It's the last COS 326 class!

David Walker
COS 326
Princeton University
COS 326 Final Exam

Logistics:
– Friday Jan 26
– 1:30pm
– McCosh 46

Note: If you are doing study abroad, make sure that you email Chris Moretti so we can arrange the exam abroad. (Many of you have.)
COS 326 Final Exam

Contents:

– The entire semester
  • the lectures
  • the assignments
– There will be more emphasis on the 2\textsuperscript{nd} half
– I will probably ask a question that is similar to something on the midterm
  • so make sure you know that stuff
Major Topics From 2\textsuperscript{nd} Half

Modules
  – signatures, structures, functors
Reasoning about modules
  – representation invariants
  – abstraction functions
  – proofs of module equivalence
Laziness, memoization
Abstractions for parallel FP
  – futures, sequences, map, reduce
  – parallel functional algorithms, work, span
Precept this Week

• A couple of questions from the 2015 exam
The Frenetic Project: Adventures in Functional Networking

David Walker
COS 326
Princeton University
Course Themes

• Functional vs. imperative programming
  – a new way to think about the algorithms you write
• Modularity
• Abstraction
• Parallelism
• Equational reasoning

Useful on a day-to-day basis and in research to transform the way people think about solving programming problems:
Cornell:
- Faculty: Nate Foster, Dexter Kozen, Gun Sirer
- Students & Post Docs: Carolyn Anderson, Shrutarshi Basu, Mark Reitblatt, Robert Soule, Alec Story

Princeton:
- Faculty: Jen Rexford, Dave Walker
- Students & Post Docs: Ryan Beckett, Jennifer Gossels, Rob Harrison, Xin Jin, Naga Katta, Chris Monsanto, Srinivas Narayana, Josh Reich, Cole Schlesinger

UMass:
- Faculty: Arjun Guha

http://frenetic-lang.org
A Quick Story Circa 2009 @ Princeton

Dave:
Hey Jen, what's networking?

Jen:
Oooh, it's super-awesome.
No lambda calculus required!

Nate:
Too bad about the lambda calculus.
But fill us in.
What is Networking?

end-hosts need
to communicate
What is Networking?

Ethernet switches connect them
What is Networking?

which decide how packets should be forwarded

Control Plane
What is Networking?

and actually forward them
A Quick Story Circa 2009 @ Princeton

Nate:
Sounds simple enough. Is that it?

Jen:
There's a little more ...
Still no lambda calculus though.

Dave:
Darn.
What is Networking?

add servers ...
connected by routers
What is Networking?

add servers ... connected by routers

plug-and-play

different control planes

structured and optimized
What is Networking?

add servers ... connected by routers

w/ similar data planes
What is Networking?

we need gateway to bridge them
What is Networking?

and load balancing
for servers
What is Networking?

there are other ISPs
What is Networking?

requiring inter-domain routers
What is Networking?

and a firewall to handle malicious traffic
What is Networking?

and mobile endpoints
What is Networking?

requiring wireless base stations
What is Networking?

and more middleboxes for billing, lawful intercept, DPI
A Quick Story Circa 2009 @ Princeton

Dave:

??? Lambda calculus is easier.

Jen:

:-) Big mess, eh?

... but there is a new way to do things ...
This is a Control Plane Issue

each color represents a different set of control-plane protocols and algorithms
The Data Planes are Similar
Software Defined Networks

decouple control and data planes by providing open standard API
Centralize Control

Controller Application

Controller Platform

OpenFlow
OpenFlow Data Plane Abstraction

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Priority</th>
<th>Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip = 1.2.<em>, dstip = 3.4.5.</em></td>
<td>drop</td>
<td>1</td>
<td>76</td>
</tr>
<tr>
<td>srcip = <em>.</em>.<em>.</em>, dstip = 3.4.5.*</td>
<td>fwd 2</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>srcip = <em>.</em>.<em>.</em>, dstip = <em>.</em>.<em>.</em></td>
<td>controller</td>
<td>3</td>
<td>22</td>
</tr>
</tbody>
</table>

Operations:
- Install rule
- Uninstall rule
- Ask for counter values

The Payoff:
- Simplicity
- Generality
OpenFlow

Events up:
- Topology changes
- Traffic statistics
- Unprocessed arriving packets

Commands down:
- Install rule
- Uninstall rule
- Query statistics
- Send packets
The Payoff

Simple, open interface:

– Easy to learn: *Even I can do it!*

– Enables rapid innovation by academics and industry

– Everything in the data center can be optimized
  • The network no longer "gets in the way"

– Commoditize the hardware
Huge Momentum in Industry

Entire backbone runs OpenFlow
Bought for $1.2 \times 10^9$
(mostly cash)
A Quick Story Circa 2009
@ Princeton

Jen:
So … SDN is a big deal.

Dave:
Cool. Let's get this party started.
The PL Perspective:
A new piece of our critical infrastructure is now available for programming

24-7 availability:
- correct-by-construction abstractions
- defect detection
- verification
- testing
- fault tolerance

A new kind of heterogeneous distributed system

multi-component applications:
- modularity
- composition
- abstraction
- information hiding

resource constraints:
- optimization problems

Controller Application

Controller Platform

shared/used by multiple entities
- security

simple, clean, narrow interface:
- a new assembly language
- ... needing domain-specific abstractions
A DSL for modular network configuration [ICFP 11, POPL 12, NSDI 13, POPL 14, NSDI 15]
The Biggest Problem: Modularity

We still need all the functionality of old networks: The only way to engineer it is through modular design.
OpenFlow is Anti-Modular

Controller Application

Repeater Module

Monitoring Module

inport =1 → fwd 2
inport =2 → fwd 1

Query web traffic:
inport = 1, dstport = 80 ?

P installed

Bottom Line: It doesn’t work:
• repeater rules are too coarse-grained for desired monitoring
• installing new monitoring rules will clobber the repeater actions
Anti-Modularity: A Closer Look

### Repeater

```python
def switch_join(switch):
    repeater(switch)

def repeater(switch):
    pat1 = {in_port:1}
    pat2 = {in_port:2}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2, DEFAULT, None, [output(1)])
```

### Web Monitor

```python
def monitor(switch):
    pat = {in_port:2, tp_src:80}
    install(switch, pat, DEFAULT, None, [])
    query_stats(switch, pat)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)
```

### Repeater/Monitor

```python
def switch_join(switch):
    repeater_monitor(switch)

def repeater_monitor(switch):
    pat1 = {in_port:1}
    pat2 = {in_port:2}
    pat2web = {in_port:2, tp_src:80}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2, DEFAULT, None, [output(1)])
    install(switch, pat2web, HIGH, None, [output(1)])
    query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)
```

- **Blue**: from repeater
- **Red**: from web monitor
- **Green**: from neither
OpenFlow is Anti-Modular

You can’t (easily and reliably) compose:

- a billing service with a repeater
- a firewall with a switch
- a load balancer with a router
- one broadcast service with another
- policy for one data center client with another
Solution: Functional Programming!

Stop thinking imperatively:
• Don’t program with update/delete commands for concrete rules
And lift the level of abstraction:
• Use pure functions as data structures that describe network forwarding policy
• Provide primitives to build complex policies from simple ones
• Let a compiler and run-time do rule synthesis & installation
Frenetic Architecture

- **Process Event**
  - Generate Policy
  - Network-wide Policy

- **Receive Event**
  - Compile Policy
  - Messages to Switches

- **frenetic application program**

- **controller platform + run time**

- **Topology Change / Network Stat / Packet In**

- **controller platform** + **run time**

- **Policy**
  - Messages to Switches
Rather than managing (un)installation of concrete rules, programmers specify what a network does using *pure functions*.

\[ f : \text{located}_{-}\text{packet} \rightarrow \text{located}_{-}\text{packet set} \]

location = (switch, port)

controller

count? bytes? packet contents?

location = bucket b
Frenetic Policy Language
[Phase 1]

Rather than managing (un)installation of concrete rules, programmers specify what a network does using *pure functions*.

\[ f : \text{located\_packet} \rightarrow \text{located\_packet set} \]
## Firewalls: The Simplest Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Explanation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>drops all packets</td>
<td>fun p -&gt; { }</td>
</tr>
<tr>
<td>true</td>
<td>admits all packets</td>
<td>fun p -&gt; { p }</td>
</tr>
<tr>
<td>srcIP=10.0.0.1</td>
<td>admits packets with srcIP = 10.0.0.1 drops others</td>
<td>fun p -&gt; if p.srcIP = 10.0.0.1 then { p } else { }</td>
</tr>
<tr>
<td>q1 ∧ q2,</td>
<td>admits packets satisfying q1 ∧ q2,</td>
<td>fun p -&gt; (q1 p) U (q2 p)</td>
</tr>
<tr>
<td>q1 ∨ q2,</td>
<td>q1 ∨ q2,</td>
<td>fun p -&gt; (q1 p) Π (q2 p)</td>
</tr>
<tr>
<td>¬q</td>
<td>¬q</td>
<td>fun p -&gt; match (q1 p) with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ -&gt; { }</td>
</tr>
</tbody>
</table>
Firewalls: The Simplest Policies

Example: Block all packets from source IP 10.0.0.1 and 10.0.0.2 and except those for web servers

Solution: ~((srcIP=10.0.0.1 ∧ srcIP=10.0.0.2) ∨ tcp_src_port = 80)

web traffic sent here
Firewalls: The Simplest Policies

Example: Allow traffic coming in to switches A, port 1 and switch B, port 2 to enter our network. Block others.

Solution: \((\text{switch}=A \land \text{inport}=1) \lor (\text{switch}=B \land \text{inport}=2)\)
Moving Packets from Place to Place

<table>
<thead>
<tr>
<th>Policy</th>
<th>Explanation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwd 2</td>
<td>forward all packets out port 2</td>
<td>fun p -&gt; { p[port:= 2] }</td>
</tr>
</tbody>
</table>
Combining Policies

**Policy**

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>port=1; fwd 2</strong></td>
<td>only consider packets with port = 1 then forward all such packets out port 2</td>
</tr>
</tbody>
</table>

**Function**

```
let filter_port x p = if p.port = x then { p } else {} in
let fwd x p = p.port <- x in
(filter_port 1) <> (fwd 2)
```

**where:**

```
a <> b = fun packet ->
let s = a packet in
Set.Union (Set.map b s)
```
Multiple Flows

Policy |
--- |
(port=1; fwd 2) + (if port = 1 then forward out port 2) and also
(port=2; fwd 3) (if port = 1 then forward out port 2)

Explanation

Function

(filter_port 1 <> fwd 2) +
(filter_port 2 <> fwd 3)

where:

(+ a b = fun packet -> Set.Union
{(a packet),
(b packet)})
Composing Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>let policyA =</td>
<td>(if port = 1 then forward out port 2) and also</td>
</tr>
<tr>
<td>(port=1; fwd 2) +</td>
<td>(if port = 1 then forward out port 3)</td>
</tr>
<tr>
<td>(port=2; fwd 3)</td>
<td></td>
</tr>
<tr>
<td>let policyB =</td>
<td>(if port = 1 then forward out port 3)</td>
</tr>
<tr>
<td>port=2; fwd 3</td>
<td></td>
</tr>
<tr>
<td>(switch = A; policyA) +</td>
<td>(if switch=A then policyA) and also</td>
</tr>
<tr>
<td>(switch = B; policyB)</td>
<td>(if port = 1 then policyB)</td>
</tr>
</tbody>
</table>

Diagram:
- Switch A:
  - Port 1
  - Port 2
  - Port 3
- Switch B:
  - Port 1
  - Port 2
  - Port 3

Connections:
- Port 1 of A to Port 2 of B
- Port 2 of A to Port 3 of B
More Composition: Routing & Monitoring

**router** =
  dstip = 1.2.* ; fwd 1
  + dstip = 3.4.* ; fwd 2

**monitor** =
  srcip = 5.6.7.8 ; bucket b1
  + srcip = 5.6.7.9 ; bucket b2

**app** = **monitor** + **router**
Server Load Balancing

Goal: Spread client traffic over server replicas
Setup: Advertise public IP address for the service

First: Split traffic on client IP & rewrite the server IP address
Then: Route to the replica

1.2.3.4 1.2.3.4

clients load balancer server replicas

10.0.0.1 10.0.0.2 10.0.0.3
Sequential Composition

\[
\text{selector} = \\
\text{srcip} = 0^* \land \text{dstip}=1.2.3.4; \\
\text{dstip} \leftarrow 10.0.0.1 \\
+ \\
\text{srcip} = 1^* \land \text{dstip}=1.2.3.4; \\
\text{dstip} \leftarrow 10.0.0.2
\]

\[
\text{forwarder} = \\
\text{dstip} = 10.0.0.1; \text{fwd} 1 \\
+ \\
\text{dstip} = 10.0.0.0; \text{fwd} 2
\]

\[
\text{lb} = \text{selector} ; \text{forwarder}
\]
Summary So Far

predicates:
q ::= f = pattern
| true
| false
| q1 ∧ q2
| q1 ∨ q2
| ¬q

simple actions:
a ::= fwd n
| f ← v
| bucket b

network policies:
p ::= a
| q
| p1 + p2
| p1 ; p2

(action)
(filter)
(parallel comp.)
(sequential comp.)

abbreviations:
if q then p1 else p2 == (q; p1) + (~q; p2)
id == true
drop == false
fwd p == port <- p
Equational Theory

A sign of a well-conceived language == a simple equational theory

\[
\begin{align*}
P + Q & \equiv Q + P & (+ \text{ commutative}) \\
(P + Q) + R & \equiv P + (Q + R) & (+ \text{ associative}) \\
P + \text{drop} & \equiv P & (+ \text{ drop unit}) \\
(P ; Q) ; R & \equiv P ; (Q ; R) & (; \text{ associative}) \\
id ; P & \equiv P & (; \text{ id left unit}) \\
P ; id & \equiv P & (; \text{ id right unit}) \\
\text{drop} ; P & \equiv \text{drop} & (; \text{ drop left zero}) \\
P ; \text{drop} & \equiv \text{drop} & (; \text{ drop right zero}) \\
\text{if } q \text{ then } (P ; R) \text{ else } (Q ; R) & \equiv (\text{if } q \text{ then } P \text{ else } Q) ; R & (\text{if commutes } ;)
\end{align*}
\]
A Simple Use Case
(Modular Reasoning)

firewall =
  if srcip = 1.1.1.1 then drop
  else id

router = ...

app = firewall ; router

app == firewall ; router
== (if srcip = 1.1.1.1 then drop else id) ; router
== if srcip = 1.1.1.1 then (drop ; router) else (id ; router)
== if srcip = 1.1.1.1 then drop else (id ; router)
== if srcip = 1.1.1.1 then drop else router
But what if we want to reason about entire networks?

Are all SSH packets dropped at some point along their path?

*Do all non-SSH packets sent from H1 arrive at H2?*

Are the optimized policies equivalent to the unoptimized one?
Encoding Topologies

\[ t = (sw = A \land pt = 2; \; sw \leftarrow B; \; pt \leftarrow 1) + (sw = B \land pt = 1; \; sw \leftarrow A; \; pt \leftarrow 2) \]

\[ net = pol; \; t; \; pol \]
Encoding Topologies

\[ t = \ldots \]

\[ \text{net} = (\text{pol}; t)^*; \text{pol} \]

Kleene iteration:
\[ p^* = \text{id} + p + pp + \ldots \]
Encoding Networks

pol = ...
t = ...
net = (pol; t)*; pol

net is a function that moves packets:
A1 ==> B2
B2 ==> A1

edge = sw=A & pt=1
|| sw=B & pt=2

and also moves packets:
A1 ==> A2
A2 ==> A1
B1 ==> B2
B2 ==> B1
## Summary So Far

### Policies

<table>
<thead>
<tr>
<th>p, q, r ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>a // filter according to a</td>
</tr>
<tr>
<td>f &lt;- v // update field f to v</td>
</tr>
<tr>
<td>p ; q // do p then q</td>
</tr>
<tr>
<td>p + q // do p and q in parallel</td>
</tr>
<tr>
<td>p* // do p zero or more times</td>
</tr>
</tbody>
</table>

### Predicates

<table>
<thead>
<tr>
<th>a, b, c ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop // drop all packets</td>
</tr>
<tr>
<td>id // accept all packets</td>
</tr>
<tr>
<td>f = v // field f matches v</td>
</tr>
<tr>
<td>~a // negation</td>
</tr>
<tr>
<td>a &amp; b // conjunction</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>

### Network Encoding

in; (policy; topology)*; policy; out
Summary So Far

Kleene Algebra

\[ p, q, r := \]

\[ a \] // filter according to a

\[ f \leftarrow v \] // update field f to v

\[ p ; q \] // do p then q

\[ p + q \] // do p and q in parallel

\[ p^* \] // do p zero or more times

Boolean Algebra

\[ a, b, c := \]

\[ \text{drop} \] // drop all packets

\[ \text{id} \] // accept all packets

\[ f = v \] // field f matches v

\[ \sim a \] // negation

\[ a \& b \] // conjunction

\[ a \mid\mid b \] // disjunction

Network Encoding

Policies

Boolean Algebra + Kleene Algebra

= Kleene Algebra with Tests
Equational Theory

\[ \text{net1} \approx \text{net2} \]

For programmers:
- a system for reasoning about programs as they are written

For compiler writers:
- a means to prove their transformations correct

For verifiers:
- sound and complete with a PSPACE decision procedure
## Equational Theory

**Boolean Algebra:**
- \(a \land b \approx b \land a\)
- \(a \land \neg a \approx \text{drop}\)
- \(a \lor \neg a \approx \text{id}\)

**Kleene Algebra:**
- \((a; b); c \approx a; (b; c)\)
- \(a; (b + c) \approx (a; b) + (a; c)\)
- \(p^* \approx \text{id} + p; p^*\)

**Packet Algebra:**
- \(f <- n; f = n \approx f <- n\)
- \(f = n; f <- n \approx f = n\)
- \(f <- n; f <- m \approx f <- m\)
- \(f = 0 + \ldots + f = n \approx \text{id}\) (finite set of possible values in \(f\))
Using the Theory

Are all SSH packets dropped?

\[ \text{forward} = (\text{dst} = \text{H1}; \text{pt} \leftarrow 1) + (\text{dst} = \text{H2}; \text{pt} \leftarrow 2) \]

\[ \text{ac} = \sim(\text{typ} = \text{SSH}); \text{forward} \]

\[ \text{t} = \ldots \]

\[ \text{edge} = \ldots \]

\[ \text{net} = \text{edge}; (\text{ac}; \text{t})*; \text{ac}; \text{edge} \]

Do all non-SSH packets sent from H1 arrive at H2?

\[ \sim\text{typ} = \text{SSH}; \text{sw} = \text{A}; \text{pt} \leftarrow 1 \]

\[ \approx \]

\[ \sim\text{typ} = \text{SSH}; \text{sw} = \text{B}; \text{pt} \leftarrow 2 \]
Using the Theory

Are all SSH packets dropped?

`typ = SSH; net ≈ drop`

Do all non-SSH packets destined for H2, sent from H1 arrive at H2?

`~typ = SSH; dst = H2; sw=A; pt=1; net ≈`

`~typ = SSH; dst = H2; sw=A; pt=1; sw <- B; pt <- 2`

forward = (dst = H1; pt <- 1) + (dst = H2; pt <- 2)

ac = ~(typ = SSH); forward

t = ...

edge = ...

net = edge; (ac; t)*; ac; edge
Traffic Isolation

Programmer 1 connects H1 and H2:

polA1 = sw = A; ( pt = 1; pt <- 2 + pt = 2; pt <- 1 )

polB1 = sw = B; ( ... )

pol1 = polA1 + polB1

net1 = (pol1; t)*

Programmer 2 connects H3 and H4:

polA2 = sw = B; ( pt = 3; pt <- 2 + pt = 1; pt <- 3 )

polB2 = sw = A; ( ... )

pol2 = polA2 + polB2

net2 = (pol2; t)*

net3 = ((pol1 + pol2); t)*  // traffic from H2 goes to H1 and H4!
Traffic Isolation

A network slice is a light-weight abstraction designed for traffic isolation:

- traffic outside the slice satisfying \textit{in} enters the slice
- traffic inside the slice obeys the policy
- traffic inside the slice satisfying \textit{out} exits the slice

slices are just a little syntactic sugar on top of NetKAT
A network slice is a light-weight abstraction designed for traffic isolation:

edge1 = sw = A \land pt = 1 \lor sw = B \land pt = 2

slice1 = \{edge1\} pol1 \{edge1\}

edge2 = sw = A \land pt = 3 \lor sw = B \land pt = 3

slice2 = \{edge2\} pol2 \{edge2\}

Theorem: (slice1; t)* + (slice2; t)* \approx ((slice1 + slice2); t)*

packet copied and sent through slice1 and slice2 networks separately

packet runs through network that combines slice1 and slice2
A network slice is a light-weight abstraction designed for traffic isolation:

**Theorem:** edge1; (slice1; t)* ≈ edge1; ((slice1 + slice2); t)*

Consider those packets at the edge1 of the slice:

- edge1 = sw = A \land pt = 1 \lor sw = B \land pt = 2
- slice1 = {edge1} pol1 {edge1}

Can’t tell the difference between slice1 alone and slice1 + slice2:
NetKAT can be implemented with OpenFlow

forward =  
(dst = H1; pt <- 1) 
+ (dst = H2; pt <- 2)

ac =  
~(typ = SSH); forward

Flow Table for Switch 1:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>typ = SSH</td>
<td>drop</td>
</tr>
<tr>
<td>dst=H1</td>
<td>fwd 1</td>
</tr>
<tr>
<td>dst=H2</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>

Flow Table for Switch 2:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>typ = SSH</td>
<td>drop</td>
</tr>
<tr>
<td>dst=H1</td>
<td>fwd 1</td>
</tr>
<tr>
<td>dst=H2</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>

Theorem: Any NetKAT policy \( p \) that does not modify the switch field can be compiled into an equivalent policy in “OpenFlow Normal Form.”
Moving Forward

Multiple implementations:

– In OCaml:
  • Nate Foster, Arjun Guha, Mark Reitblatt, and others!
  • https://github.com/frenetic-lang/frenetic

See www.frenetic-lang.org
Moving Forward

Propane [SIGCOMM 2016, best paper]
- a language for configuring BGP routers
- similar abstractions to NetKAT; different compilation strategies

Synthesizing Protocols [in progress]
- abstractions for load-sensitive routing
- synthesis of load-sensitive distributed protocols
<table>
<thead>
<tr>
<th>Concern</th>
<th>Assembly Languages</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x86</td>
<td>NOX</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Move values to/from register</td>
<td></td>
</tr>
<tr>
<td>Modularity</td>
<td>Unregulated calling conventions</td>
<td></td>
</tr>
<tr>
<td>Consistency</td>
<td>Inconsistent memory model</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td>Hardware dependent</td>
<td></td>
</tr>
<tr>
<td>Concern</td>
<td>Assembly Languages</td>
<td>Programming Languages</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td>x86</td>
<td>NOX</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Move values to/from register</td>
<td>(Un)Install policy rule-by-rule</td>
</tr>
<tr>
<td>Modularity</td>
<td>Unregulated calling conventions</td>
<td>Unregulated use of network flow space</td>
</tr>
<tr>
<td>Consistency</td>
<td>Inconsistent memory model</td>
<td>Inconsistent global policies</td>
</tr>
<tr>
<td>Portability</td>
<td>Hardware dependent</td>
<td>Hardware dependent</td>
</tr>
</tbody>
</table>
Summary

FUNCTIONAL NETWORK PROGRAMMERS: 326

OTHER NETWORK PROGRAMMERS: 0