CS 244
Network Verification
Nate Foster
Cornell University
Opening Survey

https://pollev.com/jnfoster
When I need to troubleshoot buggy code...
When I need to troubleshoot buggy code...

Credit: Wikipedia
Software Validation: A Spectrum

Social
- Code reviews
- Pair programming

Methodological
- Test-driven development
- Version control
- Bug tracking

Technological
- Static "linters"
- Fuzzers

Mathematical
- Type systems
- Formal verification

Less formal: techniques are easy to use but may miss some problems in programs

Best practice: all of these techniques should be used!

More formal: eliminate *with certainty* as many problems as possible, but may be hard to use

Credit: Benjamin Pierce
What about computer networks?

"You're On Your Own Mate" —Nick McKeown
Network Verification Pre-History

- Proposed a static analysis to determine network reachability
- Based on an underlying model that captures IP forwarding
- Extension supports richer behaviors like NAT and middleboxes
Network Verification in SDN

Pre-SDN
- Distributed control plane
- Complex data plane
  - Dozens of protocols
  - Tricky, undocumented semantics

Post-SDN
- Centralized control plane
- Streamlined data plane
  - OpenFlow 1.0: only 12 protocols!
  - Clear semantics

Key Insight
- Can instrument control plane to build a model of the data plane behavior
- Can reason statically about network-wide forwarding properties
Plan for Today

Header Space Analysis

NetKAT

Key Questions:
• How to encode networks and properties?
• How to automate reasoning?
• How scalable and how fast?
Packet Model

- Fix the set of headers (Ethernet, IPv4, TCP, UDP, etc.) used by devices
- If $L$ is the total number of bits used to encode all headers...
- Then a packet can be seen as a point in an $L$-dimensional space
- Can also assign each port a unique identifier and add a "pseudo header" to track packet's location
- Formally: $\{0, 1\}^L \times \{1, \ldots, P\}$
Packet forwarding can be viewed as a transformation on header space.
Network devices seem complex, with many different features and protocols...
Transfer Functions

... but ultimately they too can be modeled as transfer functions on packet space
Formalizing Transfer Functions

\[ T \in \text{Header} \times \text{Port} \rightarrow (\text{Header} \times \text{Port}) \text{ Set} \]
Formalizing Transfer Functions

$T \in \text{Header} \times \text{Port} \rightarrow (\text{Header} \times \text{Port}) \text{ Set}$

- 172.24.74.0 255.255.255.0 Port1
- 172.24.128.0 255.255.255.0 Port2
- 171.67.0.0 255.255.0.0 Port3

$T(h, p) = \begin{cases} 
(h,1) & \text{if } \text{dst}_\text{ip}(h) = 172.24.74.x \\
(h,2) & \text{if } \text{dst}_\text{ip}(h) = 172.24.128.x \\
(h,3) & \text{if } \text{dst}_\text{ip}(h) = 171.67.x.x 
\end{cases}$
Symbolic Representation
Scaling Challenges

• We now have a foundational model of forwarding behavior
• But the space of packets is huge...
• ...and the space of functions on that space is larger still...
• Unclear how to realize this model in an implementation that scales well!
Insight: Networks are (Mostly) Uniform

- In practice, most transfer functions transform the packet space in a (mostly) uniform way.
- Sets of packets can be viewed as regions (i.e., hypercubes) in the same N-dimensional space.
- So we can build symbolic representations that manipulate regions of header space rather than individual points in packet space.

![Diagram showing hypercube and flow with header 1xx]
Header Space Algebra

Elements

Every region of header space can be represented as a union of ternary "wildcard expressions" in $\{0, 1, x\}^*$

Operations

- Equivalence & inclusion
- Union
- Intersection
- Difference
- Complement
Intersection

Single-bit intersection

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_i</td>
<td>b'_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>z</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>z</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>

Definition

Intersect bit-wise, yield ∅ if any bit is "z"

Example

11000xxx ∩ xx00010x = 1100010x
Union

Definition
Simply take union of wildcard expressions, simplify if possible

Example
\[1111xxxx \cup 0000xxxx\]

Optimization Example
\[1100xxxx \cup 1000xxxx = 1x00xxxx\]
**Definition**

- Flip each non-wildcard bit, wildcard every other bits
- Result is union of all such expressions

**Example**

\[(010x)^c = 1xxx \cup x0xx \cup xx1x\]
Composing Transfer Functions

We can model network-wide behavior as the *composition* of transfer functions.
We can model network-wide behavior as the composition of transfer functions.

**Question:** how do you reconcile the different input and output types?

**Answer:** "lift" the second function to \((\text{Header} \times \text{Port}) \text{ Set} \rightarrow (\text{Header} \times \text{Port}) \text{ Set}\)
Domain and Range

We can compute the *domain* and *range* of an transfer function symbolically in terms of header spaces represented as wildcard expressions.
Inverse Transfer Function

Can also compute the header space produced by the inverse of a transfer function, yielding a model of the inputs that map to a given set of outputs...
HSA Applications
Reachability

Goal

Want to know whether packet originating at A can get to B

Approach

• Symbolically execute $R_{a \rightarrow b}$ on an "all wildcard" packet "xxxx..."
• Compose transfer functions for the devices path to get $R_{a \rightarrow b}$
• The result models all packets that reach B from A

Extensions

• Waypointing, Blackholing, etc.
Reachability Example

\[ T_B(h, p) = \begin{cases} \{(h, B_1)\} & \text{if } h = \text{xxxxxx10}, p = B_0 \\ \{(h, B_0)\} & \text{if } h = \text{ccccxx01}, p = B_1 \end{cases} \]

\[ T_A(h, p) = \begin{cases} \{(h, A_1)\} & \text{if } h = \text{0011xxx} \\ \{(h, A_0)\} & \text{if } h = \text{0011xxx} \\ \{(h, A_0)\} & \text{if } h = \text{0011xxx} \end{cases} \]

\[ T_D(h, p) = \begin{cases} \{(h & \& 00011111)(01000000, D_1)\} & \text{if } h = \text{160xxxxx}, p = D_0 \\ \{(h & \& 00011111)(01100000, D_0)\} & \text{if } h = \text{110xxxxx}, p = D_1 \end{cases} \]

\[ T_C(h, p) = \begin{cases} \{(h, C_1)\} & \text{if } h = \text{10011xxx} \\ \{(h, C_2)\} & \text{if } h = \text{10011xxx} \\ \{(h, C_0)\} & \text{if } h = \text{0000xxx} \\ \{(h, C_0)\} & \text{otherwise} \end{cases} \]

\[ T_E(h, p) = \begin{cases} \{(h, E_2)\} & \text{if } p = E_0 \\ \{(h, E_2)\} & \text{if } p = E_1 \end{cases} \]
Loop Freedom

Goal

Want to know whether packets can loop infinitely...

Distinction

- **Generic Loop**: a packet loops back to the same switch
- **Infinite Loop**: an *identical packet* loops back to the same switch

Approach

- Use reachability to identify generic loops
- Then analyze header spaces to identify infinite loops
Loop Freedom Example

Finite Loop

Infinite Loop

\[ T_A() \quad T_A^{-1}() \]

\[ T_C() \quad T_C^{-1}() \]

\[ T_B() \quad T_B^{-1}() \]

\[ T_D() \quad T_D^{-1}() \]

\[ a \quad a_0 \quad A \quad A_1 \quad A_2 \]

\[ b \quad b_0 \quad B \quad B_1 \quad B_2 \]

\[ C \quad C_0 \quad C_1 \]

\[ D \quad D_1 \quad D_2 \quad D_0 \]

\[ h_{\text{orig}} \quad h_{\text{ret}} \quad \text{all-x} \]
Performance
Stanford Campus Network (ca. 2012)

~750K IP fwd rule.
~1.5K ACL rules.
~100 Vlans.
Vlan forwarding.
## HSA Performance

On a single machine with 4 cores and 4GB Ram

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generating TF Rules</td>
<td>~150 sec</td>
</tr>
<tr>
<td>Loop Detection Test (30 ports)</td>
<td>~560 sec</td>
</tr>
<tr>
<td>Average Per Port</td>
<td>~18 sec</td>
</tr>
<tr>
<td>Min Per Port</td>
<td>~8 sec</td>
</tr>
<tr>
<td>Max Per Port</td>
<td>~135 sec</td>
</tr>
<tr>
<td>Reachability Test (Avg)</td>
<td>~13 sec</td>
</tr>
</tbody>
</table>
Why I ♥ NetKAT

Theory

Inspires

Applications
Why I ♥ NetKAT

Theory

Applications

- Compilation
- Verification
- New Features
Why I ♥ NetKAT

Theory

\[ \llbracket p \rrbracket \]
denotational semantics

\[ \vdash p \equiv q \]
sound & complete axiomatization

automata theory

Applications

- Compilation
- Verification
- New Features

symbolic representation
NetKAT Roadmap

Language Design & Modeling

Reasoning & Verification

Programming & Compilation
NetKAT Roadmap

Language Design & Modeling

Reasoning & Verification

Programming & Compilation
Essential Features
Essential Features

Forwarding Along Paths

Packet Classification

Packet Modification
Essential Features

Forwarding Along Paths

Packet Classification

Packet Modification

Regular Expressions

+, ;, *
Essential Features

Forwarding Along Paths

Packet Classification

Packet Modification

Regular Expressions
+ , , *

Boolean Algebra
true, false, f=n, a&b, a|b, ¬a
Essential Features

Forwarding Along Paths

Packet Classification

KAT [Kozen '96]

Packet Modification

Regular Expressions

+, ;, *

Boolean Algebra

true, false, f \equiv n, a \& b, a \mid b, \neg a

KA T
Essential Features

Forwarding Along Paths

Packet Classification

KAT [Kozen '96]

Regular Expressions

\[+, ;, \ast\]

Boolean Algebra

true, false, \(f = n\), \(a \& b\), \(a | b\), \(\neg a\)

Packet Modification

Network Primitives

\(f := n\), \(A \rightarrow B\)
Essential Features

- Forwarding Along Paths
- Packet Classification
- Packet Modification

Regular Expressions
+, *, ;

Boolean Algebra
true, false, \( f=\neg n \), \( a \& b \), \( a | b \), \( \neg a \)

Network Primitives
\( f:=n \), \( A \rightarrow B \)

KAT [Kozen '96]
NetKAT ['14]

NetKAT [Kozen '96]
“For all packets incoming on port 88 of switch 6, set the destination IP address to 10.0.0.1 and multicast the packet out of ports 50 and 51.”
Design Goal: Modular Composition

program fragments can be composed to form larger programs
Sequential Composition

“First filter out untrusted traffic, then forward.”
Sequential Composition

if dstport=22 then false
else true

;  

if dest=10.0.0.1 then port:=1
elif dest=10.0.0.2 then port:=2
elif dest=10.0.0.3 then port:=3
else false

“First filter out untrusted traffic, then forward.”
Parallel Composition

“Execute both Monitor and Forward on all incoming packets”

**Multicast:** $\text{port}_1 + \text{port}_2$
Language Design & Modeling

Reasoning & Verification

Programming & Compilation
Encoding Networks

Forwarding tables and topologies can be represented in NetKAT using straightforward encodings.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

```
if dstport=22 then false
elif srcip=10.0.0.1 then port := 1
else port := 2
```
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology

(topology; switch)*
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology.

(topology; switch)*
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology.
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology

(topology; switch)∗
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology

$(\text{topology} ; \text{switch})*$
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology.

(topology; switch)*
Given a network encoded this way, we’d like to be able to automatically answer questions like:

“Does the network forward from ingress to egress?”

Can reduce this question (and others) to program equivalence

\[ \text{in; (topology; switch)*; out} \neq \text{false} \]
pol ::=  
| false  
| true 
| field = val  
| field ::= val  
| pol₁ + pol₂  
| pol₁ ; pol₂  
| ¬pol  
| pol* 
| B → A
Denotational Semantics

\[
pol ::= \begin{array}{l}
\text{false} \\
\text{true} \\
\text{field } = \text{ val} \\
\text{field } := \text{ val} \\
\text{pol}_1 + \text{pol}_2 \\
\text{pol}_1 ; \text{pol}_2 \\
\neg \text{pol} \\
\text{pol}^* \\
B \rightarrow A
\end{array}
\]

Local: input-output behavior of switches

\[ [\Phi(\text{pol})] \in \text{Packets} \rightarrow \text{Packets} \]
Denotational Semantics

\[ \text{pol ::= } \]
\[ \quad | \text{false} \]
\[ \quad | \text{true} \]
\[ \quad | \text{field = val} \]
\[ \quad | \text{field := val} \]
\[ \quad | \text{pol}_1 + \text{pol}_2 \]
\[ \quad | \text{pol}_1 ; \text{pol}_2 \]
\[ \quad | \neg \text{pol} \]
\[ \quad | \text{pol}^* \]
\[ \quad | B \rightarrow A \]

Local: input-output behavior of switches

\[ [\Phi(\text{pol})] \in \text{Packets } \rightarrow \text{Packets} \]

Global: network-wide paths

\[ [\text{pol}] \in \text{Histories } \rightarrow \text{Histories} \]
### Kleene Algebra Axioms
\[
\begin{align*}
p + (q + r) &= (p + q) + r \\
p + q &= q + p \\
p + \text{false} &= p \\
p + p &= p \\
p; (q; r) &= (p; q); r \\
p; (q + r) &= p•q + p•r \\
(p + q); r &= p; r + q; r
\end{align*}
\]
\[
\text{true}; p &= p \\
p &= p; \text{true} \\
\text{false}; p &= \text{false} \\
p; \text{false} &= \text{false} \\
\text{true} + p; p^* &= p^* \\
\text{true} + p^*; p &= p^* \\
p + q; r + r \Rightarrow p^*; q + r \Rightarrow r \\
p + q; r + q = q \Rightarrow p; r^* + q = q
\]

### Boolean Algebra Axioms
\[
\begin{align*}
a + (b; c) &= (a + b); (a + c) \\
a + \text{true} &= \text{true} \\
a + \neg a &= \text{true} \\
a; b &= b; a \\
a; \neg a &= \text{false} \\
a; a &= a
\end{align*}
\]

### Packet Axioms
\[
\begin{align*}
f := n; f' := n' &\Rightarrow f' := n'; f := n \quad \text{if } f \neq f' \\
f := n; f' := n' &\Rightarrow f := n \quad \text{if } f \neq f' \\
f := n; f := n &\Rightarrow f := n \\
f = n; f := n &\Rightarrow f = n \\
f := n; f := n' &\Rightarrow f := n' \\
f = n; f = n' &\Rightarrow \text{false} \quad \text{if } n \neq n' \\
A \rightarrow B; f = n &\Rightarrow f = n; A \rightarrow B \quad \text{if } f \not\in \{\text{switch, port}\} \\
\Sigma_i f = n_i &\Rightarrow \text{true}
\end{align*}
\]
**Kleene Algebra Axioms**

\[
\begin{align*}
p + (q + r) &= (p + q) + r \\
p + q &= q + p \\
p + \text{false} &= p \\
p + p &= p \\
p; (q; r) &= (p; q) ; r \\
p; (q + r) &= p; q + r \\
p + q; r + r &= r \implies p^*; q + r = r \\
p + q; r + q &= q \implies p; r^* + q = q
\end{align*}
\]

**Boolean Algebra Axioms**

\[
\begin{align*}
a + (b ; c) &= (a + b) ; (a + c) \\
a + \text{true} &= \text{true} \\
a + \neg a &= \text{true} \\
a ; b &= b ; a \\
a + \neg a &= \text{false} \\
a ; a &= \text{true} \\
a + \text{true} &= \text{true} \\
a + \neg a &= \text{false} \\
a ; \text{false} &= \text{false}
\end{align*}
\]

**NetKAT Axioms**

\[
\begin{align*}
f := n ; f' = n' = f' = n' ; f := n & \text{ if } f \neq f' \\
f := n ; f = n = f := n \\
f = n ; f := n = f = n \\
f := n ; f := n' = f := n' \\
f := n ; f := n' = \text{false} & \text{ if } n \neq n' \\
f = n ; f = n = f = n ; A \rightarrow B & \text{ if } f \notin \{ \text{switch, port} \} \\
\Sigma_i f = n_i = \text{true}
\end{align*}
\]

**Soundness:** If \( \vdash p \equiv q \), then \([p] = [q]\)

**Completeness:** If \([p] = [q]\), then \(\vdash p \equiv q\)
Decision Procedure

Decides program equivalence fully automatically!

**Theoretical Insight:** NetKAT programs $\leftrightarrow$ NetKAT automata
Decision Procedure

Decides program equivalence fully automatically!

Theoretical Insight: NetKAT programs $\leftrightarrow$ NetKAT automata
Decision Procedure

Decides program equivalence fully automatically!

**Theoretical Insight:** NetKAT programs $\leftrightarrow$ NetKAT automata

Algorithm checks bisimilarity of automata
Local Program

pol_A

pol_B
**Local Program**

```
port := 3

1   2
A   3  4
   3   4
B   5
   6
```

```???
```
Local Program

port = 1; tag = 1; port = 3 +
port = 2; tag = 2; port = 3

???
Local Program

A

1

2

3

4

5

6

B

port=1; tag:=1; port:=3
port=2; tag:=2; port:=3

+ tag=1; port:=5
+ tag=2; port:=6
Local Program

Tedious for programmers… difficult to get right!

```
port=1; tag:=1; port:=3
  +
port=2; tag:=2; port:=3

tag=1; port:=5
  +
tag=2; port:=6
```
Global Program

A

pol

B

1 2

3 4

5 6
Global Program

port = 1; A → B; port := 5
+ port = 2; A → B; port := 6
Global Program

Simple and elegant!

port=1; A→B; port:=5
+ port=2; A→B; port:=6
Virtual Program
Virtual Program

virtual "big switch"
Virtual Program

Even simpler!

```
port = 1;
port = 5 + port = 2;
port = 6
```

virtual "big switch"
Virtual Program

Even simpler!

port = 1;
port := 5;

+ 

port = 2; port := 6

virtual "big switch"
Virtual Program

Even simpler!

Virtual "big switch"
Can implement **multiple** arbitrary **virtual networks** on top of **single physical network**

Even simpler!

```
Virtual Program

virtual "big switch"

port=1; port:=5 + port=2; port:=6

firewall
```
Compilation [ICFP '15]
Compilation \[ICFP \, ’15\]

Local Compiler

- Pattern Actions
  - dstpt = drop
  - srcpt = fwd 1
  - * = fwd 2

\[\text{~ 100x faster than competitors}\]
Local program

Global Compiler

Local Compiler

network-wide behavior

~ 100x faster than competitors

~ 100x faster than competitors

3

Global program

Global Compiler

Local program

Pattern

Action

dstpt= drop
srcpt= fwd 1
* fwd 2

Compilation [ICFP '15]
Compilation [ICFP '15]

- Virtual Compiler: abstract topologies
- Global Compiler: network-wide behavior
- Local Compiler: ~100x faster than competitors

- Pattern: dstpt=, srcpt=, fwd 1, fwd 2
  - Action: drop, fwd 1, fwd 2

*~ 100x faster than competitors*
Compilation [ICFP '15]

- **Virtual Compiler**: abstract topologies
- **Global Compiler**: network-wide behavior
- **Local Compiler**: ~100x faster than competitors

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=</td>
<td>fwd 1</td>
</tr>
<tr>
<td>*</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>
1. Adding Extra State
"Tagging"
Global Compilation

1. Adding Extra State
   "Tagging"

2. Avoiding Duplication
   (naive tagging is unsound!)
Global Compilation

Global Program
Global Compilation

Global Program

Automaton Translation

NetKAT NFA
Global Compilation

Global Program

Automaton Translation

NetKAT NFA

Determinization

NetKAT DFA
Global Compilation

Global Program → Automaton Translation

NetKAT NFA

Heuristic Minimization

NetKAT DFA → Determinization
Global Compilation

Global Program → Automaton Translation

NetKAT NFA

NetKAT DFA

Heuristic Minimization

Determinization

Local Program
Questions?
false drops its input
true copies its input
f = n copies its input if pk.f = n and otherwise drops it
\[ f = n \] copies its input if \( pk.f = n \) and otherwise drops it.

\[ f = n \] when \( pk.f = n \)

\[ f = n \] when \( pk.f \neq n \)

\( p ::= \) false

true

\( f = n \)

\( f := n \)

\( p_1 + p_2 \)

\( p_1 ; p_2 \)

\( \neg p \)

\( p^* \)

\( A \rightarrow B \)
\( f := n \) sets the input's f component to \( n \)
\( p := \text{false} \mid \text{true} \mid f = n \mid f := n \mid p_1 + p_2 \mid p_1 ; p_2 \mid \neg p \mid p^* \mid A \rightarrow B \)

\( p_1 + p_2 \) duplicates the input, sends one copy to each sub-policy, and takes the \textit{union} of their outputs.
\( p ::= \text{false} \\
\text{true} \\
f = n \\
f := n \\
p_1 + p_2 \\
p_1; p_2 \\
\neg p \\
p^* \\
A \rightarrow B \)

\( \langle \text{pk},.. \rangle \)

\( \langle \text{pk}_1,.. \rangle,\langle \text{pk}_2,.. \rangle \)

\( p_1; p_2 \) runs the input through \( \text{pol}_1 \) and then runs every output produced by \( p_1 \) through \( p_2 \)
\[p ::= \begin{align*}
& \text{false} \\
& \text{true} \\
& f = n \\
& f := n \\
& p_1 + p_2 \\
& p_1 ; p_2 \\
& \neg p \\
& p^* \\
& A \rightarrow B
\end{align*}\]

\[\neg p\] drops the input if \( p \) produces any output and copies it otherwise.
p* repeatedly runs packets through p to a fixpoint
A→B duplicates the packet and moves it across the link

p ::= false
| true
| f = n
| f := n
| p₁ + p₂
| p₁ ; p₂
| ¬p
| p*
| A→B

⟨pk,..⟩

A→B

⟨pk[sw=B], pk,…⟩

when pk.sw = A
A → B duplicates the packet and moves it across the link

\[
\begin{align*}
&\langle \text{pk,..} \rangle \\
&\text{A} \rightarrow \text{B} \\
\end{align*}
\]

when \( \text{pk.sw} = \text{A} \)

\[
\begin{align*}
&\langle \text{pk}[\text{sw=B}], \text{pk,..} \rangle \\
&\text{A} \rightarrow \text{B} \\
\end{align*}
\]

when \( \text{pk.sw} \neq \text{A} \)
NetKAT Semantics
NetKAT Semantics

\[ [p] \in \text{History} \rightarrow \text{History Set} \]
NetKAT Semantics

\[ [p] \in \text{History} \rightarrow \text{History Set} \]
\[ [\text{true}] \ h = \{ \ h \} \]
NetKAT Semantics

\[ p \] \in \text{History} \rightarrow \text{History Set}
\[ \text{true} \] \ h = \{ h \}
\[ \text{false} \] \ h = \{ \}
NetKAT Semantics

\[
\begin{align*}
[p] & \in \text{History} \rightarrow \text{History Set} \\
[\text{true}] & \quad h = \{ h \} \\
[\text{false}] & \quad h = \{ \} \\
[f = n] & \quad \text{pk} :: h = \begin{cases}
\{ \text{pk} :: h \} & \text{if } \text{pk.f} = n \\
\{ \} & \text{otherwise}
\end{cases}
\end{align*}
\]
NetKAT Semantics

\[ \llbracket p \rrbracket \in \text{History} \rightarrow \text{History Set} \]

\[ \llbracket \text{true} \rrbracket \ h = \{ h \} \]

\[ \llbracket \text{false} \rrbracket \ h = {} \]

\[ \llbracket f = n \rrbracket \ pk :: h = \begin{cases} \{ pk :: h \} & \text{if } pk.f = n \\ {} & \text{otherwise} \end{cases} \]

\[ \llbracket \neg p \rrbracket \ h = \{ h \} \setminus \llbracket p \rrbracket \ h \]
NetKAT Semantics

\[ [p] \in \text{History} \rightarrow \text{History Set} \]
\[ [\text{true}] \ h = \{ h \} \]
\[ [\text{false}] \ h = \{ \} \]
\[ [f = n] \ \text{pk} :: h = \begin{cases} \{ \text{pk} :: h \} & \text{if pk.f = n} \\ \{ \} & \text{otherwise} \end{cases} \]
\[ [\neg p] \ h = \{ h \} \setminus [p] \ h \]
\[ [f := n] \ \text{pk} :: h = \{ \text{pk}[f := n] :: h \} \]
NetKAT Semantics

\[ [p] \in \text{History} \to \text{History Set} \]
\[ [\text{true}] \ h = \{ h \} \]
\[ [\text{false}] \ h = \{ \} \]
\[ [f = n] \ pk :: h = \begin{cases} \{ pk :: h \} & \text{if pk.f = n} \\ \{ \} & \text{otherwise} \end{cases} \]
\[ [\neg p] \ h = \{ h \} \setminus [p] \ h \]
\[ [f := n] \ pk :: h = \{ pk[f:=n] :: h \} \]
\[ [p_1 + p_2] \ h = [p_1] \ h \cup [p_2] \ h \]
NetKAT Semantics

⟦p⟧ ∈ History → History Set

⟦true⟧ h = { h }

⟦false⟧ h = {}

⟦f = n⟧ pk :: h = \{ pk :: h \} if pk.f = n

\{ \} otherwise

⟦¬p⟧ h = \{ h \} \\⟦p⟧ h

⟦f := n⟧ pk :: h= \{ pk[f:=n] :: h \}

⟦p₁ + p₂⟧ h = ⟦p₁⟧ h u ⟦p₂⟧ h

⟦p₁; p₂⟧ h = (⟦p₁⟧ \cdot ⟦p₂⟧) h

f,g ∈ History → History Set

(f • g) h = U \{ g h' | h'∈ f h \}
NetKAT Semantics

⟦p⟧ ∈ History → History Set
⟦true⟧ h = { h }
⟦false⟧ h = {}
⟦f = n⟧ pk :: h = \{ pk :: h \} if pk.f = n
{} otherwise
⟦¬p⟧ h = { h } \ ⟦p⟧ h
⟦f := n⟧ pk :: h = { pk[f:=n] :: h }
⟦p₁ + p₂⟧ h = ⟦p₁⟧ h \cup ⟦p₂⟧ h
⟦p₁ ; p₂⟧ h = (⟦p₁⟧ • ⟦p₂⟧) h
⟦p*⟧ h = ( ∪ i ⟦p⟧^i h)

f,g ∈ History → History Set
(f • g) h = ∪ \{ g h’ | h’ ∈ f h \}
### NetKAT Semantics

\[ [p] \in \text{History} \rightarrow \text{History Set} \]
\[ [\text{true}] \ h = \{ h \} \]
\[ [\text{false}] \ h = \{} \]
\[ [f = n] \ pk :: h = \begin{cases} \{ pk :: h \} & \text{if pk.f = n} \\ \{} & \text{otherwise} \end{cases} \]
\[ [\neg p] \ h = \{ h \} \setminus [p] \ h \]
\[ [f := n] \ pk :: h = \{ pk[f := n] :: h \} \]
\[ [p_1 + p_2] \ h = [p_1] \ h \cup [p_2] \ h \]
\[ [p_1 ; p_2] \ h = ([p_1] \cdot [p_2]) \ h \]
\[ [p^*] \ h = (\bigcup_i [p]^i \ h) \]
\[ [A \rightarrow B] \ pk :: h = \begin{cases} \{ pk[sw := B] :: pk :: h \} & \text{if pk.sw = A} \\ \{} & \text{otherwise} \end{cases} \]

\( f, g \in \text{History} \rightarrow \text{History Set} \)
\( (f \cdot g) \ h = \bigcup \{ g \ h' \mid h' \in f \ h \} \)