Control Plane Programming
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
Example: Outages

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

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

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: Outages

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

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

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
Frenetic

Vision: program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically
Frenetic

Vision: program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically.
Outline

- Ox
- Frenetic
- Formal methods
OpenFlow Overview
OpenFlow Architecture

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

OpenFlow API

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

Your Program goes here!

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

OpenFlow API

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

Controller
Controller

packet_in

OpenFlow Switch

Computers
OpenFlow Switch

Controller

packet_in

packet_out all ports

OpenFlow Switch
Can write any packet processing function we want in OCaml
<table>
<thead>
<tr>
<th>Priority</th>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>All Packets</td>
<td>All Ports</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Priority | Pattern    | Action
---       | ---        | ---
...       | ...        |...
10        | All Packets| All Ports
...       | ...        |...
Switch to controller:
- switch_connected
- switch_disconnected
- packet_in
- stats_reply

Controller to switch:
- packet_out
- flow_mod
- stats_request
Machine Languages

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

One monolithic application

(Monitor | Route | Load Balance) ; Firewall

Controller Platform

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

Benefits:

• Easier to write, test, and debug programs
• Can reuse modules across applications
• Possible to port applications to new platforms
Language-Based Approach

Benefits:

• Easier to write, test, and debug programs
• Can reuse modules across applications
• Possible to port applications to new platforms

One module each task

Monitor | Route | Load Balance | Firewall

Compiler | Run-Time System

Controller Platform

Benefits:
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
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

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

\[ a,b,c ::= \text{true} \]
\[ \quad \| \quad \text{false} \quad (\text{false}) \]
\[ \quad \| \quad f = n \quad (\text{true}) \]
\[ \quad \| \quad a_1 || a_2 \quad (\text{test}) \]
\[ \quad \| \quad a_1 && a_2 \quad (\text{disjunction}) \]
\[ \quad \| \quad ! a \quad (\text{conjunction}) \]
\[ \quad \| \quad a \quad (\text{negation}) \]

\[ p,q,r ::= \text{filter} \ a \quad (\text{filter}) \]
\[ \quad \| \quad f := n \quad (\text{modification}) \]
\[ \quad \| \quad p_1 + p_2 \quad (\text{union}) \]
\[ \quad \| \quad p_1; p_2 \quad (\text{sequence}) \]
\[ \quad \| \quad p^* \quad (\text{iteration}) \]
NetKAT Language

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

\[ a, b, c ::= \text{true} \quad (* \text{false} *) \]
\[ \quad | \text{false} \quad (* \text{true} *) \]
\[ \quad | f = n \quad (* \text{test} *) \]
\[ \quad | a_1 || a_2 \quad (* \text{disjunction} *) \]
\[ \quad | a_1 \&\& a_2 \quad (* \text{conjunction} *) \]
\[ \quad | ! a \quad (* \text{negation} *) \]

\[ p, q, r ::= \text{filter} a \quad (* \text{filter} *) \]
\[ \quad | f := n \quad (* \text{modification} *) \]
\[ \quad | p_1 + p_2 \quad (* \text{union} *) \]
\[ \quad | p_1; p_2 \quad (* \text{sequence} *) \]
\[ \quad | p^* \quad (* \text{iteration} *) \]

\[ \text{if } a \text{ then } p_1 \text{ else } p_2 \triangleq (\text{filter} \ a; p_1) + (\text{filter} \ !a; p_2) \]

\[ \text{drop} \triangleq \text{filter} \ \text{false} \quad \text{id} \triangleq \text{filter} \ \text{true} \]
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

High-level application logic
Often expressed as a finite-state machine on network events (topology changes, new connections, etc.)
Network-wide packet-processing function

Expressed in terms of a set of forwarding tables, one per switch in the network
Network-wide packet-processing function

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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4, tcpdst = 22</td>
<td>Count, Drop</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Forward 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Forward 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, tcpdst = 22</td>
<td>Count</td>
</tr>
<tr>
<td>tcpdst = 22</td>
<td>Drop</td>
</tr>
</tbody>
</table>
Dynamic Applications

Application
Configurations
Run-Time System

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 : unit Deferred.t =
  let max_priority = 65535
  in
  let old_table = match 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 new_table = List.rev new_table
  in
  let del_table = List.rev (flowtable_diff old_table new_table)
  in
  let to_flow_mod prio flow =
    M.FlowModMsg (SDN_OpenFlow0x01.from_flow prio flow)
  in
  let to_flow_del prio flow =
    M.FlowModMsg ({SDN_OpenFlow0x01.from_flow prio flow with command = DeleteStrictFlow})
  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.
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)**

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

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

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

Forwarding tables can be expressed as NetKAT policies

**OpenFlow Normal Form (ONF)**

```plaintext
fwd ::= f₁ := n₁; ...; fₖ := nₖ + fwd
  | drop
pat ::= f = n; pat
  | true

tbl ::= if pat then fwd else tbl
  | 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>

- if dstport=22 then drop
- else if srcip=10.0.0.1 then port := 1
- else if true then port := 2
- else drop
Encoding Tables

Forwarding tables can be expressed as NetKAT policies

OpenFlow Normal Form (ONF)

\[
\begin{align*}
\text{fwd} & ::= f_1 := n_1; \ldots; f_k := n_k + \text{fwd} \\
& \quad | \; \text{drop} \\
\text{pat} & ::= f = n; \text{pat} \\
& \quad | \; \text{true} \\
\text{tbl} & ::= \text{if pat then fwd else tbl} \\
& \quad | \; \text{drop}
\end{align*}
\]

<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>

if dstport=22 then drop
else if srcip=10.0.0.1 then port := 1
else if true then port := 2
else 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

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

\[
\begin{align*}
l_{\text{pred}} & ::= \text{switch}=n; \text{port}=n \\
l_{\text{pol}} & ::= \text{switch}=n; \text{port}=n \\
\text{link} & ::= l_{\text{pred}}; l_{\text{pol}} \\
\text{topo} & ::= \text{link} + \text{topo} \\
& \mid \text{drop}
\end{align*}
\]

\[
\begin{align*}
\text{switch}=A; \text{port}=1; \text{switch}=B; \text{port}=2 + \\
\text{switch}=B; \text{port}=2; \text{switch}=A; \text{port}=1 + \\
\text{switch}=B; \text{port}=1; \text{switch}=C; \text{port}=2 + \\
\text{switch}=C; \text{port}=2; \text{switch}=B; \text{port}=1 + \\
\text{drop}
\end{align*}
\]
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.
Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps
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:

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

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

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

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

$$\text{id} + (\text{policy}; \topo) + (\text{policy}; \topo; \text{policy}; \topo) + (\text{policy}; \topo; \text{policy}; \topo; \text{policy}; \topo) + \ldots + (\text{policy}; \topo)^*$$
Encoding Networks

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

\[ \text{id} + (\text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo}; \text{policy}; \text{topo}) + \ldots + (\text{policy}; \text{topo})^* \]
Unlike previous network programming languages, the design of NetKAT is not an accident!
Its foundations rest upon canonical mathematical structure:

- Regular operators (\(+, ;, \ast\)) encode paths through topology
- Boolean operators (\(+, ;, \lnot\)) 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
\[ p + (q + r) \sim (p + q) + r \]
\[ p + q \sim q + p \]
\[ p + \text{drop} \sim p \]
\[ p + p \sim p \]
\[ p; (q; r) \sim (p; q); r \]
\[ p; (q + r) \sim p; q + p; r \]
\[ (p + q); r \sim p; r + q; r \]
\[ \mathbf{id}; p \sim p \]
\[ p \sim p; \mathbf{id} \]
\[ \text{drop}; p \sim \text{drop} \]
\[ p; \text{drop} \sim \text{drop} \]
\[ p + p^* \sim p^* \]
\[ \mathbf{id} + p^*; p \sim p^* \]
\[ p + q; r + r \sim r \rightarrow p^*; q + r \sim r \]
\[ p + q; r + q \sim q \Rightarrow p; r^* + q \sim q \]

Boolean Algebra Axioms
\[ a \parallel (b \&\& c) \sim (a \parallel b) \&\& (a \parallel c) \]
\[ a \parallel \text{true} \sim \text{true} \]
\[ a \parallel \neg a \sim \text{true} \]
\[ a \&\& b \sim b \&\& a \]
\[ a \&\& \neg a \sim \text{false} \]

Packet Axioms
\[ f := n; f' := n' ; f := n \quad \text{if} \ f \neq f' \]
\[ f := n; f' = n' ; f := n \quad \text{if} \ f \neq f' \]
\[ f := n; f = n \sim f := n \]
\[ f = n; f := n \sim f = n \]
\[ f := n; f := n' \sim f := n' \]
\[ f = n; f = n' \sim \text{drop} \quad \text{if} \ n \neq n' \]
\[ \text{dup}; f = n \sim f = n; \text{dup} \]
Given:
- Ingress predicate: \( \text{switch} = s_1 \)
- Egress predicate: \( \text{switch} = s_{21} \)
- Topology: \( t \)
- Switch program: \( p \)

Check:
- \( \text{switch} = s_1; \text{switch} := s_{21} + (p; t)^* \sim (p; t)^* \)
- \( \text{switch} = s_1; (p; t)^*; \text{switch} = s_{21} \sim \text{drop} \)
Meta-theory

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

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

**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
**Meta-theory**

**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 and q such that \([p] = [q]\)
Completeness Proof

\[ p \text{ and } q \text{ such that } \llbracket p \rrbracket = \llbracket q \rrbracket \]

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

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

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

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

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

\[ \vdash p \equiv q \]

Reduce and Normalize

Soundness

Language Model

Normal Forms

Kleene Algebra Completeness

Transitivity
NetKAT Automata

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

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

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.

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.
NetKAT Automata

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

Continuation Map:

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

Observation Map:

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

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

FatTree  Scalability

Stanford Backbone

Basic reachability in 0.67s (vs 13s for HSA)
Coq Implementation
Forwarding elements that implement packet-processing functionality efficiently in hardware.
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

Theorem compile_correct :
  ∀ pol sw pt pk, netcore_eval pol sw pt pk =
  table_eval (compile pol sw) pt pk.
OpenFlow Specification

42 pages...

...of informal prose

...diagrams and flow charts

...and C struct definitions
Featherweight OpenFlow

Syntax

Semantics

Models all features related to packet forwarding, and all essential asynchrony
/* 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 nw_tos[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. */
    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 */
  uint16_t dl_vlan_pcp; /* Input VLAN priority */
  uint8_t dl_type; /* Ethernet frame type */
  uint8_t dl_vlan_pcp; /* Align to 64-bits */
  uint8_t dl_vlan; /* IP ToS (DSCP field, 6 bits) */
  uint16_t nw_proto; /* IP protocol or lower 8 bits of ARP opcode */
  uint8_t nw_tos; /* IP source address */
  uint32_t nw_src; /* IP destination address */
  uint32_t nw_dst; /* IP source address */
  uint8_t nw_tos; /* IP destination address */
  uint16_t nw_proto; /* TCP/UDP source port */
  uint16_t tp_src; /* TCP/UDP destination port */
  uint16_t tp_dst; /* Wildcard fields */
};

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

Record Pattern : Type := MkPattern {
  d1Src := WildcardEthernetAddress.eqdec (ptrnDlSrc p) (ptrnNwSrc p') in
  d1Dst := WildcardEthernetAddress.eqdec (ptrnDlDst p) (ptrnNwDst p') in
  d1Type := WildcardEthernetType
  d1Vlan := WildcardVLAN
  d1VlanPcp := WildcardVLANPrio;
  nwSrc := WildcardIPAddress;
  nwDst := WildcardIPAddress;
  nwProto := WildcardIPProtocol;
  nwToS := WildcardIPTypeOfService;
  tpSrc := WildcardTransportPort;
  tpDst := WildcardTransportPort;
  inPort := WildcardPort
};

Definition Pattern_inter (p p':Pattern) :=
let d1Type := Wildcard_inter Word16.eqdec (ptrnNwProto p) (ptrnNwProto p') in
let d1Vlan := Wildcard_inter Word16.eqdec (ptrnDlVlan p) (ptrnDlVlan p') in
let d1VlanPcp := Wildcard_inter Word16.eqdec (ptrnDlVlanPcp p) (ptrnDlVlanPcp 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
let 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
  (Wildcard듦 (pktDlSrc pk)) (Wildcard démarch (pktDlDst pk))
  (Wildcard démarch (pktDlType pk))
  (Wildcard démarch (pktDlVlan pk)) (Wildcard démarch (pktDlVlanPcp pk))
  (Wildcard démarch (pktNwSrc pk)) (Wildcard démarch (pktNwDst pk))
  (Wildcard démarch (pktNwProto pk)) (Wildcard démarch (pktNwToS pk))
  (Wildcard démarch (pktTpSrc pk)) (Wildcard démarch (pktTpDst pk))
  (Wildcard démarch pt).

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

```
/* Fields to match against flows */

struct ofp_match {
    uint32_t wildcards; /* Wildcard fields. */
    uint16_t dp_src;    /* 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[1]; /* Align to 64-bits. */
    uint32_t nw_src; /* IP source address. */
    uint32_t nw_dst; /* IP destination address. */
}

// Detailed model of matching, forwarding, and flow table update

let dlVlanPcp := wildcard_inter Word8.eqdec (ptrnDlVlanPcp p) (ptrnDlVlanPcp 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
let inPort := wildcard_inter Word16.eqdec (ptrnInPort p) (ptrnInPort p') in
MkPattern dlSrc dlDst dlType dlVlan dlVlanPcp
    nwSrc nwDst nwProto nwTos
    tpSrc tpDst
    inPort.

Definition exact_pattern (pk : Packet) (pt : Word16.T) : Pattern :=
MkPattern
    (wildcardExact (pktDlSrc pk)) (wildcardExact (pktDlDst pk))
    (wildcardExact (pktDlType pk))
    (wildcardExact (pktDlVlan pk)) (wildcardExact (pktDlVlan p))
    (wildcardExact (pktNwSrc pk)) (wildcardExact (pktNwDst pk))
    (wildcardExact (pktNwProto pk)) (wildcardExact (pktNwTos pk))
    (wildcardOf_option (pktTpSrc pk)) (wildcardOf_option (pktTpDst pk))
    (wildcardExact pt).

notb (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**

\[
\text{InBuf} := \text{Bag Packet.}
\]

\[
\text{OutBuf} := \text{Bag Packet.}
\]

\[
\text{OFInBuf} := \text{Bag SwitchMsg.}
\]

\[
\text{OFOutBuf} := \text{Bag CtrlMsg.}
\]
Weak Bisimulation

\((H_1, \_\_\_\_)\)
Weak Bisimulation

\((H_1, m) \rightarrow (S_1, pt_1, m)\)
Weak Bisimulation

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

$(H_1, \text{envelope}) \xrightarrow{} (S_1, pt_1, \text{envelope}) \xrightarrow{} (S_2, pt_1, \text{envelope}) \xrightarrow{} (H_2, \text{envelope})$
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

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

\[(H_1, \text{packet}) \rightarrow (S_1, \text{pt}_1, \text{packet}) \rightarrow (S_2, \text{pt}_1, \text{packet}) \rightarrow (H_2, \text{packet})\]
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

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

Application

Configurations

Run-Time System

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

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
Update Semantics

Atomic Updates

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

Thank you to Nate Foster and Arjun Guha. These slides are based on a version from their ECOOP tutorial.