The Next 700 Network Programming Languages

Nate Foster
Cornell University

ACM SIGCOMM NetPL ‘16
NetPL
workshop (Networking & Programming Languages)
{ Brazil, 2016 }
Operating Systems
The Next 700 Programming Languages

P. J. Landin

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

"... today ... 1,200 special programming languages used to "bootstrap" in over 700 application areas."—Computer Software Environments, 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) sub-system that gives 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 hypertext of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies 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 more syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation. If the dialects are well disciplined, they might with luck be characterized as

![Image](1024x768)

The Next 700 Programming Languages

P. J. Landin

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

"... today ... 1,200 special programming languages used to "bootstrap" in over 700 application areas."—Computer Software Environments, 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) sub-system that gives 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 hypertext of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies 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 more syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation. If the dialects are well disciplined, they might with luck be characterized as

![Image](1024x768)
“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
• 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
- Regular expressions
- 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

\[ \text{pol} ::= \text{false} \mid \text{true} \mid f = n \mid f := n \mid \text{pol}_1 + \text{pol}_2 \mid \text{pol}_1 \cdot \text{pol}_2 \mid !\text{pol} \mid \text{pol}^* \mid \text{dup} \]
NetKAT Syntax

```
pol ::= false
    | true
    | f = n
    | f := n
    | pol₁ + pol₂
    | pol₁ • pol₂
    | !pol
    | pol*
    | dup
```

Boolean Predicates

- +

Regular Expressions

- +

Packet Primitives

Negation may only be applied to Boolean predicates:

true, false, f = n, closed under +, •, and !
NetKAT Syntax

\[
pol ::= \text{false} \\
| \text{true} \\
| f = n \\
| f := n \\
| pol_1 + pol_2 \\
| pol_1 \cdot pol_2 \\
| !pol \\
| pol^* \\
| \text{dup} \\
\]

Negation may only be applied to Boolean predicates: \text{true, false, f = n}, closed under +, \cdot, and !
Negation may only be applied to Boolean predicates: \textbf{true}, \textbf{false}, \textbf{f = n}, closed under +, \textbf{\cdot}, and \!.
Negation may only be applied to Boolean predicates: \texttt{true}, \texttt{false}, \texttt{f = n}, closed under +, \cdot, 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

port = 1 • A $\rightarrow$ B

port = 5

port = 2 • A $\rightarrow$ B

port = 6
Virtual Program

virtual "big switch"
Virtual Program

Virtual "big switch"

\[
\begin{align*}
\text{port} &= 1 \cdot \text{port} = 5 \\
+ \quad \text{port} &= 2 \cdot \text{port} = 6
\end{align*}
\]
Virtual Program

virtual "big switch"

port=1 • port:=5
+ port=2 • port:=6
Virtual Program

virtual "big switch"

! (tcpSrcPort = 22)

• port=1
• port:=5

+ port=2
• port:=6
Encoding Networks

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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

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.
NetKAT Semantics
NetKAT Semantics

[p]

Denotational
NetKAT Semantics

Denotational

Axiomatic

\[\vdash p \equiv q\]
NetKAT Semantics

Denotational

\[ [p] \]

Axiomatic

\[ \vdash p \equiv q \]

Automata Theoretic
NetKAT Semantics

\[ [p] \]

Denotational

\[ \vdash p \equiv q \]

Axiomatic

Automata Theoretic
NetKAT Semantics

Denotational

\[ \vdash p \equiv q \]

Soundness+Completeness

[POPL ’14]

Axiomatic

Automata Theoretic
NetKAT Semantics

Denotational

\[ [p] \]

\[ \vdash p \equiv q \]

Soundness+Completeness
[POPL ’14]

Kleene’s Theorem
[POPL ’15]

Axiomatic

Automata Theoretic
NetKAT Semantics

Denotational

⊢

\[ p \equiv q \]

Soundness+Completeness
[POPL ‘14]

Kleene’s Theorem
[POPL ‘15]

Proof-Carrying Code
[Ongoing Work]

Axiodmatic

Automata Theoretic
Denotational Semantics

\[
\text{pol ::= } \begin{cases} 
\text{false} \\
\text{true} \\
\text{field = val} \\
\text{field := val} \\
\text{pol}_1 + \text{pol}_2 \\
\text{pol}_1 \cdot \text{pol}_2 \\
\text{!pol} \\
\text{pol}^{*} \\
\text{dup} 
\end{cases}
\]
Denotational Semantics

\[
\text{pol ::= false} \mid \text{true} \mid \text{field = val} \mid \text{field := val} \mid \text{pol}_1 + \text{pol}_2 \mid \text{pol}_1 \cdot \text{pol}_2 \mid !\text{pol} \mid \text{pol}^* \mid \text{dup}
\]

Local NetKAT: input-output behavior of switches

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

pol ::= 
| false
| true
| field = val
| field := val
| pol₁ + pol₂
| pol₁ • pol₂
| !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⟧ ∈ History → History Set
⟦true⟧ h = { h }
⟦false⟧ h = {}
⟦f = n⟧ pk :: h = \{ { pk :: h } if pk.f = n \\
\{ {} \} otherwise
\}
⟦! pol⟧ h = { h } \backslash [pol]
⟦f := n⟧ pk :: h = \{ pk[f:=n] :: h \}
⟦pol_1 + pol_2⟧ h = [pol_1] h \cup [pol_2] h
⟦pol_1 \cdot pol_2⟧ h = ([pol_1] \cdot [pol_2]) h
⟦pol^*⟧ h = ( \bigcup_i [pol]^i h)
⟦dup⟧ pk :: h = \{ pk :: pk :: h \}

f,g ∈ History → History Set
(f \cdot g) h = \bigcup \{ g h' | h' ∈ 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 ≡ q
Kleene Algebra Axioms

\[ \begin{align*}
  p + (q + r) & \equiv (p + q) + r \\
  p + q & \equiv q + p \\
  p + \text{false} & \equiv p \\
  p + p & \equiv p \\
  p \cdot (q \cdot r) & \equiv (p \cdot q) \cdot r \\
  p \cdot (q + r) & \equiv p \cdot q + p \cdot r \\
  (p + q) \cdot r & \equiv p \cdot r + q \cdot r \\
  \text{true} \cdot p & \equiv p \\
  p & \equiv p \cdot \text{true} \\
  \text{false} \cdot p & \equiv \text{false} \\
  p \cdot \text{false} & \equiv \text{false} \\
  \text{true} + p \cdot p^* & \equiv p^* \\
  \text{true} + p^* \cdot p & \equiv p^* \\
  p + q \cdot r + r & \equiv r \Rightarrow p^* \cdot q + r \equiv r \\
  p + q \cdot r + q & \equiv q \Rightarrow p \cdot r^* + q \equiv q \\
\end{align*} \]

Boolean Algebra Axioms

\[ \begin{align*}
  a + (b \cdot c) & \equiv (a + b) \cdot (a + c) \\
  a + \text{true} & \equiv \text{true} \\
  a + ! a & \equiv \text{true} \\
  a \cdot b & \equiv b \cdot a \\
  a \cdot ! a & \equiv \text{false} \\
  a \cdot a & \equiv a \\
\end{align*} \]

Packet Axioms

\[ \begin{align*}
  f & := n \cdot f' := n' \equiv f' := n' \cdot f := n \quad \text{if } f \neq f' \\
  f & := n \cdot f = n' \equiv f' := n' \cdot f := n \quad \text{if } f \neq f' \\
  f & := n \cdot f = n \equiv f := n \\
  f & = n \cdot f := n \equiv f = n \\
  f & := n \cdot f := n' \equiv f := n' \\
  f & = n \cdot f = n' \equiv \text{false} \quad \text{if } n \neq n' \\
  \text{dup} \cdot f & = n \equiv f = n \cdot \text{dup} \\
  \Sigma_i f = n_i & \equiv \text{true} \\
\end{align*} \]
### Boolean Algebra Axioms

\[ a + (b \cdot c) \equiv (a + b) \cdot (a + c) \]
\[ a + \text{true} \equiv \text{true} \]
\[ a + \lnot a \equiv \text{true} \]
\[ a \cdot \lnot a \equiv \text{false} \]
\[ a \cdot a \equiv a \]

### Kleene Algebra Axioms

\[ p + (q + r) \equiv (p + q) + r \]
\[ p + q \equiv q + p \]
\[ p + \text{false} \equiv p \]
\[ p + p \equiv p \]
\[ p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r \]
\[ (p + q) \cdot r \equiv p \cdot r + q \cdot r \]
\[ \text{true} \cdot p \equiv p \]
\[ \lnot p \equiv \text{false} \]
\[ \lnot (p \cdot q) \equiv \lnot p \cdot \lnot q \]
\[ p + q + r \equiv r + q + p \]
\[ p \cdot q + r \equiv q + p \cdot r \]

### NetKAT Axioms

**Soundness:** If \( \vdash p \equiv q \), then \( \llbracket p \rrbracket = \llbracket q \rrbracket \)

**Completeness:** If \( \llbracket p \rrbracket = \llbracket q \rrbracket \), then \( \vdash p \equiv q \)

- \( f := n \cdot f' = n' \equiv f' = n' \cdot f := n \)
- \( f := n \cdot f = n \equiv f := n \)
- \( f := n \cdot f = n' \equiv f := n' \)

\( f := n \cdot f = n' \equiv \text{false} \) if \( n \not= n' \)

\( \text{dup} \cdot f = n \equiv f = n \cdot \text{dup} \)

\( \Sigma_i f = n_i \equiv \text{true} \)
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,
- $\varepsilon : S \to \text{Packet} \to \text{Packet Set}$ is the “observation” function
- $\delta : S \to \text{Packet} \to (\text{State} \times \text{Packet}) \to \text{Packet Set}$ 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
- $\delta \in S \rightarrow \text{Packet} \rightarrow \text{(State * Packet) Set}$ is the “continuation” function

Inputs: $\text{pkt}_{\text{in}} \cdot \text{pkt}_1 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \text{pkt}_n \cdot \text{dup} \cdot \text{pkt}_{\text{out}}$
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
• $\delta \in S \rightarrow \text{Packet} \rightarrow (\text{State} \times \text{Packet}) \text{ Set}$ is the “continuation” function

**Inputs:** $\text{pkt}_{\text{in}} \cdot \text{pkt}_1 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \text{pkt}_n \cdot \text{dup} \cdot \text{pkt}_{\text{out}}$

A NetKAT automaton accepts an input in state $s$ if:
• $\text{accept } s (\text{pkt}_{\text{in}} \cdot \text{pkt}_{\text{out}}) \iff \text{pkt}_{\text{out}} \in \varepsilon s \text{ pkt}_{\text{in}}$
• $\text{accept } s (\text{pkt}_{\text{in}} \cdot \text{pkt}_1 \cdot \text{dup} \cdot \text{w}) \iff \exists s'. (\text{pkt}_1, s') \in \delta s \text{ pkt}_{\text{in}}$ and accept $s'(\text{pkt}_1 \cdot \text{w})$
NetKAT Derivatives

E(pol) ∈ Pol

\[\begin{align*}
E(\text{false}) &= \text{false} \\
E(\text{true}) &= \text{true} \\
E(f = n) &= f = n \\
E(f := n) &= f := n \\
E(!pol) &= !pol \\
E(\text{dup}!) &= \text{false} \\
E(pol_1 + pol_2) &= E(pol_1) + E(pol_2) \\
E(pol_1 \cdot pol_2) &= E(pol_1) \cdot E(pol_2) \\
E(pol^*) &= E(pol)^*
\end{align*}\]

D(pol) ∈ (Pol * L * Pol) Set

\[\begin{align*}
D(\text{false}) &= \{\} \\
D(\text{true}) &= \{\} \\
D(f = n) &= \{\} \\
D(f := n) &= \{\} \\
D(!pol) &= \{\} \\
D(\text{dup}!) &= \{(\text{true}, l, \text{true})\} \\
D(pol_1 + pol_2) &= D(pol_1) + D(pol_2) \\
D(pol_1 \cdot pol_2) &= D(pol_1) \cdot pol_2 + E(pol_1) \cdot D(pol_2) \\
D(pol^*) &= E(pol)^* \cdot D(pol) \cdot pol^*
\end{align*}\]
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,
- $\varepsilon \mid \text{pkt} = \{\text{pkt'} | \langle \text{pkt'} \rangle \in \llbracket E(k_l) \rrbracket \langle \text{pkt} \rangle\}$
- $\delta \mid \text{pkt} = \{(\text{pkt'}, l') | (d, l', k) \in \llbracket D(k_l) \rrbracket \land \langle \text{pkt'} \rangle \in \llbracket d \rrbracket \langle \text{pkt} \rangle\}$

**Notation:** $k_l$ denotes the unique continuation of $\text{dup}^l$

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]

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

\[ \text{in} \cdot (\text{pol} \cdot \text{topo})^* \cdot \text{pol} \cdot \text{out} \not\equiv \text{false} \]
Loop Freedom [POPL ’15]

Can use automata to check if a network is loop free

Intuition: ∀α. in • (p • t)^* • α • (p • t)^+ • α ≡ false

- ∀ pkt, pkt’. pkt’ ∈〚E(Φ(in • (p • t)^*))〛 pkt
- Check whether pkt’ ∈〚E(Φ(p • t)^+))〛 pkt’

Notation: Φ(p) denotes replacing dup with true
Compilation [ICFP '15]

NetKAT Compiler Pipeline

1 2 3
Compilation [ICFP ’15]

NetKAT Compiler Pipeline

3 → 2 → Local Compiler

<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>

~ 100x faster than competitors
Compilation [ICFP ’15]

NetKAT Compiler Pipeline

3 global policy  Global Compiler  local policy  Local Compiler

network-wide behavior  ~ 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>
Compilation [ICFP ‘15]

NetKAT Compiler Pipeline

- **Virtual Compiler**
  - virtual policy
  - abstract topologies

- **Global Compiler**
  - global policy
  - network-wide behavior

- **Local Compiler**
  - local policy
  - ~ 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>
Compilation [ICFP '15]

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>
Compilation [ICFP '15]

NetKAT Compiler Pipeline

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

Based on:

Patterns and Actions:
- dstpt=2 drop
- srcpt=7 fwd 1
- * fwd 2
Compilation [ICFP ’15]

NetKAT Compiler Pipeline

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

Patterns and Actions:
- Pattern: dstpt=2
  - Actions: drop
- Pattern: srcpt=7
  - Actions: fwd 1
- Pattern: *
  - Actions: fwd 2

Based on abstract topologies.
Global Compilation
Global Compilation

Global Program

Adding Extra State = Translation to Automaton

NetKAT NFA
Global Compilation

Global Program

Adding Extra State
= Translation to Automaton

NetKAT NFA

Avoiding Duplication
= Determinization

NetKAT DFA
Global Compilation

Adding Extra State
= Translation to Automaton

NetKAT NFA

Automaton Minimization
= Tag Elimination

Avoiding Duplication
= Determinization

NetKAT DFA
Global Compilation

1. Global Program
2. Adding Extra State
   - = Translation to Automaton
3. NetKAT NFA
4. Automaton Minimization
   - = Tag Elimination
5. NetKAT DFA
6. Avoiding Duplication
   - = Determinization
7. Local Program
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
Can implement path queries at the network edge!

- Combine policy $p$ and query $q$ via a simple translation
- Use $\Phi$ to replace all occurrences of $\text{dup}$ with $\text{true}$
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
Can implement path queries at the network edge!

- Combine policy $p$ and query $q$ via a simple translation
- Use $\Phi$ 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


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/