Efficient Programming Abstractions for SDN

Steffen Smolka
Networks are becoming programmable

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

This Talk

High-level program

Rule tables

SDN switches

Firewall; Route

Compiler

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.

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

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

\[
pol ::= \begin{array}{l}
\text{false} \\
\text{true} \\
\text{field} = \text{val} \\
\text{field} ::= \text{val} \\
pol_1 + \pol_2 \\
pol_1 ; \pol_2 \\
!\pol \\
\pol^* \\
S \rightarrow S'
\end{array}
\]

Boolean Algebra
# NetKAT Language

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

Boolean Algebra  
+  
Kleene Algebra  
"Regular Expressions"
NetKAT Language

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

Boolean Algebra

+ Kleene Algebra

"Regular Expressions"

+ Packet Primitives
NetKAT Language

\[ \text{pol} ::= \]
\[ \text{false} \]
\[ \text{true} \]
\[ \text{fi} \]
\[ \text{field} = \text{val} \]
\[ \text{fi} \]
\[ \text{field} ::= \text{val} \]
\[ \text{pol} \]
\[ \text{pol} + \text{pol} \]
\[ \text{pol} ; \text{pol} \]
\[ !\text{pol} \]
\[ \text{pol}^* \]

Boolean Algebra

\[ \text{if } p \text{ then } q \text{ else } r \equiv p ; q + !p ; r \]

Packet Primitives

\[ \text{while } p \text{ do } q \equiv p ; q^* ; !p \]

S → S'
NetKAT Semantics

pol ::=
  | \textbf{false}
  | \textbf{true}
  | field = val
  | field ::= val
  | pol_1 + pol_2
  | pol_1 ; pol_2
  | !pol
  | pol*
  | S \rightarrow S'
NetKAT Semantics

Local NetKAT: input-output behavior of switches

\[ \boxed{\text{[pol]} \in \text{Packet} \mapsto \text{Packet Set}} \]
NetKAT Semantics

Local NetKAT: input-output behavior of switches

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

Global NetKAT: network-wide behavior

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

1 A 2

3 4

5 B 6
Local Program

\[ \text{pol}_A \quad \text{pol}_B \]
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!
Global Program

pol
Global Program

port = 1; A ⇝ B; port := 5
port := 5 + port := 6

port = 2; A ⇝ B; port := 6

Simple and elegant!
Virtual Program
Virtual Program

virtual "big switch"
Virtual Program

Even simpler!

\[
\begin{align*}
\text{port} &= 1; \\
\text{port} &:= 5 \\
+ \\
\text{port} &= 2; \\
\text{port} &:= 6
\end{align*}
\]

virtual "big switch"
Virtual Program

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"

firewall

Even simpler!
Virtual Program

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

```
virtual "big switch"
```

```
firewall
```

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

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
NetKAT Compiler

NetKAT Compiler Pipeline

- **Virtual Compiler**
  - abstract topologies

- **Global Compiler**
  - network-wide behavior

- **Local Compiler**
  - ~ 100x faster than competitors

<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>
Local Compilation

**Input:** local program

**Output:** collection of flow tables, one per switch

**Challenges:** efficiency and size of generated tables
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**

```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
```
Our Approach

let \textit{route} =
\begin{align*}
&\text{if } \text{ipDst} = 10.0.0.1 \text{ then} \\
&\quad \text{port} := 1 \\
&\text{else if } \text{ipDst} = 10.0.0.2 \text{ then} \\
&\quad \text{port} := 2 \\
&\text{else} \\
&\quad \text{port} := \text{learn}
\end{align*}

let \textit{monitor} =
\begin{align*}
&\text{if } (\text{tcpSrc} = 22 + \text{tcpDst} = 22) \text{ then} \\
&\quad \text{port} := \text{console} \\
&\text{else} \\
&\quad \text{false}
\end{align*}
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
  learn
```

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

**Key Data Structure:**

Forwarding Decision Diagram

→ now widely adopted

### Pattern vs Actions Table

<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>
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

Global Program

Adding Extra State
= Translation to Automaton

NetKAT NFA
Our Solution

Adding Extra State = Translation to Automaton

NetKAT NFA

Avoiding Duplication = Determinization

NetKAT DFA
Our Solution

Adding Extra State = Translation to Automaton

NetKAT NFA

Automaton Minimization = Tag Elimination

NetKAT DFA

Avoiding Duplication = Determinization
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": $|\text{Packet} \times \text{Packet}|$
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
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
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
Local Compiler vs State of the Art

![Graph showing time vs prefix groups for different groups and their speeds.]

- **Group**: FDD 100, FDD 200, FDD 300, SDX 100, SDX 200, SDX 300
- **Legend**: Red for FDD, Blue for SDX, Green for FDD and SDX
- **Time (seconds)**: Y-axis
- **Prefix Groups**: X-axis
- **Legend Labels**: FDD 100, FDD 200, FDD 300, SDX 100, SDX 200, SDX 300
- **About 100x speedup**: Text at the bottom right of the graph
We build an optimizer for the flow-table generation algorithm. Once a path has been chosen, the compiler scales to these topologies too. We take approximately 30 seconds to produce tables for 48-pod topologies in our dataset.

An SDN controller needs to generate new flow tables for the entire network and uses it to generate flow tables for each switch and then generate a small configuration file for the fabric.

We build a translator from Pyretic to NetFPGA. The global compiler leverages multiple overlapping paths into a unified fabric. From the source node to the target node there can be many such subgraphs and it is possible to remove all paths that can lead to such a fatal state for the entire network. We have confirmed this bug and it is a very powerful abstraction. In principle, our compiler can even specialize the generated code exactly the set of all winning strategies for a given network.

Intuitively, we define such states to be fatal since we want the SDN controller to be able to re-route traffic. In general, there can be many such subgraphs and it is possible to remove all paths that can lead to such a fatal state for the entire network. We have confirmed this bug and it is a very powerful abstraction. In principle, our compiler can even specialize the generated code exactly the set of all winning strategies for a given network. We have confirmed this bug and it is a very powerful abstraction. In principle, our compiler can even specialize the generated code exactly the set of all winning strategies for a given network.

There are two things to measure: compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark. The switch-localization and their effects on compilation time may matter. We do not report running times for local compilation time on the same benchmark.
Conclusion

First *complete* compiler pipeline for NetKAT

- Virtual Compiler
- Global Compiler
- Local Compiler

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

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