier : ∀ pk p pat cs a,
    packet_in_pattern pk p pat = true →
    interpretClassifier ((pat, a) :: cs) pk p a
  | NoMatchClassifier : ∀ pk p pat cs a a',
    packet_in_pattern pk p pat = false →
    interpretClassifier cs pk p a' →
    interpretClassifier ((pat, a) :: cs) pk p a'.

  Fixpoint eval' (pEvents : list network_event) (evq:eventQueue) (ls: linkState) (sw:switch)
(acts:list action) (p:packet) :=
    match acts with
      | [] ⇒
      (evq, ls, pEvents)
    | (Controller n) :: acts ⇒
      let evq' := (Packet_in p sw n) :: evq in
        eval' pEvents evq' ls sw acts p
    | (Forward l) :: acts ⇒
      let ls' := ls  sw#l |-> (fst (ls sw#l), p :: snd (ls sw#l))  in
        eval' ((Packet_send p sw#l) :: pEvents) evq ls' sw acts p
    | Mod fld val :: acts ⇒
      eval' pEvents evq ls sw acts (set_field p fld val)
    | Strip_vlan :: acts ⇒
      eval' pEvents evq ls sw acts (strip_vlan p)
  end.

Definition eval := eval' [] [] emptyLinkState.

Inductive step : system → trace → system → Prop :=
| ControllerStep :
  ∀ s eq swcq ss hs ls s' eq' swcq',
    controllerProgramStep s eq s' eq' swcq' →
    step
    s, eq, swcq, ss, hs, ls []


    s', eq', extend swcq swcq', ss, hs, ls
| UpdateStep :
  ∀ s evq swcq ss hs ls (sw : switch) C swcQs,
    swcQs sw = Update C :: swcq →
    step
    s, evq, swcQs, ss, hs, ls []
    s, evq, swcQs sw |-> swcq , ss sw |-> C , hs, ls
| SendStep :
  ∀ s evq swcq ss hs ls sw p acts evq' ls' pEv swcQs,
    swcQs sw = Send p acts::swcq →
    (evq',ls', pEv) = eval sw acts p →
    step
    s, evq, swcQs, ss, hs, ls pEv


    s, evq ++ evq', swcQs sw |-> swcq , ss, hs, extend2 ls ls'
| SwitchPacketStep :


  ∀ s evq swcq ss hs ls sw n p P ls' ls'' evq' pEv acts out,
    ls sw#n = (p::P, out) →
    ls' = (ls  sw#n |-> (P, out) ) →
    interpretClassifier (ss sw) p n acts →
    eval sw acts p = (evq', ls'', pEv) →
    step
    s, evq, swcq, ss, hs, ls (snoc (Packet_recv p sw#n) pEv)
    s, evq ++ evq', swcq, ss, hs, extend2 ls'' ls'
| LinkPacketStep :
  ∀ pk l l' ls ls' incoming outgoing incoming' outgoing' s evq swcq ss hs,
    ls l = (incoming, pk :: outgoing) →
    ls l' = (incoming', outgoing') →
    ls' = ls  l |-> (incoming, outgoing)  l' |-> (snoc pk incoming', outgoing')  →
    T l = Some l' →
    step
    s, evq, swcq, ss, hs, ls []
    s, evq, swcq, ss, hs, ls'
```

```
| HostSendStep :
  ∀ s evq swcq ss hs hs' ls h state state' p P outgoing,
    hs h = state →
    ls (H h) = (P, outgoing) →
    hs' = hs  h |-> state'  →
    hostProgramSendStep state state' p →
    step
    s, evq, swcq, ss, hs, ls [ Packet_send p (H h) ]
    s, evq, swcq, ss, hs', ls  H h |-> (p :: P, outgoing)
| HostRecvStep :
  ∀ s evq swcq ss hs hs' ls ls' (h:host) state state' p P incoming,
    let l := H h in


      ls l = (incoming, p::P) →
      hs h = state →
```

```
        hs  h |-> state'  = hs' →
        hostProgramRecvStep state p state' →
        ls  l |-> (incoming, P)  = ls' →
        step
        s, evq, swcq, ss, hs, ls [Packet_recv p l]
        s, evq, swcq, ss, hs', ls'.
  Inductive steps : system → trace → system → Prop :=
  | ReflSteps :
    ∀ S,
      steps
      S [] S
  | SingleSteps :
    ∀ S S' S'' lab1 lab2,
      step S lab1 S'' →
      steps S'' lab2 S' →
      steps S (lab2 ++ lab1) S'.
End OpenFlow_Semantics.
```

### A.3.3 Network Library

```
Require Import utilities.
Require Import List.
Require Import Classes.EquivDec.
Section Network.
  Inductive host : Type :=
| Host : nat → host.

Inductive port : Type :=

| Port : nat → port.

Inductive switch :=

| Switch : nat → switch

| World : switch

| Drop : switch.

Coercion Switch : nat >-> switch.

Coercion Port : nat >-> port.

Coercion hInj (h : nat) : host := Host h.

Inductive link :=

| Link : switch → port → link.

Definition host_map := host → link.

Definition graph := list (link × link).

Definition topology := (host_map, graph).

Definition path := list (link).

Parameter G : graph.

Parameter H : host_map.

Parameter H_inj : ∀ H1 H2, H H1 = H H2 → H1 = H2.

Parameter H_unique_ports : ∀ sw p h,

  ¬ In (H h, Link sw p) G ∧ ¬ In (Link sw p, H h) G.

Lemma eq_host_dec : ∀ h1 h2 : host, {h1=h2} + {h1≠h2}.

Proof.

  repeat decide equality.
```

```
Qed.

Lemma eq_port_dec : ∀ p1 p2 : port, {p1=p2} + {p1≠p2}.

Proof.

  repeat decide equality.

Qed.

Lemma eq_switch_dec : ∀ sw1 sw2 : switch, {sw1=sw2} + {sw1≠sw2}.

Proof.

  repeat decide equality.

Qed.

Lemma eq_link_dec : ∀ l1 l2 : link, {l1=l2} + {l1≠l2}.

Proof.

  repeat decide equality.

Qed.

Program Instance port_eq_eqdec : EqDec port eq := eq_port_dec.

Program Instance link_eq_eqdec : EqDec link eq := eq_link_dec.

Program Instance host_eq_eqdec : EqDec host eq := eq_host_dec.

Program Instance switch_eq_eqdec : EqDec switch eq := eq_switch_dec.

Inductive Legal_path : path → link → link → graph → Prop :=

| single_path_legal : ∀ s port1 port2 g,

  Legal_path [(Link s port1) ; (Link s port2)] (Link s port1) (Link s port2) g

| trans_path_legal : ∀ a b c d g p p',

  In (b, c) g →

  Legal_path p a b g →

  Legal_path p' c d g →

  Legal_path (p ++ p') a d g.
  Lemma Legal_path_non_empty :
    ∀ p n n' g,
      Legal_path p n n' g → p ≠ [].
  Proof.
    red in ⊢ ×.
    intros.
    induction H0; util_crush; apply app_eq_nil in H1; intuition.
  Qed.

  Definition reachable host1 host2 g := ∃ p, Legal_path p (H host1) (H host2) g.

  Inductive loop_free_path : path → Prop :=
  | empty_loop_free_path : loop_free_path []
  | unique_loop_free_path : ∀ (n : link) (p : path),
    loop_free_path p →
    not (In n p) →
    loop_free_path (n :: p).

End Network.
```

### A.3.4 ReachabilitySet Library

```
Require Import utilities.
Require Import List.
Require Import Relations.Relation_Definitions.
Require Import Datatypes.
Require Import Arith.EqNat.
Require Import Packet.
Require Import Network.

Section reachability_set.

  Definition reachabilitySet := list (host × host).

  Definition reachability_set_compatible (rs : reachabilitySet) g := ∀ h1 h2, In (h1 , h2) rs
→ reachable h1 h2 g.

  Inductive reachability_set_allows : reachabilitySet → packet → host → host → Prop
:=
  | reachability_set :
     ∀ rs h1 h2 pk,
        packet_src pk = h1 →
        packet_dst pk = h2 →
        In (h1 , h2) rs →
        reachability_set_allows rs pk h1 h2.
Notation " rs  pk h1 h2 " := (reachability_set_allows rs pk h1 h2) (at level 0).

End reachability_set.
```

### A.3.5 Updates Network Model Library

```
Require Import Arith.EqNat.

Require Import Arith.Compare_dec.

Require Import List.

Require Import Classes.EquivDec.

Require Import CpdtTactics.

Require Import utilities.

Notation "[ ]" := nil : list_scope.
Notation "[ a ; .. ; b ]" := (a :: .. (b :: []) ..) : list_scope.

Inductive packet : Type :=
| Packet : nat → nat → nat → nat → packet.

Inductive port : Type :=
| Port : nat → nat → port
| World
| Drop.

Definition locPkt := (port × packet) % type.

Definition trace := list locPkt.

Definition update := locPkt → option (list locPkt).

Definition switchFun := locPkt → list locPkt.

Definition topology := port → port.

Definition augmentedPkt := (packet × trace) % type.

Definition portQueue := port → list augmentedPkt.

Require Import Arith.EqNat.

Require Import Arith.Peano_dec.

Hint Rewrite beq_nat_refl : cpdt.

Lemma eq_locPkt_dec : ∀ (l1 l2 : locPkt), {l1 = l2} + {l1 ≠ l2}.

Proof.

  repeat decide equality.

Defined.

Lemma eq_augmentedPkt_dec : ∀ (a1 a2 : augmentedPkt), {a1 = a2} + {a1 ≠ a2}.

Proof.
```

```
  repeat decide equality.
Defined.

Lemma port_eq_dec : ∀ p1 p2 : port, {p1=p2} + {p1≠p2}.
Proof.
  repeat decide equality.
Qed.

Lemma packet_eq_dec : ∀ p1 p2 : packet, {p1=p2} + {p1≠p2}.
Proof.
  repeat decide equality.
Qed.

Lemma trace_eq_dec : ∀ t1 t2 : trace, {t1=t2} + {t1≠t2}.
Proof.
  repeat decide equality.
Qed.

Lemma port_eq_dec_refl : ∀ A p (x : A) (y : A), (if port_eq_dec p p then x else y) = x.
Proof.
  intros; case_eq (port_eq_dec p p); crush.
Qed.

Lemma packet_eq_dec_refl : ∀ A p (x : A) (y : A), (if packet_eq_dec p p then x else y) =
x.
Proof.
  intros; case_eq (packet_eq_dec p p); crush.
Qed.

Hint Rewrite port_eq_dec_refl : cpdt.
Hint Rewrite packet_eq_dec_refl : cpdt.
```

```
Program Instance locPkt_eq_eqdec : EqDec locPkt eq := eq_locPkt_dec.

Program Instance port_eq_eqdec : EqDec port eq := port_eq_dec.

Program Instance packet_eq_eqdec : EqDec packet eq := packet_eq_dec.

Program Instance trace_eq_eqdec : EqDec trace eq := trace_eq_dec.

Program Instance augmentedPkt_eq_eqdec : EqDec augmentedPkt eq := eq_augmentedPkt_dec.

Lemma equiv_reflexive' : ∀ {A} `{EqDec A} (x : A),
  x === x.
Proof.
  intros. apply equiv_reflexive.
Qed.

Lemma equiv_symmetric' : ∀ {A} `{EqDec A} (x y : A),
  x === y →
  y === x.
Proof.
  intros. apply equiv_symmetric; assumption.
Qed.

Lemma equiv_transitive' : ∀ {A} `{EqDec A} (x y z : A),
  x === y →
  y === z →
  x === z.
Proof.
  intros. eapply @equiv_transitive; eassumption.
Qed.

Definition update_fun (f : port → list augmentedPkt) (a : port) (b : list augmentedPkt)
(a' : port) :=
  if port_eq_dec a a' then b else f a'.

Notation "f a |-> b " :=
  (update_fun f a b) (at level 0).

Lemma update_fun_same_arg1 :
  ∀ f a b,
    update_fun f a b a = b.
Proof.
  unfold update_fun. crush.
Qed.

Hint Rewrite update_fun_same_arg1 : cpdt.

Lemma update_fun_diff_arg1 :
  ∀ f a b a' p,
    port_eq_dec a a' = right p → update_fun f a b a' = f a'.
Proof.
  crush. unfold update_fun; crush.
Qed.

Variable T : topology.

Definition apply_topology (pkts : list locPkt) : list locPkt :=
  map (fun lp ⇒ (T(fst lp), snd lp)) pkts.

Fixpoint update_queue (Q : portQueue) (pkts : list locPkt) (tr : trace) : portQueue :=
  match pkts with
    | [] ⇒ Q
    | (p,pkt) :: pkts' ⇒ update_queue (Q  p |-> snoc (pkt, tr) (Q p) ) pkts' tr
  end.
```

```
Inductive step: portQueue → switchFun → option update → portQueue → switchFun →
Prop :=
| Process :
  ∀ (Q : portQueue) (queue : list augmentedPkt) (S : switchFun) (p : port) (pkt : packet)
(tr : trace) (tr' : trace) (pkts : list locPkt) (Q' : portQueue) (Q'' : portQueue),
    Q p = (pkt,tr) :: queue →
    pkts = apply_topology (S (p,pkt)) →
    tr' = snoc (p,pkt) tr →
    Q' = Q  p |-> queue  →
    Q'' = update_queue Q' pkts tr' →
    step Q S None Q'' S
| Update :
  ∀ (Q : portQueue) (S : switchFun) (S' : switchFun) (u : update),
    S' = override S u →
    step Q S (Some u) Q S'.
Inductive steps : portQueue → switchFun → list update → portQueue → switchFun → Prop
:=
| ReflSteps :
  ∀ Q S,
    steps Q S [] Q S
| SingleStepsNoUpdate :
  ∀ Q S Q' S' Q'' S'' us,
    step Q S None Q'' S'' →
    steps Q'' S'' us Q' S' →
    steps Q S us Q' S'
```

| SingleStepsUpdate :

$\forall$ $Q$ $S$ $Q'$ $S'$ $Q''$ $S''$ $u$ $us$,

**step** $Q$ $S$ (Some $u$) $Q''$ $S''$ $\rightarrow$

**steps** $Q''$ $S''$ $us$ $Q'$ $S'$ $\rightarrow$

**steps** $Q$ $S$ (snoc $u$ $us$) $Q'$ $S'$.

Lemma steps_is_transitive :

$\forall$ $Q$ $S$ $os$ $Q'$ $S'$ $Q''$ $S''$ $os''$,

**steps** $Q$ $S$ $os$ $Q''$ $S''$ $\rightarrow$

**steps** $Q''$ $S''$ $os''$ $Q'$ $S'$ $\rightarrow$

**steps** $Q$ $S$ ($os''$ ++ $os$) $Q'$ $S'$.

Proof.

intros. induction $H$; *crush*. induction $us$; *crush*. apply SingleStepsNoUpdate with ($Q''$ := $Q''$) ($S''$ := $S''$); *crush*. apply SingleStepsNoUpdate with ($Q''$ := $Q''$) ($S''$ := $S''$); *crush*. rewrite app_snoc. apply SingleStepsUpdate with ($Q''$ := $Q''$) ($S''$ := $S''$); *crush*.

Qed.

Lemma empty_steps_is_transitive :

$\forall$ $Q$ $S$ $Q'$ $S'$ $Q''$ $S''$,

**steps** $Q$ $S$ [] $Q''$ $S''$ $\rightarrow$

**steps** $Q''$ $S''$ [] $Q'$ $S'$ $\rightarrow$

**steps** $Q$ $S$ [] $Q'$ $S'$.

Proof.

*crush*. rewrite $\leftarrow$ app_nil with ($ls$ := []). apply steps_is_transitive with ($Q''$ := $Q''$) ($S''$ := $S''$); *crush*.

Qed.

Lemma steps_implies_override :

∀ *Q* *Q'* *S1* *S2* *us*,

**steps** *Q* *S1* *us* *Q'* *S2* → *S2* = override_list *S1* *us*.

`Proof.`

`intros.` `induction` *H*; (*crush*; `inversion` *H*; *crush*). apply override_list_override.

Qed.

Lemma step_empty_same_switch :

∀ *Q* *S* *Q'* *S'*,

**step** *Q* *S* None *Q'* *S'* → *S* = *S'*.

`Proof.`

*crush*. `inversion` *H*; *crush*.

Qed.

Lemma steps_empty_same_switch :

∀ *Q* *S* *Q'* *S'*,

**steps** *Q* *S* `[]` *Q'* *S'* → *S* = *S'*.

`Proof.`

*crush*. `assert` (*S* = override_list *S* `[]`). apply override_empty. `rewrite` *H0*. `symmetry`.
apply steps_implies_override `with` (*Q* := *Q*) (*Q'* := *Q'*). `assumption`.

Qed.

Lemma initial_queue_empty_means_empty :

∀ *Qi* *S* *Q'* *us*,

(∀ *p*,

*Qi* *p* = `[]`) →

**steps** *Qi* *S* *us* *Q'* *S* →

*Qi* = *Q'*.

`Proof.`

```
  crush. induction H0; crush. destruct H0. subst. rewrite H in H0. apply nil_cons in
H0. contradiction. apply IHsteps; crush. destruct H0. subst. rewrite H in H0. apply
nil_cons in H0. contradiction. apply IHsteps. assumption.
Qed.

Inductive reasonableConfig : switchFun → Prop :=
  | ReasonableConfig : ∀ (S : switchFun),
    (∀ (pk : packet),
      S (Drop, pk) = [(Drop, pk)] ∧ S (World, pk) = [(World, pk)])
    → (T Drop = Drop ∧ T World = World)
    → (∀ (p : port) (pk : packet),
      S (p,pk) ≠ [])
    → reasonableConfig S.

Inductive is_prefix : trace → trace → Prop :=
| PrefixRefl : ∀ tr,
  is_prefix tr tr
| PrefixSnoc : ∀ tr tr' a,
  is_prefix tr tr' →
  is_prefix tr (snoc a tr').

Fixpoint isPrefix (tr : trace) (tr' : trace) : Prop :=
  match tr with
    | [] ⇒ True
    | (p, pk)::trs ⇒ match tr' with
                          | [] ⇒ False
                          | (p', pk')::trs' ⇒ if (port_eq_dec p p')
                            then if (packet_eq_dec pk pk')
```

```
                                      then isPrefix trs trs'
                                      else False
                                    else False
                                end
  end.

Lemma prefix_nil :
  ∀ l1,
    isPrefix l1 [] → l1 = [].
Proof.
  intros. induction l1. crush. compute in H. destruct a. contradiction.
Qed.

Lemma snoc_is_prefix :
  ∀ a ls,
    isPrefix ls (snoc a ls).
Proof.
  intros. induction ls; crush. destruct a0; crush.
Qed.

Lemma isPrefix_is_refl :
  ∀ tr,
    isPrefix tr tr.
Proof.
  induction tr; crush. destruct a. crush.
Qed.

Hint Rewrite isPrefix_is_refl : cpdt.

Definition property := trace → Prop.
Definition isTraceProperty prop :=
  ∀ tr tr',
    is_prefix tr' tr → prop tr → prop tr'.

Definition ordinaryPort (p : port) :=
  match p with
    | World ⇒ False
    | Drop ⇒ False
    | Port _ _ ⇒ True
  end.

Definition internalPort (p : port) :=
  ordinaryPort p ∧ ∃ p',
    T p' = p.

Definition externalPort (p : port) :=
  ordinaryPort p ∧ not (internalPort p).

Axiom ports_internal_or_external :
  ∀ p,
    ordinaryPort p → (internalPort p) ∨ (externalPort p).

Definition initialQueue (Q : portQueue) :=
  (∀ p,
    internalPort p → Q p = []) ∧
  (∀ p,
    externalPort p →
    (∀ pkt tr,
      In (pkt,tr) (Q p) → tr = [])) ∧
  Q World = [] ∧
  Q Drop = [].

Inductive generates Q S (tr : trace) :=
| Generates :
  ∀ Q' p pkt,
    initialQueue Q
    →
    In (pkt, tr) (Q' p)
    →
    steps Q S [] Q' S
    → generates Q S tr.

Definition queueContainsTrace (Q : portQueue) (tr : trace) :=
  ∃ p, ∃ pk,
    In (pk, tr) (Q p).

Definition queueSatisfies (Q : portQueue) (prop : property) :=
  ∀ tr,
    queueContainsTrace Q tr
    → prop tr.

Definition satisfies os S (prop : property) :=
  ∀ Q Q',
    initialQueue Q
    → steps Q S os Q' (override_list S os)
    → queueSatisfies Q' prop.

Lemma initial_queue_empty_traces :
  ∀ Qi tr,
    initialQueue Qi →
    queueContainsTrace Qi tr →
    tr = [].
Proof.
  intros Qi tr H H'. destruct H'. unfold initialQueue in H. destruct H. destruct H1.
destruct H0. destruct x.
  assert (ordinaryPort (Port n n0)). crush. apply ports_internal_or_external in H3; util_crush.
apply H in H2. rewrite H2 in H0; util_crush. apply H1 with (pkt := x0) (tr := tr) in
H2; util_crush. intuition. rewrite H3 in H0; util_crush. intuition. rewrite H4 in
H0. intuition.
Qed.

Ltac initial_queue_snoc_tac :=
  match goal with
    | [ H : initialQueue ?Qi, H' : queueContainsTrace ?Qi (snoc ?B ?Tr) ⊢ _ ] ⇒
      let H'' := fresh "H''" in remember_clear H' H''; idtac; apply initial_queue_empty_traces
with Qi (snoc B Tr) in H'; util_crush
  end.
```

### A.3.6 Per-Packet Proofs

```
Require Import sigcomm.
Require Import CpdtTactics.
Require Import utilities.

Require Import Classes.Equivalence.
Require Import Relations.Relation_Definitions.

Section per_packet.
```

```
Notation "[ ]" := nil : list_scope.

Require Import List.

Notation "[ a ; .. ; b ]" := (a :: .. (b :: []) ..) : list_scope.

Require Import Bool.Bool.

Hypothesis R_pkt : relation packet.

Hypothesis R_pkt_equiv : Equivalence R_pkt.

Inductive R : trace → trace → Prop :=

| R_nil : R [] []

| R_snoc : ∀ p1 pk1 pk2 l1 l2,

  R_pkt pk1 pk2 →

  R l1 l2 →

  R (snoc (p1,pk1) l1) (snoc (p1, pk2) l2).

Lemma R_refl :

  Reflexive R.

Proof.

  crush; unfold Reflexive; destruct x using snoc_induction; crush. apply R_nil.

  destruct a; apply R_snoc; crush.

Qed.

Lemma R_sym :

  Symmetric R.

Proof.

  crush; unfold Symmetric. destruct x using snoc_induction; destruct y using
snoc_induction; util_crush.

  repeat (inversion H; util_crush).

  inversion H; snoc_tac.
  inversion H; util_crush. apply IHx in H3. apply R_snoc; crush.
Qed.

Ltac R_nil_tac :=
  match goal with
    | [ H : R [] _ ⊢ _ ] ⇒ inversion H; snoc_tac
    | [ H : R _ [] ⊢ _ ] ⇒ apply R_sym in H; inversion H; snoc_tac
end.

Lemma R_trans :
  Transitive R.
Proof.
  crush; unfold Transitive. destruct x using snoc_induction; destruct y using snoc_induction;
destruct z using snoc_induction; util_crush; try R_nil_tac.
  inversion H0; inversion H; util_crush.
  apply IHx with (z := z) (y := y) in H9; crush.
  apply R_snoc. unfold Transitive in Equivalence_Transitive. eapply Equivalence_Transitive.
  apply H8.
  assumption.
  assumption.
Qed.

Lemma R_Equiv :
  Equivalence R.
Proof.
  crush.
  apply R_refl.
  apply R_sym.
  apply R_trans.
Qed.

Lemma R_prefix :
  ∀ t1 tr1 t2 tr2,
    R (snoc t1 tr1) (snoc t2 tr2) →
    R tr1 tr2.
Proof.
  intros t1 tr1 t2 tr2 H.
  inversion H; util_crush.
Qed.

Lemma R_snoc_inv :
  ∀ t1 tr1 tr2,
    R (snoc t1 tr1) tr2 →
    ∃ t2,
      ∃ tr2',
        tr2 = snoc t2 tr2'.
Proof.
  intros t1 tr1 tr2 H. inversion H; util_crush.
  ∃ (p1, pk2), l2; util_crush.
Qed.

Lemma is_prefix_nil :
  ∀ tr,
    is_prefix tr [] → tr = [].
Proof.
  intros tr H. inversion H; util_crush.
  Qed.

  Ltac prefix_nil_tac :=
    match goal with
      | [ H : is_prefix ?A [] ⊢ _ ] ⇒ apply is_prefix_nil in H; subst
    end.

  Ltac step_none_tac :=
      match goal with
        | [ H : step _ ?S None _ ?S ⊢ _ ] ⇒ fail 1
        | [ H : step _ _ None _ _ ⊢ _ ] ⇒ let H' := fresh "H'" in
          remember_clear H H'; apply step_empty_same_switch in H'; subst
      end.

  Ltac steps_none_tac :=
    progress
      match goal with
        | [ H : steps _ ?S [] _ ?S ⊢ _ ] ⇒ fail 1
        | [ H : steps _ _ [] _ _ ⊢ _ ] ⇒ let H' := fresh "H'" in
          remember_clear H H'; apply steps_empty_same_switch in H'; subst
        | [ H : _ ⊢ steps ?A ?B [] ?A ?B ] ⇒ apply ReflSteps
      end.

  Inductive steps' : portQueue → switchFun → list update → portQueue → switchFun →
Prop :=
  | ReflSteps' :
    ∀ Q S,
      steps' Q S [] Q S
  | SingleSteps'NoUpdate :
    ∀ Q S us Q' S' Q'' S'',
      steps' Q S us Q'' S'' →
      step Q'' S'' None Q' S' →
      steps' Q S us Q' S'
  | SingleSteps'Update :
    ∀ Q S us u Q' S' Q'' S'',
      steps' Q S us Q'' S'' →
      step Q'' S'' (Some u) Q' S' →
      steps' Q S (u :: us) Q' S'.

  Inductive steps'' : portQueue → switchFun → list update → portQueue → switchFun →
Prop :=
    | ReflSteps'' :
      ∀ Q S,
        steps'' Q S [] Q S
    | SingleStepNoUpdate:
      ∀ Q S Q' S',
        step Q S None Q' S' →
        steps'' Q S [] Q' S'
    | SingleStepUpdate:
      ∀ Q S u Q' S',
        step Q S (Some u) Q' S' →
        steps'' Q S [u] Q' S'
    | Steps''Trans:
      ∀ Q S us1 us2 Q' S' Q'' S'',
        steps'' Q S us1 Q'' S'' →
        steps'' Q'' S'' us2 Q' S' →
        steps'' Q S (us2 ++ us1) Q' S'.
  Lemma steps_is_refl_trans_closure :
    ∀ Q S us Q' S',
      steps Q S us Q' S' ↔ steps'' Q S us Q' S'.
  Proof.
    intros Q S us Q' S'. crush.
      induction H.
      apply ReflSteps''.
      apply SingleStepNoUpdate in H. rewrite ← app_nil_r with (l := us). apply Steps''Trans
with (Q'' := Q'') (S'' := S''); crush.
      rewrite snoc_app. apply SingleStepUpdate in H. apply Steps''Trans with (Q'' :=
Q'') (S'' := S''); crush.
      induction H.
      apply ReflSteps.
      apply SingleStepsNoUpdate with (Q'' := Q') (S'' := S'); crush. apply ReflSteps.
      rewrite ← app_nil_l with (l := [u]). rewrite ← snoc_app. apply SingleStepsUpdate
with (Q'' := Q') (S'' := S'); crush. apply ReflSteps.
      apply steps_is_transitive with (Q'' := Q'') (S'' := S''); assumption.
   Qed.
  Lemma steps'_is_transitive :
    ∀ Q S os Q' S' Q'' S'' os'',
      steps' Q S os Q'' S'' →
      steps' Q'' S'' os'' Q' S' →
      steps' Q S (os'' ++ os) Q' S'.
```

Proof.

*crush.* induction *H0*.

rewrite app_nil_l with $(l := os)$. assumption.

apply *IHsteps'* in *H*. apply SingleSteps'NoUpdate with $(Q'' := Q'')$ $(S'' := S'')$; *crush.*

apply *IHsteps'* in *H*. rewrite ← app_comm_cons. apply SingleSteps'Update with $(Q''$ $:= Q'')$ $(S'' := S'')$; *crush.*

Qed.

Lemma steps'_is_refl_trans_closure :

∀ *Q S us Q' S'*,

**steps'** *Q S us Q' S'* ↔ **steps''** *Q S us Q' S'*.

Proof.

*crush.*

induction *H*.

apply ReflSteps''.

apply SingleStepNoUpdate in *H0*. rewrite ← app_nil_l with $(l := us)$. apply Steps''Trans with $(Q'' := Q'')$ $(S'' := S'')$; *crush.*

apply SingleStepUpdate in *H0*. rewrite ← app_nil_l with $(l := us)$. rewrite app_comm_cons. apply Steps''Trans with $(Q'' := Q'')$ $(S'' := S'')$; *crush.*

induction *H*.

apply ReflSteps'.

apply SingleSteps'NoUpdate with $(Q := Q)$ $(S := S)$ $(us :=$ []$)$ in *H*; *crush.* apply ReflSteps'.

apply SingleSteps'Update with $(Q := Q)$ $(S := S)$ $(us :=$ []$)$ in *H*; *crush.* apply ReflSteps'.

apply steps'_is_transitive with $(Q'' := Q'')$ $(S'' := S'')$; *crush.*

```
Qed.

Lemma steps'_is_steps :
  ∀ Q S us Q' S',
    steps' Q S us Q' S' ↔ steps Q S us Q' S'.
Proof.
  intros. rewrite steps'_is_refl_trans_closure. rewrite steps_is_refl_trans_closure. intuition.
Qed.

Lemma steps'_empty_same_switch :
  ∀ Q S Q' S',
    steps' Q S [] Q' S' → S = S'.
Proof.
  intros Q S Q' S' H. rewrite steps'_is_steps in *. steps_none_tac; reflexivity.
Qed.

Ltac steps'_none_tac :=
  progress
    match goal with
      | [ H : steps' _ ?S [] _ ?S ⊢ _ ] ⇒ fail 1
      | [ H : steps' _ _ [] _ _ ⊢ _ ] ⇒ let H' := fresh "H'" in
          remember_clear H H'; apply steps'_empty_same_switch in H'; subst
      | [ H : _ ⊢ steps' ?A ?B [] ?A ?B ] ⇒ apply ReflSteps'
    end.

Ltac R_tac :=
  match goal with
    | [ H : _ ⊢ R ?A ?A ] ⇒ apply R_refl
    | [ H : R ?A ?B ⊢ R ?B ?A ] ⇒ apply R_sym
    end.

  Ltac R_trans_tac :=
    match goal with
      | [ H : R ?A ?B, H' : R ?B ?C ⊢ R ?A ?C ] ⇒ let H'' := fresh "H''" in assert
(H'' : Transitive R); (apply R_trans || unfold Transitive in H''; apply H'' with (z := C)
in H; assumption)
      | [ H : R ?B ?A, H' : R ?B ?C ⊢ R ?A ?C ] ⇒ apply R_sym in H; R_trans_tac
    end.

  Ltac crush' :=
    repeat (util_crush || R_tac || R_trans_tac || prefix_nil_tac || steps_none_tac || steps'_none_tac
|| step_none_tac).

  Definition initially_reachable Q S :=
    ∃ Qi,
      initialQueue Qi ∧
      steps Qi S [] Q S.

  Definition per_packet_consistent os S S' : Prop :=
    ∀ Q Q' tr ,
      initialQueue Q
      → steps Q S os Q' S'
      → queueContainsTrace Q' tr
      → ∃ Qi, ∃ Q'', ∃ tr',
        initialQueue Qi
        ∧ ( steps Qi S [] Q'' S ∨ steps Qi S' [] Q'' S')
        ∧ R tr' tr
        ∧ queueContainsTrace Q'' tr'.
```

```
Definition blind_property (prop : property) :=
  ∀ tr tr', R tr tr' → (prop tr ↔ prop tr').

Definition universal_property_preservation (os : list update) S S' :=
  ∀ (P : property) (proof : isTraceProperty P),
    blind_property P →
    S' = override_list S os
    → satisfies [] S P
    → satisfies [] S' P
    → satisfies os S P.

Theorem per_packet_preserves_properties :
  ∀ os S S',
    per_packet_consistent os S S' → universal_property_preservation os S S'.
  Proof.
    unfold universal_property_preservation.
    intros os S S' PPC P TraceP BlindP override_os Sat_S Sat_S'.
    unfold per_packet_consistent in PPC.
    unfold satisfies.
    intros Q Q' Q_initial steps_Q_os.
    unfold queueSatisfies.
    intros tr Q'_tr.
    specialize (PPC Q Q' tr).
    unfold blind_property in BlindP.
    crush.
    unfold satisfies in Sat_S.
    apply Sat_S in H2; crush.
```

```
      unfold queueSatisfies in H2. apply H2 in H3. apply BlindP in H1. crush.
      unfold satisfies in Sat_S'. apply Sat_S' in H2; crush. apply H2 in H3; apply BlindP in H1; crush.
    Qed.
  Notation "f  a |-> b  " :=
    (update_fun f  a  b) (at level 0).
  Ltac obvious :=
    match goal with
      | [ H : ?P, H' : ?Q ⊢ ∃ _ : ?P , ∃ _ : ?Q , _ ] ⇒ ∃ H, H'; crush'
      | [ H : ?P, H' : queueContainsTrace ?H [] ⊢ ∃ _ : ?P , ∃ _ : trace, _ ] ⇒ ∃ H, []; crush'
    end.
  Lemma update_queue_contains_trace:
    ∀ pkts  Q  tr  tr',
      tr ≠ tr' →
      queueContainsTrace (update_queue Q pkts tr') tr →
      queueContainsTrace Q tr.
  Proof.
    induction pkts; crush. destruct a. assert (queueContainsTrace (Q)  p |-> snoc (p0 , tr') (Q p)  tr). apply IHpkts with (tr' := tr'). assumption. assumption. repeat destruct H1. unfold queueContainsTrace. ∃ x, x0. case_eq (port_eq_dec p x); intros. crush. apply in_snoc3 in H1; crush.
    rewrite update_fun_diff_arg1 with (p := n) in H1; crush.
  Qed.
  Lemma step_prefix_backwards1 :
```

∀ *tr a Q Q' S S' u*,

**step** *Q S u Q' S'* →

queueContainsTrace *Q'* (snoc *a tr*) →

queueContainsTrace *Q* (snoc *a tr*) ∨

queueContainsTrace *Q tr*.

`Proof`.

`intros`. `inversion` *H*; *crush'*. *case_eq* (trace_eq_dec *tr0 tr*); `intros`; *crush*. `unfold` queueContainsTrace. `right`. `intros`. ∃ *p*, *pkt*. *crush*.

*crush'*. `left`. `apply` update_queue_contains_trace `in` *H0*. `repeat destruct` *H0*. `unfold` queueContainsTrace. ∃ *x*, *x0*. *case_eq* (port_eq_dec *p x*); `intros`.

*crush*.

`rewrite` update_fun_diff_arg1 `with` $(p := n0)$ `in` *H0*. `assumption`. `assumption`.

*util_crush*.

`Qed`.

Lemma steps'_prefix_closed :

∀ *Qi S Q*,

**steps'** *Qi S* `[]` *Q S* →

initialQueue *Qi* →

∀ *tr tr'*,

**is_prefix** *tr' tr* →

queueContainsTrace *Q tr* →

∃ *Q'*,

**steps'** *Qi S* `[]` *Q' S* ∧

queueContainsTrace *Q' tr'*.

`Proof`.

```
    intros Qi S Q H H0. generalize_eqs H. induction H; intros. inversion H3;
crush'.
    ∃ Q; crush'.
    remember_clear H3 H3'; apply initial_queue_empty_traces in H3; crush'.
    inversion H4; crush'.
    ∃ Q'; crush'. apply SingleSteps'NoUpdate with (Q'' := Q'') (S'' := S''); crush'.
    match goal with [ H : step Q'' S'' None Q' S'' ⊢ _ ] ⇒ let H' := fresh "H'" in
remember_clear H H'; apply step_prefix_backwards1 with (tr := tr'0) (a := a) in H'; crush'
end.
    apply H2 with (tr := (snoc a tr'0)); crush'.
    apply H2 with (tr := tr'0); crush'.
    crush'.
  Qed.

  Lemma steps_prefix_closed :
    ∀ Qi S Q,
      steps Qi S [] Q S →
      initialQueue Qi →
      ∀ tr tr',
        is_prefix tr' tr →
        queueContainsTrace Q tr →
        ∃ Q',
          steps Qi S [] Q' S ∧
          queueContainsTrace Q' tr'.
  Proof.
    intros. rewrite ← steps'_is_steps in *. apply steps'_prefix_closed with (tr := tr) (tr'
:= tr') in H; crush'. rewrite steps'_is_steps in *. ∃ x; crush'.
  Qed.

  Lemma steps'_prefix_backwards2:
    ∀ Qi Q S,
      steps' Qi S [] Q S →
      initialQueue Qi →
    ∀ t tr tr',
      R tr t →
      is_prefix tr' tr →
      queueContainsTrace Q t →
      ∃ Q',
        ∃ t',
          steps' Qi S [] Q' S ∧
          R tr' t' ∧ queueContainsTrace Q' t'.
  Proof.
    intros Qi Q S H H'. generalize_eqs H. induction H; crush'.
    inversion H2; crush'.
    ∃ Q, t; crush'.
    remember_clear H1 H''. apply R_snoc_inv in H''; crush'. initial_queue_snoc_tac.
    inversion H5; crush'.
    ∃ Q', t; crush'. apply SingleSteps'NoUpdate with (Q'' := Q'') (S'' := S''); crush'.
    remember_clear H3 H''. apply R_snoc_inv in H''; crush'.
    match goal with [ H : step Q'' S'' None Q' S'' ⊢ _ ] ⇒ let H' := fresh "H'" in
remember_clear H H'; apply step_prefix_backwards1 with (tr := x0) (a := x) in H'; crush'
end.
```

apply $H1$ with $(t := (\text{snoc}\ x\ x0))$ $(tr := (\text{snoc}\ a\ tr'0))$; *crush'*.

apply $H1$ with $(t := x0)$ $(tr := tr'0)$; *crush'*. apply R_prefix in $H3$; *crush'*.

Qed.

Lemma steps_prefix_backwards2:

$\forall\ Qi\ Q\ S,$

**steps** $Qi\ S$ [] $Q\ S \rightarrow$

initialQueue $Qi \rightarrow$

$\forall\ t\ tr\ tr',$

**R** $tr\ t \rightarrow$

**is_prefix** $tr'\ tr \rightarrow$

queueContainsTrace $Q\ t \rightarrow$

$\exists\ Q'$,

$\exists\ t'$,

**steps** $Qi\ S$ [] $Q'\ S \wedge$

**R** $tr'\ t' \wedge$ queueContainsTrace $Q'\ t'$.

Proof.

intros. rewrite $\leftarrow$ steps'_is_steps in $*$. apply steps'_prefix_backwards2 with $(t := t)$ $(tr := tr)$ $(tr' := tr')$ in $H$; *crush'*. rewrite steps'_is_steps in $*$; *crush'*. $\exists\ x, x0$; *crush'*.

Qed.

Lemma steps_prefix_backwards :

$\forall\ Q\ x0\ S,$

initialQueue $Q \rightarrow$

**steps** $Q\ S$ [] $x0\ S \rightarrow$

$\forall\ tr\ tr'\ x1,$

**is_prefix** $tr'\ tr \rightarrow$

```
      R tr x1 →
      queueContainsTrace x0 x1 →
        ∃ Q' : portQueue,
          ∃ t' : trace,
            steps Q S [] Q' S ∧
            R tr' t' ∧ queueContainsTrace Q' t'.
    Proof. intros Q x0 S H H0 tr tr' x1. inversion H0; crush.
      inversion H3; crush.
      ∃ x0, x1. crush.
      apply steps_prefix_backwards2 with (Q := x0) (tr := (snoc a tr'0)) (t := x1); crush.
      apply steps_prefix_backwards2 with (Q := x0) (tr := tr) (t := x1); crush.
      apply steps_prefix_backwards2 with (Q := x0) (tr := tr) (t := x1); crush.
    Qed.
  Definition P_or S1 S2 t :=
    ∃ Q, ∃ Q', ∃ t',
      initialQueue Q ∧
      (steps Q S1 [] Q' S1 ∨ steps Q S2 [] Q' S2)
      ∧ R t t'
      ∧ queueContainsTrace Q' t'.
  Ltac obvious' :=
    match goal with
      | [ H : ?P, H' : ?Q, H'' : ?R ⊢ ∃ _ : ?P, ∃ _ : ?Q, ∃ _ : ?R, _ ] ⇒ solve [ ∃ H, H',
H''; crush' ]
    end.
  Lemma P_or_trace_property :
```

```
  ∀ S1 S2 ,
    isTraceProperty (P_or S1 S2).
Proof.
  unfold isTraceProperty; intros. unfold P_or in *. crush'.
  apply steps_prefix_backwards with (tr := tr) (tr' := tr') (x1 := x1) in H3; crush'.
  obvious'.
  apply steps_prefix_backwards with (tr := tr) (tr' := tr') (x1 := x1) in H3; crush'.
  obvious'.
Qed.
Lemma P_or_blind :
  ∀ S1 S2,
    blind_property (P_or S1 S2).
Proof.
  unfold blind_property; unfold P_or in *; crush'; obvious'.
Qed.
Lemma P_or_satisfies_S :
  ∀ S1 S2,
    satisfies [] S1 (P_or S1 S2).
Proof.
  crush. unfold satisfies. intros. unfold queueSatisfies. intros. unfold P_or. obvious'.
Qed.
Lemma P_or_satisfies_S' :
  ∀ S1 S2,
    satisfies [] S2 (P_or S1 S2).
Proof.
```

```
  crush. unfold satisfies. intros. unfold queueSatisfies. intros. unfold P_or. obvious'.
Qed.

Theorem property_preservation_implies_per_packet :
  ∀ os S S',
    universal_property_preservation os S S' → per_packet_consistent os S S'.
Proof.
  intros os S S' UPP.
  unfold universal_property_preservation in UPP.
  unfold per_packet_consistent.
  intros Q Q' tr Q_initial steps_S_S' Q'_tr.
  specialize (UPP (P_or S S')).
  crush.
  remember_clear steps_S_S' S'_S.
  apply steps_implies_override in S'_S.
  unfold satisfies at 3 in UPP.
  unfold queueSatisfies in UPP.
  apply UPP with (Q := Q) in Q'_tr; crush.
  unfold P_or in Q'_tr.
  destruct Q'_tr as [ x Q'_tr ].
  destruct Q'_tr as [ x' Q'_tr ].
  destruct Q'_tr as [ tr' Q'_tr ].
  destruct Q'_tr as [ x_initial Q'_tr ].
  obvious'.
  apply P_or_trace_property.
  apply P_or_blind.
```

```
    apply P_or_satisfies_S.
    apply P_or_satisfies_S'.
  Qed.

  Theorem property_preservation_is_per_packet :
    ∀ os S S',
      universal_property_preservation os S S' ↔ per_packet_consistent os S S'.
  Proof.
    intros os S S'.
    split.
    apply property_preservation_implies_per_packet.
    apply per_packet_preserves_properties.
  Qed.

End per_packet.
```