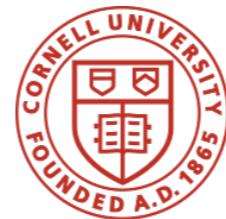


# Compiling Network Programs

Nate Foster  
Cornell University



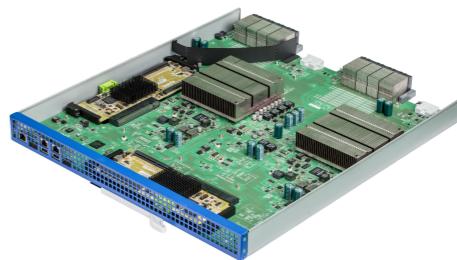
# Networking

# Computing



# Networking

# Computing



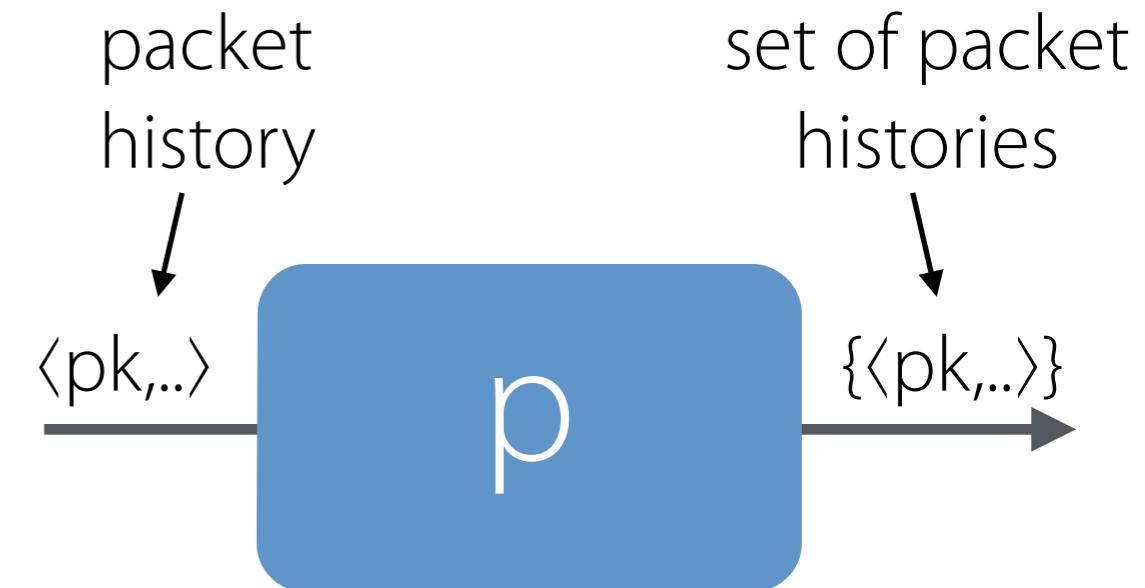
**P4**

*A machine model* based  
on flexible parsers and  
match-action pipelines

Match	Actions
ethType=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:01	Drop
ethType=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:02	Drop
ethType=0x800, ipProto=0x06, tcpDstPort=22,	Inport
ethType=0x800, ipProto=0x06	Inport
ethType=0x800	Inport
*	Inport

*A machine model* based  
on flexible parsers and  
match-action pipelines

Match	Actions
ethType=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:01	Drop
ethType=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:02	Drop
ethType=0x800, ipProto=0x06, tcpDstPort=22,	Inport
ethType=0x800, ipProto=0x06	Inport
ethType=0x800	Inport
*	Inport



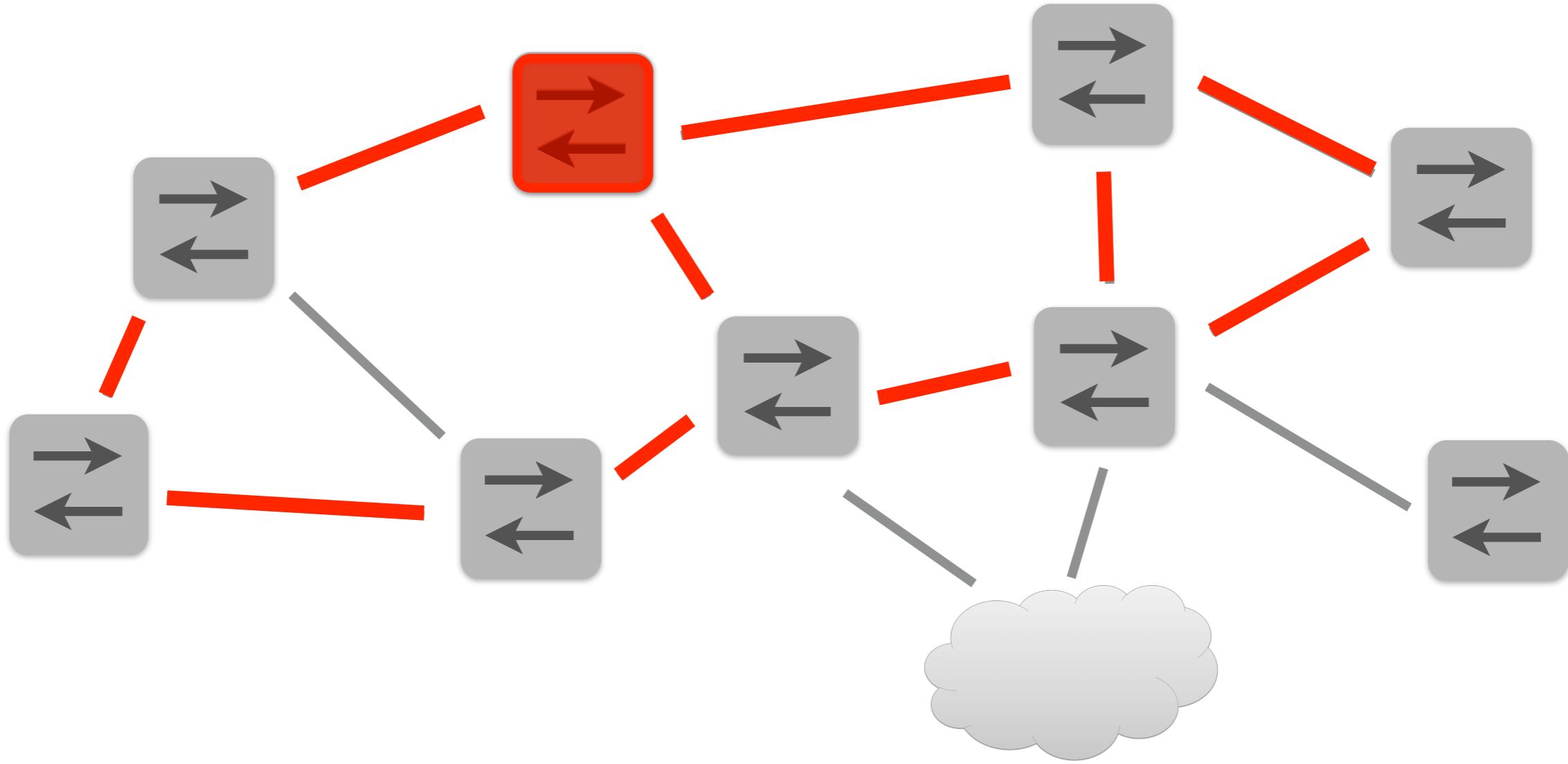
# This Talk

Design and implementation of a high-level  
network programming language called NetKAT



## Outline

- Language Design
- Compilation
- Experience



Two essential features:

- Packet classification
- Network-wide forwarding

# NetKAT Syntax

```
pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S⇒T
```

# NetKAT Syntax

```
pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S⇒T
```

Boolean  
Algebra

# NetKAT Syntax

```
pol ::= false  
      | true
```

```
      | field = val
```

```
      | pol1 + pol2
```

```
      | pol1; pol2
```

```
      | !pol
```

```
      | pol*
```

```
      | field := val
```

```
      | S  $\Rightarrow$  T
```

Boolean  
Algebra

+

Kleene  
Algebra

# NetKAT Syntax

```
pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S⇒T
```

Boolean  
Algebra

+

Kleene  
Algebra

+

Packet  
Primitives

# NetKAT Syntax

```
pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S  $\Rightarrow$  T
```

Boolean  
Algebra  
+  
Kleene  
Algebra  
+  
Packet  
Primitives

} KAT  
[Kozen '96]

# NetKAT Syntax

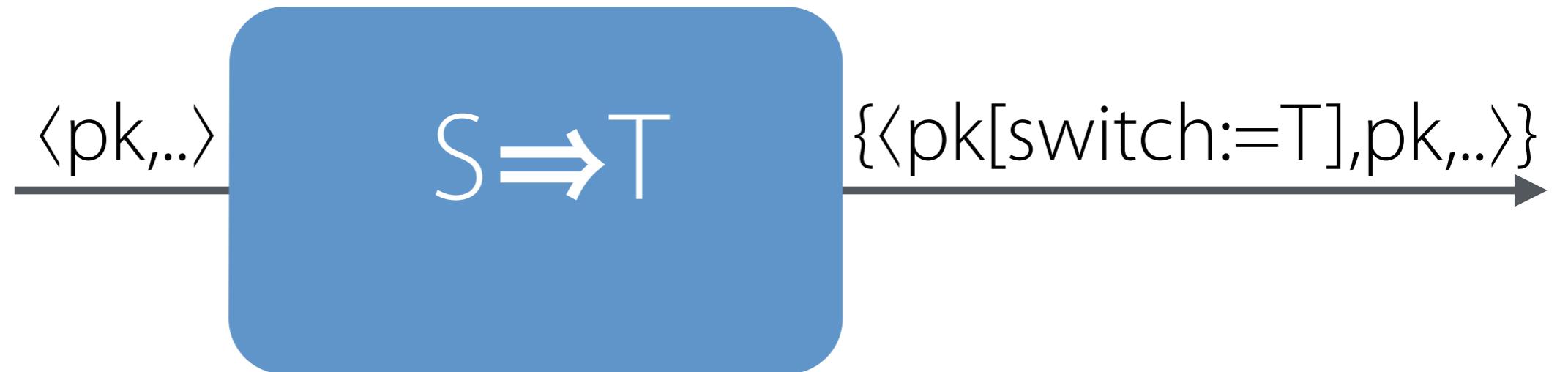
```
pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S  $\Rightarrow$  T
```

Boolean  
Algebra  
+  
Kleene  
Algebra  
+  
Packet  
Primitives

KAT  
[Kozen '96]

NetKAT  
[Anderson et al. '14]

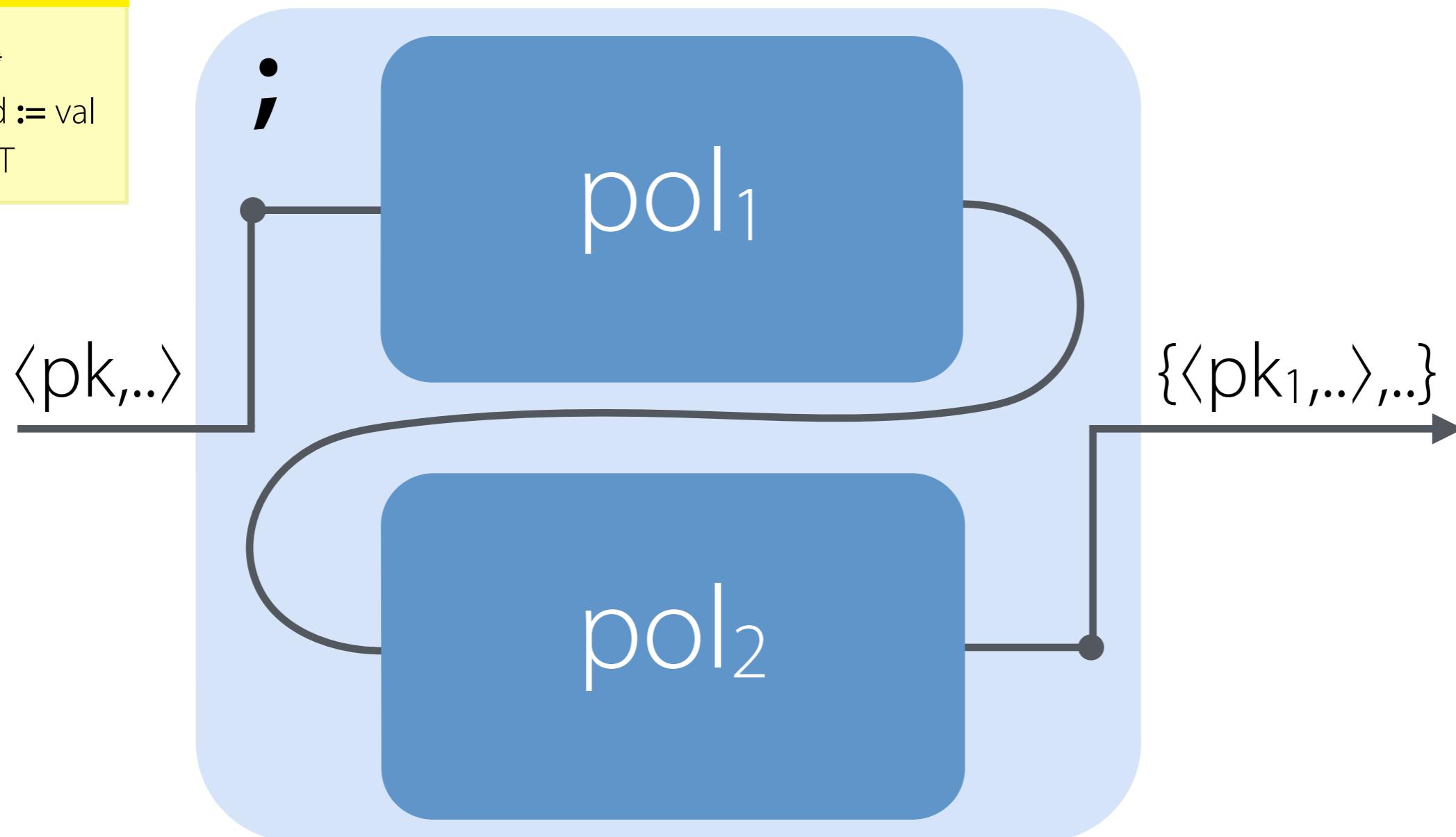
```
pol ::= false  
| true  
| field = val  
| pol1 + pol2  
| pol1; pol2  
| !pol  
| pol*  
| field := val  
| S⇒T
```



$S \Rightarrow T$  forwards packets across the link between S and T

```

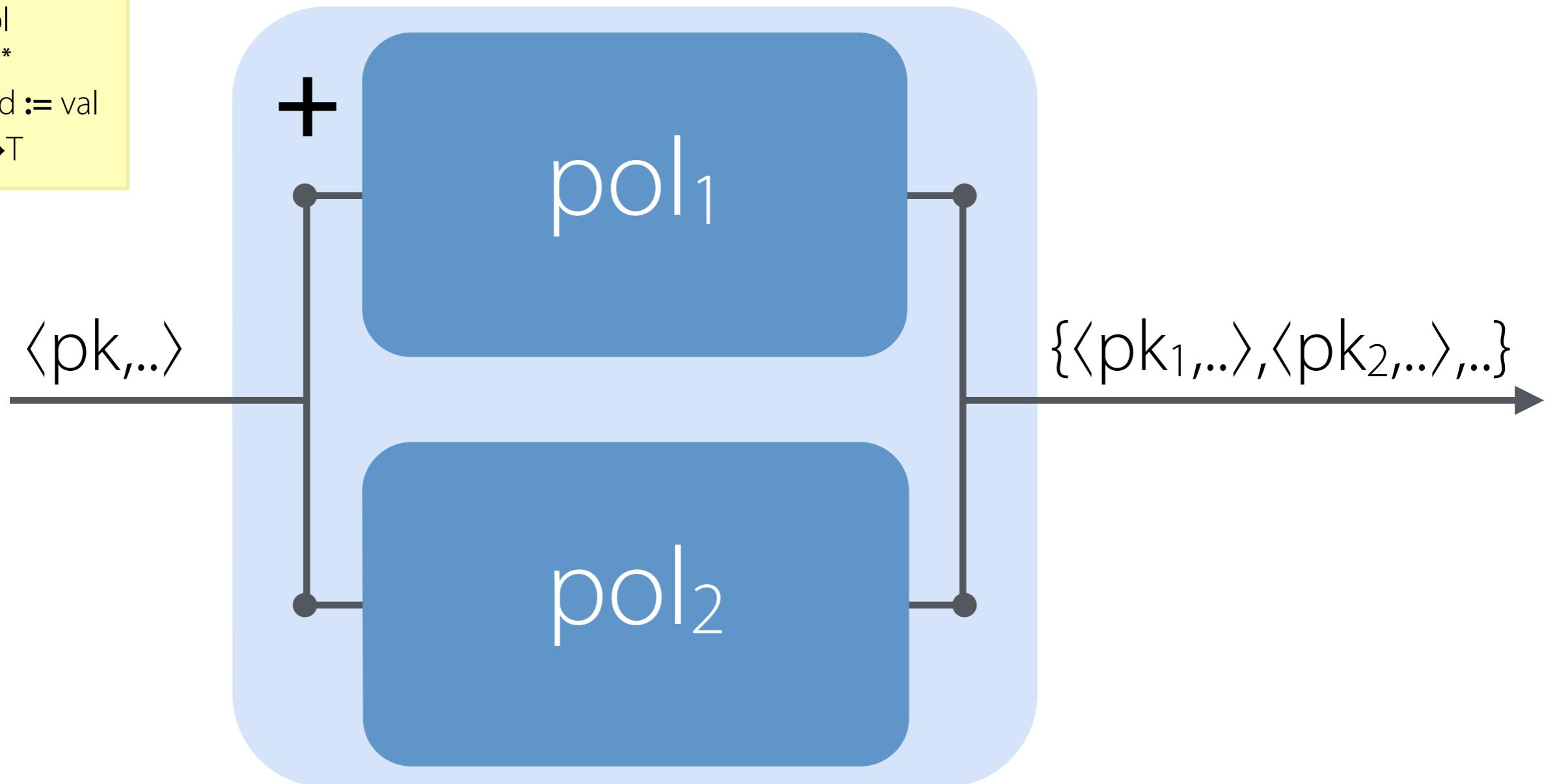
pol ::= false
      | true
      | field = val
      | pol1 + pol2
| pol1 ; pol2
      | !pol
      | pol*
      | field := val
      | S  $\Rightarrow$  T
  
```



$pol_1 ; pol_2$  runs the input through  $pol_1$  and then runs every output produced by  $pol_1$  through  $pol_2$

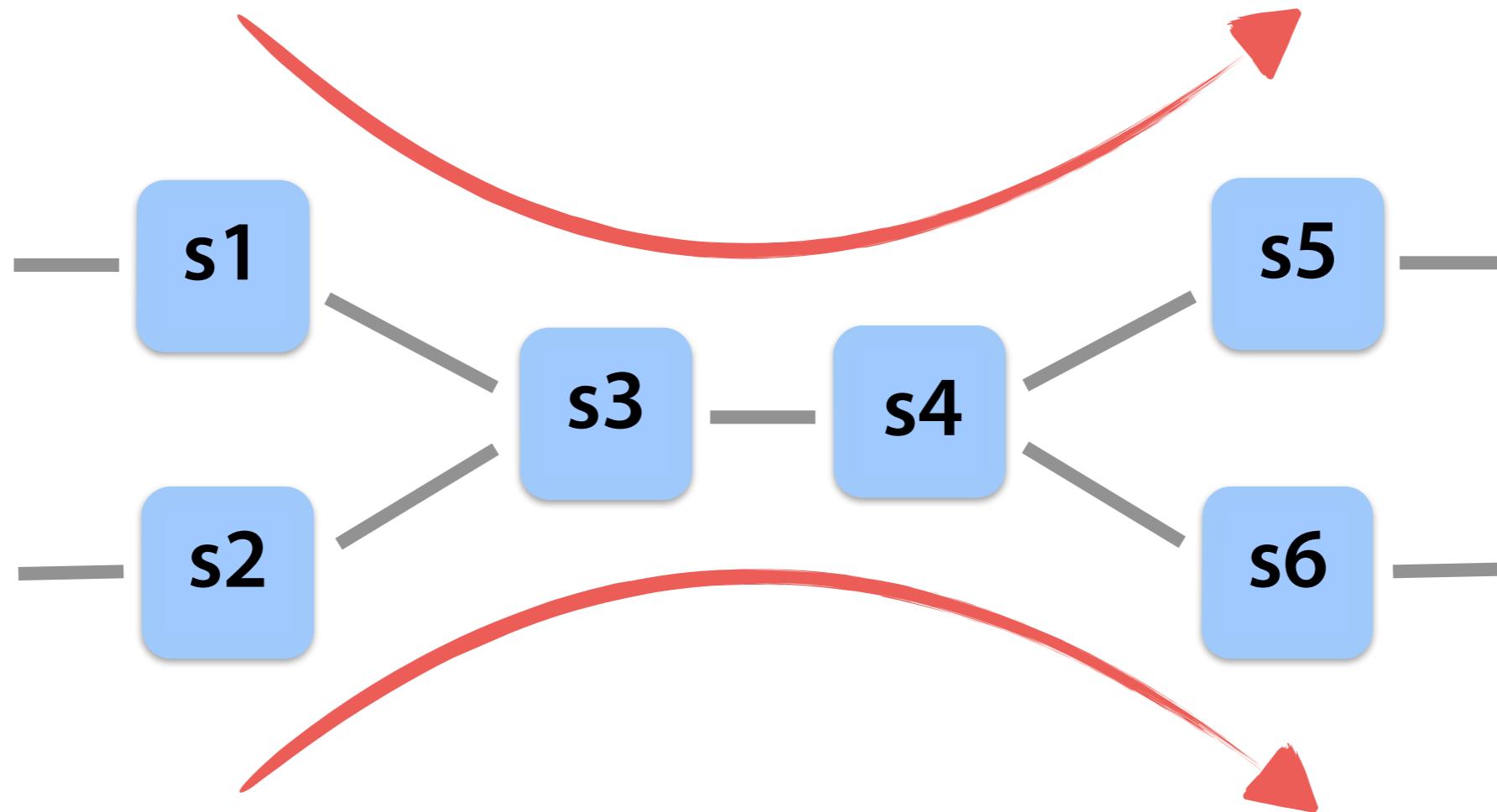
```

pol ::= false
      | true
      | field = val
      | pol1 + pol2
      | pol1; pol2
      | !pol
      | pol*
      | field := val
      | S⇒T
  
```



**pol<sub>1</sub> + pol<sub>2</sub>** duplicates the input, sends one copy to each sub-policy, and takes the union of their outputs

# Example



$$(s_1 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_5) + (s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_6)$$

# Other Applications

- Automatic Verification
  - Network Virtualization
  - Traffic Engineering
  - Fault Tolerance
  - Application Intent



# NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College<sup>\*</sup>

Nate Foster  
Cornell University

Arijit Ghosh  
University of Massachusetts Amherst<sup>†</sup>

Jean-Baptiste Jeannin  
Carnegie Mellon University<sup>\*</sup>

Dexter Kozen  
Cornell University

Cole Schleginger  
Princeton University

Dan id Walker  
Princeton University

## Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc. In this paper we propose NetKAT, a language for specifying the capabilities of networks built from standard components. The key idea is that the behavior of a network is best understood as a composition of the behaviors of its individual components, and little guidance is needed to help one incorporate new features.

This paper presents NetKAT, a new network programming language. It is based on a formal semantics that is built on top of a language equipped with a sound and complete equational theory. We describe the semantics and the language in detail, and show how it can be used to reason about a Kλ interpreter. We also show how to build a compiler that translates Kλ programs to NetKAT programs. Finally, we demonstrate that our language can be used to express practical applications of network programming, including the analysis of reachability, proving non-interfering properties that ensure the consistency of network configurations, and establishing the correctness of connection algorithms.

**Categories and Subject Descriptors** D.2.2 [Programming Languages]: Design at a high level; D.2.3 [Programming Languages]: Semantics; D.2.4 [Programming Languages]: Semantics—Formal methods

**Keywords** Software-defined networking, Functional, Network programming languages, Domain-specific languages, Kλte language, network composition

## 1. Introduction

Traditional network devices have been called “the last bastion of mainframe computing” [3]. Until modern computers, which are

implemented with commodity hardware and programmed using imperative languages, networks have been considered too slow and error-prone for specialized purpose devices such as routers, switches, and firewalls. This has changed with the rise of software-defined networks. This design makes it difficult to extend networks with new features without changing their internal structure or understanding their behavior.

Imperative languages have taken over with the rise of more abstract software-defined networking (SDN). In this, a general purpose computer runs a controller program that manages the network.

The controller responds to network events such as new connections and updates to the network topology by sending messages to the switches, telling them what to do. This is a good way to program the switches accurately. Because the controller is a general purpose computer, it can run a wide variety of applications, from a wide variety of standard applications such as shortest-path routing to complex applications such as distributed ledger systems. Practical applications include load balancing, intrusion detection, and security.

A major appeal of SDN is that it allows open standards that can be easily extended. The OpenFlow protocol [1] clearly specifies the capabilities and behavior of switch hardware and provides a standard interface for controller communication. However, programs written directly for SDN platforms such as Open vSwitch [2] are not portable across different hardware or different controllers.

**Network programming languages.** In recent years, several different network programming languages have been proposed. These languages are built on top of existing functional programming languages to raise the level of abstraction of network management. One approach is to build a domain-specific language, thereby making it easier to build sophisticated and reliable SDN applications. For example, the Nettle language [11] is a domain-specific language for building SDN applications. The Nettle language is designed to support static type checking, modular programming, and incremental compilation. The Nettle language also supports expressiveness: dynamic policies can be defined using the Nettle language. Another approach is to use a sequence of static policies—simply reusing policies from one application to another. This is the approach taken by the NetKAT language [12].

NetKAT is a domain-specific language for specifying the behavior of a sequence of static policies—simply reusing policies from one application to another. This is the approach taken by the NetKAT language [12].

Still, it has never been clear what features a static policy has to support. In this paper, we propose a semantics for NetKAT that provides a precise definition of what actions are legal and what actions are illegal. We also propose a set of rules for specifying the list of pre-action rules as policies, where the actions in-

<sup>\*</sup>The work performed at Cornell University.

<sup>†</sup> Permission to make digital or hard copies of part or all of this work for personal research and educational purposes is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a license from the publisher. Requests for permission should be addressed to permissions@acm.org. Copyright © 2014, ACM, Inc. ISBN 978-1-4503-3581-1/14/06. Article 10. \$15.00.  
<http://doi.acm.org/10.1145/2590011.2590040>

# FatTire: Declarative Fault Tolerance for Software-Defined Networks

[POPL '14]

[POPL '15]

[ICFP '15]

[CoNext '14]

[HotSDN '13]

# NetKAT Compilation

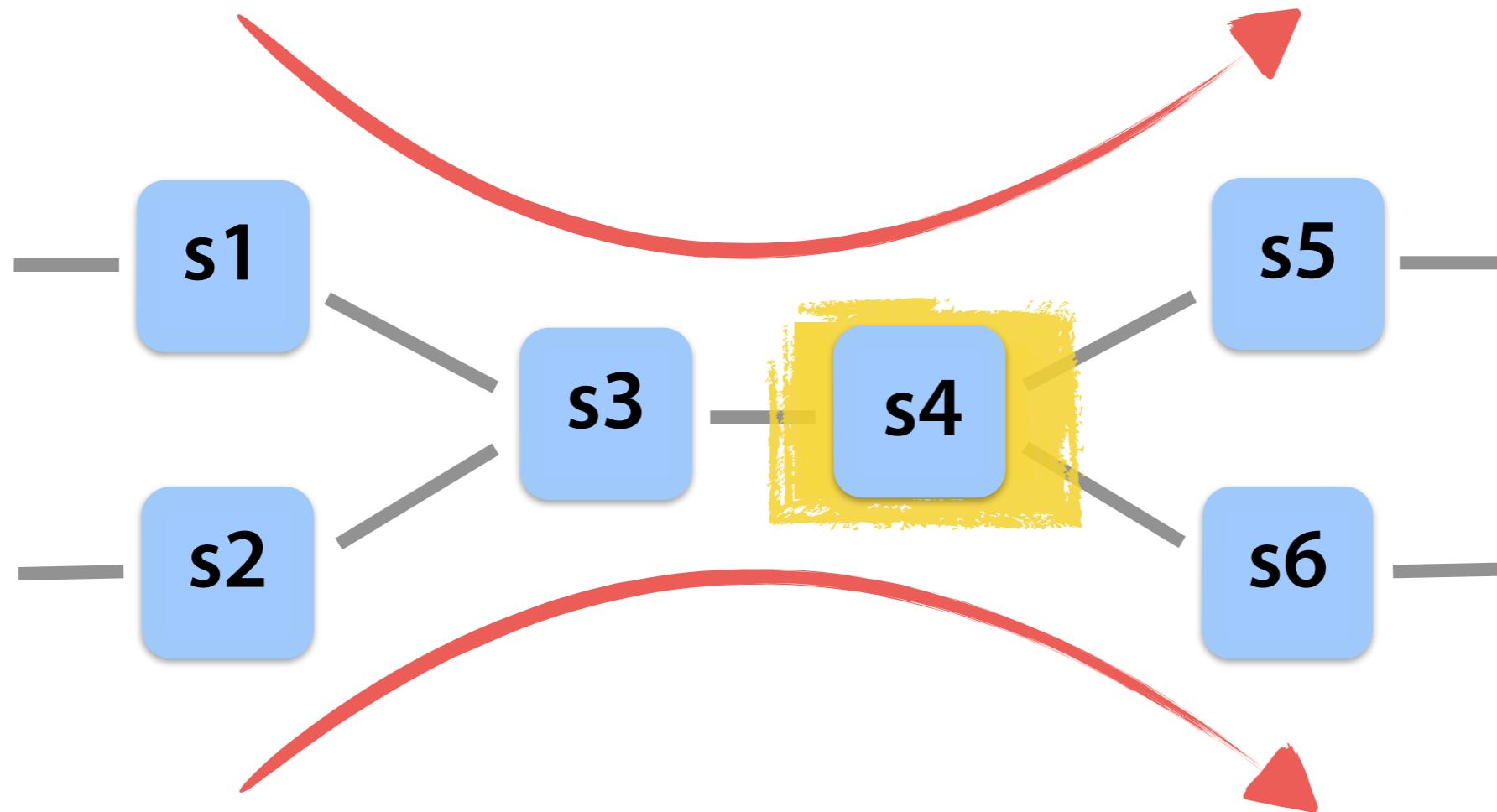
## Compiler Pipeline



## Key Challenges

- Implementing network-wide functions using local packet-processing elements
- Developing efficient data structures and algorithms to allow compilation to scale

# Example



$$(s_1 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_5) + (s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_6)$$

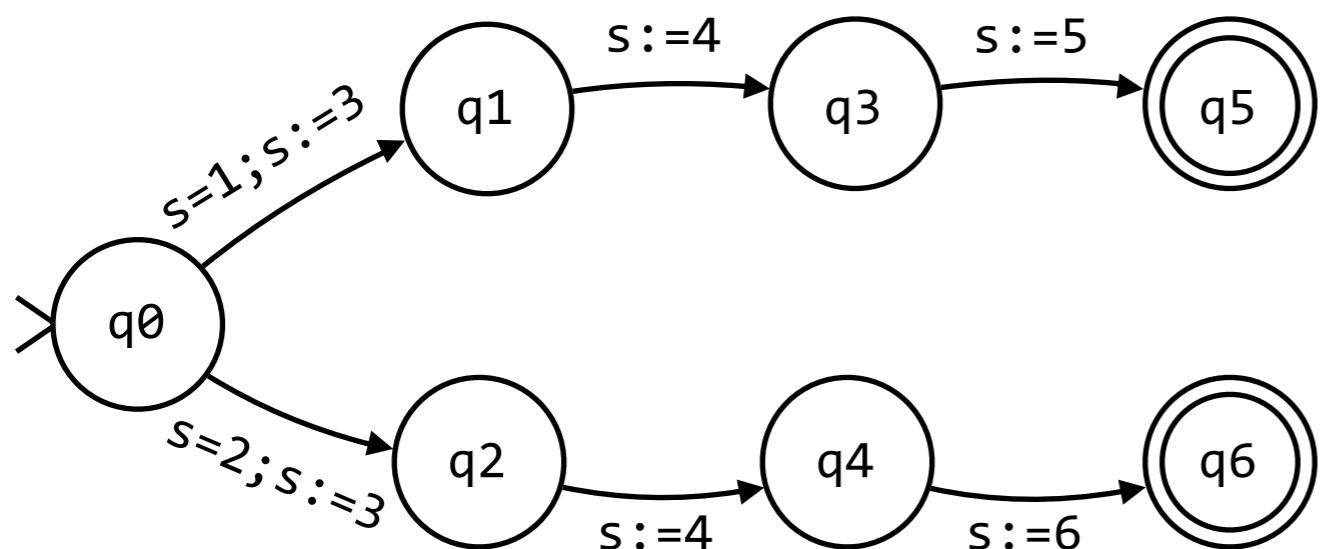
# Global program

```
(s1⇒s3⇒s4⇒s5) + (s2⇒s3⇒s4⇒s6)
```

# Global program

```
(s1⇒s3⇒s4⇒s5) + (s2⇒s3⇒s4⇒s6)
```

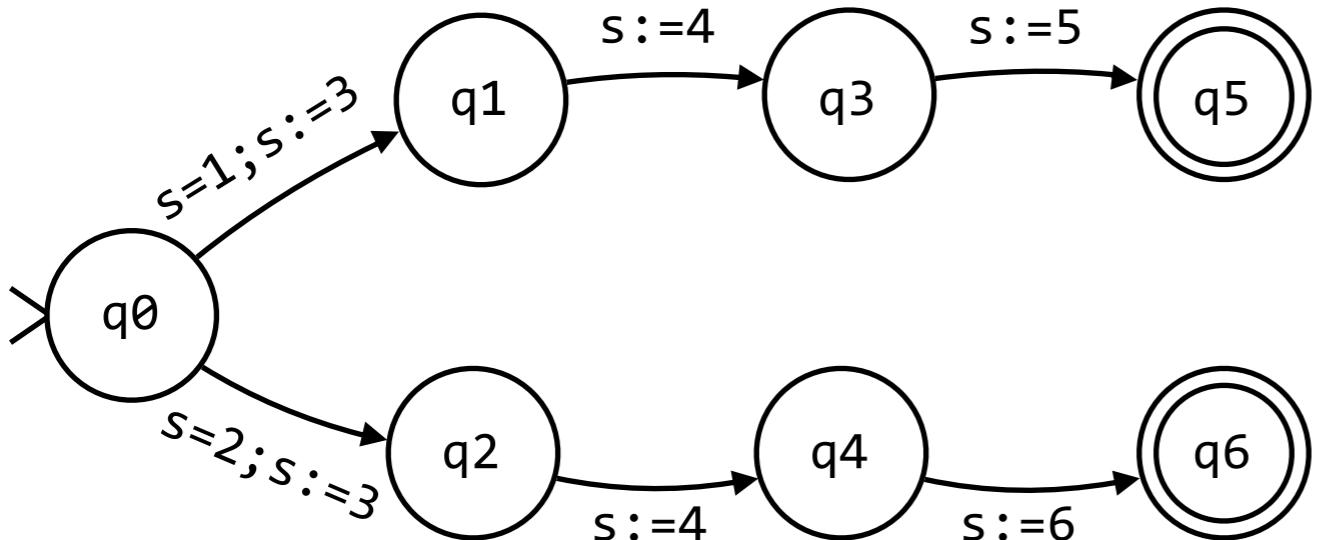
## Automaton



# Global program

$(s_1 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_5) + (s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_6)$

## Automaton



calculated using  
regular expression  
derivatives



[POPL '15]

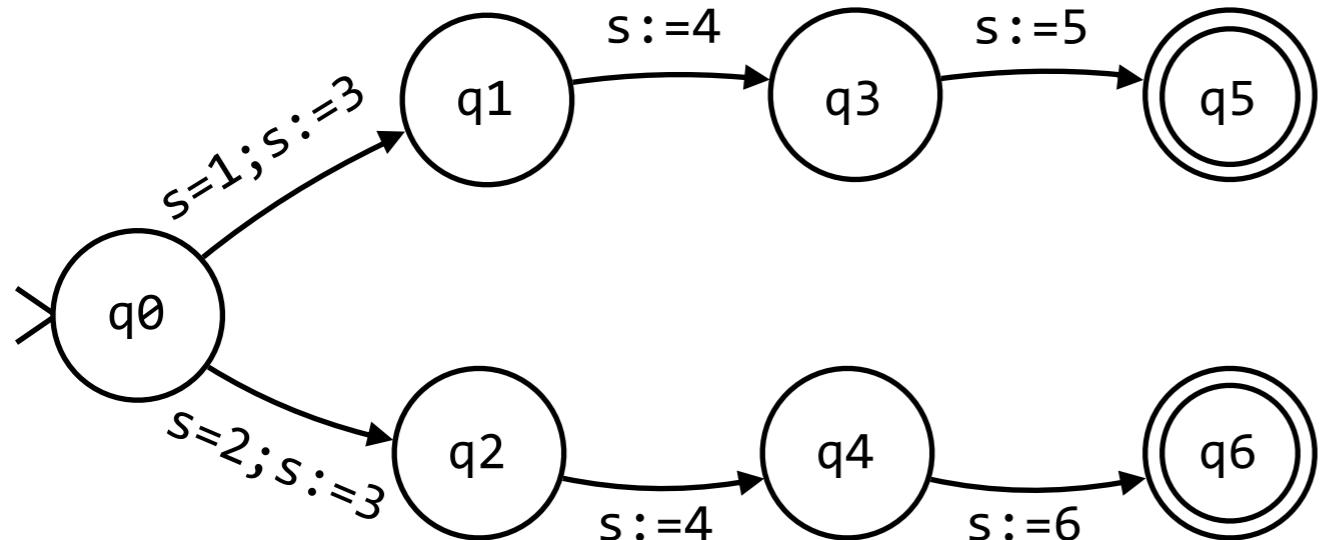


[ICFP '15]

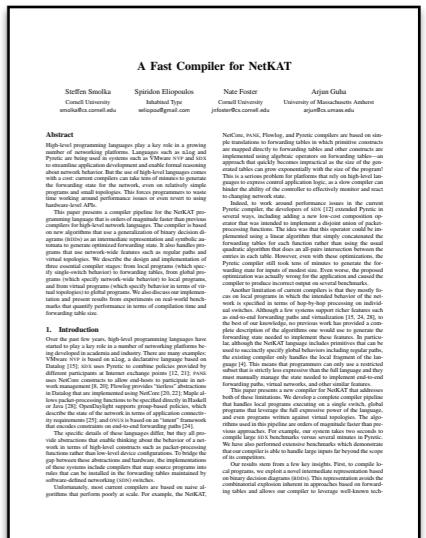
# Global program

$(s1 \Rightarrow s3 \Rightarrow s4 \Rightarrow s5) + (s2 \Rightarrow s3 \Rightarrow s4 \Rightarrow s6)$

## Automaton



calculated using  
regular expression  
derivatives



[POPL '15]

## Local program

```
s=1;pc=q0;pc:=q1 +
s=2;pc=q0;pc:=q2 +
s=3;(pc=q1;pc:=q3 + pc=q2;pc:=q4) +
s=4;(pc=q3;pc:=q5 + pc=q4;pc:=q6)
```

[ICFP '15]

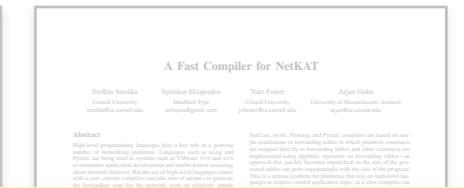
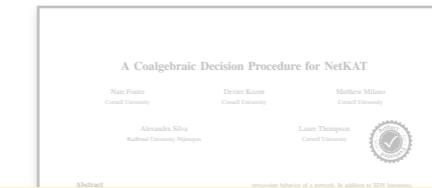
# Global program

$(s_1 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_5) + (s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_6)$

## Automaton



calculated using  
regular expression  
derivatives



Can apply global optimizations to NetKAT automata  
to reduce the forwarding complexity at each hop

[POPL '15]

[ICFP '15]

# Local program

```
s=1;pc=q0;pc:=q1 +
s=2;pc=q0;pc:=q2 +
s=3;(pc=q1;pc:=q3 + pc=q2;pc:=q4) +
s=4;(pc=q3;pc:=q5 + pc=q4;pc:=q6)
```

# Table-Based Compilation

```
let route =
  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 if ethDst = ff:ff:ff:ff:ff:ff then
    flood
  else
    port := learn
```

```
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
```



Match	Actions
ethDst=00:00:00:00:00:01	Forward 1
ethDst=00:00:00:00:00:02	Forward 2
ethDst=00:00:00:00:00:03	Forward 3
ethDst=00:00:00:00:00:04	Forward 4
ethDst=ff:ff:ff:ff:ff:ff	Flood
*	Controller

;



Match	Actions
ethSrc=00:00:00:00:00:01 ethTyp=0x800, ipProto=0x06, tcpDstPort=22	Drop
ethSrc=00:00:00:00:00:01 ethTyp=0x800, ipProto=0x06, tcpDstPort=22	Drop
*	Import

;

# Table-Based Compilation

Pattern	Actions
ethSrc=00:00:00:00:00:01, ethDst=00:00:00:00:00:01, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:01, ethDst=00:00:00:00:00:02, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:01, ethDst=00:00:00:00:00:03, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:01, ethDst=00:00:00:00:00:04, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:01, ethDst=ff:ff:ff:ff:ff:ff, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:01, ethType=0x800, ipProto=0x06, tcpPort=22	Controller
ethSrc=00:00:00:00:00:02, ethDst=00:00:00:00:00:01, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:02, ethDst=00:00:00:00:00:02, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:02, ethDst=00:00:00:00:00:03, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:02, ethDst=00:00:00:00:00:04, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:02, ethDst=ff:ff:ff:ff:ff:ff, ethType=0x800, ipProto=0x06, tcpPort=22	Drop
ethSrc=00:00:00:00:00:02, ethType=0x800, ipProto=0x06, tcpPort=22	Controller
ethDst=00:00:00:00:00:01	Forward 1
ethDst=00:00:00:00:00:02	Forward 2
ethDst=00:00:00:00:00:03	Forward 3
ethDst=00:00:00:00:00:04	Forward 4
ethDst=ff:ff:ff:ff:ff:ff	Flood
*	Controller

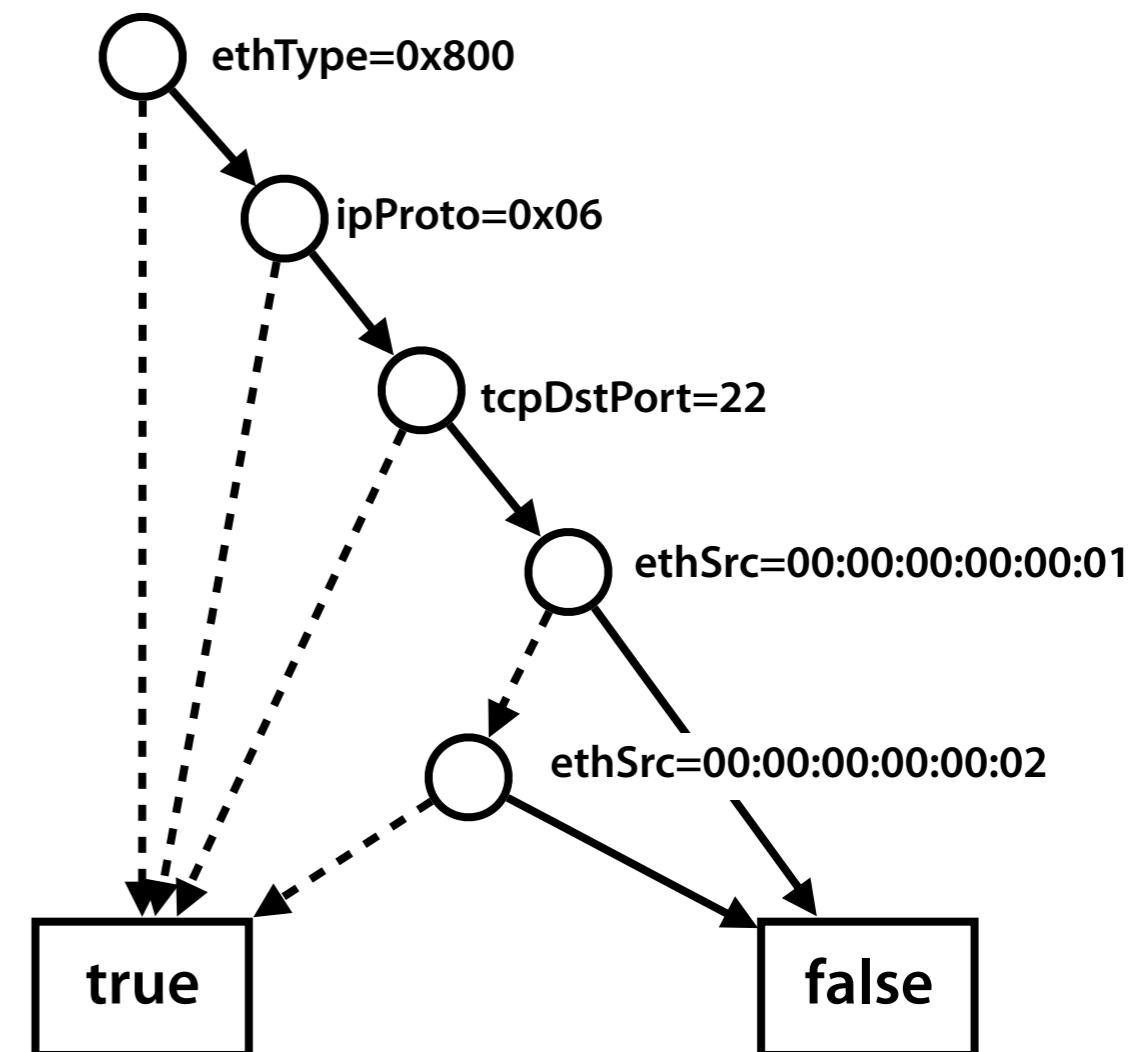
# Forwarding Decision Diagrams

**Observation:** fundamentally, NetKAT compilation involves constructing a boolean classifier on packets

**Idea:** use representation based on decision diagrams

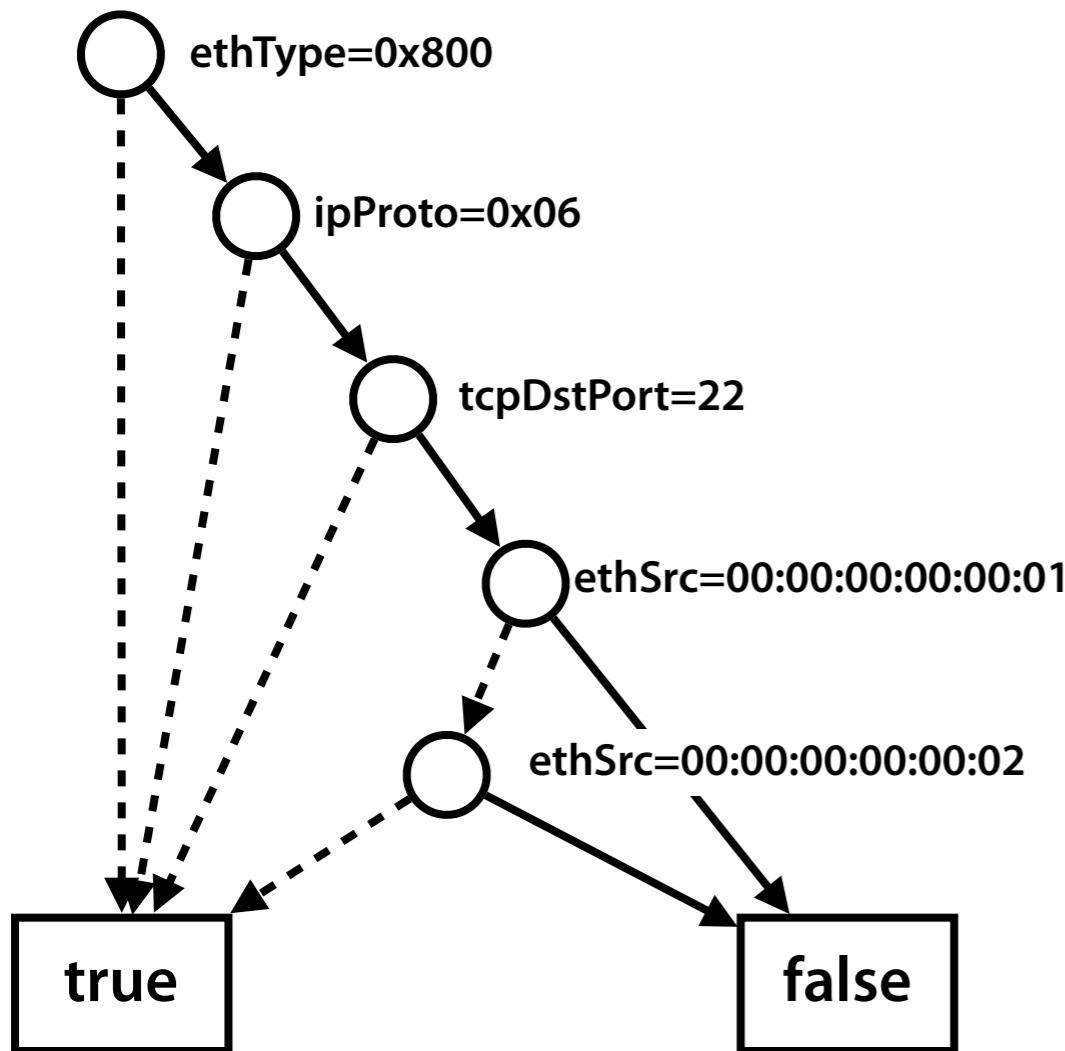
**Example:**

```
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
```



# OpenFlow Table Generation

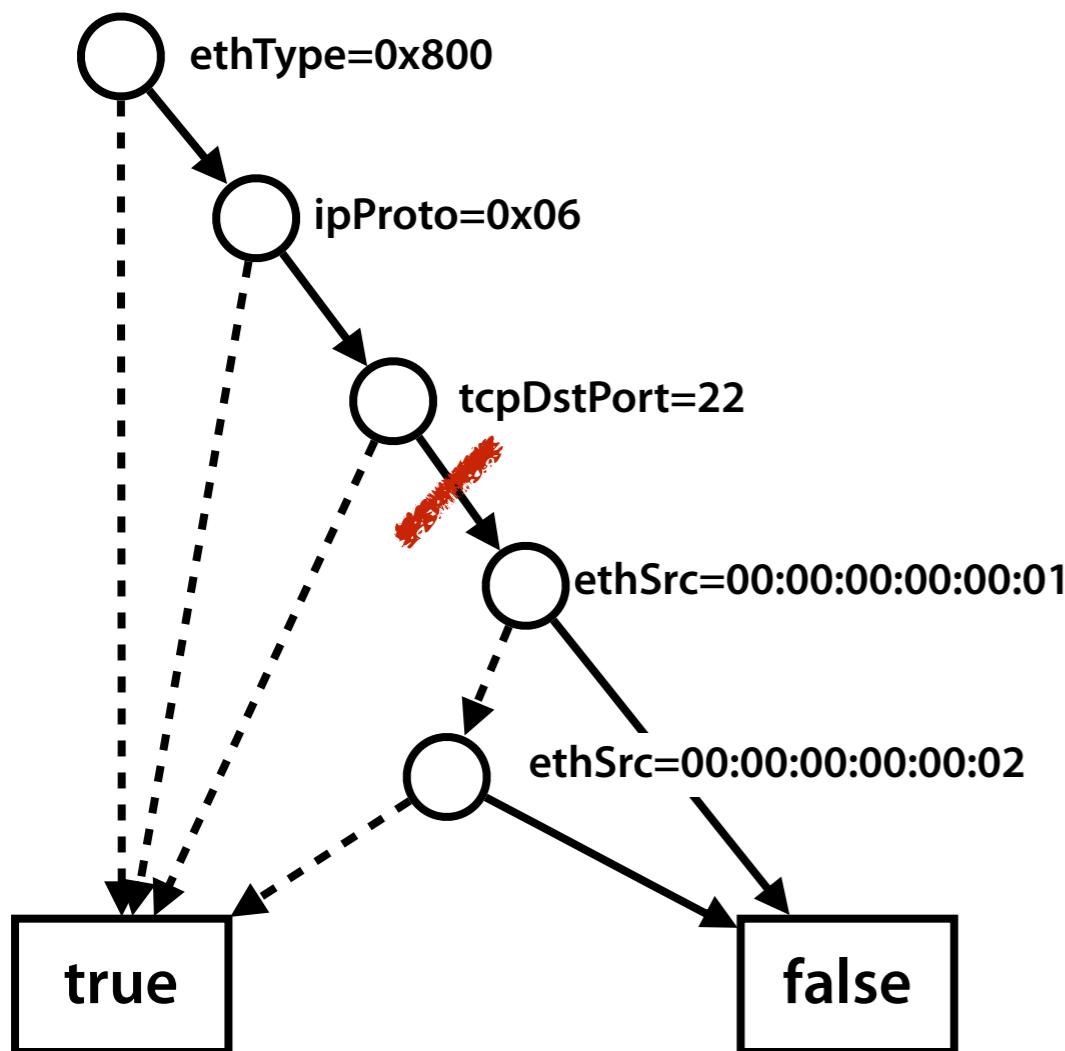
Can “read off” OpenFlow tables by enumerating paths



Match	Actions
<code>ethTyp=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:01</code>	Drop
<code>ethTyp=0x800, ipProto=0x06, tcpDstPort=22, ethSrc=00:00:00:00:00:02</code>	Drop
<code>ethTyp=0x800, ipProto=0x06, tcpDstPort=22,</code>	Import
<code>ethTyp=0x800, ipProto=0x06</code>	Import
<code>ethType=0x800</code>	Import
*	Import

# P4 Table Generation

Can generate more compact tables by splitting FDDs on fields and using metadata to track control flow



Two tables representing generated P4 rules:

SSH	
Match	Actions
<code>ethType=0x800, ipProto=0x06, tcpDstPort=22</code>	<code>ssh=1</code>
*	<code>ssh=0</code>

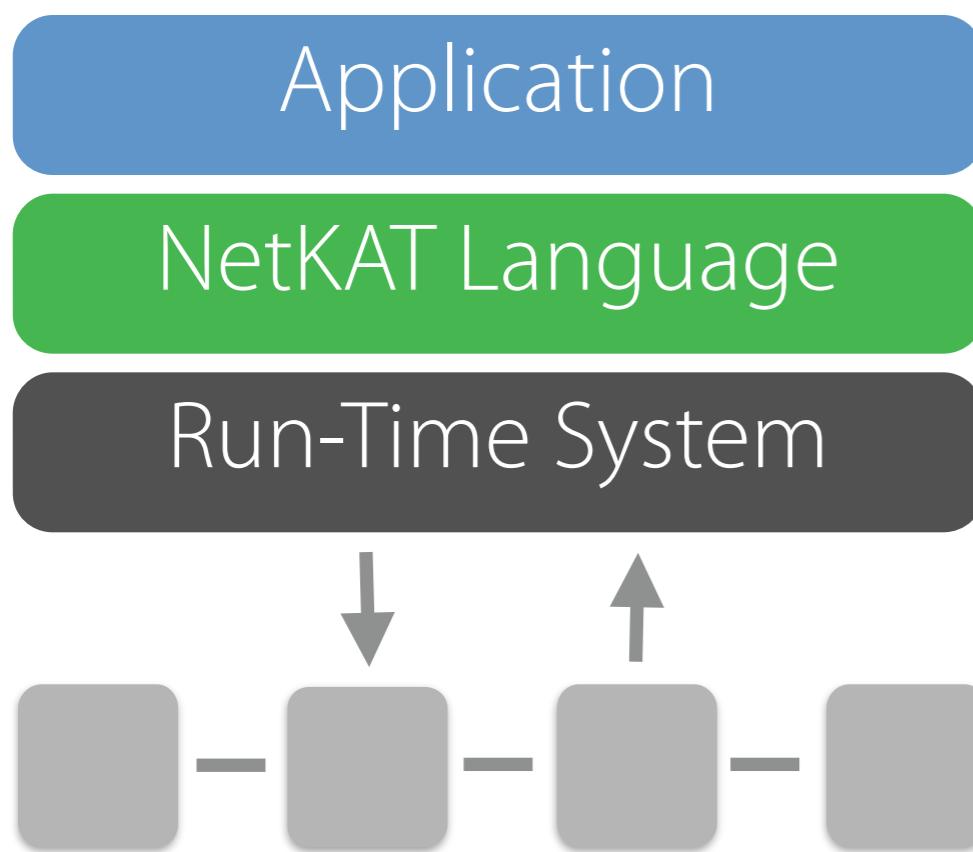
  

ACL	
Match	Actions
<code>ssh=1, ethSrc=00:00:00:00:00:01</code>	Drop
<code>ssh=1, ethSrc=00:00:00:00:00:02</code>	Drop
*	Inport

# Experience

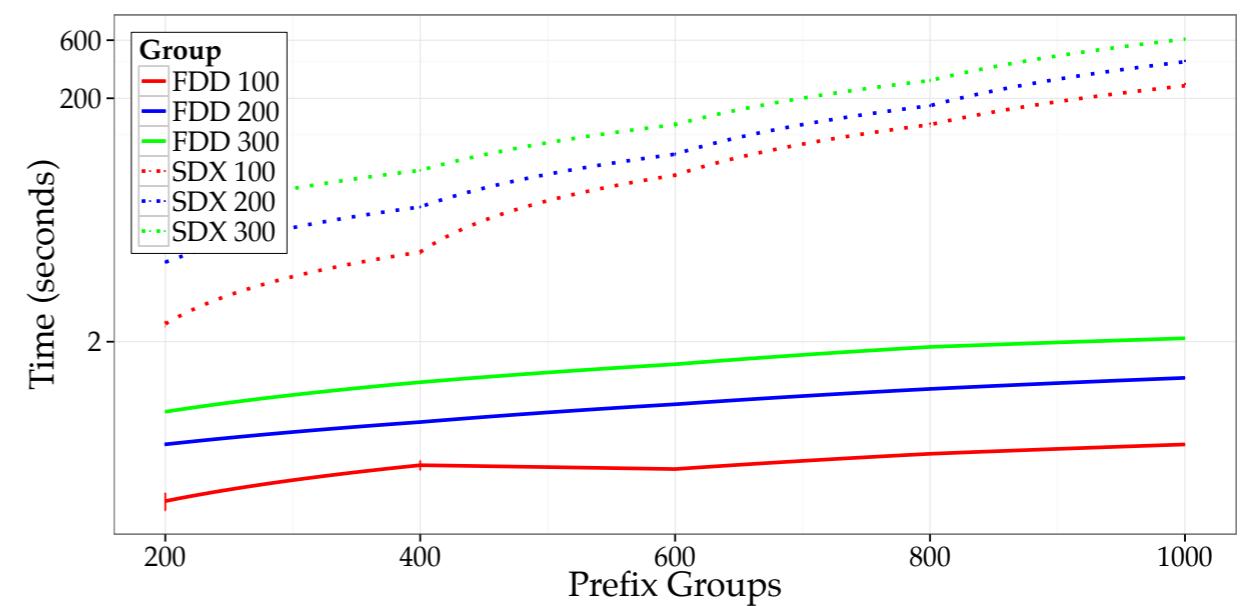
## Implementation

Frenetic controller now based on NetKAT



## Performance

Compiler is ~100X faster than its competitors

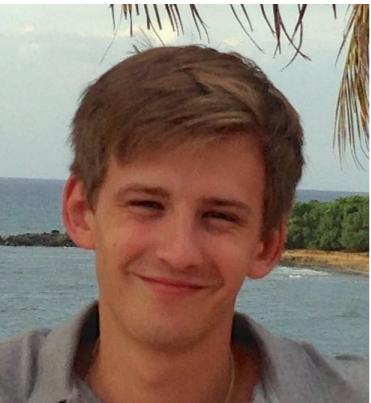


SDX [SIGCOMM '14]

# Conclusion

- High-level languages have an essential role to play in modern networking platforms
- NetKAT combines natural constructs for building packet classifiers and forwarding paths
- Can compile NetKAT efficiently using BDDs, symbolic automata, and (soon) P4 :-)

# Thank you!



Steffen Smolka  
Cornell PhD



Arjun Guha  
UMass Faculty



Spiros Eliopoulos  
Inhabited Type



Xiang Long  
Cornell PhD

## ***Other collaborators***

- Carolyn Anderson (McGill)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Matthew Milano (Cornell)
- Cole Schlesinger (Princeton)
- Alexandra Silva (Nijmegen)
- Laure Thompson (Cornell)
- Dave Walker (Princeton)

<http://frenetic-lang.org/>