The Next 700 Network Programming Languages

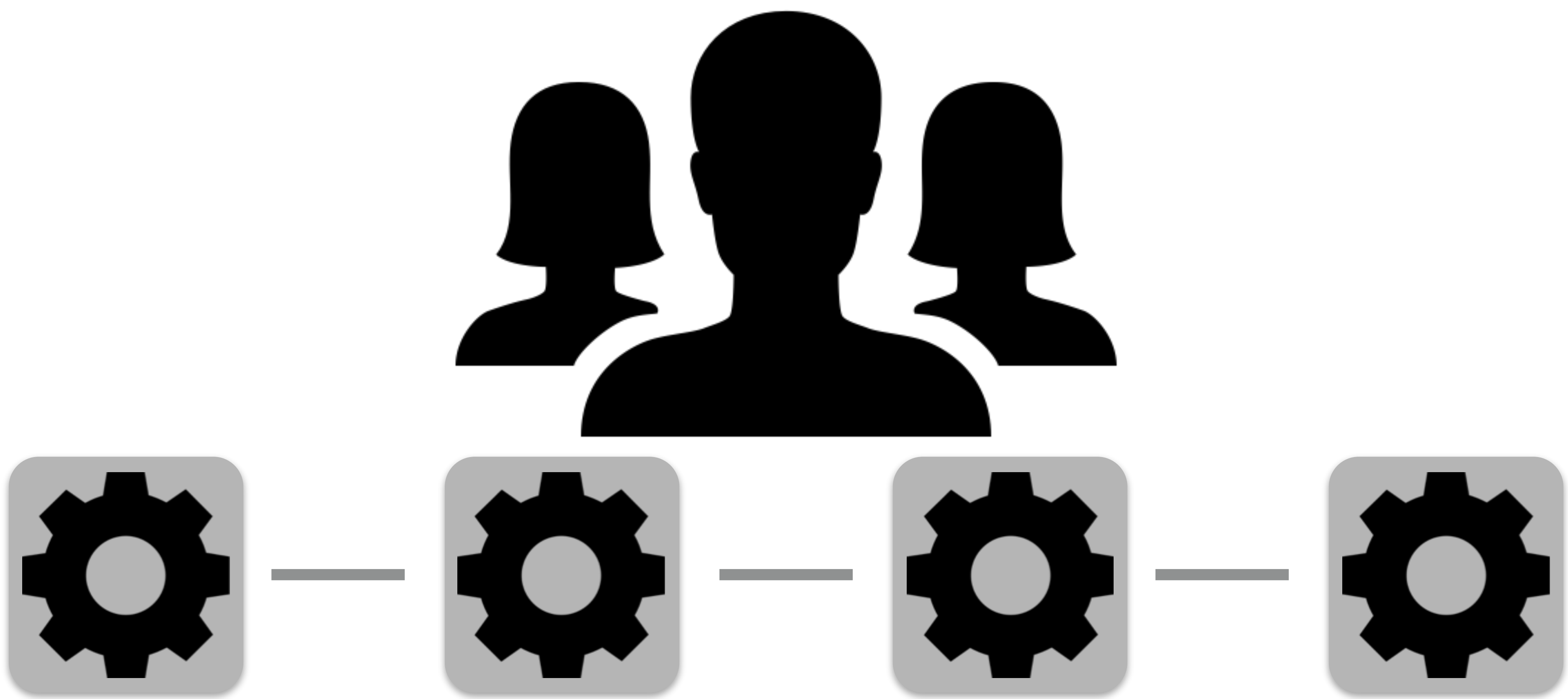
Nate Foster
Cornell University

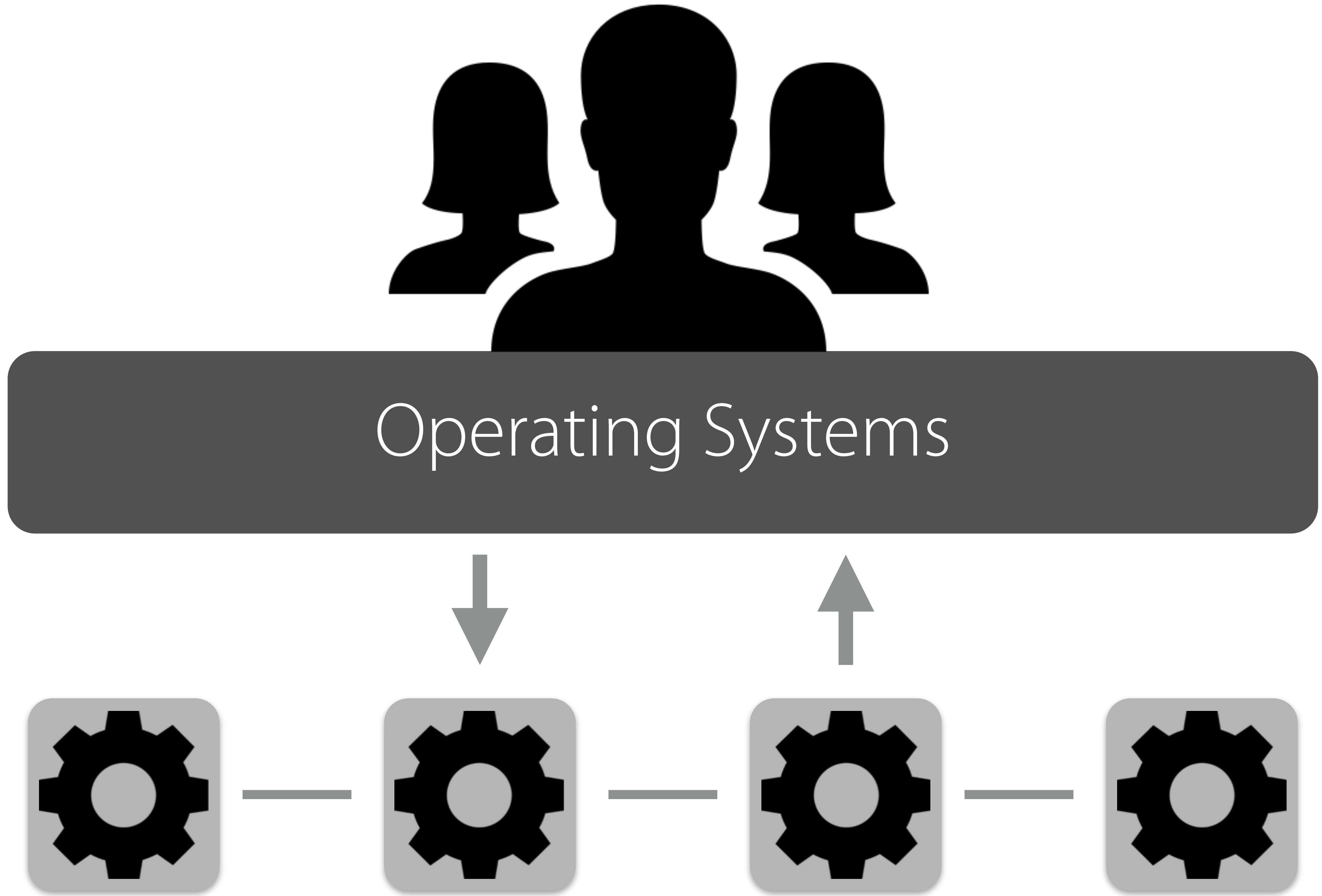
ACM SIGCOMM NetPL '16

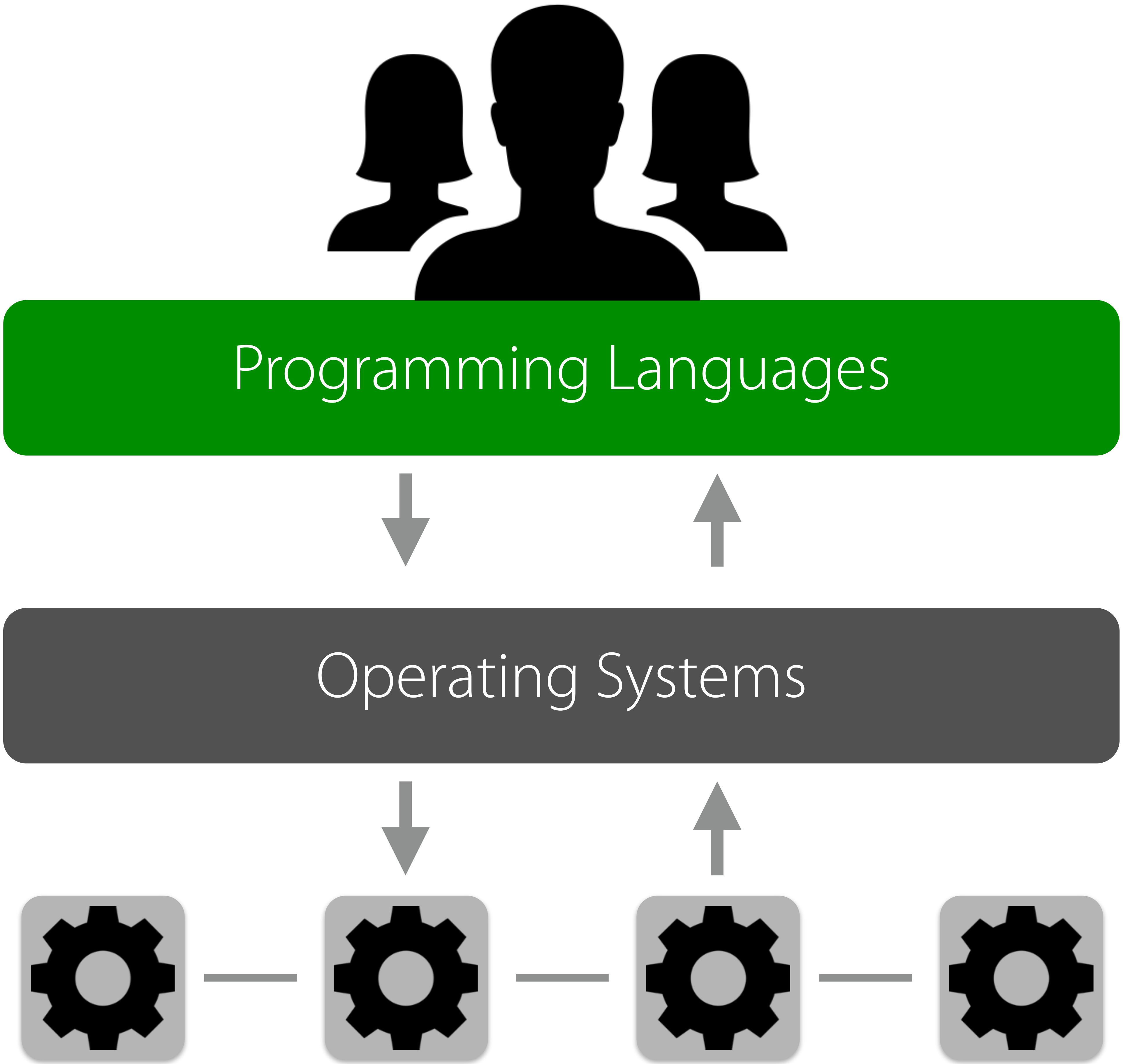
NetPL

workshop(Networking & Programming Languages)

{ Brazil, 2016 }




Operating Systems


Programming Languages
Operating Systems

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncracies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$	$f(b+2c)$
where $x = b + 2c$	**where** $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

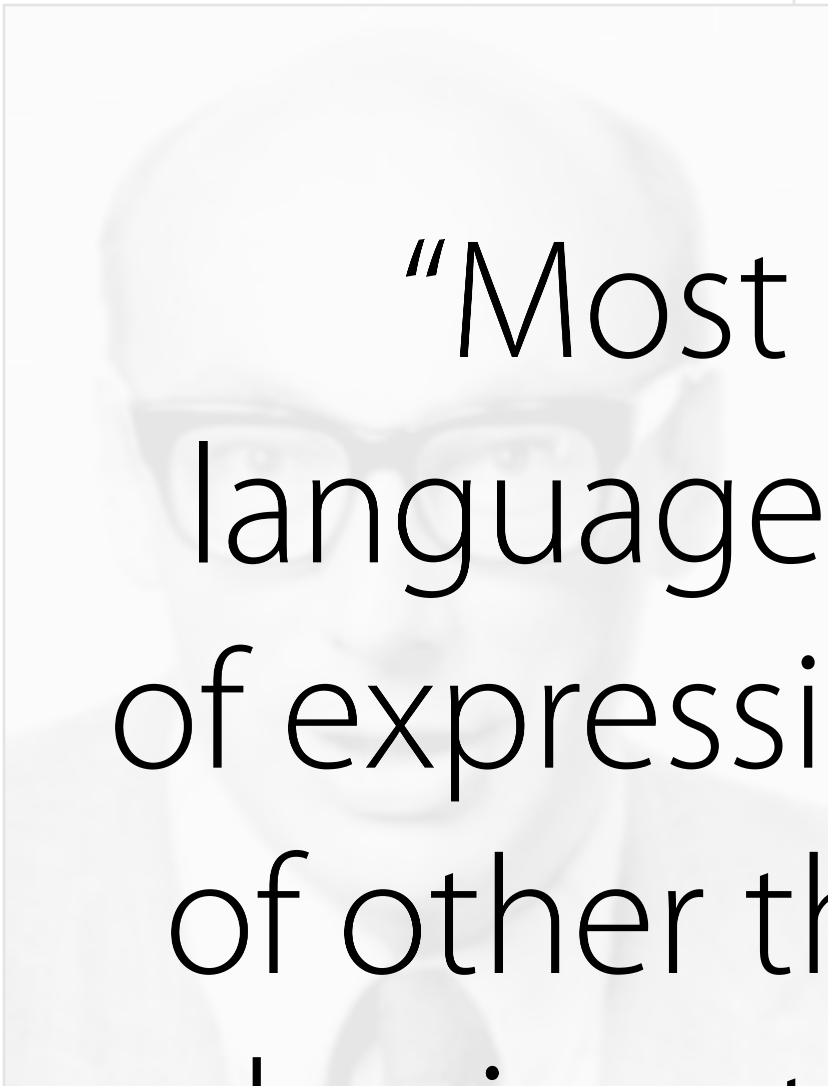
$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$

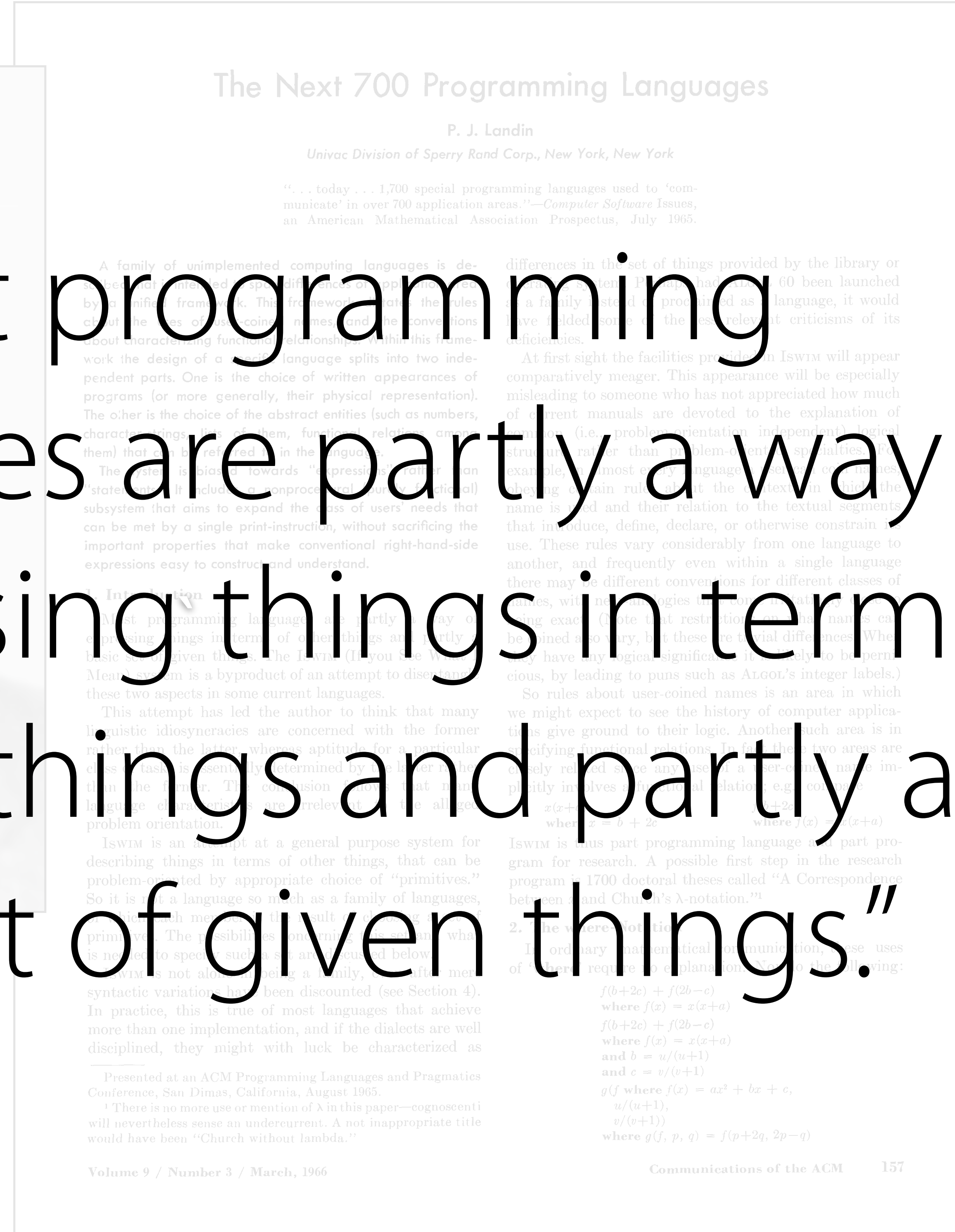
$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$

$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(u+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without lambda."





“Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.”

NetKAT Language

A domain-specific language for network programming that offers...

- Boolean predicates
- Regular expressions
- Modular composition
- Network-wide visibility and control

... embedded within a language with standard programming constructs (assignment, conditionals, loops, etc.)

NetKAT Language

A domain-specific language for network programming that offers...

- Boolean predicates *Packet classification!*
- Regular expressions
- Modular composition
- Network-wide visibility and control

... embedded within a language with standard programming constructs (assignment, conditionals, loops, etc.)

NetKAT Language

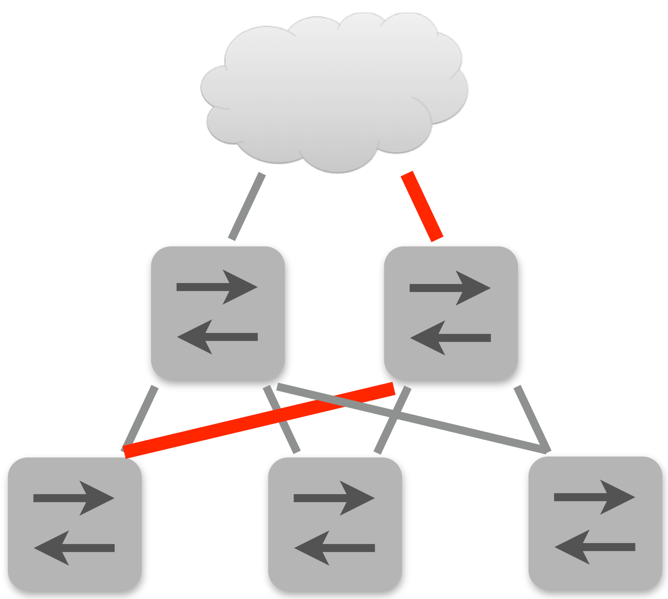
A domain-specific language for network programming that offers...

- Boolean predicates ← *Packet classification!*
- Regular expressions ← *Forwarding paths!*
- Modular composition
- Network-wide visibility and control

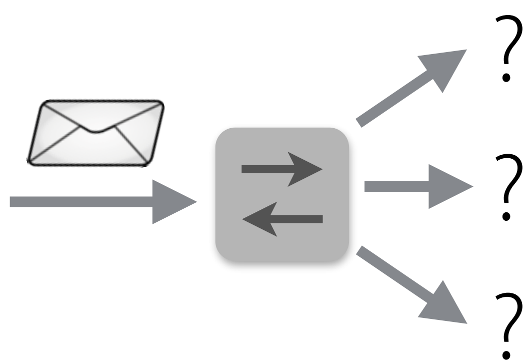
... embedded within a language with standard programming constructs (assignment, conditionals, loops, etc.)

NetKAT Design

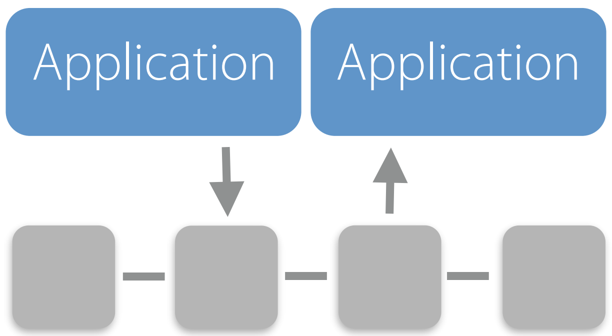
NetKAT Features



Network-wide Abstractions



Rich Packet Classification



Modular Composition

NetKAT Syntax

$$
\begin{aligned}
\text{pol} ::= \ & \textbf{false} \\
\mid\ & \textbf{true} \\
\mid\ & f \mathbin{\textbf{=}} n \\
\mid\ & f \mathbin{\textbf{:=}} n \\
\mid\ & \text{pol}_1 \mathbin{\textbf{+}} \text{pol}_2 \\
\mid\ & \text{pol}_1 \bullet \text{pol}_2 \\
\mid\ & !\text{pol} \\
\mid\ & \text{pol}^* \\
\mid\ & \textbf{dup}
\end{aligned}
$$

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
| $\text{pol}_1 \bullet \text{pol}_2$
| !pol
| pol*
| **dup**

$\left.\begin{array}{c}\text{Boolean Predicates} \\ + \\ \text{Regular Expressions}\end{array}\right\}$ KAT

\+

Packet Primitives

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

NetKAT Syntax

pol ::= **false**
| **true**
| $f = n$
| $f := n$
| $pol_1 + pol_2$
| $pol_1 \bullet pol_2$
| !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**

Boolean

$$\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2 \triangleq (b \bullet p_1) + (!b \bullet p_2)$$

$$\textbf{while}\ b\ \textbf{do}\ p \triangleq (b \bullet p)^* \bullet !b$$

$$p^+ \triangleq p \bullet p^*$$

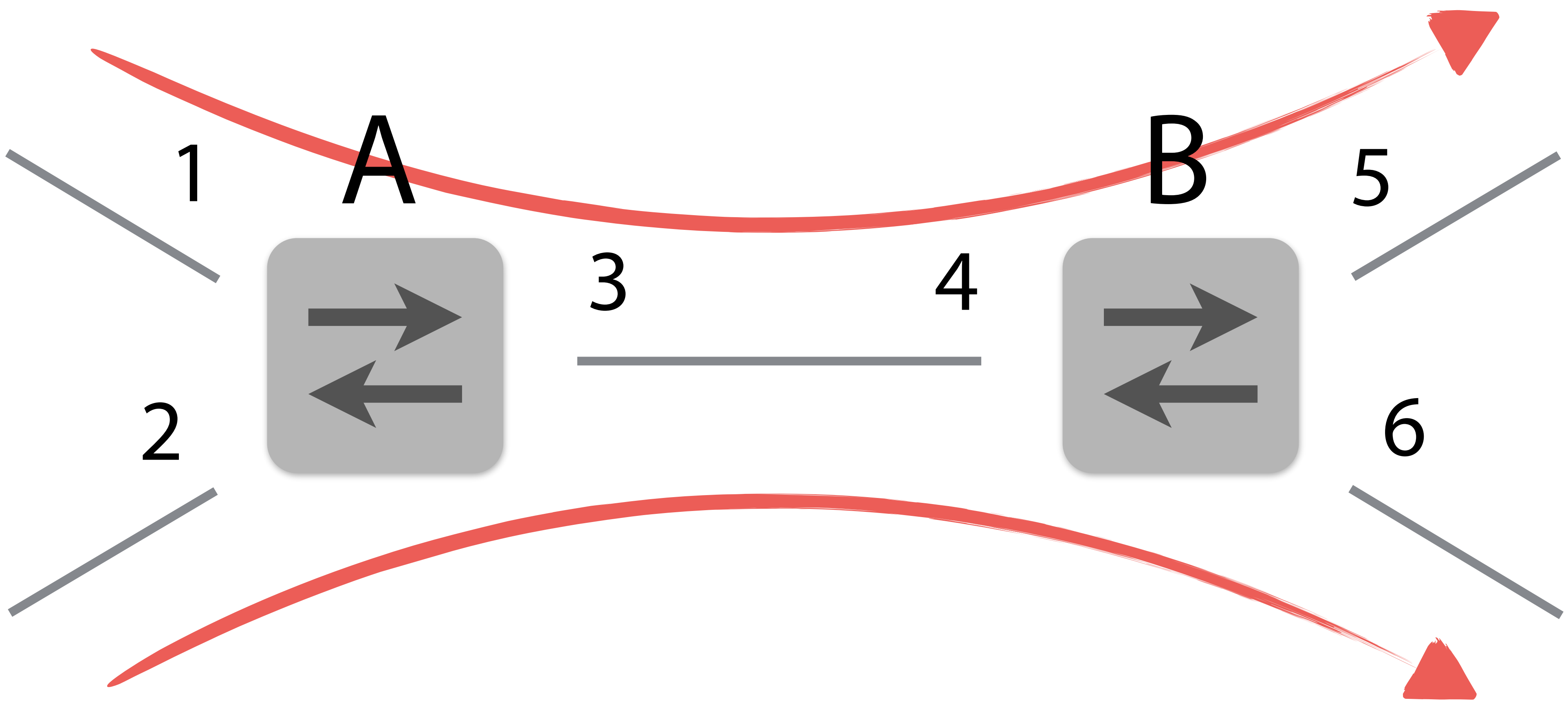
$$S \rightarrow S' \triangleq sw{=}S \bullet \textbf{dup} \bullet sw{:=}S' \bullet \textbf{dup}$$

Primitives

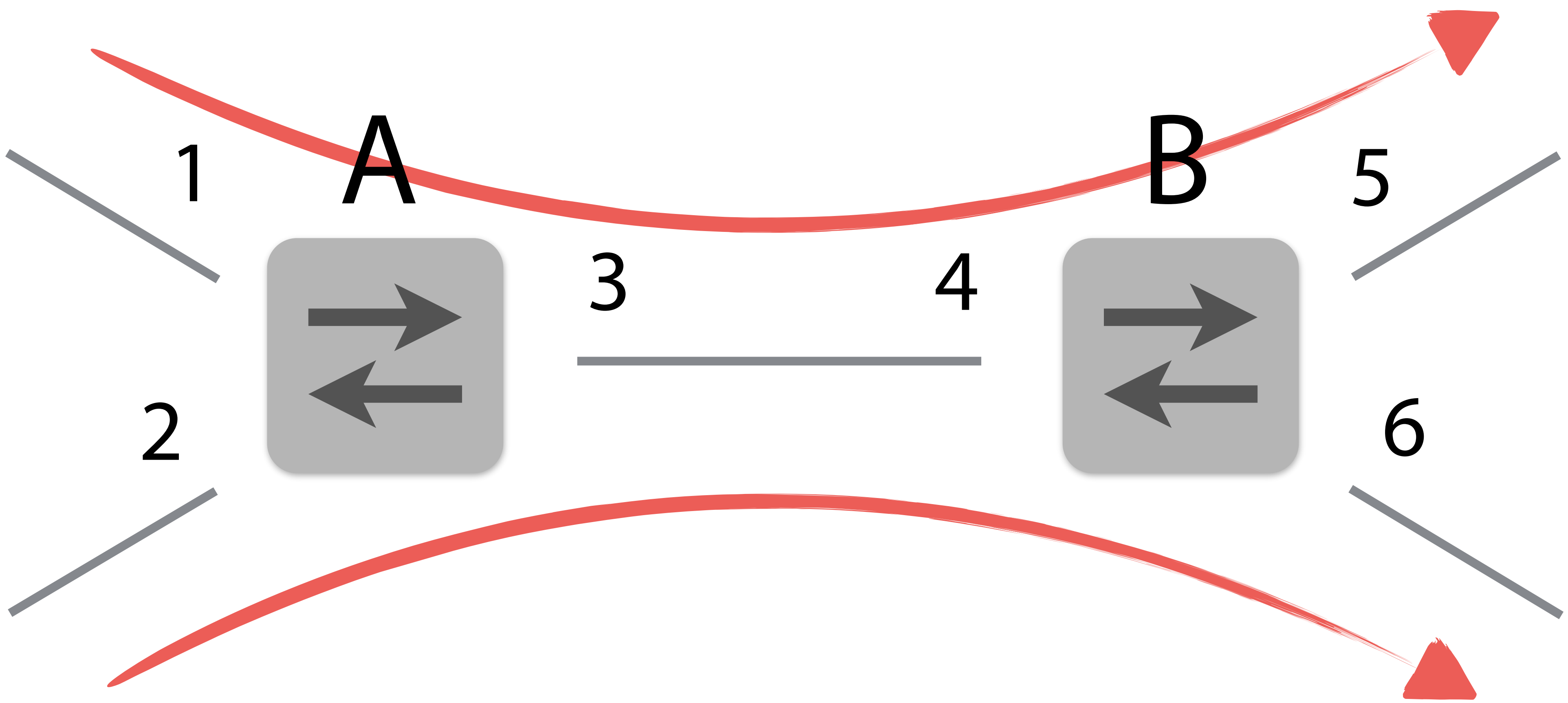
| **dup**

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

Example



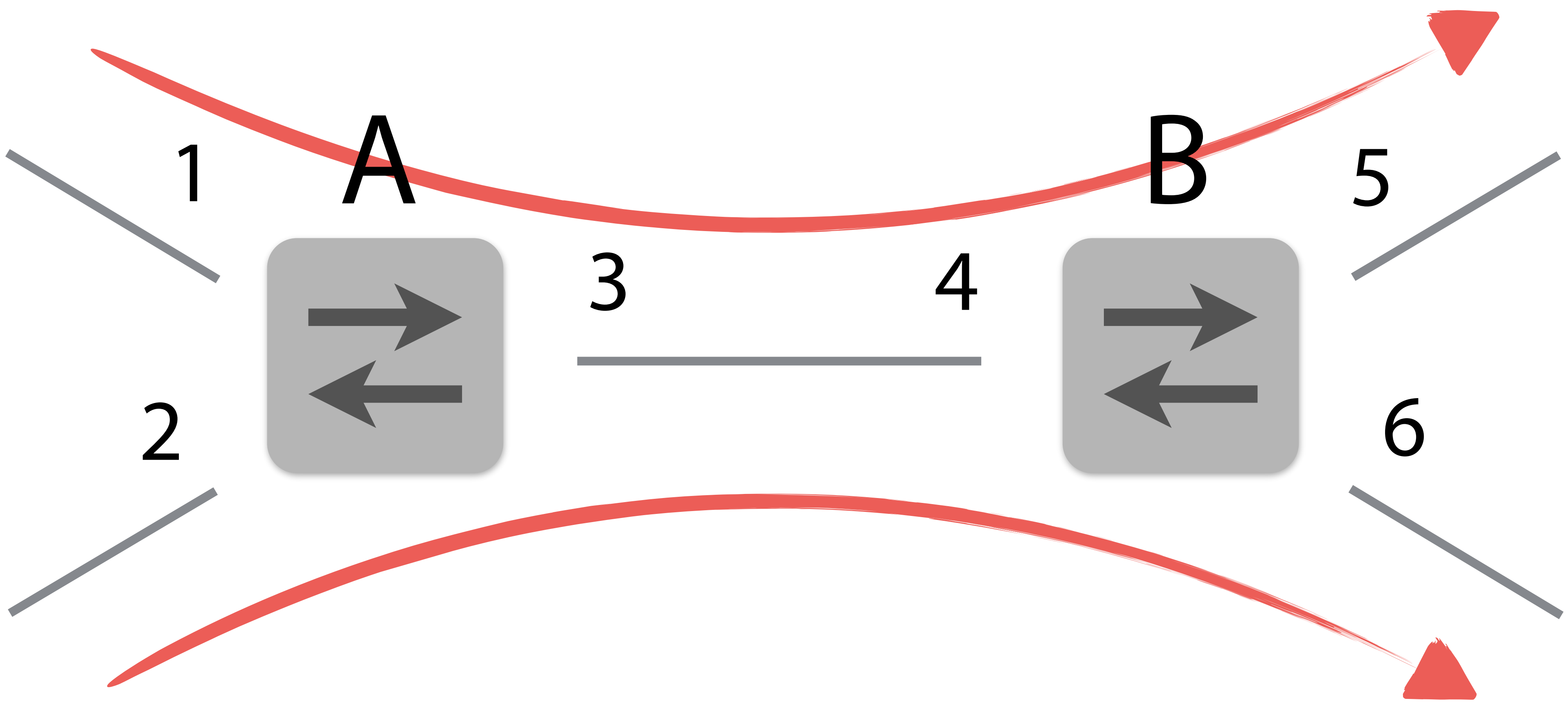
Local Program



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

tag=1 • port:=5
+
tag=2 • port:=6

Global Program

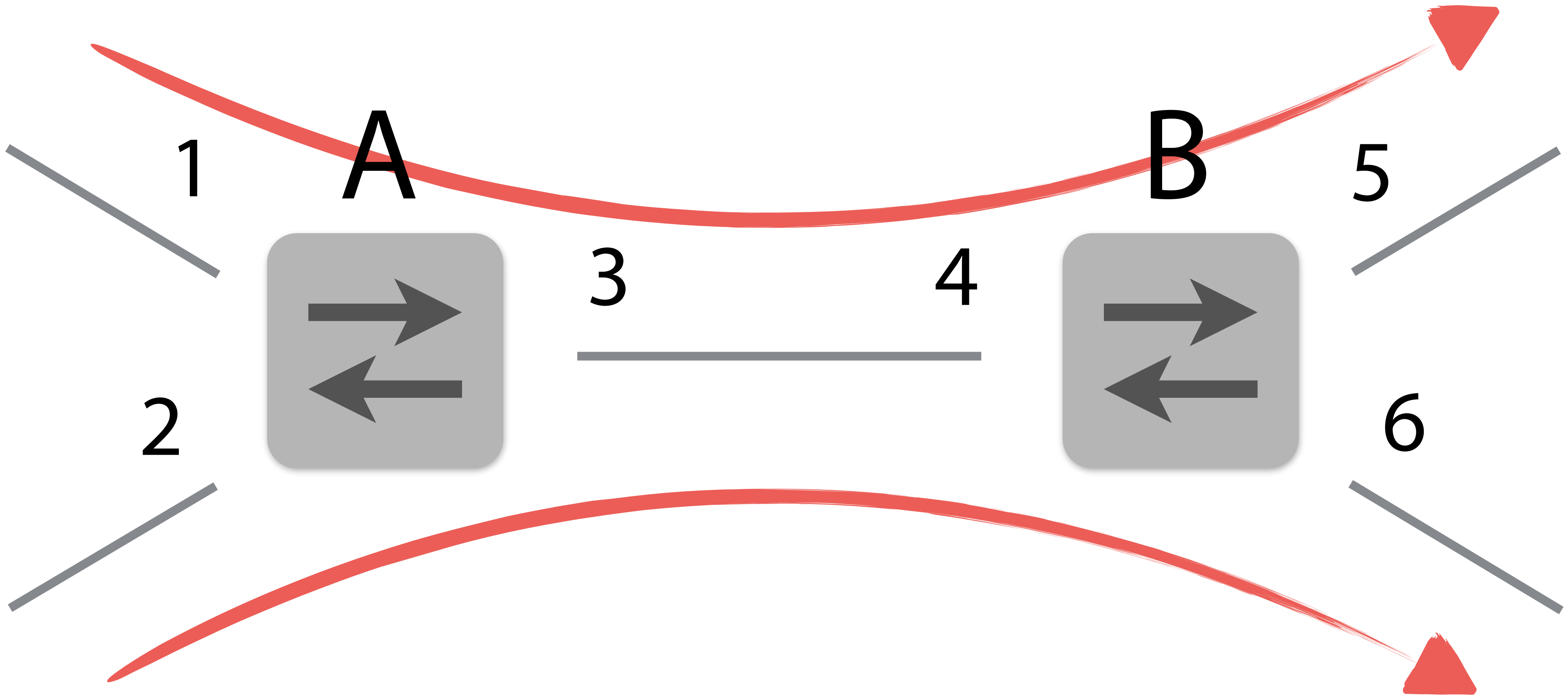


$$\mathrm{port}=1 \bullet \mathbf{A} \rightharpoonup \mathbf{B} \bullet \mathrm{port}:=5$$

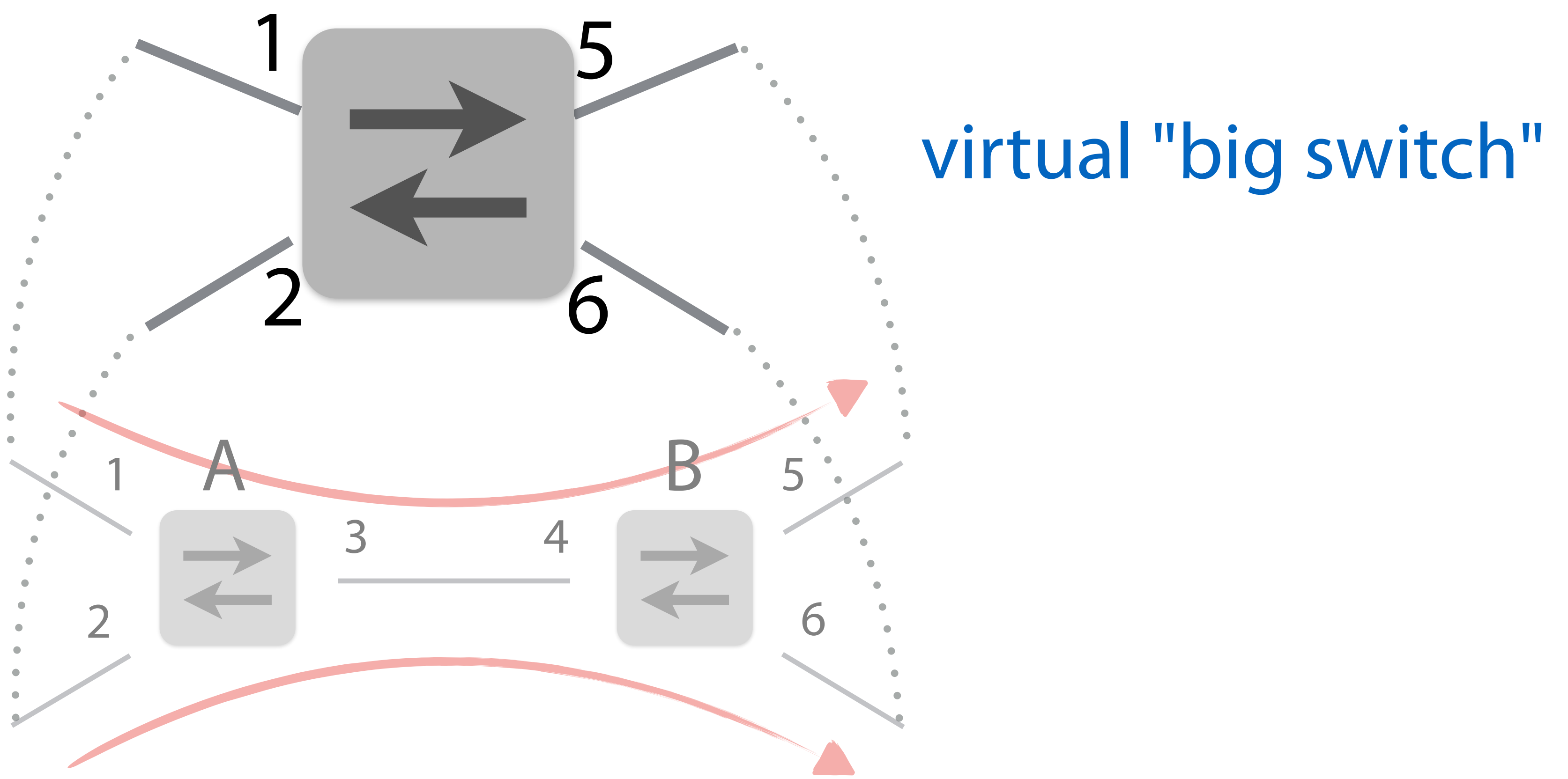
$$+$$

$$\mathrm{port}=2 \bullet \mathbf{A} \rightharpoonup \mathbf{B} \bullet \mathrm{port}:=6$$

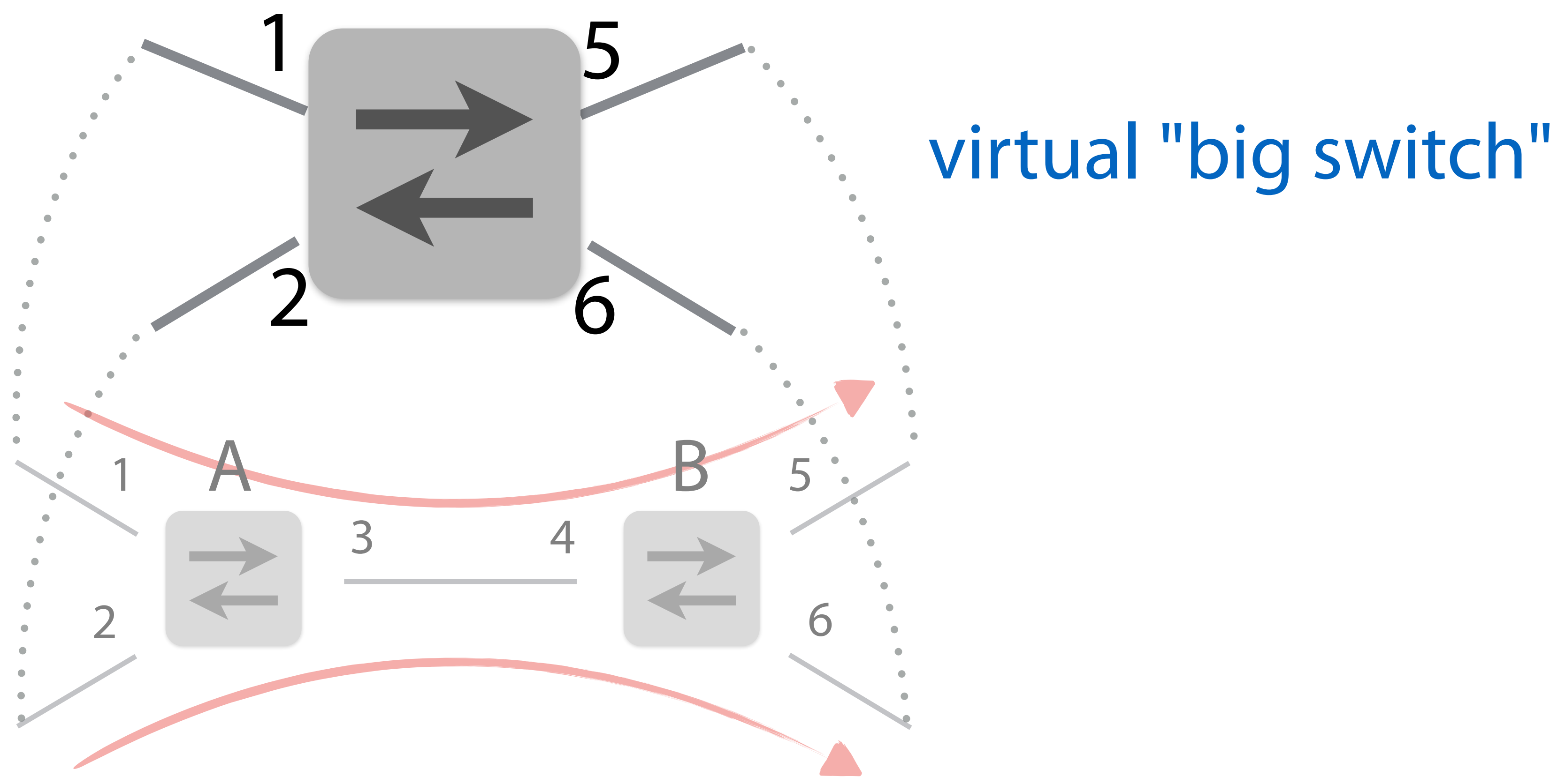
Virtual Program



Virtual Program

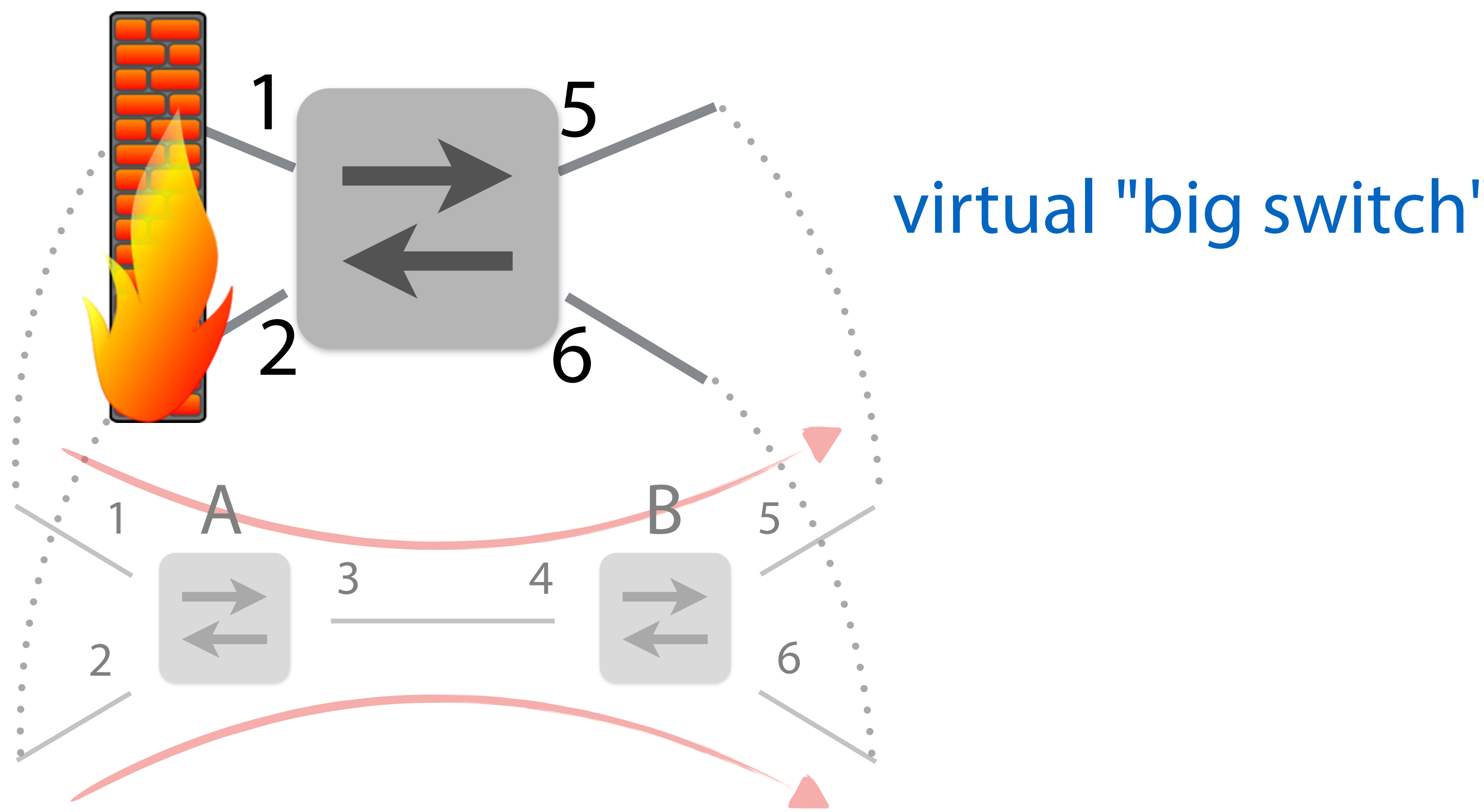


Virtual Program



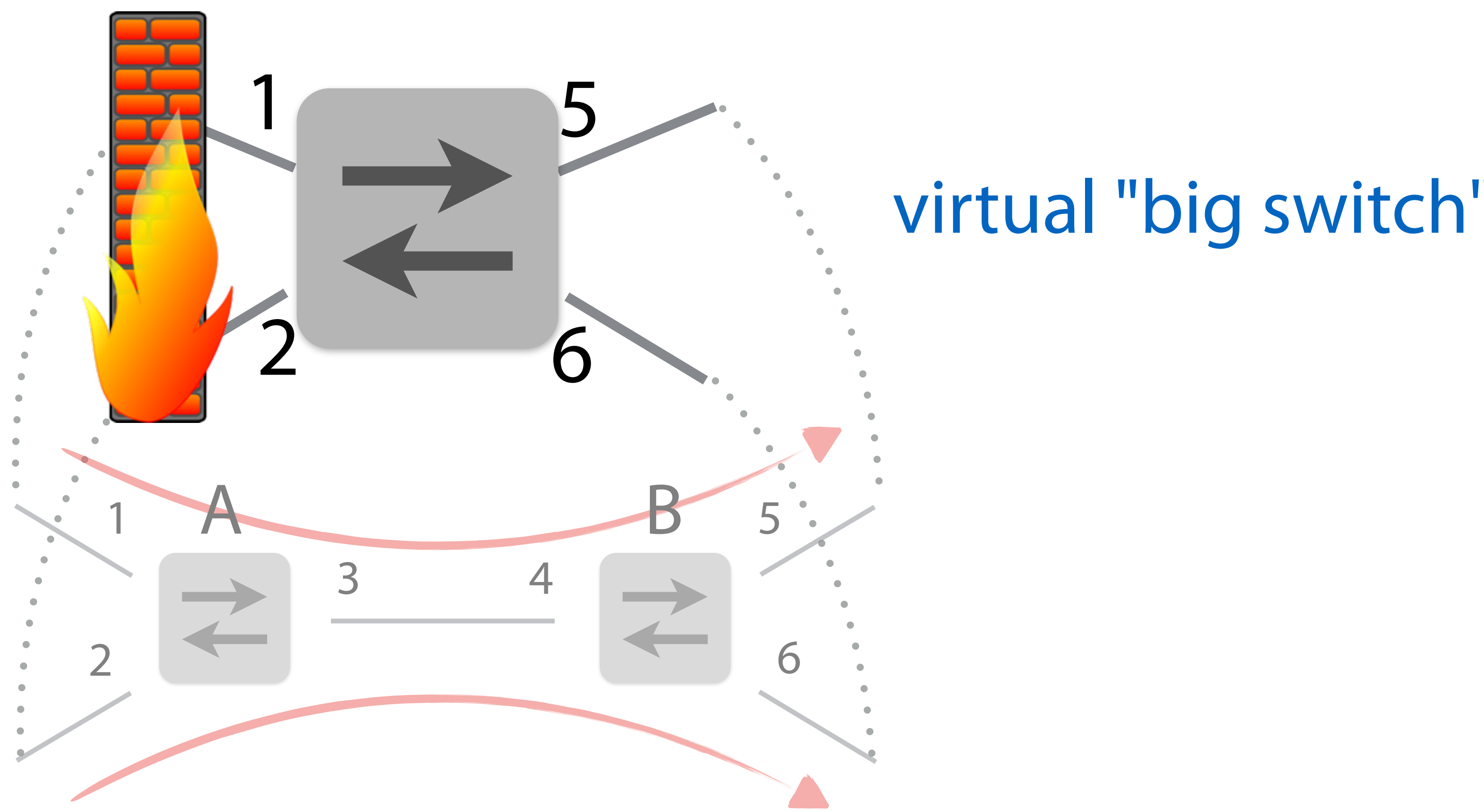
port=1 • port:=5
+
port=2 • port:=6

Virtual Program



port=1• port:=5
+
port=2• port:=6

Virtual Program



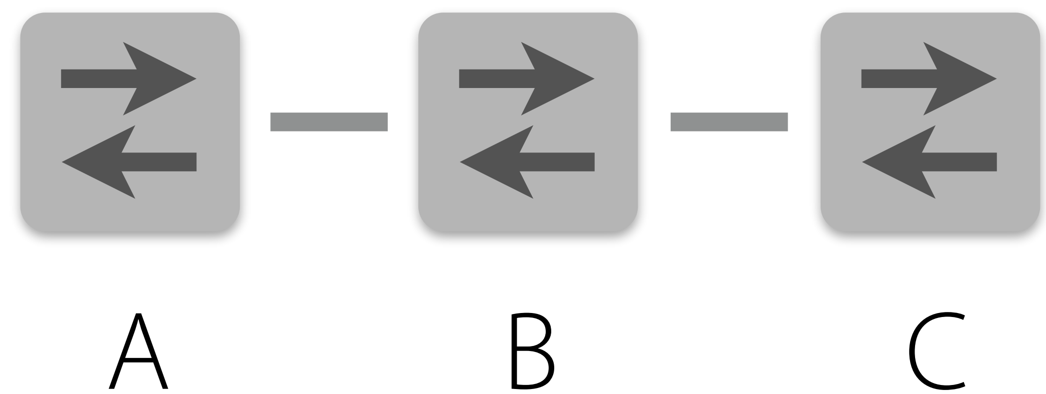
$!(\text{tcpSrcPort} = 22) \cdot \left(\begin{array}{c} \text{port}{=}1 \cdot \text{port}{:=}5 \\ + \\ \text{port}{=}2 \cdot \text{port}{:=}6 \end{array} \right)$

Encoding Networks

Switch forwarding tables and network topologies can be represented in NetKAT using straightforward encodings

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

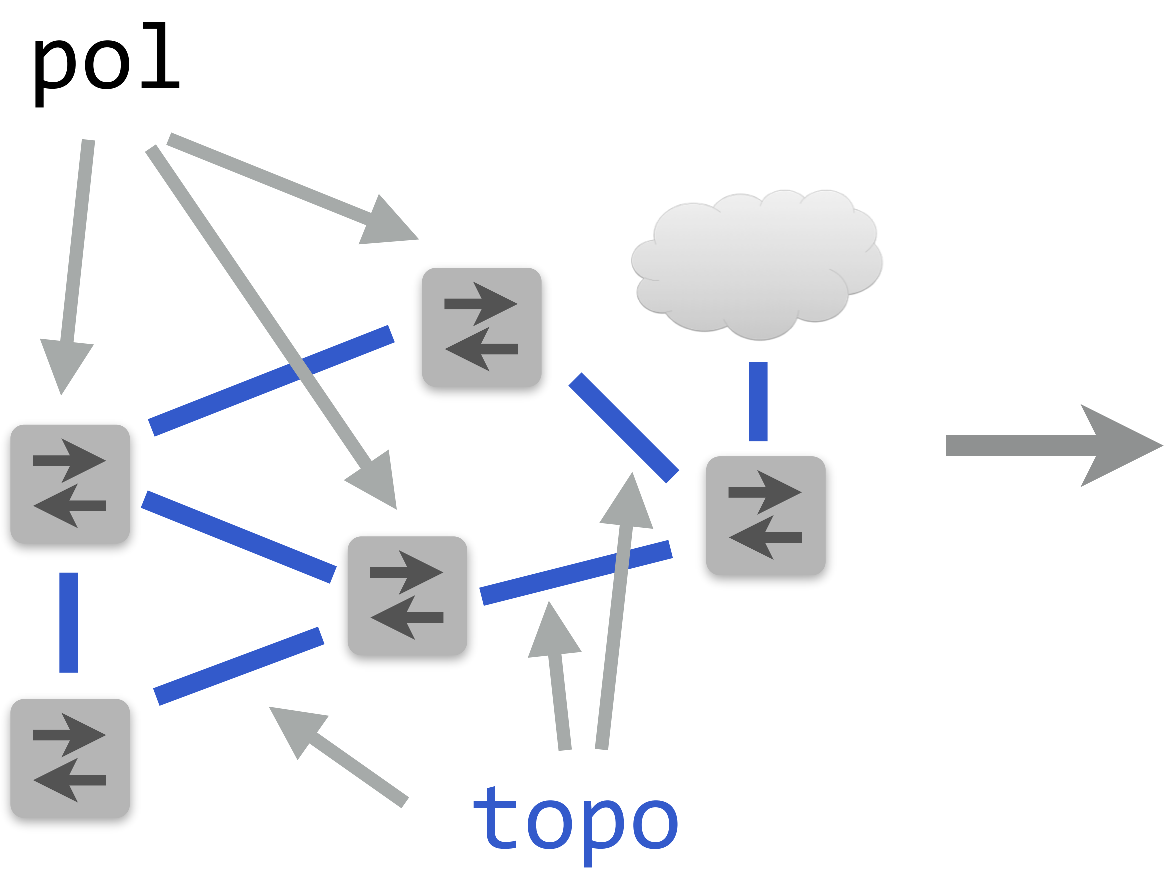
if dstport=22 **then false**
elsif srcip=10.0.0.1 **then** port := 1
else port := 2



$A \rightarrow B + B \rightarrow A + B \rightarrow C + C \rightarrow B$

Encoding Networks

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



(pol • topo)* • pol

NetKAT Semantics

NetKAT Semantics

Denotational

NetKAT Semantics

Denotational

Axiomatic

$\vdash p \equiv q$

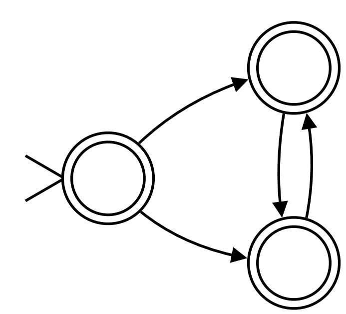
NetKAT Semantics

Denotational

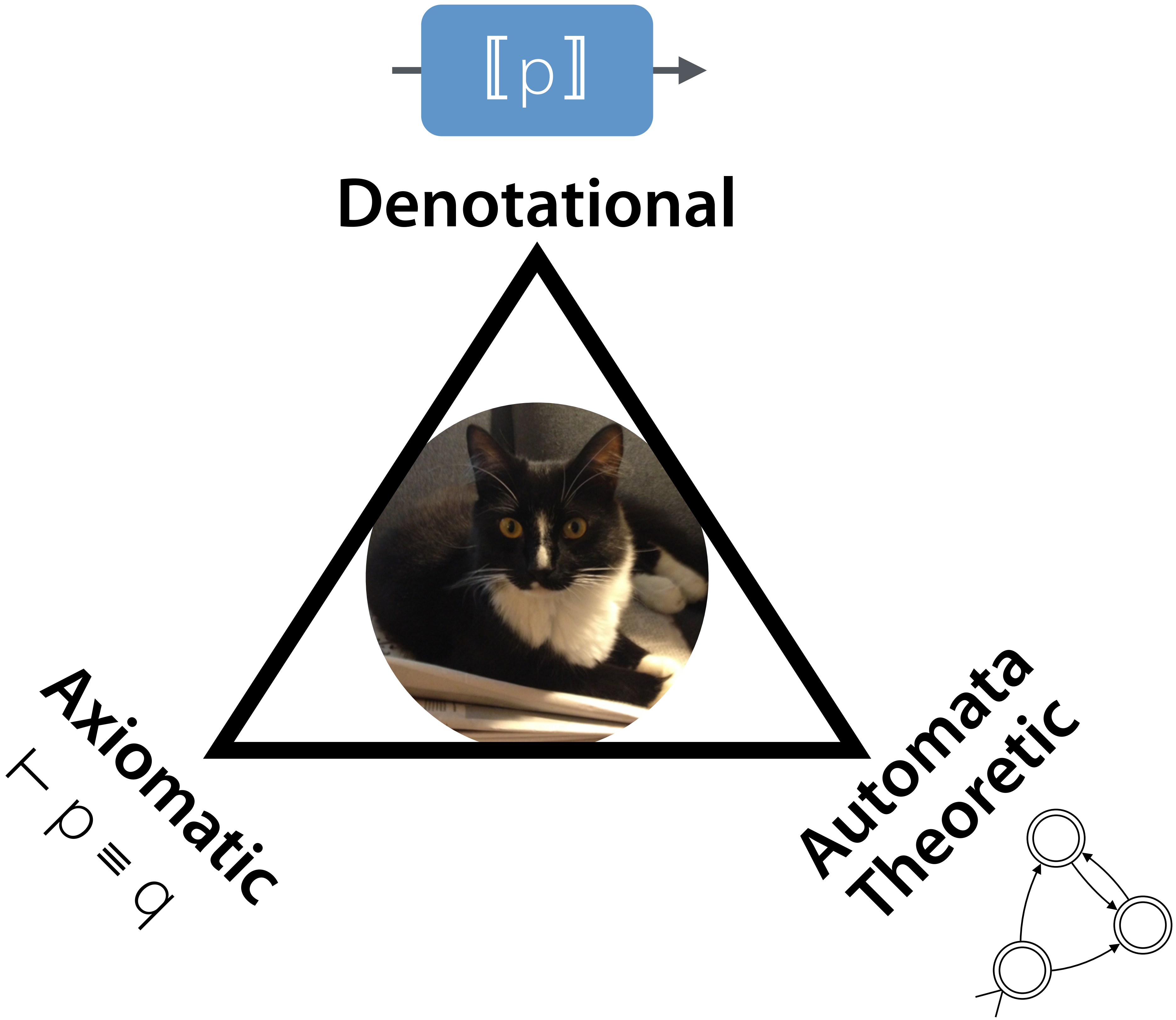
Axiomatic

$\vdash p \equiv q$

Automata Theoretic



NetKAT Semantics



NetKAT Semantics

$[\![p]\!]$

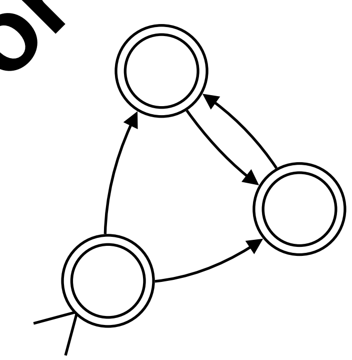
Denotational

Soundness+Completeness
[POPL '14]

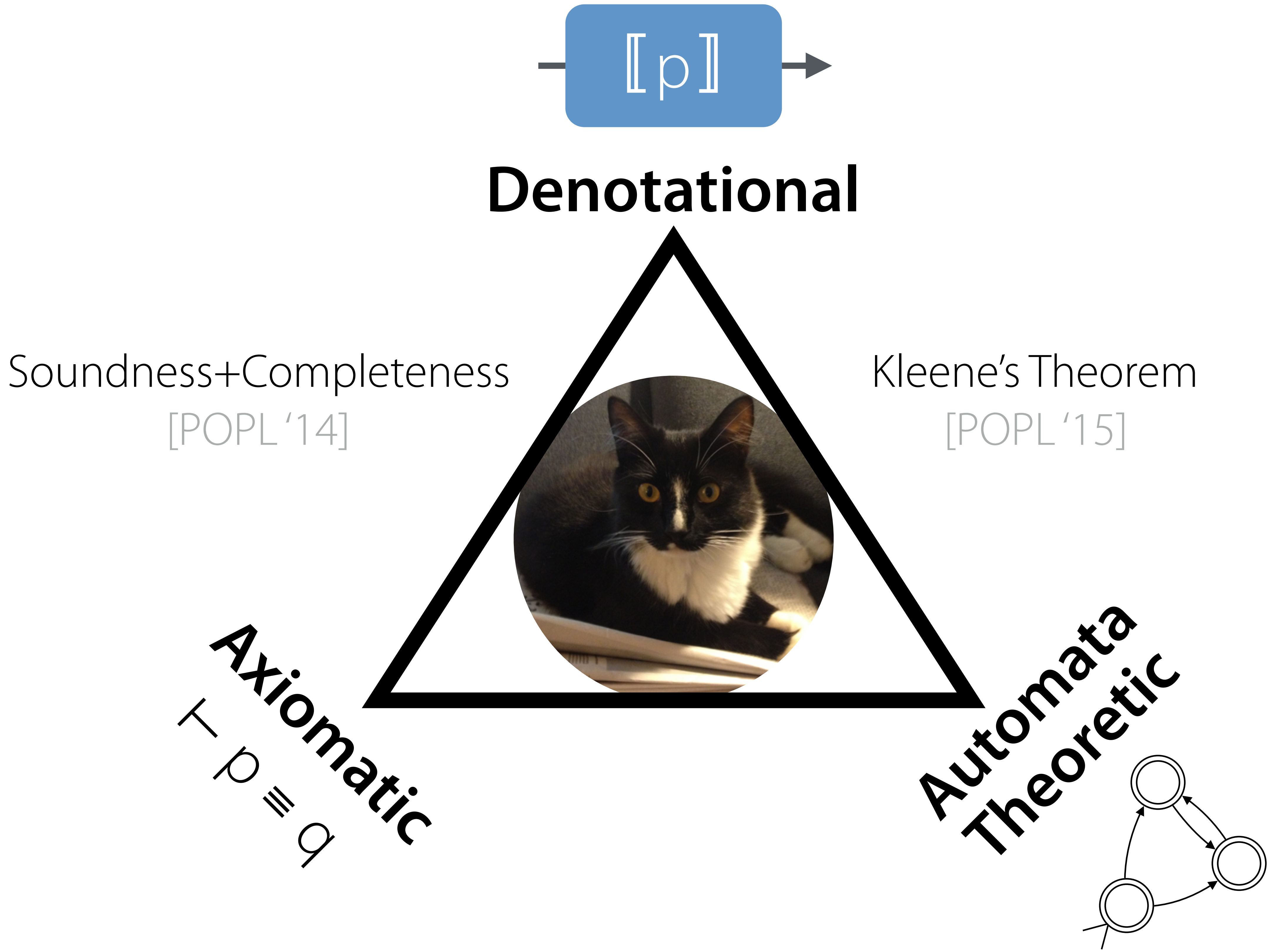
Axiomatic

$\vdash p \equiv q$

Automata Theoretic



NetKAT Semantics



NetKAT Semantics

$[\![p]\!]$

Denotational

Soundness+Completeness
[POPL '14]

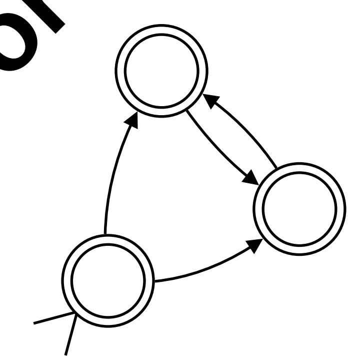
Kleene's Theorem
[POPL '15]

Axiomatic

$\vdash p \equiv q$

Proof-Carrying
Code
[Ongoing Work]

Automata Theoretic



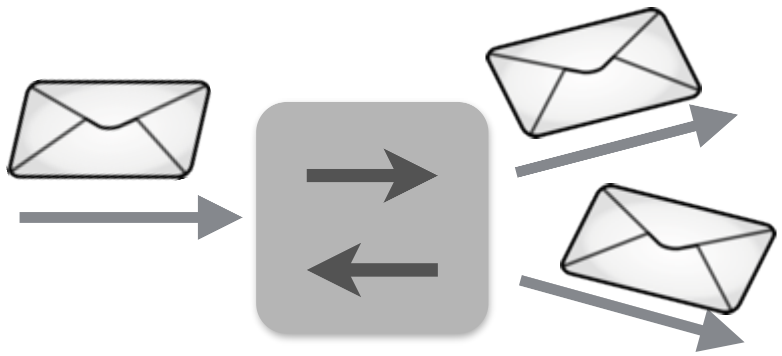
Denotational Semantics

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 NetKAT: input-output behavior of switches

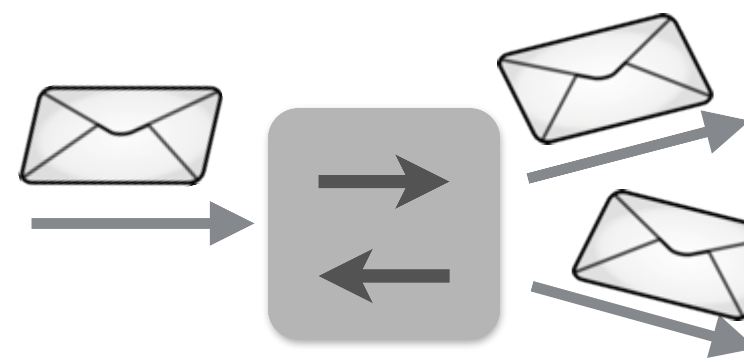


$[\![\text{pol}]\!] \in$ Packet → Packet Set

Denotational Semantics

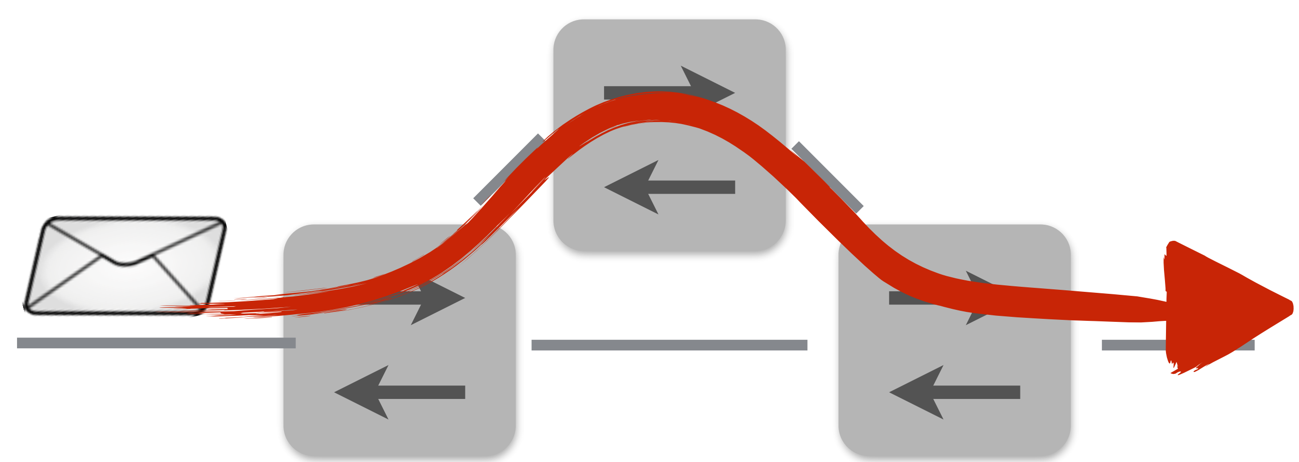
pol ::=
  | **false**
  | **true**
  | field = val
  | field **:=** val
  | pol_1 **+** pol_2
  | pol_1 • pol_2
  | !pol
  | pol*
  | **dup**

Local NetKAT: input-output behavior of switches



⟦pol⟧ ∈ Packet → Packet Set

Global NetKAT: network-wide paths



⟦pol⟧ ∈ History → History Set

NetKAT Semantics

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

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

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

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

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

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

$[\![pol_1 \mathbf{+} pol_2]\!]\ h = [\![pol_1]\!]\ h \cup [\![pol_2]\!]\ h$

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

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

$[\![\ \mathbf{dup}\ ]\!]\ pk \mathbin{\mathbf{::}} h = \{\ pk \mathbin{\mathbf{::}} pk \mathbin{\mathbf{::}} h\ \}$

$f,g \in$ **History → History Set**

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

Axiomatic Semantics

NetKAT's design is based upon canonical structures:

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

The combination of a Boolean Algebra and a Kleene Algebra is called a ***Kleene Algebra with Tests (KAT)*** [Kozen '96]

KAT has an accompanying proof system for showing equivalences of the form $p \equiv q$

NetKAT Axioms

Kleene Algebra Axioms

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

$p + q \equiv q + p$

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

$p + p \equiv p$

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

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

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

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

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

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

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

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

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

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

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

Boolean Algebra Axioms

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

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

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

$a \bullet b \equiv b \bullet a$

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

$a \bullet a \equiv a$

Packet Axioms

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

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

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

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

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

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

$\text{dup} \bullet f = n \equiv f = n \bullet \text{dup}$

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

NetKAT Axioms

Kleene Algebra Axioms

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

$p + q \equiv q + p$

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

$p + p \equiv p$

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

$p \bullet (q + r)$

$(p + q) \bullet r$

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

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

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

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

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

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

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

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

Boolean Algebra Axioms

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

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

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

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

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

if $f \neq f'$

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

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

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

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

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

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

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

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,
- $\varepsilon \in S \rightarrow \text{Packet} \rightarrow \text{Packet Set}$ is the "observation" function
- $\delta \in S \rightarrow \text{Packet} \rightarrow (\text{State} * \text{Packet})\ \text{Set}$ 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,
- $\varepsilon \in S \rightarrow \text{Packet} \rightarrow \text{Packet Set}$ is the "observation" function
- $\delta \in S \rightarrow \text{Packet} \rightarrow (\text{State} * \text{Packet}) \text{ Set}$ is the "continuation" function

Inputs: $pkt_{in} \cdot pkt_1 \cdot$ **dup** $\cdot \ldots \cdot$ **dup** $\cdot\, pkt_n$ **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,
- $\varepsilon \in S \rightarrow \text{Packet} \rightarrow \text{Packet Set}$ is the "observation" function
- $\delta \in S \rightarrow \text{Packet} \rightarrow (\text{State} * \text{Packet})\ \text{Set}$ is the "continuation" function

Inputs: $pkt_{in} \cdot pkt_1 \cdot \textbf{dup} \cdot \ldots \cdot \textbf{dup} \cdot pkt_n\ \textbf{dup} \cdot pkt_{out}$

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

- $\text{accept } s\ (pkt_{in} \cdot pkt_{out}) \Leftrightarrow pkt_{out} \in \varepsilon\ s\ pat_{in}$
- $\text{accept } s\ (pkt_{in} \cdot pkt_1 \cdot \textbf{dup} \cdot w) \Leftrightarrow$

  $\exists\, s'.\ (pkt_1, s') \in \delta\ s\ pkt_{in} \text{ and accept } s'\ (pkt_1 \cdot w)$

NetKAT Derivatives

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

$E(pol) \in Pol$

$D(pol) \in (Pol * L * Pol)\ Set$

$E(\textbf{\textit{false}}) = \textbf{\textit{false}}$
$E(\textbf{true}) = \textbf{true}$
$E(f \mathbf{=} n) = f \mathbf{=} n$
$E(f \mathbf{:=} n) = f \mathbf{:=} n$
$E(\mathbf{!}pol) = \mathbf{!}pol$
$E(dup^l) = \textbf{false}$
$E(pol_1 \mathbf{+} pol_2) = E(pol_1) \mathbf{+} E(pol_2)$
$E(pol_1 \bullet pol_2) = E(pol_1) \bullet E(pol_2)$
$E(pol^*) = E(pol)^*$

$D(\textbf{false}) = \{\}$
$D(\textbf{true}) = \{\}$
$D(f \mathbf{=} n) = \{\}$
$D(f \mathbf{:=} n) = \{\}$
$D(\mathbf{!}pol) = \{\}$
$D(dup^l) = \{ (\textbf{true}, l, \textbf{true}) \}$
$D(pol_1 \mathbf{+} pol_2) = D(pol_1) \mathbf{+} D(pol_2)$
$D(pol_1 \bullet pol_2) = D(pol_1) \bullet pol_2 + E(pol_1) \bullet D(pol_2)$
$D(pol^*) = E(pol)^* \bullet D(pol) \bullet pol^*$

NetKAT Automata

We can build an automaton using derivatives as follows:

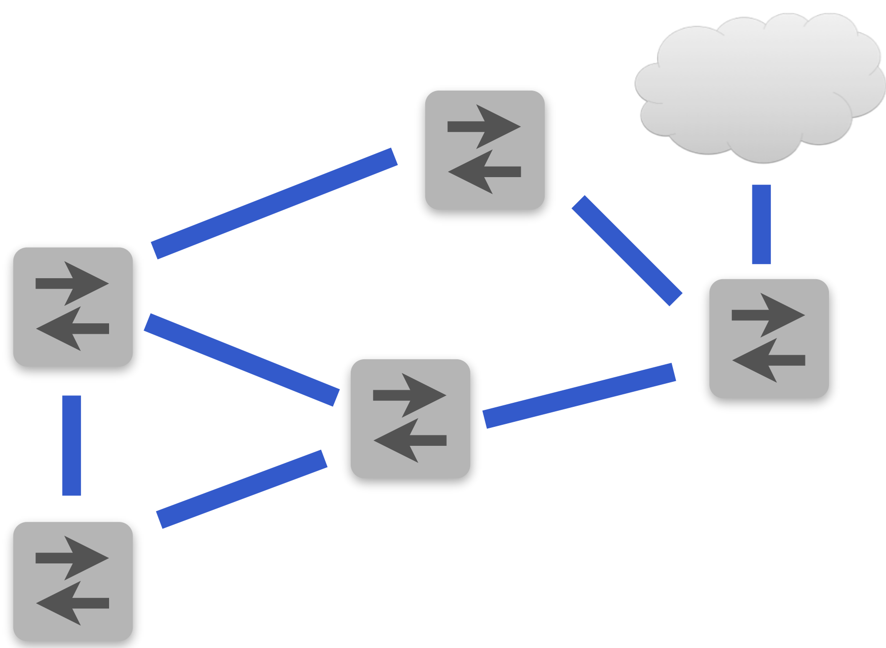
- S is the set of labels in pol, plus a fresh start state 0,
- $\varepsilon\ l\ pkt = \{pkt' \mid \langle pkt' \rangle \in [\![E(k_l)]\!]\ \langle pkt \rangle\}$
- $\delta\ l\ pkt = \{(pkt', l') \mid (d, l', k) \in [\![D(k_l)]\!] \wedge \langle pkt' \rangle \in [\![d]\!]\ \langle pkt \rangle\}$

Notation: k_l denotes the unique continuation of dup^l

Of course, in practice, it is important to use symbolic representations (e.g., FDDs [ICFP '15]) to keep the size of the automata manageable.

Applications

Reachability [POPL '14]



$\models \; \phi$

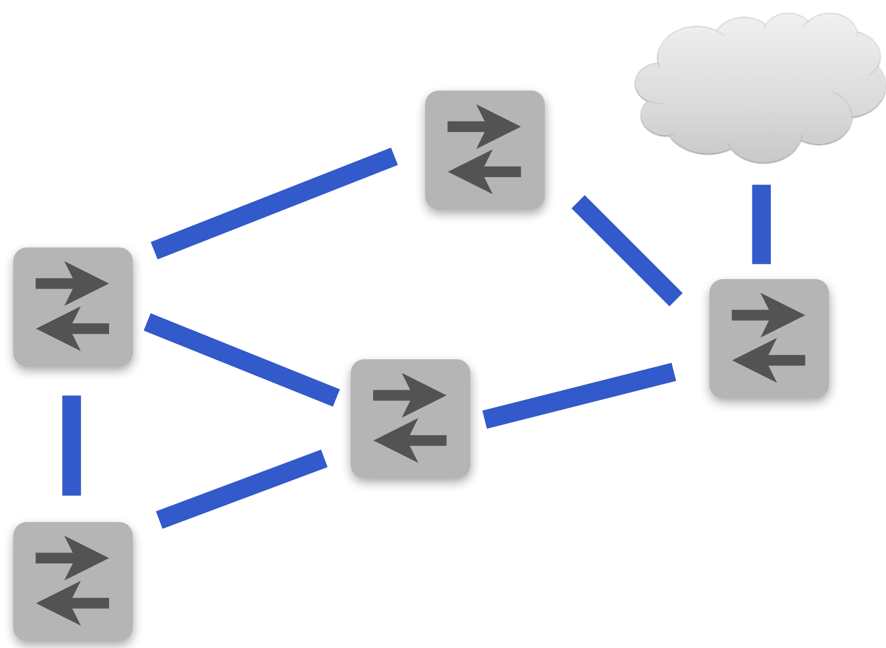
We'd like to be able to answer questions like:

"Does the network forward from ingress to egress?"

Can reduce this question (and others) to (in)equivalence

in • (pol • topo)* • pol • out $\not\equiv$ **false**

Loop Freedom [POPL '15]

 $\models \Phi$

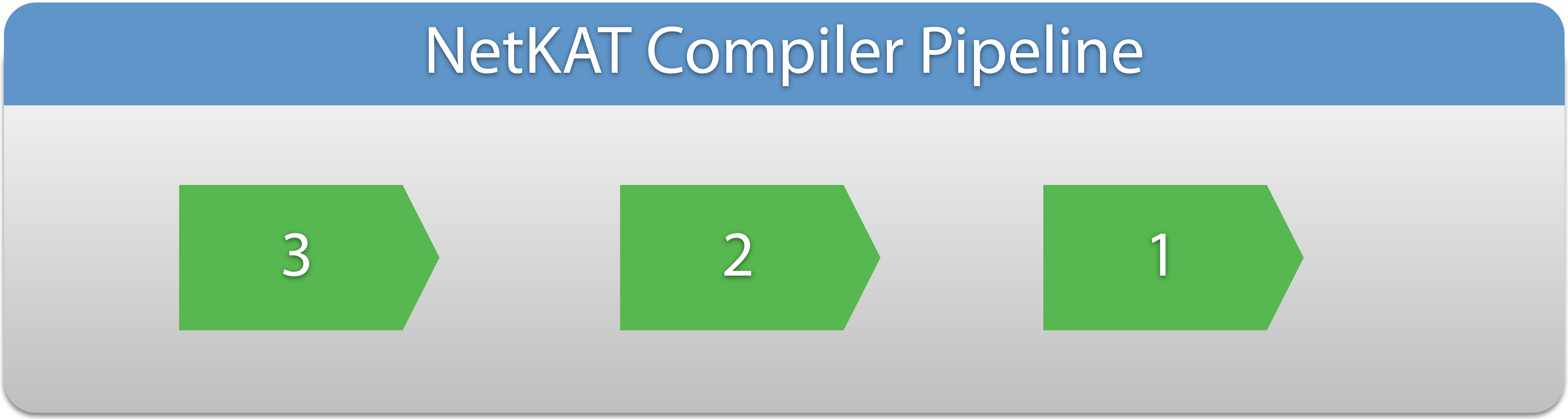
Can use automata to check if a network is loop free

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

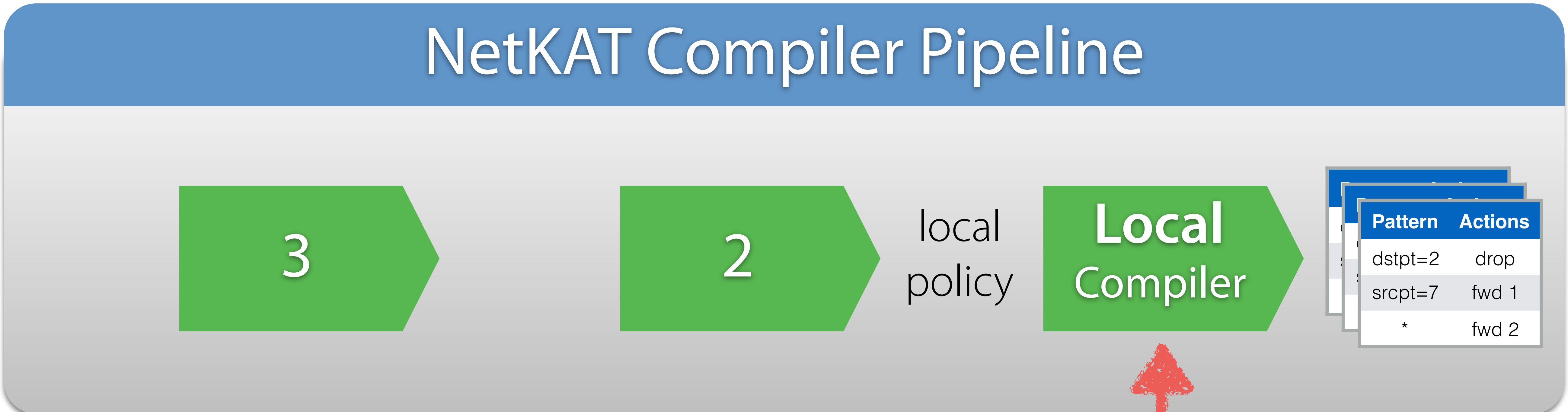
- $\forall$ pkt, pkt'. pkt' $\in [\![E(\Phi(in \bullet (p \bullet t)^{*}))]\!]$ pkt
- Check whether pkt' $\in [\![E(\Phi(p \bullet t)^{+}))]\!]$ pkt'

Notation: $\Phi(p)$ denotes replacing **dup** with **true**

Compilation [ICFP '15]

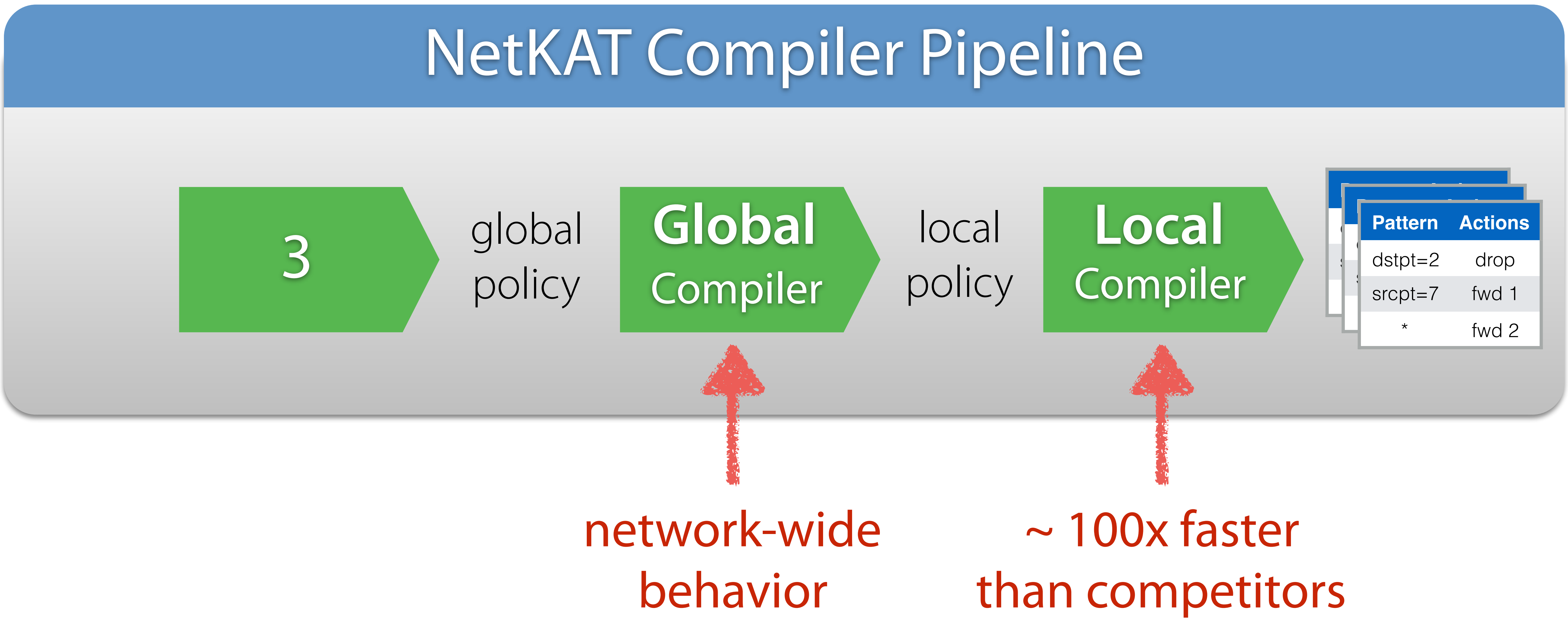


Compilation [ICFP '15]

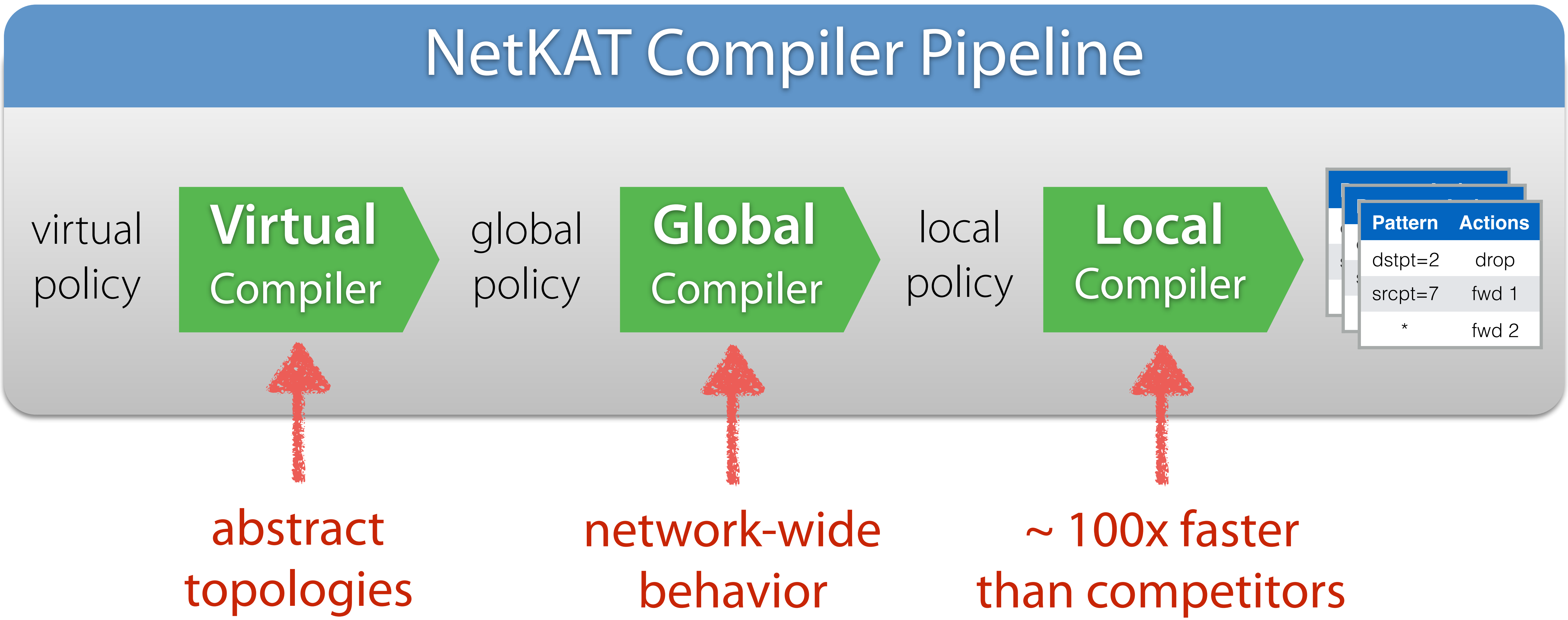


~ 100x faster
than competitors

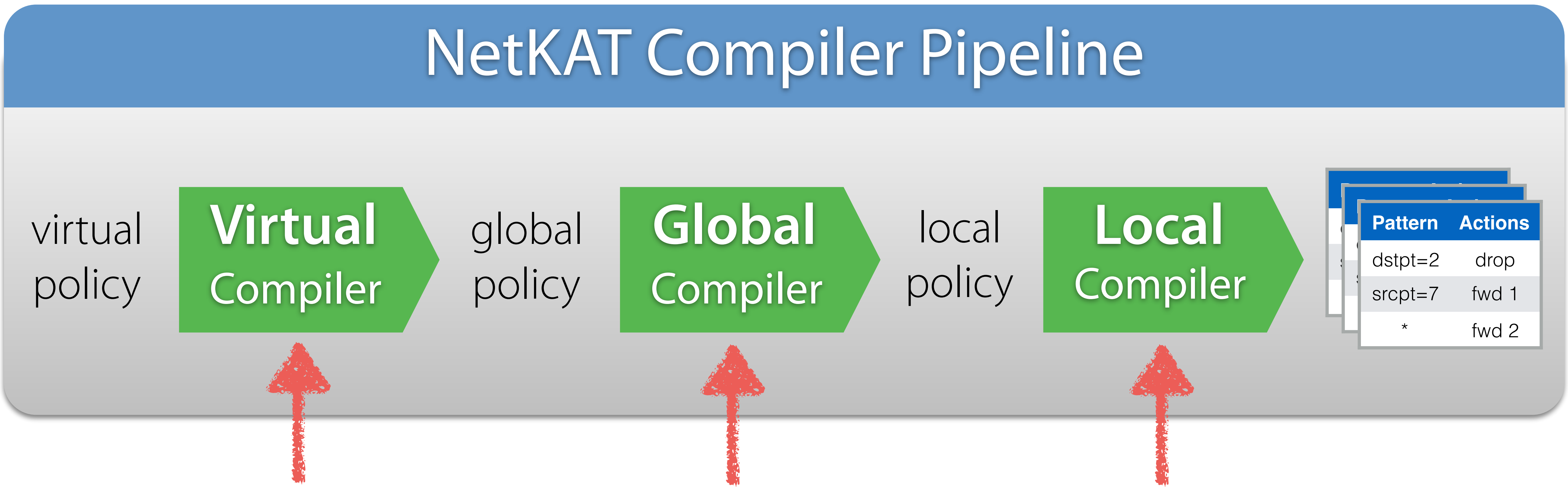
Compilation [ICFP '15]



Compilation [ICFP '15]



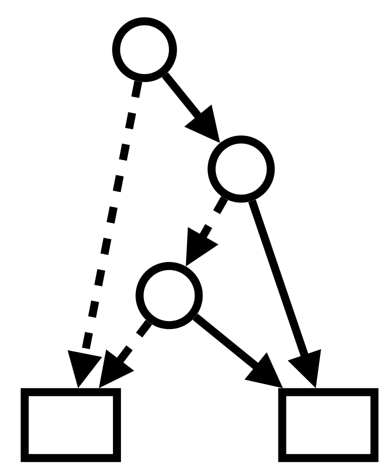
Compilation [ICFP '15]



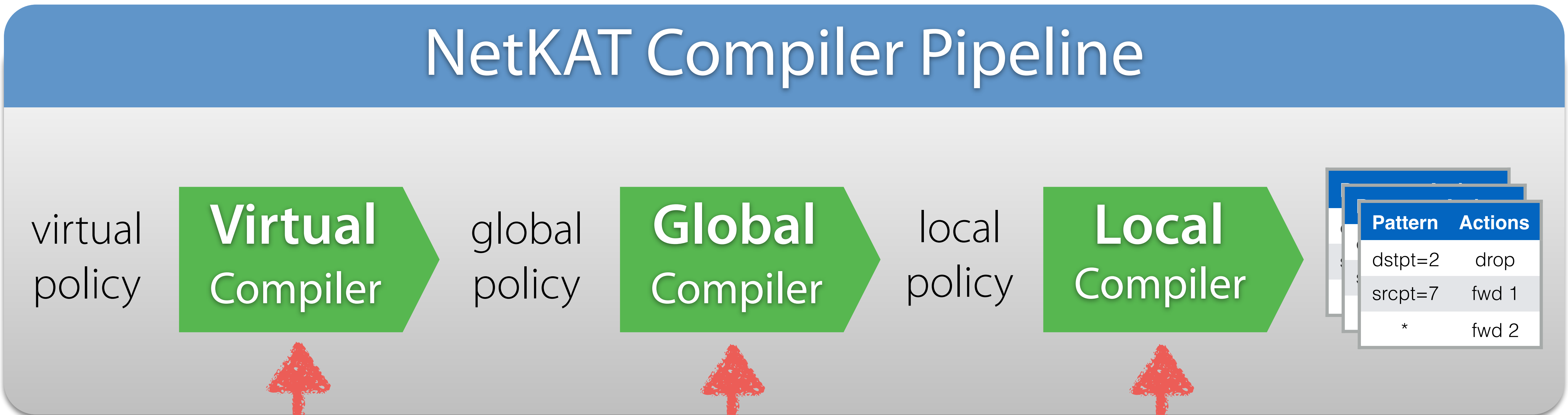
abstract topologies

network-wide behavior

~ 100x faster than competitors



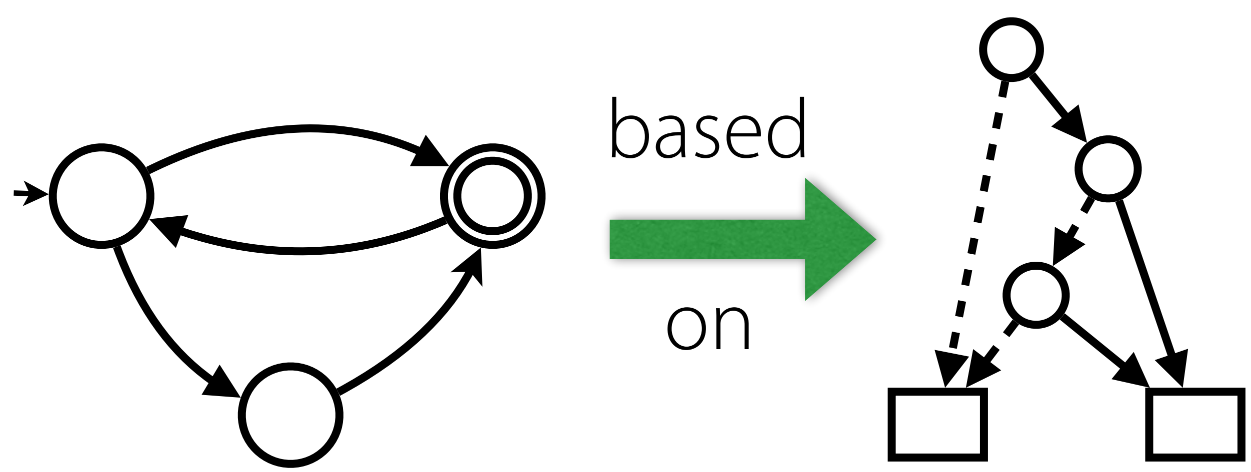
Compilation [ICFP '15]



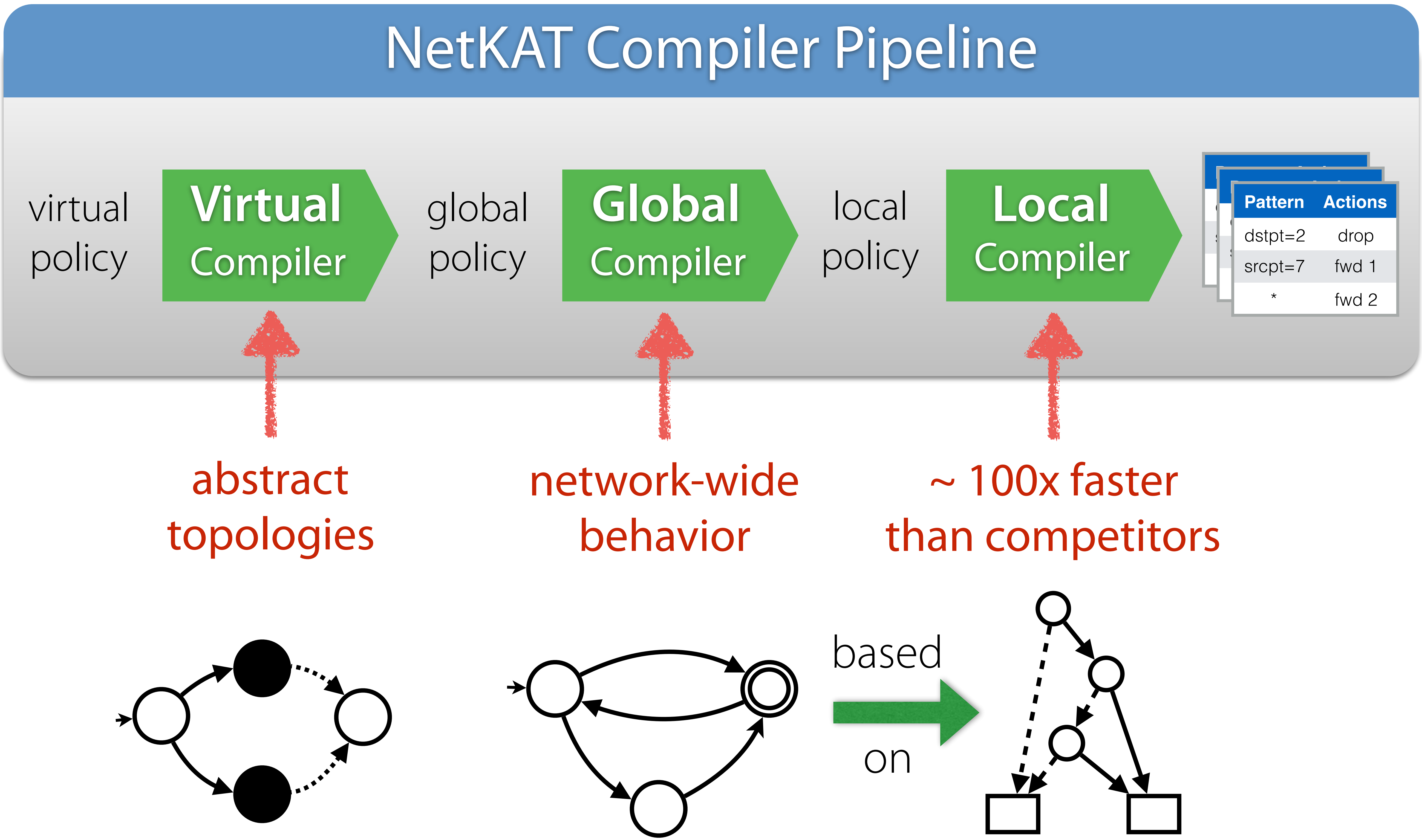
abstract topologies

network-wide behavior

~ 100x faster than competitors

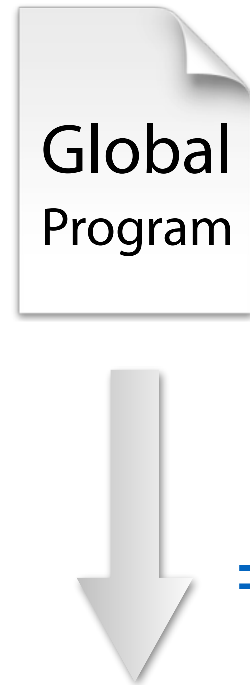


Compilation [ICFP '15]



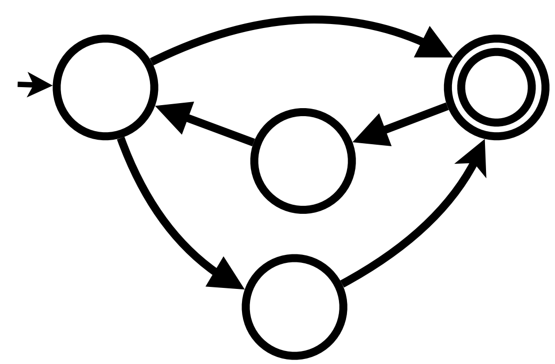
Global Compilation

Global Compilation



Adding Extra State
= Translation to Automaton

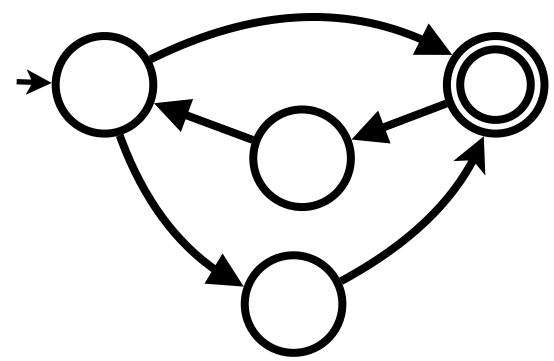
NetKAT NFA



Global Compilation

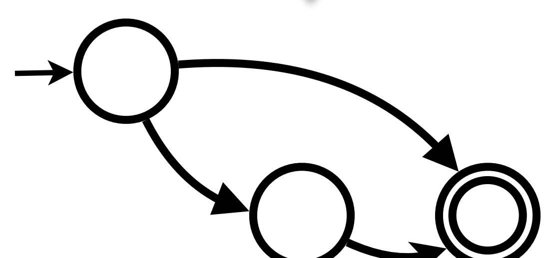
Adding Extra State
= Translation to Automaton

NetKAT NFA

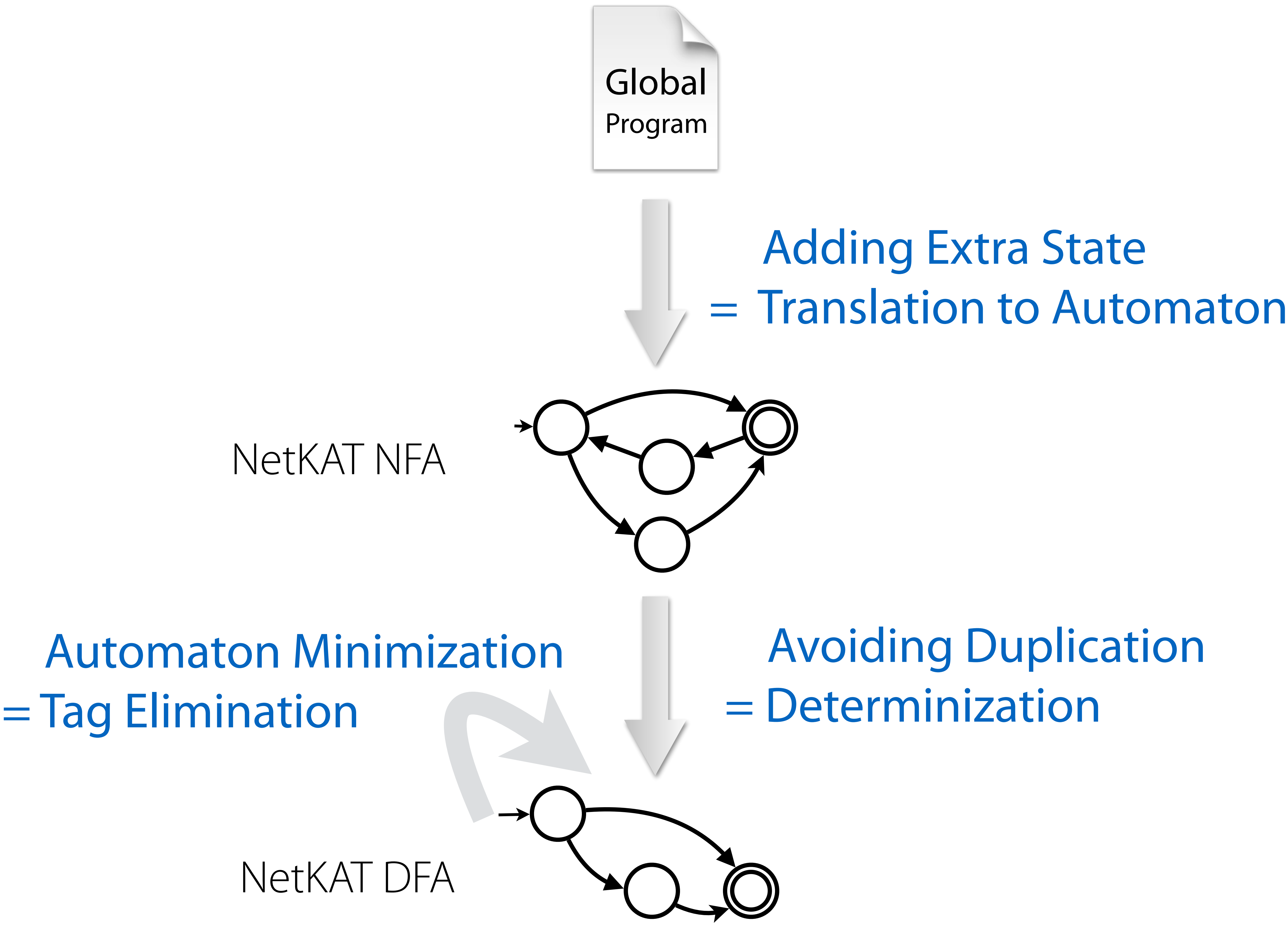


Avoiding Duplication
= Determinization

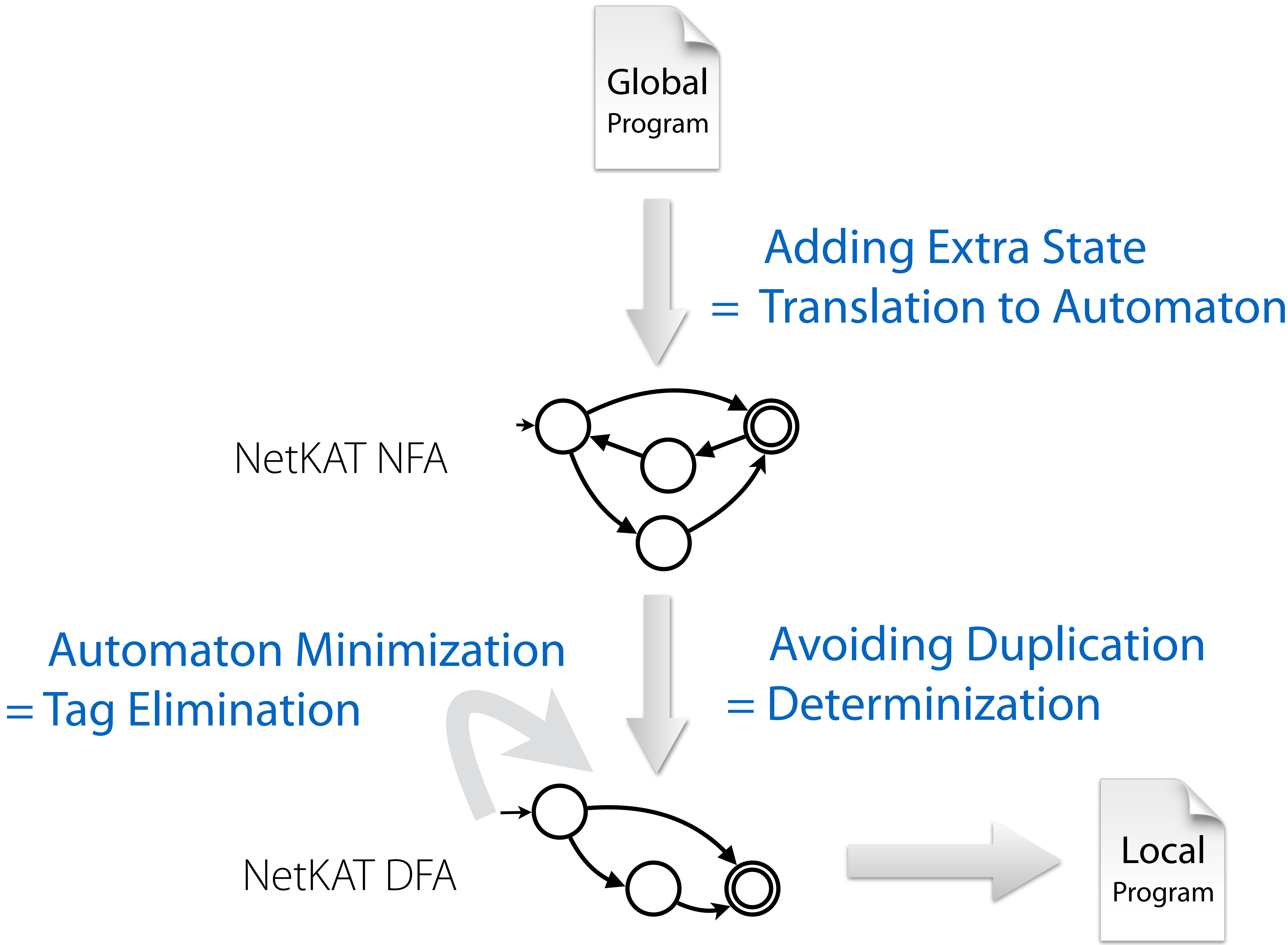
NetKAT DFA



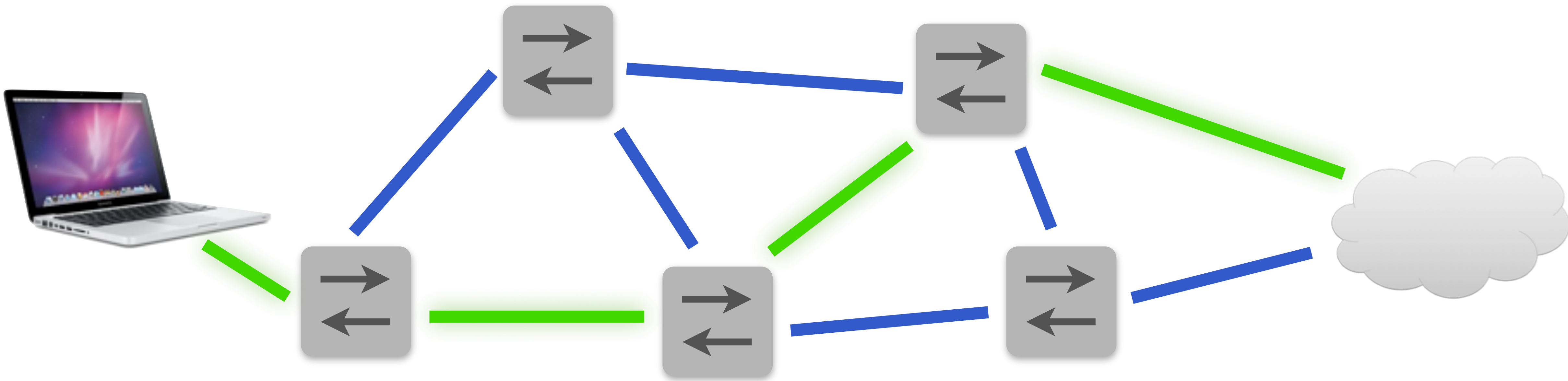
Global Compilation



Global Compilation

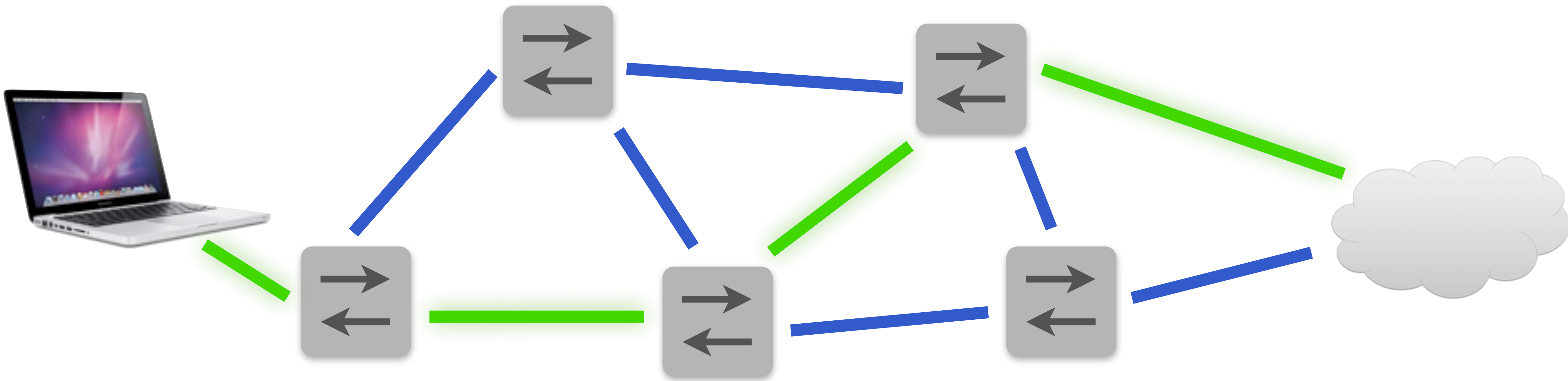


Measurement [SOSR '16]



Can implement path queries at the network edge!

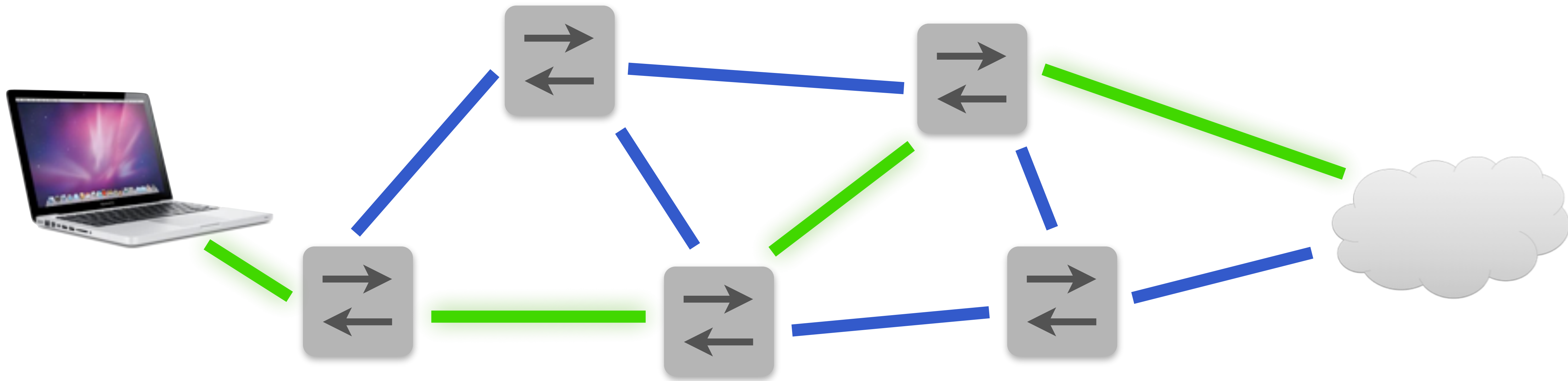
Measurement [SOSR '16]



Can implement path queries at the network edge!

- Combine policy p and query q via a simple translation

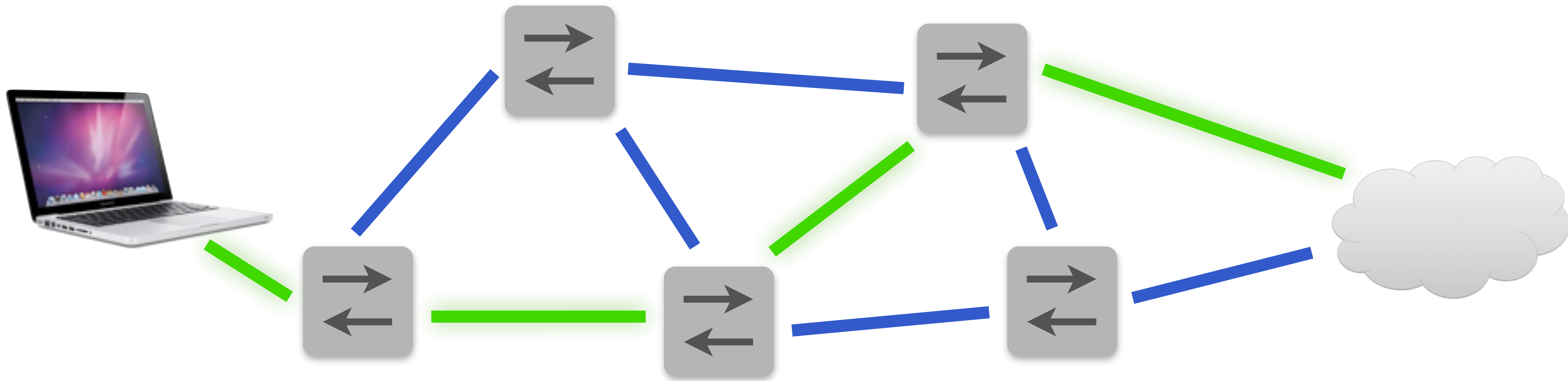
Measurement [SOSR '16]



Can implement path queries at the network edge!

- Combine policy p and query q via a simple translation
- Use Φ to replace all occurrences of **dup** with **true**

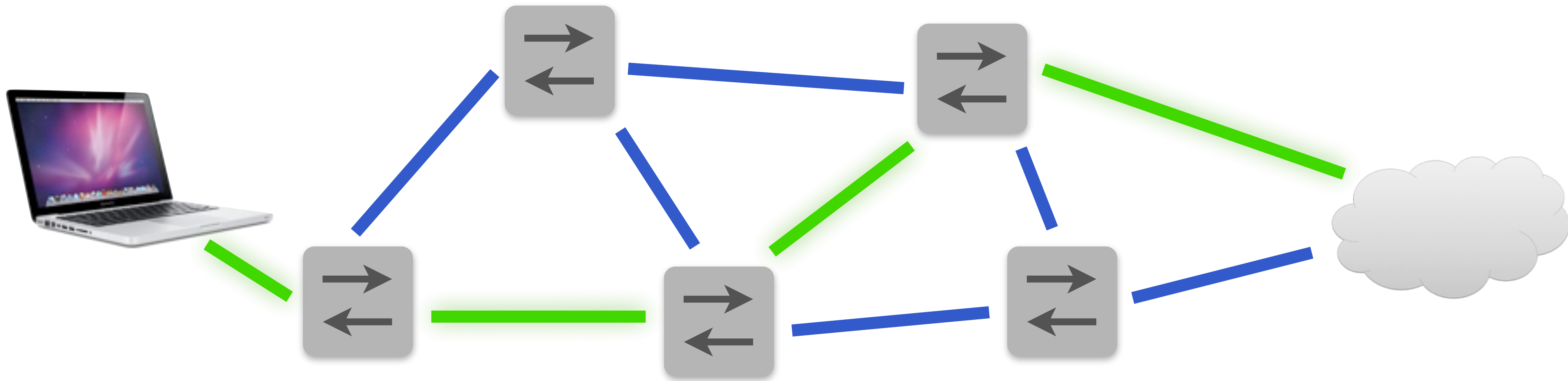
Measurement [SOSR '16]



Can implement path queries at the network edge!

- Combine policy p and query q via a simple translation
- Use Φ to replace all occurrences of **dup** with **true**
- Compute predicates from observation map E

Measurement [SOSR '16]



Can implement path queries at the network edge!

- Combine policy p and query q via a simple translation
- Use Φ to replace all occurrences of **dup** with **true**
- Compute predicates from observation map E
- Tabulate packets matching predicates on end hosts

Wrapping Up...

Ongoing Work

Stateful NetKAT [PLDI '16]

- Enriches the language with new features for programming stateful data planes
- Semantics is based on causal consistency and Winskel's "event structures"

Probabilistic NetKAT [ESOP '16]

- Enriches the language with random choice
- Can be used to model uncertainty about demands and failures, and randomized algorithms (e.g., ECMP)
- Semantics is based on Markov Kernels

Conclusion

- NetKAT offers a rich foundation for network programming
- Key features include boolean predicates, regular paths, and modular composition operators
- Semantics has been extensively studied and offers powerful mathematical tools for transforming and reasoning about programs, as well as guidance when designing extensions
- Many practical applications can be built using NetKAT including verification tools, compilers, and analysis tools

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.
- 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)*, January 2015.
- 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.

Collaborators

- Carolyn Anderson (UMass)
- Spiros Eliopoulos (Inhabited Type)
- **Arjun Guha** (UMass)
- Jean-Baptiste Jeannin (Samsung Labs)
- **Dexter Kozen** (Cornell)
- Matthew Milano (Cornell)
- Mark Reitblatt (Facebook)
- Cole Schlesinger (Samsung Labs)
- **Alexandra Silva** (UCL)
- **Steffen Smolka** (Cornell)
- Laure Thompson (Cornell)
- David Walker (Princeton)

Questions?

Questions?

http://github.com/frenetic-lang/frenetic/

frenetic >> Pyretic SDX FUJITSU