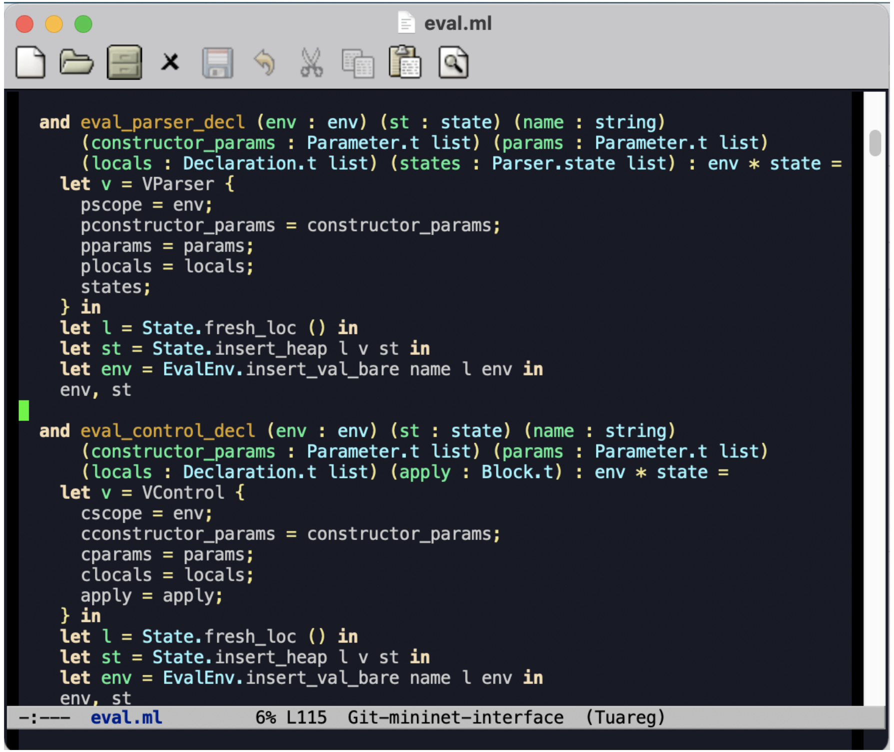
CS 244
Network Verification

Nate Foster
Cornell University

Opening Survey

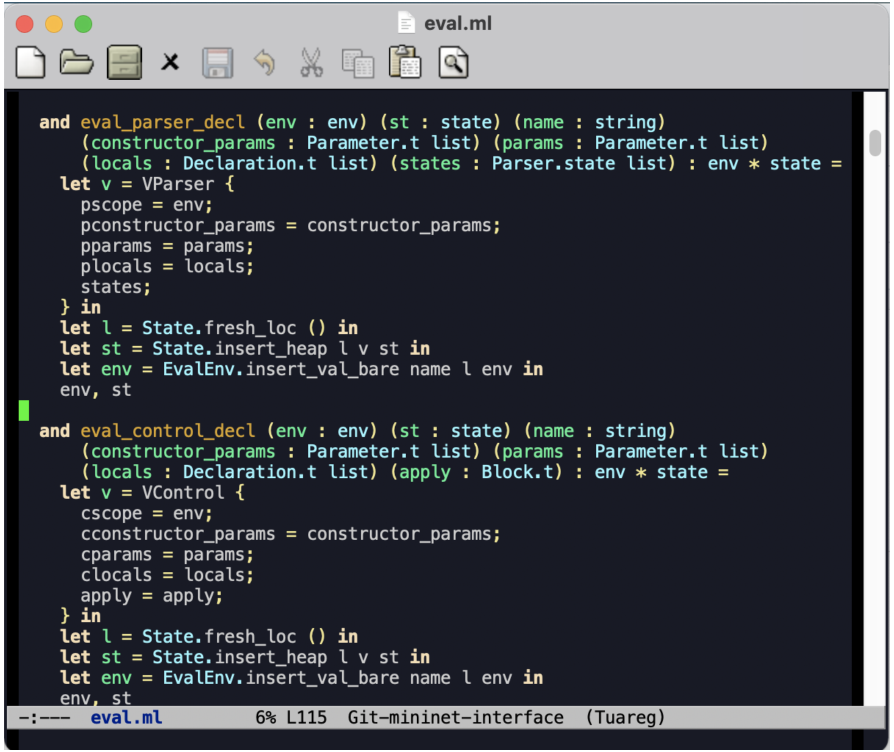
https://pollev.com/jnfoster

When I need to troubleshoot buggy code...



Credit: Wikipedia

When I need to troubleshoot buggy code...



eval.ml

```
and eval_parser_decl (env : env) (st : state) (name : string)
    (constructor_params : Parameter.t list) (params : Parameter.t list)
    (locals : Declaration.t list) (states : Parser.state list) : env * state =
  let v = VParser {
    pscope = env;
    pconstructor_params = constructor_params;
    pparams = params;
    plocals = locals;
    states;
  } in
  let l = State.fresh_loc () in
  let st = State.insert_heap l v st in
  let env = EvalEnv.insert_val_bare name l env in
  env, st

and eval_control_decl (env : env) (st : state) (name : string)
    (constructor_params : Parameter.t list) (params : Parameter.t list)
    (locals : Declaration.t list) (apply : Block.t) : env * state =
  let v = VControl {
    cscope = env;
    cconstructor_params = constructor_params;
    cparams = params;
    clocals = locals;
    apply = apply;
  } in
  let l = State.fresh_loc () in
  let st = State.insert_heap l v st in
  let env = EvalEnv.insert_val_bare name l env in
  env, st
-:---  eval.ml          6% L115  Git-mininet-interface  (Tuareg)
```

Credit: Wikipedia

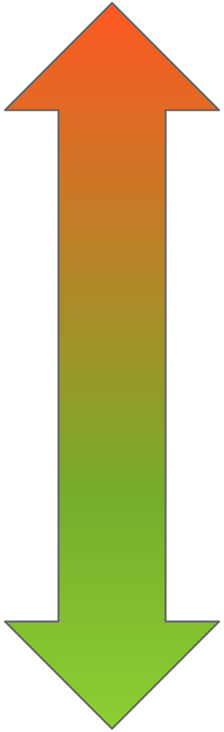
Software Validation: A Spectrum

Social
- Code reviews
- Pair programming

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

Technological
- Static "linters"
- Fuzzers

Mathematical
- Type systems
- Formal verification



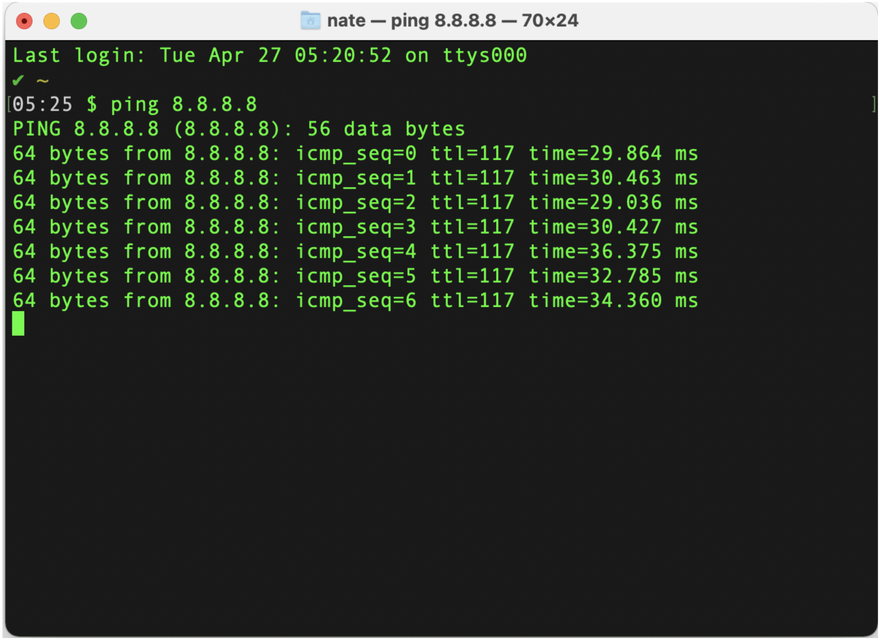
Less formal: techniques are easy to use but may miss some problems in programs

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

On Static Reachability Analysis of IP Networks

Geoffrey G. Xie* Jibin Zhan† David A. Maltz† Hui Zhang†
Albert Greenberg‡ Gisli Hjalmtysson‡ Jennifer Rexford‡

ABSTRACT

The primary purpose of a network is to provide reachability between applications running on end hosts. In this paper, we describe how to compute the reachability a network provides from a snapshot of the configuration state from each of the routers. Our primary contribution is the precise definition of the *potential reachability* of a network and a substantial simplification of the problem through a unified modeling of packet filters and routing protocols. In the end, we reduce a complex, important practical problem to computing the transitive closure to set union and intersection operations on reachability set representations. We then extend our algorithm to model the influence of packet transformations (e.g., by NATs or ToS remapping) along the path. Our technique for static analysis of network reachability is valuable for verifying the intent of the network designer, troubleshooting reachability problems, and performing "what-if" analysis of failure scenarios.

***Index Terms*—Routing, Static Configuration Analysis.**

I. INTRODUCTION

While the ultimate goal of networking is to enable communication between hosts that are not directly connected, a wide variety of mechanisms are being used to *limit* the set of destinations the hosts can reach. For example, backbone networks may provide Virtual Private Network services to connect only remote offices belonging to the same enterprise, and enterprise networks themselves are often segmented into departments or offices whose hosts must

Determining what kinds of packets can be exchanged between two hosts connected to a network is a difficult and critical problem facing network designers and operators. To our knowledge, the problem is largely unexamined in the networking research literature. Solving the problem requires knowing far more than the network's topology or the routing protocols it uses. For example, despite having a route to a remote end-point, a sender's packets may be discarded by a packet filter on one of the links in the path. The network's packet filters, routing policies, and packet transformations all must be taken into account to even ask the simple and very important question of "can these two hosts communicate?"

This paper crystallizes the problem of calculating the *reachability* provided by a network. By mapping packet filters, routing information, and packet transformations to a single unified model of reachability we have determined how to transform this seemingly intractable problem into a classical graph problem that can be solved with polynomial time algorithms such as transitive closure. This is the primary contribution of this paper.

A. *Advantages of Automated Static Analysis*

Currently, the common practice to determine if packets can reach from one point in a network to another is to use tools such as `ping` and `traceroute` to send probe traffic that experimentally test whether reachability exists. In contrast, we have developed a *static-analysis* approach that can be applied even if only a description of the network is available. Static analysis has many advantages

Network Verification in SDN

Pre-SDN

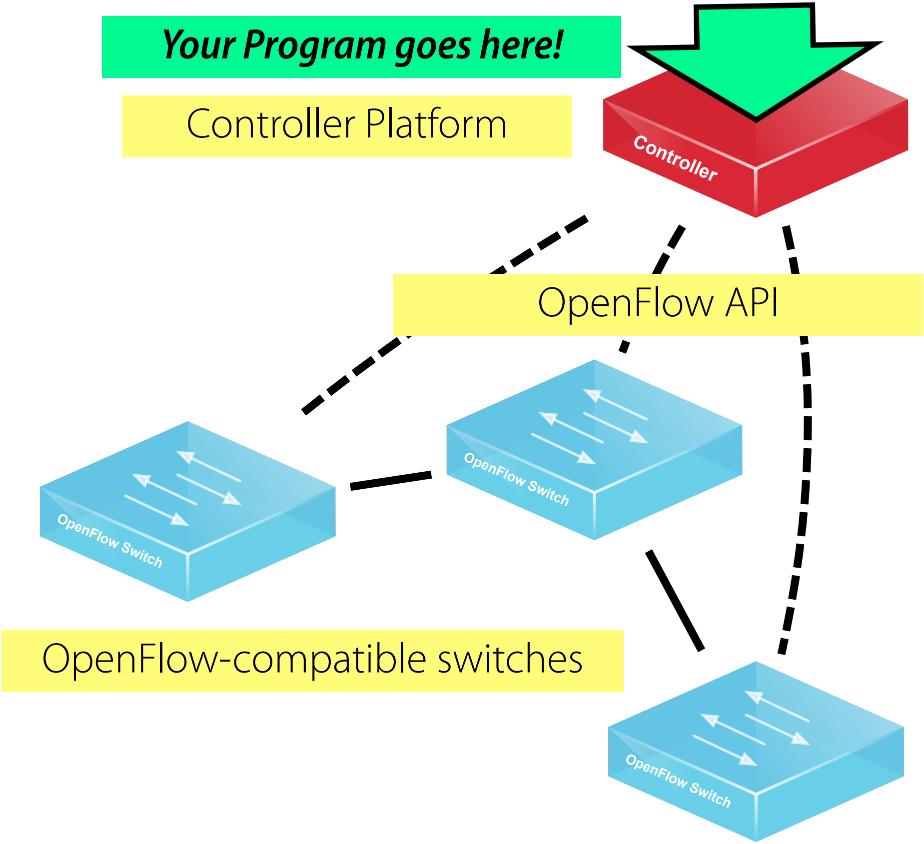
- Distributed control plane
- Complex data plane
  - Dozens of protocols
  - Tricky, undocumented semantics

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:

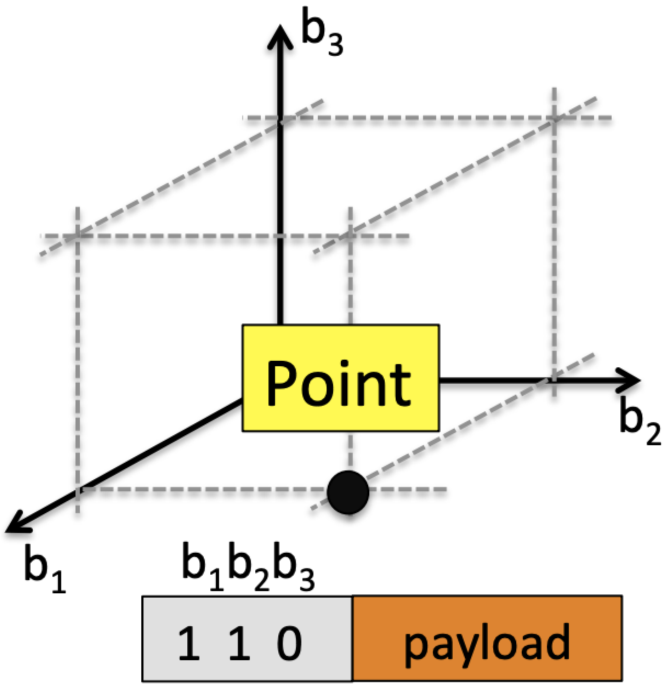
- How to encode networks and properties?
- How to automate reasoning?
- ~~How scalable and how fast?~~

HSA [NSDI '12]

Slide credits: Peyman Kazemian

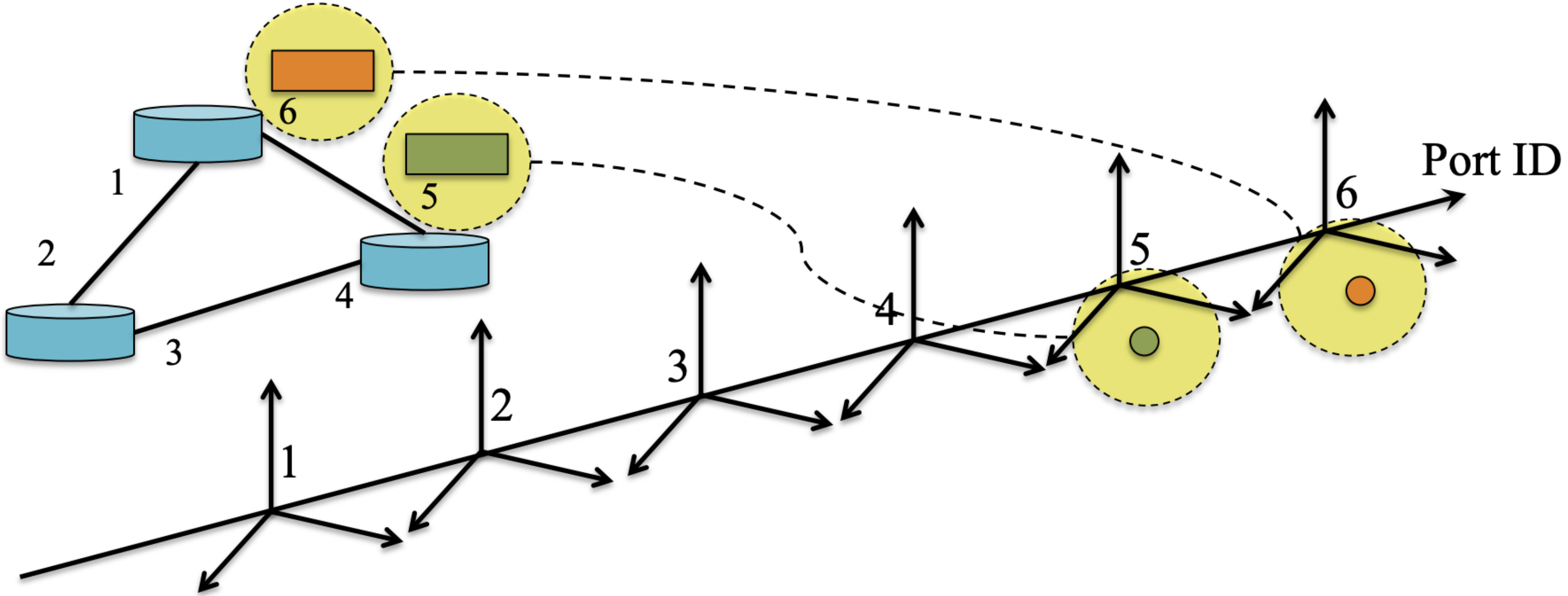
Packet Model

- 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: $\{0, 1\}^L \times \{1, ..., P\}$



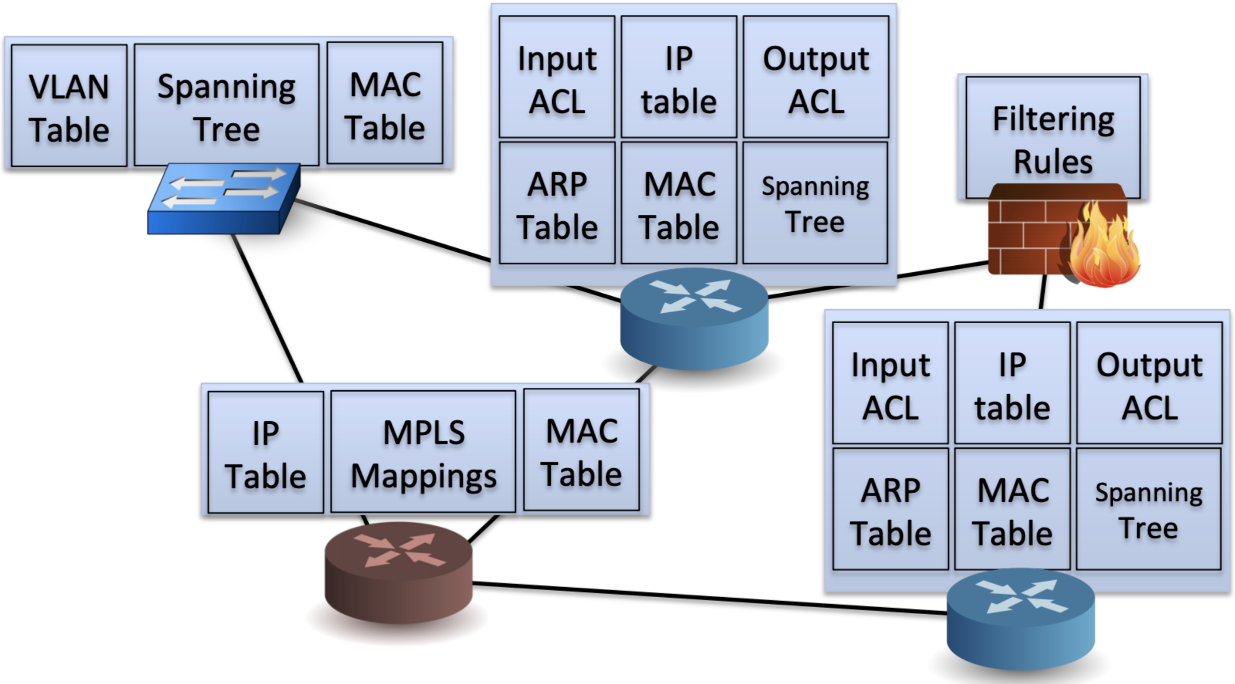
Packet with header 110

Forwarding Model



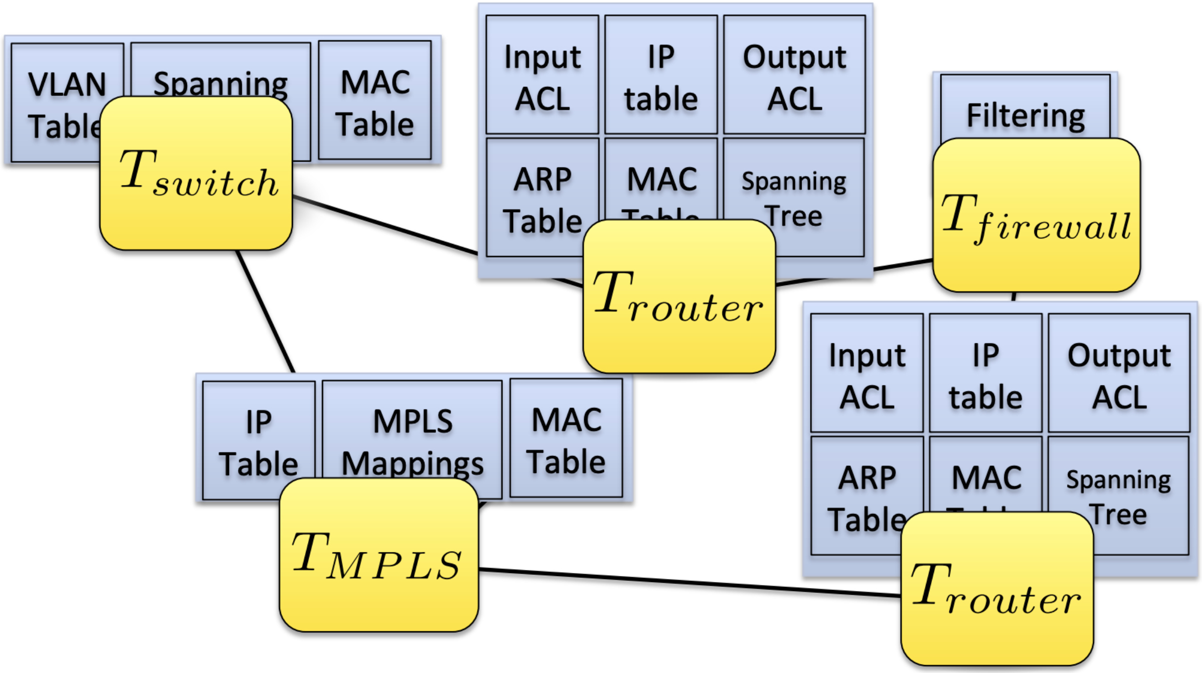
Packet forwarding can be viewed as a transformation on *header space*

Device Complexity



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

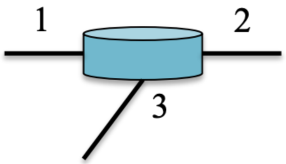
Formalizing Transfer Functions

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

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

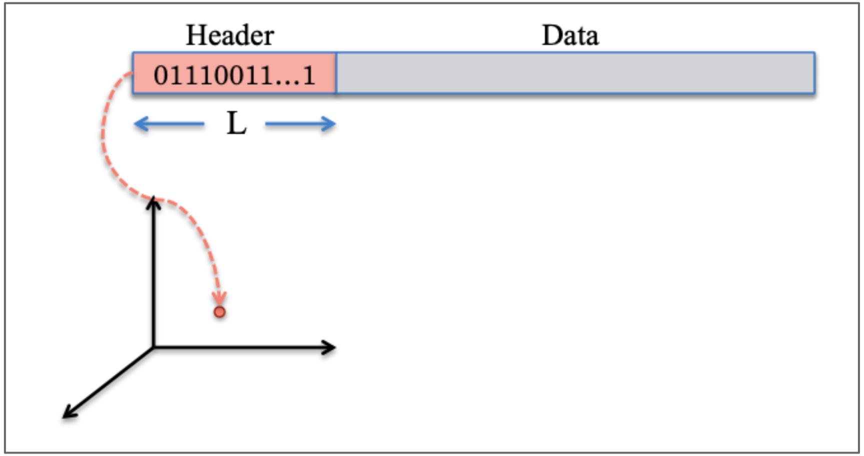


$$T(h, p) = \begin{cases} (h,1) & \text{if } dst\_ip(h) = 172.24.74.x \\ (h,2) & \text{if } dst\_ip(h) = 172.24.128.x \\ (h,3) & \text{if } dst\_ip(h) = 171.67.x.x \end{cases}$$

Symbolic Representation

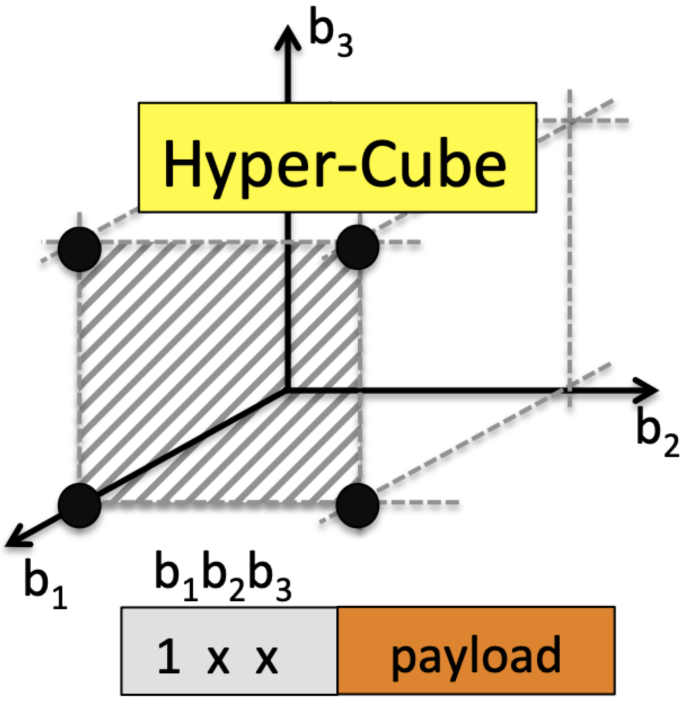
Scaling Challenges

- 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



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

b_i \ b'_i	0	1	x
0	0	z	0
1	z	1	1
x	0	1	x

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 ∪ 0000xxxx
```

Optimization Example

```
1100xxxx ∪ 1000xxxx = 1x00xxxx
```

Complement

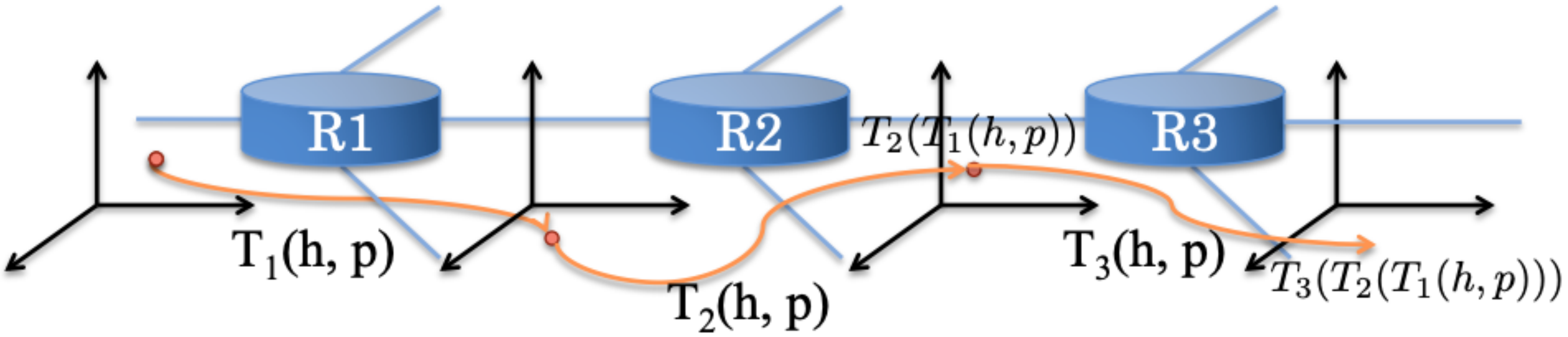
Definition

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

Example

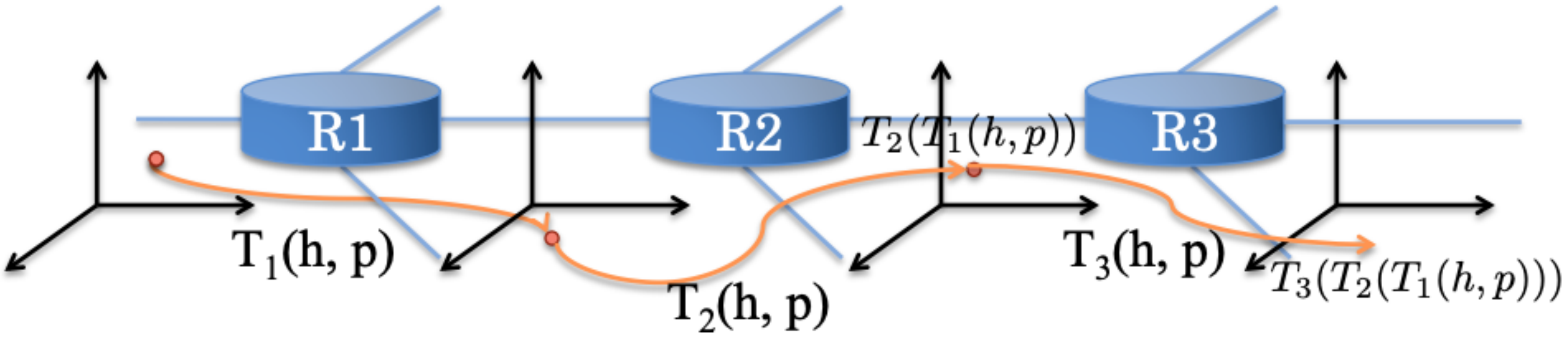
$(010x)^c = 1xxx \cup x0xx \cup xx1x$

Composing Transfer Functions



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

Composing Transfer Functions

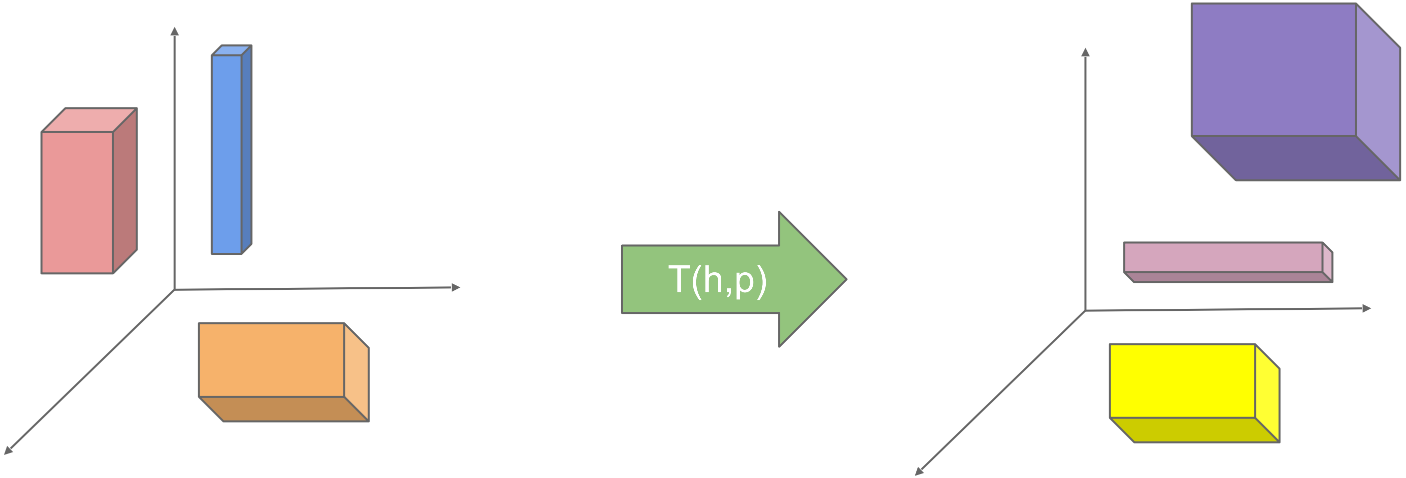


We can model network-wide behavior as the *composition* of transfer functions

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

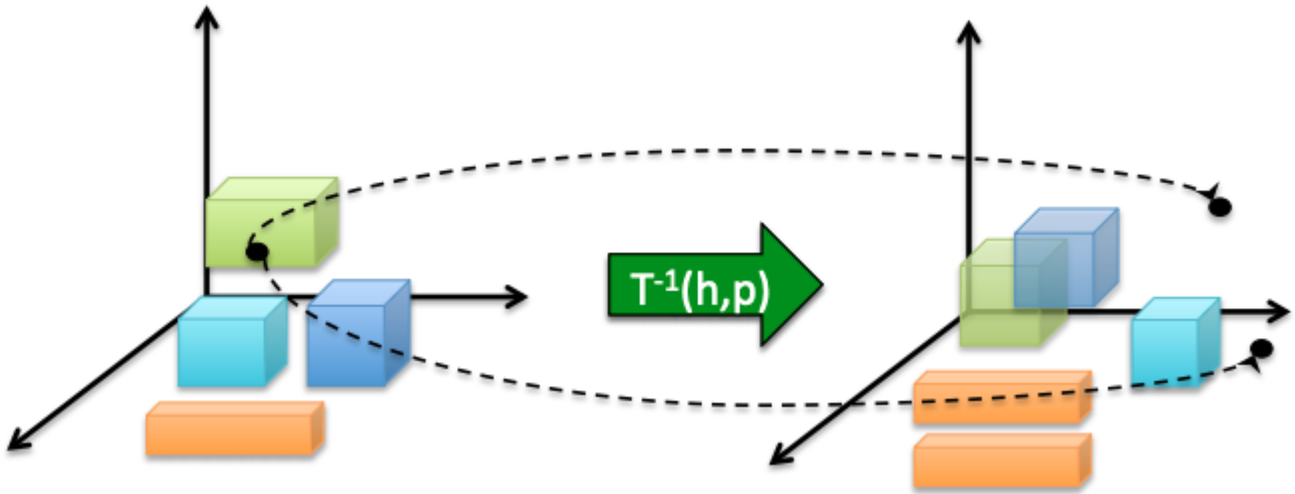
Answer: "lift" the second function to (Header × Port) Set → (Header × Port) Set

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

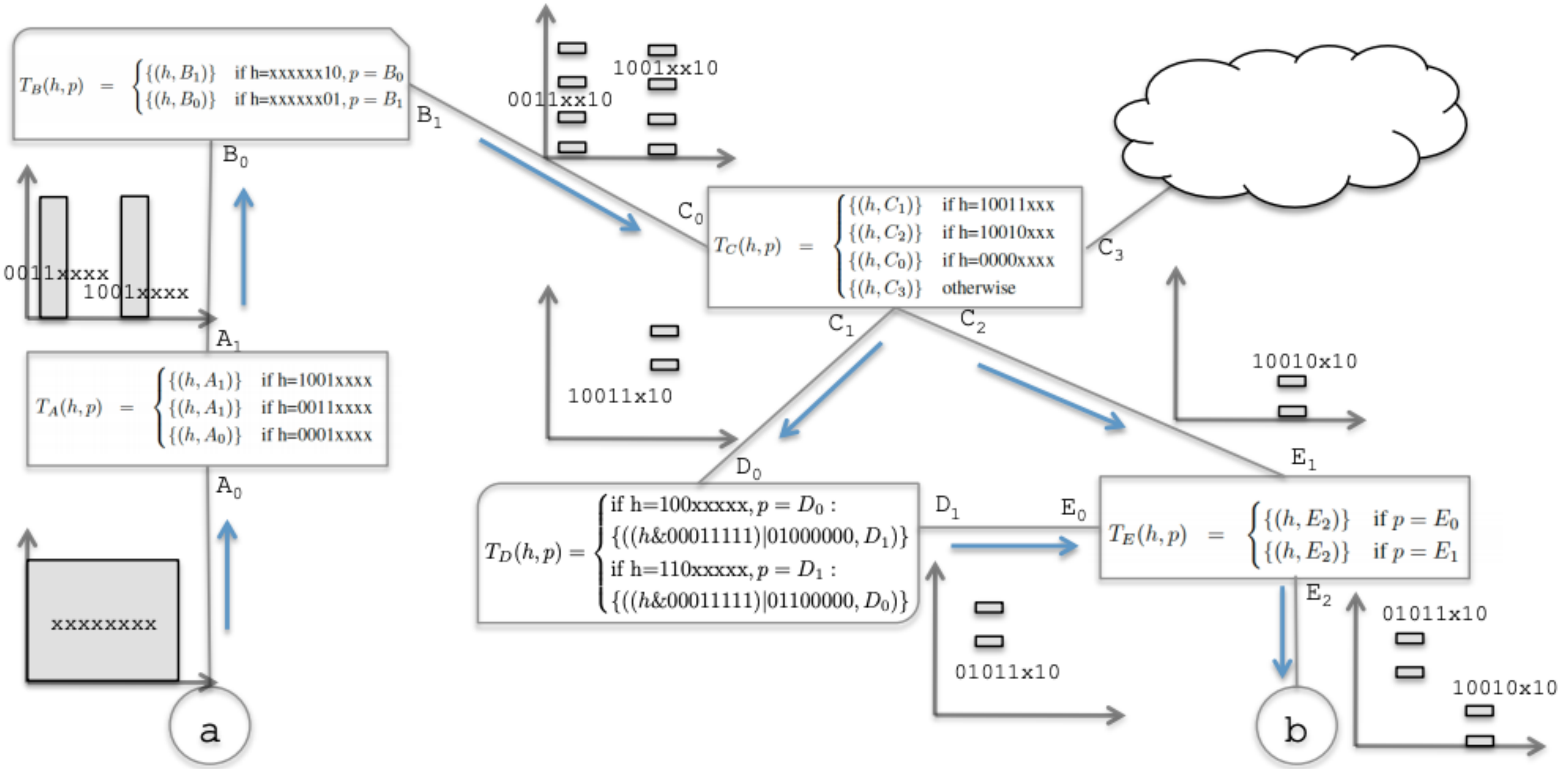
Approach

- 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 [NSDI '12, Fig 2]



Loop Freedom

Goal

Want to know whether packets can loop infinitely…

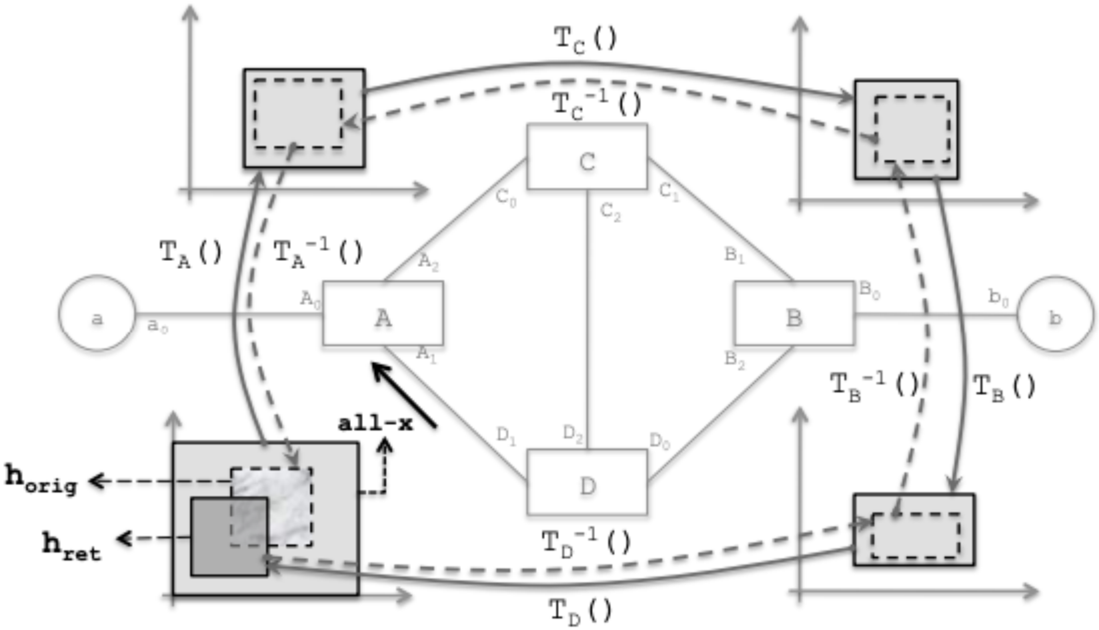
Distinction

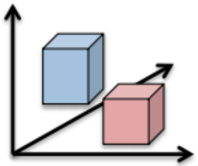
- **Generic Loop:** a packet loops back to the same switch
- **Infinite Loop:** an *identical packet* loops back to the same switch

Approach

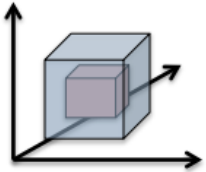
- Use reachability to identify generic loops
- Then analyze header spaces to identify infinite loops

Loop Freedom Example

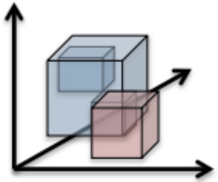




Finite Loop



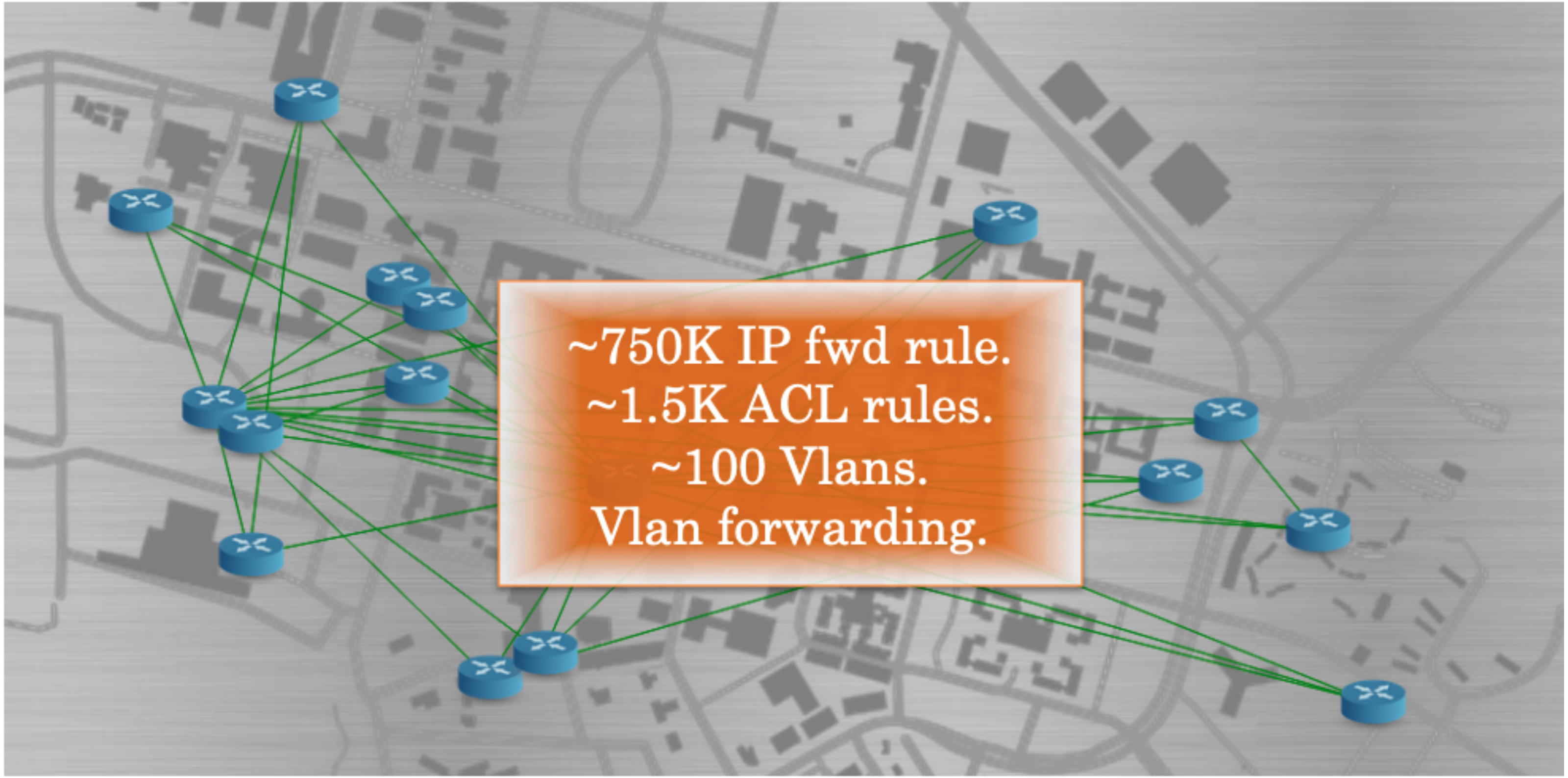
Infinite Loop



?

Performance

Stanford Campus Network (ca. 2012)



HSA Performance

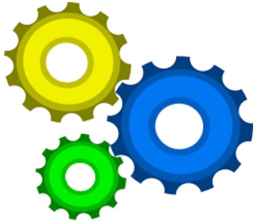
On a single machine with 4 cores and 4GB Ram

Generating TF Rules	~150 sec
Loop Detection Test (30 ports)	~560 sec
Average Per Port	~18 sec
Min Per Port	~ 8 sec
Max Per Port	~ 135 sec
Reachability Test (Avg)	~13 sec

NetKAT [POPL '14]

Why I ♥ NetKAT

Theory

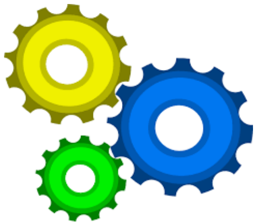


Applications

Why I ♥ NetKAT

Inspires

Theory

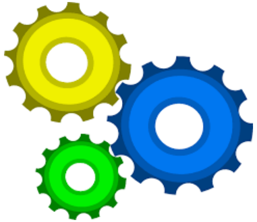


Applications

- Compilation
- Verification
- New Features

Why I ❤ NetKAT

Inspires



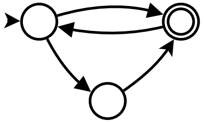
Theory

$[\![p]\!]$

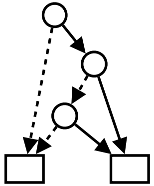
denotational semantics

$\vdash p \equiv q$

sound & complete axiomatization



automata theory



symbolic representation

Applications

- Compilation
- Verification
- New Features

NetKAT Roadmap

Language Design & Modeling

Reasoning & Verification

Programming & Compilation

NetKAT Roadmap

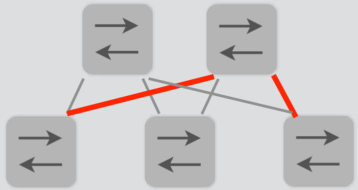
Language Design & Modeling

Reasoning & Verification

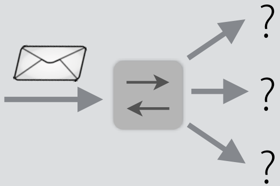
Programming & Compilation

Essential Features

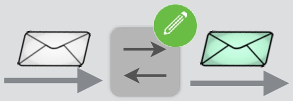
Essential Features



Forwarding Along Paths

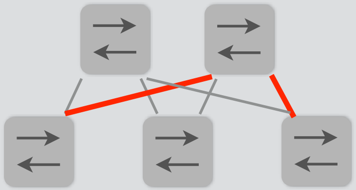


Packet Classification

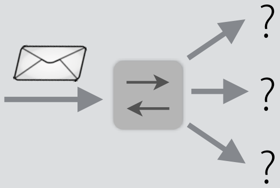


Packet Modification

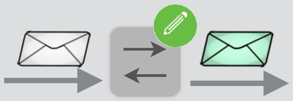
Essential Features



Forwarding Along Paths



Packet Classification

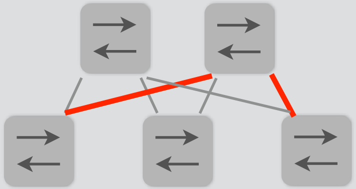


Packet Modification

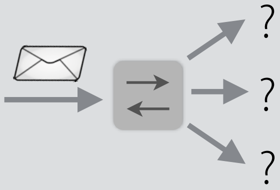
Regular Expressions

+, ;, *

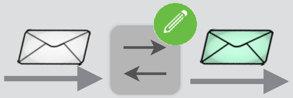
Essential Features



Forwarding Along Paths



Packet Classification



Packet Modification

Regular Expressions

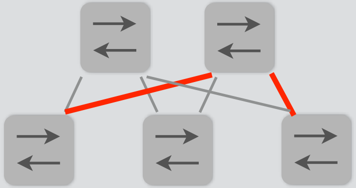
+, ;, *

Boolean Algebra

true, false, f=n,

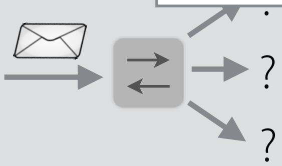
a&b, a|b, ¬a

Essential Features

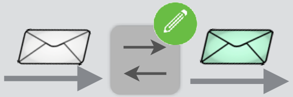


Forwarding Along Paths

KAT [Kozen '96]



Packet Classification



Packet Modification

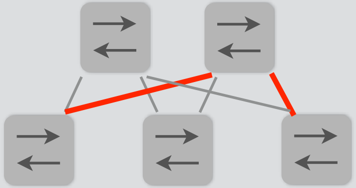
Regular Expressions

+, ;, *

Boolean Algebra

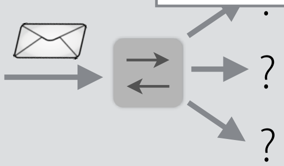
true, false, f=n,
a&b, a|b, ¬a

Essential Features

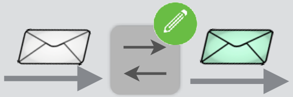


Forwarding Along Paths

KAT [Kozen '96]



Packet Classification



Packet Modification

Regular Expressions

+, ;, *

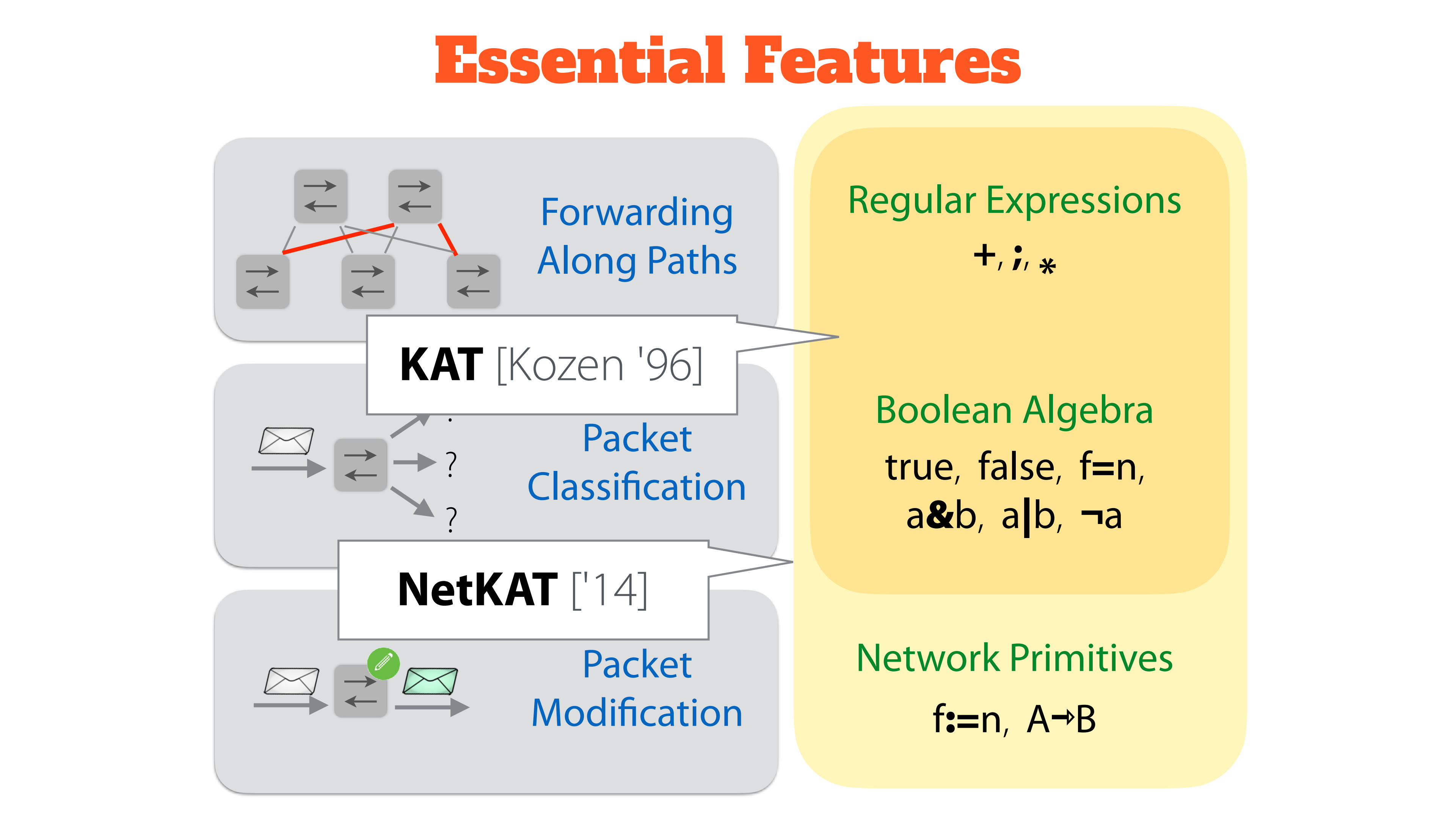
Boolean Algebra

true, false, f=n,

a&b, a|b, ¬a

Network Primitives

f:=n, A→B

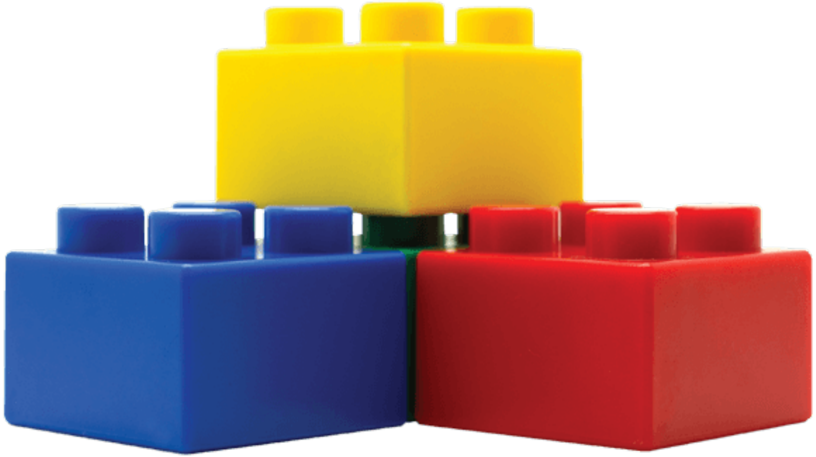

Essential Features
Forwarding Along Paths
KAT [Kozen '96]
?
?
Packet Classification
NetKAT ['14]
Packet Modification
Regular Expressions
+, ;, *
Boolean Algebra
true, false, f=n,
a&b, a|b, ¬a
Network Primitives
f:=n, A→B

Example

$$port = 88;\ switch = 6;$$
$$dest := 10.0.0.1;$$
$$(port := 50 + port := 51)$$

"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

Firewall ; Forward

"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

Monitor **+** Forward

"Execute both Monitor and Forward on all incoming packets"

Multicast: port:=1 + port:=2

Language Design & Modeling

Reasoning & Verification

Programming & Compilation

Language Design & Modeling

Reasoning & Verification

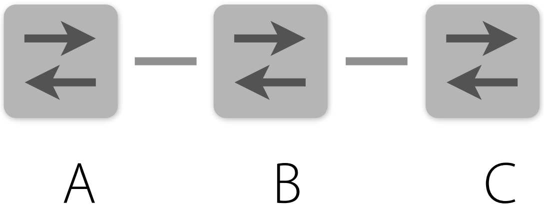
Programming & Compilation

Encoding Networks

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

Pattern	Actions
dstport=22	Drop
srcip=10.0.0.1	Forward 1
*	Forward 2

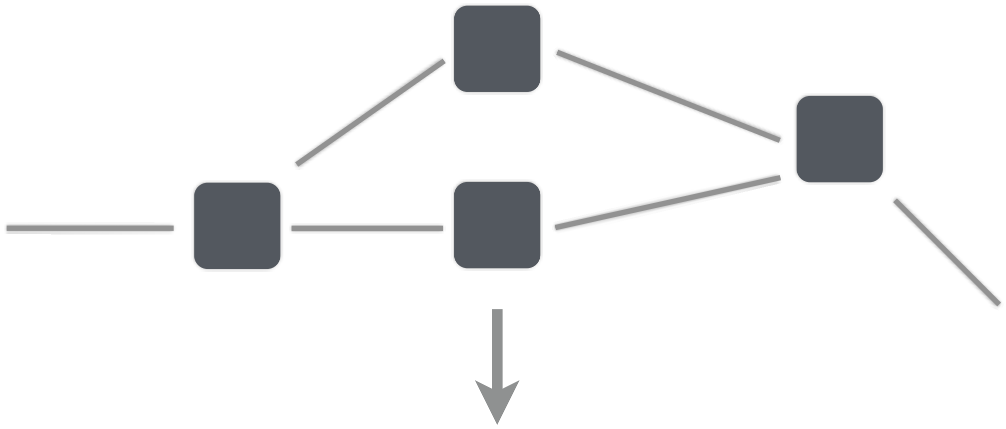
if dstport=22 **then false**
elsif srcip=10.0.0.1 **then** port := 1
else port := 2



A→B + B→A + B→C + C→B

Encoding Networks

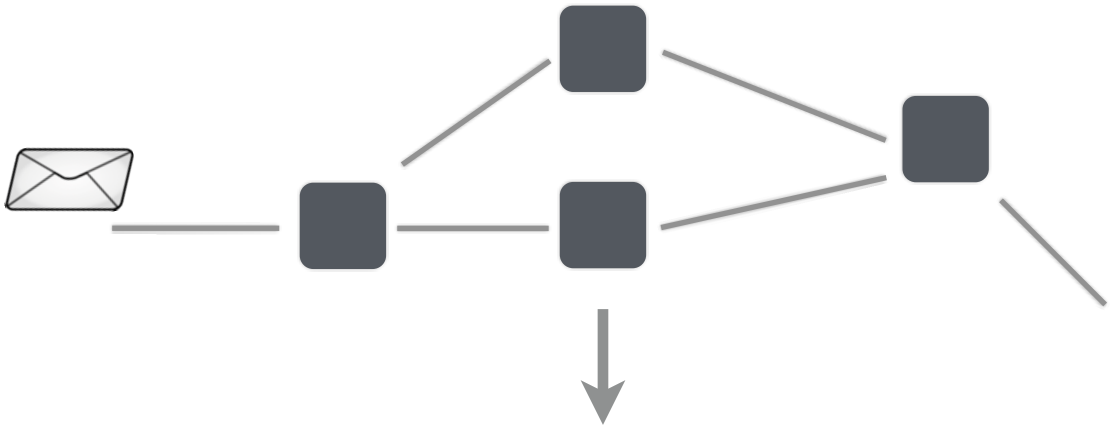
A network can be encoded in NetKAT by interleaving steps of processing by switches and topology



(topology; switch)*

Encoding Networks

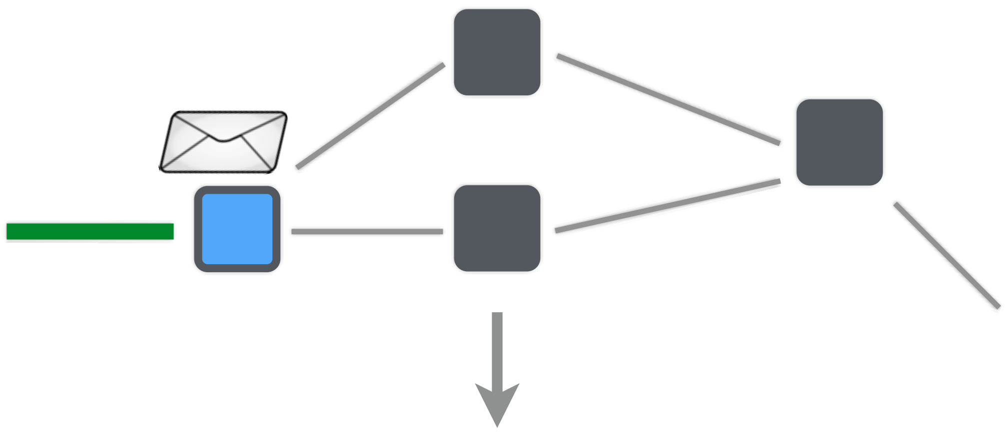
A network can be encoded in NetKAT by interleaving steps of processing by switches and topology



(topology; switch)*

Encoding Networks

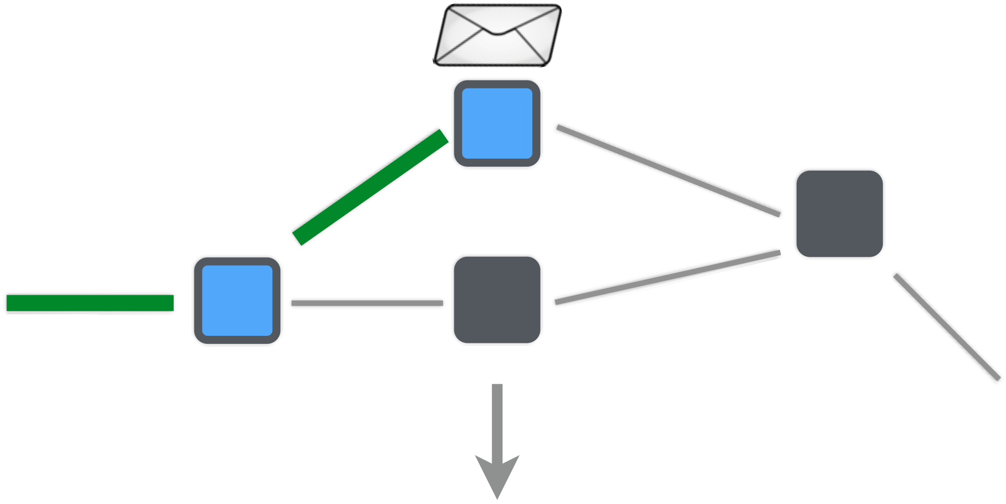
A network can be encoded in NetKAT by interleaving steps of processing by switches and topology



(topology; switch)*

Encoding Networks

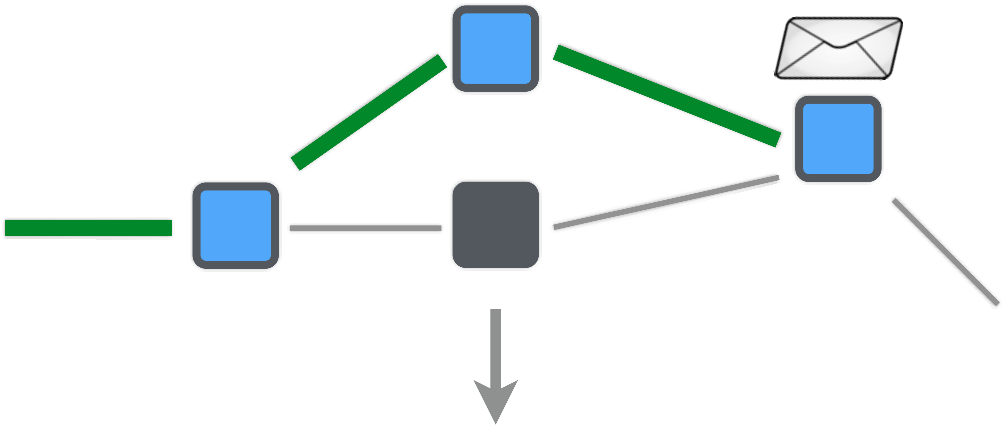
A network can be encoded in NetKAT by interleaving steps of processing by switches and topology



(topology; switch)*

Encoding Networks

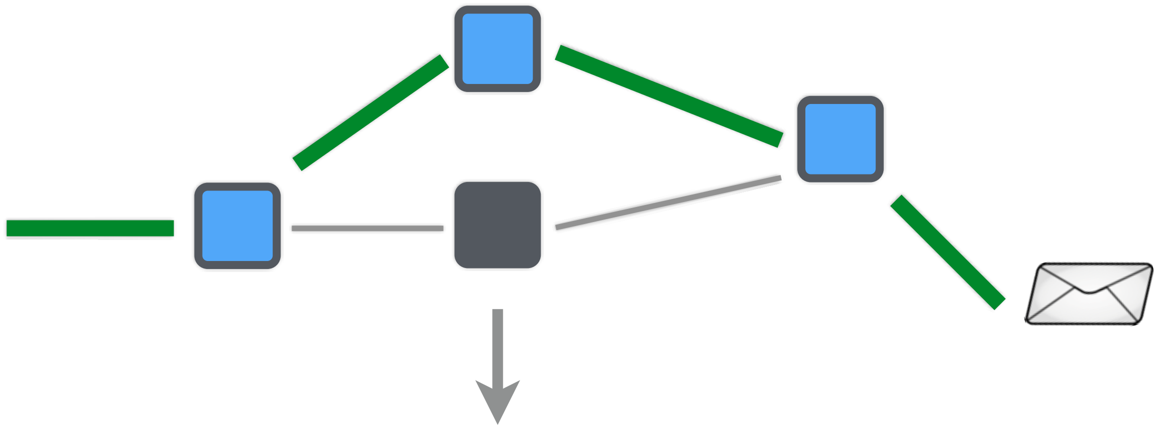
A network can be encoded in NetKAT by interleaving steps of processing by switches and topology



(topology; switch)*

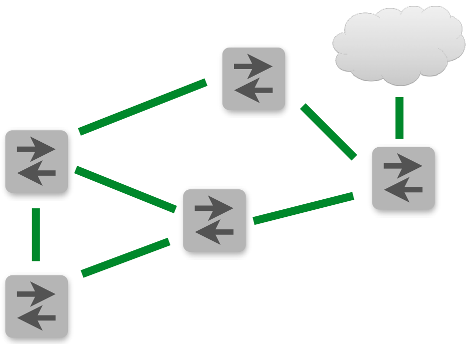
Encoding Networks

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



(topology; switch)*

Checking Reachability



$\models \phi$

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

in; (topology; switch)*; out ≢ false

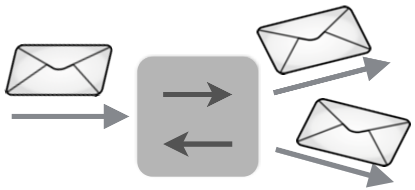
Denotational Semantics

pol ::=
  | **false**
  | **true**
  | field **=** val
  | field **:=** val
  | pol_1 **+** pol_2
  | pol_1 **;** pol_2
  | ¬pol
  | pol*
  | B → A

Denotational Semantics

pol ::=
| **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| ¬pol
| pol*
| ~~B → A~~

Local: input-output behavior of switches

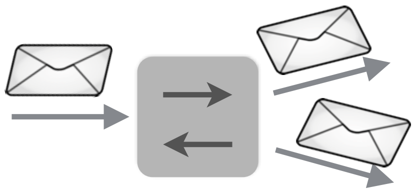


$[\![\Phi(pol)]\!] \in$ Packets → Packets

Denotational Semantics

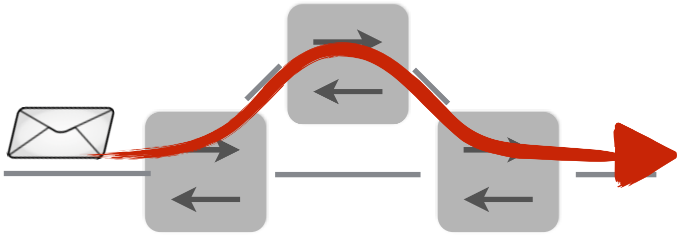
pol ::=
| **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| ¬pol
| pol*
| B → A

Local: input-output behavior of switches



$[\![\Phi(pol)]\!] \in$ Packets → Packets

Global: network-wide paths



$[\![pol]\!] \in$ Histories → Histories

NetKAT Axioms

Kleene Algebra Axioms

$p + (q + r) \equiv (p + q) + r$

$p + q \equiv q + p$

$p + \textbf{false} \equiv p$

$p + p \equiv p$

$p ; (q ; r) \equiv (p ; q) ; r$

$p ; (q + r) \equiv p \cdot q + p ; r$

$(p + q) ; r \equiv p ; r + q ; r$

$\textbf{true} ; p \equiv p$

$p \equiv p ; \textbf{true}$

$\textbf{false} ; p \equiv \textbf{false}$

$p ; \textbf{false} \equiv \textbf{false}$

$\textbf{true} + p ; p^* \equiv p^*$

$\textbf{true} + p^* ; p \equiv p^*$

$p + q ; r + r \equiv r \Rightarrow p^* ; q + r \equiv r$

$p + q ; r + q \equiv q \Rightarrow p ; r^* + q \equiv q$

Boolean Algebra Axioms

$a + (b ; c) \equiv (a + b) ; (a + c)$

$a + \textbf{true} \equiv \textbf{true}$

$a + \neg a \equiv \textbf{true}$

$a ; b \equiv b ; a$

$a ; \neg a \equiv \textbf{false}$

$a ; a \equiv a$

Packet Axioms

$f := n ; f' := n' \equiv f' := n' ; f := n \quad \text{if } f \neq f'$

$f := n ; f' = n' \equiv f' = n' ; f := n \quad \text{if } f \neq f'$

$f := n ; f = n \equiv f := n$

$f = n ; f := n \equiv f = n$

$f := n ; f := n' \equiv f := n'$

$f = n ; f = n' \equiv \textbf{false} \quad \text{if } n \neq n'$

$\textbf{A} \to \textbf{B} ; f = n \equiv f = n ; \textbf{A} \to \textbf{B} \quad \text{if } f \notin \{\text{switch, port}\}$

$\Sigma_i f = n_i \equiv \textbf{true}$

NetKAT Axioms

Kleene Algebra Axioms

p + (q + r) ≡ (p + q) + r

p + q ≡ q + p

p + **false** ≡ p

p + p ≡ p

p ; (q ; r) ≡ …

p ; (q + r) ≡ …

(p + q) ; r ≡ …

true ; p ≡ p …

p ≡ p ; **true** …

false ; p ≡ **false**

p ; **false** ≡ **false**

true + p ; p* ≡ p*

true + p* ; p ≡ p*

p + q ; r + r ≡ r ⇒ p* ; q + r ≡ r

p + q ; r + q ≡ q ⇒ p ; r* + q ≡ q

Boolean Algebra Axioms

a + (b ; c) ≡ (a + b) ; (a + c)

a + **true** ≡ **true**

a + ¬a ≡ **true**

a ; b ≡ b ; a …

Soundness: If ⊢ p ≡ q, then ⟦p⟧ = ⟦q⟧

Completeness: If ⟦p⟧ = ⟦q⟧, then ⊢ p ≡ q

… ≠ f′

f := n ; f′ = n′ ≡ f′ = n′ ; f := n   if f ≠ f′

f := n ; f = n ≡ f := n

f = n ; f := n ≡ f = n

f := n ; f := n′ ≡ f := n′

f = n ; f = n′ ≡ **false**   if n ≠ n′

A → B ; f = n ≡ f = n ; **A → B**   if f ∉ {switch, port}

Σ_i f = n_i ≡ **true**

Decision Procedure

Decides program equivalence fully automatically!

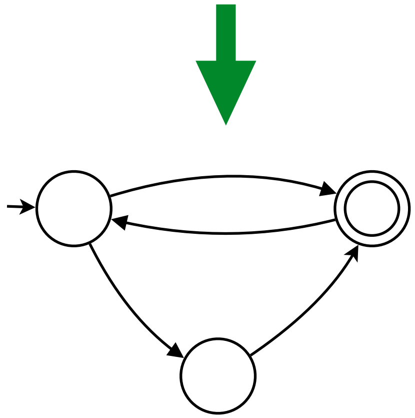
Theoretical Insight: NetKAT programs ↔ NetKAT automata

Decision Procedure

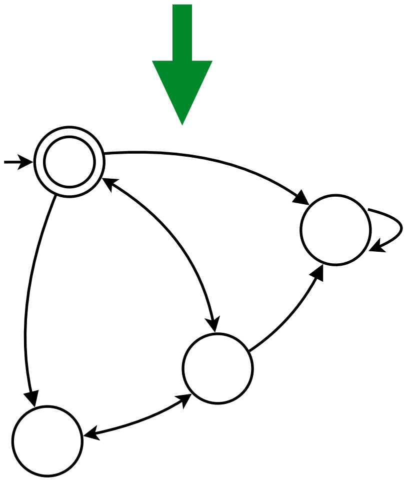
Decides program equivalence fully automatically!

Theoretical Insight: NetKAT programs ↔ NetKAT automata

p



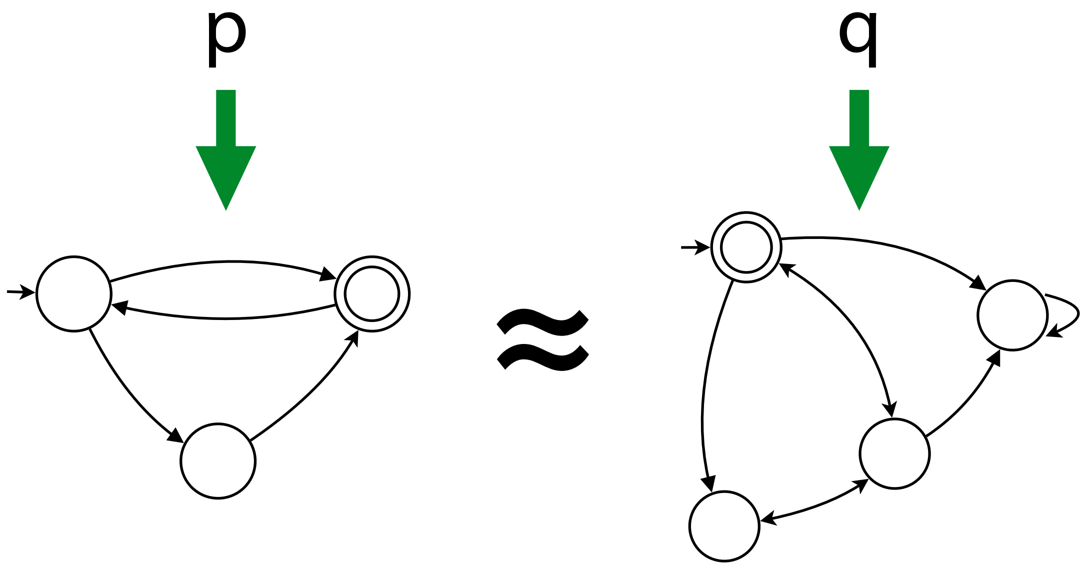
q



Decision Procedure

Decides program equivalence fully automatically!

Theoretical Insight: NetKAT programs ↔ NetKAT automata



Algorithm checks bisimilarity of automata

Language Design & Modeling

Reasoning & Verification

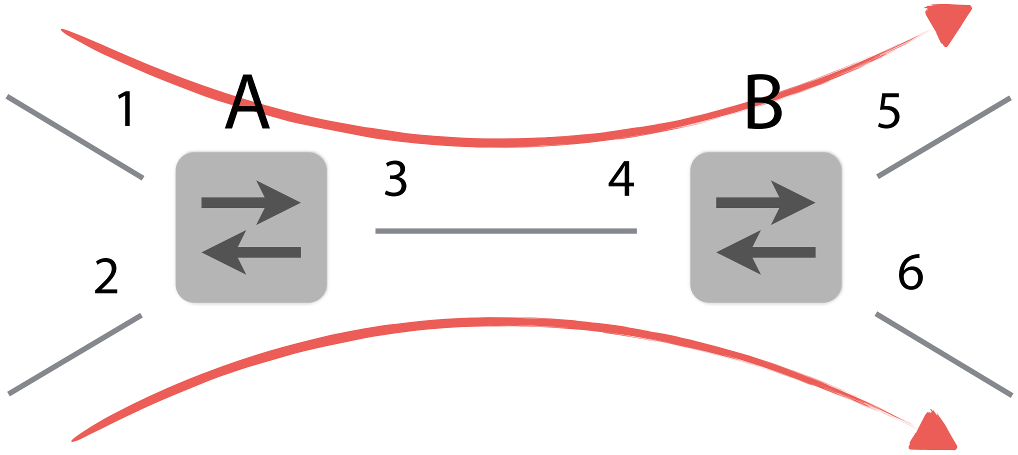
Programming & Compilation

Language Design & Modeling ✓

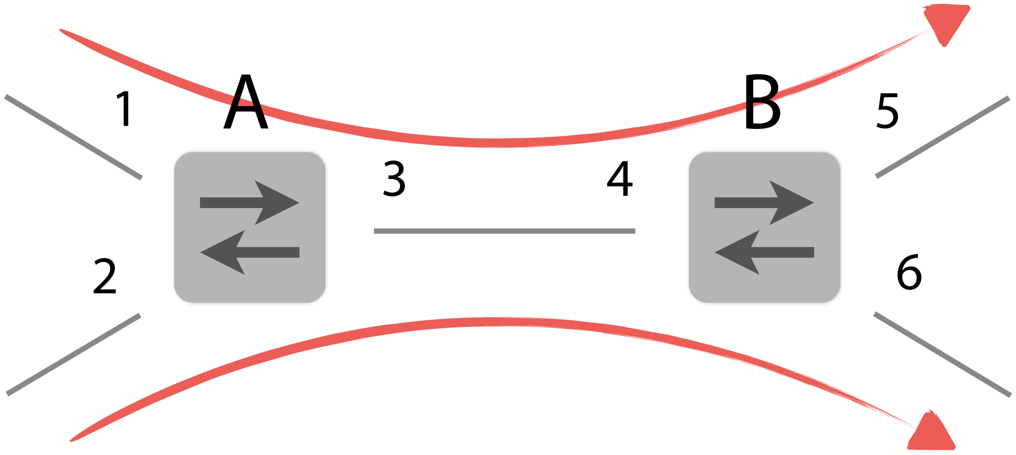
Reasoning & Verification ✓

Programming & Compilation

Example



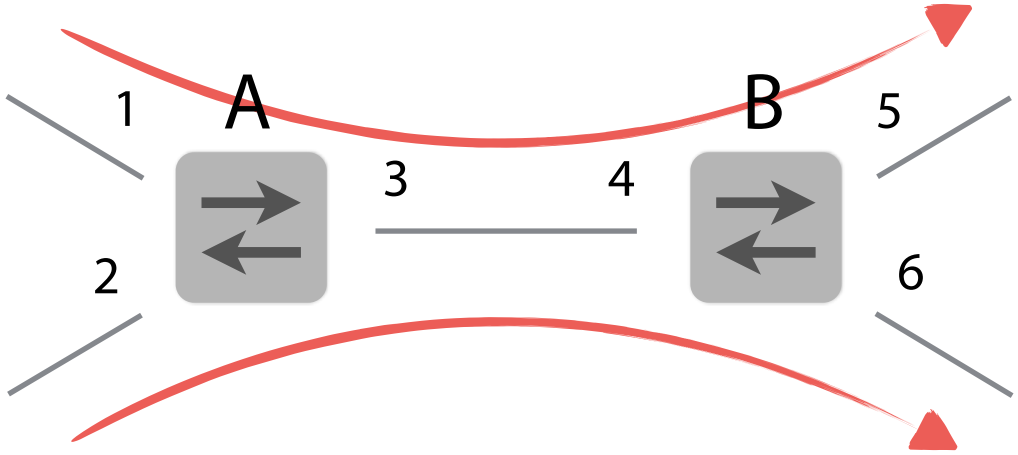
Local Program



pol_A

pol_B

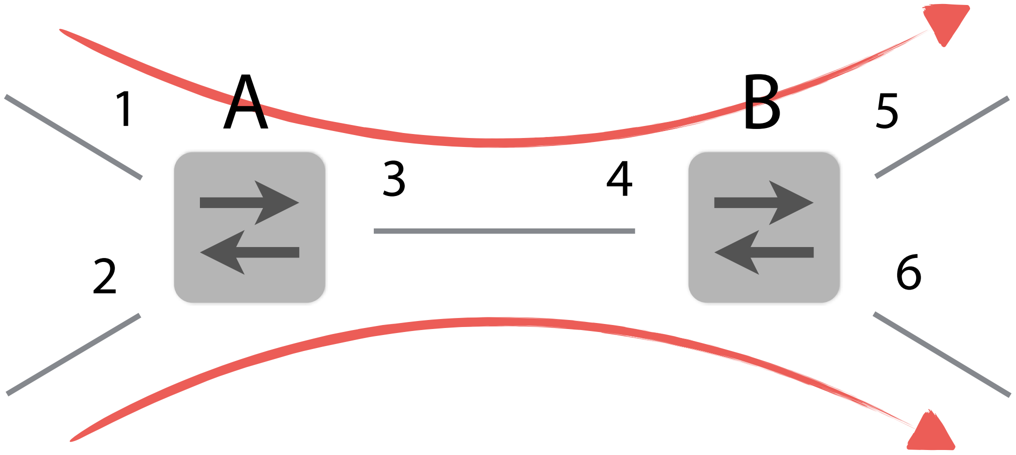
Local Program



port:=3

???

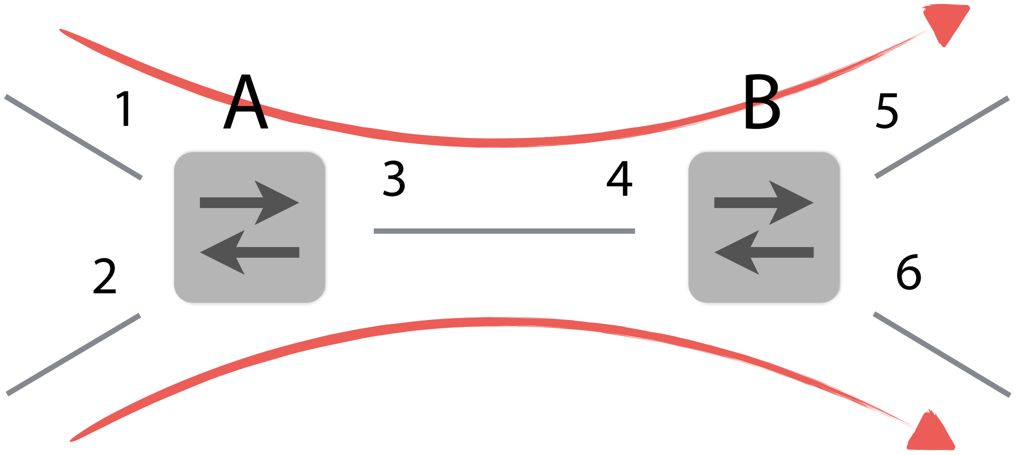
Local Program



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

???

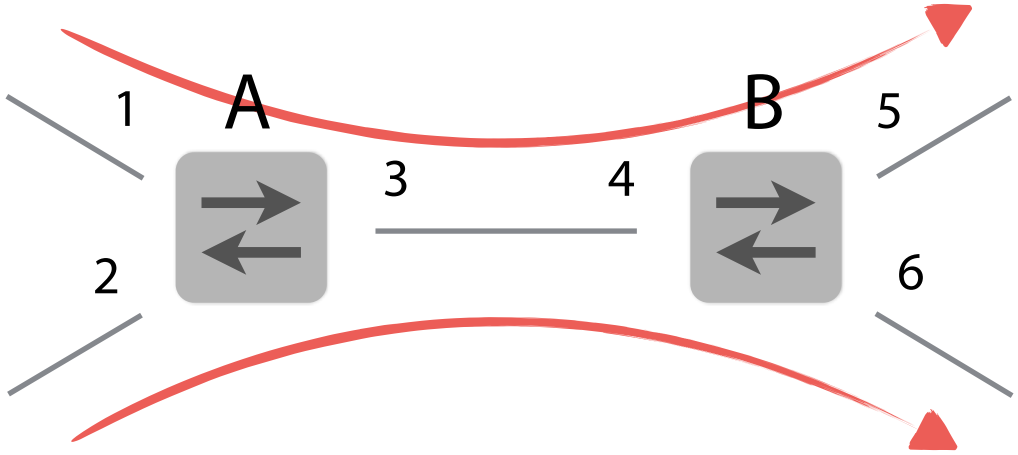
Local Program



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

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

Local Program

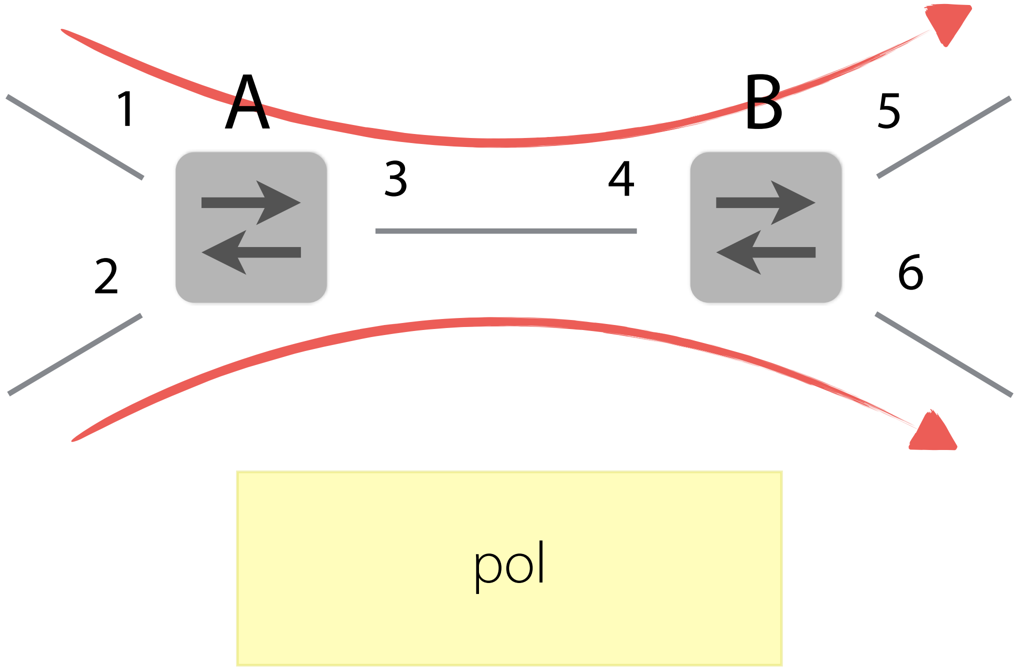


port=1; tag:=1; port:=3
+
port=2; tag:=2; port:=3

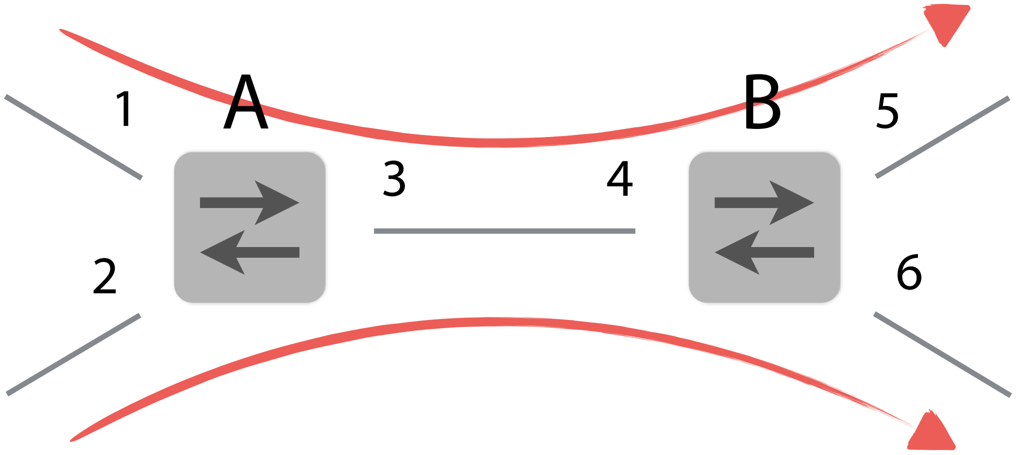
tag=1; port:=5
+
tag=2; port:=6

Tedious for programmers… difficult to get right!

Global Program

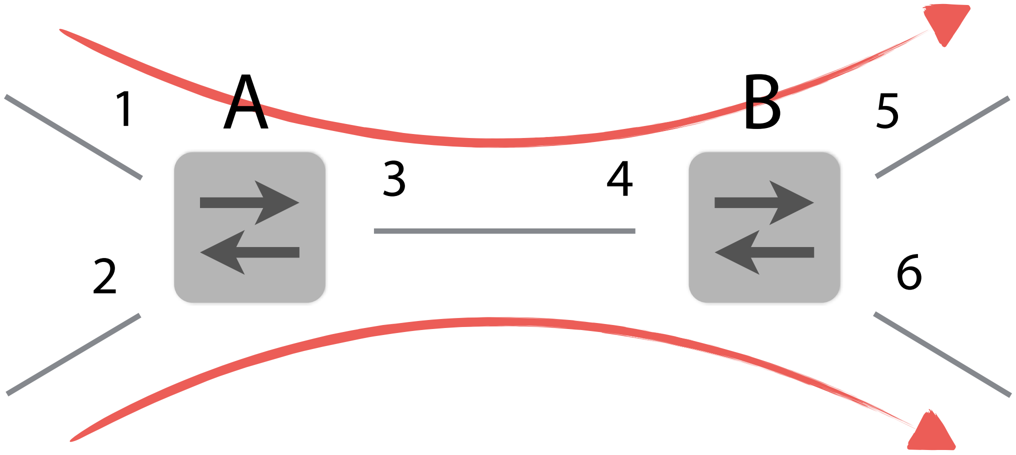


Global Program



port=1**; A→B;** port**:=**5
\+
port=2**; A→B;** port**:=**6

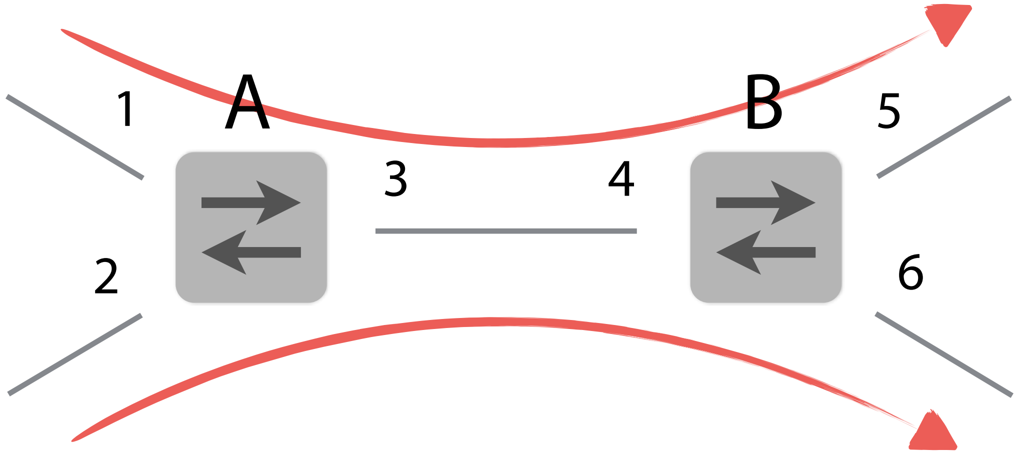
Global Program



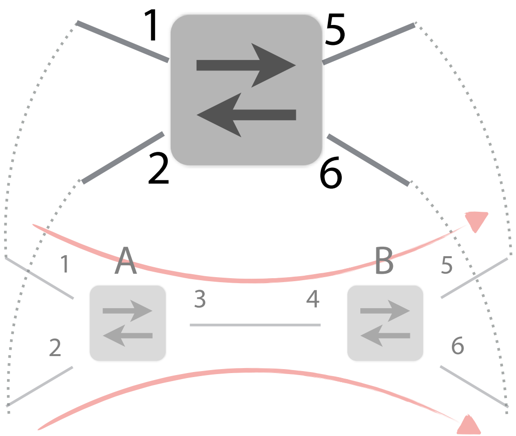
port=1**; A→B;** port**:=**5
\+
port=2**; A→B;** port**:=**6

Simple and elegant!

Virtual Program

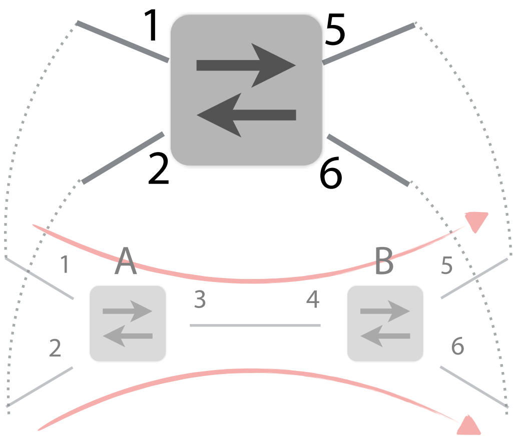


Virtual Program



virtual "big switch"

Virtual Program

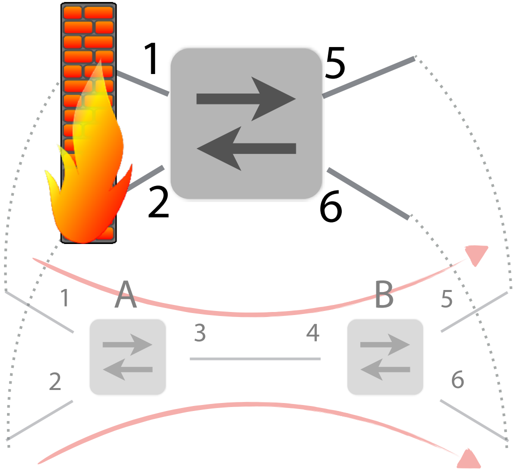


virtual "big switch"

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

Even simpler!

Virtual Program

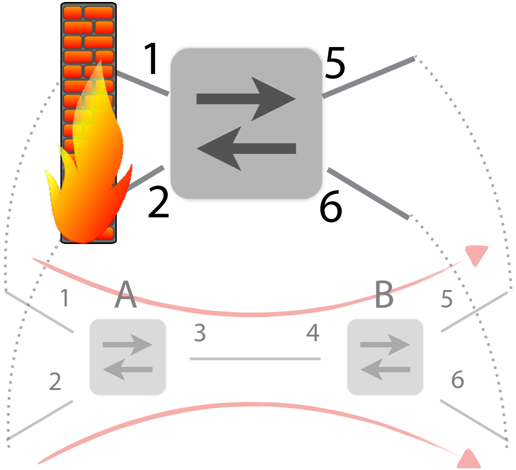


virtual "big switch"

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

Even simpler!

Virtual Program

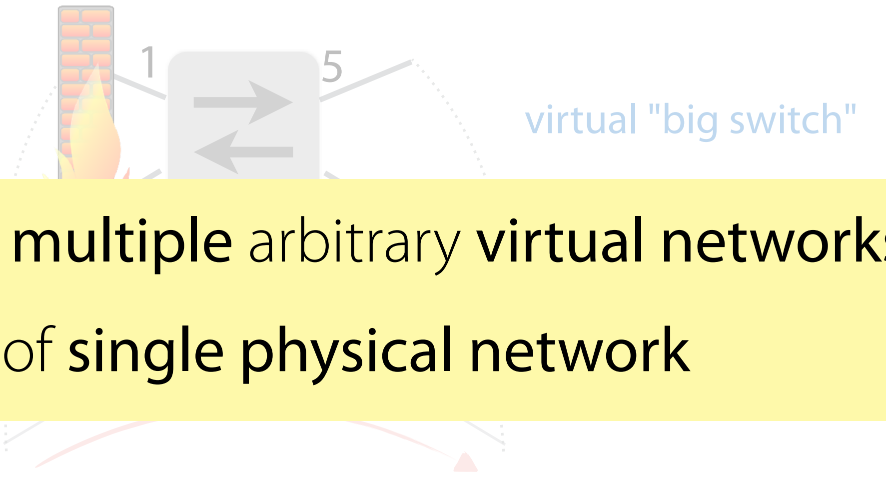


virtual "big switch"

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

Even simpler!

Virtual Program



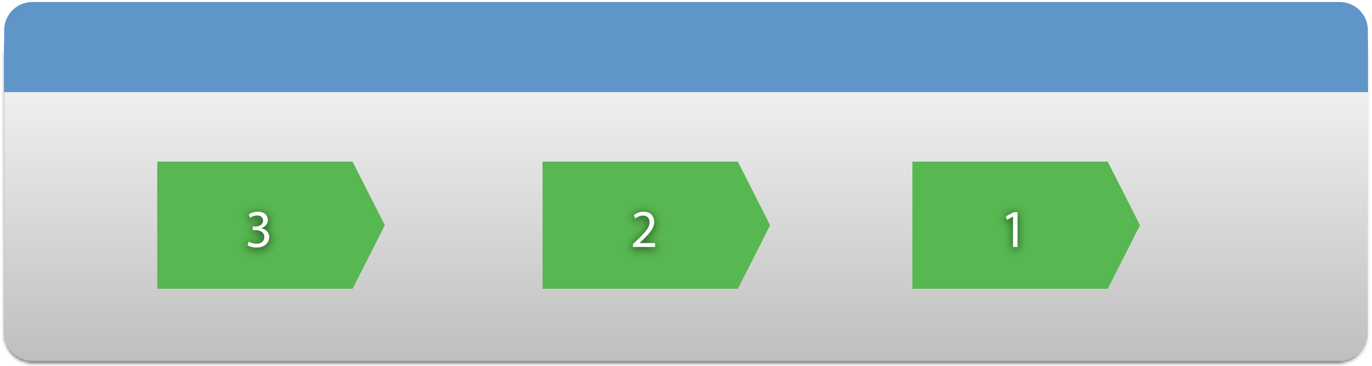
Can implement multiple arbitrary virtual networks on top of single physical network

Can implement **multiple** arbitrary **virtual networks** on top of **single physical network**

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

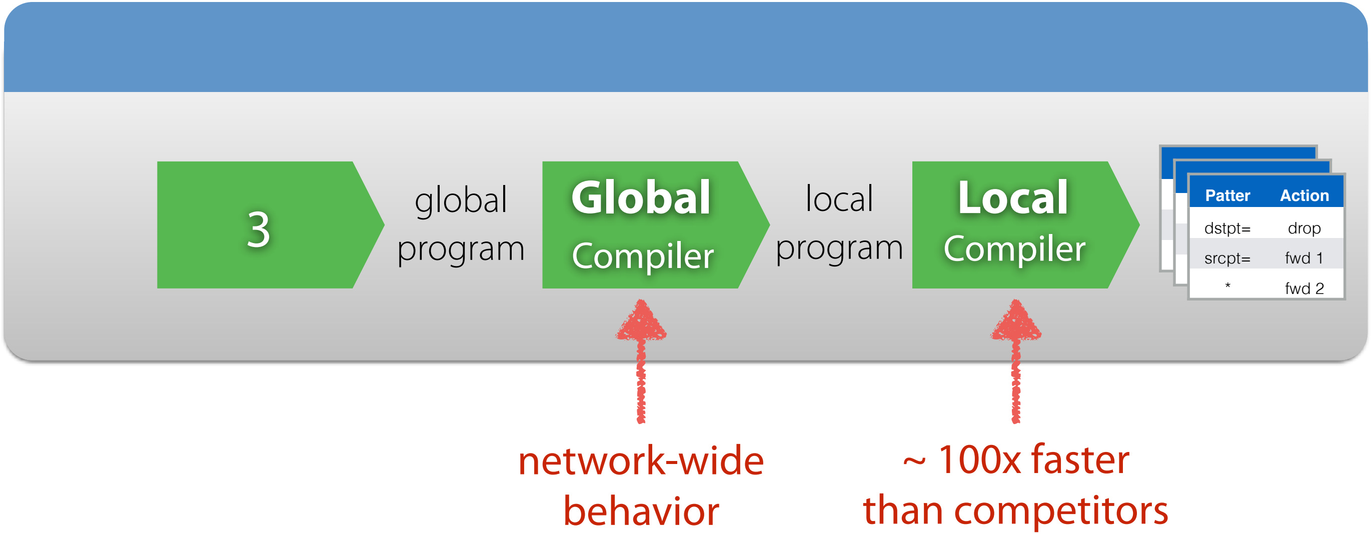
Even simpler!

Compilation [ICFP '15]

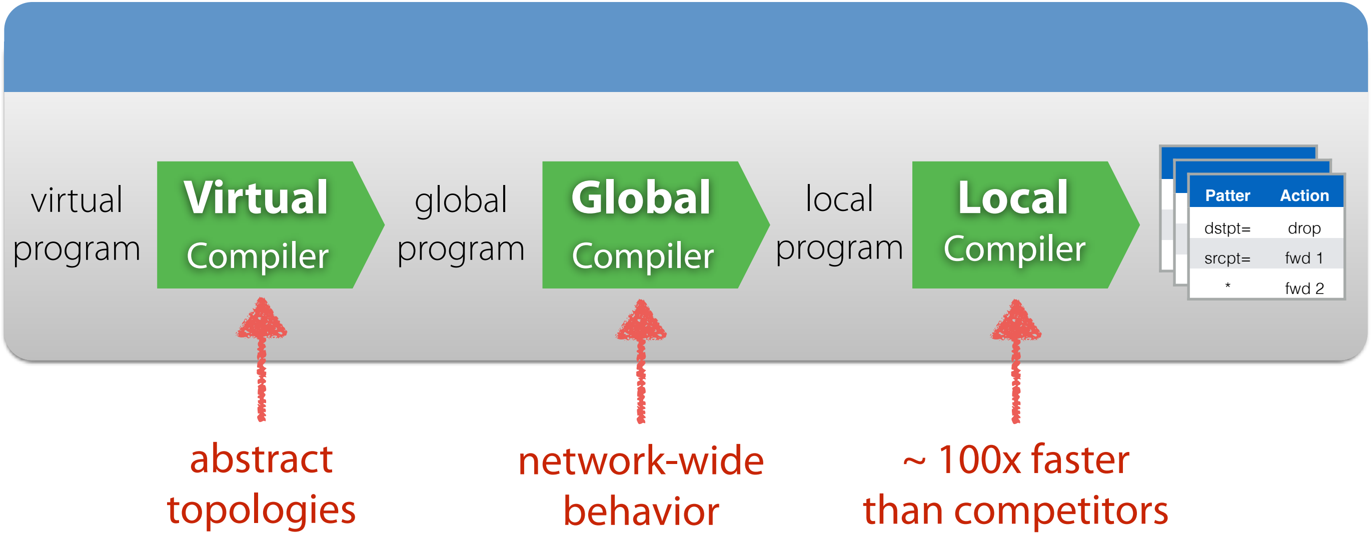


Compilation [ICFP '15]

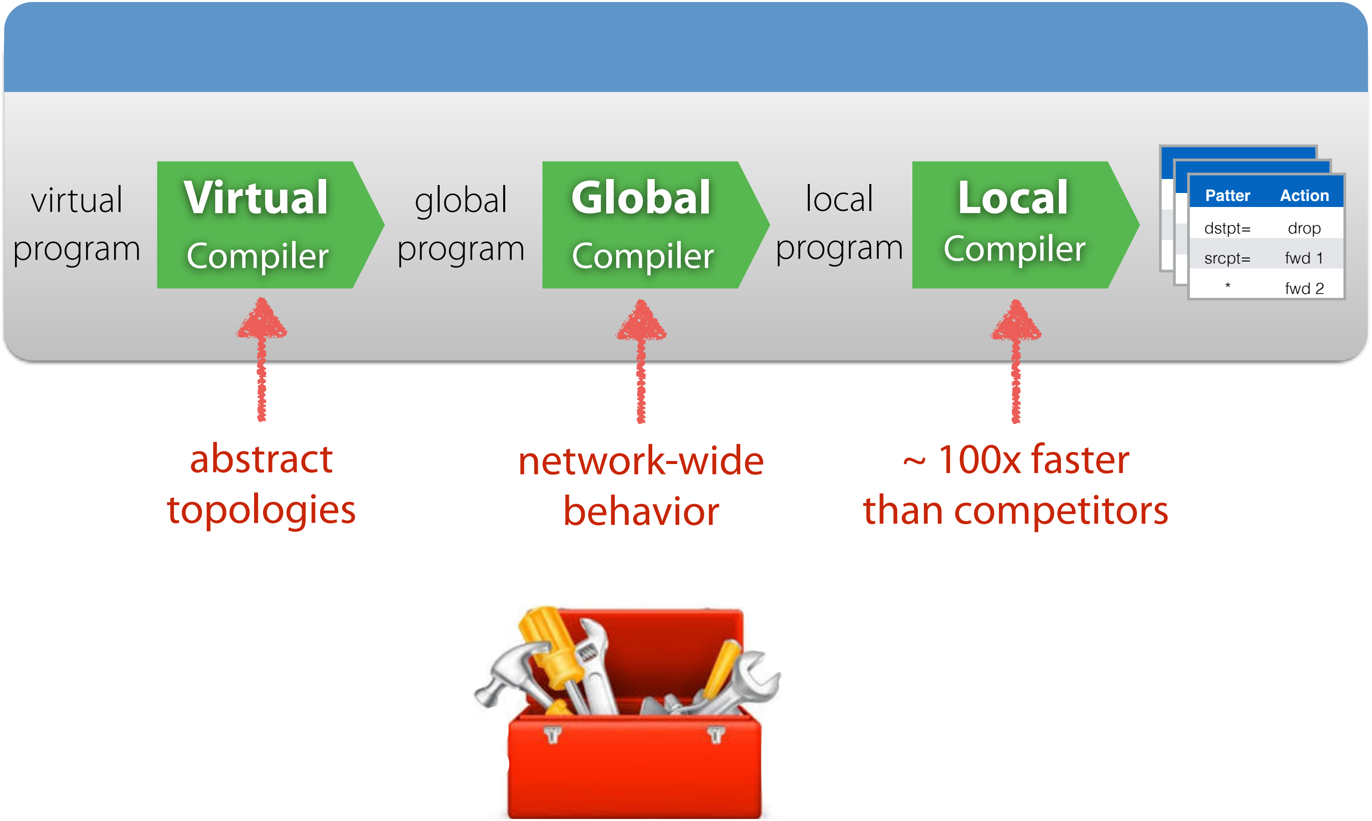
Compilation [ICFP '15]



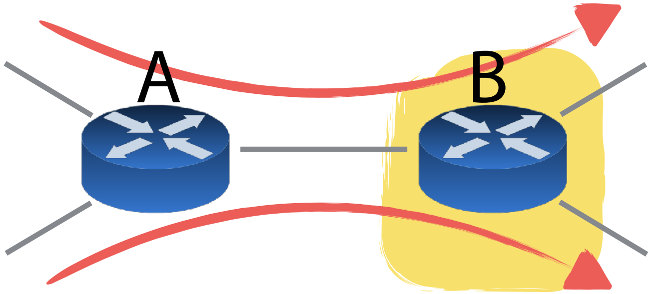
Compilation [ICFP '15]



Compilation [ICFP '15]

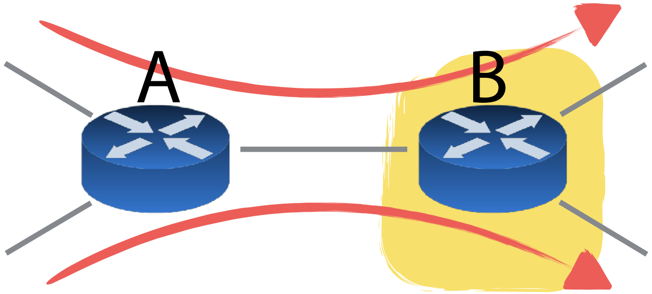


Global Compilation



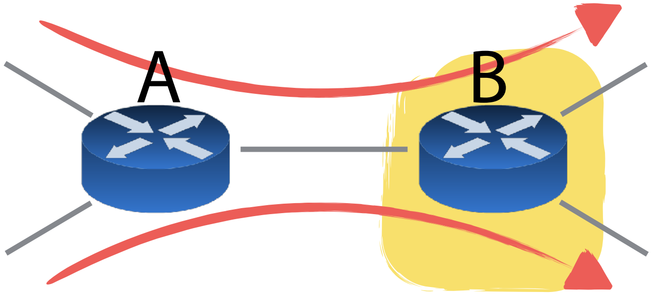
Global Compilation

1. Adding Extra State
"Tagging"

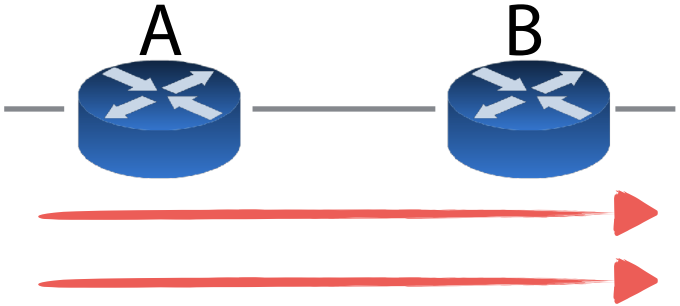


Global Compilation

1. Adding Extra State
"Tagging"

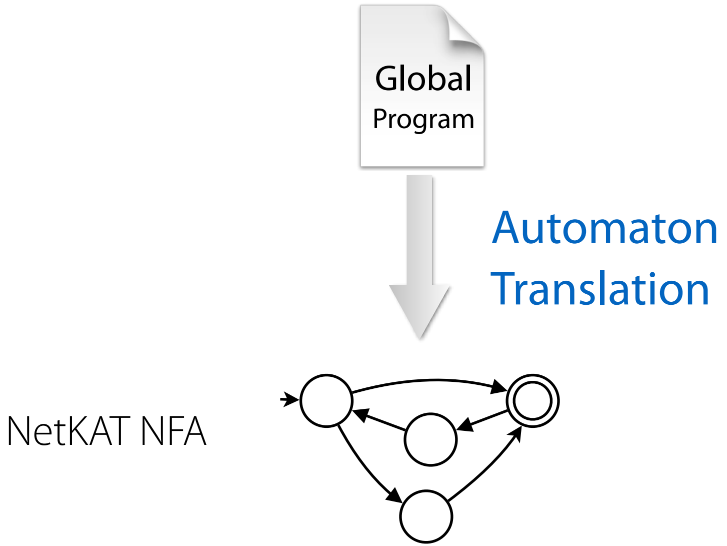


2. Avoiding Duplication
(naive tagging is unsound!)

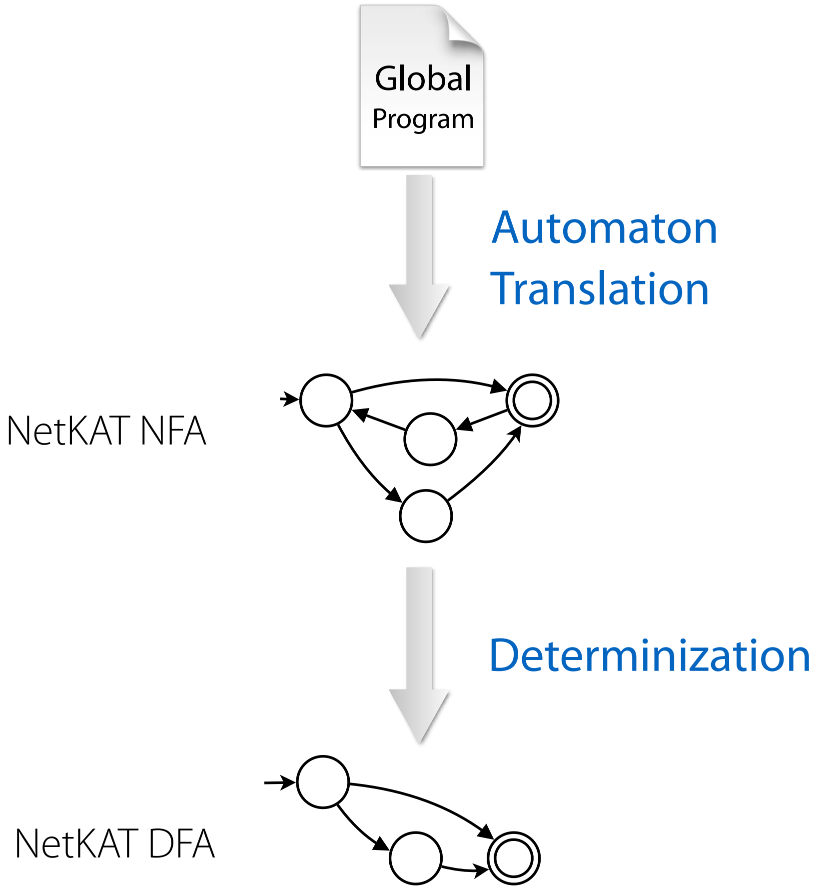


Global Compilation

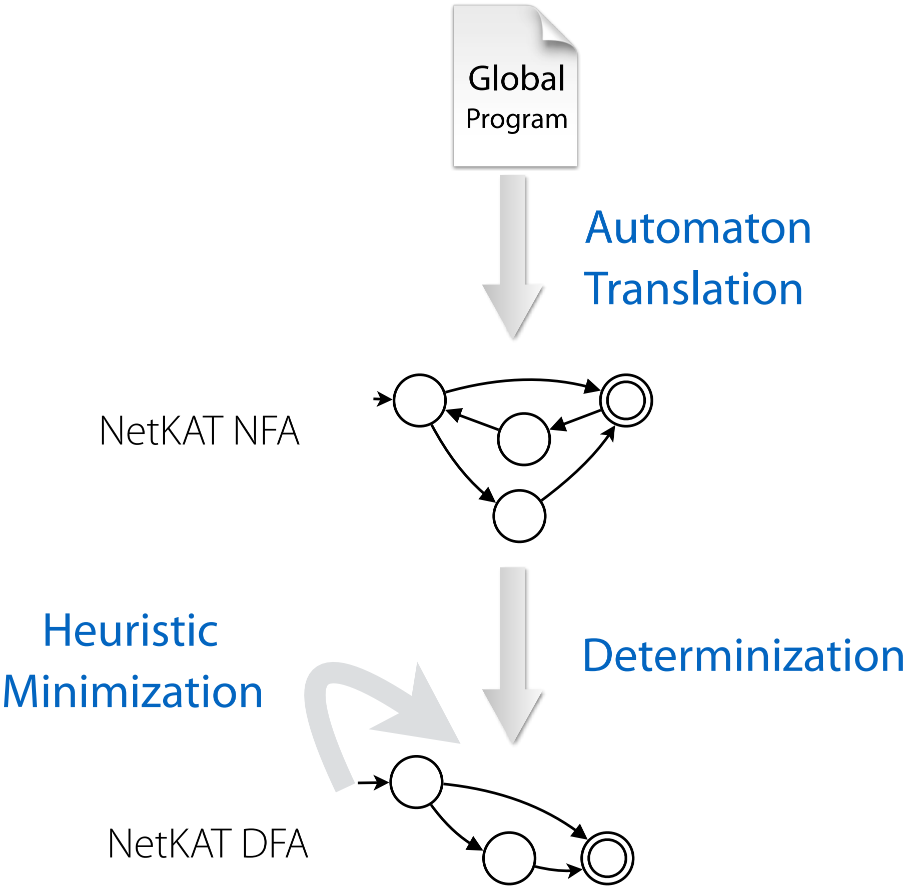
Global Compilation



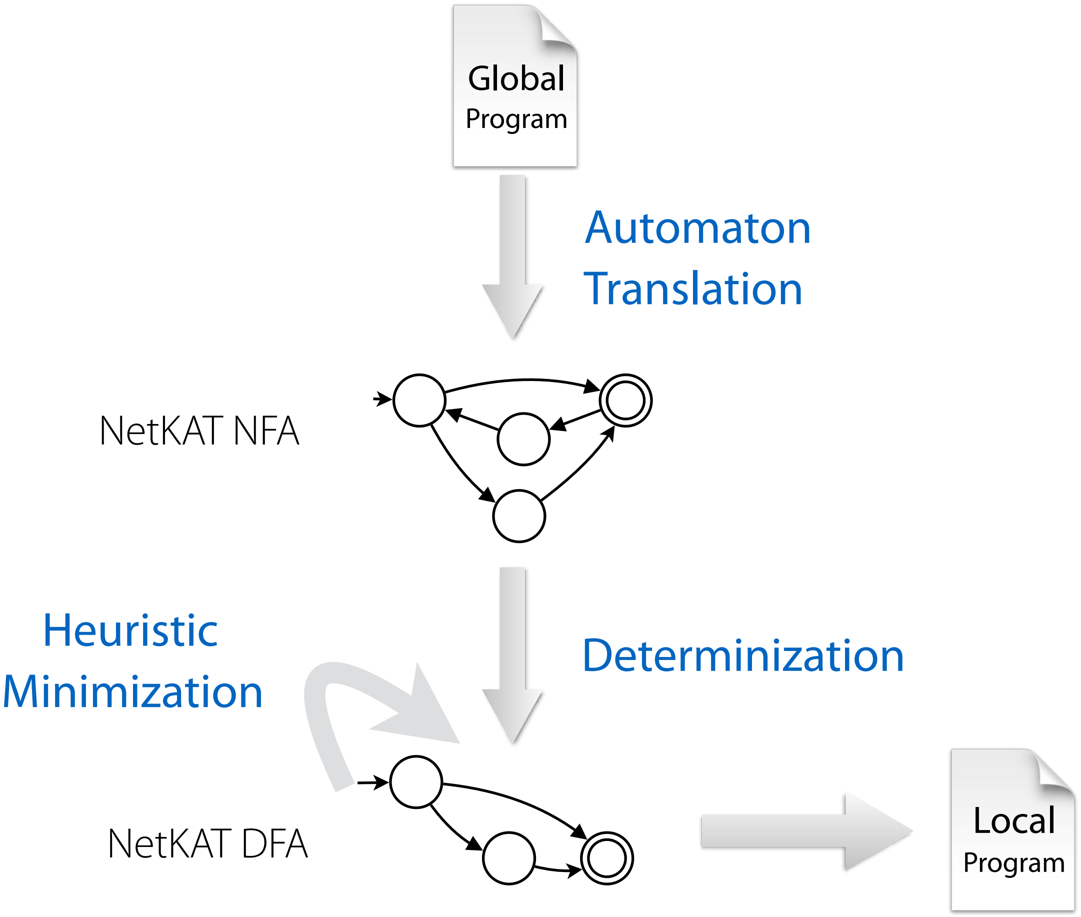
Global Compilation



Global Compilation

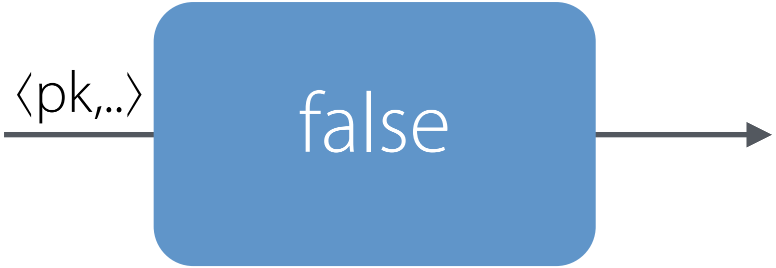


Global Compilation



Questions?

p ::= **false**
  | **true**
  | f **=** n
  | f **:=** n
  | p_1 **+** p_2
  | p_1 **;** p_2
  | ¬p
  | p*
  | A→B

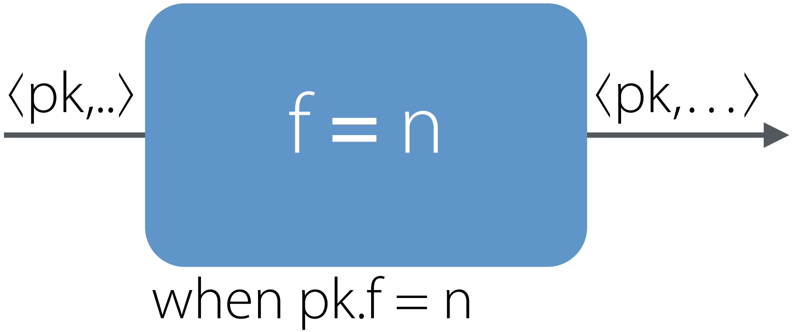


false drops its input

p ::= **false**
| **true**
| $f = n$
| $f := n$
| $p_1 + p_2$
| $p_1 ; p_2$
| $\neg p$
| p^*
| $A \rightarrow B$

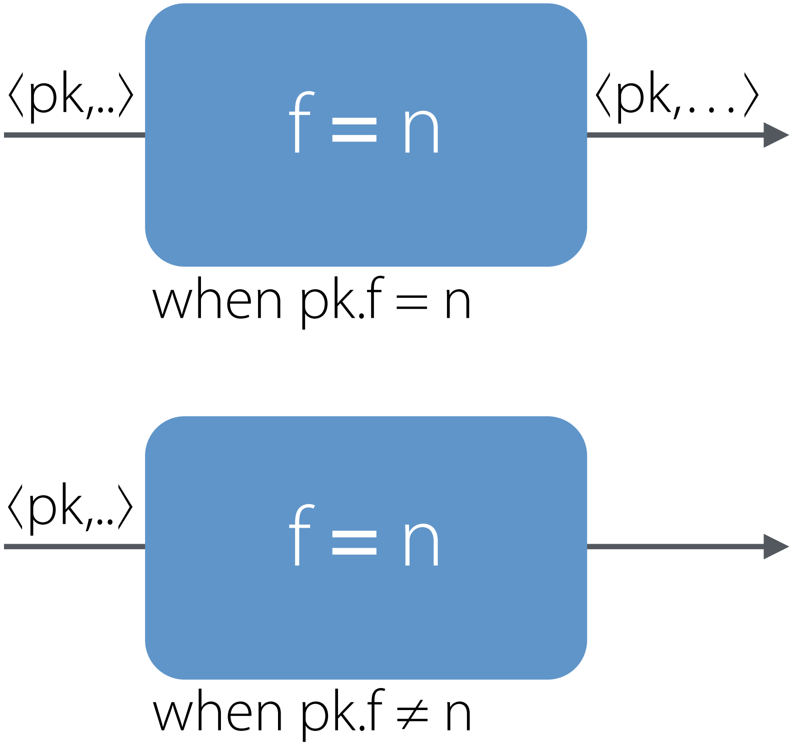
true copies its input

$$
\begin{aligned}
p ::= \;& \textbf{false} \\
\mid\; & \textbf{true} \\
\mid\; & f = n \\
\mid\; & f := n \\
\mid\; & p_1 + p_2 \\
\mid\; & p_1 \,;\, p_2 \\
\mid\; & \neg p \\
\mid\; & p^* \\
\mid\; & A \rightarrow B
\end{aligned}
$$



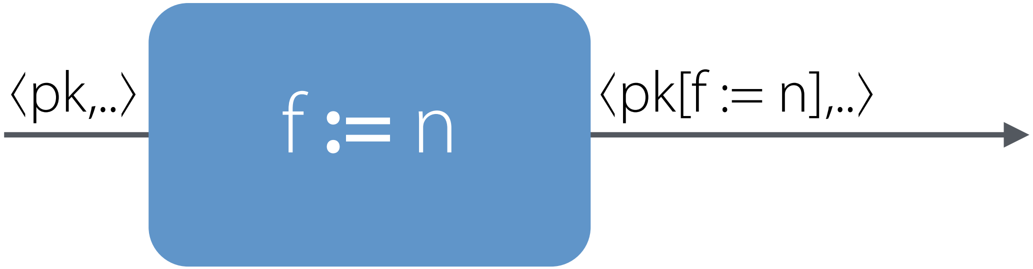
f **=** n copies its input if pk.f = n and otherwise drops it

$$
\begin{aligned}
p ::= \; & \mathbf{false} \\
| \; & \mathbf{true} \\
| \; & f \mathbf{=} n \\
| \; & f \mathbf{:=} n \\
| \; & p_1 \mathbf{+} p_2 \\
| \; & p_1 \mathbf{;} p_2 \\
| \; & \neg p \\
| \; & p^* \\
| \; & A \rightarrow B
\end{aligned}
$$



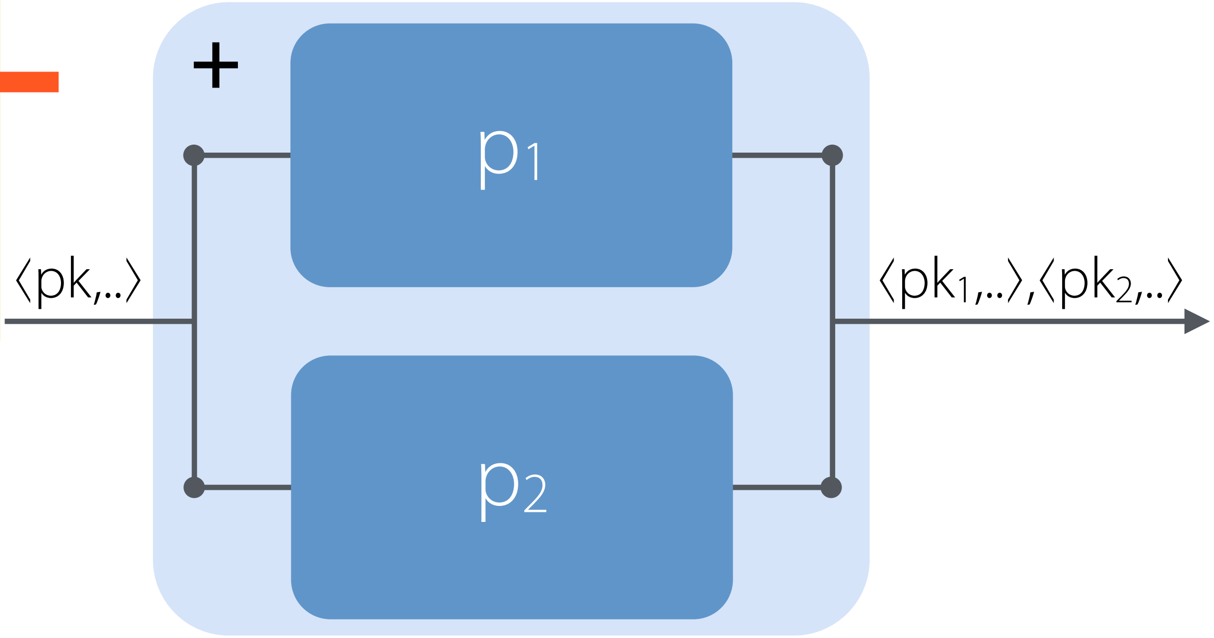
$f \mathbf{=} n$ copies its input if $pk.f = n$ and otherwise drops it

p ::= **false**
| **true**
| f **=** n
| f **:=** n
| p_1 **+** p_2
| p_1 **;** p_2
| ¬p
| p*
| A → B



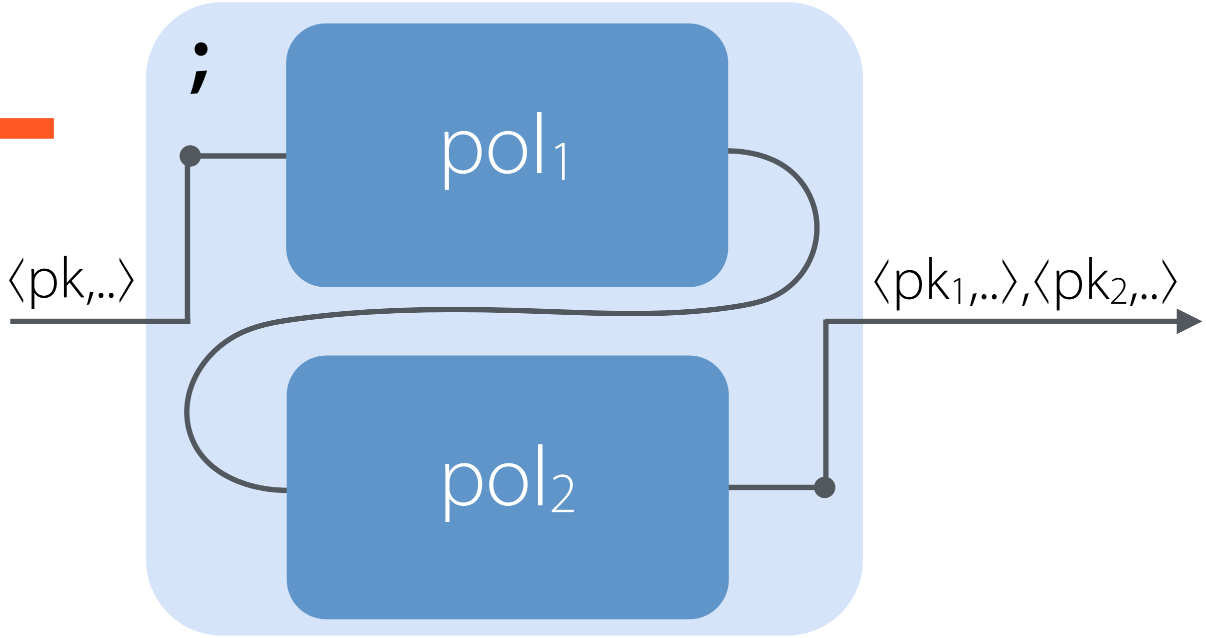
f **:=** n sets the input's f component to n

$p ::=$ **false**
| **true**
| $f = n$
| $f := n$
| $p_1 + p_2$
| $p_1 \,;\, p_2$
| $\neg p$
| p^*
| $A \rightarrow B$



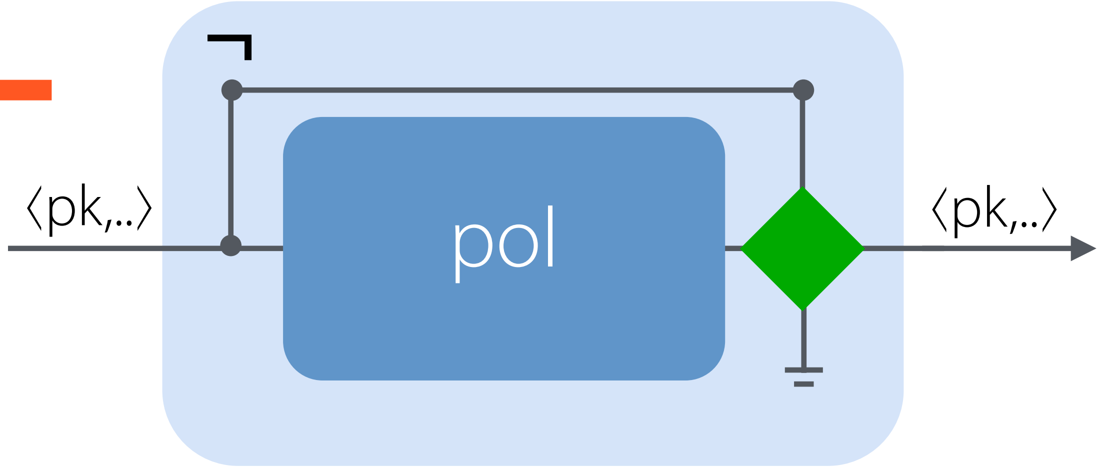
$p_1 + p_2$ duplicates the input, sends one copy to each sub-policy, and takes the *union* of their outputs

$p ::= \textbf{false}$
$\mid \textbf{true}$
$\mid f = n$
$\mid f := n$
$\mid p_1 + p_2$
$\mid p_1 \,;\, p_2$
$\mid \neg p$
$\mid p^*$
$\mid A \rightarrow B$



$p_1 \,;\, p_2$ runs the input through pol_1 and then runs every output produced by p_1 through p_2

p ::= **false**
| **true**
| f = n
| f := n
| $p_1 + p_2$
| $p_1 ; p_2$
| ¬p
| p*
| A→B



¬p drops the input if p produces any output and copies it otherwise

p ::= **false**

| **true**

| f = n

| f := n

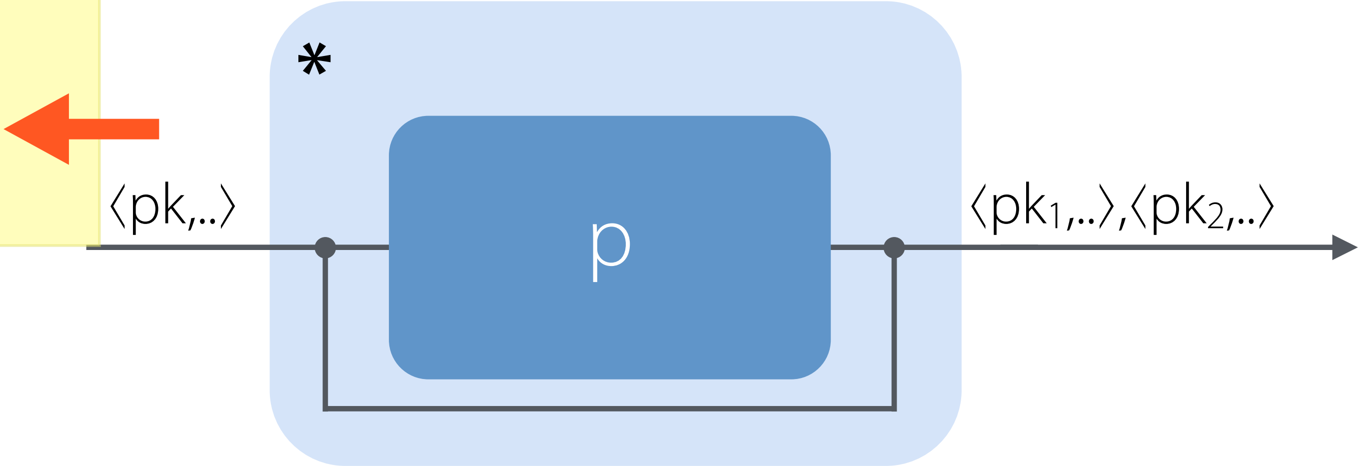
| p_1 + p_2

| p_1 ; p_2

| ¬p

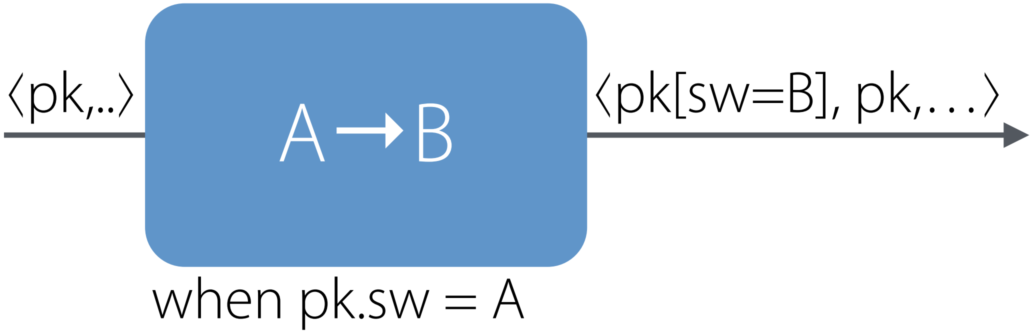
| p*

| A→B



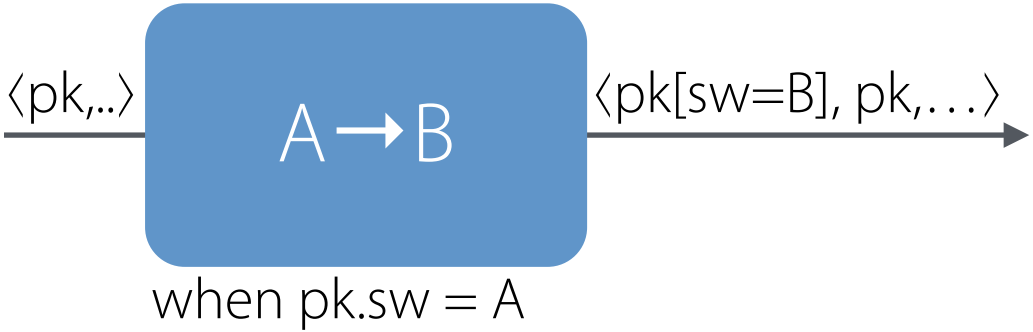
p* repeatedly runs packets through p to a fixpoint

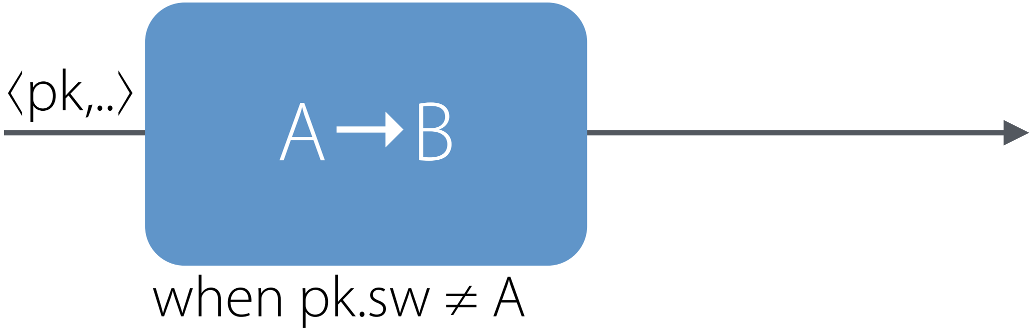
p ::= **false**
| **true**
| f **=** n
| f **:=** n
| p_1 **+** p_2
| p_1 **;** p_2
| ¬p
| p*
| A→B



A→B duplicates the packet and moves it across the link

p ::= **false**
| **true**
| f **=** n
| f **:=** n
| p_1 **+** p_2
| p_1 **;** p_2
| ¬p
| p*
| A→B





A→B duplicates the packet and moves it across the link

NetKAT Semantics

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

$$[\![f \mathbf{=} n]\!]\ pk \mathbin{\mathbf{::}} h = \begin{cases} \{\ pk \mathbin{\mathbf{::}} h\ \} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

NetKAT Semantics

$[\![p]\!] \in$ **History** $\rightarrow$ **History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

$$[\![f \mathbf{=} n]\!]\ pk :: h = \begin{cases} \{\ pk :: h\ \} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$[\![\neg p]\!]\ h = \{\ h\ \} \setminus [\![p]\!]\ h$

NetKAT Semantics

$[\![p]\!] \in$ **History** $\rightarrow$ **History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

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

$[\![\neg p]\!]\ h = \{\ h\ \} \setminus [\![p]\!]\ h$

$[\![f := n]\!]\ pk :: h = \{\ pk[f{:=}n] :: h\ \}$

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

$$[\![f \mathbin{\textbf{=}} n]\!]\ pk :: h = \begin{cases} \{\ pk :: h\ \} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$[\![\neg p]\!]\ h = \{\ h\ \} \setminus [\![p]\!]\ h$

$[\![f := n]\!]\ pk :: h = \{\ pk[f:=n] :: h\ \}$

$[\![p_1 + p_2]\!]\ h = [\![p_1]\!]\ h \cup [\![p_2]\!]\ h$

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\textbf{true}]\!]\ h = \{\ h\ \}$

$[\![\textbf{false}]\!]\ h = \{\}$

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

$[\![\neg p]\!]\ h = \{\ h\ \} \setminus [\![p]\!]\ h$

$[\![f := n]\!]\ pk :: h = \{\ pk[f{:=}n] :: h\ \}$

$[\![p_1 + p_2]\!]\ h = [\![p_1]\!]\ h \cup [\![p_2]\!]\ h$

$[\![p_1\ ;\ p_2]\!]\ h = ([\![p_1]\!] \bullet [\![p_2]\!])\ h$

f,g $\in$ **History → History Set**

$(f \bullet g)\ h = \bigcup \{\ g\ h' \mid h' \in f\ h\ \}$

NetKAT Semantics

$[\![p]\!] \in$ **History** → **History Set**

$[\![\textbf{true}]\!]\ h = \{ h \}$

$[\![\textbf{false}]\!]\ h = \{\}$

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

$[\![\neg p]\!]\ h = \{ h \} \setminus [\![p]\!]\ h$

$[\![f := n]\!]\ pk :: h = \{ pk[f:=n] :: h \}$

$[\![p_1 + p_2]\!]\ h = [\![p_1]\!]\ h \cup [\![p_2]\!]\ h$

$[\![p_1 ; p_2]\!]\ h = ([\![p_1]\!] \bullet [\![p_2]\!])\ h$

$[\![p^*]\!]\ h = (\bigcup_i [\![p]\!]^i\ h)$

$f,g \in$ **History** → **History Set**

$(f \bullet g)\ h = \bigcup \{ g\ h' \mid h' \in f\ h \}$

NetKAT Semantics

$[\![p]\!] \in$ **History → History Set**

$[\![\mathbf{true}]\!]\ h = \{ h \}$

$[\![\mathbf{false}]\!]\ h = \{\}$

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

$[\![\neg p]\!]\ h = \{ h \} \setminus [\![p]\!]\ h$

$[\![f := n]\!]\ pk :: h = \{ pk[f{:=}n] :: h \}$

$[\![p_1 + p_2]\!]\ h = [\![p_1]\!]\ h \cup [\![p_2]\!]\ h$

$[\![p_1 ; p_2]\!]\ h = ([\![p_1]\!] \bullet [\![p_2]\!])\ h$

$[\![p^*]\!]\ h = (\bigcup_i [\![p]\!]^i\ h)$

$$[\![A \rightarrow B]\!]\ pk :: h = \begin{cases} \{ pk[sw{:=}B] :: pk :: h \} & \text{if } pk.sw = A \\ \{\} & \text{otherwise} \end{cases}$$

$f,g \in$ **History → History Set**

$(f \bullet g)\ h = \bigcup \{ g\ h' \mid h' \in f\ h \}$