Efficient Programming
Abstractions for SDN

Steffen Smolka
Networks are becoming programmable

- Rule tables
- SDN switches
- Open interface (OpenFlow / P4)
Networks are becoming programmable

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.0</td>
<td>Forward</td>
</tr>
<tr>
<td>*</td>
<td>Forward</td>
</tr>
</tbody>
</table>

SDN switches

rule tables

Machine abstraction

Open interface (OpenFlow / P4)
Networks are becoming programmable

This Talk

- High-level program
- Compiler
- Rule tables
- SDN switches
- Machine abstraction
- Open interface (OpenFlow / P4)
Language Design
Language Design

Rich Packet Classification

Network-wide Abstractions

Modular Composition
NetKAT
Model

Packets are records of values.
Programs are functions on packets.

{ switch = A, 
  port = 3, 
  ethSrc = 8:8:::8:8, 
  ethDst = 2:2:::2:2, 
  vLan = 8, 
  ipSrc = 192.168.2.1, 
  ipDst = 127.0.0.1, 
  ...
}
NetKAT Language

$\text{pol ::= false} \\
\text{true} \\
\text{field = val} \\
\text{field ::= val} \\
\text{pol}_1 + \text{pol}_2 \\
\text{pol}_1; \text{pol}_2 \\
\text{!pol} \\
\text{pol}^* \\
\text{S \rightarrow S'}$
NetKAT Language

<table>
<thead>
<tr>
<th>pol ::=</th>
<th>Boolean Algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{false}</td>
<td></td>
</tr>
<tr>
<td>\texttt{true}</td>
<td></td>
</tr>
<tr>
<td>field = val</td>
<td></td>
</tr>
<tr>
<td>field := val</td>
<td></td>
</tr>
<tr>
<td>pol₁ + pol₂</td>
<td></td>
</tr>
<tr>
<td>pol₁ ; pol₂</td>
<td></td>
</tr>
<tr>
<td>!pol</td>
<td></td>
</tr>
<tr>
<td>pol*</td>
<td></td>
</tr>
<tr>
<td>( S \rightarrow S' )</td>
<td></td>
</tr>
</tbody>
</table>
**NetKAT Language**

<table>
<thead>
<tr>
<th>pol ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{false}</td>
</tr>
<tr>
<td>\textbf{true}</td>
</tr>
<tr>
<td>field = val</td>
</tr>
<tr>
<td>field ::= val</td>
</tr>
<tr>
<td>pol_1 + pol_2</td>
</tr>
<tr>
<td>pol_1 ; pol_2</td>
</tr>
<tr>
<td>!pol</td>
</tr>
<tr>
<td>pol*</td>
</tr>
<tr>
<td>S \rightarrow S'</td>
</tr>
</tbody>
</table>

- Boolean Algebra
- Kleene Algebra
- "Regular Expressions"
NetKAT Language

\[
\text{pol ::= false | true | field = val | field := val | pol_1 + pol_2 | pol_1 ; pol_2 | !pol | pol* | S} \rightarrow S'
\]

Boolean Algebra
+ Kleene Algebra
"Regular Expressions"
+ Packet Primitives
NetKAT Language

pol ::= false | true | field = val | field := val | pol 1 + pol 2 | pol 1 ; pol 2 | !pol | pol * | S \rightarrow S'

Boolean Algebra

if p then q else r \equiv p;q + !p;r

while p do q \equiv p;q*;!p

Packet Primitives
NetKAT Semantics

\[ \text{pol ::= } \]
\[ \quad \text{false} \]
\[ \quad \text{true} \]
\[ \quad \text{field = val} \]
\[ \quad \text{field := val} \]
\[ \quad \text{pol}_1 + \text{pol}_2 \]
\[ \quad \text{pol}_1 ; \text{pol}_2 \]
\[ \quad \neg \text{pol} \]
\[ \quad \neg \text{pol} \]
\[ \quad \text{pol}^* \]
\[ \quad S \rightarrow S' \]
NetKAT Semantics

Local NetKAT: input-output behavior of switches

\[
\begin{align*}
\text{pol} &::= \\
&\mid \text{false} \\
&\mid \text{true} \\
&\mid \text{field} = \text{val} \\
&\mid \text{field} := \text{val} \\
&\mid \text{pol}_1 + \text{pol}_2 \\
&\mid \text{pol}_1 ; \text{pol}_2 \\
&\mid \neg \text{pol} \\
&\mid \text{pol}^* \\
&\mid S \rightarrow S'
\end{align*}
\]
NetKAT Semantics

Local NetKAT: input-output behavior of switches

\[ \llbracket \text{pol} \rrbracket \in \text{Packet} \rightarrow \text{Packet Set} \]

Global NetKAT: network-wide behavior

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

1 \rightarrow A \leftarrow 2

3 \rightarrow 4

5 \rightarrow B \leftarrow 6
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

Tedious for programmers… difficult to get right!

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

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

1. A

2. B

3. 3

4. 4

5. 5

6. 6

pol
Global Program

Simple and elegant!

port = 1; A \rightarrow B; \text{port} := 5

+ 

port = 2; A \rightarrow B; \text{port} := 6
Virtual Program
Virtual Program

1 5
2 6
3 4

virtual "big switch"
Even simpler!

port = 1;
port += 5

port = 2;
port += 6

Even simpler!
Virtual Program

Even simpler!

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

virtual "big switch"
Virtual Program

Even simpler!

```
port := 1;
port := 5 +
port := 2; port := 6
```
Can implement **multiple** arbitrary **virtual** networks on top of **single physical network**

Even simpler!
Compilation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=2</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=7</td>
<td>fwd 1</td>
</tr>
<tr>
<td></td>
<td>fwd 2</td>
</tr>
</tbody>
</table>
Compilation

Program

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=2</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=7</td>
<td>fwd 1</td>
</tr>
<tr>
<td>*</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>
NetKAT Compiler

NetKAT Compiler Pipeline

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

Pattern Actions:
- dstpt=2 drop
- srcpt=7 fwd 1
- * fwd 2
Local Compilation

Input: local program

Output: collection of flow tables, one per switch

Challenges: efficiency and size of generated tables
Traditional 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
Traditional Approach

```plaintext
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
```

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>src=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>src=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
<tr>
<td>*</td>
<td>Controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpSrc=22</td>
<td>Controller</td>
</tr>
<tr>
<td>tcpDst=22</td>
<td>Controller</td>
</tr>
<tr>
<td>*</td>
<td>Drop</td>
</tr>
</tbody>
</table>
Traditional 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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>src=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>src=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
<tr>
<td>*</td>
<td>Controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpSrc=22</td>
<td>Controller</td>
</tr>
<tr>
<td>tcpDst=22</td>
<td>Controller</td>
</tr>
<tr>
<td>*</td>
<td>Drop</td>
</tr>
</tbody>
</table>
Traditional 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!!
Our Approach

let \textit{route} =
    if ipDst = 10.0.0.1 then
        port := 1
    else if ipDst = 10.0.0.2 then
        port := 2
    else
        port := learn

let \textit{monitor} =
    if (tcpSrc = 22 + tcpDst = 22) then
        port := console
    else
        false
Our 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
Our 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

Efficient!
Our 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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipDst=10.0.0.1, tcpSrc=22</td>
<td>Forward 1, Controller</td>
</tr>
<tr>
<td>ipDst=10.0.0.1, tcpDst=22</td>
<td>Forward 1, Controller</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Our Approach

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

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

Key Data Structure:
Forwarding Decision Diagram
→ now widely adopted
Input: NetKAT program *(with links)*

Output: equivalent local program *(without links)*
Main Challenges

1. Adding Extra State
   "Tagging"

2. Avoiding Duplication
   (naive tagging is unsound!)
Our Solution

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

Automaton Minimization = Tag Elimination

Avoiding Duplication = Determinization

NetKAT DFA
Our Solution

Global Program

Adding Extra State
= Translation to Automaton

NetKAT NFA

Automaton Minimization
= Tag Elimination

NetKAT DFA

Avoiding Duplication
= Determinization

Local Program
NetKAT Automata [Foster et al, POPL '15]

Transition relation \( \delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet}) \)

"Alphabet size": \(|\text{Packet} \times \text{Packet}|\)

Can represent \( \delta \) *symbolically* using FDDs!

Automata construction:

*Antimirov partial derivatives & Position Automata*
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
Evaluation
This image shows a graph with various data points. The x-axis represents the number of prefix groups ranging from 200 to 1000, and the y-axis represents time in seconds ranging from 0 to 600. The graph compares the performance of different groups labeled 'FDD', 'SDX', and 'IN' with prefixes '100', '200', and '300'. The performance metrics are indicated by colored lines, with 'FDD' in red, 'SDX' in blue, and 'IN' in green. The legend is present on the left side of the graph, showing the groups and their corresponding colors.

The graph indicates that the local compiler shows a significant speedup compared to state-of-the-art systems, with an approximately 100x speedup as mentioned in the image.
In the resulting graph, we simply simulate a large fabric to work for any virtual policy the programmer may choose, since they specify local policies. Thus, the global compiler performs the porting and specialization, and use it to evaluate the global compiler behavior. Legal contracts between networks are often implemented by IXP routing policies at each switch and then generate a small set of rules. A Software-defined networking (SDN) architecture is fundamental to this, because it allows for the separation of the control and data planes. SDX, which is a language and presented an algorithm for compiling programs based on NetKAT's network-wide abstractions play key roles in the implementation of SDX. Subsequent work by Guha et al. developed a language and presented an algorithm for compiling programs based on NetKAT's network-wide abstractions. We build a translator from Pyretic to NetKAT, which is realistic firewall rules that exercise another type uses Pyretic as its language and presents several examples of routing programs at multiple overlapping paths into a unified fabric.

In general, there can be many such subgraphs and it is possible to implement other canonical rules.

The benchmarks discussed so far only use the VLAN configuration. When new hosts connect, when links fail, the global compiler ports, specializations, and presents several examples of routing programs at multiple overlapping paths into a unified fabric. We build a translator from Pyretic to NetKAT, which is realistic firewall rules that exercise another type uses Pyretic as its language and presents several examples of routing programs at multiple overlapping paths into a unified fabric.

The three fabrics give rise to three very different implementations. We build an optimizer for the flow-table generation algorithm that proposes a new application of SDX. RANS. Legal contracts between networks are often implemented by IXP routing policies at each switch and then generate a small set of rules.
Conclusion

First complete compiler pipeline for NetKAT

Virtual Compiler  Global Compiler  Local Compiler

Go ahead and use it!
(others are using it already)

Fast, Flexible, and Fully implemented in OCaml:
http://github.com/frenetic-lang/frenetic/

FUJITSU  SDX  frenetic  Pyretic