Network Programming with frenetic

Nate Foster (Cornell)
Arjun Guha (UMass)
Victoria Yang (Cornell)

ECOOP Summer School
Networks Today

Reasoning about network behavior is extremely difficult…
Reasoning about network behavior is extremely difficult…
Reasoning about network behavior is extremely difficult…

…due to the proliferation of devices, protocols, languages
Networks Today

Reasoning about network behavior is extremely difficult…

Does correctness matter? The Internet is best effort…

…due to the proliferation of devices, protocols, languages
Networks Today

Reasoning about network behavior is extremely difficult…

Does correctness matter? The Internet is best effort…
…the end-to-end principle says that hosts are best equipped to deal with failures!

…due to the proliferation of devices, protocols, languages
The malware utilized is absolutely unsophisticated [...] If Target had had a firm grasp on its network security [...] they absolutely would have observed this behavior.

A network change was [...] executed incorrectly [...] more "stuck" volumes and added more requests to the re-mirroring storm.

We discovered a misconfiguration on this pair of switches that caused what's called a "bridge loop" in the network.

Experienced a network connectivity issue [...] interrupted the airline's flight departures, airport processing and reservations systems.
Example: Outages

We discovered a misconfiguration on this pair of switches that caused what's called a “bridge loop” in the network.

Even technically sophisticated companies are struggling to build networks that provide reliable performance.

The malware utilized is absolutely unsophisticated [...] If Target had had a firm grasp on its network security [...] they absolutely would have observed this behavior.

Experienced a network connectivity issue [...] interrupted the airline's flight departures, airport processing and reservations systems.
Example: Cloud Computing

Would you relocate critical infrastructure to the cloud…
Example: Cloud Computing

Would you relocate critical infrastructure to the cloud...

...if your traffic was not guaranteed to be isolated from other tenants during periods of routine maintenance?
Example: Cloud Computing

Would you relocate critical infrastructure to the cloud…

…if your traffic was not guaranteed to be isolated from other tenants during periods of routine maintenance?
Example: Cloud Computing

Would you relocate critical infrastructure to the cloud...

Networks are critical for ensuring the security of many systems... so it is important they function as expected

...if your traffic was not guaranteed to be isolated from other tenants during periods of routine maintenance?
Software-Defined Networking

A clean-slate programmable network architecture
A Major Trend in Networking

Backbone network runs OpenFlow

Acquired for $1.2B
**Vision:** program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically.
Vision: program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically.
Tutorial Outline
Tutorial Outline

Part I: Ox
- OpenFlow Overview
- Ox Applications

Part II: Frenetic
- NetKAT Overview
- NetKAT Applications

Part III: Formal methods
- Update consistency
- Verification and reasoning
OpenFlow Overview
OpenFlow Architecture

Controller

Ox Controller Platform
or POX, Beacon, Floodlight, etc.

OpenFlow API

OpenFlow Switch

OpenFlow-compatible switches
Pica8, Dell, NEC, HP, and many others
OpenFlow Switch

Controller
OpenFlow Switch

Controller
OpenFlow Switch

Controller

packet_in

packet_out all ports
Can write any packet processing function we want in OCaml
OpenFlow API

Switch to controller:
- `switch_connected`
- `switch_disconnected`
- `packet_in`
- `stats_reply`

Controller to switch:
- `packet_out`
- `flow_mod`
- `stats_request`
Demo: Ox Repeater
Frenetic Overview
OpenFlow is a machine language

Programmers must think in terms of low-level concepts such as:

- Flow tables
- Matches
- Priorities
- Timeouts
- Events
- Callbacks

Key issue: programs don’t compose!
Current Controllers

(Monitor | Route | Load Balance) ; Firewall

Controller Platform
Current Controllers

One monolithic application

(Monitor | Route | Load Balance); Firewall

Controller Platform
Current Controllers

Challenges:
- Writing, testing, and debugging programs
- Reusing code across applications
- Porting applications to new platforms
Language-Based Approach

Monitor | Route | Load Balance | Firewall

Compiler | Run-Time System

Controller Platform

---
Language-Based Approach

One module for each task

Monitor | Route | Load Balance | Firewall

Compiler | Run-Time System

Controller Platform

---
Language-Based Approach

One module for each task

Benefits:
- Easier to write, test, and debug programs
- Can reuse modules across applications
- Possible to port applications to new platforms
Frenetic is a programming language

Programmers work in terms of natural constructs:

- Functions
- Predicates
- Relational operators
- Logical properties

Compiler bridges the gap between these abstractions and their implementations in OpenFlow
Frenetic is a programming language

Programmers work in terms of natural constructs:

- Functions
- Predicates
- Relational operators
- Logical properties

Compiler bridges the gap between these abstractions and their implementations in OpenFlow
Network-Wide Programming

What features should an SDN language provide?
Network-Wide Programming

What features should an SDN language provide?

• Packet predicates
• Packet transformations
Network-Wide Programming

What features should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
Network-Wide Programming

What features should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
Network-Wide Programming

What features should an SDN language provide?

• Packet predicates
• Packet transformations
• Path construction
• Path concatenation
• Path union
Network-Wide Programming

What features should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union
- Path iteration
NetKAT Language

\[ f ::= \text{switch} | \text{port} | \text{ethSrc} | \text{ethDst} | ... \]
\[ a, b, c ::= \text{true} \]
\[ | \text{false} \]
\[ | f = n \]
\[ | a_1 \| a_2 \]
\[ | a_1 \&\& a_2 \]
\[ | ! a \]
\[ p, q, r ::= \text{filter} a \]
\[ | f ::= n \]
\[ | p_1 + p_2 \]
\[ | p_1; p_2 \]
\[ | p^* \]
NetKAT Language

\[ f ::= \text{switch} | \text{port} | \text{ethSrc} | \text{ethDst} | \ldots \]

\[ a, b, c ::= \text{true} \quad (\text{false}) \]

\[ f = n \quad (\text{test}) \]

\[ a_1 || a_2 \quad (\text{disjunction}) \]

\[ a_1 && a_2 \quad (\text{conjunction}) \]

\[ ! a \quad (\text{negation}) \]

\[ p, q, r ::= \text{filter} \ a \quad (\text{filter}) \]

\[ f ::= n \quad (\text{modification}) \]

\[ p_1 + p_2 \quad (\text{union}) \]

\[ p_1; p_2 \quad (\text{sequence}) \]

\[ p^* \quad (\text{iteration}) \]

\[
\text{if } a \text{ then } p_1 \text{ else } p_2 \triangleq \text{(filter } a; p_1) + \text{(filter } !a; p_2) \\
\text{drop } \triangleq \text{filter false} \\
\text{id } \triangleq \text{filter true}
\]
Demo: NetKAT Repeater
Demo: Ox Firewall
Demo: NetKAT Firewall
Dynamic Applications

- Application
- Configurations
- Run-Time System
Dynamic Applications

High-level application logic
Often expressed as a finite-state machine on network events (topology changes, new connections, etc.)
Dynamic Applications

Network-wide packet-processing function

Expressed in terms of a set of forwarding tables, one per switch in the network
Code that manages the rules installed on switches

Translate configuration updates into sequences of OpenFlow instructions

let swap_update_for (t : t) sw_id c_id new_table =
  Deferred.t =
  let max_priority = 65535 in
  let old_table = SwitchMap.find t.edge sw_id with |
  Some ft -> ft |
  None -> [] in
  let new_table = List.fold new_table ~init:([], max_priority) ~f:(fun (acc,pri) x -> (x,pri) :: acc, pri - 1) in
  let del_table = flowtable_diff old_table new_table in
  Deferred.List.iter new_table ~f:(fun (flow,prio) ->
    send t.ctl c_id (0l, to_flow_mod prio flow))
  >>= fun () ->
    Deferred.List.iter del_table ~f:(fun (flow,prio) ->
      send t.ctl c_id (0l, to_flow_del prio flow))
  >>| fun () ->
    t.edge <- SwitchMap.add t.edge sw_id new_table
Dynamic Applications

Forwarding elements that implement packet-processing functionality efficiently in hardware
Demo: Ox Learning
Demo: NetKAT Learning
Reasoning in NetKAT
Network-wide packet-processing function

Expressed in terms of a set of forwarding tables, one per switch in the network
Encoding Tables

Forwarding tables can be expressed as NetKAT policies

**OpenFlow Normal Form (ONF)**

\[
\text{fwd} ::= \text{f}_1 := n_1; \ldots; \text{f}_k := n_k + \text{fwd} \\
\quad | \quad \text{drop} \\
\text{pat} ::= \text{f} = n; \text{pat} \\
\quad | \quad \text{true} \\
\text{tbl} ::= \text{if} \text{ pat then} \text{ fwd} \text{ else} \text{ tbl} \\
\quad | \quad \text{drop}
\]
Encoding Tables

Forwarding tables can be expressed as NetKAT policies

**OpenFlow Normal Form (ONF)**

\[
fwd ::= f_1 := n_1; \ldots; f_k := n_k + fwd \\
\quad | \quad \text{drop}
\]

\[
pat ::= f = n; \quad \text{pat} \\
\quad | \quad \text{true}
\]

\[
tbl ::= \text{if } \text{pat} \text{ then } \text{fwd} \text{ else } \text{tbl} \\
\quad | \quad \text{drop}
\]

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.0/8</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

\[
\text{if } \text{dstport=22 then drop} \\
\text{else if } \text{srcip=10.0.0.0/8 then port := 1} \\
\text{else if } \text{true then port := 2} \\
\text{else drop}
\]
Encoding Tables

Forwarding tables can be expressed as NetKAT policies.

OpenFlow Normal Form (ONF)

\[
fwd ::= f_1 := n_1 ; \ldots ; f_k := n_k + fwd \\
| \text{drop}
\]

\[
pat ::= f = n ; pat \\
| \text{true}
\]

\[
tbl ::= \text{if } pat \text{ then } fwd \text{ else } tbl \\
| \text{drop}
\]

NetKAT compiler rewrites (local) policies into tables.

This encoding also facilitates using NetKAT as the "composition substrate" for other platforms.
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links.
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links.
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links.

**Topology Normal Form**

\[
\text{lpred ::= } \text{switch=n; port=n} \\
\text{lpol ::= switch=n; port=n} \\
\text{link ::= lpred; lpol} \\
\text{topo ::= link + topo} \\
\text{| drop}
\]
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links.

**Topology Normal Form**

\[
\begin{align*}
\text{lpred} & ::= \text{switch}=n; \text{port}=n \\
\text{lpol} & ::= \text{switch}:=n; \text{port}:=n \\
\text{link} & ::= \text{lpred}; \text{lpol} \\
\text{topo} & ::= \text{link} + \text{topo} \\
& | \text{drop}
\end{align*}
\]

\[
\text{switch}=A; \text{port}=1; \text{switch}=B; \text{port}=2 + \\
\text{switch}=B; \text{port}=2; \text{switch}=A; \text{port}=1 + \\
\text{switch}=B; \text{port}=1; \text{switch}=C; \text{port}=2 + \\
\text{switch}=C; \text{port}=2; \text{switch}=B; \text{port}=1 + \\
\text{drop}
\]
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps

\[
\text{id} + (\text{policy}; \text{topo})
\]
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.

\[
id + (\text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo})\
\]
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.

\[
\text{id} + (\text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}; \text{policy}; \text{topo})
\]
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps.

$$
\text{id} + (\text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}; \text{policy}; \text{topo}) + \ldots + (\text{policy}; \text{topo})^*$$
Semantic Foundations

Unlike previous network programming languages, the design of NetKAT is not an accident!

Its foundations rest upon canonical mathematical structure:

- Regular operators \((+, ;, *)\) encode paths through topology
- Boolean operators \((+, ;, !)\) encode forwarding tables

Such structures are called *Kleene Algebras with Tests (KAT)* [Kozen ’96]

KAT has an accompanying proof system for establishing equivalences of the form \(p \sim q\)

Many reasoning tasks can be reduced to checking equivalences between terms
NetKAT Proof System

Kleene Algebra Axioms

\[
\begin{align*}
  p + (q + r) &\sim (p + q) + r \\
  p + q &\sim q + p \\
  p + \text{drop} &\sim p \\
  p + p &\sim p \\
  p; (q; r) &\sim (p; q); r \\
  p; (q + r) &\sim p; q + p; r \\
  (p + q); r &\sim p; r + q; r \\
  \text{id}; p &\sim p \\
  p &\sim p; \text{id} \\
  \text{drop}; p &\sim \text{drop} \\
  p; \text{drop} &\sim \text{drop} \\
  \text{id} + p; p^* &\sim p^* \\
  \text{id} + p^*; p &\sim p^* \\
  p + q; r + r &\sim r \Rightarrow p^*; q + r &\sim r \\
  p + q; r + q &\sim q \Rightarrow p; r^* + q &\sim q
\end{align*}
\]

Boolean Algebra Axioms

\[
\begin{align*}
  a || (b \&\& c) &\sim (a || b) \&\& (a || c) \\
  a || \text{true} &\sim \text{true} \\
  a || !a &\sim \text{true} \\
  a \&\& b &\sim b \&\& a \\
  a \&\& !a &\sim \text{false} \\
  a \&\& a &\sim a
\end{align*}
\]

Packet Axioms

\[
\begin{align*}
  f := n; f' := n' &\sim f' := n'; f := n \quad \text{if } f \neq f' \\
  f := n; f' := n' &\sim f' := n'; f := n \quad \text{if } f \neq f' \\
  f := n; f := n &\sim f := n \\
  f := n; f := n &\sim f := n \\
  f := n; f := n' &\sim f := n' \\
  f := n; f := n' &\sim \text{drop} \quad \text{if } n \neq n' \\
  \text{dup}; f = n &\sim f = n; \text{dup}
\end{align*}
\]
Given:
- Ingress predicate: switch = s₁
- Egress predicate: switch = s₂₁
- Topology: t
- Switch program: p

Check:
- switch = s₁; switch := s₂₁ + (p; t)* ~ (p; t)*
- switch=s₁; (p; t)*; switch = s₂₁ ~ drop
Metatheory

**Soundness:** If $\vdash p \sim q$, then $\llbracket p \rrbracket = \llbracket q \rrbracket$

**Completeness:** If $\llbracket p \rrbracket = \llbracket q \rrbracket$, then $\vdash p \sim q$
Metatheory

**Soundness:** If $\vdash p \sim q$, then $\llbracket p \rrbracket = \llbracket q \rrbracket$

**Completeness:** If $\llbracket p \rrbracket = \llbracket q \rrbracket$, then $\vdash p \sim q$

Established previously for KAT [Kozen & Smith ’96]…
… but NetKAT’s packet histories add extra structure
Soundness: If $\vdash p \sim q$, then $⟦p⟧ = ⟦q⟧$

Completeness: If $⟦p⟧ = ⟦q⟧$, then $\vdash p \sim q$

Established previously for KAT [Kozen & Smith ’96]…
… but NetKAT’s packet histories add extra structure

Idea: develop an alternate semantics based on a language model, and leverage completeness of Kleene Algebra over regular sets [Kozen ’94]

Proof outline:
• Reduced NetKAT
• Regular interpretation
• Normal form
Completeness Proof

\[ p \text{ and } q \text{ such that } \lceil p \rceil = \lceil q \rceil \]
Completeness Proof

\( p \) and \( q \) such that  \([p] = [q]\)

\[ \vdash p = \hat{p} \text{ and } \vdash q = \hat{q} \]

\[ [\hat{p}] = [\hat{q}] \]

\[ G(\hat{p}) = G(\hat{q}) \]

\[ R(\hat{p}) = R(\hat{q}) \]

\[ \vdash \hat{p} = \hat{q} \]

\[ \vdash p = q \]

Reduce and Normalize

Soundness

Language Model

Normal Forms

Kleene Algebra Completeness

Transitivity

\[ \llbracket p \rrbracket = \llbracket q \rrbracket \]

\[ G(\hat{p}) = G(\hat{q}) \]

\[ R(\hat{p}) = R(\hat{q}) \]

\[ \vdash \hat{p} = \hat{q} \]

\[ \vdash p = q \]
NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative
Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

<table>
<thead>
<tr>
<th>Continuation Map:</th>
<th>Observation Map:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{\alpha\beta}(f = n) = 0$</td>
<td>$E_{\alpha\beta}(f = n) = [\alpha = \beta \leq f = n]$</td>
</tr>
<tr>
<td>$D_{\alpha\beta}(\text{dup}) = \alpha \cdot [\alpha = \beta]$</td>
<td>$E_{\alpha\beta}(\text{dup}) = \alpha \cdot [\alpha = \beta]$</td>
</tr>
<tr>
<td>$D_{\alpha\beta}(f := n) = 0$</td>
<td>$E_{\alpha\beta}(f := n) = [f := n = p_{\beta}]$</td>
</tr>
<tr>
<td>$D_{\alpha\beta}(p + q) = D_{\alpha\beta}(p) + D_{\alpha\beta}(q)$</td>
<td>$E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) + E_{\alpha\beta}(q)$</td>
</tr>
<tr>
<td>$D_{\alpha\beta}(p \cdot q) = D_{\alpha\beta}(p) \cdot q + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(q)$</td>
<td>$E_{\alpha\beta}(p \cdot q) = \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(q)$</td>
</tr>
<tr>
<td>$D_{\alpha\beta}(p^<em>) = D_{\alpha\beta}(p) \cdot p^</em> + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(p^*)$</td>
<td>$E_{\alpha\beta}(p^<em>) = [\alpha = \beta] + \sum_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(p^</em>)$</td>
</tr>
</tbody>
</table>
NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

**Continuation Map:**

\[
\begin{align*}
D_{\alpha\beta}(f = n) &= 0 \\
D_{\alpha\beta}(\text{dup}) &= \alpha \cdot [\alpha = \beta] \\
D_{\alpha\beta}(f:=n) &= 0 \\
D_{\alpha\beta}(p + q) &= D_{\alpha\beta}(p) + D_{\alpha\beta}(q) \\
D_{\alpha\beta}(p \cdot q) &= D_{\alpha\beta}(p) \cdot q + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(q) \\
D_{\alpha\beta}(p^*) &= D_{\alpha\beta}(p) \cdot p^* + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(p^*)
\end{align*}
\]

**Observation Map:**

\[
\begin{align*}
E_{\alpha\beta}(f = n) &= [\alpha = \beta \leq f = n] \\
E_{\alpha\beta}(\text{dup}) &= \alpha \cdot [\alpha = \beta] \\
E_{\alpha\beta}(f:=n) &= [f:=n = p_{\beta}] \\
E_{\alpha\beta}(p + q) &= E_{\alpha\beta}(p) + E_{\alpha\beta}(q) \\
E_{\alpha\beta}(p \cdot q) &= \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(q) \\
E_{\alpha\beta}(p^*) &= [\alpha = \beta] + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(p^*)
\end{align*}
\]

Intuitively, these automata recognize the (guarded) strings denoted in NetKAT’s language model.
NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

**Continuation Map:**
\[
\begin{align*}
D_{\alpha\beta}(f = n) &= 0 \\
D_{\alpha\beta}(\text{dup}) &= \alpha \cdot [\alpha=\beta] \\
D_{\alpha\beta}(f:=n) &= 0 \\
D_{\alpha\beta}(p + q) &= D_{\alpha\beta}(p) + D_{\alpha\beta}(q) \\
D_{\alpha\beta}(p \cdot q) &= D_{\alpha\beta}(p) \cdot q + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(q) \\
D_{\alpha\beta}(p^*) &= D_{\alpha\beta}(p) \cdot p^* + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot D_{\gamma\beta}(p^*)
\end{align*}
\]

**Observation Map:**
\[
\begin{align*}
E_{\alpha\beta}(f = n) &= [\alpha=\beta \leq f=n] \\
E_{\alpha\beta}(\text{dup}) &= \alpha \cdot [\alpha=\beta] \\
E_{\alpha\beta}(f:=n) &= [f:=n = p_{\beta}] \\
E_{\alpha\beta}(p + q) &= E_{\alpha\beta}(p) + E_{\alpha\beta}(q) \\
E_{\alpha\beta}(p \cdot q) &= \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(q) \\
E_{\alpha\beta}(p^*) &= [\alpha=\beta] + \Sigma_{\gamma} E_{\alpha\gamma}(p) \cdot E_{\gamma\beta}(p^*)
\end{align*}
\]

Intuitively, these automata recognize the (guarded) strings denoted in NetKAT’s language model

Automata can be represented compactly using sparse matrices, yielding an efficient decision procedure based on bisimulation
Experiments

Networks:
• Topology Zoo
• FatTree
• Stanford Backbone

Programs:
• Shortest Paths
• Stanford Policy

Queries:
• Reachability
• All-Pairs Connectivity
• Loop Freedom
• Translation Validation
Results

Topology Zoo

Connectivity

Loop Freedom

Translation Validation

FatTree

Scalability

Relative Performance

Stanford Backbone

Basic reachability in 0.67s (vs 13s for HSA)
Coq Implementation
Machine Model

Forwarding elements that implement packet-processing functionality efficiently in hardware.
Verified Software Stack

Formalized in Coq
- Denotational semantics of NetCore (an earlier version of NetKAT)
- Operational semantics of OpenFlow
- Compiler
- Run-time system
- Correctness proofs
Verified Software Stack

Formalized in Coq
- Denotational semantics of NetCore (an earlier version of NetKAT)
- Operational semantics of OpenFlow
- Compiler
- Run-time system
- Correctness proofs
Compiler Correctness

Highlights

- Library of algebraic properties of tables
- New tactic for proving equalities on bags
- General-purpose table optimizer
- Key invariant: all synthesized predicates are well-formed (w.r.t. protocol types)

Theorem

\[ \text{Theorem compile_correct :} \]
\[ \forall \text{ pol \ sw \ pt \ pk,} \]
\[ \text{netcore_eval \ pol \ sw \ pt \ pk =} \]
\[ \text{table_eval (compile \ pol \ sw) \ pt \ pk.} \]
Featherweight OpenFlow

Abstract Openflow Protocol

\[
\begin{align*}
\text{Controller output relation} \quad & f_{\text{out}}(\sigma) \rightarrow (sw, CM, \sigma') \\
\text{Receive} \quad & (\sigma, f_{\text{in}}, f_{\text{out}}) | M(sw, SMS, CMS) \rightarrow (\sigma', f_{\text{in}}, f_{\text{out}}) | M(sw, [CM] + SMS, CMS) \\
\text{Switch-FlowMod} \quad & (\sigma, f_{\text{in}}, f_{\text{out}}) | M(sw, SMS, CMS) \rightarrow (\sigma, f_{\text{in}}, f_{\text{out}}) | M(sw, SMS + [CM], CMS) \\
\end{align*}
\]

Syntax

<table>
<thead>
<tr>
<th>Devices</th>
<th>Packet ID</th>
<th>Location</th>
<th>Packet (src, dst)</th>
<th>Located Packet (src, dst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller Components</td>
<td>Controller input relation</td>
<td>Controller output relation</td>
<td>Rule table</td>
<td>Rule table modifier</td>
</tr>
<tr>
<td>Switch Components</td>
<td>Rule table &amp; interpretation</td>
<td>Rule table modifier</td>
<td>Ports on switch</td>
<td>Consumed packets</td>
</tr>
<tr>
<td>OpenFlow Link to Controller</td>
<td>FlowMod (CM)</td>
<td>Message from controller</td>
<td>Message queue to controller</td>
<td>Abstract OpenFlow Protocol</td>
</tr>
</tbody>
</table>
Forwarding

```c
/* Fields to match against flows */
struct ofp_match {
  uint32_t wildcards; /* Wildcard fields. */
  uint32_t in_port;   /* Input switch port. */
  uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
  uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
  uint16_t dl_vlan;   /* Input VLAN. */
  uint8_t dl_vlan_pcp; /* Input VLAN priority. */
  uint8_t pad1[1];    /* Align to 64-bits. */
  uint16_t dl_type;  /* Ethernet frame type. */
  uint8_t nw_tos;    /* IP ToS (DSCP field, 6 bits). */
  uint8_t nw_proto;  /* IP protocol or lower 8 bits of ARP opcode. */
  uint8_t pad2[2];   /* Align to 64-bits. */
  uint32_t nw_src;   /* IP source address. */
  uint32_t nw_dst;   /* IP destination address. */
  uint16_t tp_src;   /* TCP/UDP source port. */
  uint16_t tp_dst;   /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```
/* Fields to match against flows */

struct ofp_match {
    uint32_t wildcards; /* Wildcard fields. */
    uint16_t in_port; /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan; /* Input VLAN. */
    uint8_t dl_vlan_pcp; /* Input VLAN priority. */
    uint8_t pad1[1]; /* Align to 64-bits. */
    uint16_t dl_type; /* Ethernet frame type. */
    uint8_t nw_tos; /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto; /* IP protocol or lower 8 bits of ARP opcode. */
    uint8_t pad2[2]; /* Align to 64-bits. */
    uint32_t nw_src; /* IP source address. */
    uint32_t nw_dst; /* IP destination address. */
    uint16_t tp_src; /* TCP/UDP source port. */
    uint16_t tp_dst; /* TCP/UDP destination port. */
};

OFP_ASSERT(sizeof(struct ofp_match) == 40);
/* Fields to match against flows */

struct ofp_match {
    uint32_t wildcards;    /* Wildcard fields. */
    uint16_t in_port;     /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN];    /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN];    /* Ethernet destination address. */
    uint16_t dl_vlan;      /* Input VLAN. */
    uint8_t dl_vlan_pcp;   /* Input VLAN priority. */
    uint8_t pad1[1];       /* Align to 64-bits. */
    uint16_t dl_type;      /* Ethernet frame type. */
    uint8_t nw_tos;        /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;      /* IP protocol or lower 8 bits of ARP opcode. */
    uint8_t pad2[2];       /* Align to 64-bits. */
    uint16_t nw_src;       /* IP source address. */
    uint16_t nw_dst;       /* IP destination address. */
    uint16_t tp_src;       /* TCP/UDP source port. */
    uint16_t tp_dst;       /* TCP/UDP destination port. */
};

OFP_ASSERT(sizeof(struct ofp_match) == 40);

Record Pattern : Type := MkPattern {
    dlSrc : Wildcard EthernetAddress;
    dlDst : Wildcard EthernetAddress;
    dlType : Wildcard EthernetType;
    dlVlan : Wildcard VLAN;
    dlVlanPcp : Wildcard VLANPriority;
    nwSrc : Wildcard IPAddress;
    nwDst : Wildcard IPAddress;
    nwProto : Wildcard IPProtocol;
    nwTos : Wildcard IPTypeOfService;
    tpSrc : Wildcard TransportPort;
    tpDst : Wildcard TransportPort;
    inPort : Wildcard Port
}.
/* Fields to match against flows */

```c
struct ofp_match {
    uint32_t wildcards; /* Wildcard fields */
    uint32_t in_port; /* Input switch port */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address */
    uint16_t d1_vlan; /* Input VLAN */
    uint16_t d1_dscp; /* Input DSCP */
    uint16_t d1_vlan_pcp; /* Input VLAN priority */
    uint16_t pad1[1]; /* Align to 64-bits */
    uint16_t d1_type; /* Ethernet frame type */
    uint8_t nw_tos; /* IP ToS (DSCP field, 6 bits) */
    uint8_t nw_proto; /* IP protocol or lower 8 bits of ARP opcode */
    uint16_t pad2[2]; /* Align to 64-bits */
    uint32_t nw_src; /* IP source address */
    uint32_t nw_dst; /* IP destination address */
    uint16_t tp_src; /* TCP/UDP source port */
    uint16_t tp_dst; /* TCP/UDP destination port */
};
```

OFP_ASSERT(sizeof(struct ofp_match) == 40);

### Definition: Pattern_inter (p p':Pattern) :=

- Let `d1Src := Wildcard_inter EtherAddress.eqdec (ptrnD1Src p) (ptrnD1Src p')` in
- Let `d1Dst := Wildcard_inter EtherAddress.eqdec (ptrnD1Dst p) (ptrnD1Dst p')` in
- Let `d1Type := Wildcard_inter Word16.eqdec (ptrnD1Type p) (ptrnD1Type p')` in
- Let `d1Vlan := Wildcard_inter Word16.eqdec (ptrnD1Vlan p) (ptrnD1Vlan p')` in
- Let `d1VlanPcp := Wildcard_inter Word8.eqdec (ptrnD1VlanPcp p) (ptrnD1VlanPcp p')` in
- Let `nwSrc := Wildcard_inter Word32.eqdec (ptrnNwSrc p) (ptrnNwSrc p')` in
- Let `nwDst := Wildcard_inter Word32.eqdec (ptrnNwDst p) (ptrnNwDst p')` in
- Let `nwProto := Wildcard_inter Word8.eqdec (ptrnNwProto p) (ptrnNwProto p')` in
- Let `nwTos := Wildcard_inter Word8.eqdec (ptrnNwTos p) (ptrnNwTos p')` in
- Let `tpDst := Wildcard_inter Word16.eqdec (ptrnTpDst p) (ptrnTpDst p')` in
- Let `inPort := Wildcard_inter Wild16.eqdec (ptrnInPort p) (ptrnInPort p')` in
- MkPattern `d1Src d1Dst d1Type d1Vlan d1VlanPcp` `nwSrc nwDst nwProto nwTos` `tpSrc tpDst inPort`.

### Definition: exact_pattern (pk : Packet) (pt : Word16.T) : Pattern :=

MkPattern (WildcardExact (pktD1Src pk) (WildcardExact (pktD1Dst pk)) (WildcardExact (pktD1Type pk)) (WildcardExact (pktD1Vlan pk) (WildcardExact (pktD1VlanPcp pk))) (WildcardExact (pktNwSrc pk)) (WildcardExact (pktNwProto pk)) (WildcardExact (pktNwTos pk)) (Wildcard_of_option (pktTpSrc pk)) (Wildcard_of_option (pktTpDst pk)) (WildcardExact pt)).


negb (Pattern_is_empty (Pattern_inter (exact_pattern pk pt) pat)).
Forwarding

Detailed model of matching, forwarding, and flow table update

Record Pattern : Type := MkPattern {
  d1Src := Wildcard EthernetAddress; 
  d1Type := Wildcard EthernetType; 
  d1Vlan := Wildcard VLAN; 
  d1VlanPcp := Wildcard VLANPriority; 
  nwSrc := Wildcard IPAddress; 
  nwDst := Wildcard IPAddress; 
  nwProto := Wildcard IPProtocol; 
  nwTos := Wildcard IPTypeOfService; 
  tpSrc := Wildcard TransportPort; 
  tpDst := Wildcard TransportPort; 
  inPort := Wildcard Port 
}.

let d1Dst := Wildcard_inter EthernetAddress.eqdec (ptrnD1Dst p) (ptrnD1Dst p') in 
let d1Type := Wildcard_inter Word16.eqdec (ptrnD1Type p) (ptrnD1Type p') in 
let d1Vlan := Wildcard_inter_word16.eqdec (ptrnD1Vlan p) (ptrnD1Vlan p') in 
let d1VlanPcp := Wildcard_inter_word8.eqdec (ptrnD1VlanPcp p) (ptrnD1VlanPcp p') in 
let nwSrc := Wildcard_inter_word32.eqdec (ptrnNwSrc p) (ptrnNwSrc p') in 
let nwDst := Wildcard_inter_word32.eqdec (ptrnNwDst p) (ptrnNwDst p') in 
let nwProto := Wildcard_inter_word8.eqdec (ptrnNwProto p) (ptrnNwProto p') in 
let nwTos := Wildcard_inter_word8.eqdec (ptrnNwTos p) (ptrnNwTos p') in 
let tpSrc := Wildcard_inter_word16.eqdec (ptrnTpSrc p) (ptrnTpSrc p') in 
let tpDst := Wildcard_inter_word16.eqdec (ptrnTpDst p) (ptrnTpDst p') in 
inPort := Wildcard_inter_word16.eqdec (ptrnInPort p) (ptrnInPort p') in 
MkPattern d1Src d1Dst d1Type d1Vlan d1VlanPcp 
  nwSrc nwDst nwProto nwTos 
  tpSrc tpDst 
inPort.

Definition exact_pattern (pk : Packet) (pt : Word16.T) : Pattern := 
  MkPattern 
  (WildcardExact (pktD1Src pk)) (WildcardExact (pktD1Dst pk)) 
  (WildcardExact (pktD1Type pk)) 
  (WildcardExact (pktD1Vlan pk)) (WildcardExact (pktD1VlanPcp pk)) 
  (WildcardExact (pktNwSrc pk)) (WildcardExact (pktNwDst pk)) 
  (WildcardExact (pktNwProto pk)) (WildcardExact (pktNwTos pk)) 
  (WildcardOf_option (pktTpSrc pk)) (WildcardOf_option (pktTpDst pk)) 
  (WildcardExact pt).

  negb (Pattern_is_empty (Pattern_inter (exact_pattern pk pt) pat)).
Asynchrony

“In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance.”

“There is no packet output ordering guaranteed within a port.”
Asynchrony

“In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance.”

“There is no packet output ordering guaranteed within a port.”

Definition \text{InBuf} := \text{Bag Packet}.
Definition \text{OutBuf} := \text{Bag Packet}.
Definition \text{OFInBuf} := \text{Bag SwitchMsg}.
Definition \text{OFOutBuf} := \text{Bag CtrlMsg}.
Asynchrony

“In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance.”

“There is no packet output ordering guaranteed within a port.”

Essential asynchrony: packet buffers, message reordering, and barriers

Definition InBuf := Bag Packet.
Definition OutBuf := Bag Packet.
Definition OFInBuf := Bag SwitchMsg.
Definition OFOutBuf := Bag CtrlMsg.
Weak Bisimulation

$(H_1, \rightarrow)$
Weak Bisimulation

$$(H_1, \square) \rightarrow (S_1, pt_1, \square)$$
Weak Bisimulation

\[(H_1, pt) \xrightarrow{} (S_1, pt_1, pt) \xrightarrow{} (S_2, pt_1, pt)\]
Weak Bisimulation

$$(H_1,\text{✉}) \rightarrow (S_1,pt_1,\text{✉}) \rightarrow (S_2,pt_1,\text{✉}) \rightarrow (H_2,\text{✉})$$
Weak Bisimulation

\[(H_1, \text{mail}) \rightarrow (S_1, pt_{1}, \text{mail}) \rightarrow (S_2, pt_{1}, \text{mail}) \rightarrow (H_2, \text{mail})\]
Weak Bisimulation

\[(H_1, \text{message}) \rightarrow (S_1, pt_1, \text{message}) \rightarrow (S_2, pt_1, \text{message}) \rightarrow (H_2, \text{message})\]
Weak Bisimulation

\[(H_1, \text{Envelope}) \rightarrow (S_1, pt_{1_1}, \text{Envelope}) \rightarrow (S_2, pt_{1_2}, \text{Envelope}) \rightarrow (H_2, \text{Envelope})\]
Theorem: NetCore abstract semantics is weakly bisimilar to Featherweight OpenFlow + NetCore controller
Parameterized Weak Bisimulation

Invariants

- **Safety**: at all times, the rules installed on switches are a *subset* of the controller function
- **Liveness**: the controller eventually processes all packets diverted to it by switches

Theorem

```ocaml
Module RelationDefinitions :=
  FwOF.FwOFRelationDefinitions.Make (AtomsAndController).
...
Theorem fwof_abst_weak_bisim :
  weak_bisimulation
  concreteStep
  abstractStep
  bisim_relation.
```
Consistent Updates
Run-Time Model

Code that manages the rules installed on switches

Translate configuration updates into sequences of OpenFlow instructions
Network Updates

Challenges

• The network is a distributed system
• Can only update one element at a time

Our Approach

• Provide programmers with a construct for updating the entire network at once
• Semantics ensures “reasonable” behavior
• Engineer efficient implementations:
  - Compiler constructs update protocols
  - Optimizations applied automatically
Atomic Updates

• Seem sensible...
• but costly to implement...
• and difficult to reason about, due to behavior on in-flight packets
Atomic Updates
- Seem sensible...
- but costly to implement...
- and difficult to reason about, due to behavior on in-flight packets

Per-Packet Consistent Updates
Every packet processed with old or new configuration, but not a mixture of the two

Per-Flow Consistent Updates
Every set of related packets processed with old or new configuration, but not a mixture of the two
Update Semantics

Atomic Updates
- Seem sensible...
- but costly to implement...
- and difficult to reason about, due to behavior on in-flight packets

Per-Packet Consistent Updates
Every packet processed with old or new configuration, but not a mixture of the two

Per-Flow Consistent Updates
Every set of related packets processed with old or new configuration, but not a mixture of the two

Theorem (Universal Property Preservation)
An update is per-packet consistent if and only if it preserves all safety properties.
Implementation

Two-phase commit
- Build versioned internal and edge switch configurations
- Phase 1: Install internal configuration
- Phase 2: Install edge configuration

Pure Extension
- Update strictly adds paths

Pure Retraction
- Update strictly removes paths

Sub-space updates
- Update modifies a small number of paths
Wrapping Up
Conclusion

- Lots of great PL problems in networking!
- SDN is an enabling technology for this kind of research
- Frenetic is a new platform for programming and reasoning about SDNs:
  - Automated formal reasoning in NetKAT [POPL ’14]
  - Consistent updates [SIGCOMM ’12]
  - Machine-verified controller [PLDI ’13]
- Other work
  - Traffic isolation [HotSDN ’12]
  - Joint host-network programming [SIGCOMM ’13, HotNets ’13]
  - Declarative fault tolerance [HotSDN ’13]
  - Dynamic software updates [HotSDN ’13]
  - Configuration synthesis [SYNT ’13]
  - Tierless programming [HotSDN ’13]
Frenetic @ Home

TP-Link TL-WR1043ND

$50

Open firmware:
www.openwrt.com
Frenetic @ Home

TP-Link TL-WR1043ND

$50

Open firmware:
www.openwrt.com
Thank you!

Collaborators

- Carolyn Anderson (Swarthmore)
- Spiridon Aristides Eliopoulos (Cornell)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Matthew Milano (Cornell)
- Jennifer Rexford (Princeton)
- Mark Reitblatt (Cornell)
- Cole Schlesinger (Princeton)
- Alexandra Silva (Nijmegen)
- Laure Thompson (Cornell)
- Dave Walker (Princeton)

Papers, Code, etc.

http://frenetic-lang.org/