Programming and Reasoning with Kleene Algebra with Tests

Part 1: Fundamentals

(Nate Foster + Dexter Kozen + Alexandra Silva)*

Plan for today

- Kleene Algebra (with Tests): syntax, semantics, and automata (45min)
- Examples of program equivalence reasoning (30 min)
- Extensions: KAT+B!, NetKAT (90min)

Context

```
while a & b do
      p;
od;
while a do
      q ;
      while a & b do
            p;
      od
od
```

≡ ?

```
while a do
      if b then
            p;
      else
      q;
od
```

Historical remarks

Historical remarks

- Flowchart schemes [Ianov 1960, Manna 1972, …]
- Regular expressions = finite automata [Kleene 1956]
- Completeness of equivalence [Salomaa 1966]

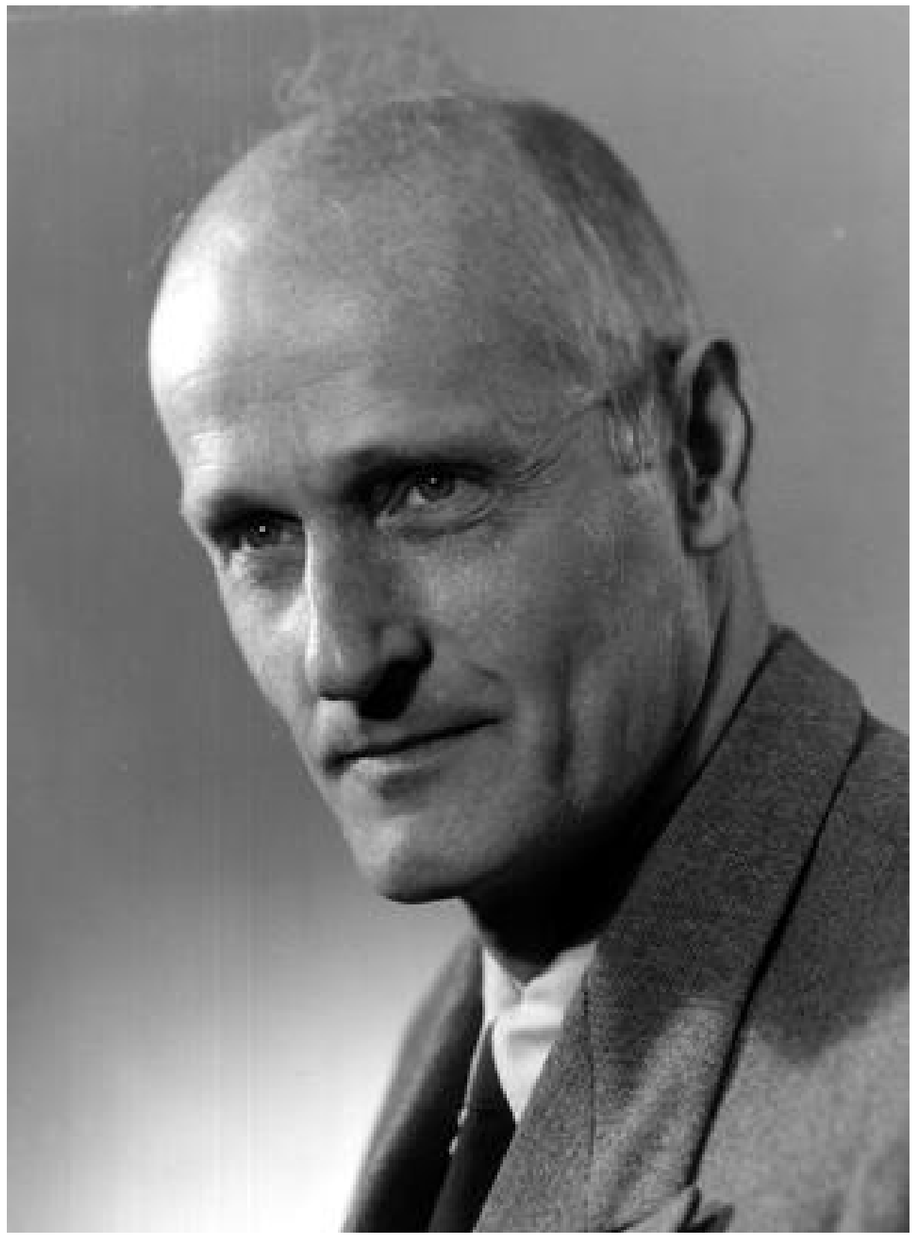
Historical remarks

- Flowchart schemes [Ianov 1960, Manna 1972, …]
- Regular expressions = finite automata [Kleene 1956]
- Completeness of equivalence [Salomaa 1966]
- Algebraic Theory [Conway 1971, Kozen 1990]
- Decision procedure PSPACE [Stockmeyer, Meyer 1974]

Historical remarks

- Flowchart schemes [Ianov 1960, Manna 1972, …]
- Regular expressions = finite automata [Kleene 1956]
- Completeness of equivalence [Salomaa 1966]
- Algebraic Theory [Conway 1971, Kozen 1990]
- Decision procedure PSPACE [Stockmeyer, Meyer 1974]
- KAT and program schematology [Angus, Kozen 2010]
- NetKAT [Anderson et al 2015]
- GKAT [Smolka et al 2020]

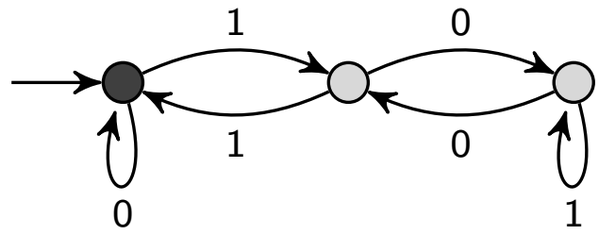
Kleene Algebra



Stephen Cole Kleene
(1909–1994)

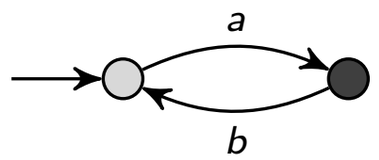
$(0 + 1(01^*0)^*1)^*$

{multiples of 3 in binary}



$(ab)^*a = a(ba)^*$

$\{a, aba, ababa, \ldots\}$



$(a + b)^* = a^*(ba^*)^*$

{all strings over $\{a, b\}$}

Kleene Algebra - syntax

```
e,f :=  0              abort
      | 1              skip
      | a              action
      | e + f          choice
      | e ; f          sequential
      | e*             iteration
```

Kleene Algebra - examples

```
(1+a);(1+b)  – a or b or ab

(a+b)*       – any sequence of a’s and b’s

(a*b)*a*     – ?
```

Kleene Algebra - examples

```
(1+a);(1+b)  – a or b or ab

(a+b)*       – any sequence of a’s and b’s

(a*b)*a*     – ?
```

$$(a+b)^* = (a^*b)^*a^*$$

Kleene Algebra - examples

```
(1+a);(1+b)  – a or b or ab

(a+b)*       – any sequence of a's and b's

(a*b)*a*     – ?
```

(a+b)* = (a*b)*a*

Standard Semantics

Regular Sets over A*

sem(0) = { }

sem(1) = {ε}

sem(a) = {a}

sem(e+f) = sem(e) $\cup$ sem(f)

sem(e;f) = sem(e) . sem(f)

sem(e*) = $\bigcup_n$ sem(e)^n

Standard Semantics

Regular Sets over A*

$$sem(0) = \{\ \}$$

$$sem(1) = \{\varepsilon\}$$

Empty word

$$sem(a) = \{a\}$$

$$sem(e+f) = sem(e) \cup sem(f)$$

$$sem(e;f) = sem(e) \,.\, sem(f)$$

$$sem(e^*) = \bigcup_n sem(e)^n$$

Standard Semantics

Regular Sets over A*

$$\begin{aligned}
\mathtt{sem}(0) &= \{\,\} \\
\mathtt{sem}(1) &= \{\varepsilon\} \\
\mathtt{sem}(a) &= \{a\} \\
\mathtt{sem}(e+f) &= \mathtt{sem}(e) \cup \mathtt{sem}(f) \\
\mathtt{sem}(e;f) &= \mathtt{sem}(e) \,.\, \mathtt{sem}(f) \\
\mathtt{sem}(e^*) &= \bigcup_n \mathtt{sem}(e)^n
\end{aligned}$$

Empty word

pointwise concatenation
$L.K = \{xy \mid x \in L, y \in K\}$

Equivalence

$$(a+b)^* \equiv (a^*b)^*a^*$$

$$sem((a+b)^*) = sem((a^*b)^*a^*)$$

Equivalence

Syntactic equivalence

$$(a+b)^* \equiv (a^*b)^*a^*$$

$$sem((a+b)^*) = sem((a^*b)^*a^*)$$

Kleene's theorem

Let L be a regular language. TFAE:

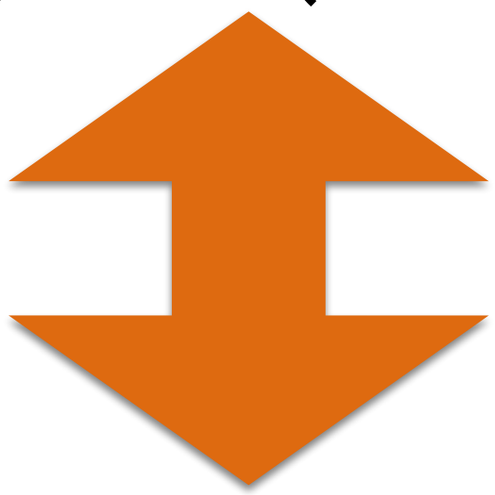
1. L = sem(e), for some regular expression e
2. L is accepted by a deterministic finite automaton

Kleene's theorem

Let L be a regular language. TFAE:

1. L = sem(e), for some regular expression e
2. L is accepted by a deterministic finite automaton

```
(a+b)* ≡ (a*b)*a*
```

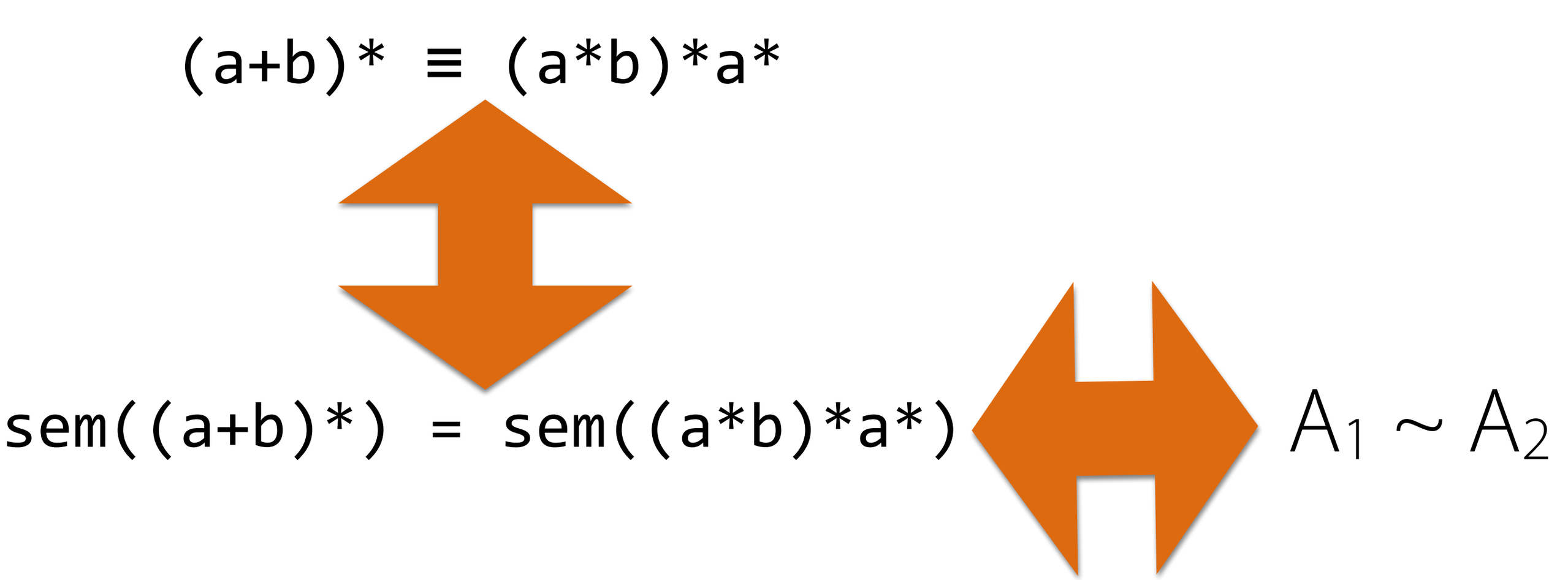


```
sem((a+b)*) = sem((a*b)*a*)
```

Kleene's theorem

Let L be a regular language. TFAE:

1. L = sem(e), for some regular expression e
2. L is accepted by a deterministic finite automaton



Fundamental Questions

➡ Can we axiomatize the valid equations?

➡ Can we decide if expressions are equivalent?

Fundamental Questions

➡ Can we axiomatize the valid equations?

➡ Can we decide if expressions are equivalent?

Program verification via Equivalence

Axioms

```
Z: e + 0 ≡ e
C: e + f ≡ f + e
A: (e + f) + g ≡ e + (f + g)
I: e + e ≡ e
```

```
SN: e;1 ≡ 1;e ≡ e
SZ: e;0 ≡ 0;e ≡ 0
SA: p;(q;r) ≡ (p;q);r
SDl: p;(q + r) = p;q + p;r
SDr:(p + q);r = p;r + q;r
```

Axioms

```
Z: e + 0 ≡ e
C: e + f ≡ f + e
A: (e + f) + g ≡ e + (f + g)
I: e + e ≡ e
```

```
SN: e;1 ≡ 1;e ≡ e
SZ: e;0 ≡ 0;e ≡ 0
SA: p;(q;r) ≡ (p;q);r
SDl: p;(q + r) = p;q + p;r
SDr:(p + q);r = p;r + q;r
```

idempotent semiring

Axioms

```
Z: e + 0 ≡ e
C: e + f ≡ f + e
A: (e + f) + g ≡ e + (f + g)
I: e + e ≡ e
```

```
SN: e;1 ≡ 1;e ≡ e
SZ: e;0 ≡ 0;e ≡ 0
SA: p;(q;r) ≡ (p;q);r
SDl: p;(q + r) = p;q + p;r
SDr:(p + q);r = p;r + q;r
```

Axioms

Z: e + 0 ≡ e

C: e + f ≡ f + e

A: (e + f) + g ≡ e + (f + g)

I: e + e ≡ e

*** is a least fix point**

FP: 1 + a;a* ≡ a*

1 + a*;a ≡ a*

LFP: f + ex ≤ x

→ e*f ≤ x

SN: e;1 ≡ 1;e ≡ e

SZ: e;0 ≡ 0;e ≡ 0

SA: p;(q;r) ≡ (p;q);r

SDl: p;(q + r) = p;q + p;r

SDr:(p + q);r = p;r + q;r

Kleene Algebra, formally

A Kleene algebra is a tuple (**K**, **+**, **;** , ***** , **0**, **1**) satisfying:

Z: e + 0 ≡ e

C: e + f ≡ f + e

A: (e + f) + g ≡ e + (f + g)

I: e + e ≡ e

SN: e;1 ≡ 1;e ≡ e

SZ: e;0 ≡ 0;e ≡ 0

SA: p;(q;r) ≡ (p;q);r

SDl: p;(q + r) = p;q + p;r

SDr:(p + q);r = p;r + q;r

FP: 1 + a;a* ≡ a*

1 + a*;a ≡ a*

LFP: f + ex ≤ x

→ e*f ≤ x

Kleene Algebra, examples

- Regular languages
    - **+ is union**
    - **; is pointwise concatenation**
    - *** is iteration**

Kleene Algebra, examples

- Regular languages
  - **+ is union**
  - **; is pointwise concatenation**
  - *** is iteration**
- Binary Relations
  - **+ is union**
  - **; is relational composition**
  - *** is reflexive transitive closure**

Kleene Algebra, examples

- Regular languages
  - **+ is union**
  - **; is pointwise concatenation**
  - *** is iteration**
- Binary Relations
  - **+ is union**
  - **; is relational composition**
  - *** is reflexive transitive closure**
- Square Matrices over a KA **K**
  - **+ and ; lifted to usual matrices ops**
  - *** iteratively from 2 x 2**

Kleene Algebra, examples

- Regular languages
  - **+ is union**
  - **; is pointwise concatenation**
  - *** is iteration**
- Binary Relations
  - **+ is union**
  - **; is relational composition**
  - *** is reflexive transitive closure**
- Square Matrices over a KA **K**

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$

s

Kleene Algebra, examples

- Regular languages
  - **+ is union**
  - **; is pointwise co…n**
  - *** is iteration**
- Binary Relations
  - **+ is union**
  - **; is relational composition**
  - *** is reflexive transitive closure**
- Square Matrices over a KA **K**

Important for relational verification

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$

s

Kleene Algebra, examples

- Regular languages
  - **+ is union**
  - **; is pointwise co**
  - *** is iteration**
- Binary Relations
  - **+ is union**
  - **; is relational compositi**
  - *** is reflexive transitive c**
- Square Matrices over a KA **K**

Important for relational verification

Important for automata representations

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$

s

Some results on KA

Some results on KA

➡ Can we axiomatize the valid equations?

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

YES [Kozen 90]

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

YES [Kozen 90]

➡ Can we decide if expressions are equivalent?

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

YES [Kozen 90]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [(Stock+1)Meyer 74]

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

YES [Kozen 90]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [(Stock+1)Meyer 74]

➡ Can we devise more efficient procedures?

Some results on KA

➡ Can we axiomatize the valid equations?

YES [Salomaa 66]

➡ Equational axiomatization?

YES [Kozen 90]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [(Stock+1)Meyer 74]

➡ Can we devise more efficient procedures?

YES, e.g. antichains [Doyen et al, 2010] or up-to bisimulation [Bonchi-Pous, 2015]

Kleene Algebra - applications

➡ Design and analysis of algorithms
- shortest paths
- connectivity

➡ Program logic and verification
- program analysis
- optimization

Kleene Algebra - applications

➡ Design and analysis of algorithms
- shortest paths
- connectivity

➡ Program logic and verification
- program analysis
- optimization

Missing in KA: **Boolean** tests for control structures

Adding tests

(**K**, **B**, **+**, **;**, *, **~**, **0**, **1**)

- (**K**, **+**, **;**, *, **0**, **1**) is a Kleene algebra
- (**B**, **+**, **;** , **~**, **0**, **1**) is a Boolean algebra
- **B** ⊆ **K**

Adding tests

(**K**, **B**, **+**, **;**, *, **~**, **0**, **1**)

- (**K**, **+**, **;**, *, **0**, **1**) is a Kleene algebra
- (**B**, **+**, **;** , **~**, **0**, **1**) is a Boolean algebra
- **B** ⊆ **K**

```
if b then p else q = b;p + ~b;q
while b do p = (b;p)*~b
```

Axioms of Boolean Algebra

$$a + 0 \equiv a \qquad a + 1 \equiv 1$$

$$a + b \equiv b + a \qquad a + a \equiv a$$

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

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

$$\sim (\sim a) \equiv a$$

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

$$a;1 \equiv a \qquad a;0 \equiv 0 \qquad a;b \equiv b;a$$

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

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

Axioms of Boolean Algebra

$a + 0 \equiv a$ $\quad$ $a + 1 \equiv 1$ $\quad$ $\sim(b+c) \equiv \sim b\ ;\sim c$

$a + b \equiv b + a$ $\quad$ $a + a \equiv a$ $\quad$ $\sim (\sim a) \equiv a$

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

$a;1 \equiv a$ $\quad$ $a;0 \equiv 0$ $\quad$ $a;b \equiv b;a$

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

+ and ; have double meaning

Guarded Strings

Σ - action symbols T - set of tests

B - Free Boolean algebra generated by T

At = atoms of B = {α,β,...}

$\alpha_0 p_0 \alpha_1 p_1 \alpha_2 \cdots \alpha_{n-1} p_{n-1} \alpha_n \in (\mathrm{At} \cdot \Sigma)^* \cdot \mathrm{At}$

state of program variables

KAT - semantics

$$\begin{aligned}
sem(b) &= \{\alpha \mid \alpha \leq b\} \\
sem(p) &= \{\alpha p \beta \mid \alpha,\beta \text{ atoms}\} \\
sem(e+f) &= sem(e) \cup sem(f) \\
sem(e;f) &= sem(e) \,.\, sem(f) \\
sem(e^*) &= \bigcup_n sem(e)^n
\end{aligned}$$

KAT - semantics

$$\begin{aligned}
\mathrm{sem}(b) &= \{\alpha \mid \alpha \leq b\} \\
\mathrm{sem}(p) &= \{\alpha p \beta \mid \alpha, \beta \in \mathrm{At}\} \\
\mathrm{sem}(e+f) &= \mathrm{sem}(e) \cup \mathrm{sem}(f) \\
\mathrm{sem}(e;f) &= \mathrm{sem}(e) \cdot \mathrm{sem}(f) \\
\mathrm{sem}(e^*) &= \bigcup_n \mathrm{sem}(e)^n
\end{aligned}$$

guarded concatenation

$L.K = \{x\alpha y \mid x\alpha \in L, \alpha y \in K\}$

Examples

```
if b then p else q = b;p + ~b;q
while b do p = (b;p)*~b
```

Examples

```
if b then p else q = b;p + ~b;q
while b do p = (b;p)*~b
```

```
sem(b;p + ~b;q)
= {αpβ|α <= b} U {αqβ|α <= ~b}
```

Examples

```
if b then p else q = b;p + ~b;q
while b do p = (b;p)*~b
```

sem(b;p + ~b;q)

= {αpβ|α <= b} U {αqβ|α <= ~b}

sem((b;p)*~b)

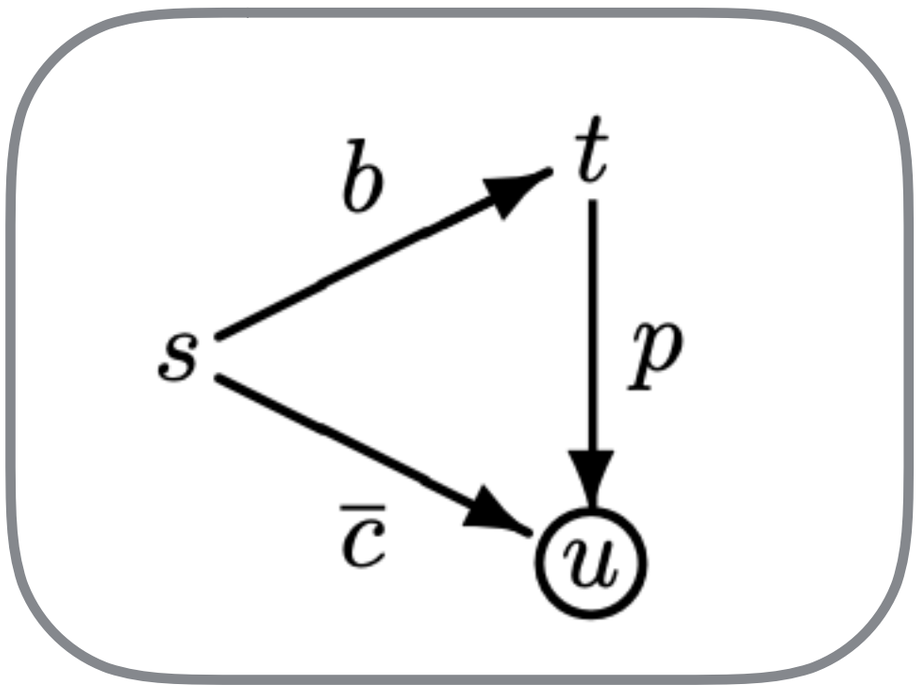
= {α|α <= ~b} U {αpβ | α<=b, β<=~b}

U {$α_0$p$α_1$pβ | $α_i$<=b, β<=~b} U …

Automata

$\alpha_0 p_0 \alpha_1 p_1 \alpha_2 \cdots \alpha_{n-1} p_{n-1} \alpha_n \in (At \cdot \Sigma)^* \cdot At$

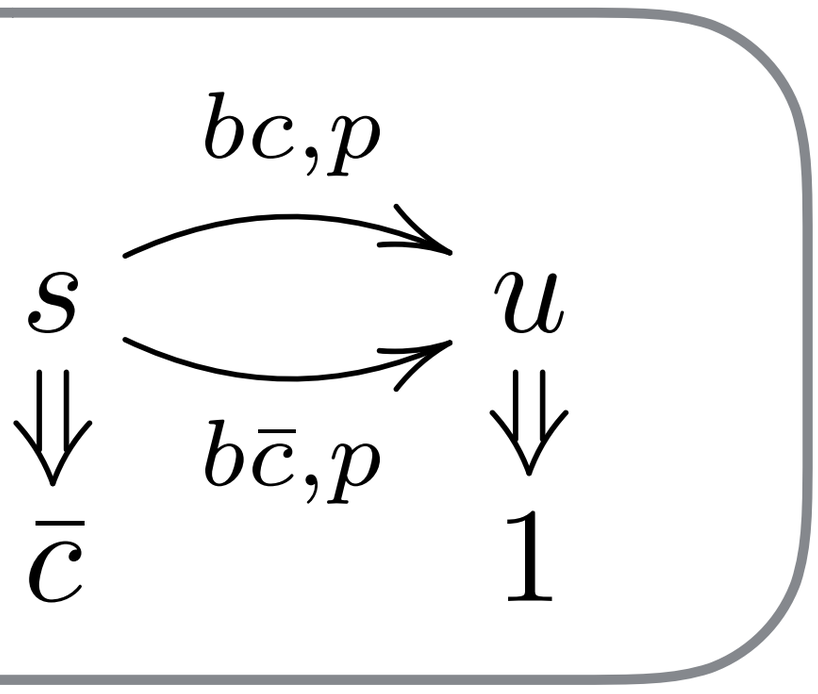
non-deterministic



t: S x (B + Σ) —>P(S)

o: S —> 2

deterministic



t: S x (At x Σ) —>S

o: S —> B

KAT - some results

KAT - some results

- ➡ Kleene Theorem

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

YES, equationally [Kozen 03]

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

YES, equationally [Kozen 03]

➡ Can we decide if expressions are equivalent?

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

YES, equationally [Kozen 03]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [Chen-Pucella 08, Kozen 09]

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

YES, equationally [Kozen 03]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [Chen-Pucella 08, Kozen 09]

➡ Connections to propositional Hoare Logic and PDL

KAT - some results

➡ Kleene Theorem

YES, deterministic automata [Kozen 09]

➡ Can we axiomatize the valid equations?

YES, equationally [Kozen 03]

➡ Can we decide if expressions are equivalent?

YES, PSPACE complete [Chen-Pucella 08, Kozen 09]

➡ Connections to propositional Hoare Logic and PDL

```
{b}p{c} <=> bp~c ≡ 0
```

Break exercises

1. x*x* = x*
2. x*=x**
3. (x+y)* = (x*y)*x* **denesting**
4. x(yx)* = (xy)*x **sliding**
5. xy=yz => x*y = yz*

Denesting

$$(x+y)^* = (x^*y)^*x^*$$

Denesting

$$(x+y)^* = (x^*y)^*x^*$$

$1 \leq (x^*y)^*x^*$

$xx^*(yx^*)^* \leq x^*(yx^*)^*$

$yx^*(yx^*)^* \leq (yx^*)^* \leq x^*(yx^*)^*$

Denesting

$$(x+y)^* = (x^*y)^*x^*$$

$1 \le (x^*y)^*x^*$

$xx^*(yx^*)^* \le x^*(yx^*)^*$

$yx^*(yx^*)^* \le (yx^*)^* \le x^*(yx^*)^*$

monoticity + distributivity

$1 + (x+y)\,x^*(yx^*)^* \le x^*(yx^*)^*$

Denesting

$$(x+y)^* = (x^*y)^*x^*$$

$1 \le (x^*y)^*x^*$

$xx^*(yx^*)^* \le x^*(yx^*)^*$

$yx^*(yx^*)^* \le (yx^*)^* \le x^*(yx^*)^*$

monoticity + distributivity

$1 + (x+y)\,x^*(yx^*)^* \le x^*(yx^*)^*$

fixpoint rule

$f + ex \le x \to e^*f \le x$

$$(x+y)^* \le (x^*y)^*x^*$$

KAT - applications

Compiler Optimizations [Kozen & Patron 00]

- dead code elimination
- common subexpression elimination
- copy propagation
- loop hoisting
- **loop unrolling**
- **reducing nesting depth**
- induction variable elimination
- instruction scheduling
- elimination of redundant instructions
- array bounds check elimination

Example - loop denesting

```
while b do {                    if b then {
  p;                      ≡       p;
  while c do q;                   while b ∨ c do {
}                                   if c then q else p
                                  }
                                }
```

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

Proof

$$(bp(cq)*\sim c)*\sim b \equiv bp((b+c)(cq+\sim cp))*\sim(b+c) + \sim b$$

Proof

$$(bp(cq)^*{\sim}c)^*{\sim}b \equiv bp((b+c)(cq+{\sim}cp))^*{\sim}(b+c) + {\sim}b$$

$$bp((b+c)(cq+{\sim}cp))^*{\sim}(b+c) + {\sim}b$$

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

bp(bcq+b~cp+cq + c~cq)*~b~c + ~b

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

bp(bcq+b~cp+cq + c~cq)*~b~c + ~b

≡ {Comm + Dist + Boolean Alg}

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

bp(bcq+b~cp+cq + c~cq)*~b~c + ~b

≡ {Comm + Dist + Boolean Alg}

bp(cq+~cbp)*~b~c + ~b

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

bp(bcq+b~cp+cq + c~cq)*~b~c + ~b

≡ {Comm + Dist + Boolean Alg}

bp(cq+~cbp)*~b~c + ~b

≡ {Denesting: (e+f)*=(e*f)*e*}

Proof

(bp(cq)*~c)*~b ≡ bp((b+c)(cq+~cp))*~(b+c) + ~b

bp((b+c)(cq+~cp))*~(b+c) + ~b

≡ {Distr+Boolean Alg}

bp(bcq+b~cp+cq + c~cq)*~b~c + ~b

≡ {Comm + Dist + Boolean Alg}

bp(cq+~cbp)*~b~c + ~b

≡ {Denesting: (e+f)*=(e*f)*e*}

bp((cq)*~cbp)*(cq)*)~b~c + ~b

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$

$\equiv$ {Distr+Boolean Alg}

$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$

$\equiv$ {Comm + Dist + Boolean Alg}

$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$

$\equiv$ {Denesting: $(e+f)^*=(e^*f)^*e^*$}

$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$

$\equiv$ {Sliding: $(ef)^*e=e(fe)^*$}

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Distributivity}\}$$

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Distributivity}\}$$

$$((bp(cq)^*\sim c)^*bp(cq)^*\sim c + 1)\sim b$$

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Distributivity}\}$$

$$((bp(cq)^*\sim c)^*bp(cq)^*\sim c + 1)\sim b$$

$$\equiv \{\text{FP}\}$$

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Distributivity}\}$$

$$((bp(cq)^*\sim c)^*bp(cq)^*\sim c + 1)\sim b$$

$$\equiv \{\text{FP}\}$$

$$(bp(cq)^*\sim c)^*\sim b$$

Proof

$$(bp(cq)^*\sim c)^*\sim b \equiv bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$bp((b+c)(cq+\sim cp))^*\sim(b+c) + \sim b$$

$$\equiv \{\text{Distr+Boolean Alg}\}$$

$$bp(bcq+b\sim cp+cq + c\sim cq)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Comm + Dist + Boolean Alg}\}$$

$$bp(cq+\sim cbp)^*\sim b\sim c + \sim b$$

$$\equiv \{\text{Denesting: } (e+f)^*=(e^*f)^*e^*\}$$

$$bp((cq)^*\sim cbp)^*(cq)^*)\sim b\sim c + \sim b$$

$$\equiv \{\text{Sliding: } (ef)^*e=e(fe)^*\}$$

$$(bp(cq)^*\sim c)^*bp(cq)^*\sim b\sim c + \sim b$$

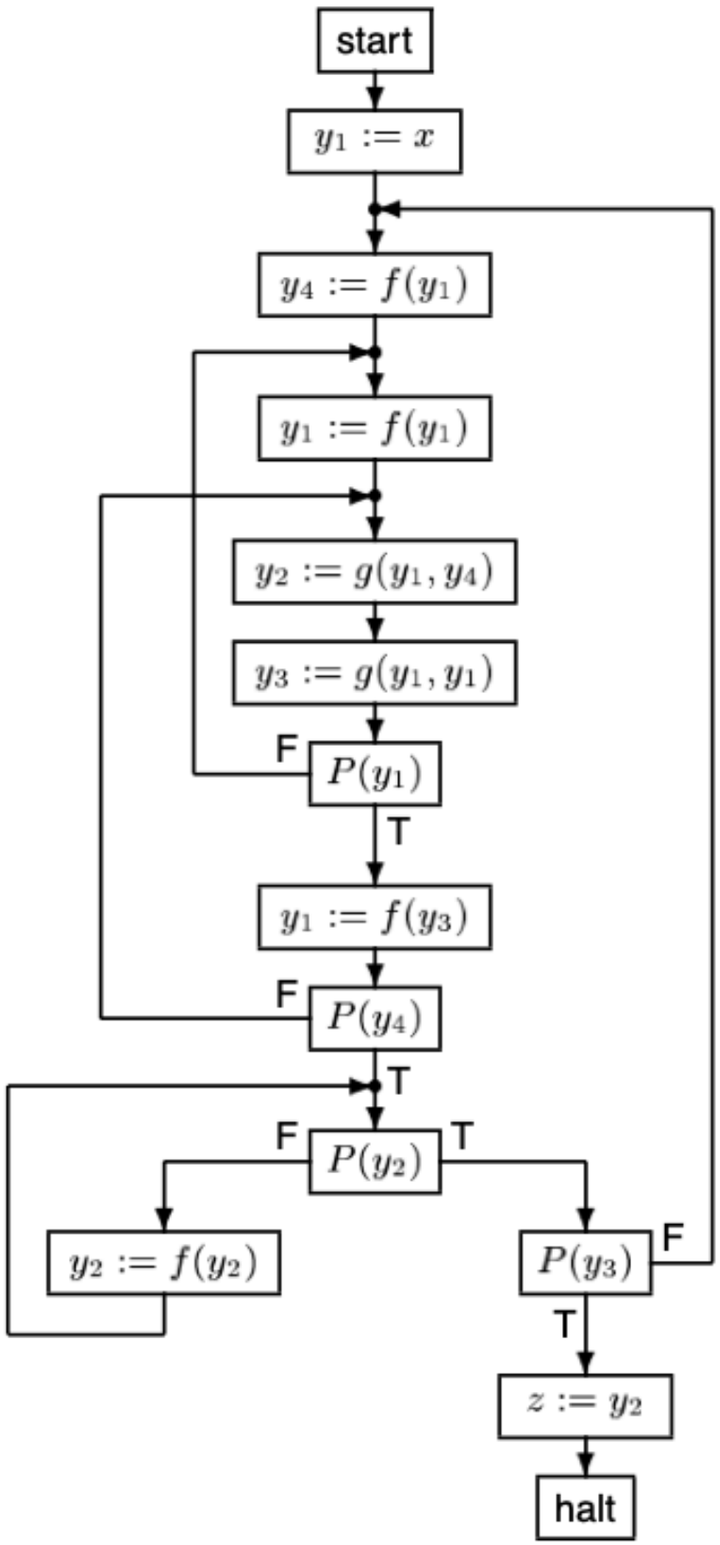
$$\equiv \{\text{Distributivity}\}$$

$$((bp(cq)^*\sim c)^*bp(cq)^*\sim c + 1)\sim b$$

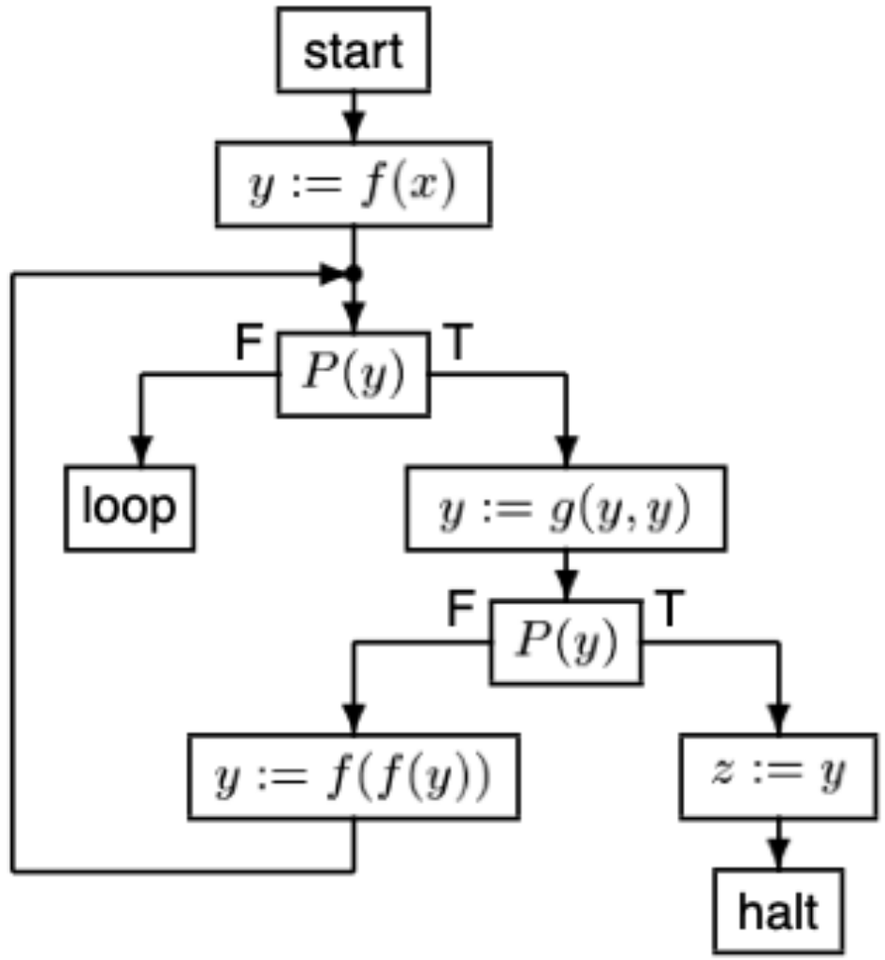
$$\equiv \{\text{FP}\}$$

$$(bp(cq)^*\sim c)^*\sim b$$

Scheme equivalence

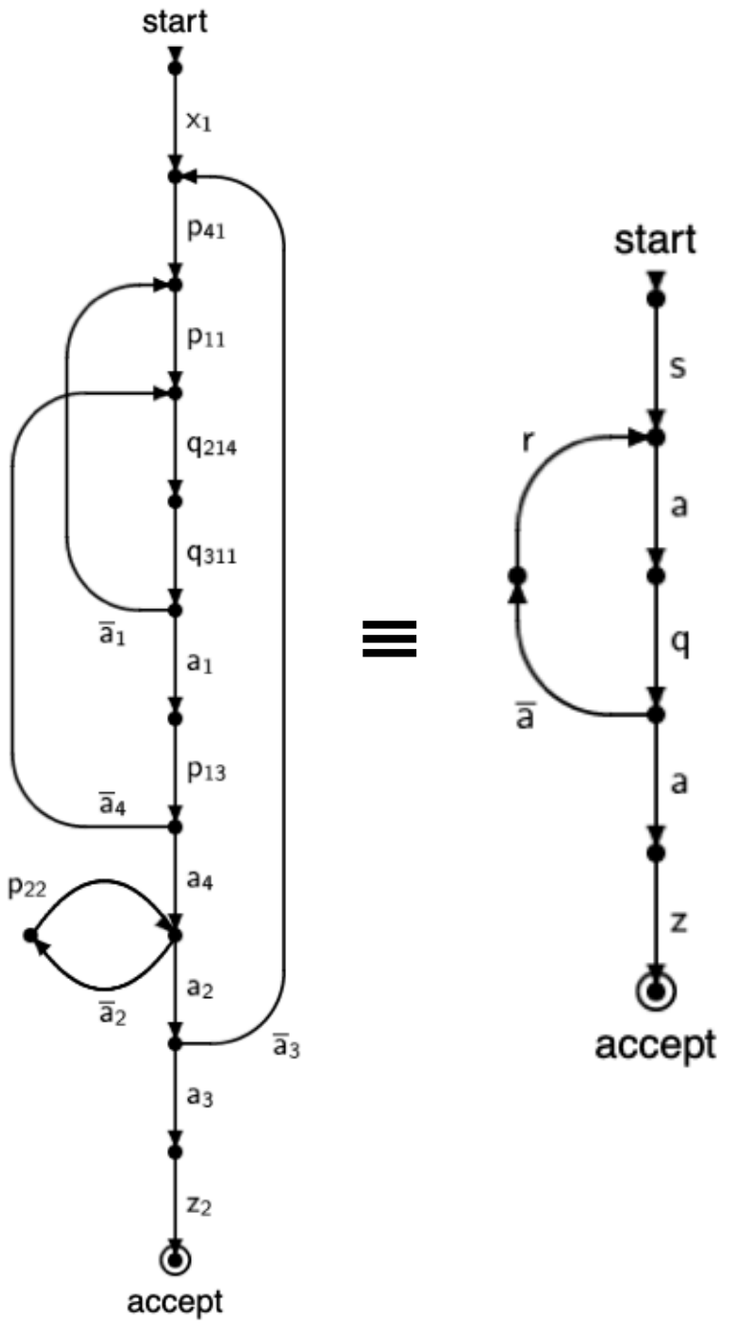


≡



Example of Paterson from [Manna 74]

Scheme equivalence



≡

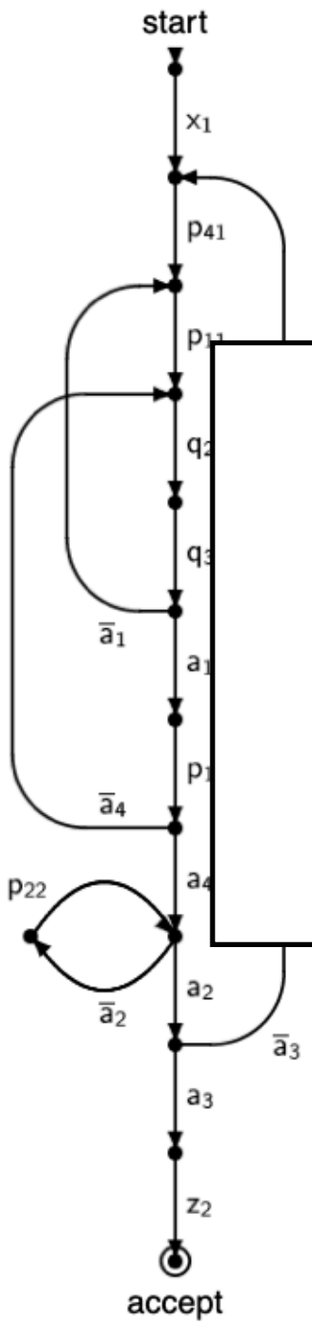
Kleene Theorem

$x_1 p_{41} p_{11} q_{214} q_{311}$
$(\sim a_1 p_{11} q_{214} q_{311})^* a_1 p_{13}$ …
$a_4 (\sim a_2 p_{22})^* a_2 a_3 z_2$

≡

$saq(\sim araq)^* az$

Scheme equivalence



start

$$
\begin{aligned}
p_{41}p_{11}q_{214} &= y_4 := f(y_1)\,;\, y_1 := f(y_1)\,;\, y_2 := g(y_1, y_4) \\
&= y_1 := f(y_1)\,;\, y_4 := y_1\,;\, y_2 := g(y_1, y_4) \\
&= y_1 := f(y_1)\,;\, y_4 := y_1\,;\, y_2 := g(y_1, y_1) \\
&= y_4 := f(y_1)\,;\, y_1 := f(y_1)\,;\, y_2 := g(y_1, y_1) \\
&= p_{41}p_{11}q_{211}.
\end{aligned}
$$

accept

p13 …

Bohm-Jacopini Theorem

Folk Theorem

Every **while** program can be simulated by a **while** program with at most one **while** loop as long as extra Boolean variables are allowed.

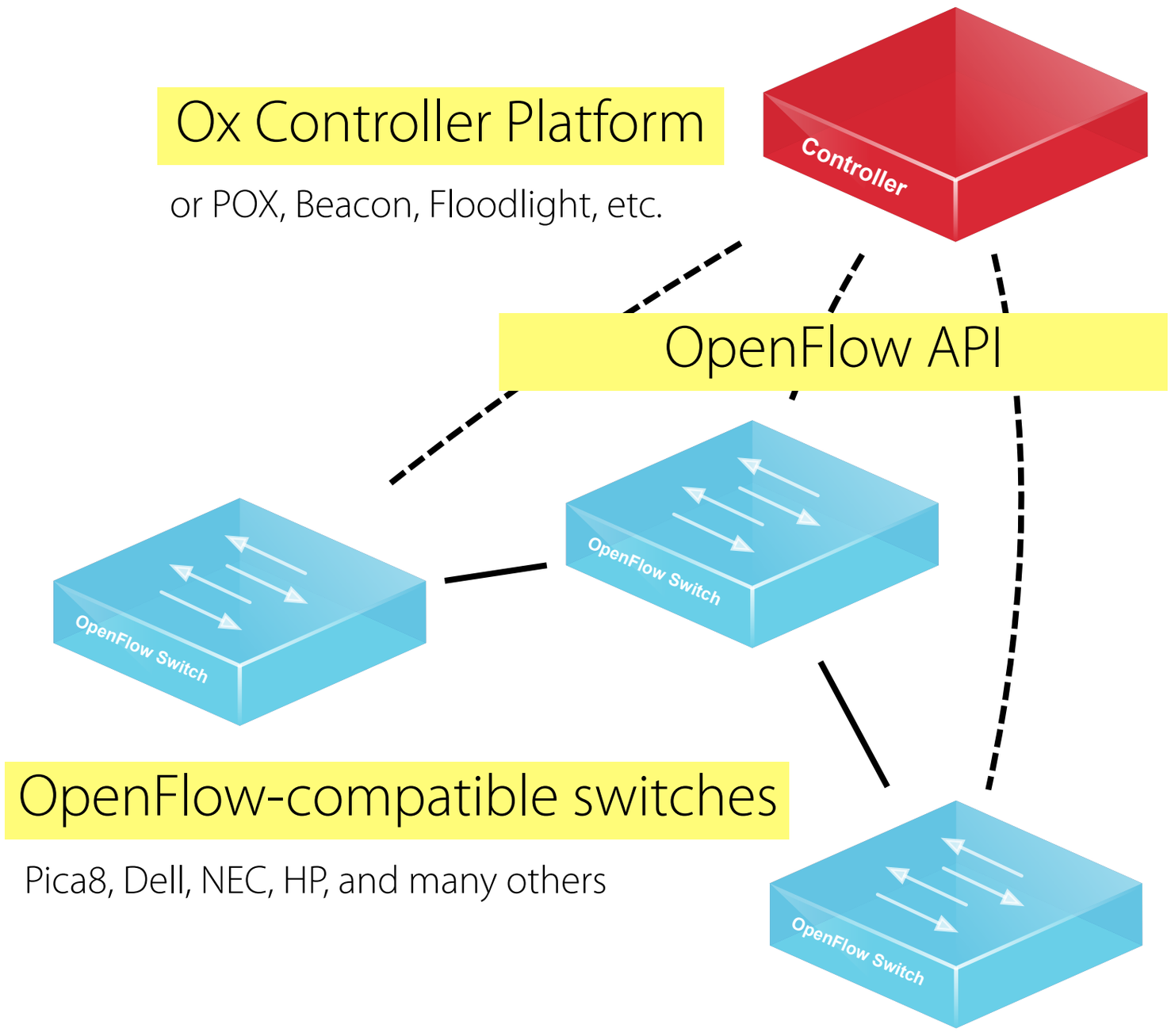
Dexter Kozen. ***Kleene algebra with tests****. ACM Trans. Programming Languages and Systems (TOPLAS), 19(3):427-443, May 1997*

Programming and Reasoning with Kleene Algebra with Tests

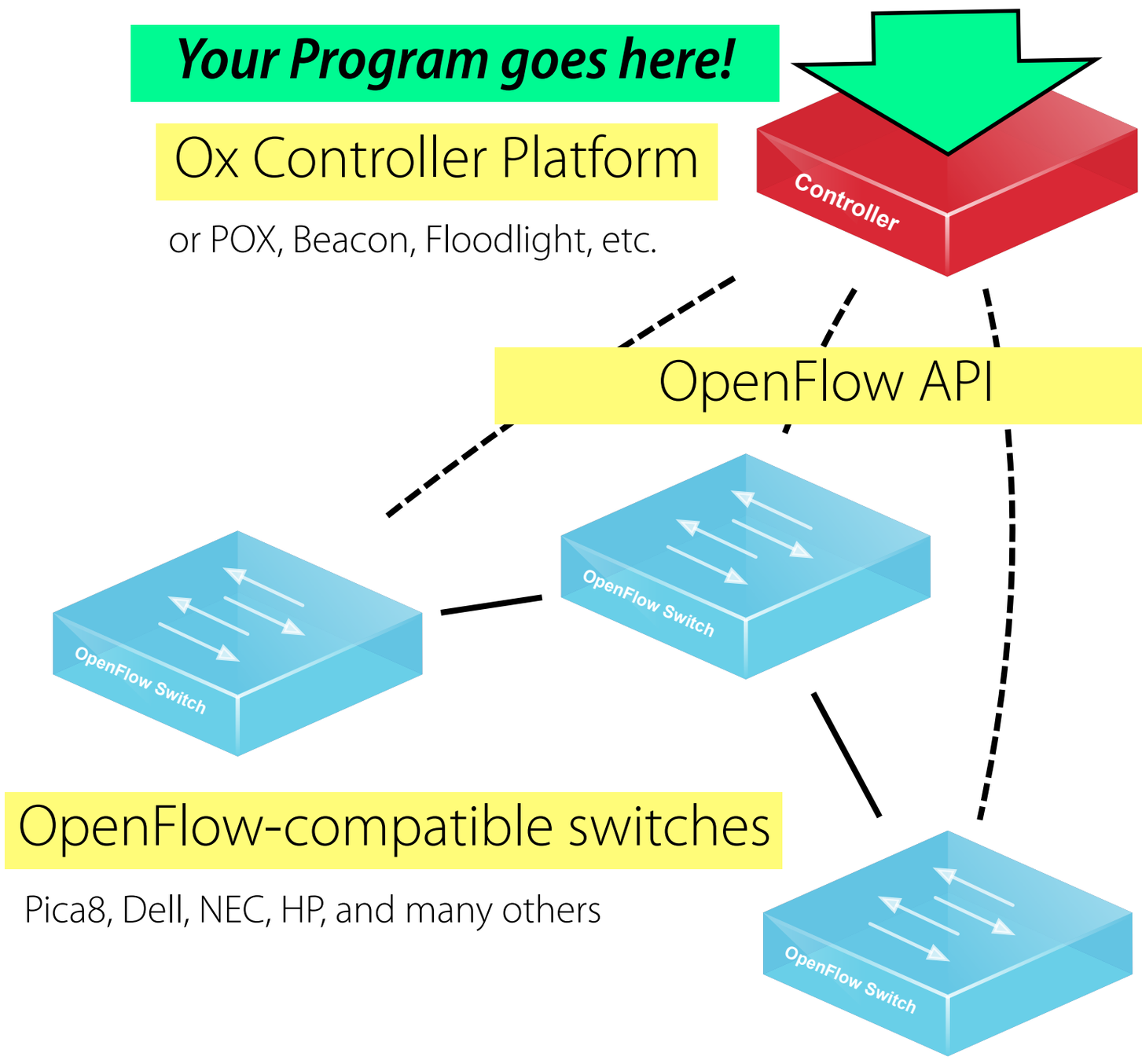
Part 2: Extensions & Applications

(Nate Foster + Dexter Kozen + Alexandra Silva)*

SDN Architecture

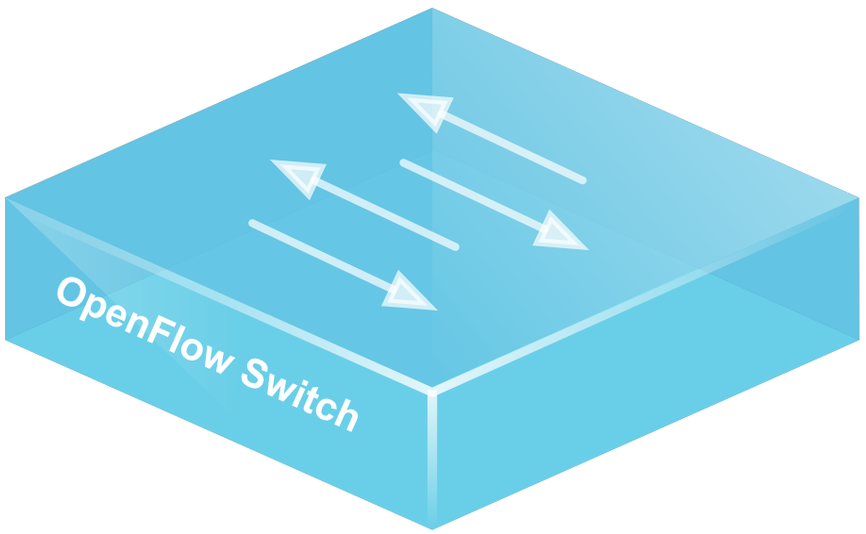


SDN Architecture



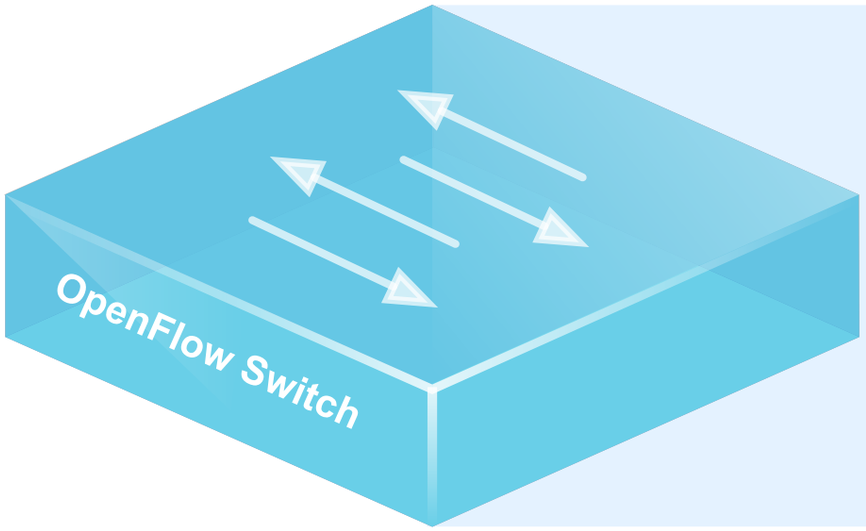
OpenFlow Switch

General-purpose packet-processing device that can be used to implement switches, routers, firewalls, etc.



OpenFlow Switch

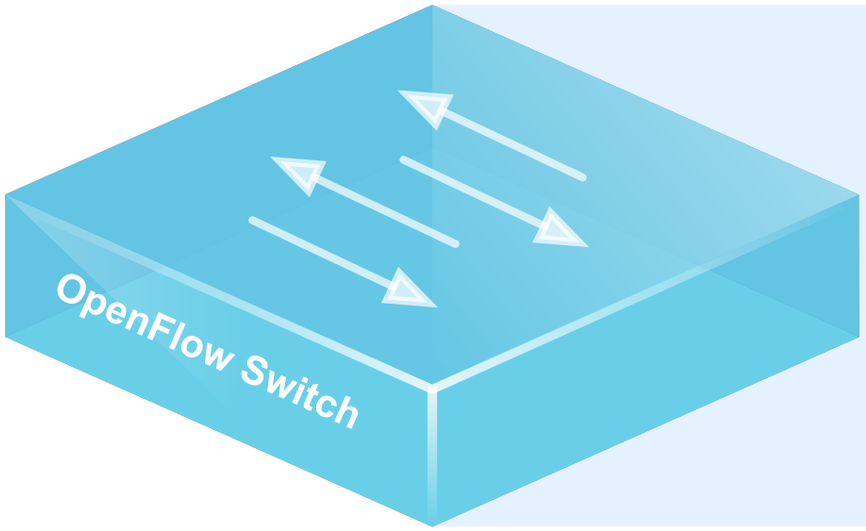
General-purpose packet-processing device that can be used to implement switches, routers, firewalls, etc.



Match	Actions	Counters
10.0.0.1	Drop	(73,2458)
10.0.0.2	Forward 2	(16,846)
10.0.0.3	Forward 3	(23,5729)
*	Controller	(5,472)

OpenFlow Switch

General-purpose packet-processing device that can be used to implement switches, routers, firewalls, etc.

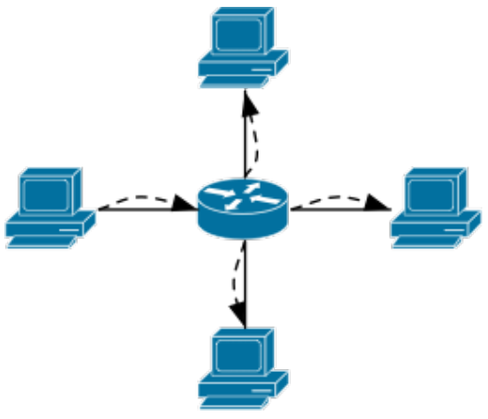


Match	Actions	Counters
10.0.0.1	Drop	(73,2458)
10.0.0.2	Forward 2	(16,846)
10.0.0.3	Forward 3	(23,5729)
*	Controller	(5,472)

Key data structure is a *flow table* containing a prioritized list of match-action *rules* and *counters*

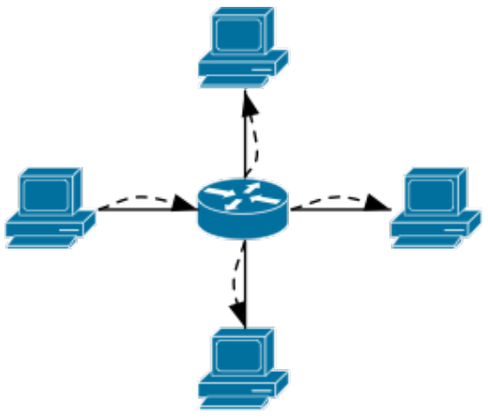
Example

Topology



Example

Topology



Specification

- Forward packets to hosts 1-4
- Monitor traffic to unknown hosts
- Flood broadcast traffic to all hosts
- Disallow SSH traffic from hosts 1-2

Flow Table

```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=1},action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=2},action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=3},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=4},action=[Output(3), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff},action=[]}
{pattern={},action=[Controller]}
```

Example: Forward

```
let forward =
  if ethDst = 00:00:00:00:00:01 then
    port := 1
  else if ethDst = 00:00:00:00:00:02 then
    port := 2
  else if ethDst = 00:00:00:00:00:03 then
    port := 3
  else if ethDst = 00:00:00:00:00:04 then
    port := 4
  else
    false
```

Example: Broadcast

```
let flood =
  if port = 1 then
    port := 2 + port := 3 + port := 4
  else if port = 2 then
    port := 1 + port := 3 + port := 4
  else if port = 3 then
    port := 1 + port := 2 + port := 4
  else if port = 4 then
    port := 1 + port := 2 + port := 3
  else
    false

let broadcast =
  if ethDst = ff:ff:ff:ff:ff:ff then
    flood
  else
    false
```

Example: Routing

```
let route = forward + broadcast
```

Example: Monitor

```
let monitor =
  if !(ethDst = 00:00:00:00:00:01 +
       ethDst = 00:00:00:00:00:02 +
       ethDst = 00:00:00:00:00:03 +
       ethDst = 00:00:00:00:00:04 +
       ethDst = ff:ff:ff:ff:ff:ff) then
    port := unknown
  else
    false
```

Example: Firewall

```
let firewall =
  if (ethSrc = 00:00:00:00:00:01 +
      ethSrc = 00:00:00:00:00:02) ;
     ethTyp = 0x800 ;
     ipProto = 0x06 ;
     tcpDstPort = 22 then
     false
  else
     true
```

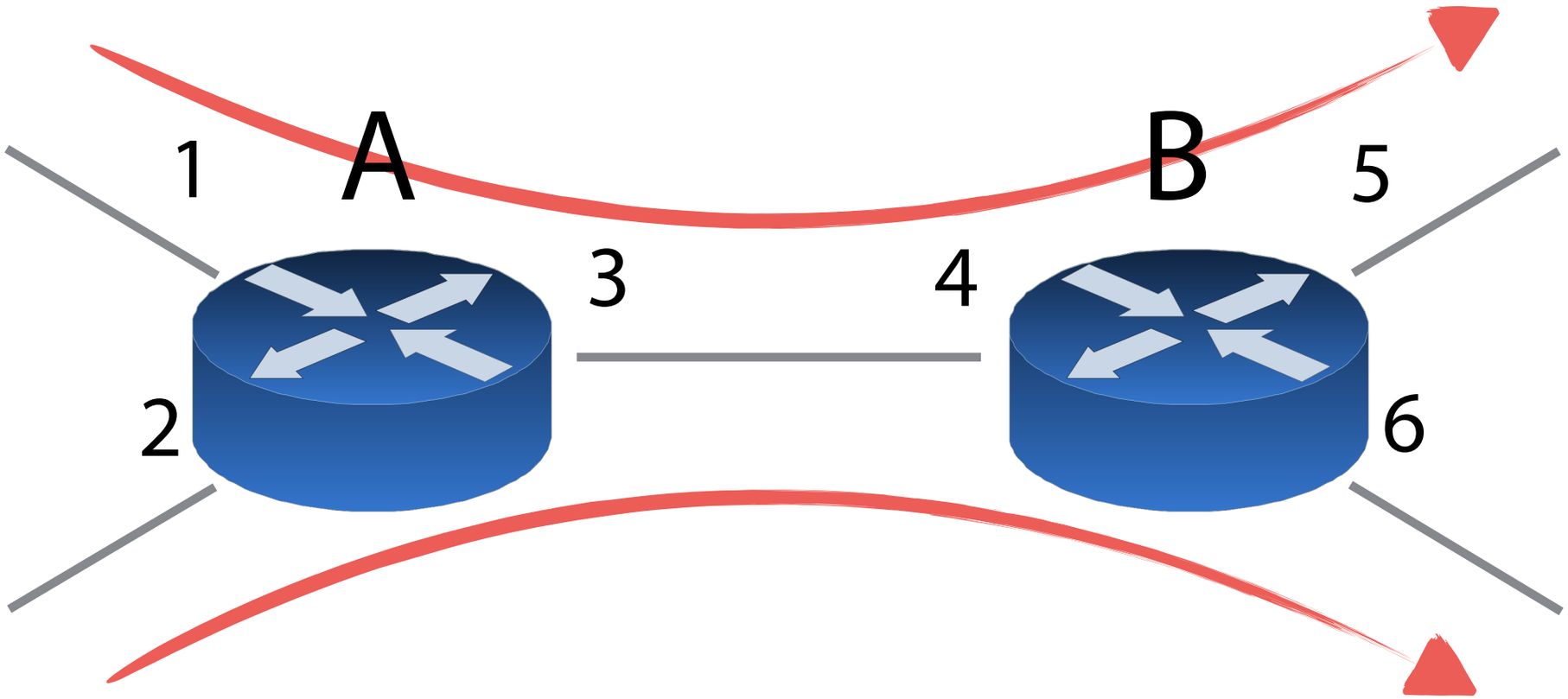
Example: Main Policy

```
let main = (route + monitor); firewall
```

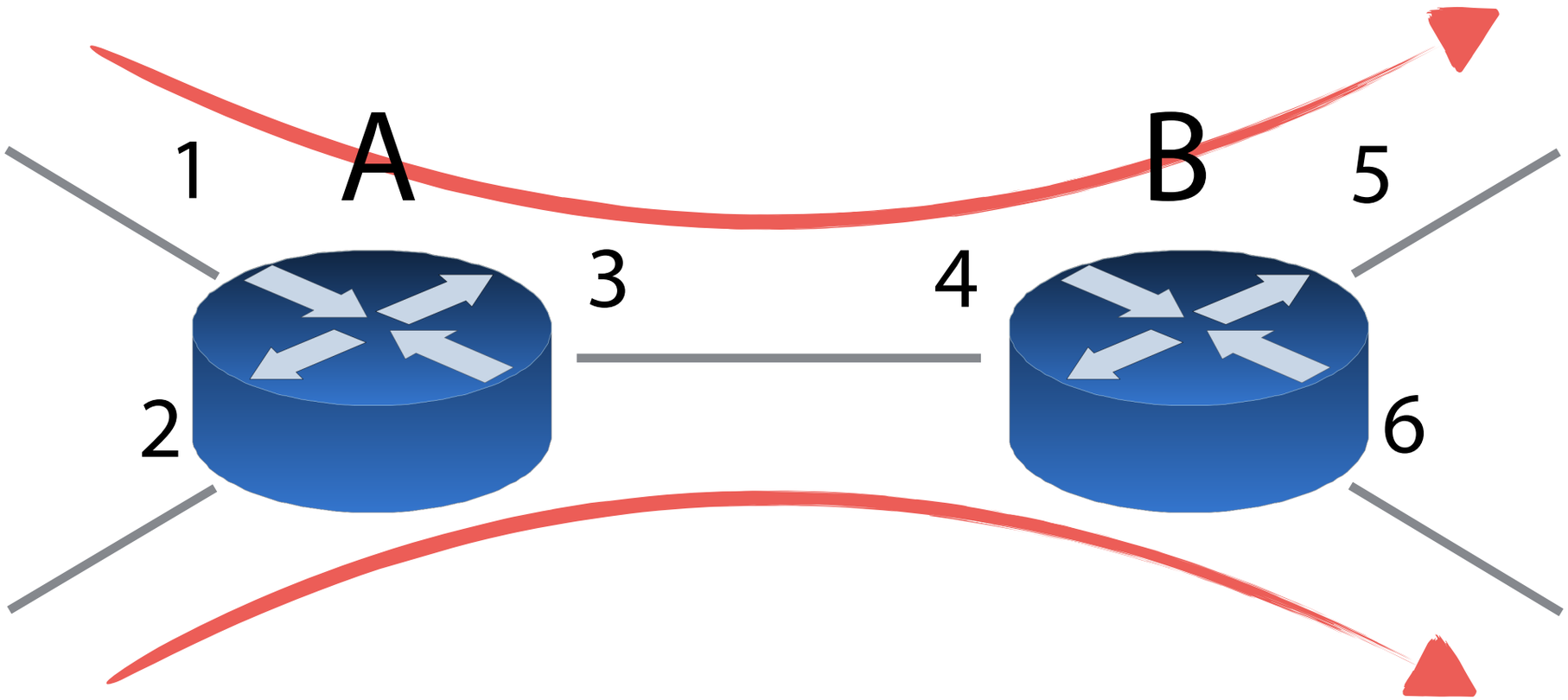
compiles to…

```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=1},action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=2},action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=3},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=4},action=[Output(3), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff},action=[]}
{pattern={},action=[Controller]}
```

Example: Paths



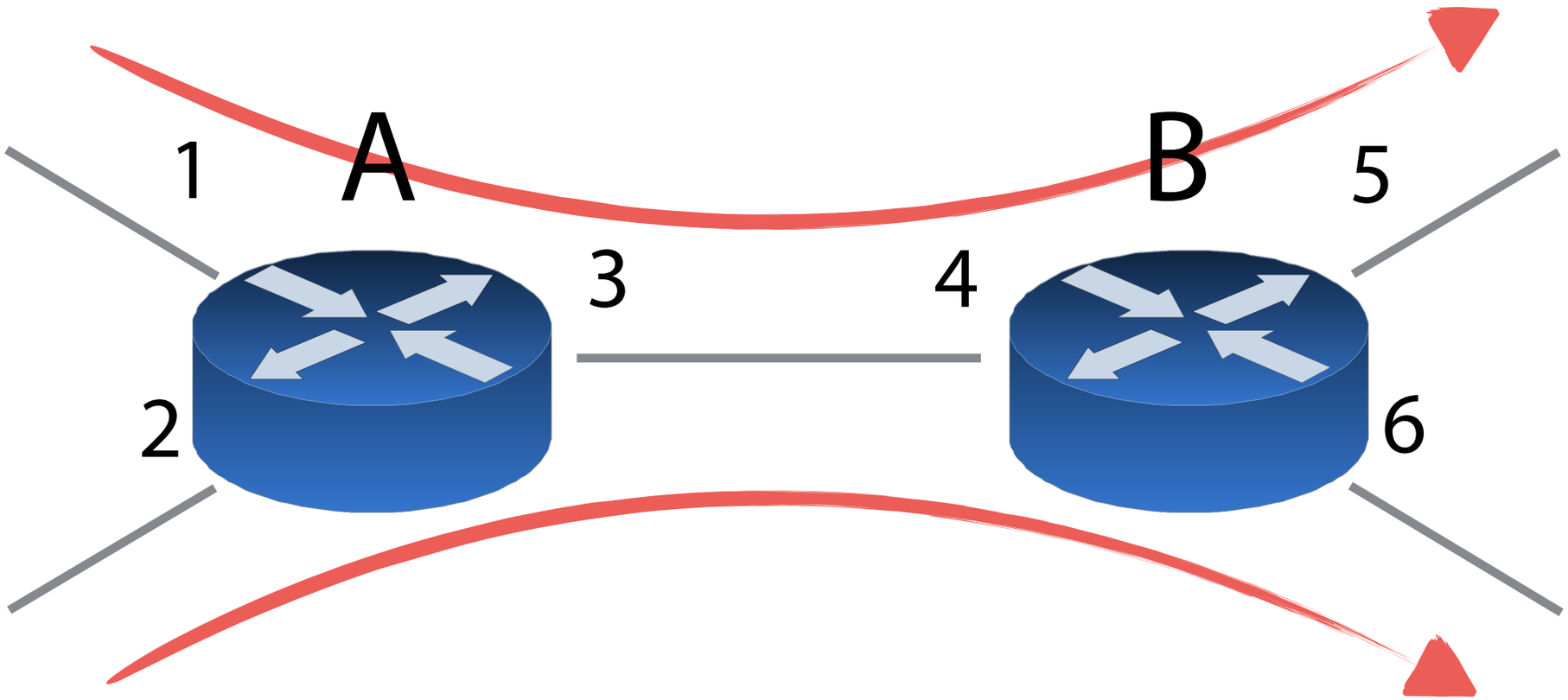
Local Program



pol_A

pol_B

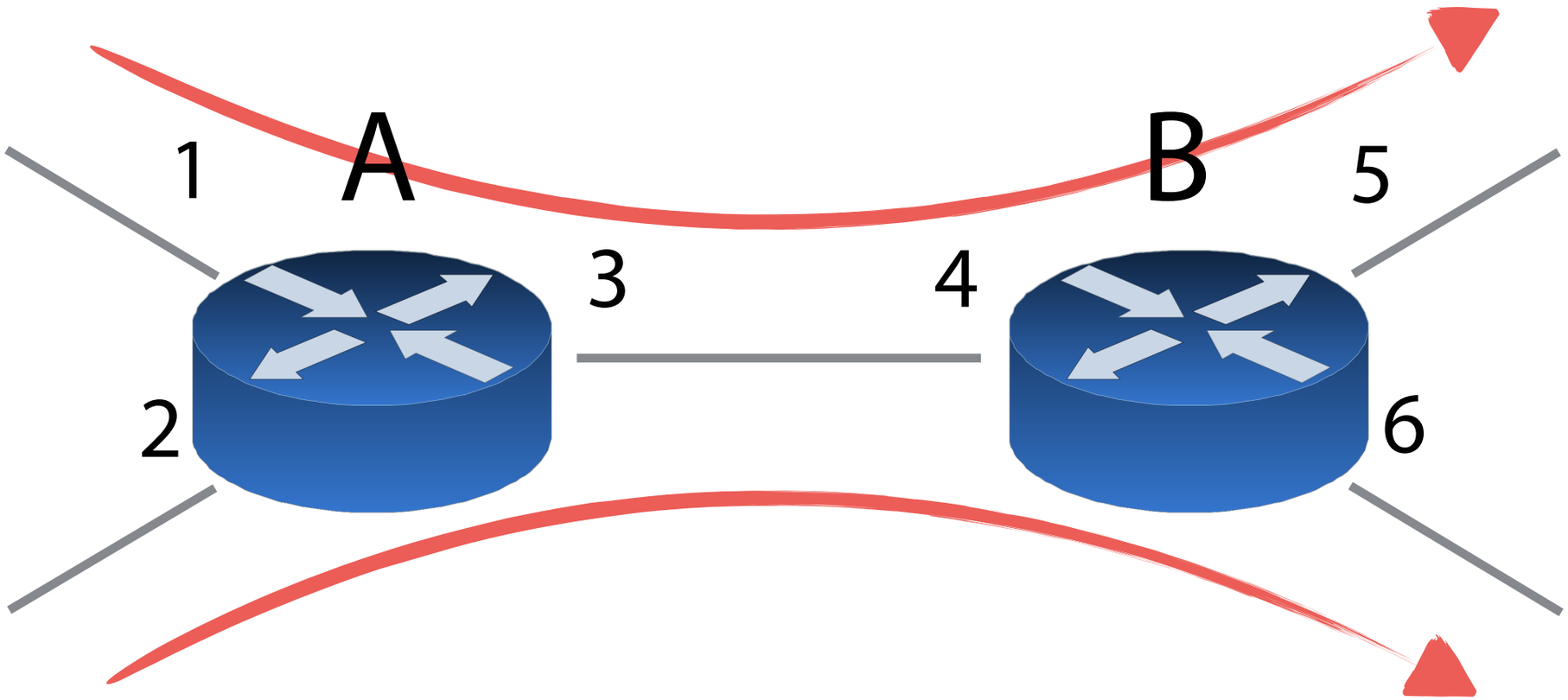
Local Program



port:=3

???

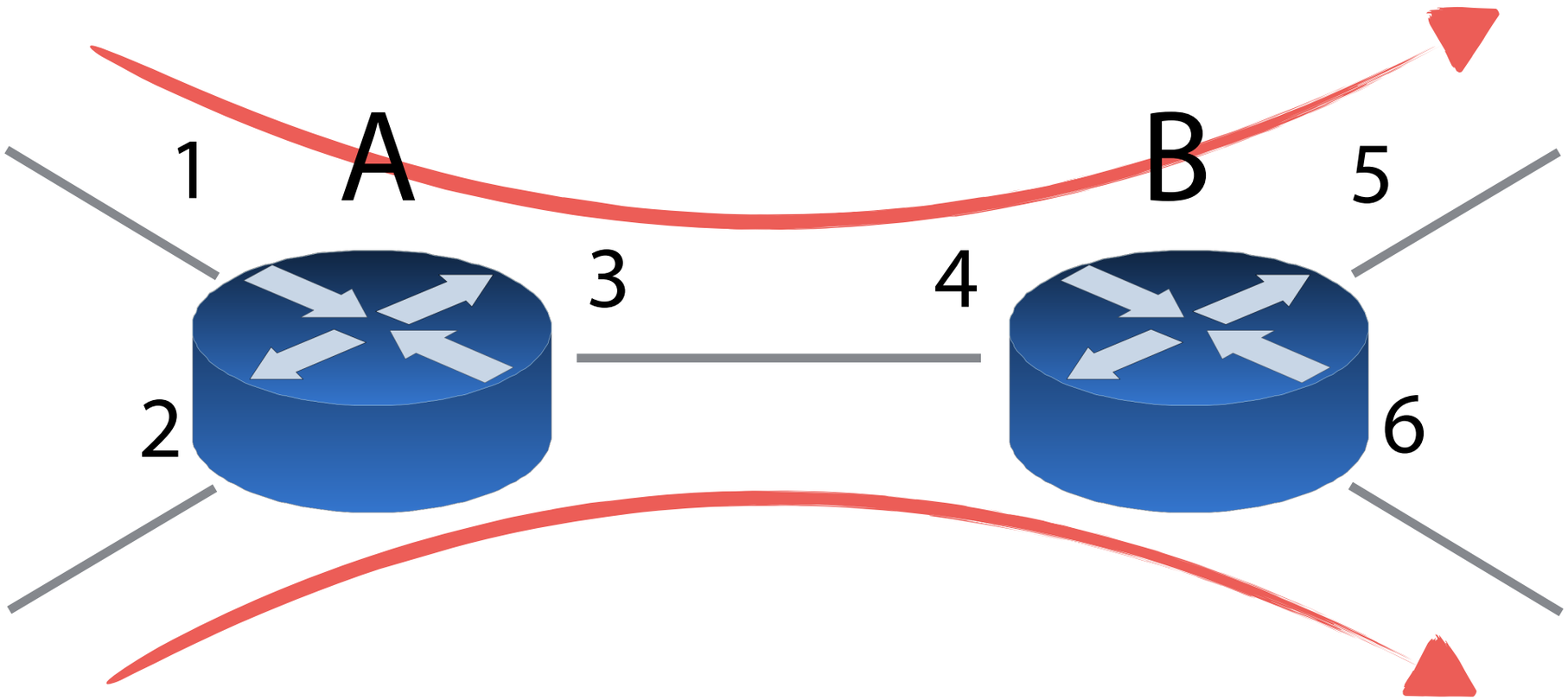
Local Program



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

???

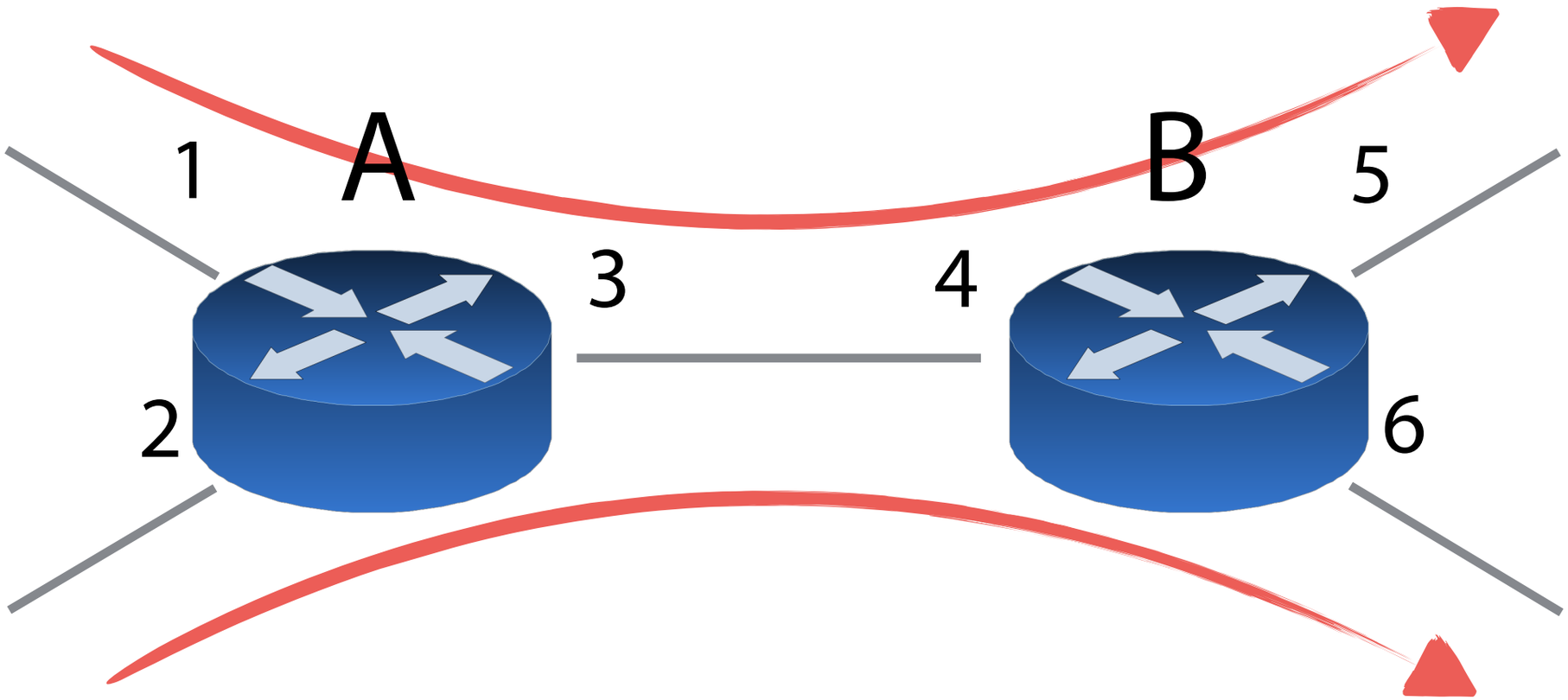
Local Program



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

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

Local Program

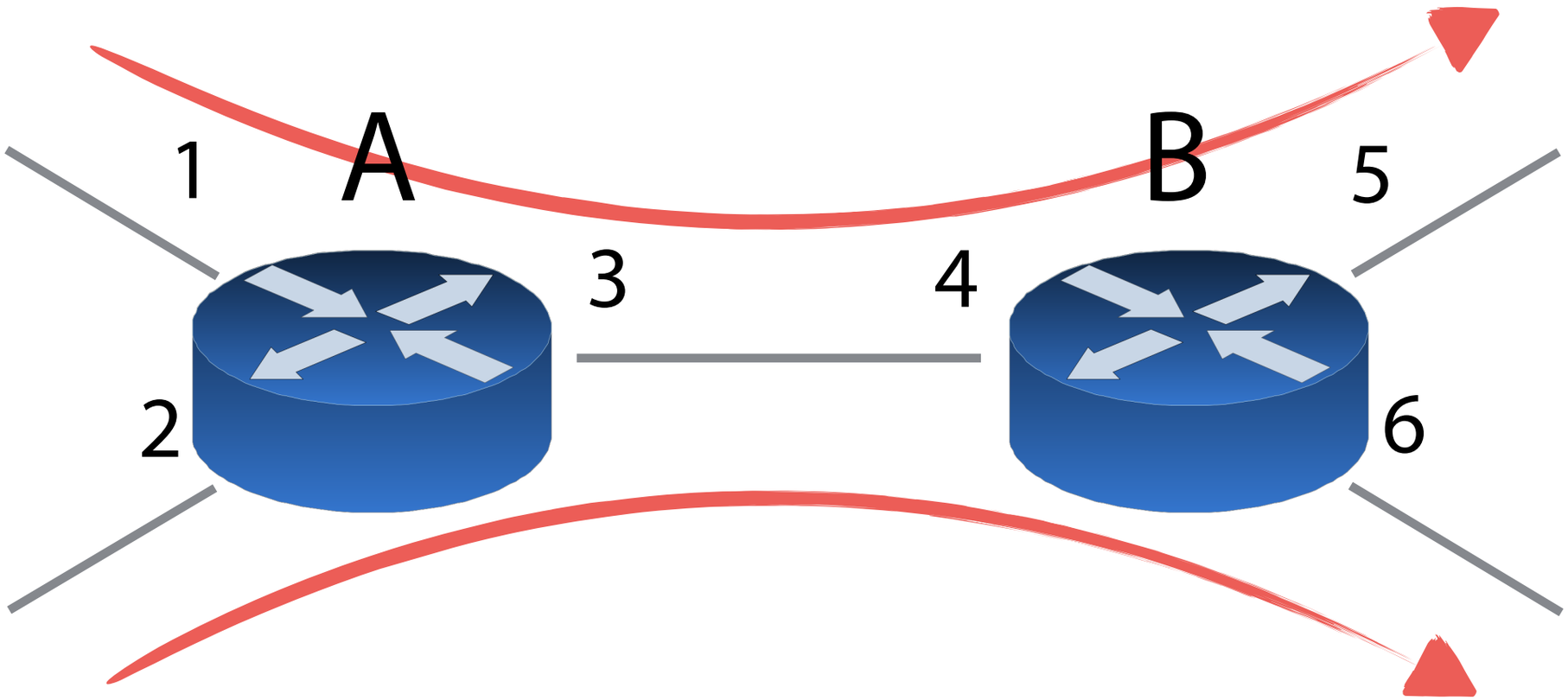



```
port=1; tag:=1; port:=3
        +
port=2; tag:=2; port:=3
```

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

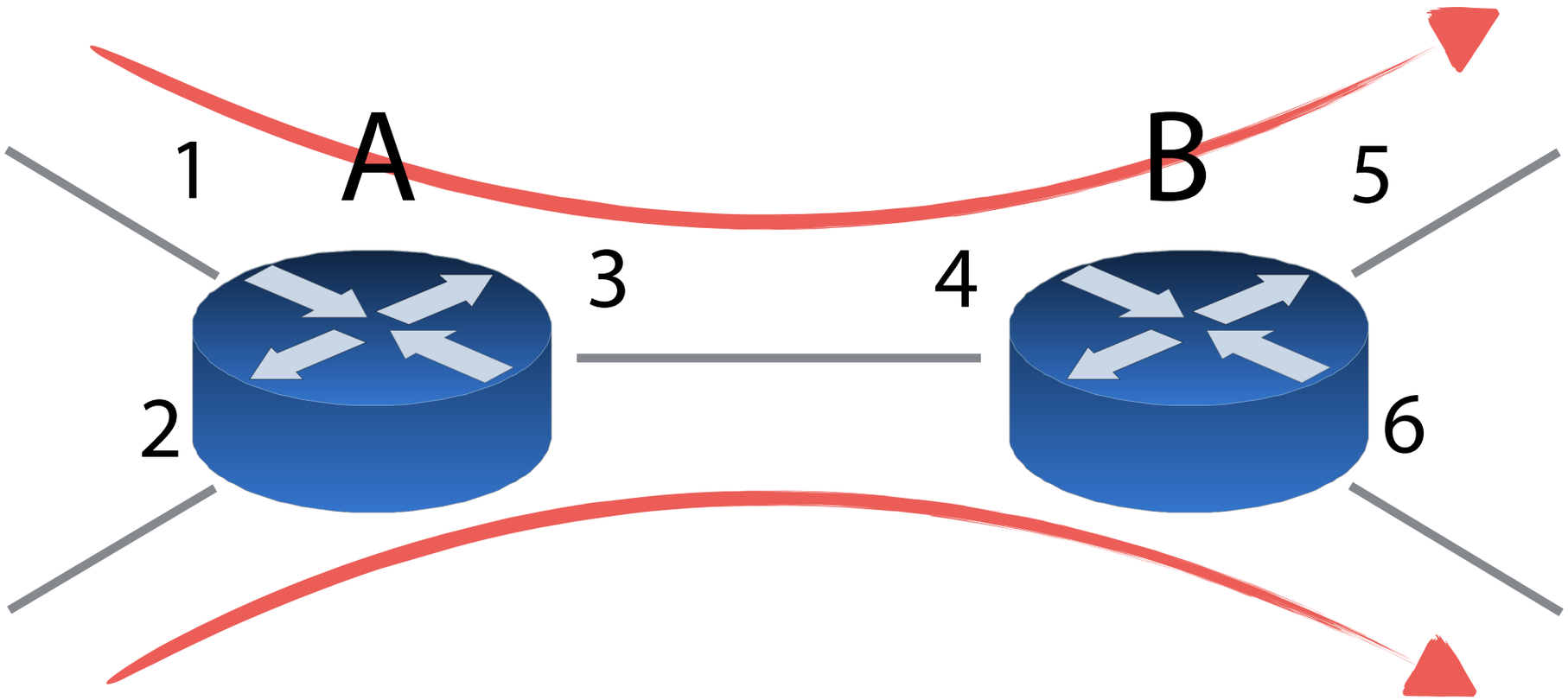
tedious for programmers… difficult to get right!

Global Program



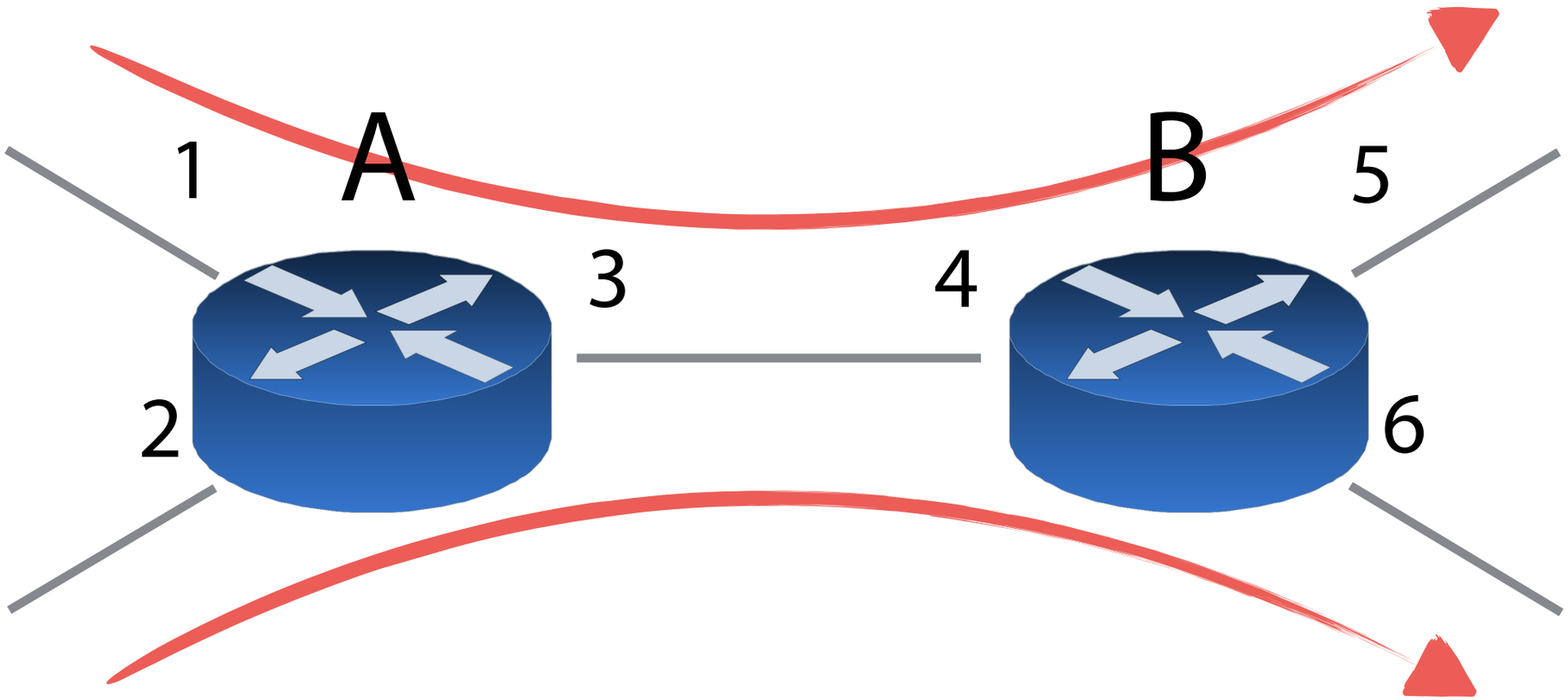
pol

Global Program



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

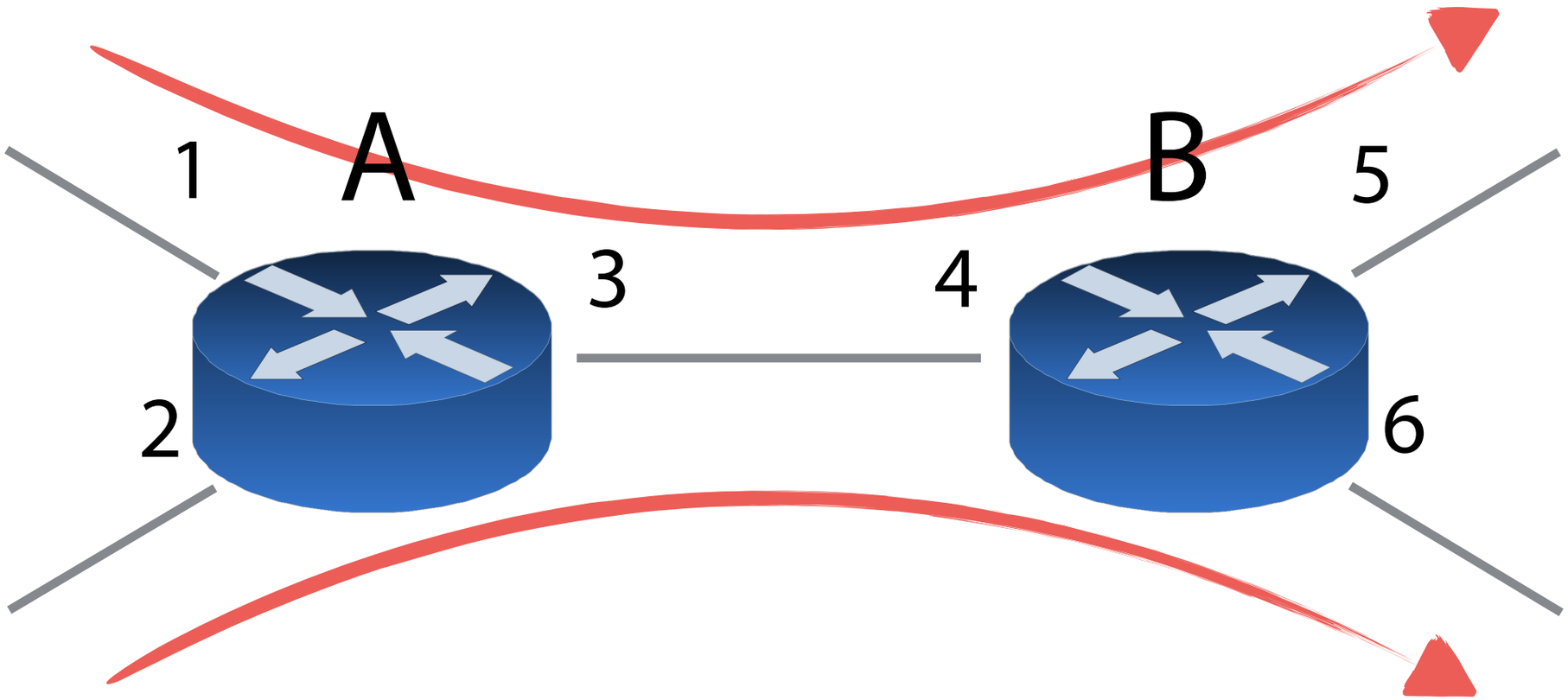
Global Program



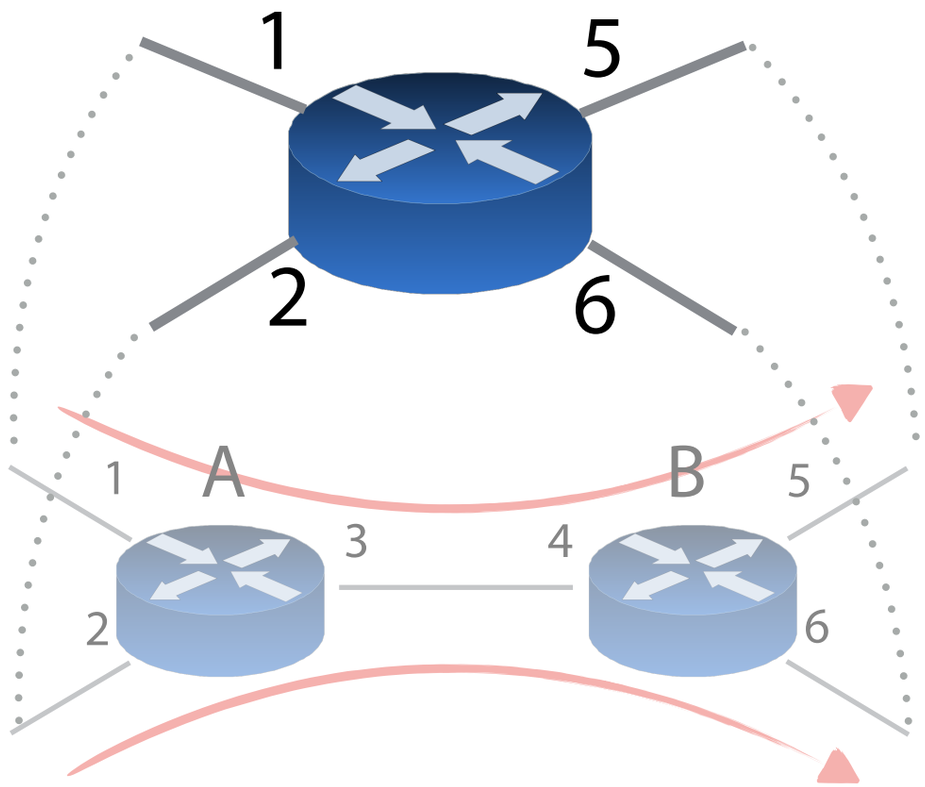
port=1; **A→B;** port**:=**5
+
port=2; **A→B;** port**:=**6

simple and elegant!

Virtual Program

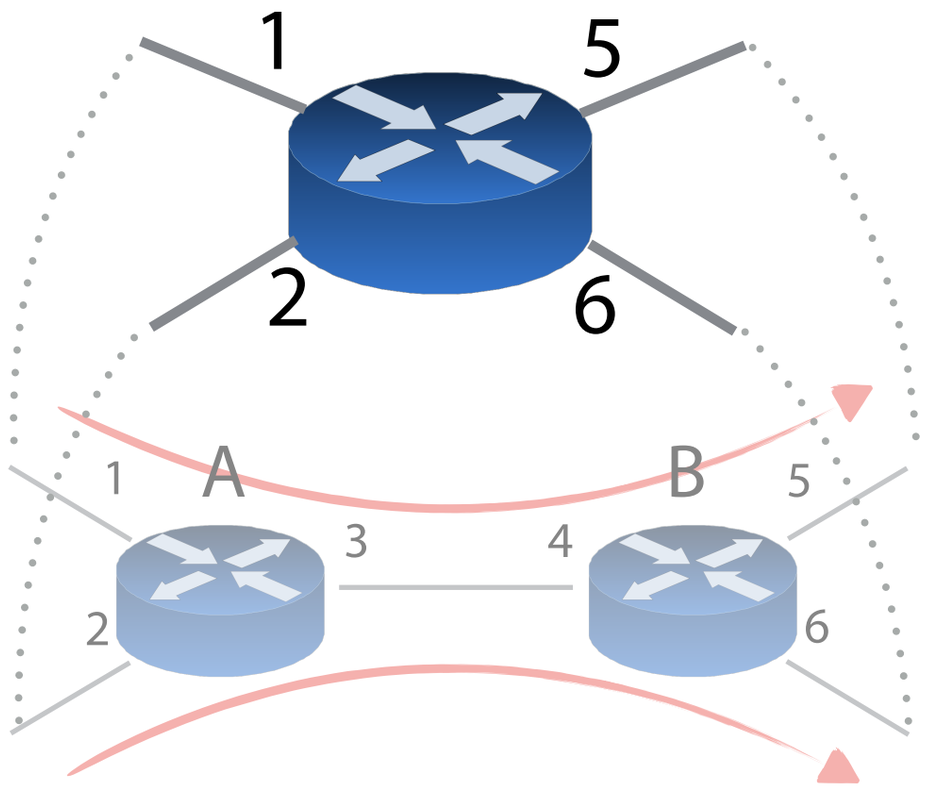


Virtual Program



virtual "big switch"

Virtual Program



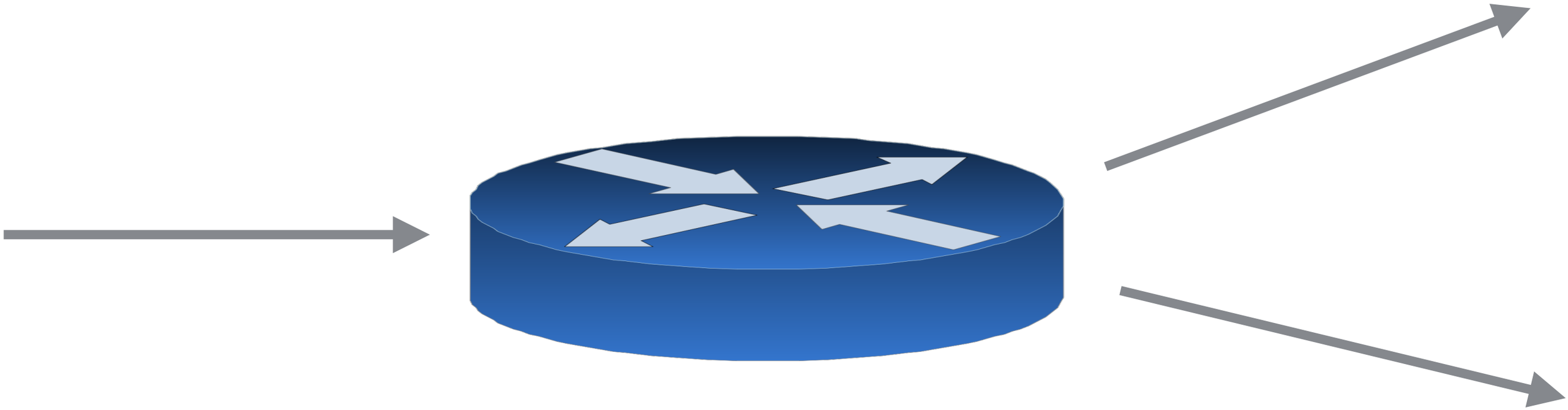
virtual "big switch"

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

even simpler!

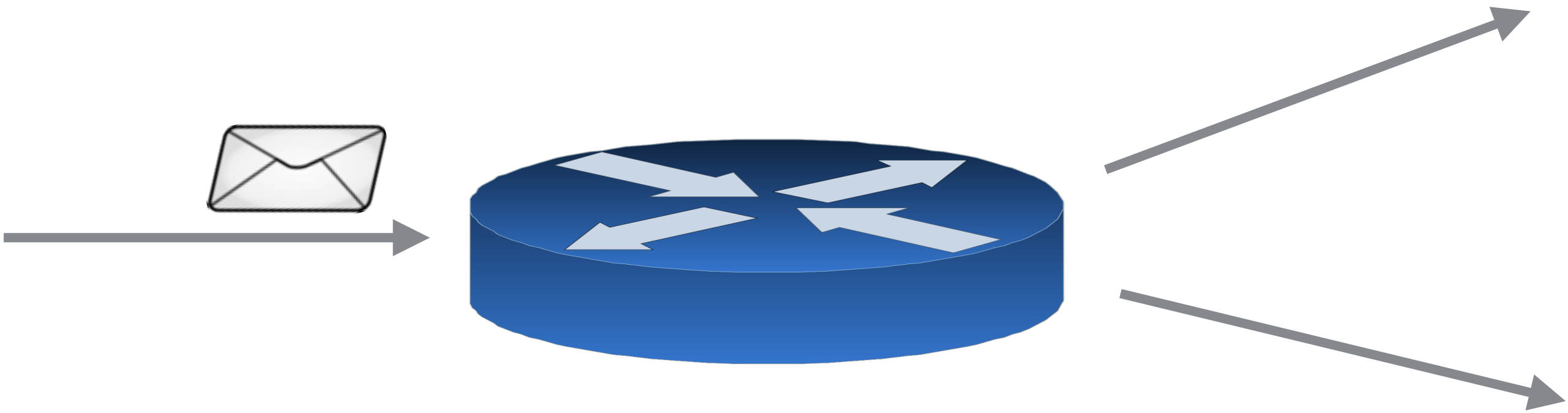
A Functional Forwarding Model

Model



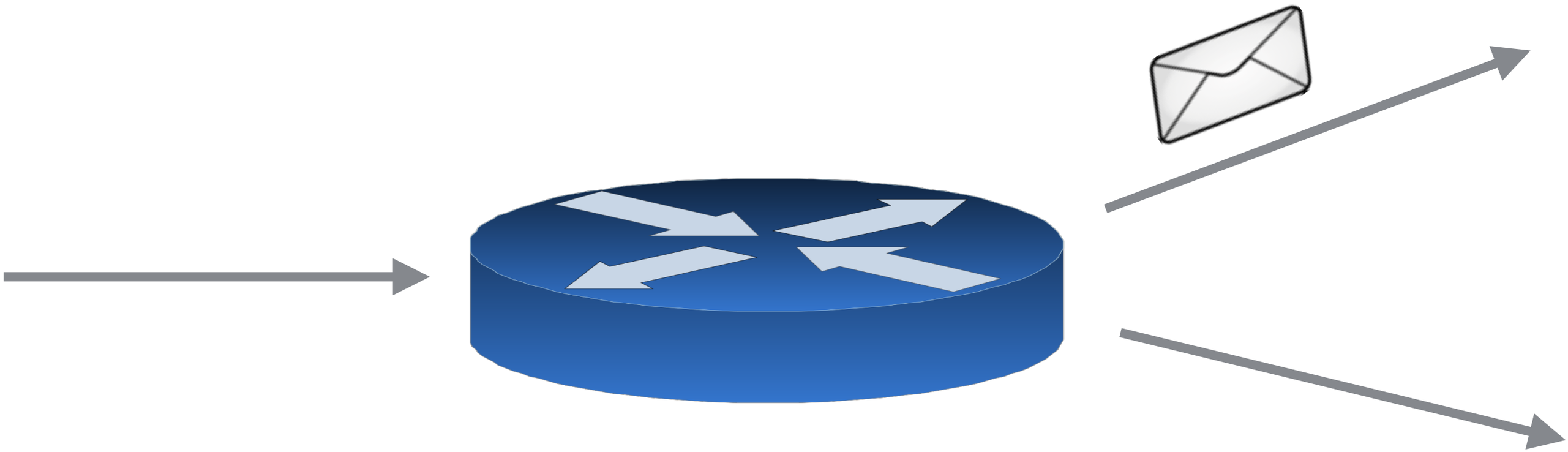
Packets → Packets

Model



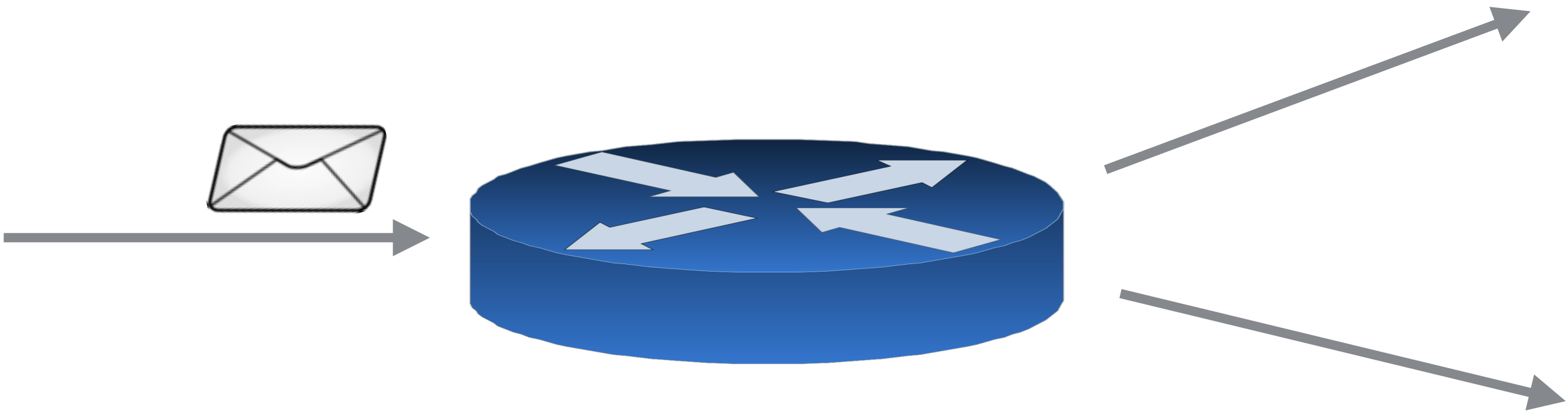
Packets → Packets

Model



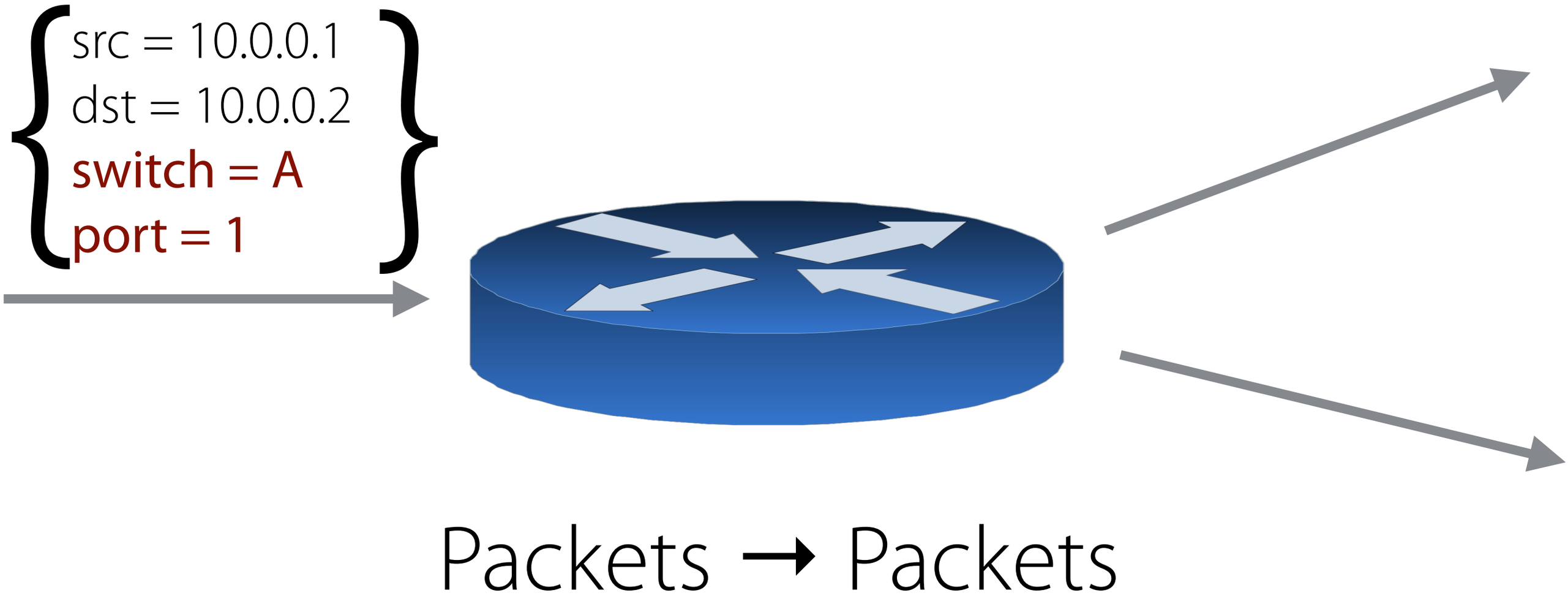
Packets → Packets

Model



Packets → Packets

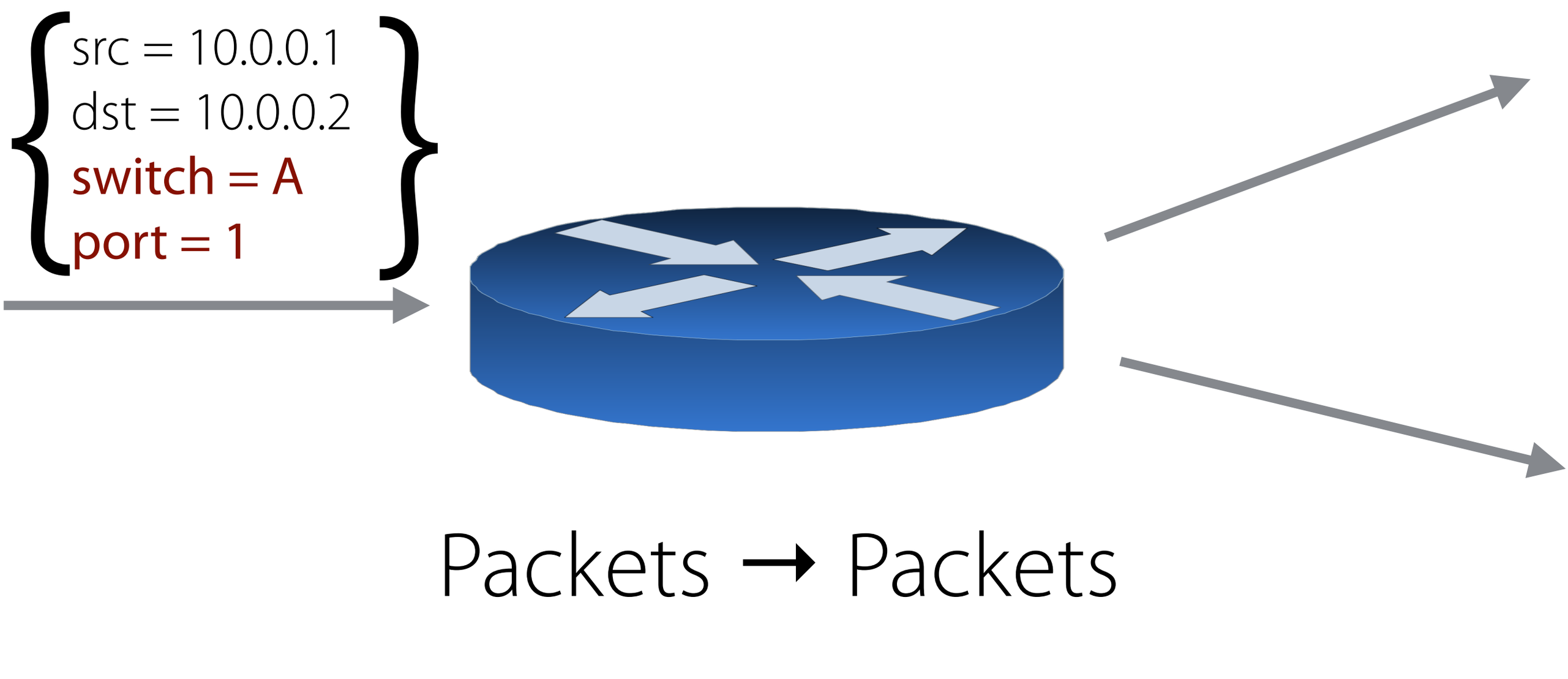
Model



Packets → Packets

Packet Manipulation

- packet filtering
- packet modification



Packets → Packets

Packet Manipulation

- **packet filtering**
- packet modification

Packets → Packets

Packet Manipulation

- **packet filtering**
- packet modification

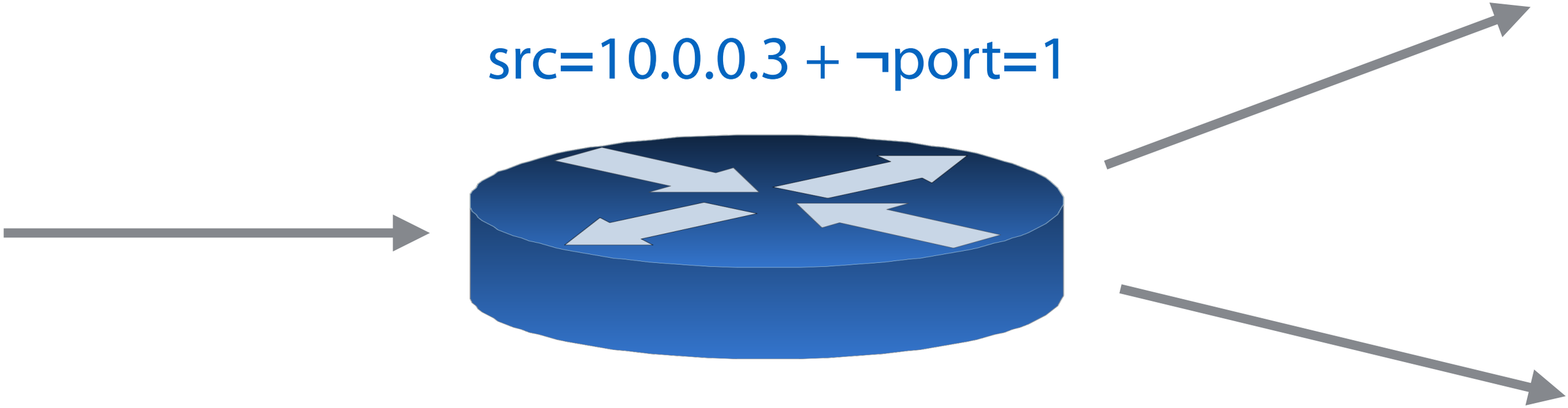
{ src = 10.0.0.1
dst = 10.0.0.2
switch = A
port = 1 }

src=10.0.0.3 + ¬port=1

Packets → Packets

Packet Manipulation

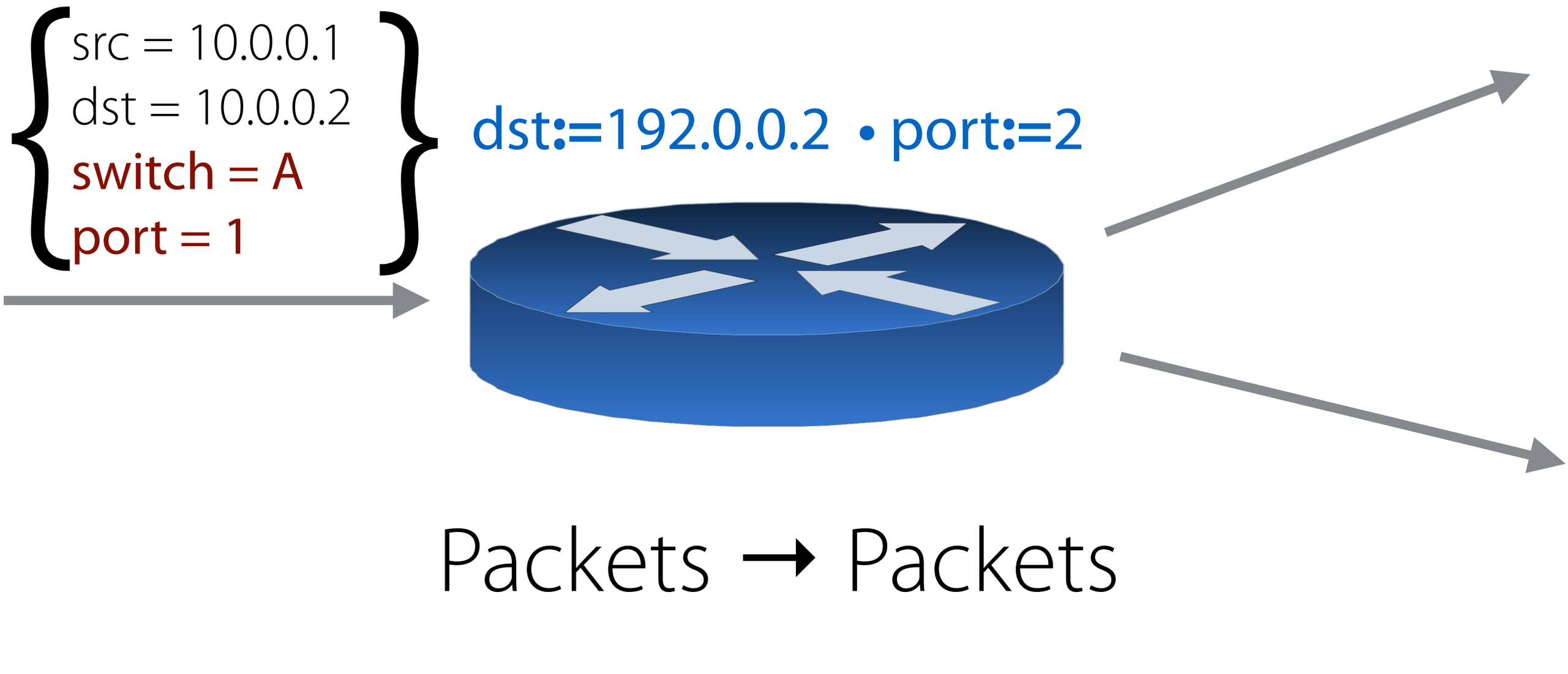
- **packet filtering**
- packet modification



Packets → Packets

Packet Manipulation

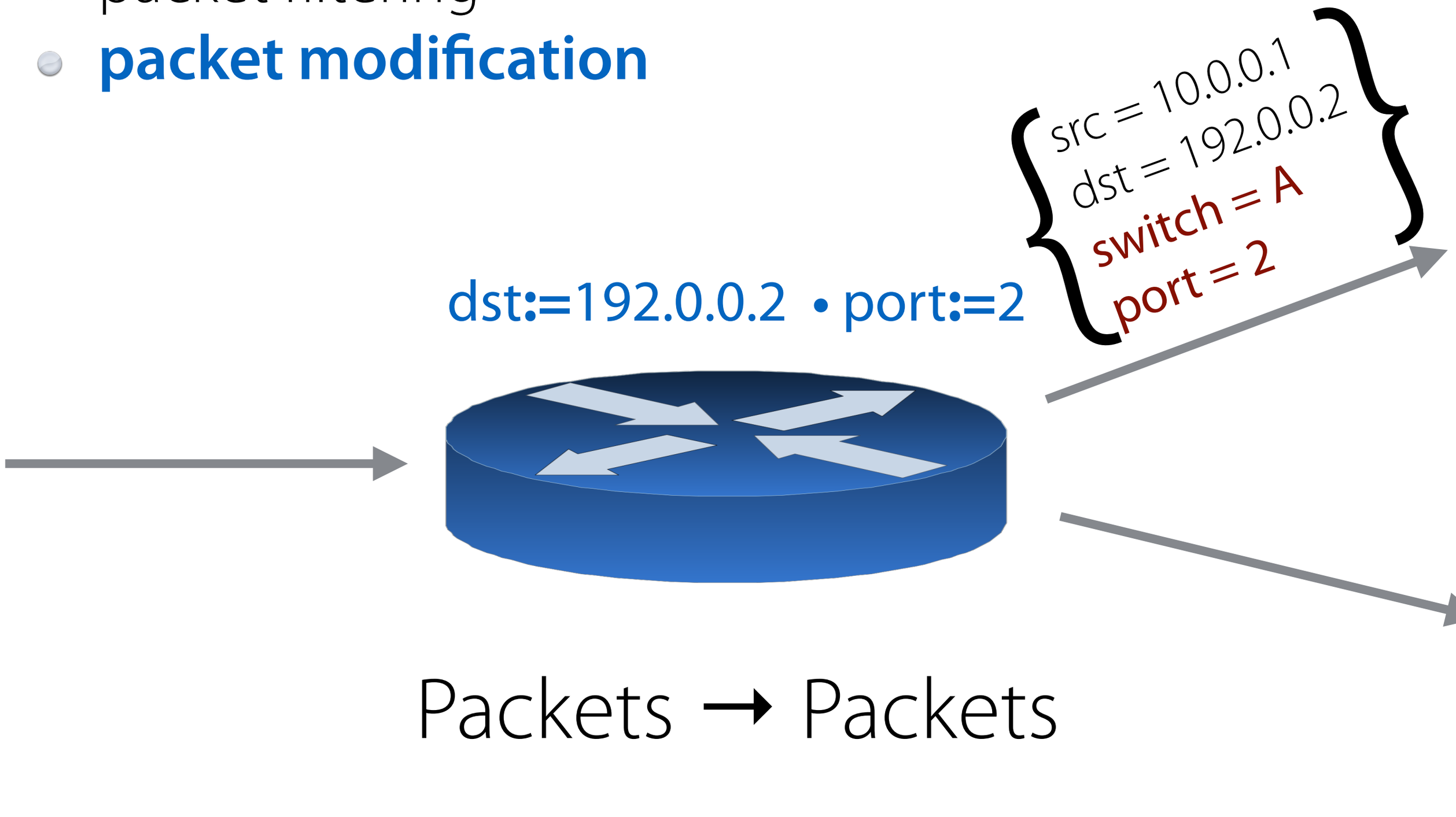
- packet filtering
- **packet modification**



Packets → Packets

Packet Manipulation

- packet filtering
- **packet modification**



Packets → Packets

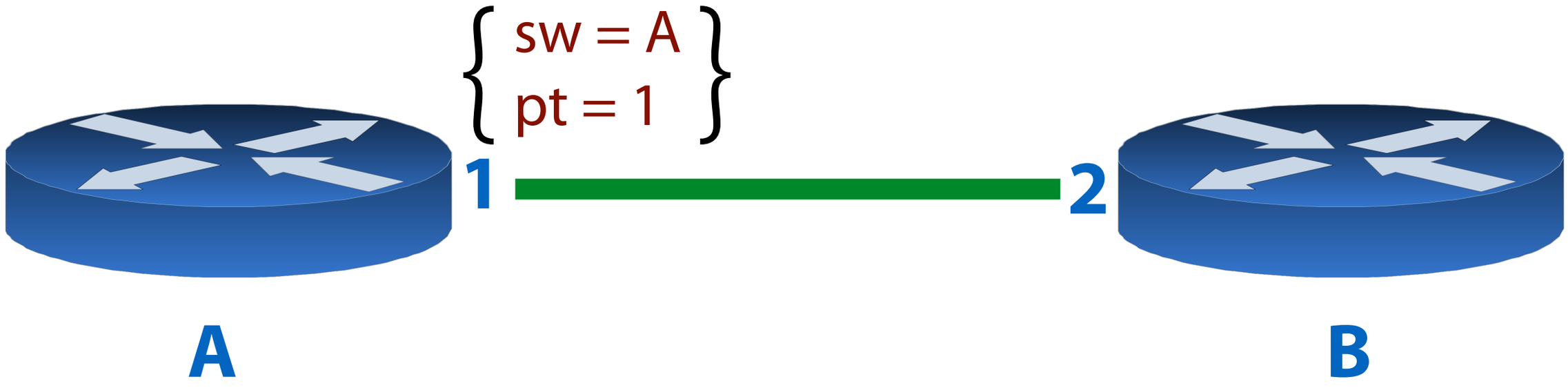
Topology

Topology

switch=A • pt=1 • switch:=B • pt:=2

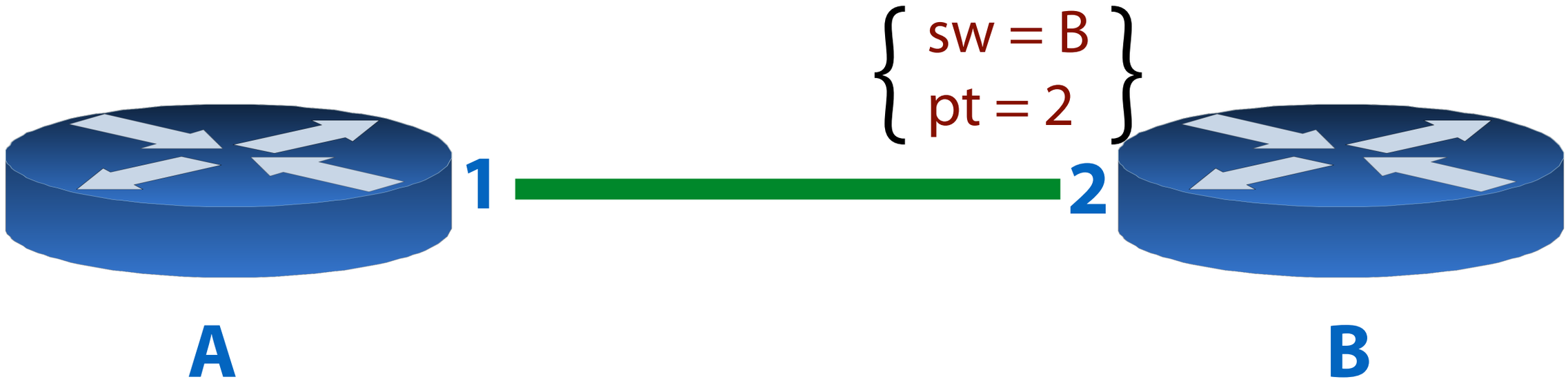
Topology

switch=A • pt=1 • switch:=B • pt:=2



Topology

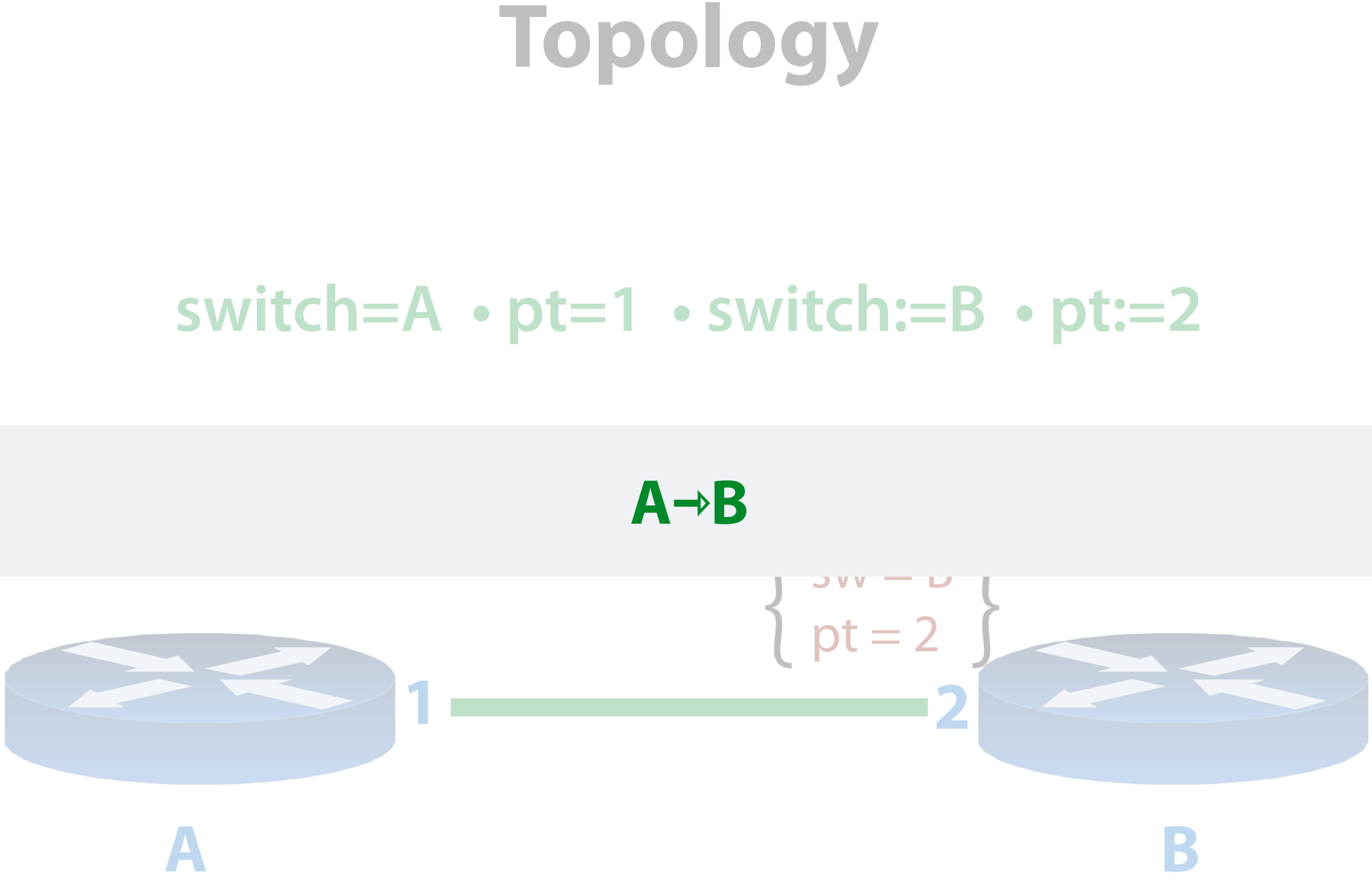
switch=A • pt=1 • switch:=B • pt:=2



Topology

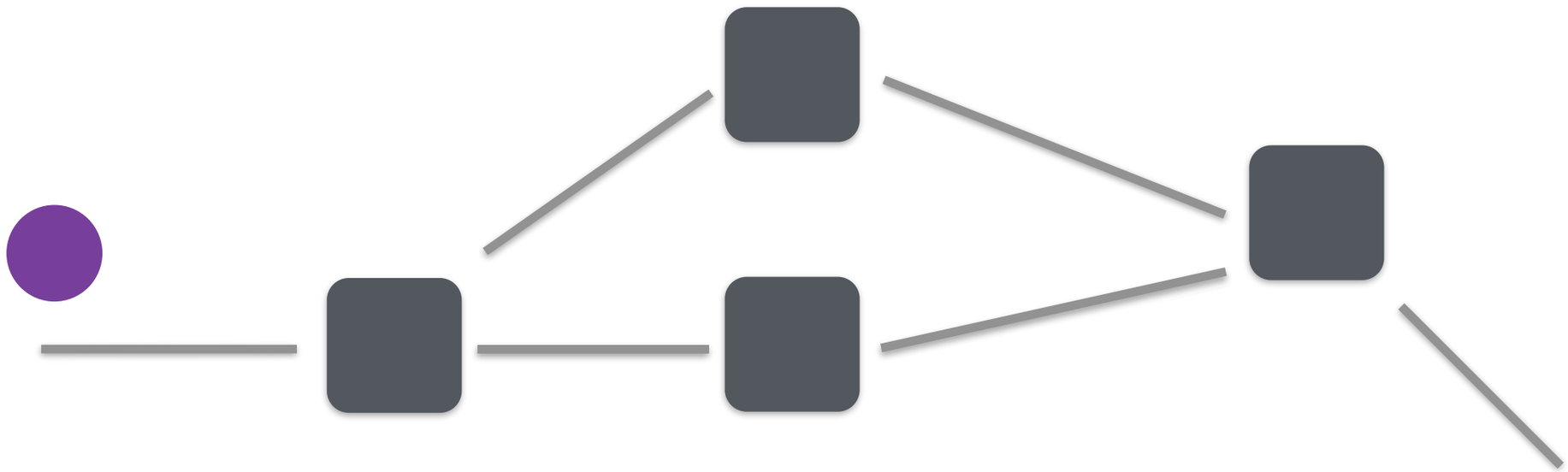
switch=A • pt=1 • switch:=B • pt:=2

A→B



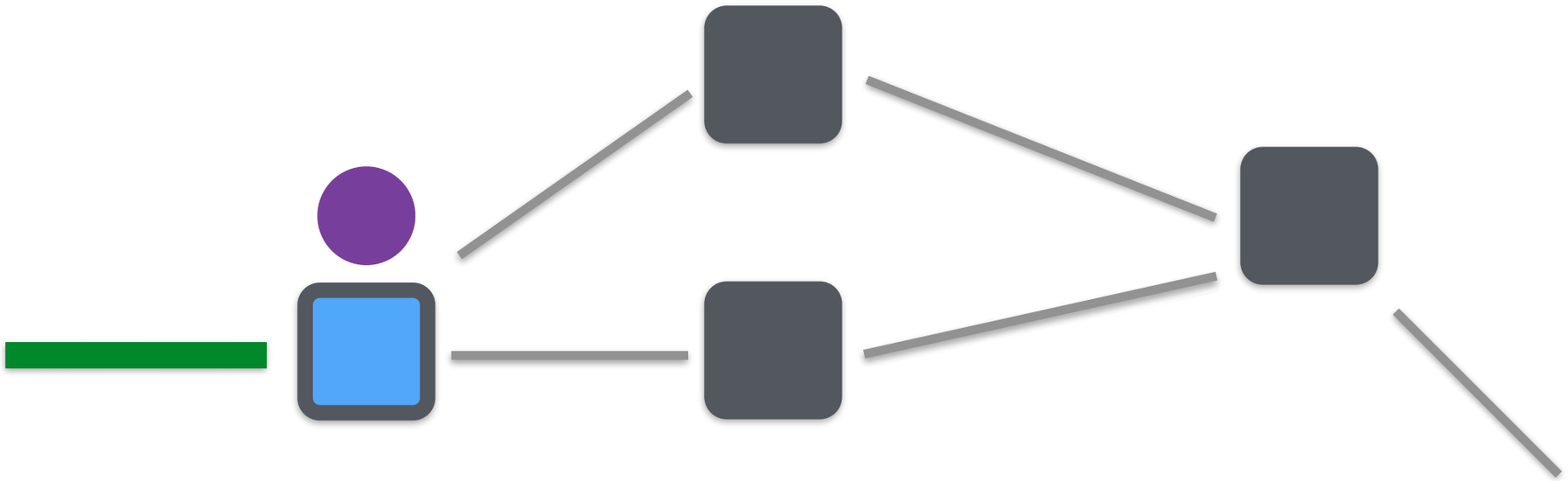
Iteration

(topology • switch)*



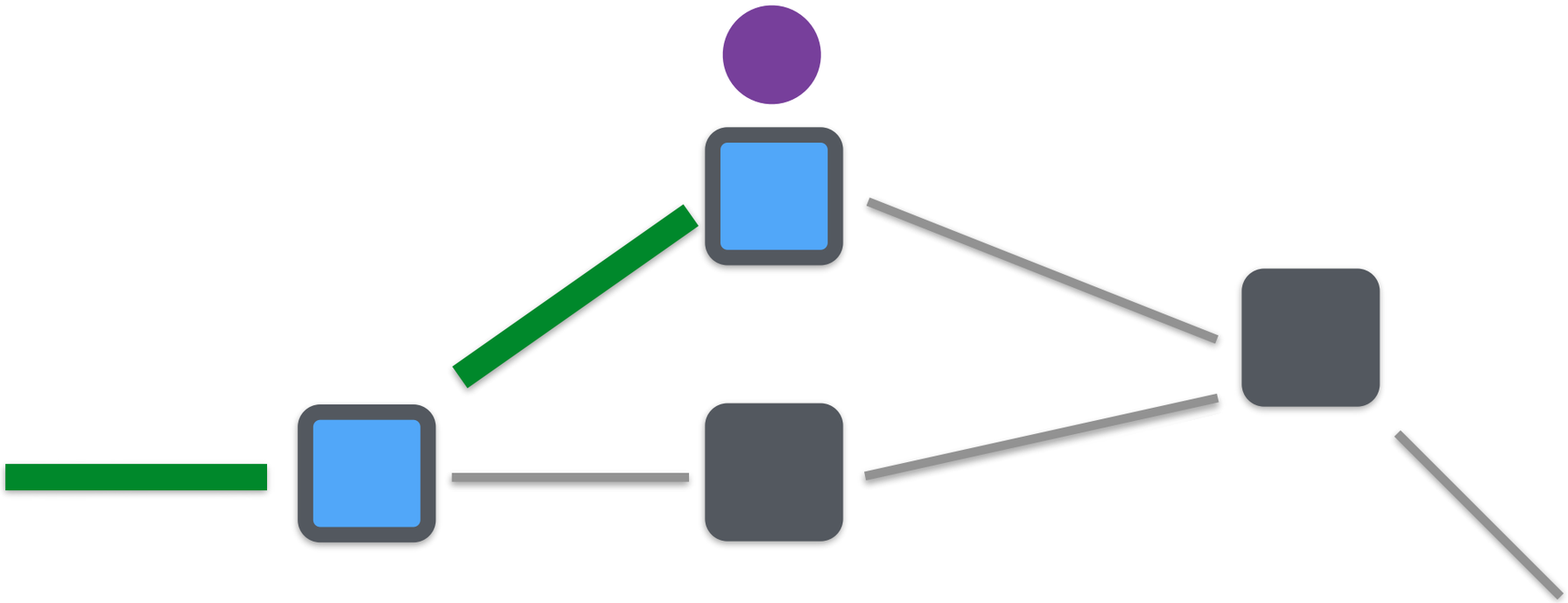
Iteration

(topology • switch)*



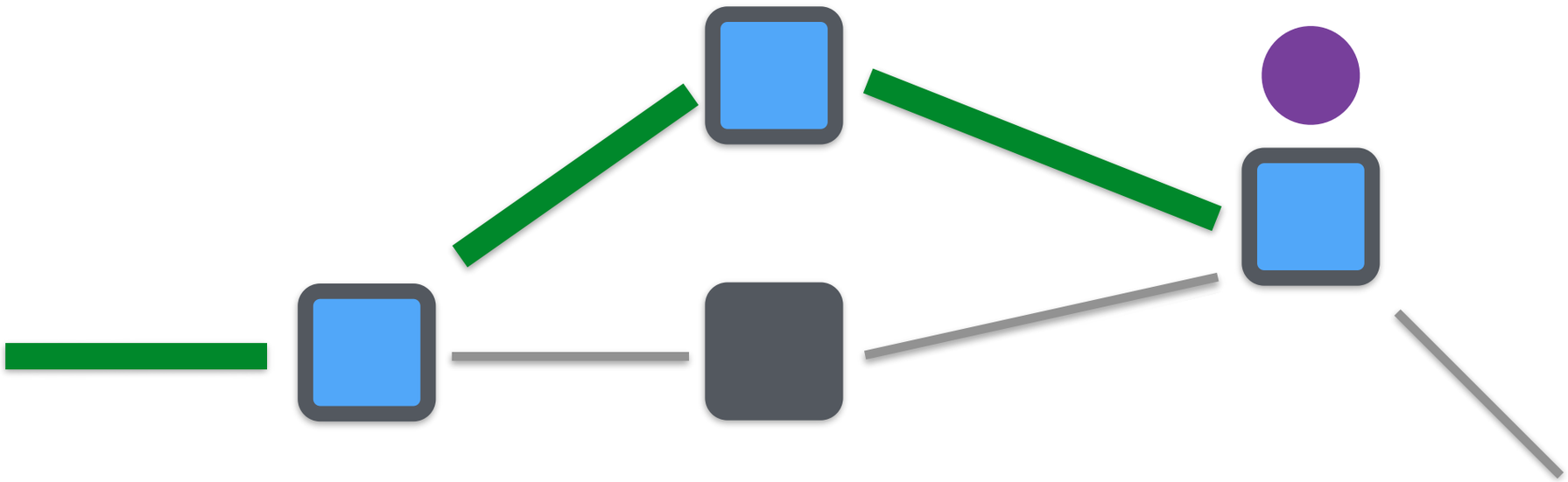
Iteration

(topology • switch)*



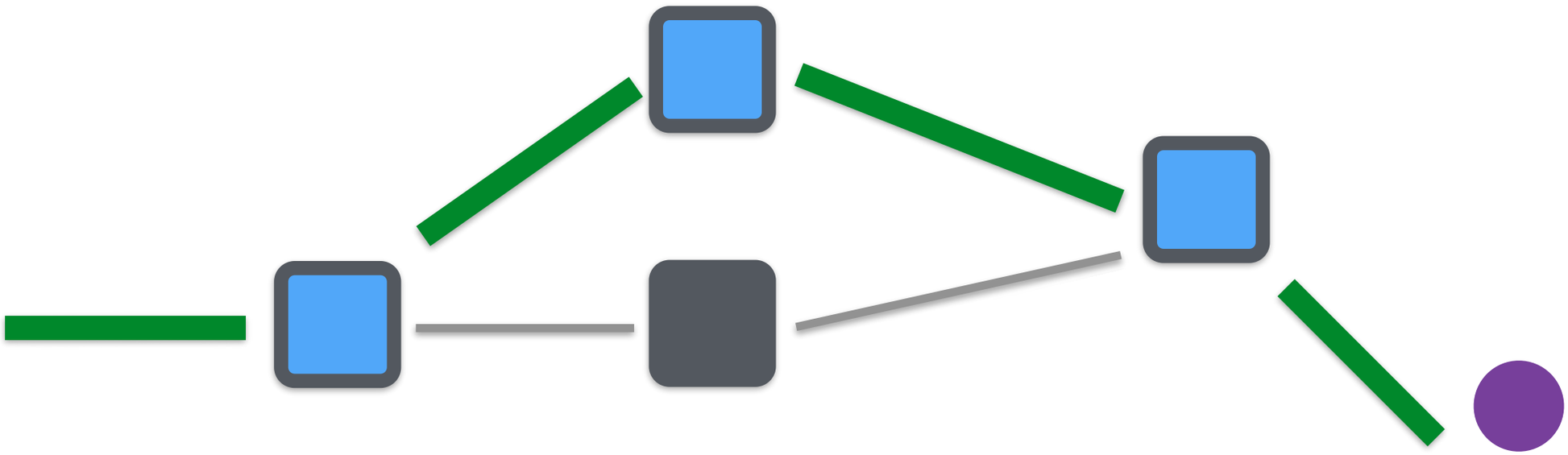
Iteration

(topology • switch)*



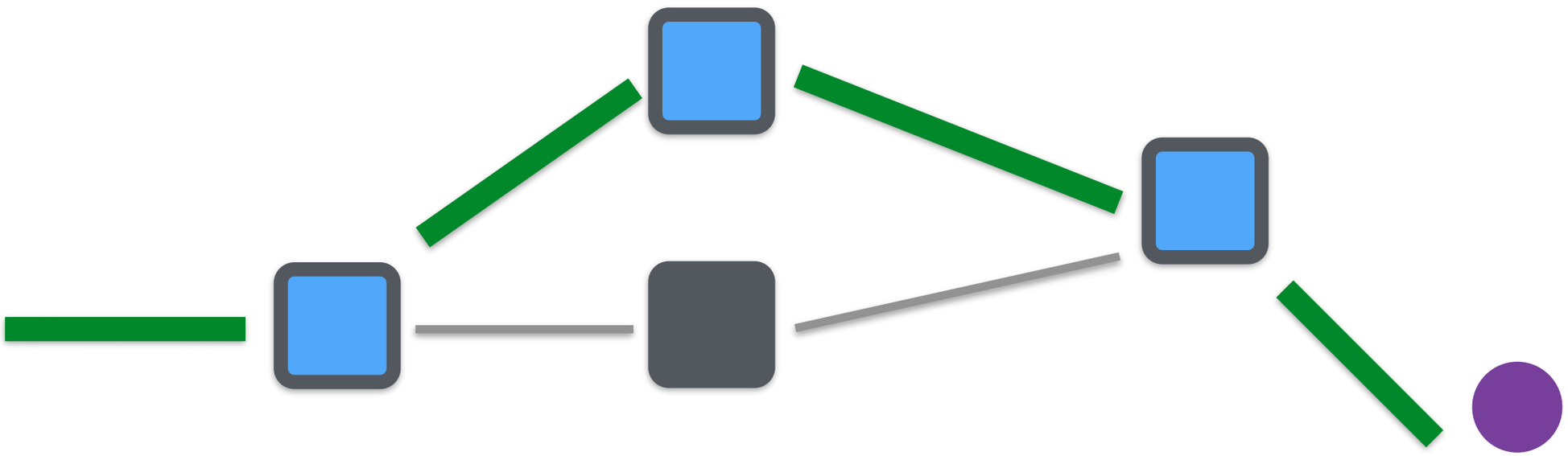
Iteration

(topology • switch)*



Iteration

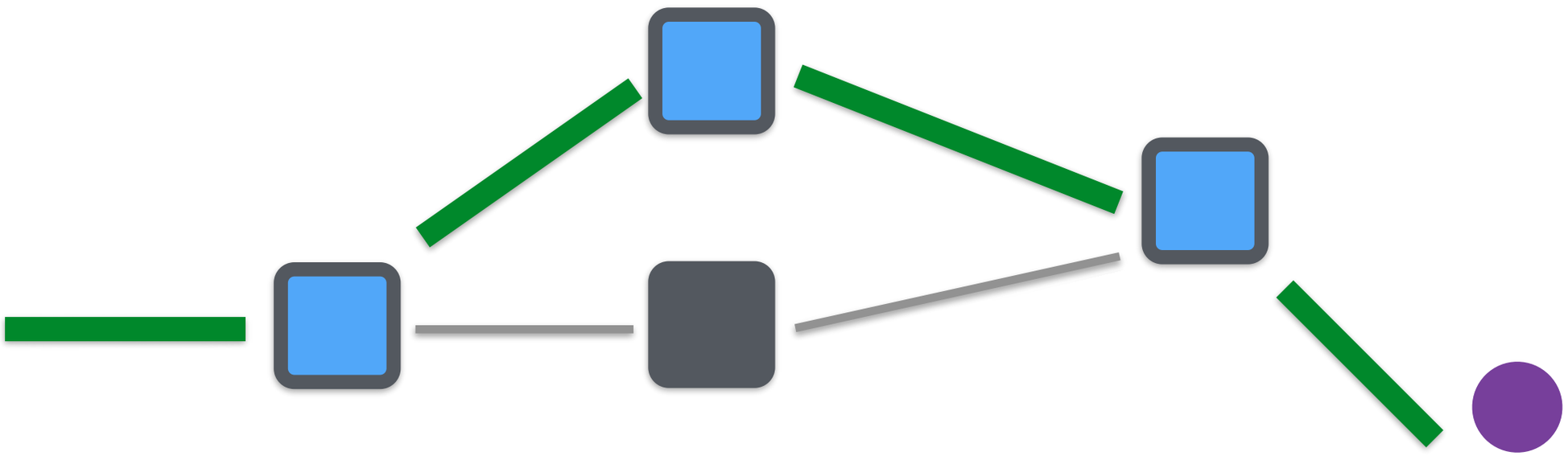
(topology • switch)*



Packets → Packets

Iteration

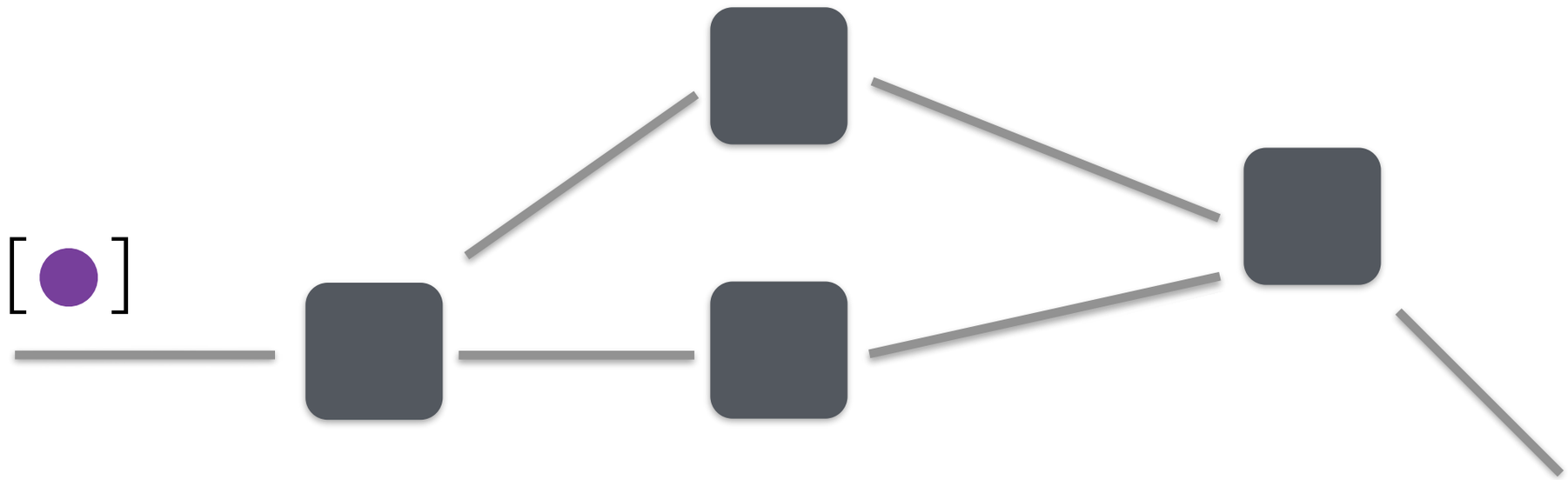
(topology • switch)*



~~Packets~~ ~~Packets~~

Iteration

(topology • switch)*

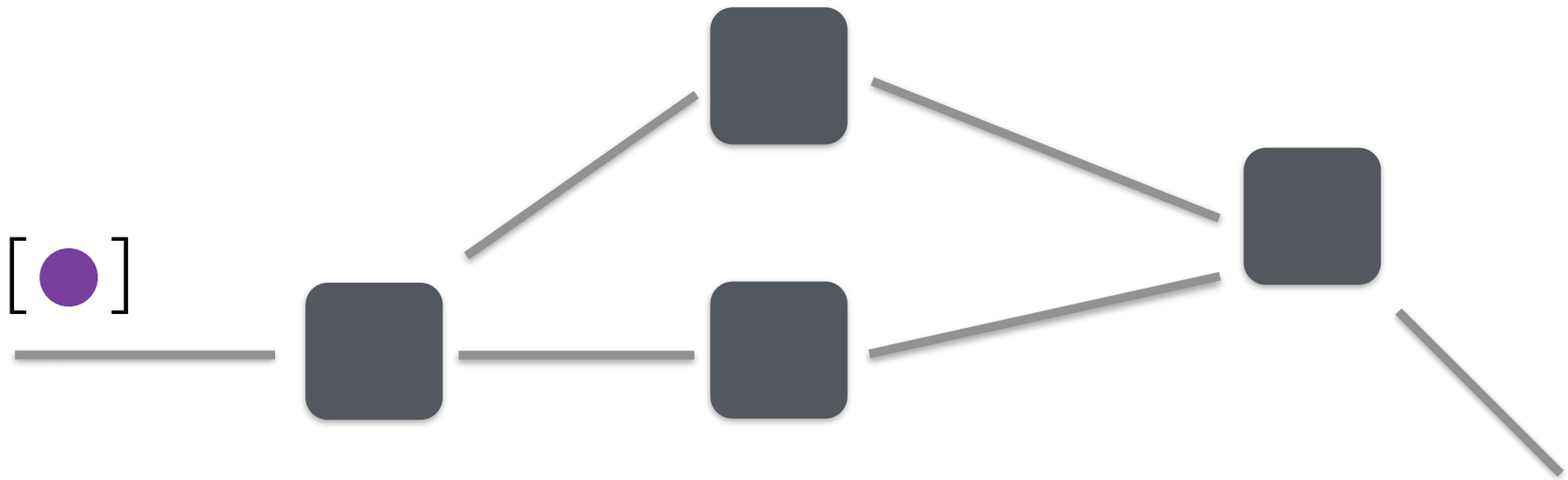


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*

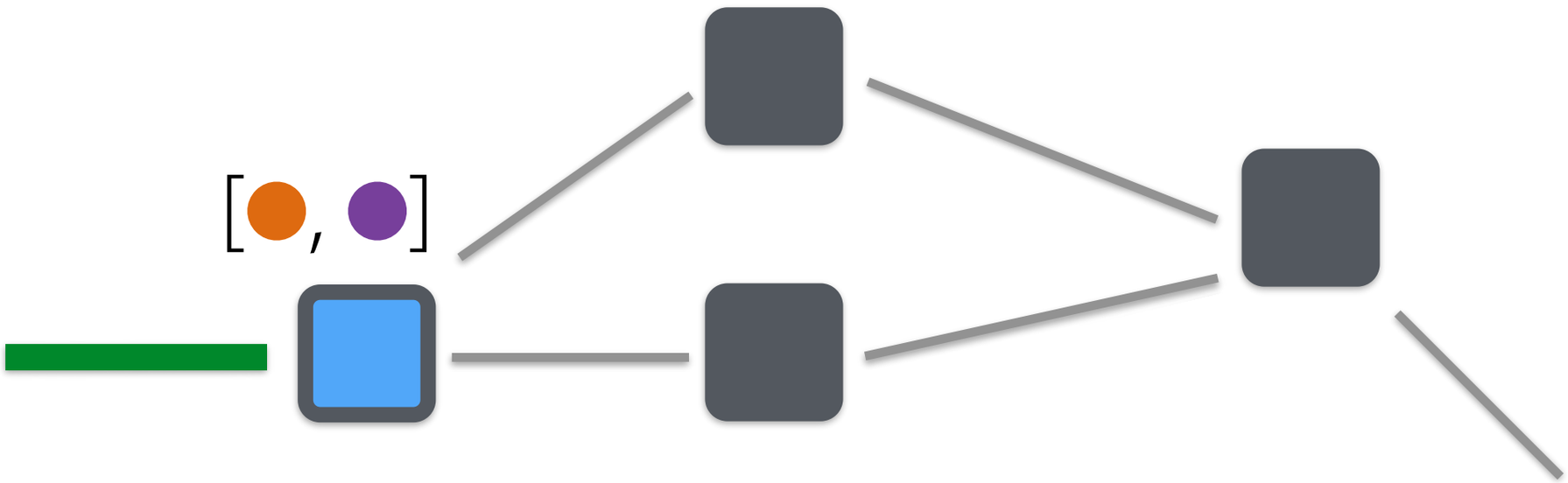


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

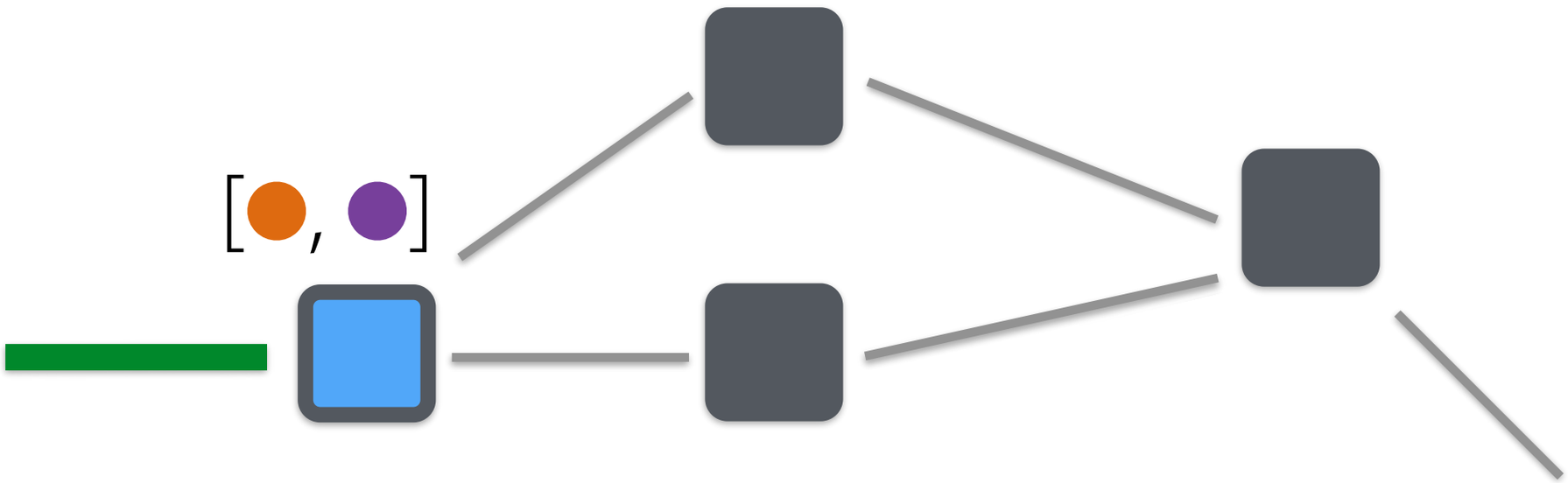


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*

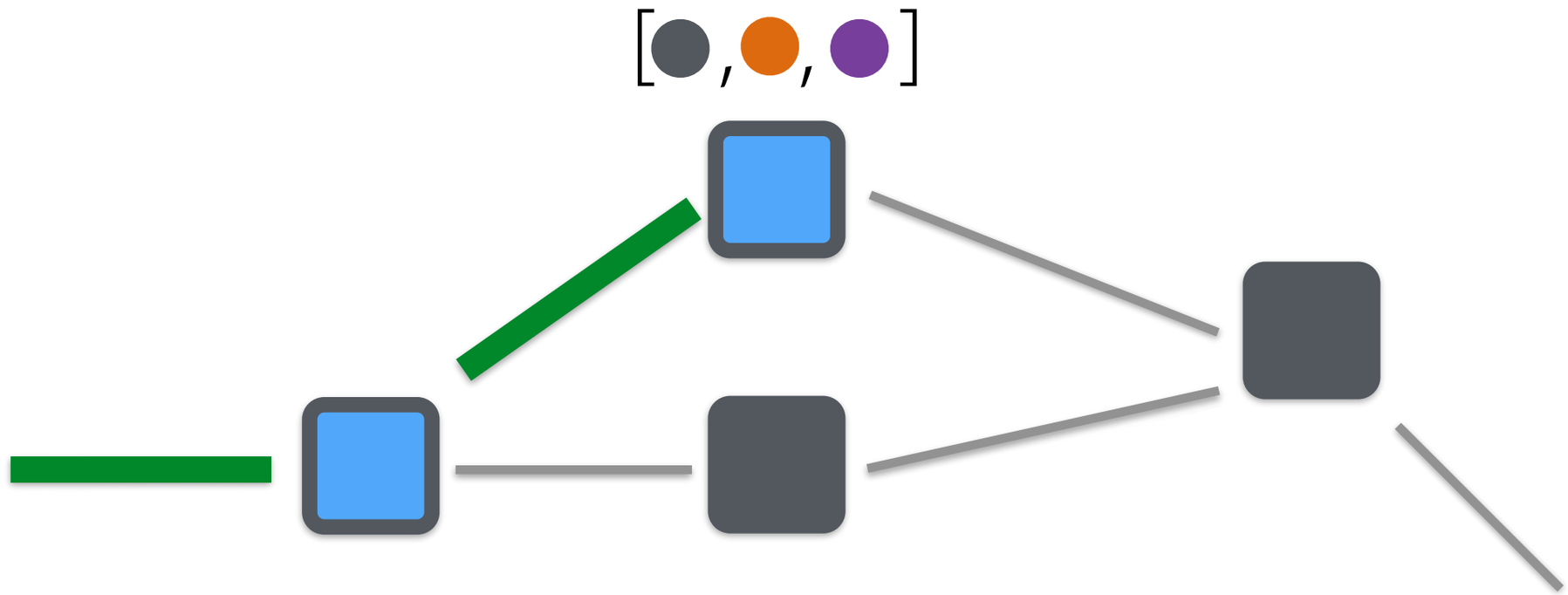


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

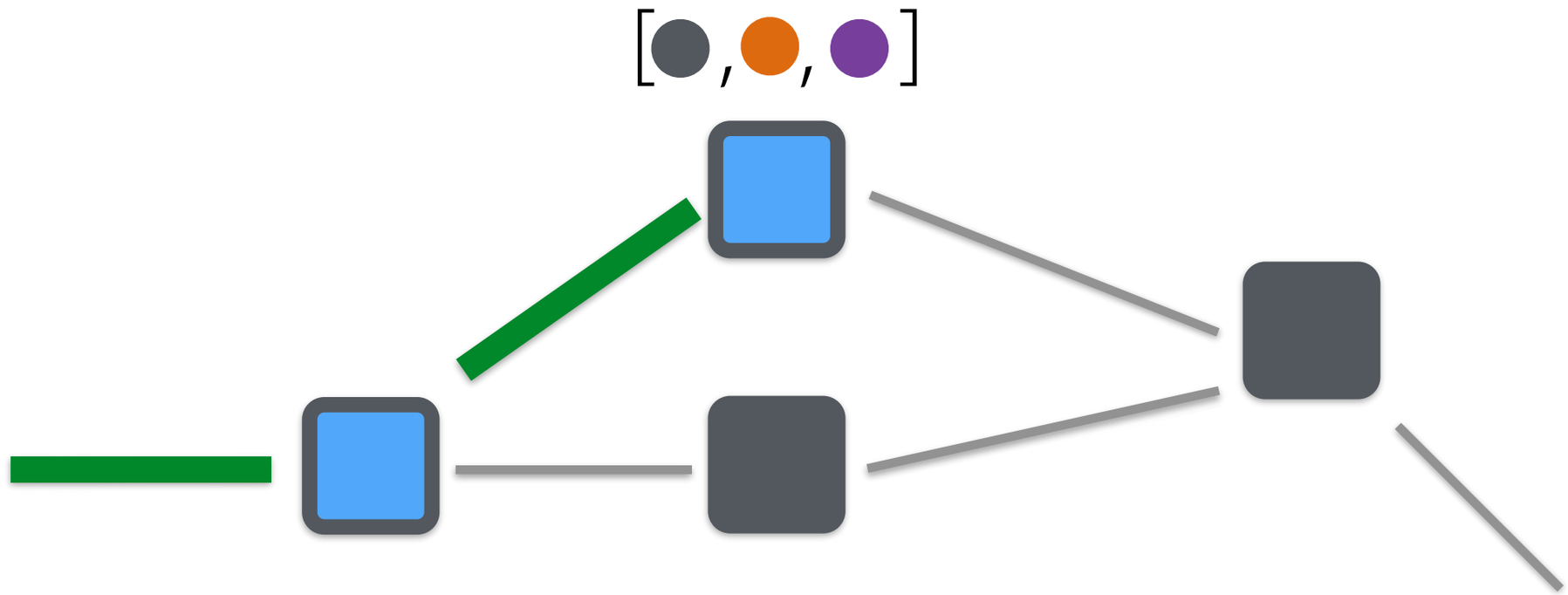


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*

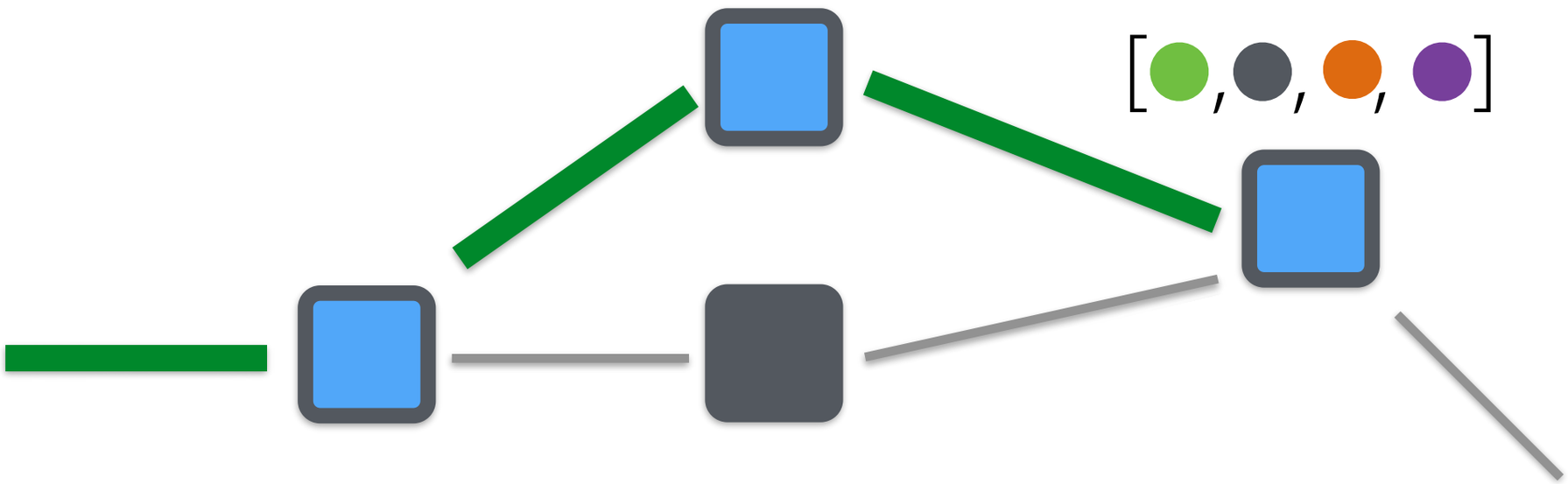


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

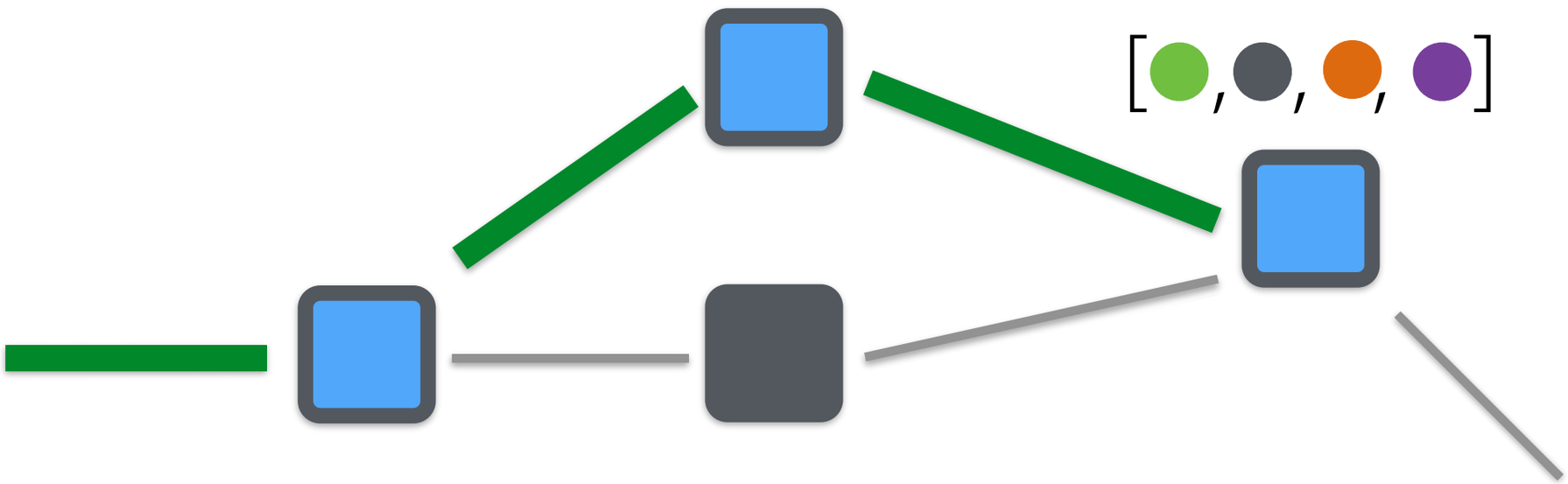


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*

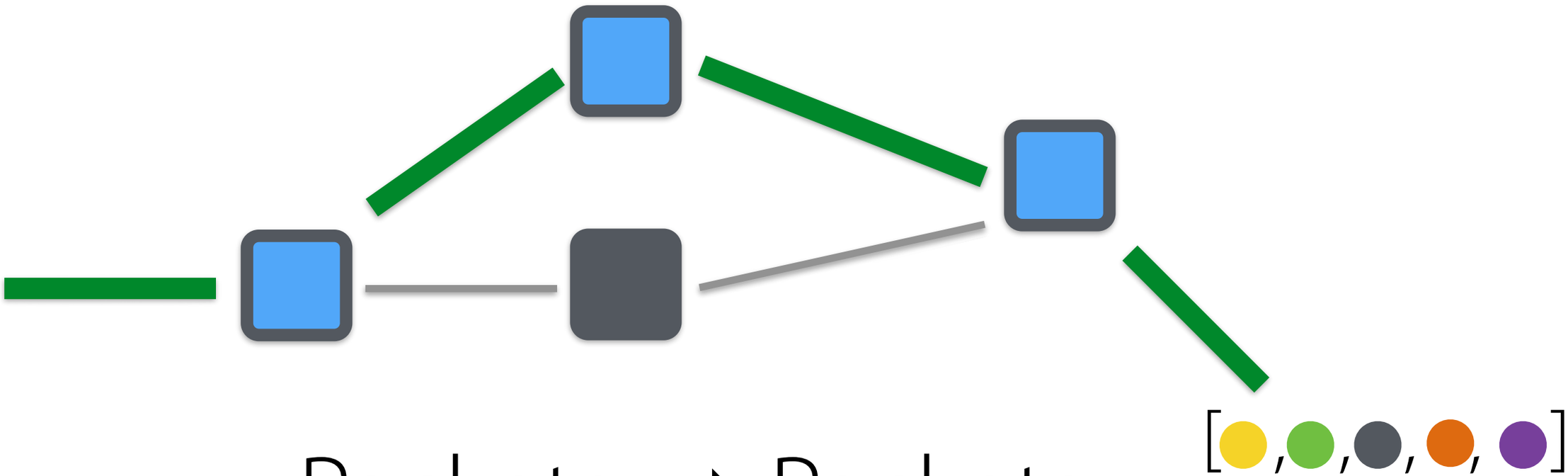


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

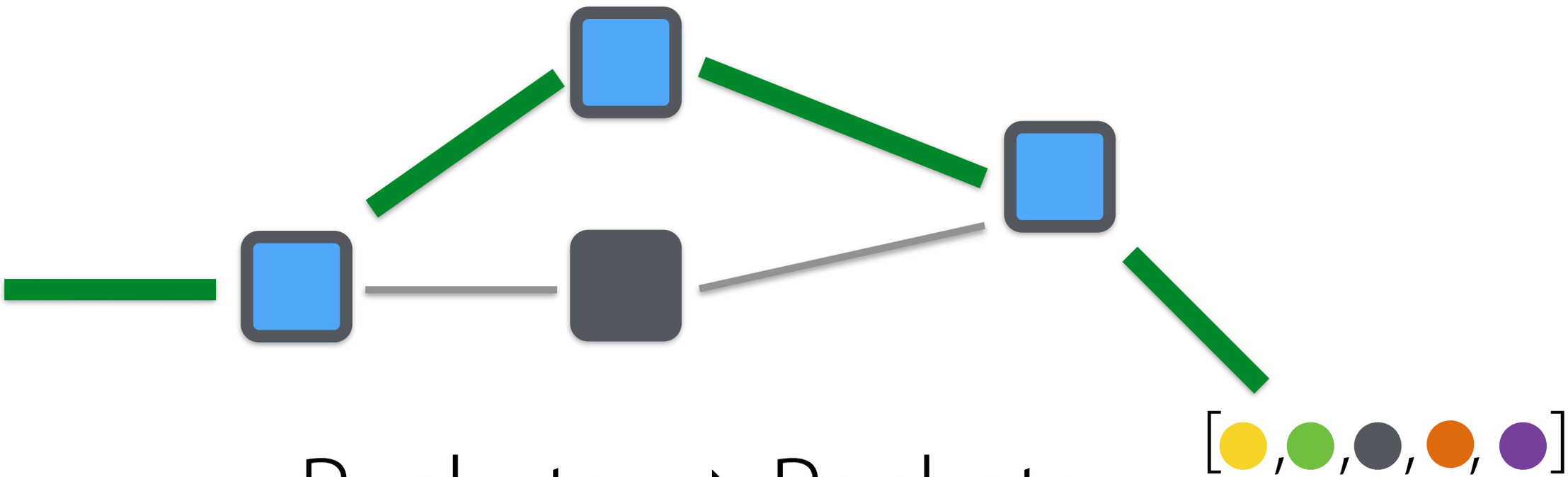


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*

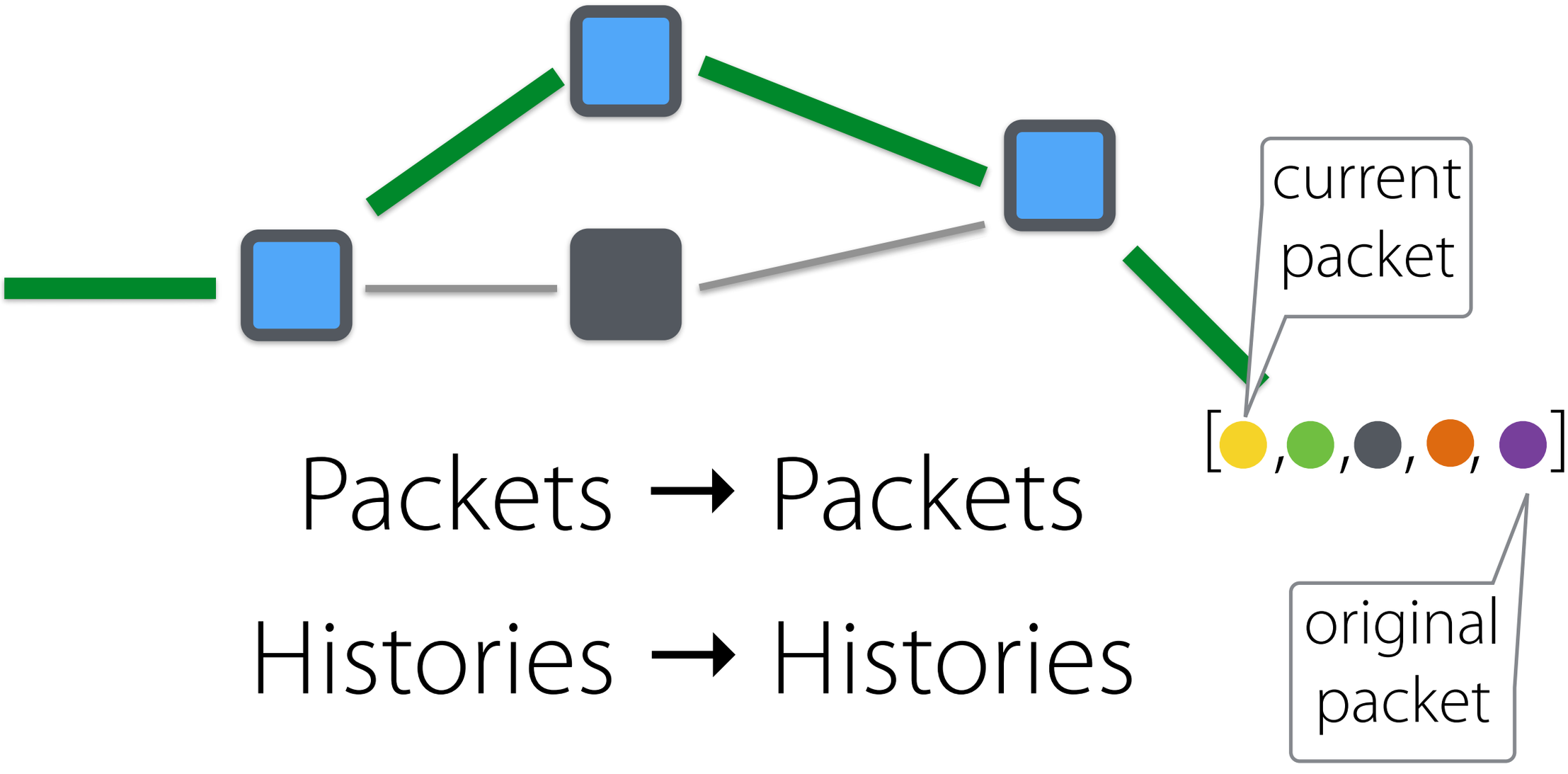


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

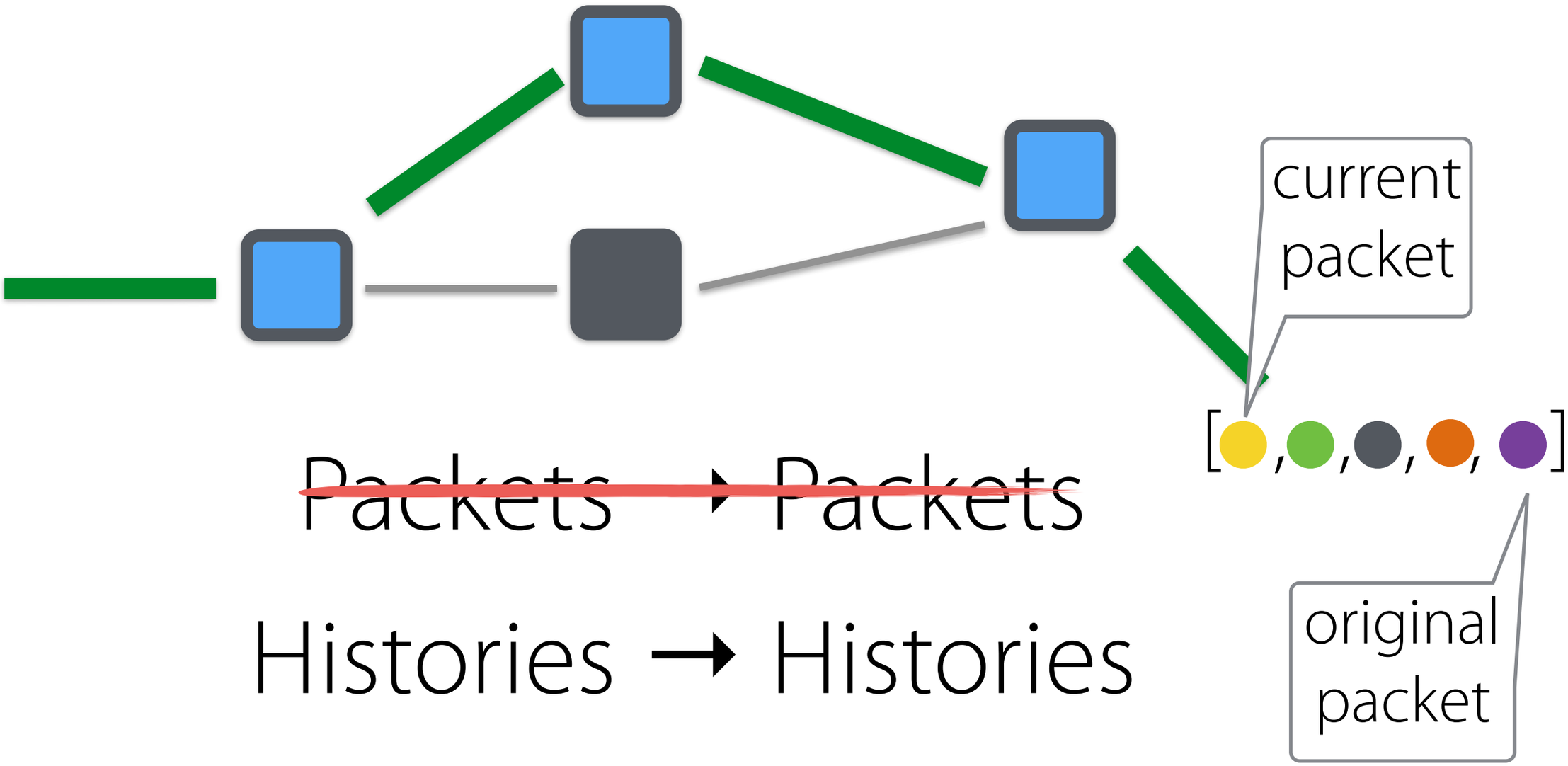


Packets → Packets

Histories → Histories

Iteration

(topology • switch)*



~~Packets → Packets~~

Histories → Histories

NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| f **:=** n
| pol_1 **+** pol_2
| pol_1 • pol_2
| !pol
| pol*
| **dup**

NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| f **:=** n
| pol_1 **+** pol_2
| pol_1 • pol_2
| !pol
| pol*
| **dup**

Boolean Predicates

+

Regular Expressions

+

Packet Primitives

Negation may only be applied to B*oolean predicates:*
true, **false**, f **=** n, closed under +, •, and **!**

NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| f **:=** n
| pol_1 **+** pol_2
| pol_1 • pol_2
| !pol
| pol*
| **dup**

Boolean Predicates + Regular Expressions + Packet Primitives } KAT

Negation may only be applied to B*oolean predicates:*
true, **false**, f **=** n, closed under +, •, and **!**

NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| f **:=** n
| pol_1 **+** pol_2
| pol_1 • pol_2
| !pol
| pol*
| **dup**

Boolean Predicates + Regular Expressions + Packet Primitives

KAT

NetKAT

Negation may only be applied to B*oolean predicates:*
true, **false**, f **=** n, closed under +, •, and **!**

NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| pol **+** pol
| pol **•** pol
| **!**pol
| **dup**

Boolean Predicates

Primitives

if b **then** p_1 **else** p_2 ≜ (b • p_1) **+** (**!**b • p_2)

while b **do** p ≜ (b • p)* • **!**b

S➙S′ ≜ sw**=**S • **dup** • sw**:=**S′ • **dup**

Negation may only be applied to B*oolean predicates:*
true, **false**, f **=** n, closed under +, •, and **!**

Related: Kat + B! [LICS '14]

p ::= **false**
| **true**
| **t**?
| $\overline{\mathbf{t}}$?
| **t**!
| $\overline{\mathbf{t}}$!
| p_1 + p_2
| $p_1 \bullet p_2$
| !p
| p^*

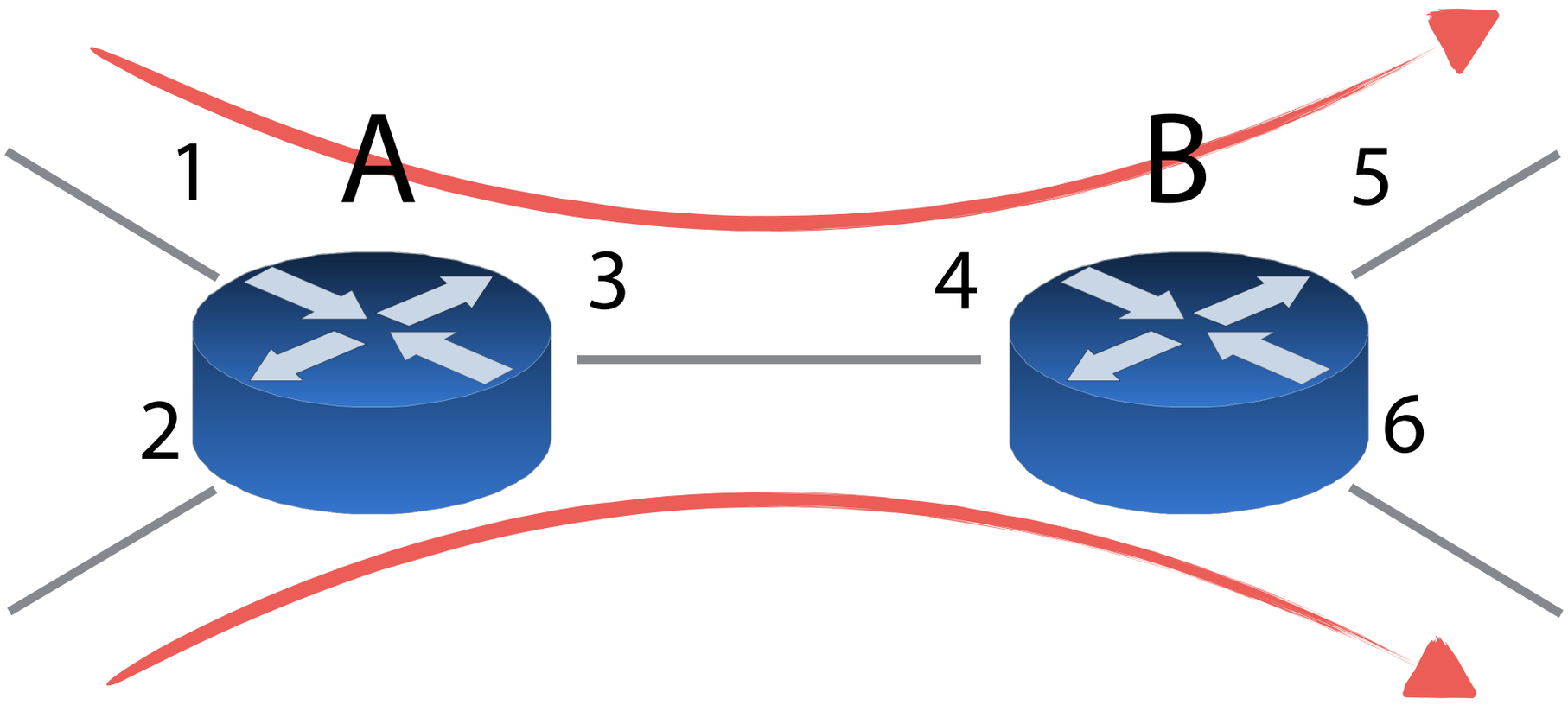
Idea: extend KAT with a finite number of mutable boolean state variables

Formally:

- Fix a finite set of tests $T = \{ t_1, \ldots t_n \}$
- Add primitive tests, both positive (**t**?) and negative ($\overline{\mathbf{t}}$?)
- Add primitive actions, both positive (**t!**) and negative ($\overline{\mathbf{t}}$!)

15-minute Break

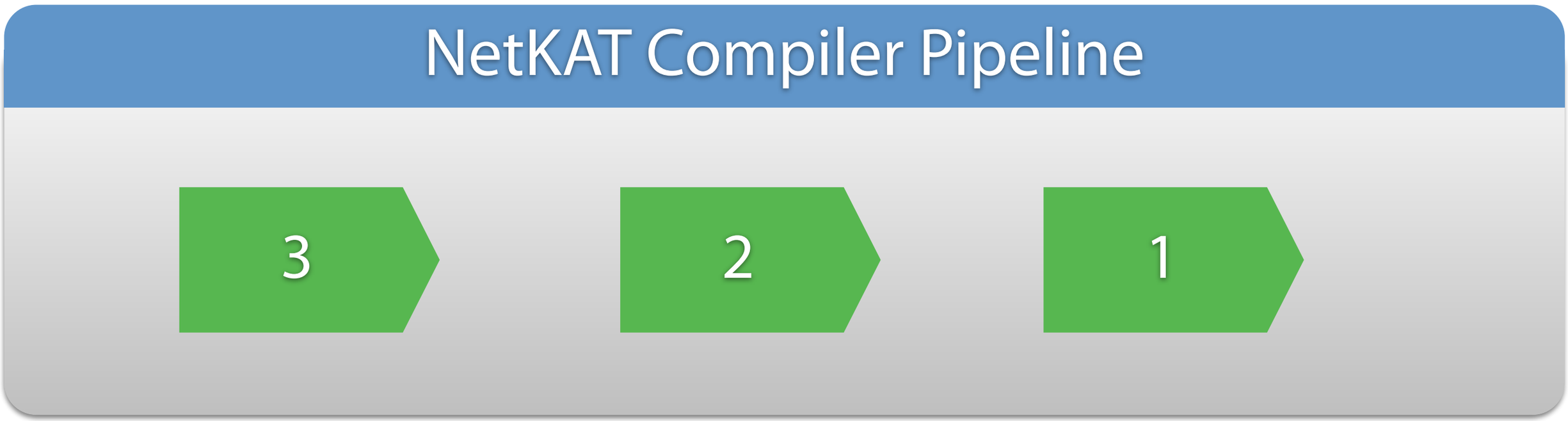
Question: How can we compile global programs like this?



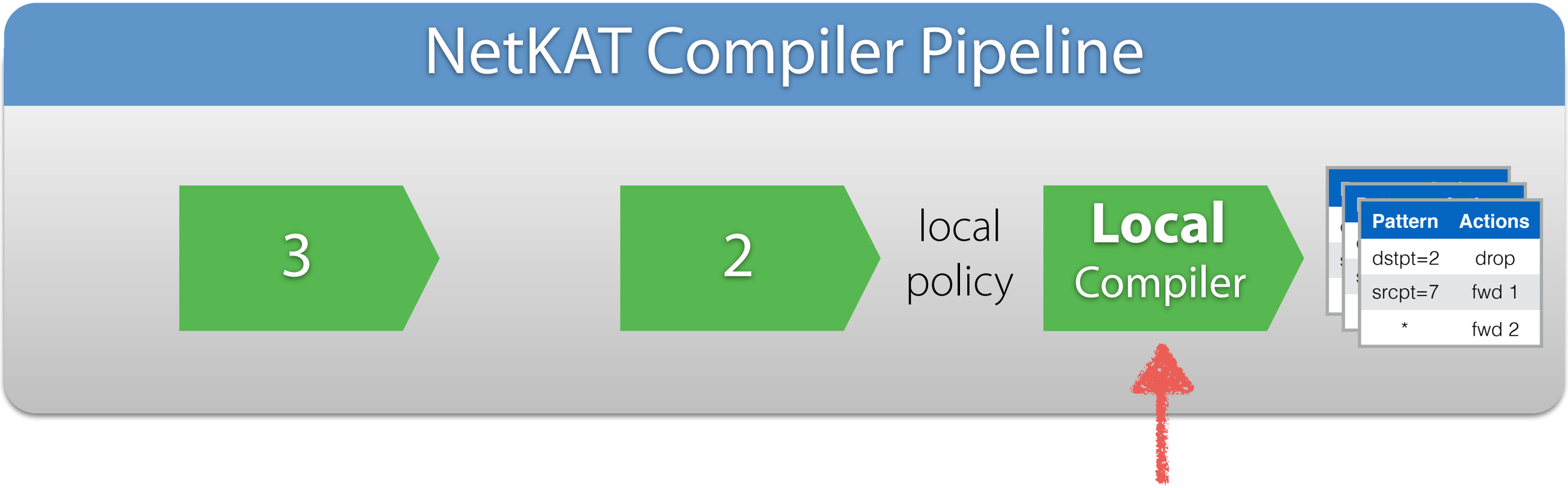
port=1**; A→B;** port**:=**5
+
port=2**; A→B;** port**:=**6

Compilation

Overview

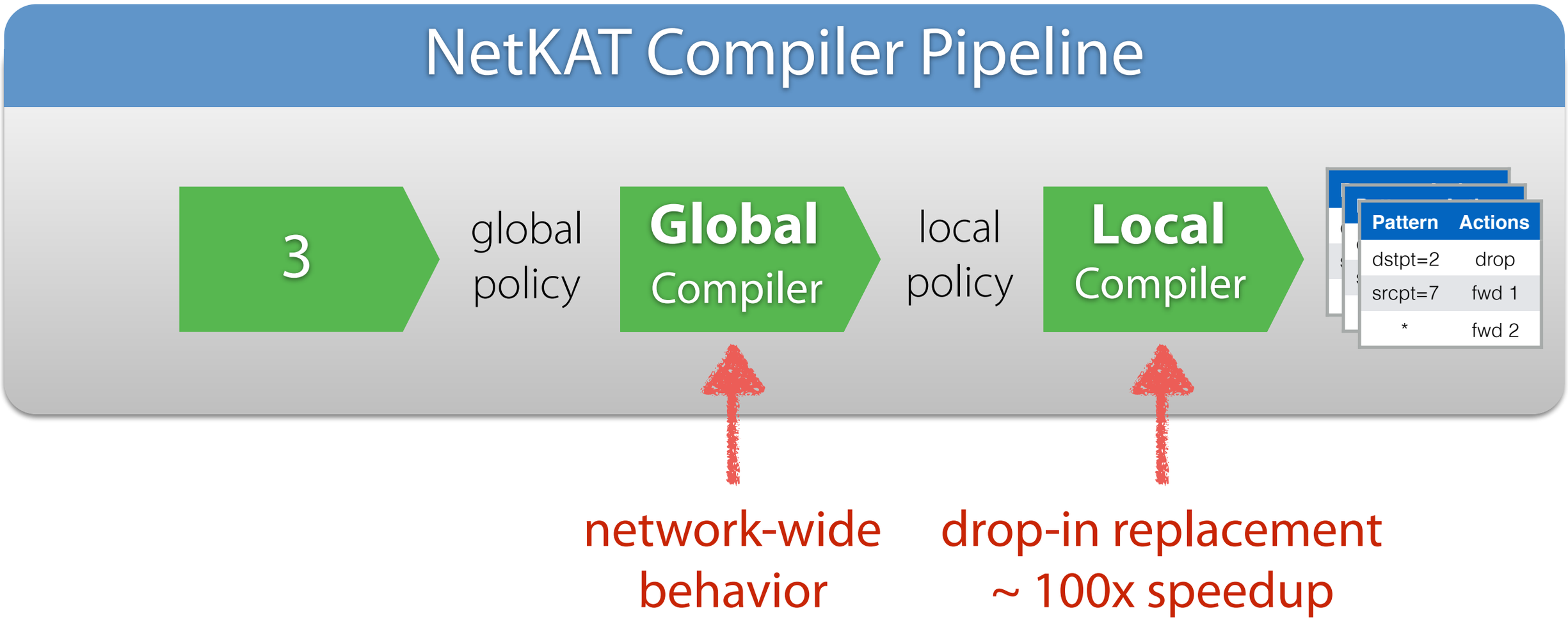


Overview

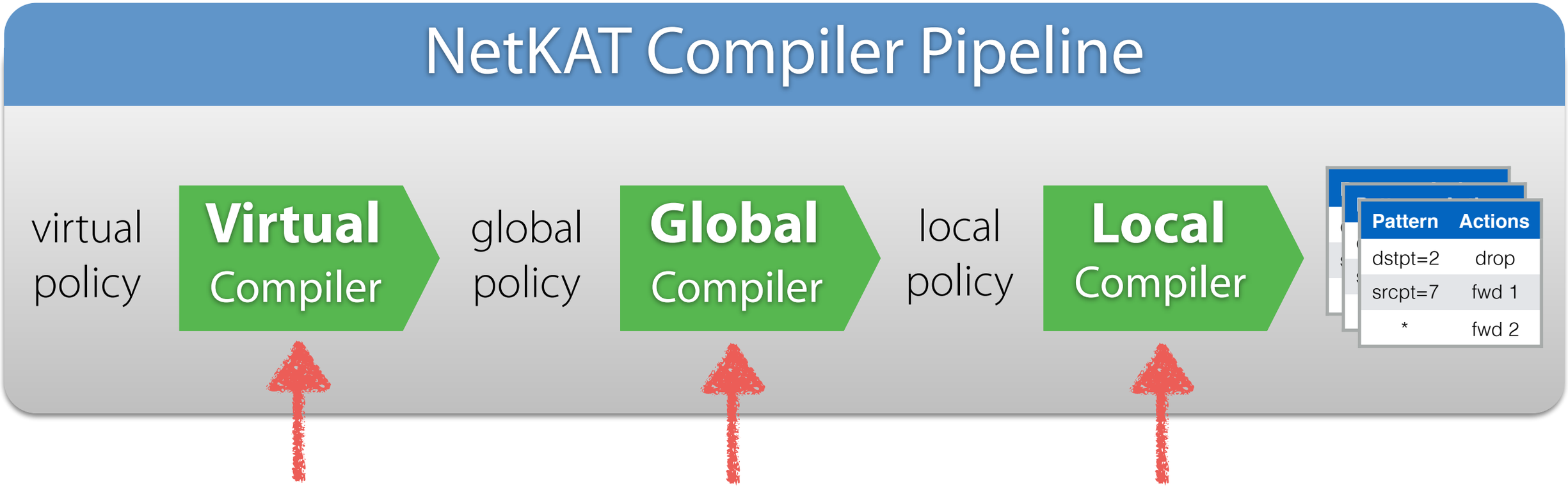


drop-in replacement
~ 100x speedup

Overview


NetKAT Compiler Pipeline
3
global policy
Global Compiler
local policy
Local Compiler
Pattern
Actions
dstpt=2
drop
srcpt=7
fwd 1
*
fwd 2
network-wide behavior
drop-in replacement
~ 100x speedup

Overview



abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup

Overview

NetKAT Compiler Pipeline

virtual policy

Virtual Compiler

global policy

Global Compiler

local policy

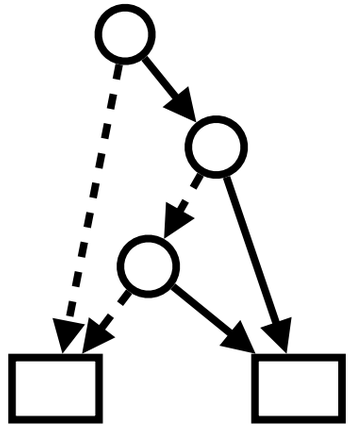
Local Compiler

Pattern	Actions
dstpt=2	drop
srcpt=7	fwd 1
*	fwd 2

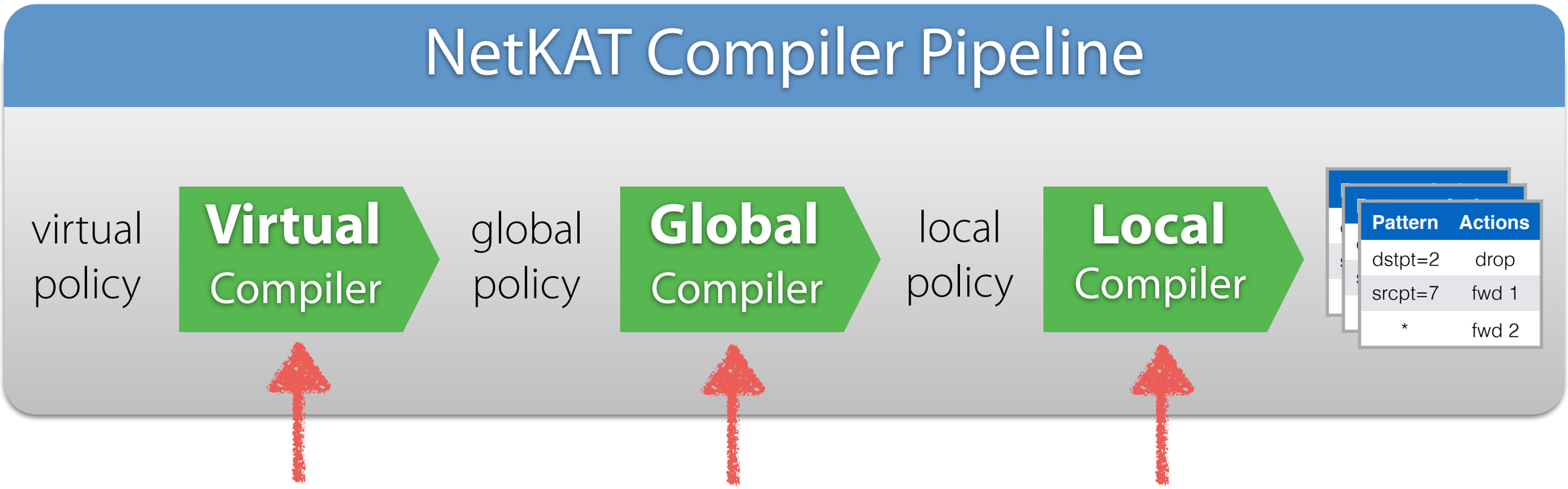
abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup



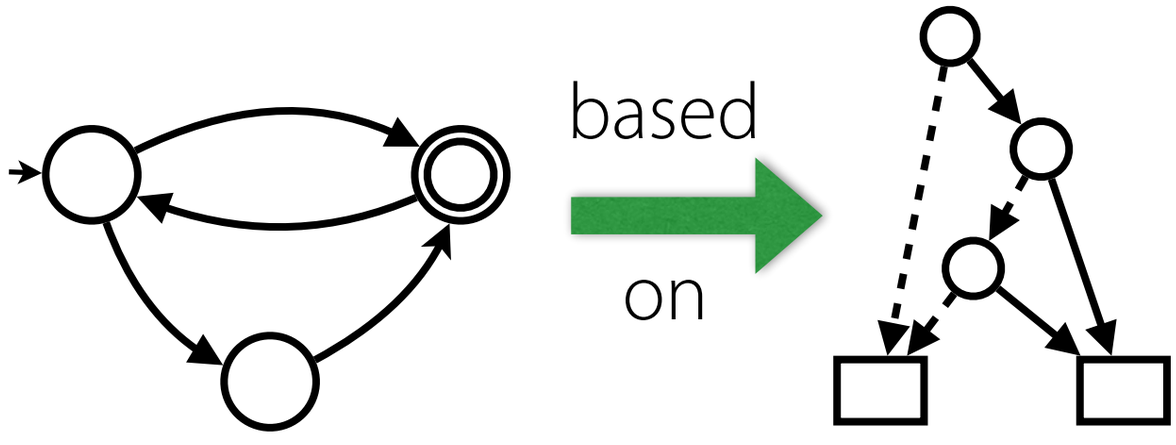
Overview



abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup



Overview

NetKAT Compiler Pipeline

virtual policy

Virtual Compiler

global policy

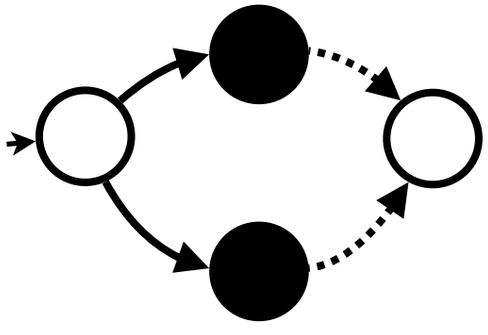
Global Compiler

local policy

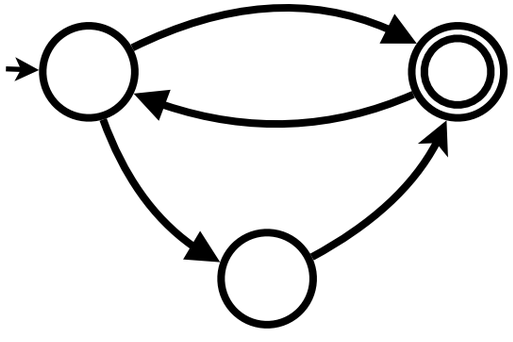
Local Compiler

Pattern	Actions
dstpt=2	drop
srcpt=7	fwd 1
*	fwd 2

abstract topologies

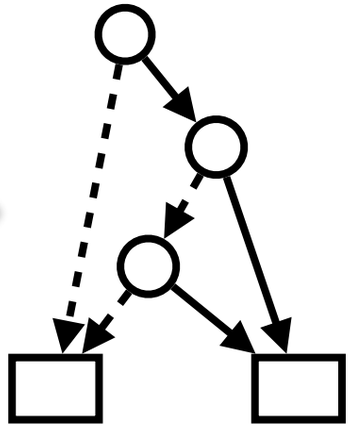


network-wide behavior



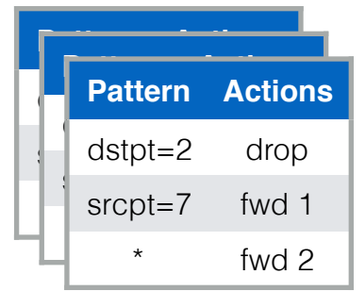
based on

drop-in replacement ~ 100x speedup



Local Compilation

Virtual Compiler → **Global** Compiler → **Local** Compiler



Input: local program

Output: collection of flow tables, one per switch

Challenges: efficiency and size of generated tables

Local Compilation

Input: local program

Output: collection of flow tables, one per switch

Challenges: efficiency and size of generated tables

Strawman Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Strawman Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Strawman Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

+

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Strawman Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Inefficient!

Tables are a hardware abstraction,
not an efficient data structure!!

Better Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

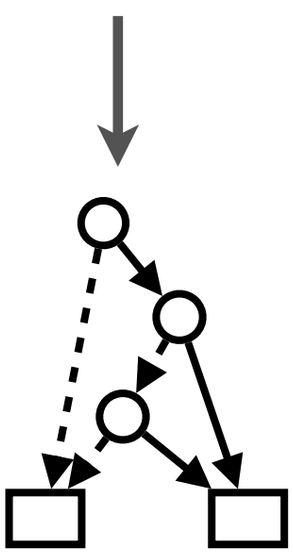
```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

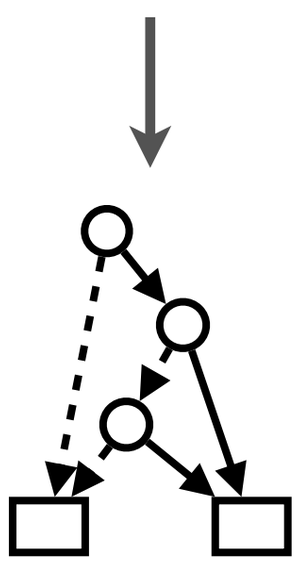
Better Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```



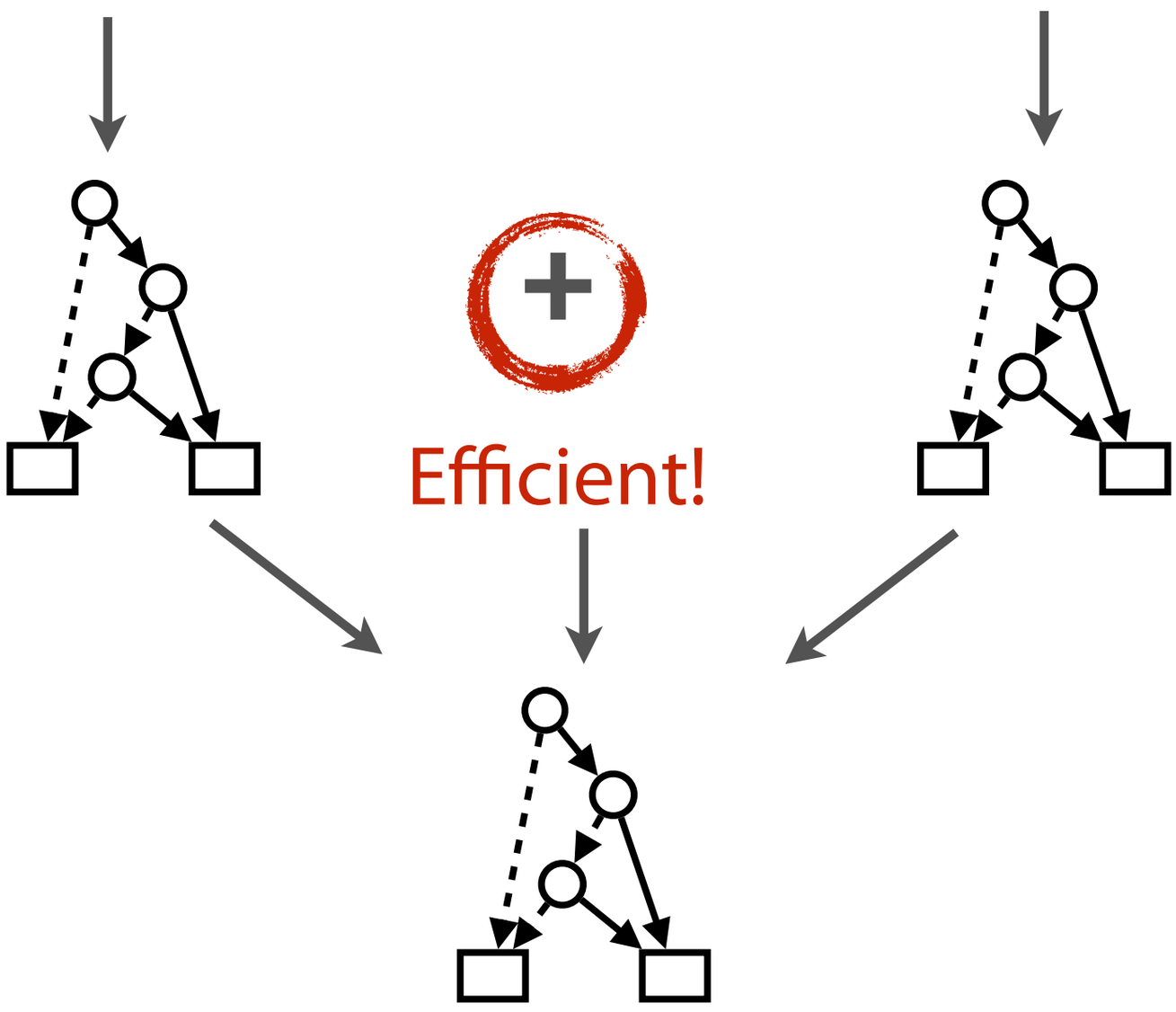


Better Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

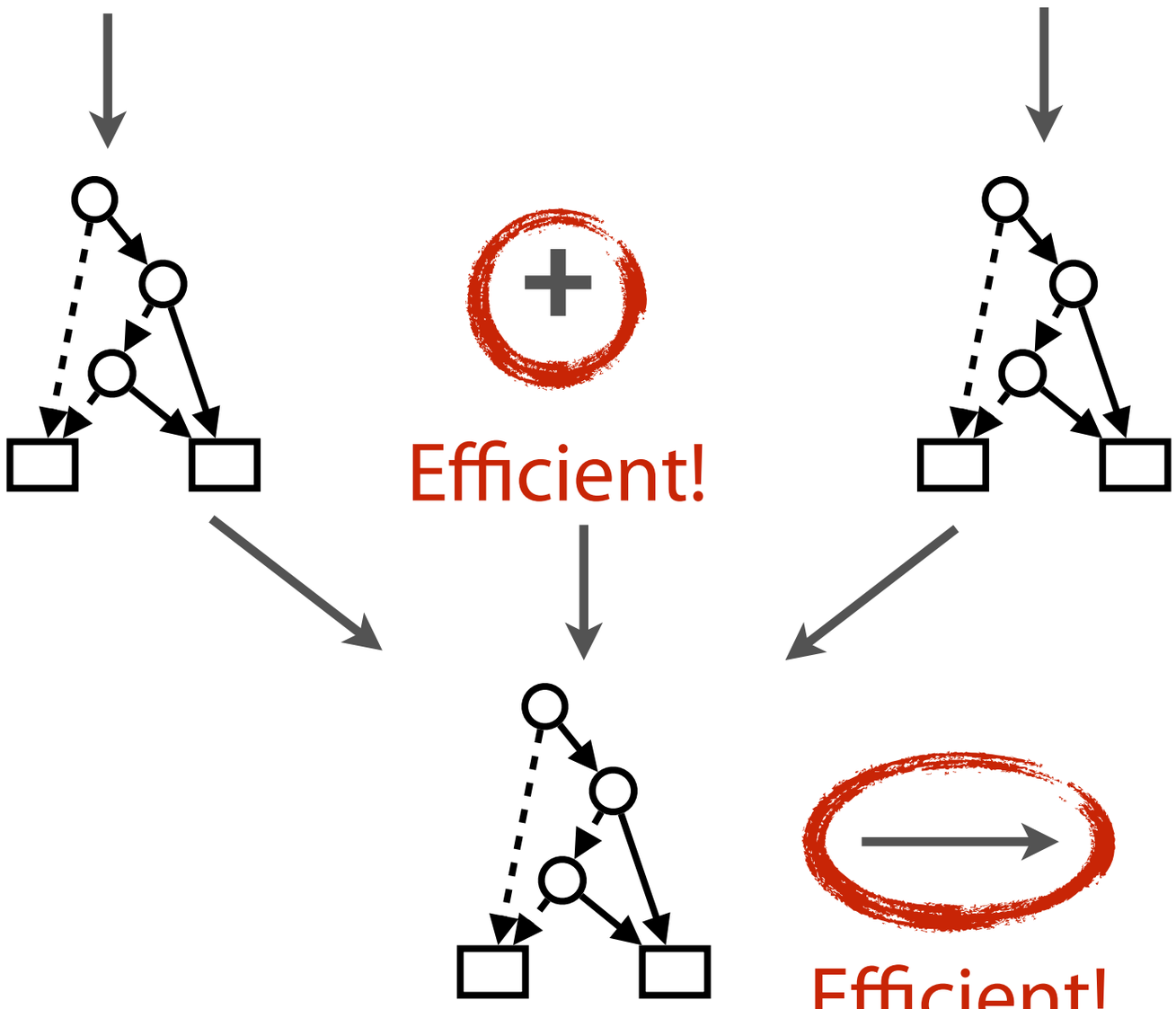


Better Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

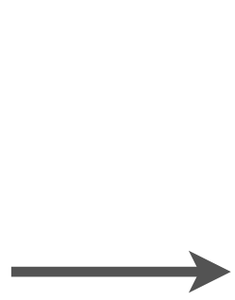
```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

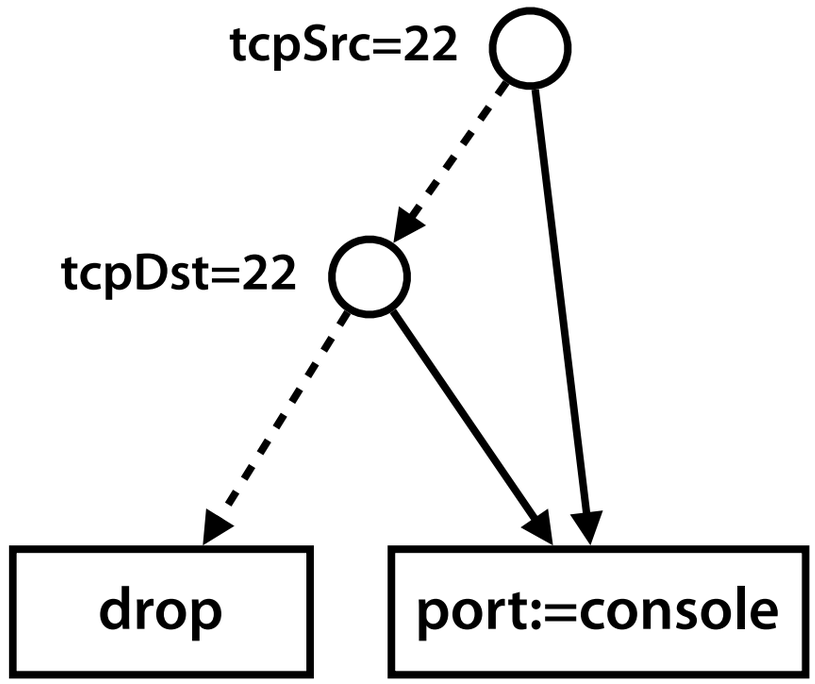


Pattern	Actions
ipDst=10.0.0.1, tcpSrc=22	Forward 1, Controller
ipDst=10.0.0.1, tcpDst=22	Forward 1, Controller
…	

IR: Forwarding Decision Diagrams

```
if (tcpSrc = 22
  + tcpDst = 22)
then
  port := console
else
  drop
```

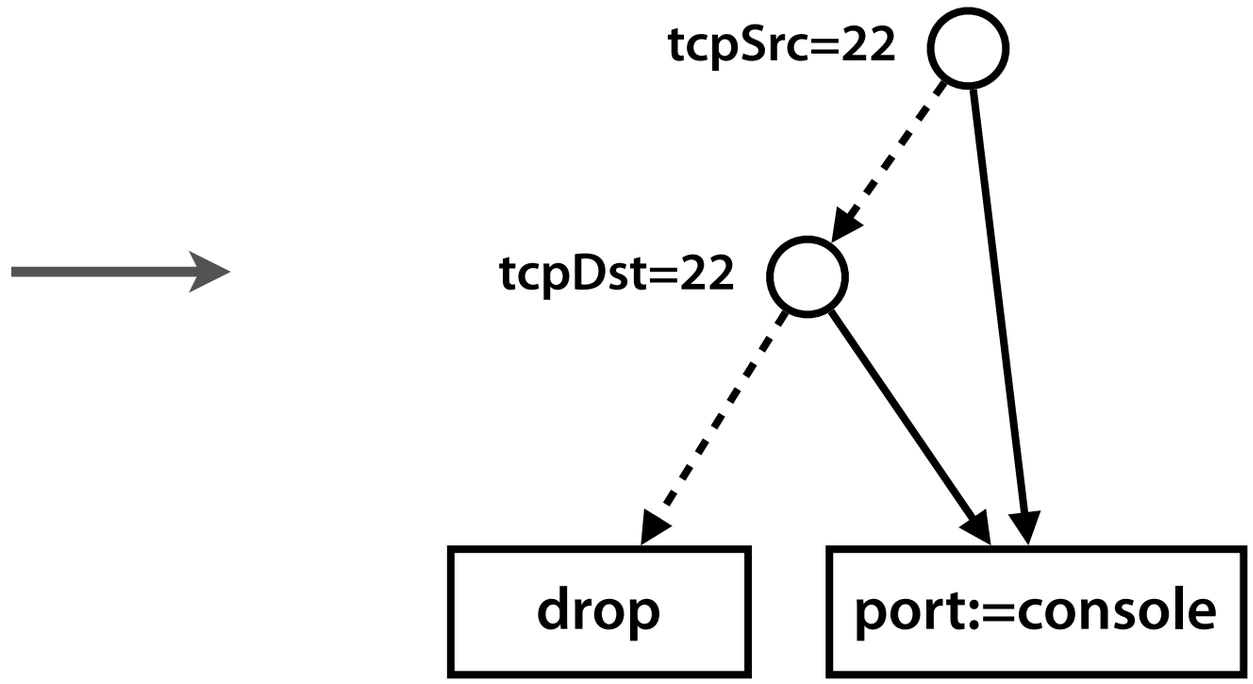




Inspired by Binary Decision Diagrams

IR: Forwarding Decision Diagrams

```
if (tcpSrc = 22
  + tcpDst = 22)
then
  port := console
else
  drop
```



Inspired by Binary Decision Diagrams

NetKAT operators (**+**, **;** , *****, **!**) can be implemented efficiently on FDDs using standard BDD techniques

Global Compilation

Input: NetKAT program (with links)

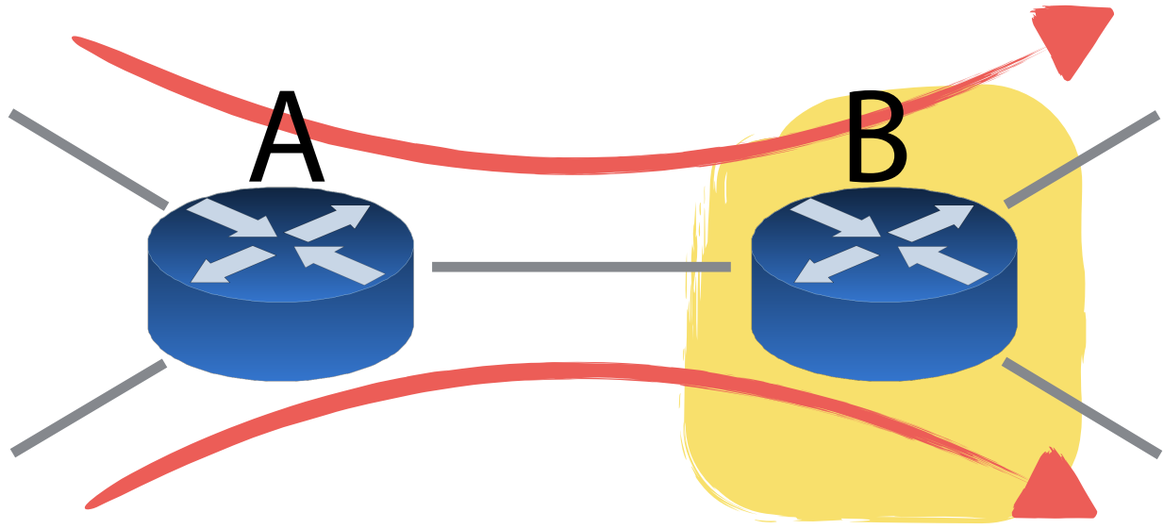
Output: equivalent local program (without links)

Global Compilation

Input: NetKAT program (with links)

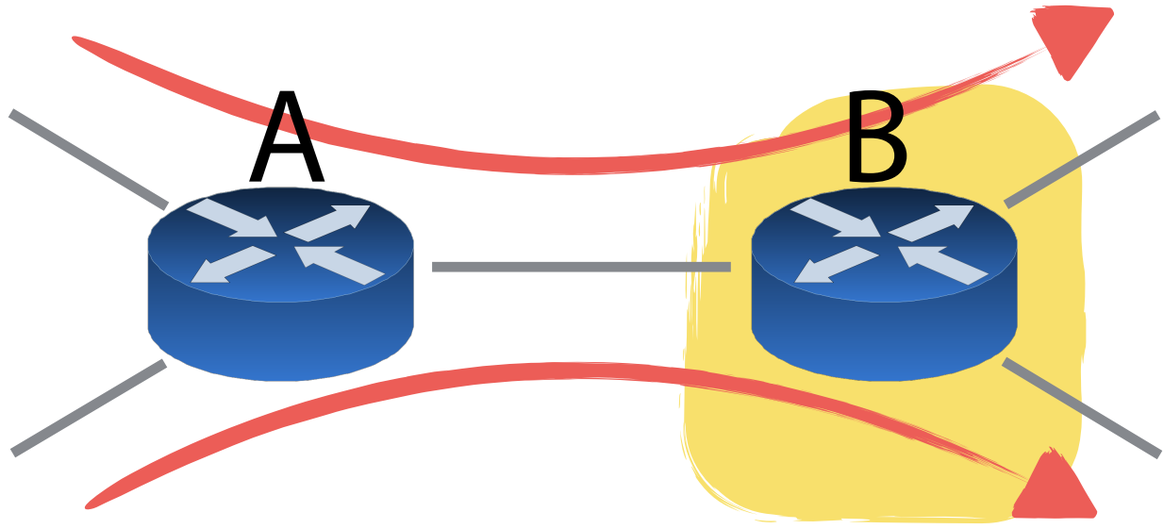
Output: equivalent local program (without links)

Main Challenges



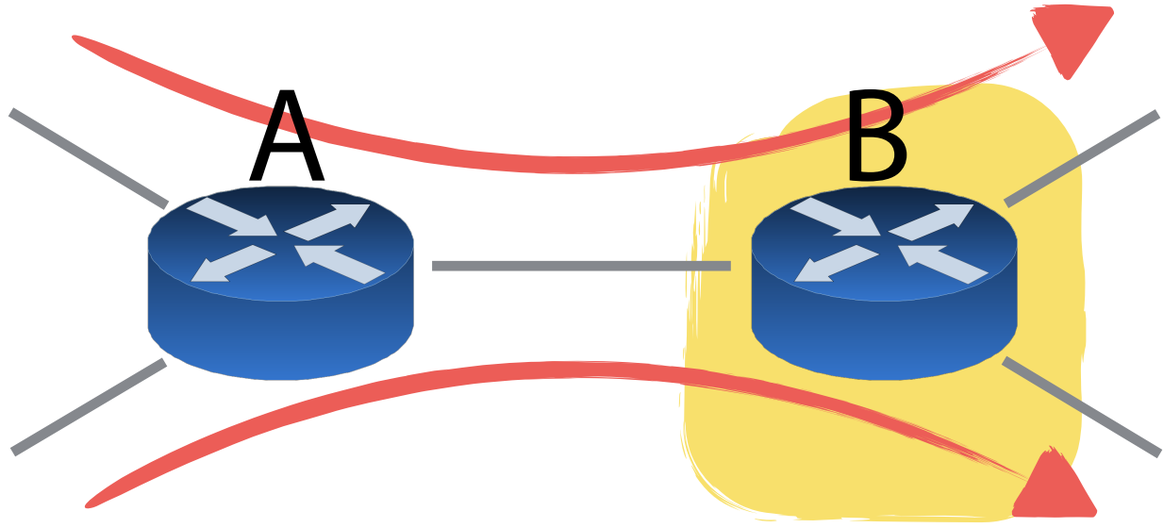
Main Challenges

1. Adding Extra State
"Tagging"

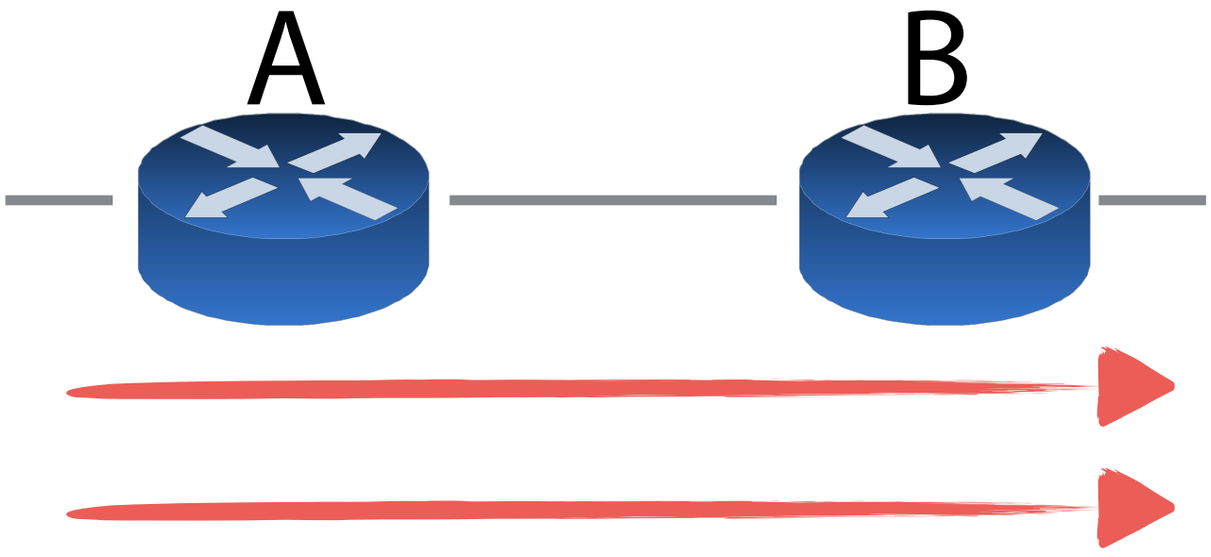


Main Challenges

1. Adding Extra State
"Tagging"

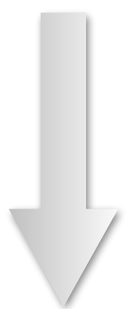


2. Avoiding Duplication
(naive tagging is unsound!)



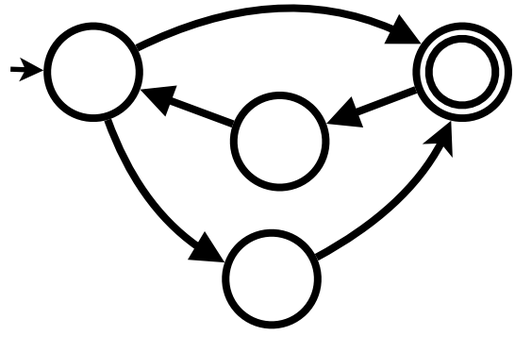
Our Solution

Our Solution



Adding Extra State
= Translation to Automaton

NetKAT NFA

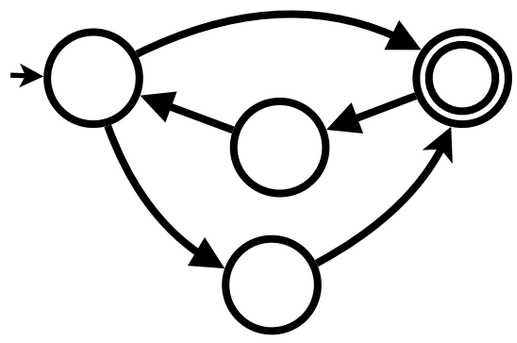


Our Solution

Global Program

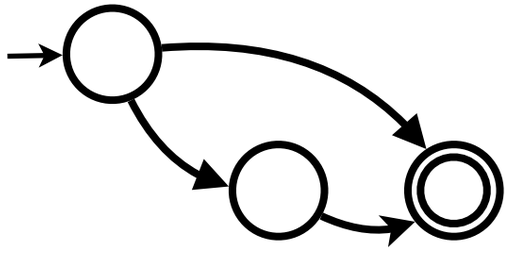
Adding Extra State
= Translation to Automaton

NetKAT NFA



Avoiding Duplication
= Determinization

NetKAT DFA

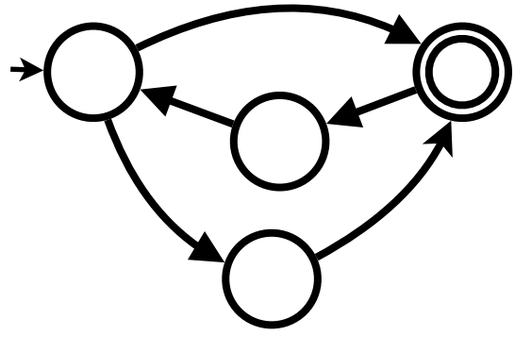


Our Solution

Global Program

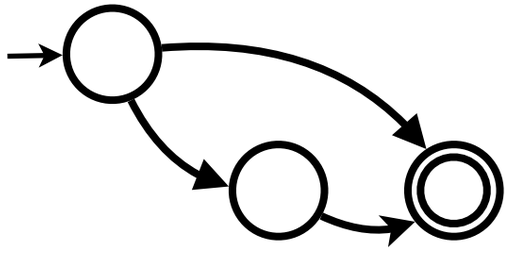
Adding Extra State
= Translation to Automaton

NetKAT NFA



Avoiding Duplication
= Determinization

NetKAT DFA



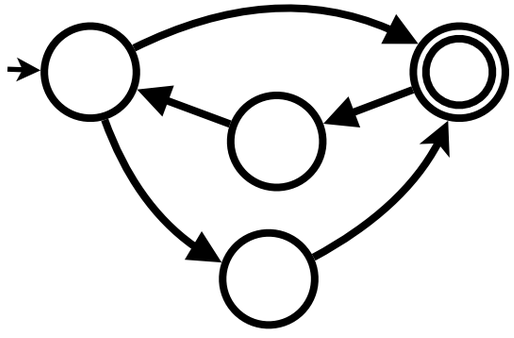
Local Program

Our Solution

Global Program

Adding Extra State
= Translation to Automaton

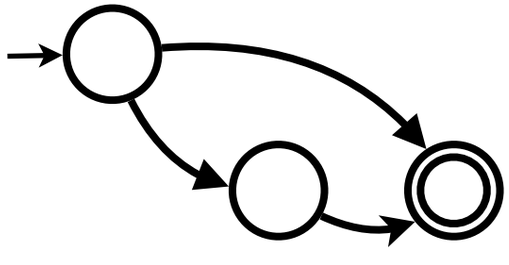
NetKAT NFA



Automaton Minimization
= Tag Elimination

Avoiding Duplication
= Determinization

NetKAT DFA



Local Program

Virtual Compilation

Input: program written against virtual topology

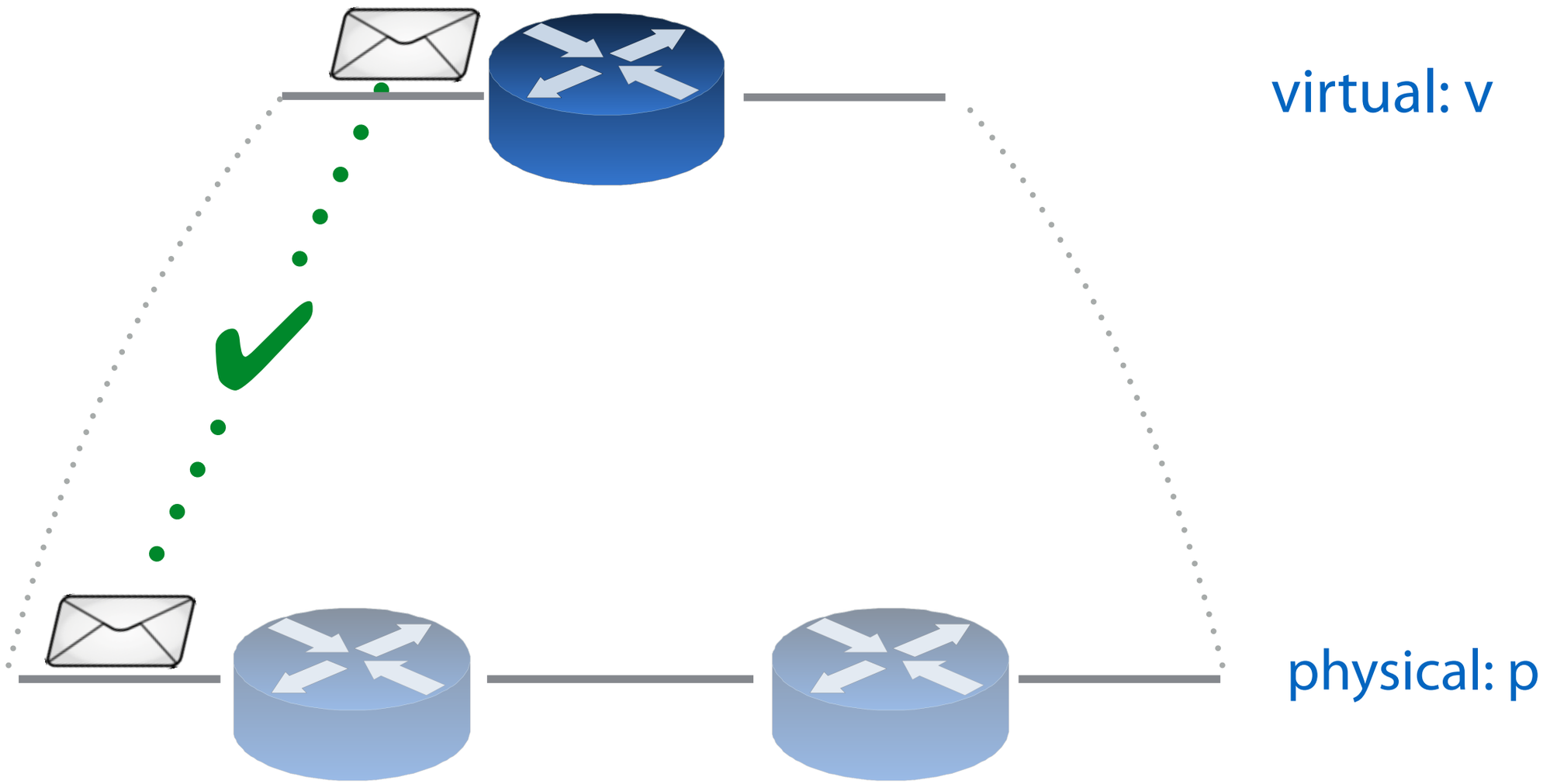
Output: global program that simulates virtual behavior

Virtual Compilation

Input: program written against virtual topology

Output: global program that simulates virtual behavior

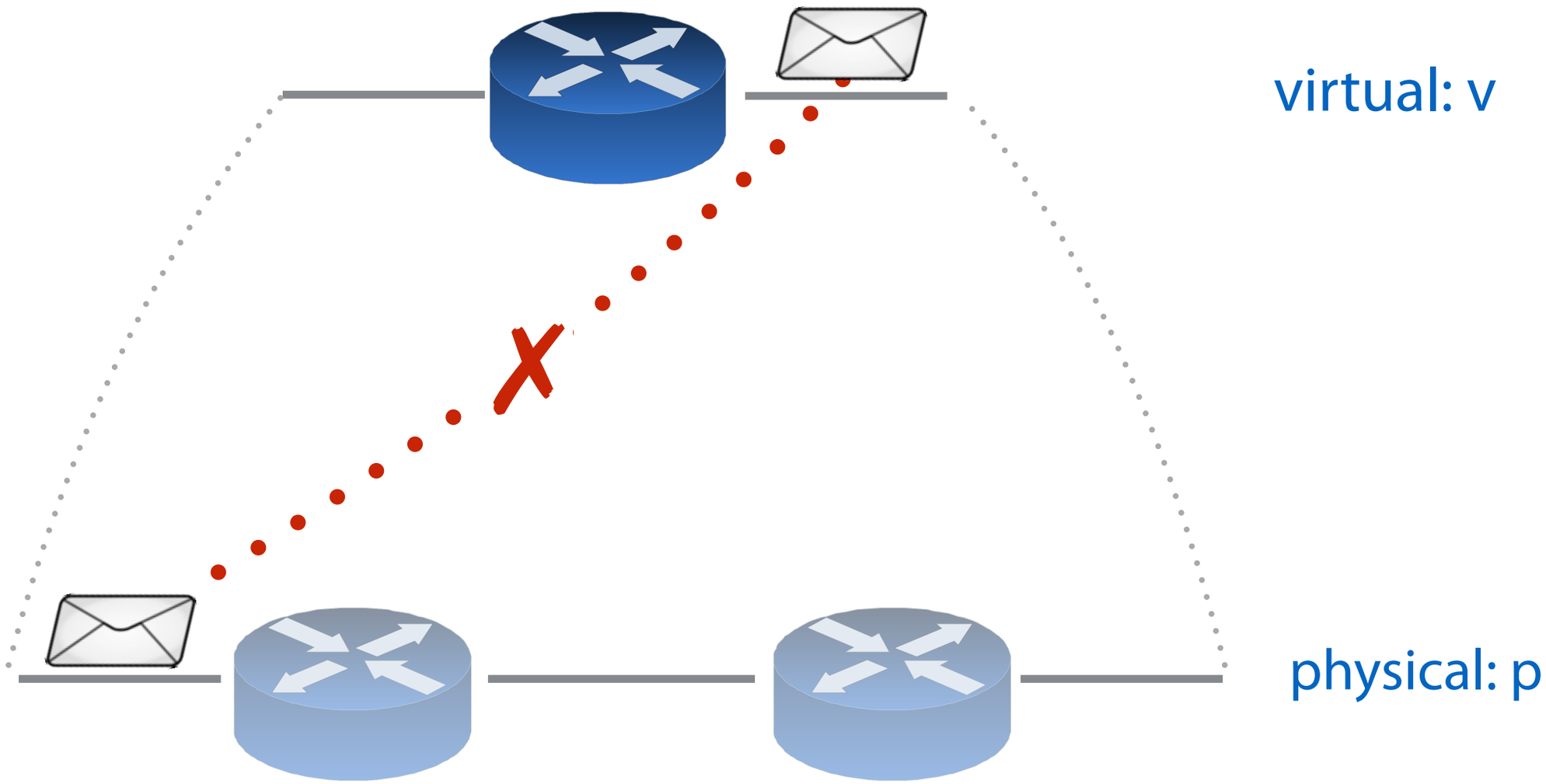
Virtualization



virtual: v

physical: p

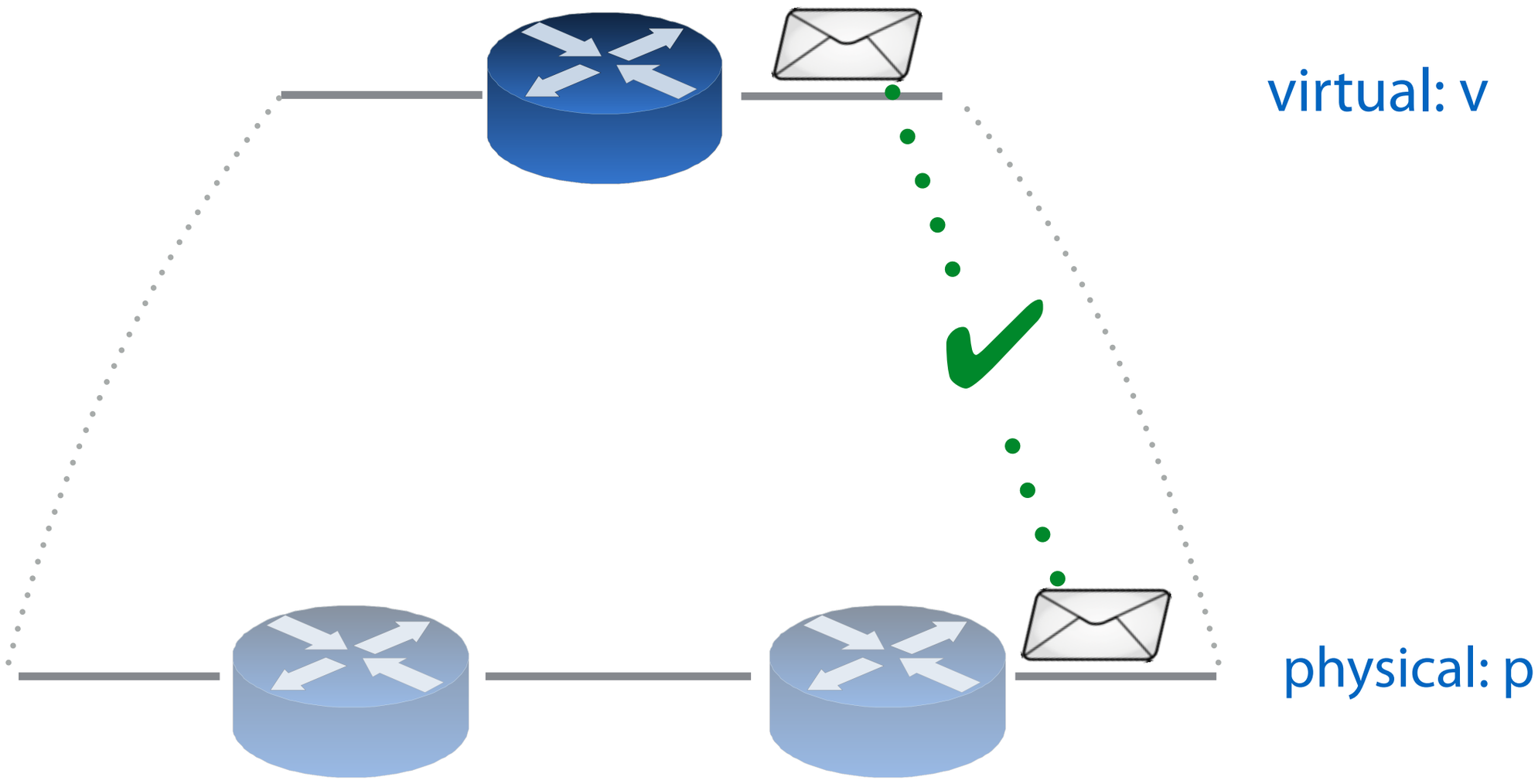
Virtualization



virtual: v

physical: p

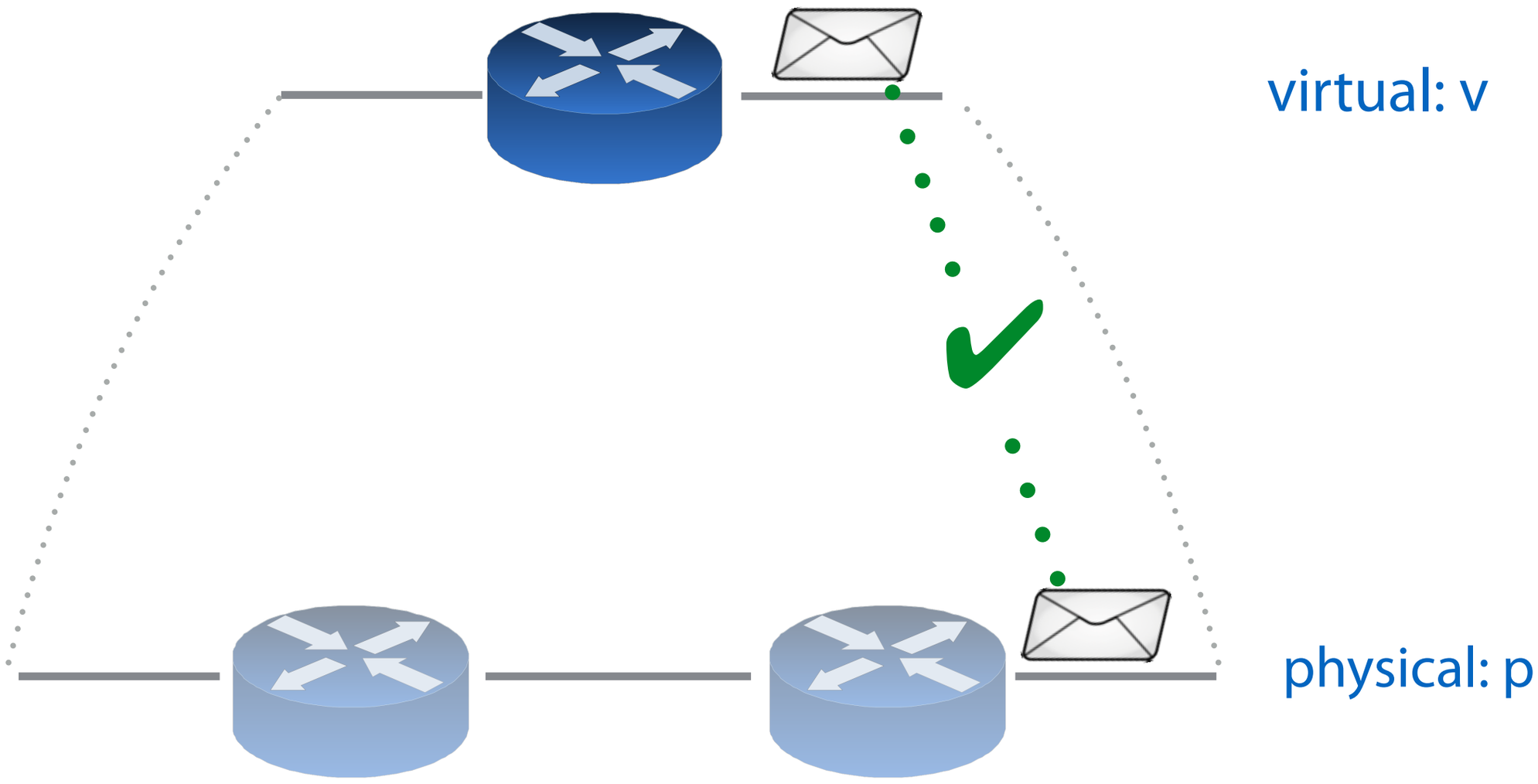
Virtualization



virtual: v

physical: p

Virtualization



Observation: can formulate execution of a virtual program as a two-player game

Compiler: synthesizes physical program p that encodes a winning strategy to all instances of that game

NetKAT Semantics

Denotational

NetKAT Semantics

Denotational

Axiomatic

$\vdash p \equiv q$

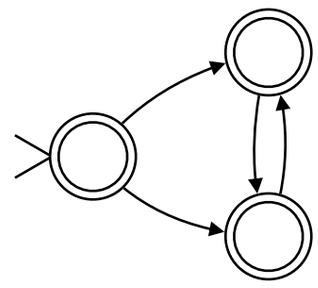
NetKAT Semantics

Denotational

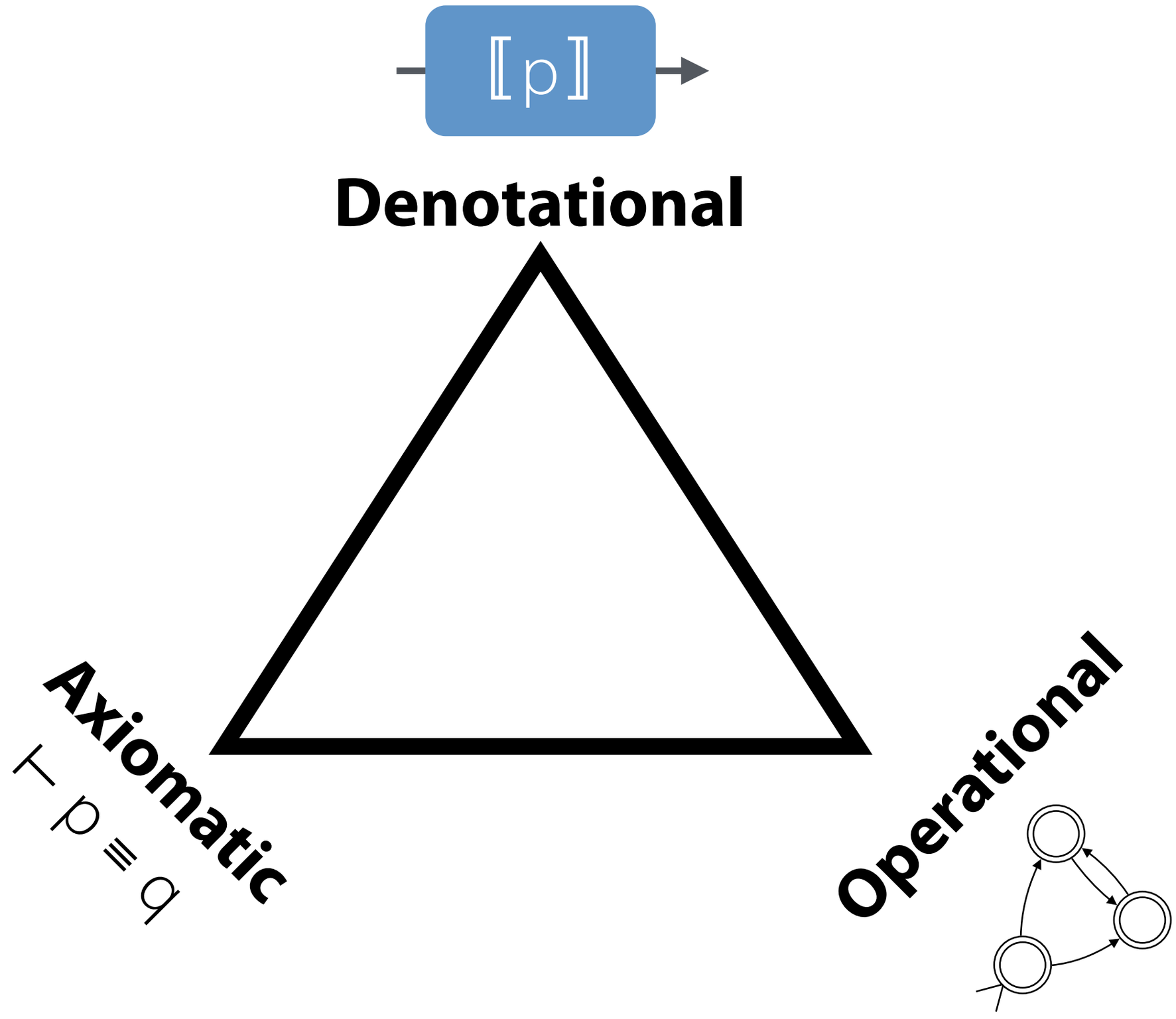
Axiomatic

$\vdash p \equiv q$

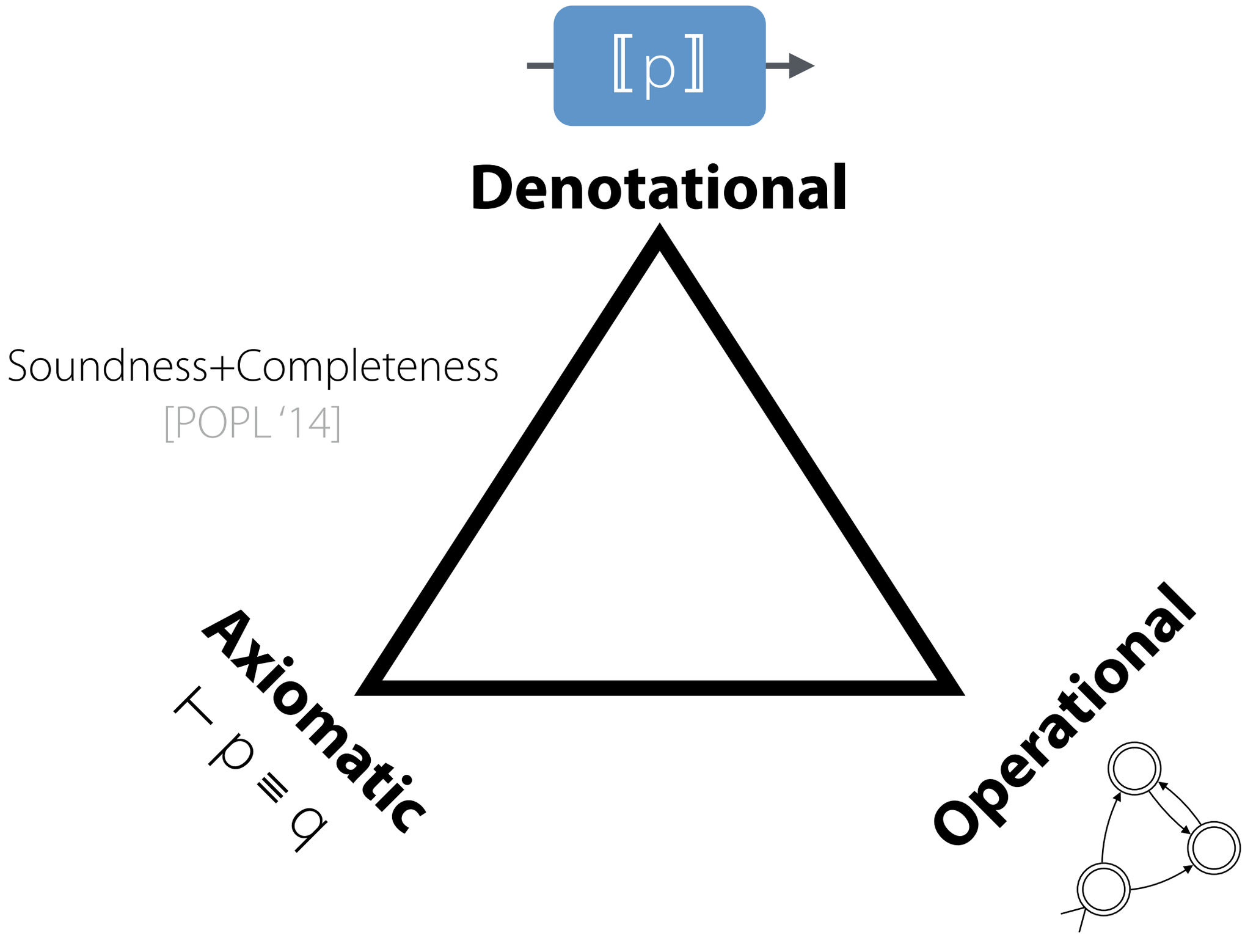
Operational



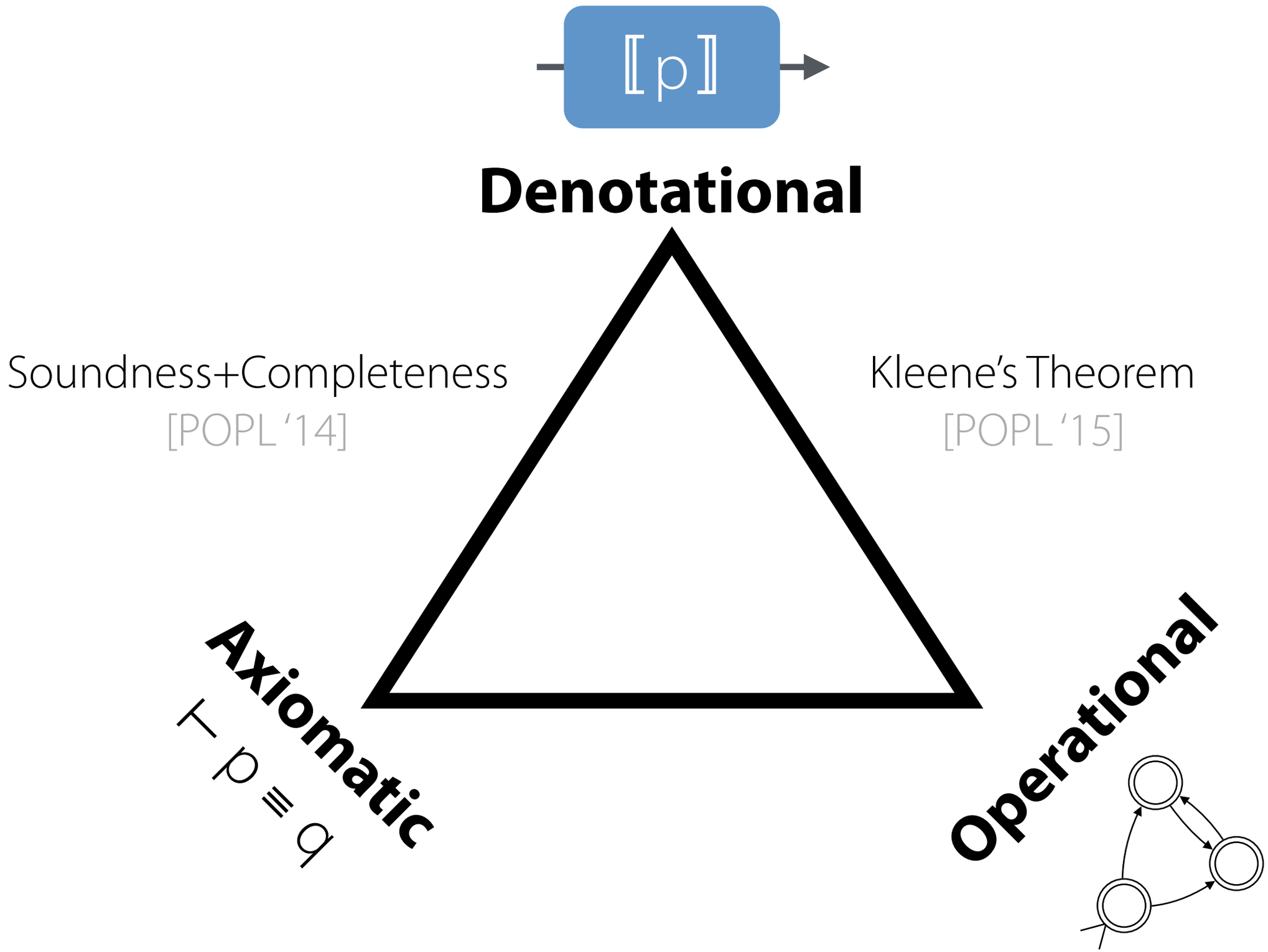
NetKAT Semantics



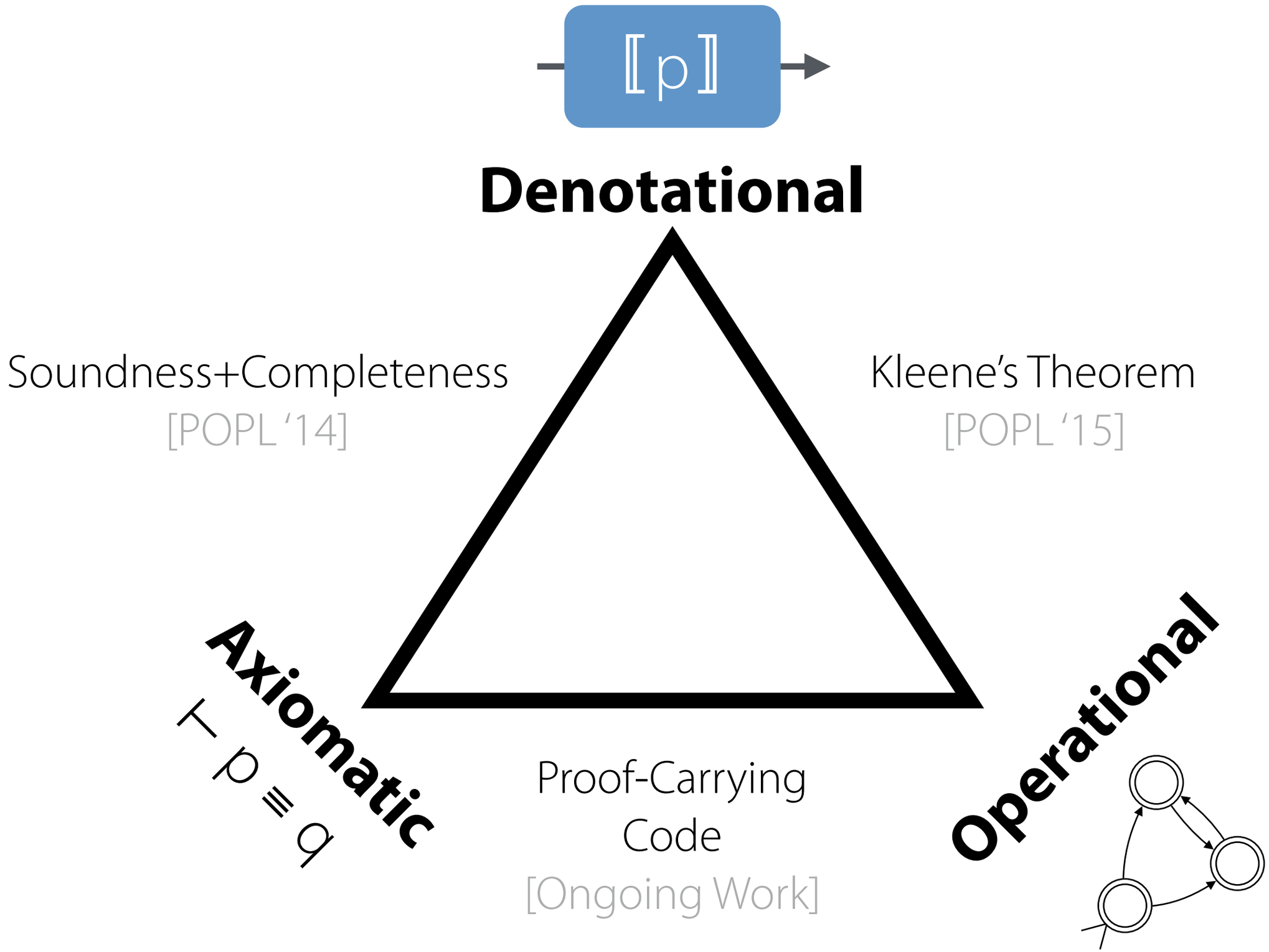
NetKAT Semantics



NetKAT Semantics



NetKAT Semantics



Denotational Semantics

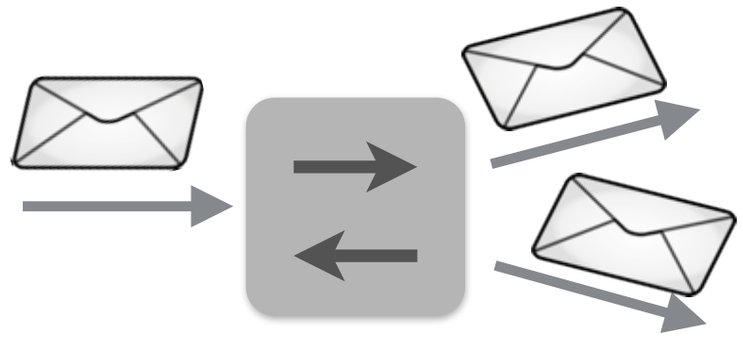
Denotational Semantics

pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 • pol_2
- | !pol
- | pol*
- | **dup**

Denotational Semantics

pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 • pol_2
- | !pol
- | pol*
- | ~~**dup**~~

Local: input-output behavior of switches

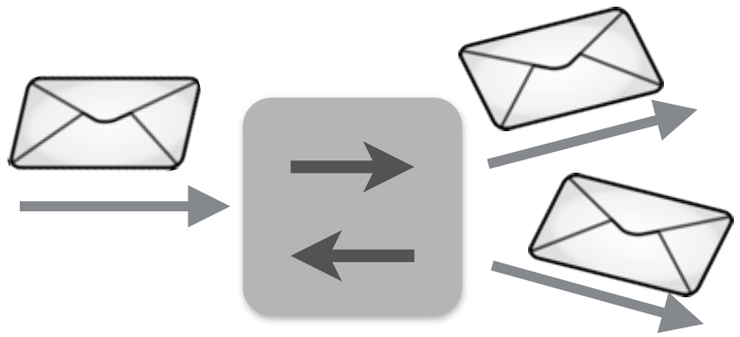


⟦pol⟧ ∈ Packets → Packets

Denotational Semantics

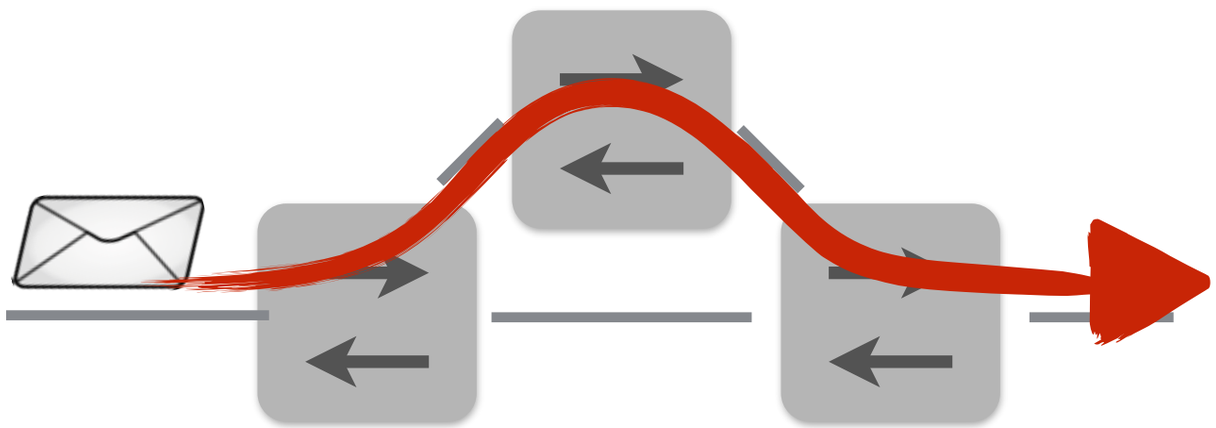
pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 • pol_2
- | !pol
- | pol*
- | **dup**

Local: input-output behavior of switches



⟦pol⟧ ∈ Packets → Packets

Global: network-wide paths



⟦pol⟧ ∈ Histories → Histories

NetKAT Semantics

$[\![pol]\!] \in \textbf{History} \rightarrow \textbf{History Set}$

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

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

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

$[\![\textbf{!}\ pol]\!]\ h = \{ h \} \setminus [\![pol]\!]$

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

$[\![pol_1 + pol_2]\!]\ h = [\![pol_1]\!]\ h \cup [\![pol_2]\!]\ h$

$[\![pol_1 \bullet pol_2]\!]\ h = ([\![pol_1]\!] \bullet [\![pol_2]\!])\ h$

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

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

$f,g \in \textbf{History} \rightarrow \textbf{History Set}$

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

Axiomatic Semantics

Kleene Algebra with Tests

The design of NetKAT is not an accident!

Its foundation rests upon canonical structure:

- Regular operators (**+**, •, and ***) encode network paths
- Boolean operators (**+**, •, and **!**) encode switch tables

Need to extend KAT with additional axioms to obtain a sound and complete system

NetKAT Equational Axioms

Kleene Algebra Axioms

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

$p + q \equiv q + p$

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

$p + p \equiv p$

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

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

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

$\mathbf{true} \cdot p \equiv p$

$p \equiv p \cdot \mathbf{true}$

$\mathbf{false} \cdot p \equiv \mathbf{false}$

$p \cdot \mathbf{false} \equiv \mathbf{false}$

$\mathbf{true} + p \cdot p^* \equiv p^*$

$\mathbf{true} + p^* \cdot p \equiv p^*$

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

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

Boolean Algebra Axioms

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

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

$a + \mathbf{!}\, a \equiv \mathbf{true}$

$a \cdot b \equiv b \cdot a$

$a \cdot \mathbf{!}a \equiv \mathbf{false}$

$a \cdot a \equiv a$

Packet Axioms

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

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

$f := n \cdot f = n \equiv f := n$

$f = n \cdot f := n \equiv f = n$

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

$f = n \cdot f = n' \equiv \mathbf{false} \quad \text{if } n \neq n'$

$dup \cdot f = n \equiv f = n \cdot dup$

$\sum_i f = n_i \equiv \mathbf{true}$

NetKAT Equational Axioms

Kleene Algebra Axioms

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

$p + q \equiv q + p$

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

$p + p \equiv p$

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

$p \bullet (q + r)$

$(p + q) \bullet$

$\mathbf{true} \bullet p \equiv$

$p \equiv p \bullet \mathbf{true}$

$\mathbf{false} \bullet p \equiv \mathbf{false}$

$p \bullet \mathbf{false} \equiv \mathbf{false}$

$\mathbf{true} + p \bullet p^* \equiv p^*$

$\mathbf{true} + p^* \bullet p \equiv p^*$

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

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

Boolean Algebra Axioms

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

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

$a + !\, a \equiv \mathbf{true}$

$a \bullet b = b \bullet a$

Packet Algebra Axioms

$f := n \bullet f := n' \equiv f := n'$ if $f \neq f'$

$f := n \bullet f' = n' \equiv f' = n' \bullet f := n$ if $f \neq f'$

$f := n \bullet f = n \equiv f := n$

$f = n \bullet f := n \equiv f = n$

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

$f = n \bullet f = n' \equiv \mathbf{false}$ if $n \neq n'$

$dup \bullet f = n \equiv f = n \bullet dup$

$\sum_i f = n_i \equiv \mathbf{true}$

Soundness: If $\vdash p \equiv q$, then $[\![p]\!] = [\![q]\!]$

Completeness: If $[\![p]\!] = [\![q]\!]$, then $\vdash p \equiv q$

Reduced NetKAT

Complete tests

α ::= **switch =** n • **port =** n

Complete assignments

β ::= **switch :=** n • **port :=** n

Reduced terms

```
p,q::= α          (* complete test *)
     | β          (* complete assignment *)
     | p + q      (* union *)
     | p • q      (* sequence *)
     | p*         (* Kleene star *)
     | dup        (* Duplication *)
```

Reduced NetKAT

Complete tests

α ::= **switch =** n • **port =** n

For simplicity, only consider two fields

Complete assignments

β ::= **switch :=** n • **port :=** n

Reduced terms

```
p,q::= α          (* complete test *)
     | β          (* complete assignment *)
     | p + q      (* union *)
     | p • q      (* sequence *)
     | p*         (* Kleene star *)
     | dup        (* Duplication *)
```

Reduced NetKAT

Complete tests

α ::= **switch =** n • **port =** n

Complete assignments

β ::= **switch :=** n • **port :=** n

Reduced terms

```
p,q::= α          (* complete test *)
     | β          (* complete assignment *)
     | p + q      (* union *)
     | p • q      (* sequence *)
     | p*         (* Kleene star *)
     | dup        (* Duplication *)
```

For simplicity, only consider two fields

Lemma: For every NetKAT term p, there is a reduced NetKAT term p' such that ⊢ p ≡ p'

Regular Interpretation

Can interpret reduced terms as regular languages over an "alphabet" of complete tests, complete assignments, and **dup**:

Regular Interpretation

Can interpret reduced terms as regular languages over an "alphabet" of complete tests, complete assignments, and **dup**:

Regular Interpretation: $R(p) \subseteq (A \cup B \cup \{\textbf{dup}\})^*$

$R(\alpha) = \{\alpha\}$

$R(\beta) = \{\beta\}$

$R(p + q) = R(p) \cup R(q)$

$R(p \bullet q) = R(p) \bullet R(q)$

$R(p^*) = R(p)^*$

$R(\textbf{dup}) = \{\textbf{dup}\}$

Regular Interpretation

Can interpret reduced terms as regular languages over an "alphabet" of complete tests, complete assignments, and **dup**:

Regular Interpretation: $R(p) \subseteq (A \cup B \cup \{\textbf{dup}\})^*$

$R(\alpha) = \{\alpha\}$

$R(\beta) = \{\beta\}$

$R(p + q) = R(p) \cup R(q)$

$R(p \bullet q) = R(p) \bullet R(q)$

$R(p^*) = R(p)^*$

$R(\textbf{dup}) = \{\textbf{dup}\}$

Unfortunately $[\![p]\!] = [\![q]\!]$ does not imply $R(p) = R(q)$

Regular Interpretation

Can interpret reduced terms as regular languages over an "alphabet" of complete tests, complete assignments, and **dup**:

Regular Interpretation: $R(p) \subseteq (A \cup B \cup \{\textbf{dup}\})^*$

$R(\alpha) = \{\alpha\}$

$R(\beta) = \{\beta\}$

$R(p + q) = R(p) \cup R(q)$

$R(p \bullet q) = R(p) \bullet R(q)$

$R(p^*) = R(p)^*$

$R(\textbf{dup}) = \{\textbf{dup}\}$

Unfortunately $[\![p]\!] = [\![q]\!]$ does not imply $R(p) = R(q)$

Counterexample:

switch=1 · port=1 · switch=1 · port=2 and
switch=1 · port=1 · switch=2 · port=1

Language Model

Language Model

Language Interpretation: $G(p) \subseteq A \bullet (B \bullet \{\mathbf{dup}\})^* \bullet B$

$G(\alpha) = \{\alpha \bullet \pi_\alpha\}$

$G(\beta) = \{\alpha \bullet \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \bullet q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \bullet \beta_\alpha \bullet \mathbf{dup} \bullet \beta_\alpha \mid \alpha \in A\}$

Language Model

Language Interpretation: $G(p) \subseteq A \bullet (B \bullet \{\mathbf{dup}\})^* \bullet B$

$G(\alpha) = \{\alpha \bullet \pi_\alpha\}$

$G(\beta) = \{\alpha \bullet \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \bullet q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \bullet \beta_\alpha \bullet \mathbf{dup} \bullet \beta_\alpha \mid \alpha \in A\}$

Guarded strings

Language Model

Language Interpretation: $G(p) \subseteq A \bullet (B \bullet \{\mathbf{dup}\})^* \bullet B$

$G(a) = \{a \bullet \pi_a\}$

$G(\beta) = \{a \bullet \beta \mid a \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \bullet q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{a \bullet \beta_a \bullet \mathbf{dup} \bullet \beta_a \mid a \in A\}$

Guarded strings

Guarded concatenation

Language Model

Language Interpretation: $G(p) \subseteq A \bullet (B \bullet \{\mathbf{dup}\})^* \bullet B$

$G(\alpha) = \{\alpha \bullet \pi_\alpha\}$

$G(\beta) = \{\alpha \bullet \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \bullet q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \bullet \beta_\alpha \bullet \mathbf{dup} \bullet \beta_\alpha \mid \alpha \in A\}$

Guarded strings

Guarded concatenation

Example: $\alpha_1 \bullet \beta_2 \bullet \mathbf{dup} \bullet \beta_3 \bullet \mathbf{dup} \bullet \ldots \bullet \mathbf{dup} \bullet \beta_n$

Language Model

Language Interpretation: $G(p) \subseteq A \bullet (B \bullet \{\mathbf{dup}\})^* \bullet B$

$G(\alpha) = \{\alpha \bullet \pi_\alpha\}$

$G(\beta) = \{\alpha \bullet \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \bullet q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \bullet \beta_\alpha \bullet \mathbf{dup} \bullet \beta_\alpha \mid \alpha \in A\}$

Guarded strings

Guarded concatenation

Example: $\alpha_1 \bullet \beta_2 \bullet \mathbf{dup} \bullet \beta_3 \bullet \mathbf{dup} \bullet \ldots \bullet \mathbf{dup} \bullet \beta_n$

Intuition: models trajectories through the network

Language Model

Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\mathbf{dup}\})^* \cdot B$

$G(\alpha) = \{\alpha \cdot \pi_\alpha\}$

$G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \cdot \beta_\alpha \cdot \mathbf{dup} \cdot \beta_\alpha \mid \alpha \in A\}$

Guarded strings

Guarded concatenation

Example: $\alpha_1 \cdot \beta_2 \cdot \mathbf{dup} \cdot \beta_3 \cdot \mathbf{dup} \cdot \ldots \cdot \mathbf{dup} \cdot \beta_n$

Intuition: models trajectories through the network

Theorem: $[\![p]\!] = [\![q]\!]$ if and only if $G(p) = G(q)$

Language Model

Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\mathbf{dup}\})^* \cdot B$

$G(\alpha) = \{\alpha \cdot \pi_\alpha\}$

$G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\mathbf{dup}) = \{\alpha \cdot \beta_\alpha \cdot \mathbf{dup} \cdot \beta_\alpha \mid \alpha \in A\}$

Guarded strings

Guarded concatenation

Example: $\alpha_1 \cdot \beta_2 \cdot \mathbf{dup} \cdot \beta_3 \cdot \mathbf{dup} \cdot \ldots \cdot \mathbf{dup} \cdot \beta_n$

Intuition: models trajectories through the network

Theorem: $[\![p]\!] = [\![q]\!]$ if and only if $G(p) = G(q)$

Lemma: For all p, there exists a normal form $\hat{p}$ such that $\vdash p \equiv \hat{p}$ and $G(\hat{p})$ is regular

Completeness Proof

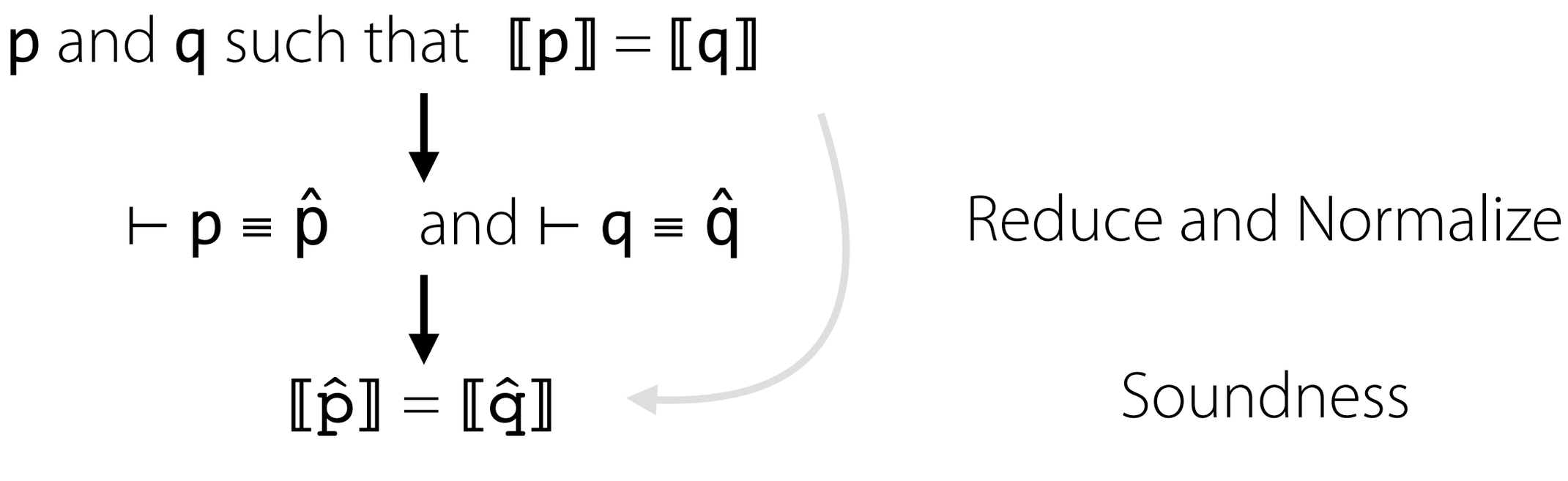
p and q such that $[\![\mathsf{p}]\!] = [\![\mathsf{q}]\!]$

Completeness Proof

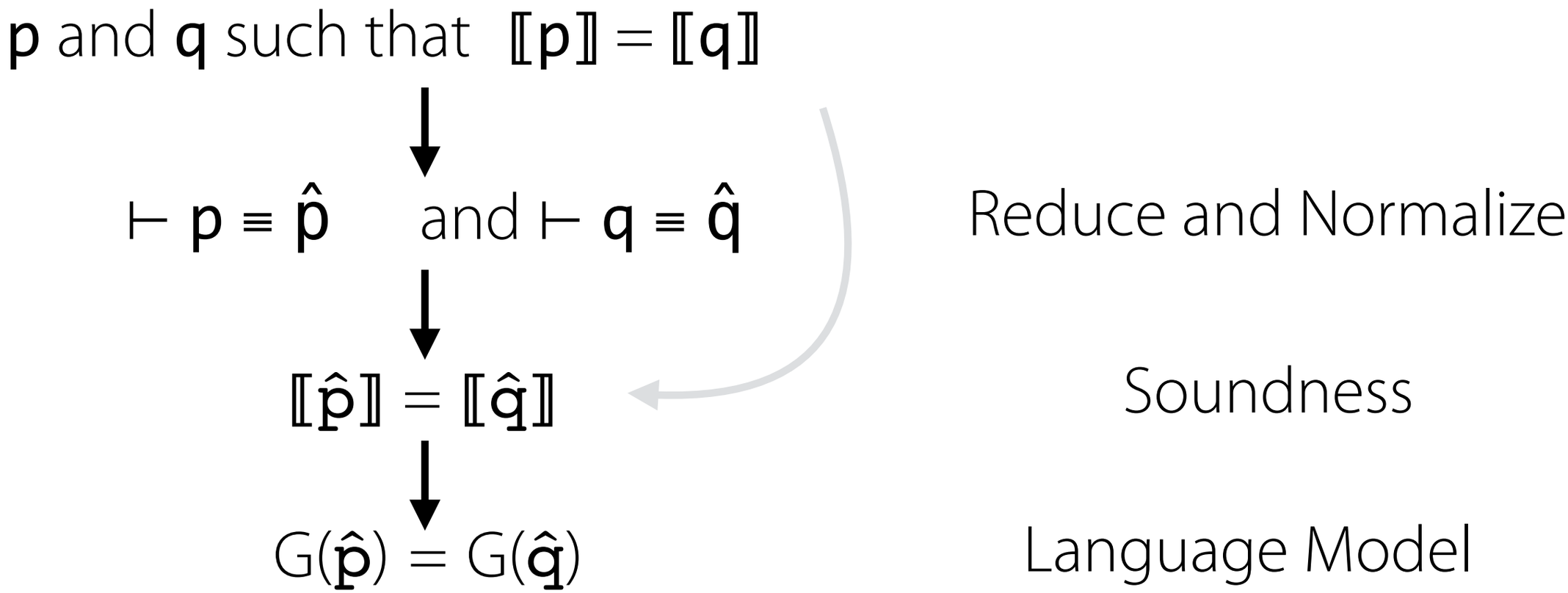
p and q such that $[\![\mathsf{p}]\!] = [\![\mathsf{q}]\!]$

$\vdash \mathsf{p} \equiv \hat{\mathsf{p}}$ and $\vdash \mathsf{q} \equiv \hat{\mathsf{q}}$ Reduce and Normalize

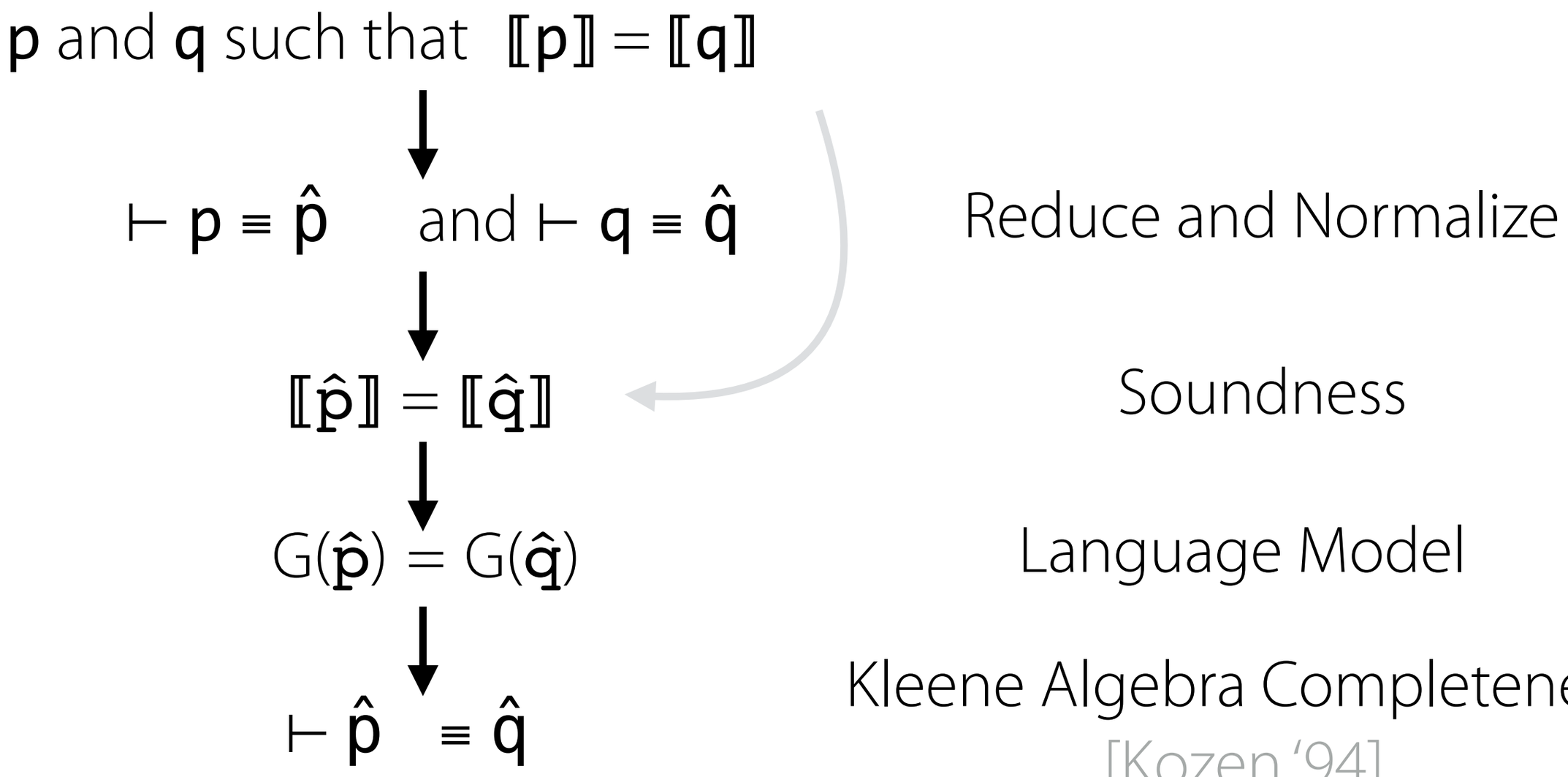
Completeness Proof



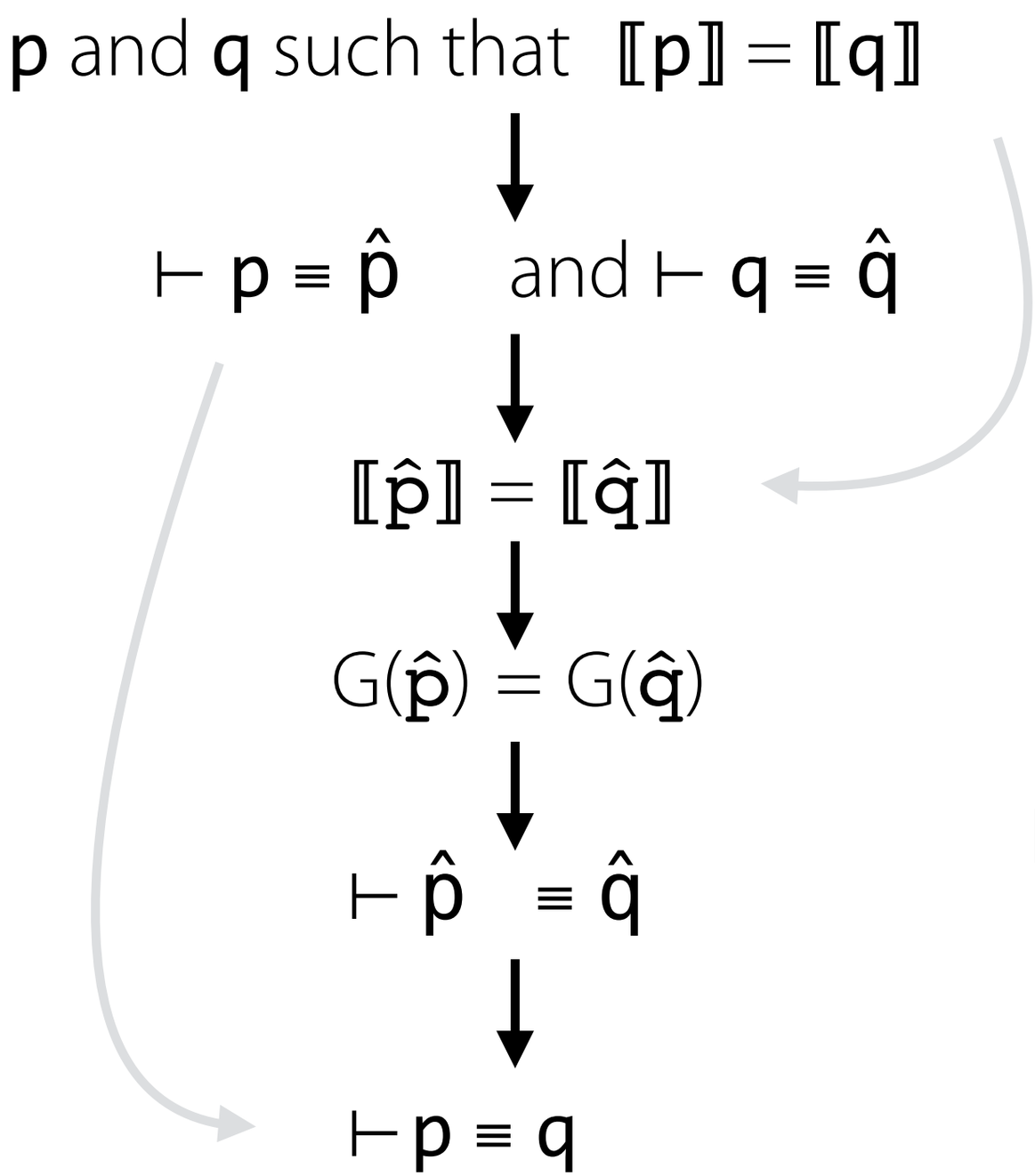
Completeness Proof



Completeness Proof



Completeness Proof



Reduce and Normalize

Soundness

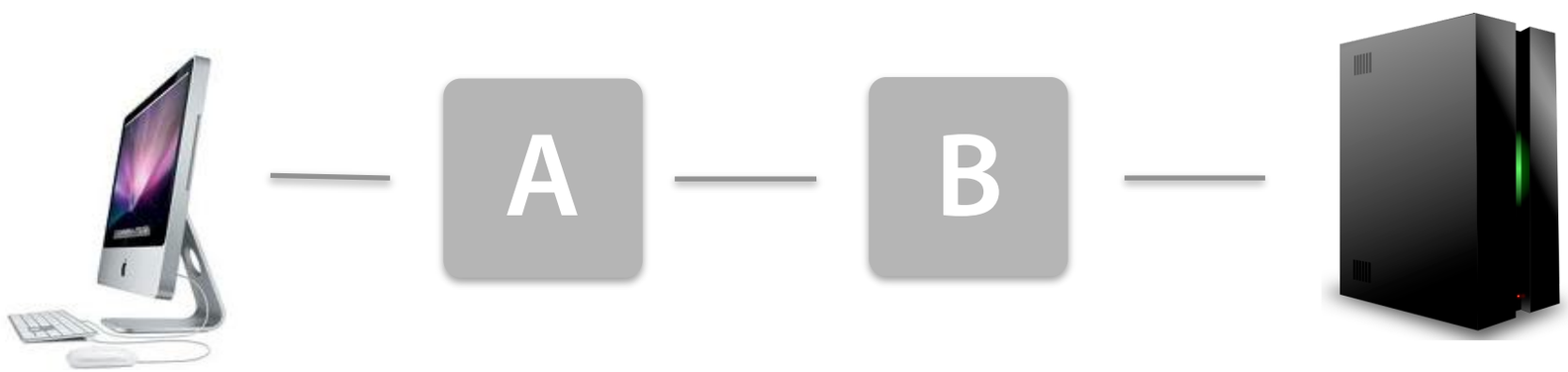
Language Model

Kleene Algebra Completeness
[Kozen '94]

Transitivity

Application: Optimization

Given a program and a topology:

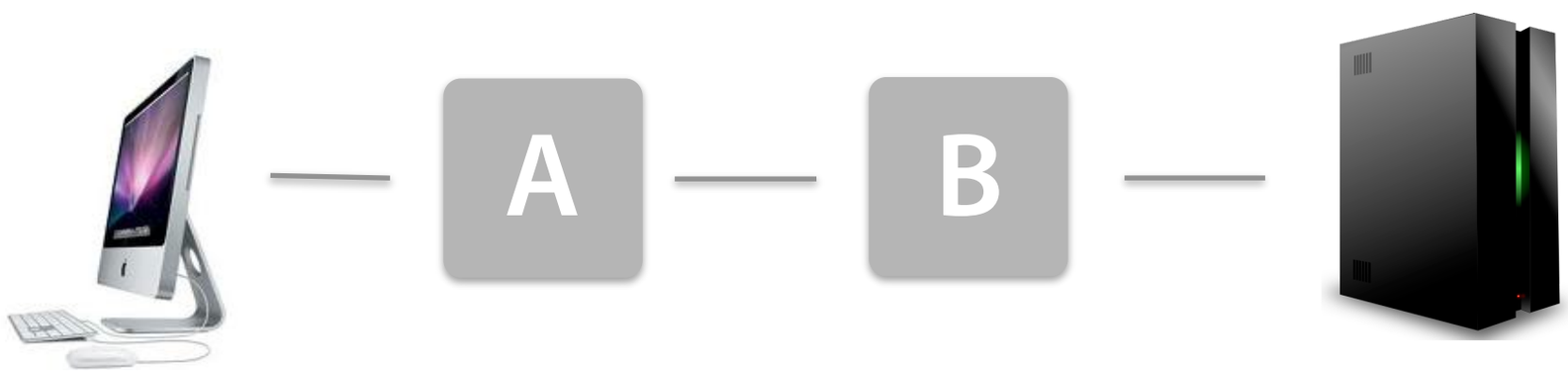


Want to be able to answer questions like:

> "Will my network behave the same if I put the firewall rules on A, or on switch B (or both)?"

Application: Optimization

Given a program and a topology:



Want to be able to answer questions like:

> "Will my network behave the same if I put the firewall rules on A, or on switch B (or both)?"

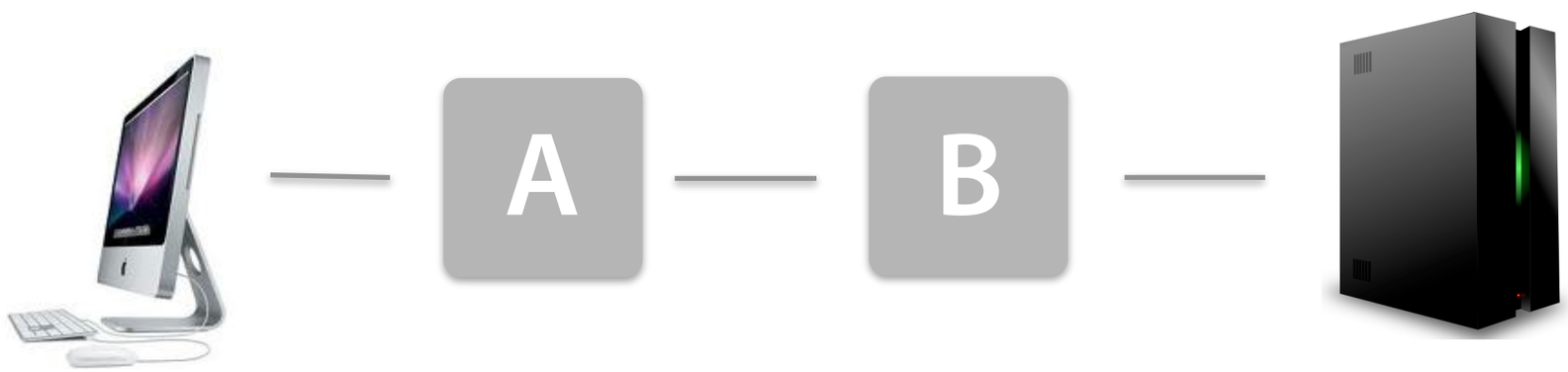
Formally, does the following equivalence hold?

$$(sw = A \bullet \underline{firewall} \bullet routing) + (sw = B \bullet routing)$$

$$\equiv$$

$$(sw = A \bullet routing) + (sw = B \bullet \underline{firewall} \bullet routing)$$

Code Motion Proof

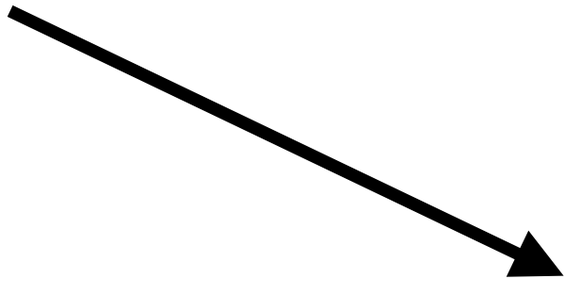


$in \cdot \text{SSH} \cdot (p_A \cdot t)^* \cdot out$
$\equiv$ { KAT-Invariant, definition p_A }
$in \cdot \text{SSH} \cdot ((a_A \cdot \neg\text{SSH} \cdot p + a_B \cdot p) \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KA-Seq-Dist-R }
$in \cdot \text{SSH} \cdot (a_A \cdot \neg\text{SSH} \cdot p \cdot t \cdot \text{SSH} + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KAT-Commute }
$in \cdot \text{SSH} \cdot (a_A \cdot \neg\text{SSH} \cdot \text{SSH} \cdot p \cdot t + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { BA-Contra }
$in \cdot \text{SSH} \cdot (a_A \cdot 0 \cdot p \cdot t + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KA-Seq-Zero/Zero-Seq, KA-Plus-Comm, KA-Plus-Zero }
$in \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KA-Unroll-L }
$in \cdot \text{SSH} \cdot (1 + (a_B \cdot p \cdot t \cdot \text{SSH}) \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^*) \cdot out$
$\equiv$ { KA-Seq-Dist-L, KA-Seq-Dist-R, definition out }
$in \cdot \text{SSH} \cdot a_B \cdot a_2 +$
$in \cdot \text{SSH} \cdot a_B \cdot p \cdot t \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot a_B \cdot a_2$
$\equiv$ { KAT-Commute }
$in \cdot a_B \cdot \text{SSH} \cdot a_2 +$
$in \cdot a_B \cdot \text{SSH} \cdot p \cdot t \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot a_B \cdot a_2$
$\equiv$ { Lemma 1 }
$0 + 0$
$\equiv$ { KA-Plus-Idem }
0

$\equiv$ { KA-Plus-Idem }
$0 + 0$
$\equiv$ { Lemma 1, Lemma 2 }
$in \cdot a_B \cdot \text{SSH} \cdot a_2 +$
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot p \cdot \text{SSH} \cdot a_A \cdot t \cdot out$
$\equiv$ { KAT-Commute, definition out }
$in \cdot \text{SSH} \cdot out +$
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot a_A \cdot p \cdot t \cdot \text{SSH} \cdot out$
$\equiv$ { KA-Seq-Dist-L, KA-Seq-Dist-R }
$in \cdot \text{SSH} \cdot (1 + (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot (a_A \cdot p \cdot t \cdot \text{SSH})) \cdot out$
$\equiv$ { KA-Unroll-R }
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KA-Seq-Zero/Zero-Seq, KA-Plus-Zero }
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot 0 \cdot p \cdot t)^* \cdot out$
$\equiv$ { BA-Contra }
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot \neg\text{SSH} \cdot \text{SSH} \cdot p \cdot t)^* \cdot out$
$\equiv$ { KAT-Commute }
$in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot \neg\text{SSH} \cdot p \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KA-Seq-Dist-R }
$in \cdot \text{SSH} \cdot ((a_A \cdot p + a_B \cdot \neg\text{SSH} \cdot p) \cdot t \cdot \text{SSH})^* \cdot out$
$\equiv$ { KAT-Invariant, definition p_B }
$in \cdot \text{SSH} \cdot (p_B \cdot t)^* \cdot out$

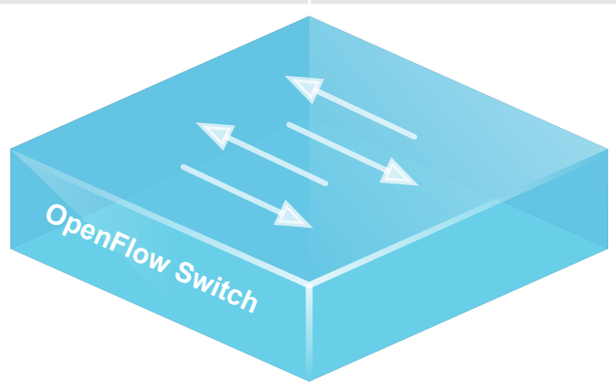
Application: Compiler Validation

```
!(type=SSH)·(pt=1·pt←2 + pt=2·pt←1)
```



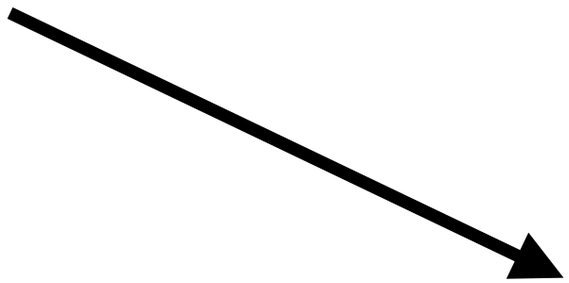
OpenFlow rule table

Pattern	Action
type=SSH	Drop
port=1	Output 2
port=2	Output 1
*	Drop



Application: Compiler Validation

```
!(type=SSH)·(pt=1·pt←2 + pt=2·pt←1)
```

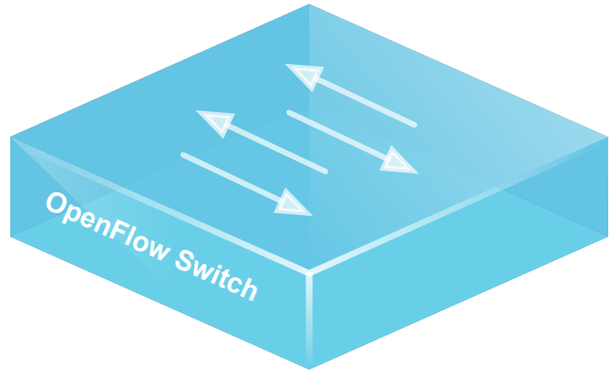


```
type=SSH·0 +
!type=SSH · pt=1·pt←2+
!type=SSH · !pt=1 · pt=2·pt←1+
!type=SSH · !pt=1 · !pt=2 · 0
```

OpenFlow rule table

Pattern	Action
type=SSH	Drop
port=1	Output 2
port=2	Output 1
*	Drop

Theorem: p ≡ compile(p)



Operational Semantics

NetKAT Automata

A *NetKAT automaton* $M = (S, s_0, \varepsilon, \delta)$ is a tuple where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \to \text{Packet} \to \text{Packet Set}$ is the observation function
- $\delta \in S \to \text{Packet} \to (\text{State} * \text{Packet}) \text{ Set}$ is the continuation function

NetKAT Automata

A *NetKAT automaton* $M = (S, s_0, \varepsilon, \delta)$ is a tuple where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \to \text{Packet} \to \text{Packet Set}$ is the observation function
- $\delta \in S \to \text{Packet} \to (\text{State} * \text{Packet}) \text{ Set}$ is the continuation function

Input "strings": $pkt_{in} \cdot pkt_1 \cdot \textbf{dup} \cdot \ldots \cdot \textbf{dup} \cdot pkt_n \; \textbf{dup} \cdot pkt_{out}$

NetKAT Automata

A *NetKAT automaton* $M = (S, s_0, \varepsilon, \delta)$ is a tuple where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \to \text{Packet} \to \text{Packet Set}$ is the observation function
- $\delta \in S \to \text{Packet} \to (\text{State} * \text{Packet}) \text{ Set}$ is the continuation function

Input "strings": $\text{pkt}_{in} \bullet \text{pkt}_1 \bullet \textbf{dup} \bullet \ldots \bullet \textbf{dup} \bullet \text{pkt}_n \textbf{ dup} \bullet \text{pkt}_{out}$

A NetKAT automaton M *accepts* an input in state s if:

- accept s $(\text{pkt}_{in} \bullet \text{pkt}_{out}) \Leftrightarrow \text{pkt}_{out} \in \varepsilon\ s\ \text{pat}_{in}$
- accept s $(\text{pkt}_{in} \bullet \text{pkt}_1 \bullet \textbf{dup} \bullet w) \Leftrightarrow$

  $\exists\ s'.\ (\text{pkt}_1, s') \in \delta\ s\ \text{pkt}_{in}$ and accept s' $(\text{pkt}_1 \bullet w)$

NetKAT Derivatives

E(pol) ∈ Pol

D(pol) ∈ (Pol * L * Pol) Set

NetKAT Derivatives

E(pol) ∈ Pol

D(pol) ∈ (Pol * L * Pol) Set

Labels, one per occurrence of ***dup***

NetKAT Derivatives

E(pol) ∈ Pol

D(pol) ∈ (Pol * L * Pol) Set

Labels, one per occurrence of ***dup***

Observation map

E(**false**) = **false**

E(**true**) = **true**

E(f **=** n) = f **=** n

E(f **:=** n) = f **:=** n

E(**!**pol) = **!**pol

E(**dup**^{l}) = **false**

E(pol_1 **+** pol_2) = E(pol_1) **+** E(pol_2)

E(pol_1 • pol_2) = E(pol_1) • E(pol_2)

E(pol*) = E(pol)*

NetKAT Derivatives

E(pol) ∈ Pol

D(pol) ∈ (Pol * L * Pol) set

Continuation map

D(**false**) = {}

D(**true**) = {}

D(f**=**n) = {}

D(f**:=**n) = {}

D(**!**pol) = {}

D(**dup**^{l}) = { (**true**, l, **true**) }

D(pol_1 **+** pol_2) = D(pol_1) **+** D(pol_2)

D(pol_1 • pol_2) = D(pol_1) • pol_2 + E(pol_1) • D(pol_2)

D(pol*) = E(pol)* • D(pol) • pol*

NetKAT Derivatives

E(pol) ∈ Pol

D(pol) ∈ (Pol * L * Pol) set

Continuation map

D(**false**) = {}

D(f **:=** n) = {}

D(**!**pol) = {}

D(**dup**^{l}) = { (**true**, l, **true**) }

D(pol_1 **+** pol_2) = D(pol_1) **+** D(pol_2)

D(pol_1 • pol_2) = D(pol_1) • pol_2 + E(pol_1) • D(pol_2)

D(pol*) = E(pol)* • D(pol) • pol*

Theorem [POPL '15]: every NetKAT program has a finite number of derivatives (modulo ACI)

Program Automaton

Notation: Write k_l for the unique continuation of dup^l

The *program automaton* M for a program pol is a tuple where:

- S is the set of labels occurring in pol, plus the initial label 0
- $s_0 \triangleq 0$ is the start state,
- $\varepsilon\ l\ pkt = \{\ pkt' \mid \langle pkt' \rangle \in [\![E(k_l)]\!] \langle pkt \rangle\ \}$
- $\delta\ l\ pkt = \{\ (pkt', l') \mid (d, l', k) \in D(k_l) \text{ and } \langle pkt' \rangle \in [\![d]\!] \langle pkt \rangle\ \}$

Program Automaton

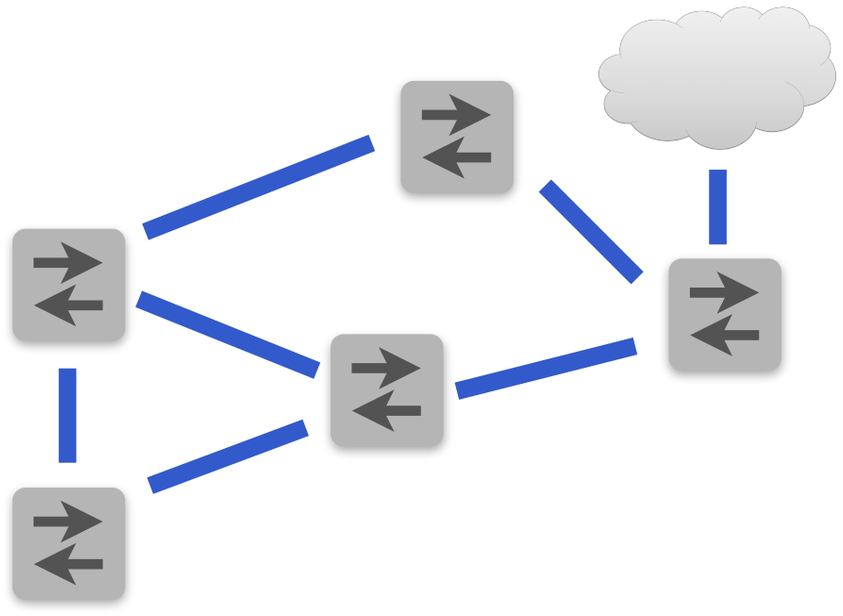
Lemma: For every label l occurring in a NetKAT program pol there exist unique programs d and k such that (d,l,k) ∈ D(pol).

Notation: Write k_l for the unique continuation of dup^l

The *program automaton* M for a program pol is a tuple where:

- S is the set of labels occurring in pol, plus the initial label 0
- $s_0 \triangleq 0$ is the start state,
- $\varepsilon\ l\ pkt = \{\ pkt' \mid \langle pkt' \rangle \in [\![E(k_l)]\!] \langle pkt \rangle\ \}$
- $\delta\ l\ pkt = \{\ (pkt', l') \mid (d, l', k) \in D(k_l) \text{ and } \langle pkt' \rangle \in [\![d]\!] \langle pkt \rangle\ \}$

Application: Loop Freedom



Can also exploit automata representations to *efficiently* check whether a network is free of forwarding loops...

Intuition: $\forall \alpha.\ in \bullet (p \bullet t)^* \bullet \alpha \bullet (p \bullet t)^* \bullet \alpha \equiv$ **false**

- $\forall$ pkt, pkt′. pkt′ $\in [\![E(\Phi(in \bullet (p \bullet t)^*))]\!]$ pkt
- Check whether pkt′ $\in [\![E(\Phi(p \bullet t)^*))]\!]$ pkt′

Limits and Open Questions

Open Questions

The Horn theory of a KA/T is undecidable in general.
What theories are useful in practice?

How can we extend KA/T to model concurrency?

- Concurrent KA [CONCUR '09]
- KA with Observations [CONCUR '19]
- Concurrent KA with Observations [FoSSaCS '19]

Come to Tobias's POPL talk on Wednesday at 4:18pm!

Guarded Kleene Algebra with Tests

Reading

- Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. **NetKAT: Semantic Foundations for Networks**. In *ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages (POPL),* January 2014.
- Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. **A Fast Compiler for NetKAT.** *In ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2015.
- Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. **A Coalgebraic Decision Procedure for NetKAT.** In ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India, January 2015.