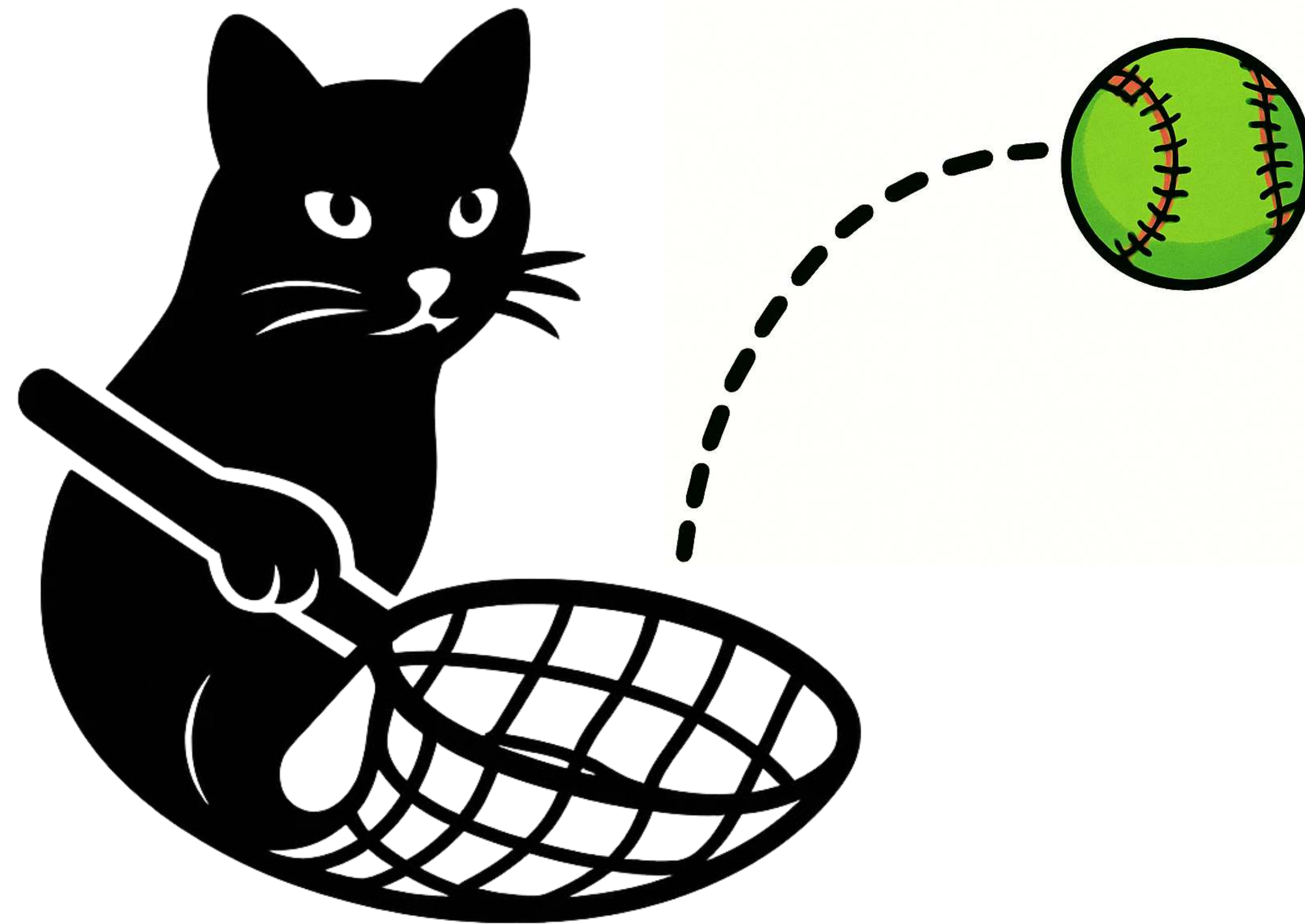


KATch: Algebraic Network Verification





Paradigm shift with Software-Defined Networking





Paradigm shift with Software-Defined Networking

Why do we trust networks?

Privacy
Security

....

Modelling and Verification

Modelling and Verification

Modelling and Verification

+

Principled approach
Design adjustments

Applicable pre- and post-
production

+

-

Too expensive
Too slow

Scalability
Compositionality

-

Modelling and Verification

+

Principled approach
Design adjustments

Applicable pre- and post-
production

+

-

Too expensive
Too slow

Scalability
Compositionality

-

Language Design

Learning

+

Principled approach
Design adjustments

Applicable pre- and post-
production

+

-

Too expensive
Too slow

Scalability
Compositionality

-

Language
Design

Automata

Learning

+

Principled approach
Design adjustments

Applicable pre- and post-
production

+

-

Too expensive
Too slow

Scalability
Compositionality

-

Language
Design

Automata

Learning

+

Principled approach
Design adjustments

Applicable pre- and post-
production

+

-

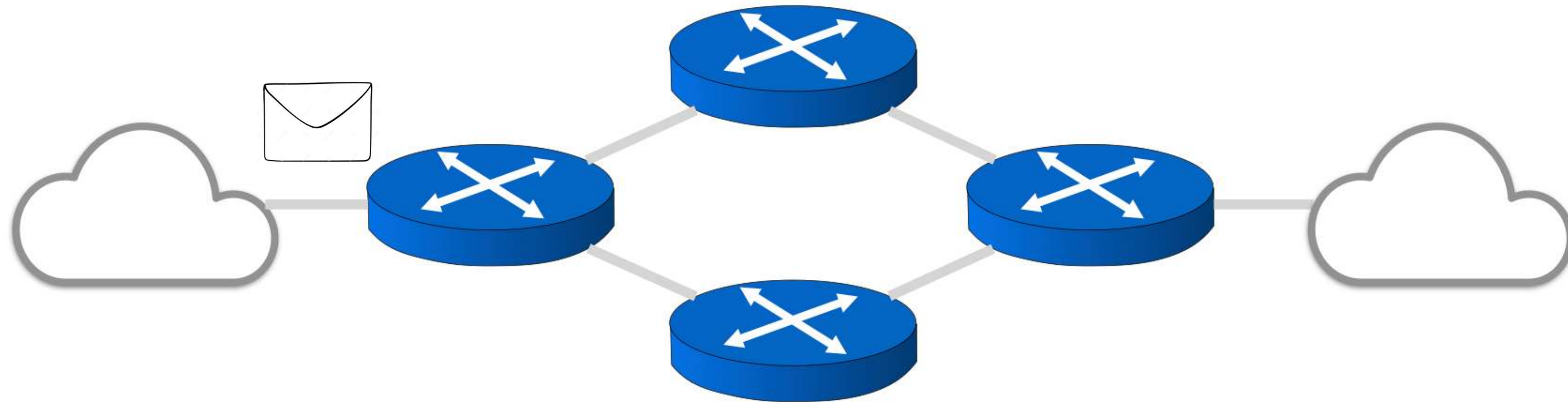
Too expensive
Too slow

Improves

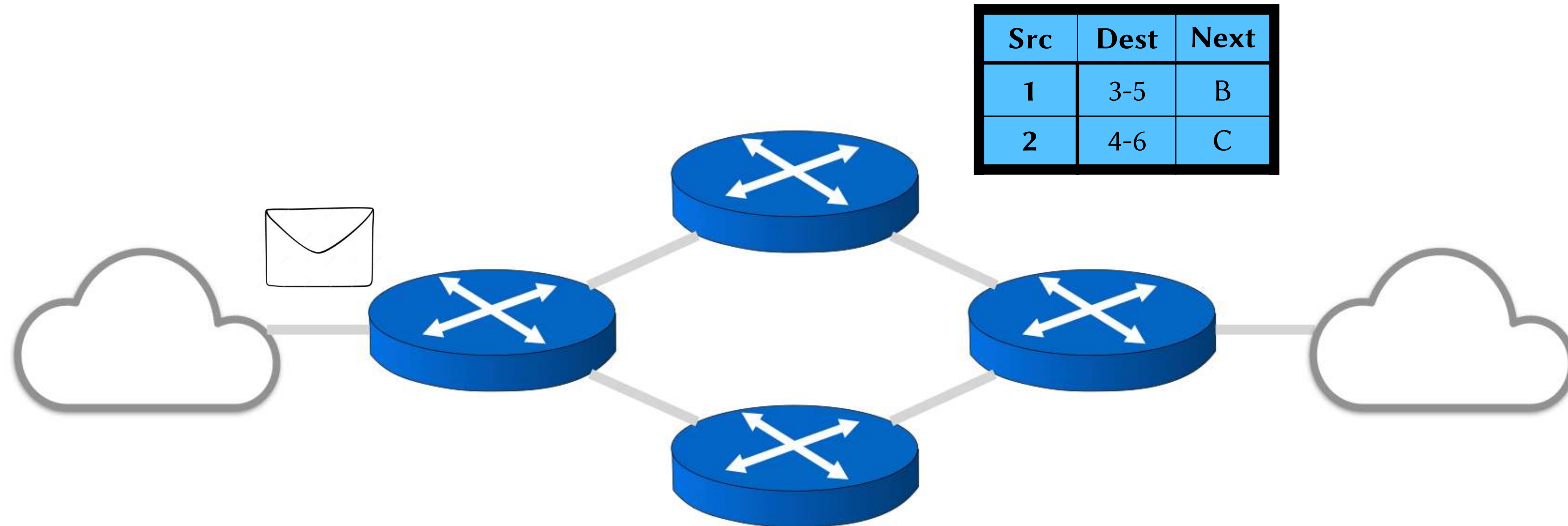
Scalability
Compositionality

-

Verification in Network Routing Policies



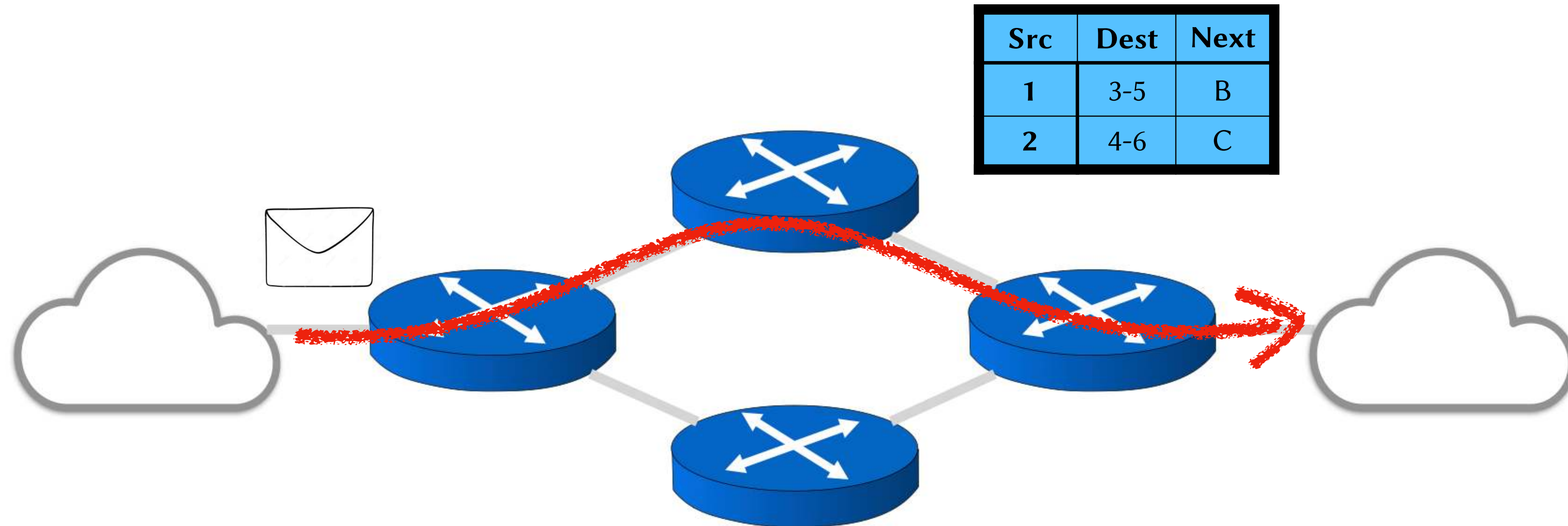
Verification in Network Routing Policies



Each table contains rules that:

- **Match** on packet headers
- Execute **Actions** that transform, forward, or drop packets

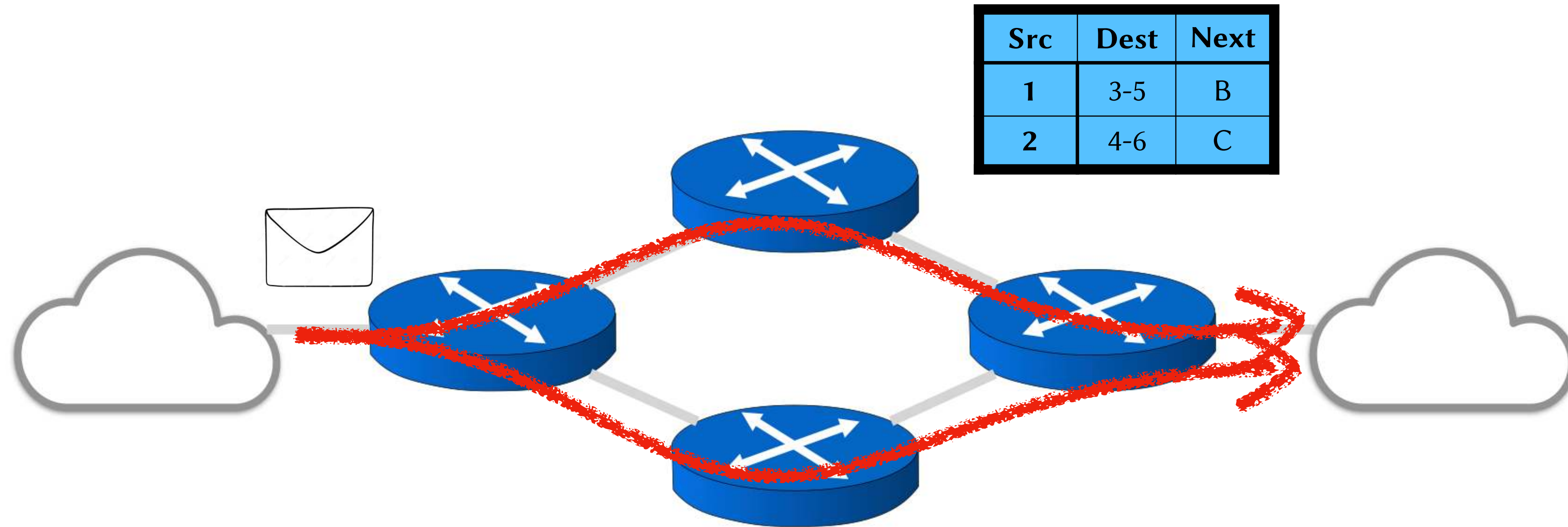
Verification in Network Routing Policies



Each table contains rules that:

- **Match** on packet headers
- Execute **Actions** that transform, forward, or drop packets

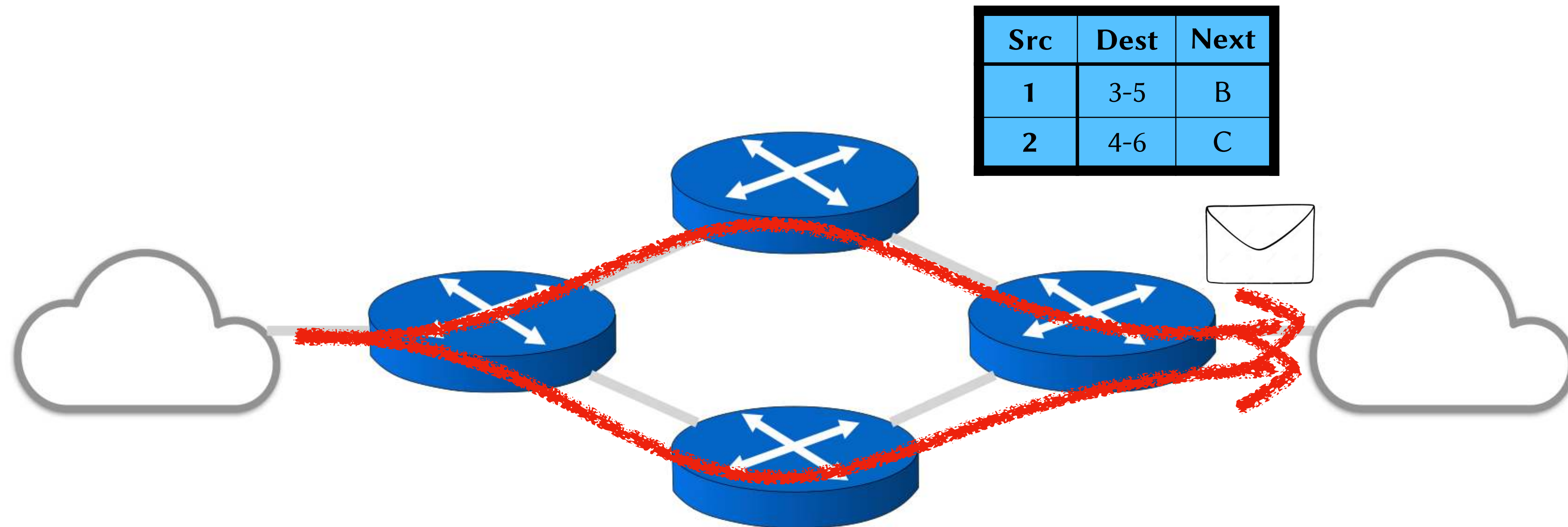
Verification in Network Routing Policies



Each table contains rules that:

- **Match** on packet headers
- Execute **Actions** that transform, forward, or drop packets

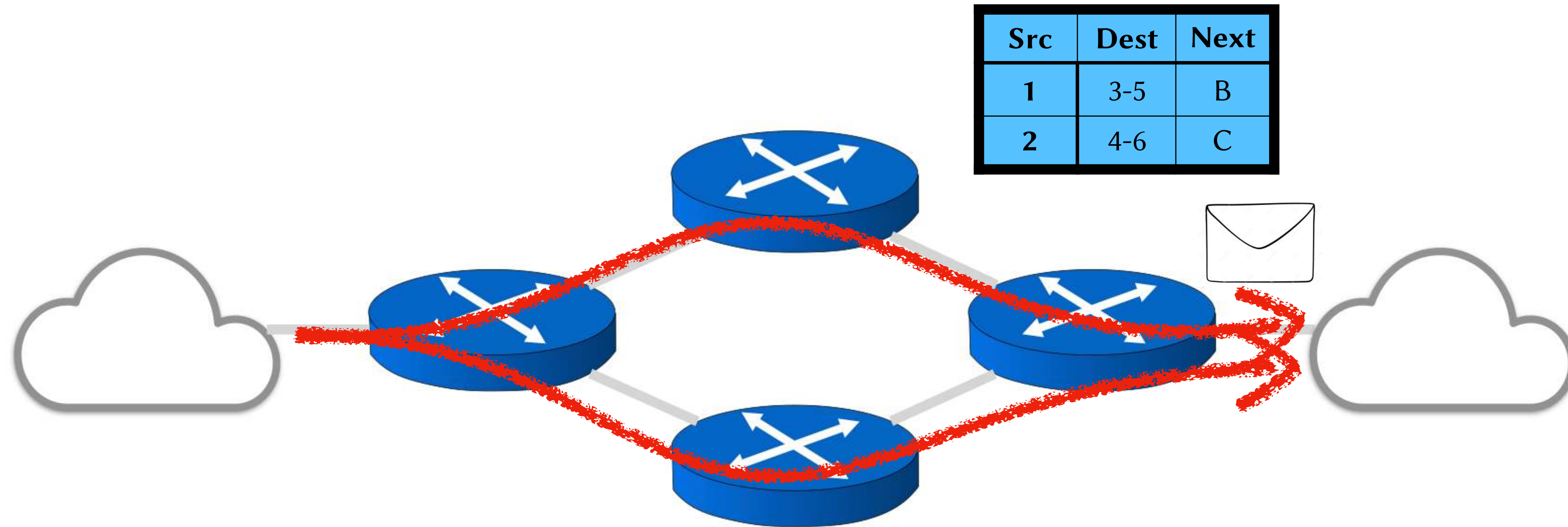
Verification in Network Routing Policies



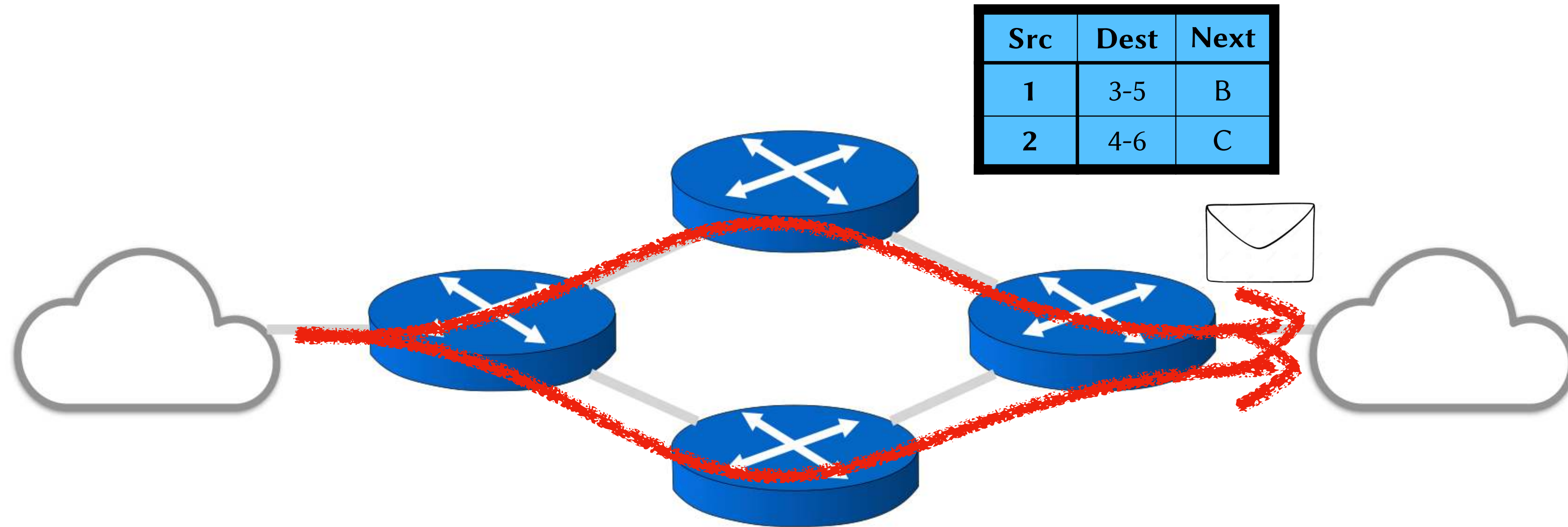
Each table contains rules that:

- **Match** on packet headers
- Execute **Actions** that transform, forward, or drop packets

Verification in Network Routing Policies

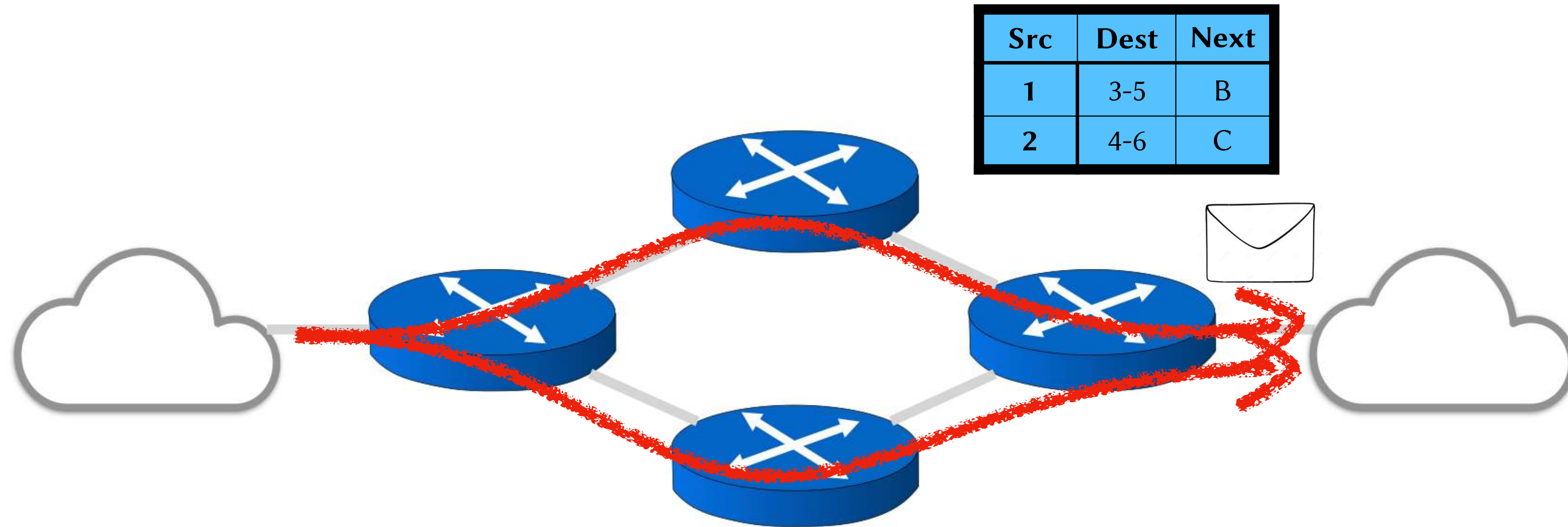


Verification in Network Routing Policies



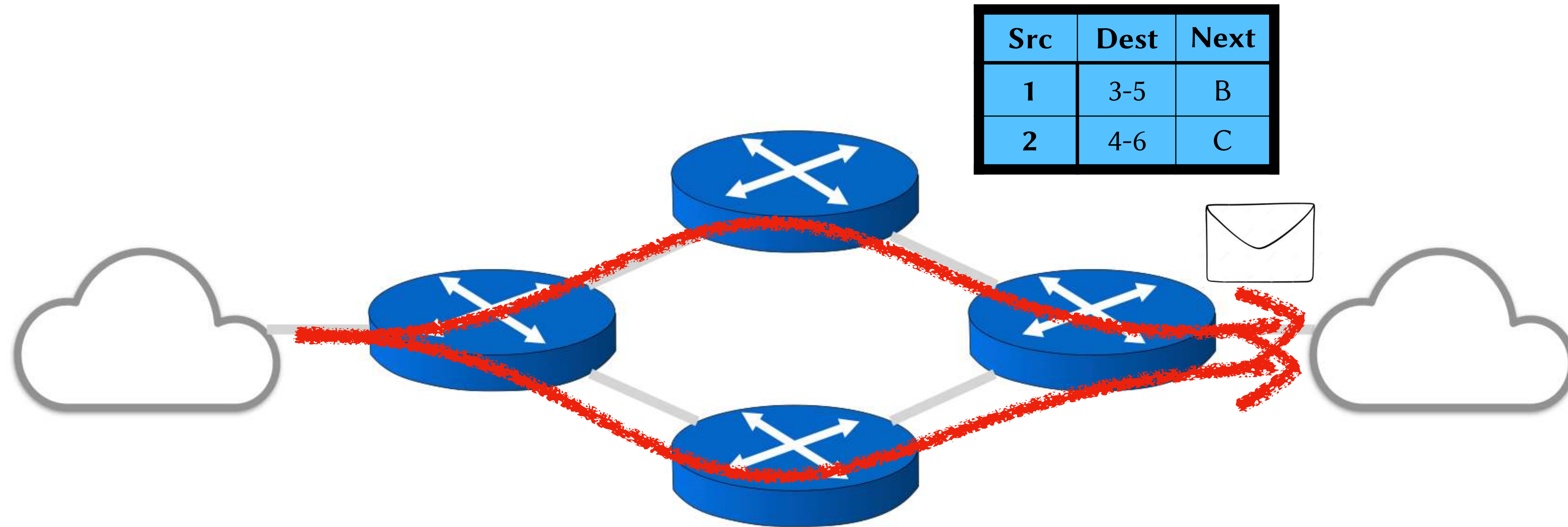
Can a packet travel from A to B?

Verification in Network Routing Policies



Can a packet travel from A to B?
Are all hosts reachable from every other hosts?

Verification in Network Routing Policies

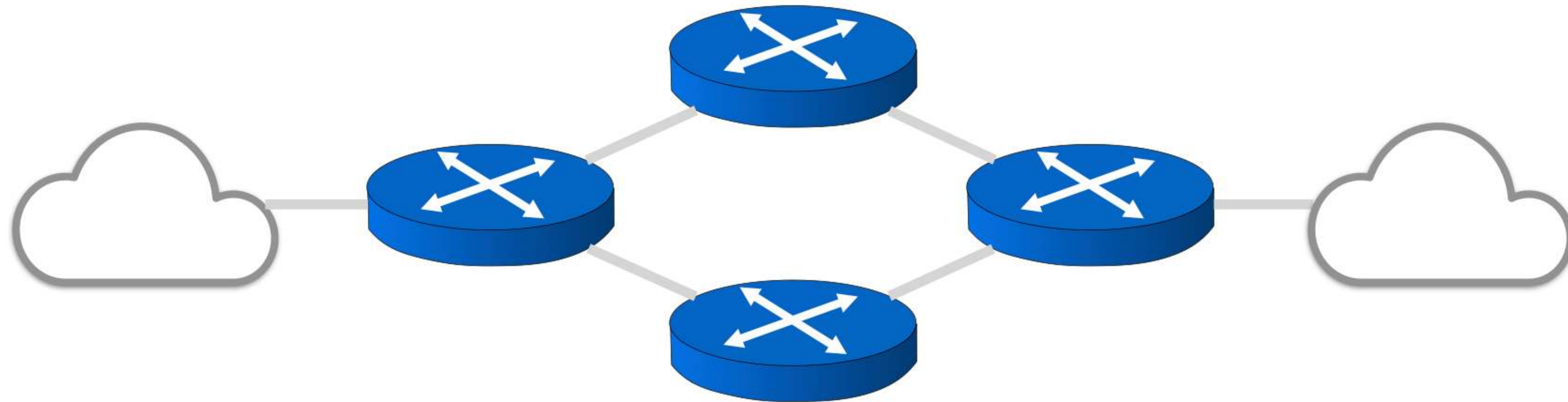


Can a packet travel from A to B?

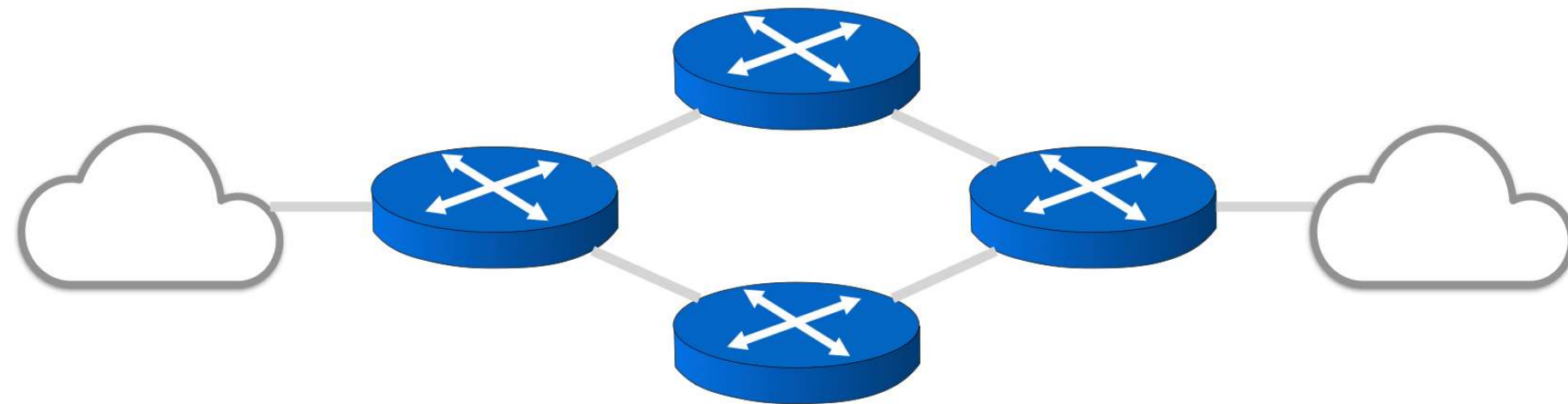
Are all hosts reachable from every other hosts?

Is subnet X isolated from subnet Y?

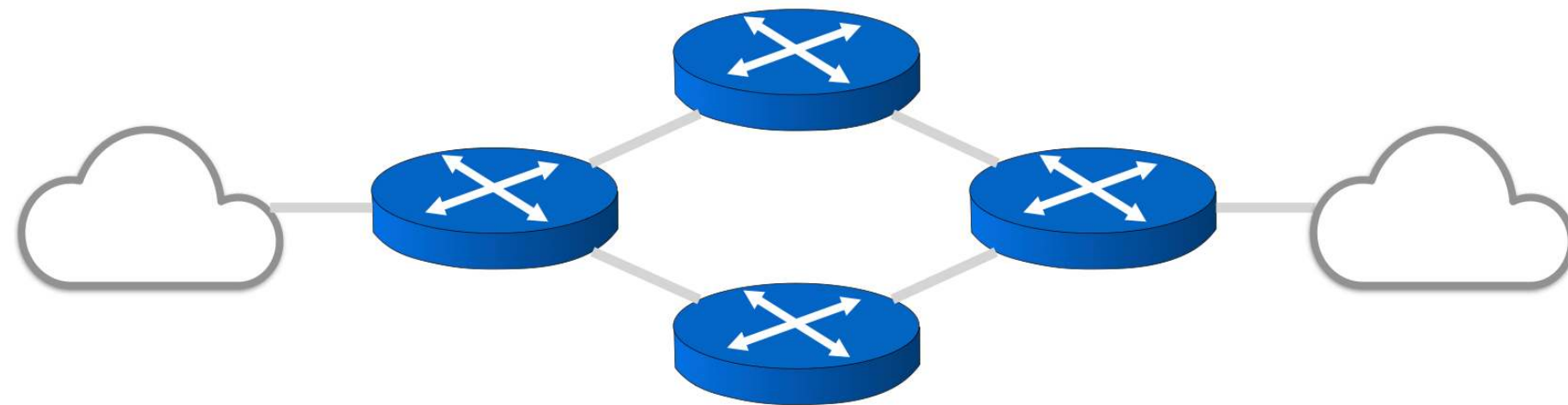
Verification in Network Routing Policies



Verification in Network Routing Policies

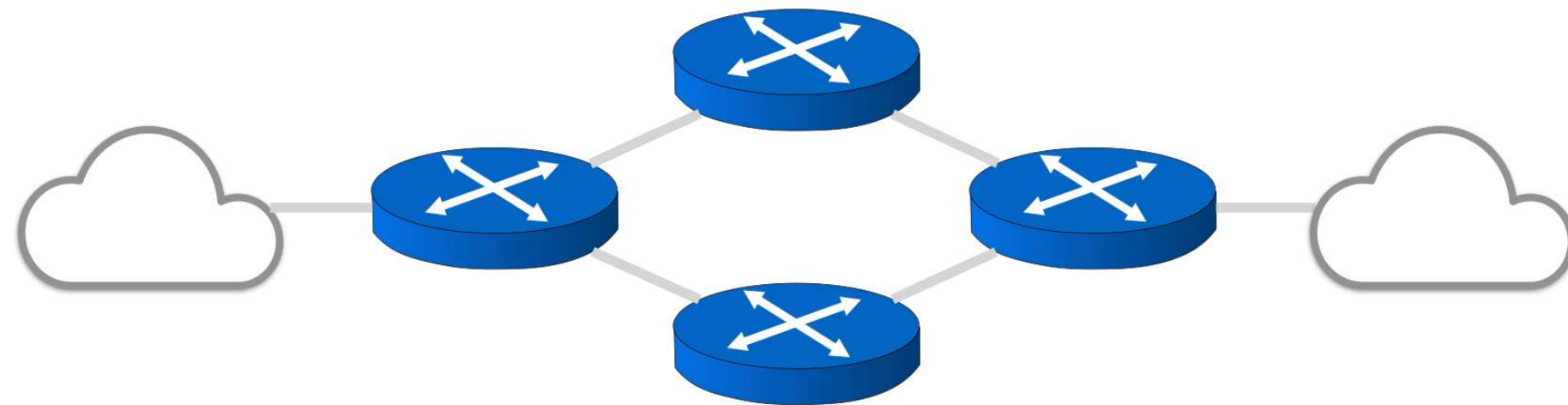


Verification in Network Routing Policies



Topology + configurations

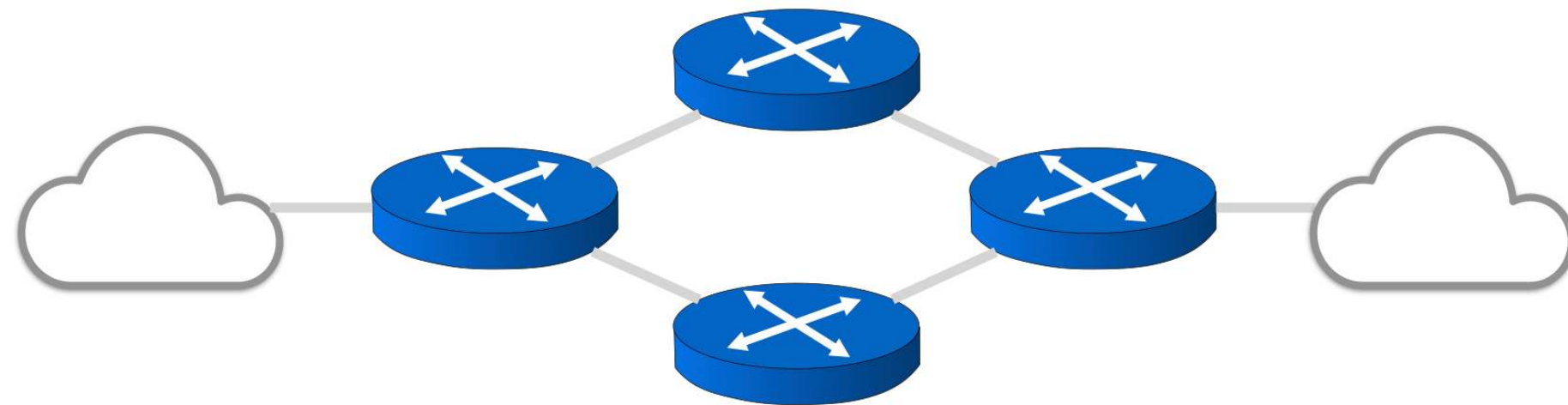
Verification in Network Routing Policies



Topology + configurations

Spec: *The network should be free of forwarding loops, and every packet should be delivered to its destination*

Verification in Network Routing Policies



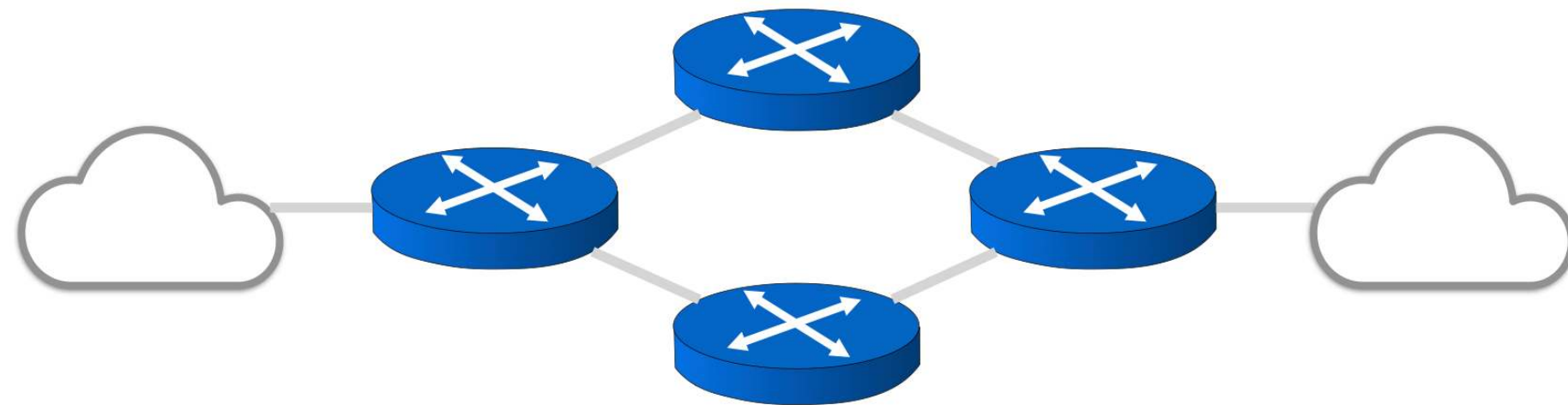
Topology + configurations

Spec: *The network should be free of forwarding loops, and every packet should be delivered to its destination*

Solver

φ

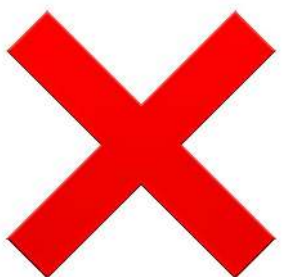
Verification in Network Routing Policies



Topology + configurations

Spec: *The network should be free of forwarding loops, and every packet should be delivered to its destination*

Solver



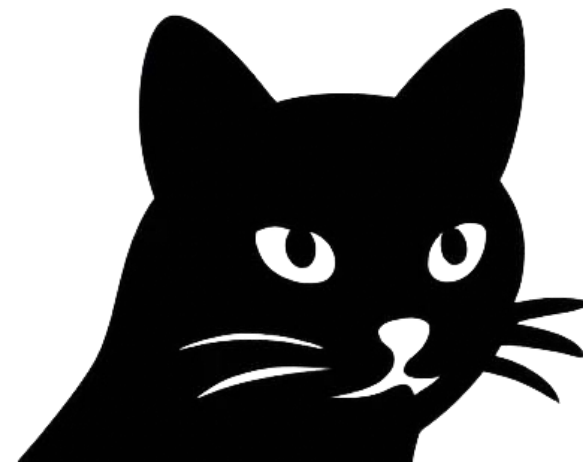
φ

Challenging the status quo

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems

Challenging the status quo

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems



Design **high-level languages** that model essential network features
Develop **semantics** that enables reasoning precisely about behaviour
Build **tools** to synthesise low-level implementations automatically

What are essential network features?

Can a packet travel from A to B?

Are all hosts reachable from every other hosts?

Is subnet X isolated from subnet Y?

What are essential network features?

Nodes

Can a packet travel from A to B?

Are all hosts reachable from every other hosts?

Is subnet X isolated from subnet Y?

What are essential network features?

Nodes

Forwarding Policies

Can a packet travel from A to B?

Are all hosts reachable from every other hosts?

Is subnet X isolated from subnet Y?

What are essential network features?

Nodes — **Forwarding Policies**

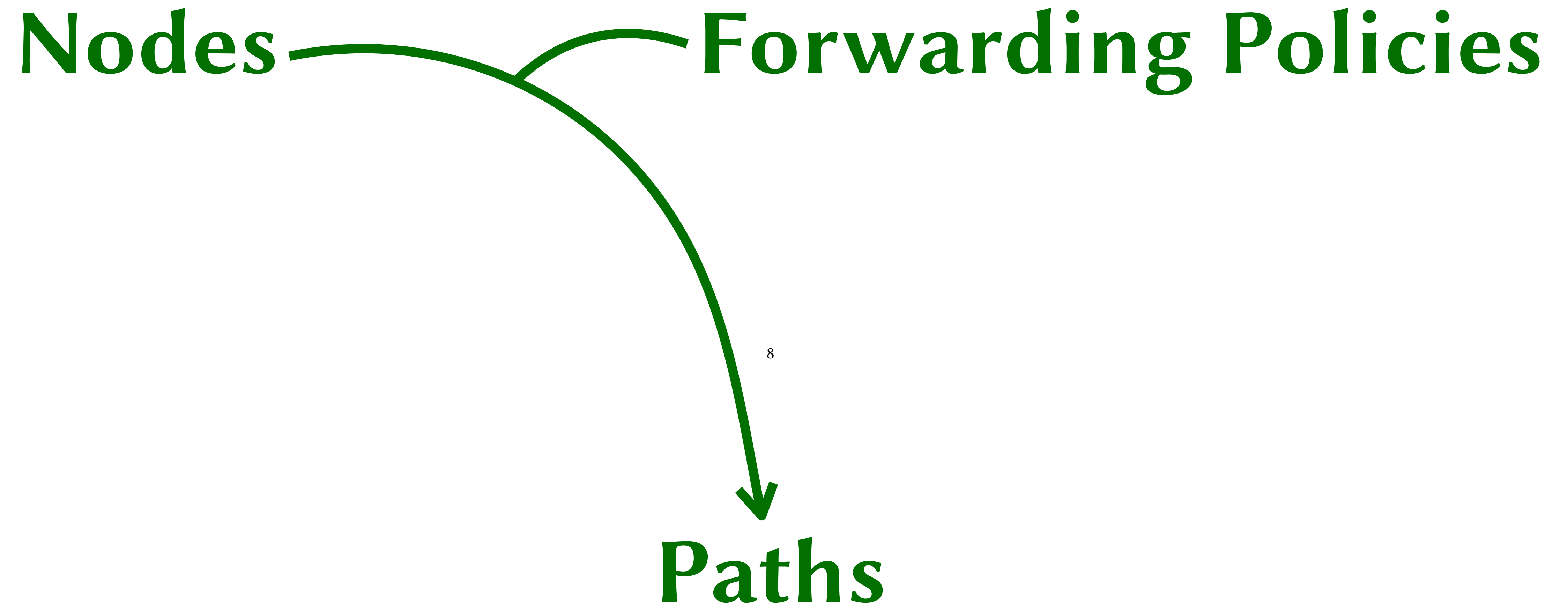
Can a packet travel from A to B?

Are all hosts reachable from every other hosts?

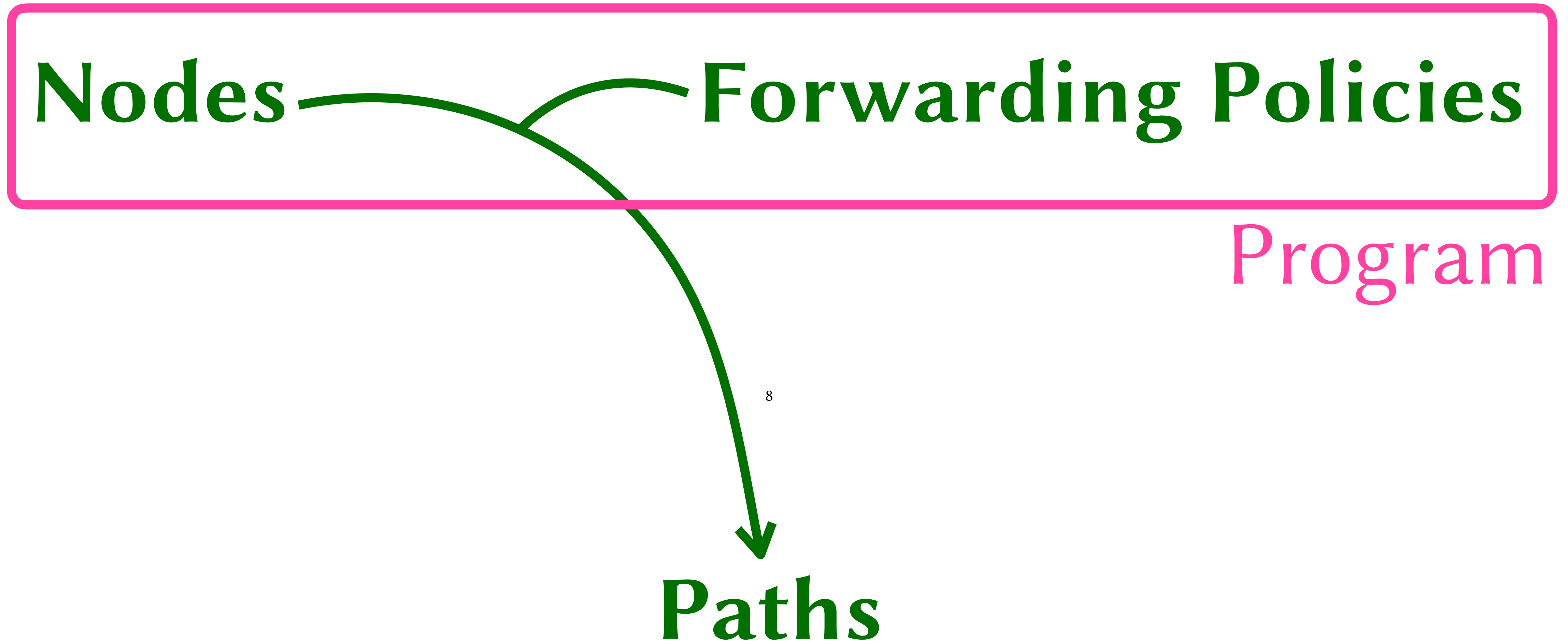
Is subnet X isolated from subnet Y?

Paths

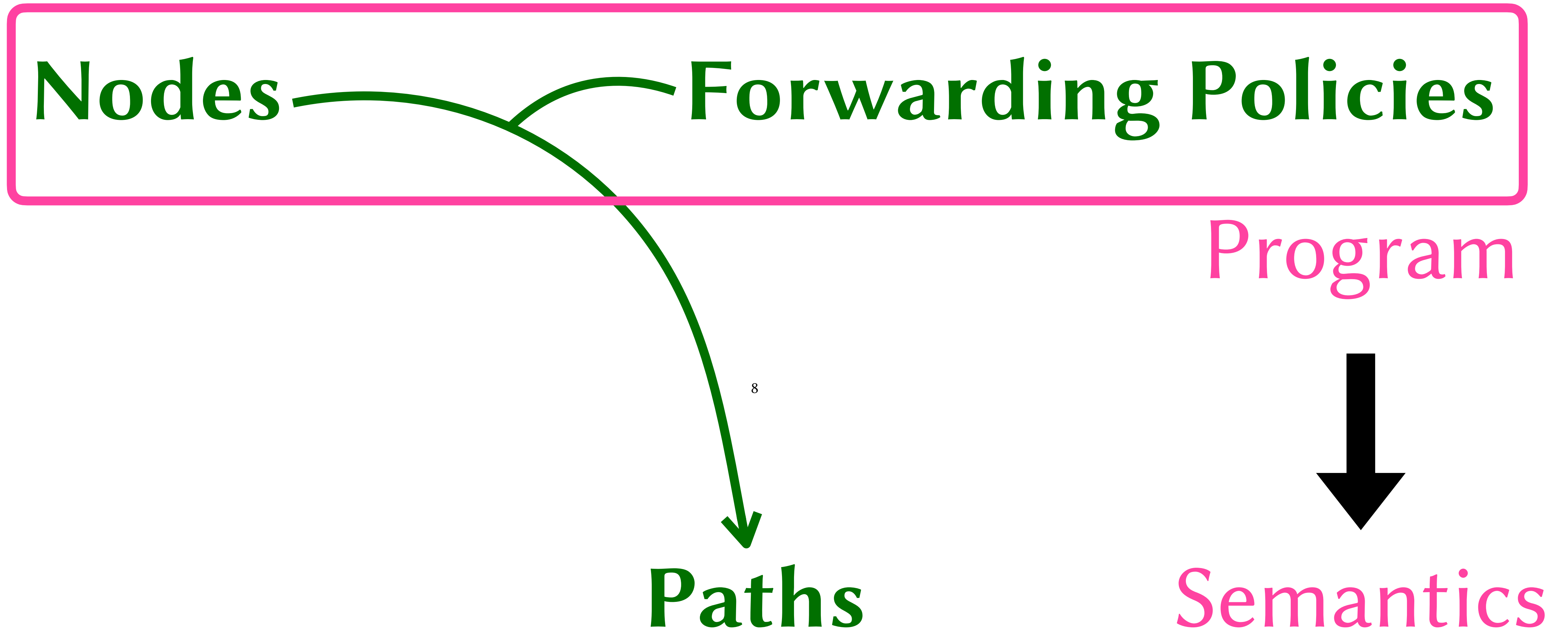
What are essential network features?



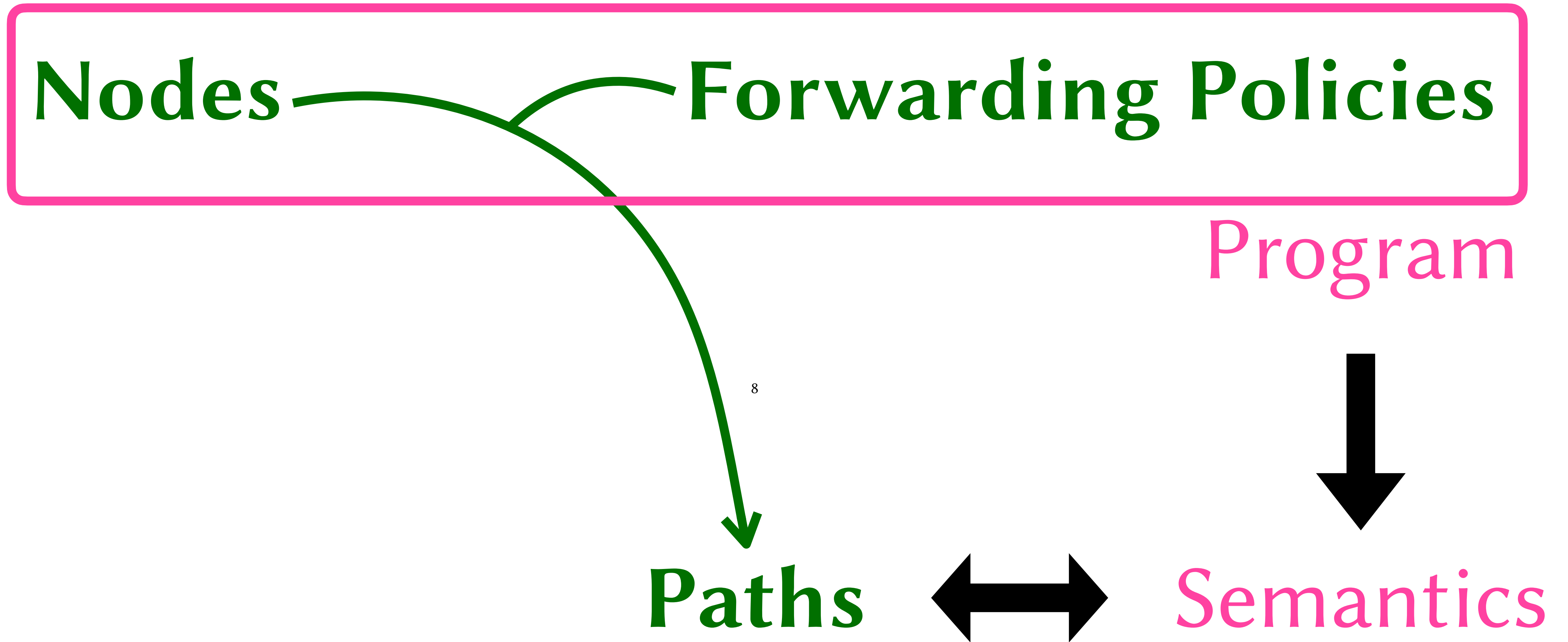
What are essential network features?



What are essential network features?



What are essential network features?

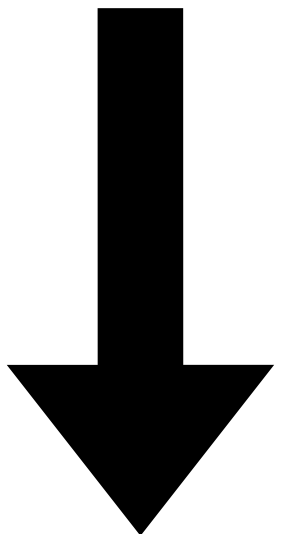


What are essential network features?

Nodes

Forwarding Policies

Program

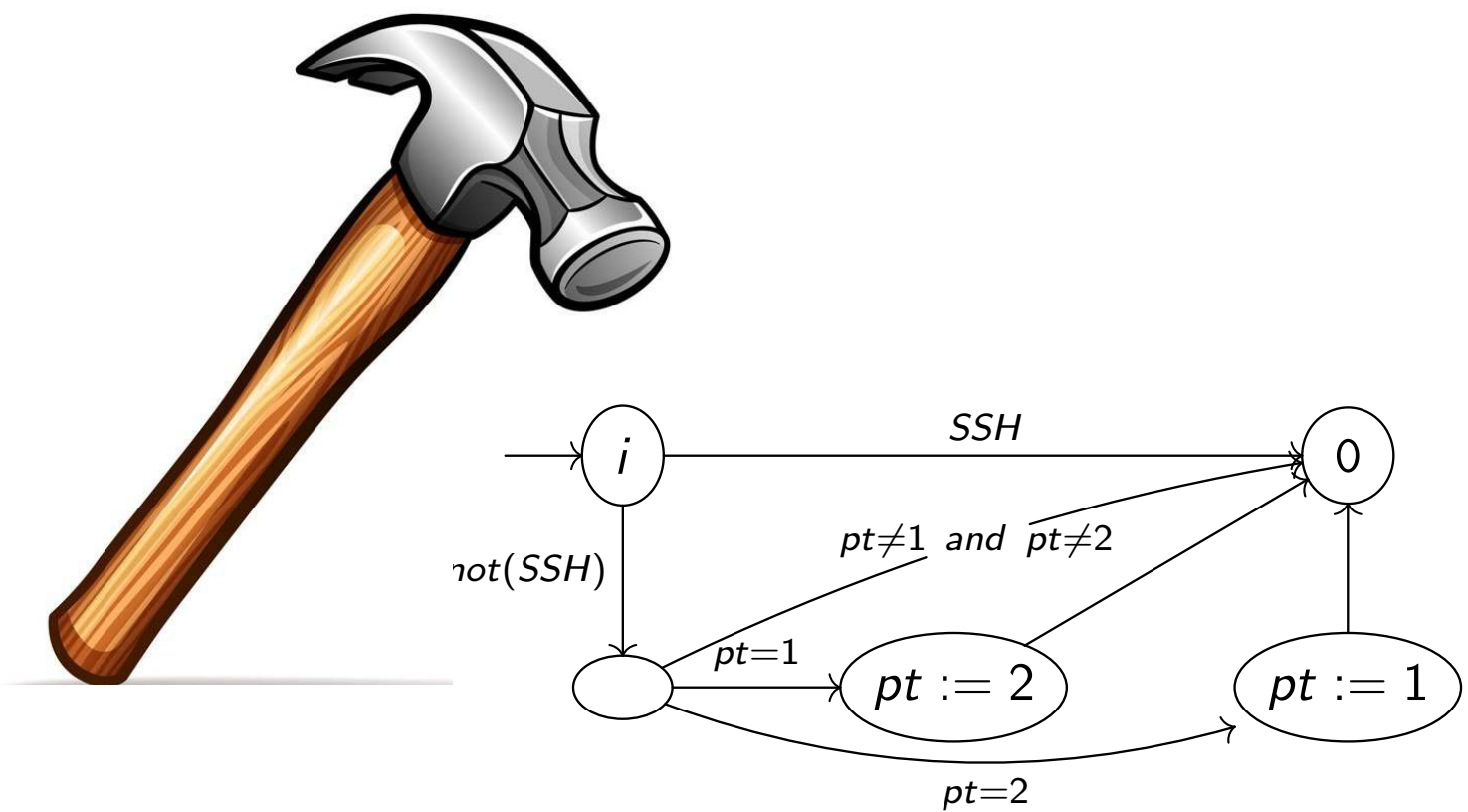


Semantics

Paths



Automata



NetKAT: a network language for reachability

Automata  Paths

NetKAT: a network language for reachability



NetKAT: a network language for reachability

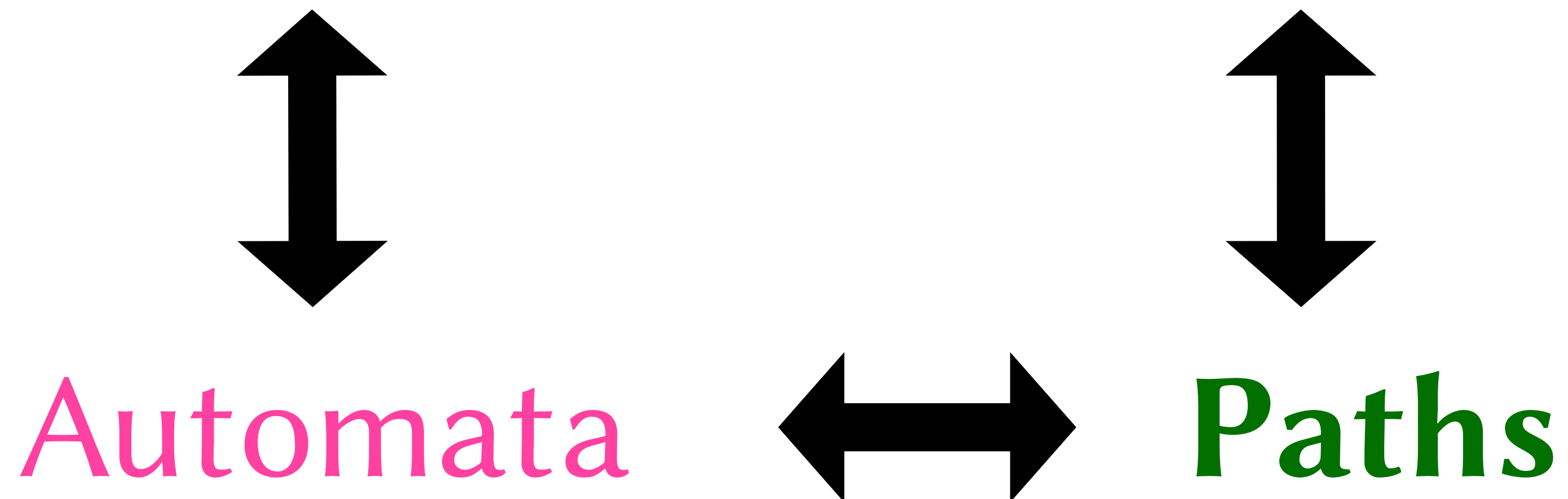
Nodes + Tests + Forwarding Policies



NetKAT: a network language for reachability

Nodes + Tests + Forwarding Policies

Generalised Regular Expressions



Design choices

Design choices

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Design choices

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Transformations

To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Design choices

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Transformations

To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Compositionality

To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

Design choices

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Transformations

To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Compositionality

To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

Iteration

To model the potentially iterated processing performed via the topology, need general loops

Design choices

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Transformations

To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Compositionality

To model richer pipelines, need programs both conditionally and in sequence

Iteration

To model the potentially iterated topology, need general loops

Free from using regular expressions as core of the language

Design choices

Packet Classification

Challenge: how to add these without breaking the toolkit of regular expressions (decidability, scalability)?

Transformations

To model the potentially iterated topology, need general loops

Compositionality

Free from using regular expressions as core of the language

Iteration

To model richer pipelines, need programs both conditionally and in sequence

Design choices

Packet Classification

Transformations

Challenge: how to add these without breaking the toolkit of regular expressions (decidability, scalability)?

Bonus points: beyond reachability?

Compositionality

Iteration

Free from using regular expressions as core of the language



[illegible]

POPL'14

POPL'17

PLDI'19

2014

2015

2017

2019

2022

2024

2025

2026

[illegible]

POPL'15

[illegible]

POPL'14

2014 2015

representation that results in a significant reduction in the size of the state space. We discuss the details of our implementation and optimizations that exploit NOKeAT's equiguarded actions and confluence structures to yield significantly improved performance. The present results from experiments demonstrating that our tool is competitive with state-of-the-art tools on several benchmarks including all pairs connectivity, loop-freeness, and translation validation.

Categories and Subject Descriptors: F.4.3 [Formal Languages]; Classes defined by grammars or automata

Keywords: Confluents; Kleene algebras; with tests; Brzozowski derivatives; automatic network verification; NOKeAT

1. Introduction

Nets have received widespread attention in recent years as a target for domain-specific language design. The emergence of *replacer-defined networking* (RDN) as a popular paradigm for software programming has led to the appearance of a number of SDN programming languages including Promela, Nite, NetSim, PyNet, Magma, and P4ML among others [10–12, 26, 27, 49]. The details of these languages differ, but each tends to provide high-level abstractions to simplify the task of specifying the packet-

POP

A horizontal timeline with a vertical line marking the year 2015. The text "L'14" is positioned above the line, and "14" and "2015" are positioned below the line on either side of the vertical marker.

POPL'15

A horizontal timeline with a red line. Two vertical red tick marks are present. The first tick mark is labeled 'POPL'17' above and '2017' below. The second tick mark is labeled 'PLDI'17' above and '2017' below.

Response	Percentage
Yes	20
No	9

A horizontal timeline with a thick red line. A vertical red tick mark is positioned in the middle. The year '2022' is written at the left end, and '2024' is written at the right end.

A horizontal timeline with a pink arrow pointing to the right. A vertical pink tick mark is positioned between the years 2025 and 2026, which are written below the line.



NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson
Seawarren College *

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts Amherst *

Jean-Baptiste Jeannin
Carnegie Mellon University *

Dexter Kozen
Cornell University

Cole Schlesinger
Princeton University

David 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, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation has left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKat, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKat, including primitives for filtering, modifying, and transmitting packets, as well as sequential composition operators, and a Kleene star operator that iterates programs. We show that NetKat is an instance of a canonical and well-studied mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking reachability, proving non-interference properties that ensure isolation between programs, and establishing the correctness of compilation algorithms.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

Keywords Software-defined networking, Formal, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKat.

1. Introduction

Traditional network devices have been called “the last bastion of mainframe computing” [9]. Unlike modern computers, which are

* This work performed at Cornell University.

implemented with commodity hardware and programmed using standard interfaces, networks have been built the same way since the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, a revolution has taken place with the recent rise of *software-defined networking* (SDN). In SDN, a general-purpose *controller machine* manages a collection of programmable *switches*. The controller responds to network events such as new connections from hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly. Because the controller has a global view of the network, it is easy to use SDN to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault tolerance.

A major appeal of SDN is that it deflates open standards that any vendor can implement. For example, the OpenFlow API [21] clearly specifies the capabilities and behavior of switch hardware and defines a low-level language for manipulating their configurations. However, programs written directly for SDN platforms such as OpenFlow are also to assembly: easy for hardware to implement, but difficult for humans to write.

Network programming languages. In recent years, several different research groups have proposed domain-specific languages for SDN [2–7, 23–25, 31, 32]. The goal of these network programming languages is to raise the level of abstraction of network programs above hardware-oriented APIs such as OpenFlow, thereby making it easier to build sophisticated and reliable network applications. For example, the languages developed in the *Formal project* [26] support a two-phase programming model: (i) a general-purpose program responds to network events by generating a static forwarding policy; and (ii) the static policy is compiled and passed to a run-time system that configures the switches using OpenFlow messages. This model balances expressiveness—forwarding policies can be expressed by having the general-purpose program generate a sequence of static policies—and simplicity—forwarding policies are written in a simple domain-specific language with a clear semantics, so programs can be analyzed and even verified using automated tools [7, 26].

Still, it has never been clear what features a static policy language should support. The initial version of *Formal* [6] used simple lists of predicate-action rules as policies, where the actions in

POPL’14

POPL’17

PLDI’19

ESOP’22

PLDI’25

2014

2015

2017

2019

2022

2024

2025

2026



A Coalgebraic Decision Procedure for NetKAT

Nate Foster
Cornell University

Dexter Kozen
Cornell University

Mae Milano
Cornell University

Alexandra Silva
Radboud University Nijmegen

Laure Thompson
Cornell University

Abstract

NetKat is a domain-specific language and logic for specifying and verifying network packet-processing functions. It consists of Kleene algebra with tests (KAT) augmented with primitives for routing and modifying packet headers and encoding network topologies. Previous work developed the design of the language and its standard semantics, proved its soundness and completeness of the logic, defined a PSPACE algorithm for deciding equivalence, and presented several practical applications.

This paper develops the coalgebraic theory of NetKat, including a specialized version of the Brzozowski derivative, and presents a new efficient algorithm for deciding the equational theory using bisimulation. The coalgebraic structure admits an efficient sparse representation that results in a significant reduction in the size of the state space. We discuss the details of our implementation and optimizations that explain NetKat’s exceptional actions and coalgebraic structure to yield significantly improved performance. We present results from experiments demonstrating that our tool is competitive with state-of-the-art tools on several benchmarks including all-pairs connectivity, loop-broken, and translation validation.

Categories and Subject Descriptors F.4.3 [Formal Languages]: Classes defined by grammars or automata

Keywords Coalgebra, Kleene algebra with tests, Brzozowski derivatives, automata, network verification, NetKat.

1. Introduction

Networks have received widespread attention in recent years as a target for domain-specific language design. The emergence of *software-defined networking* (SDN) as a popular paradigm for network programming has led to the appearance of a number of SDN programming languages including *Formal*, *Nettle*, *NetCore*, *Formal*, *MaMa*, and *PNSE*, among others [10–12, 26, 27, 29, 40]. The details of these languages differ, but each seeks to provide high-level abstractions to simplify the task of specifying the packets

processing behavior of a network. In addition to SDN languages, a number of verification tools including *RSN*, *NetFlow*, *Flowlog*, and *VeriCon* are also being actively developed [2, 16, 17, 28]. As SDN is being deployed in production enterprise, data center, and wide-area networks [14, 19, 20], it is becoming clear that SDN is the next major step in the evolution of network technology and is destined to have a significant impact.

Previous work by Anderson et al. [1] introduced NetKat, a language and logic for specifying and verifying the packet-processing behavior of networks. NetKat provides general-purpose programming constructs such as parallel and sequential composition, conditional tests, and iteration, as well as special-purpose primitives for querying and modifying packet headers and encoding network topologies. The language allows the desired behavior of a network to be specified equationally. In contrast to competing approaches, NetKat has a formal mathematical semantics and an equational deductive system that is sound and complete over that semantics, as well as a PSPACE decision procedure. It is based on Kleene algebra with tests (KAT), an algebraic system for propositional program verification that has been extensively studied for nearly two decades [22]. Several practical applications of NetKat have been developed, including algorithms for testing reachability and non-interference and a systematic correctness proof for a compiler that translates programs to hardware instructions for SDN switches.

This paper develops the coalgebraic theory of NetKat, defines a new algorithm for deciding equivalence based on this technology, and presents a full implementation in OCaml. The new algorithm is significantly more efficient than the previous naive algorithm [1], which was PSPACE in the best case and the worst case, as it was based on the overestimation of a nondeterministic algorithm.

The contributions of this paper are both theoretical and practical. On the theoretical side, we introduce a new coalgebraic model of NetKat, including a specialized version of the Brzozowski derivative in both semantic and syntactic forms. We prove a version of Kleene’s theorems for NetKat that shows that the coalgebraic model is equivalent to the standard packet-processing and language models introduced previously [1]. A highlight of our theoretical development is a representation theorem showing that the Brzozowski derivative can be concisely encoded in matrix form. On the practical side, we develop a new coalgebraic decision procedure for term equivalence based on our theoretical results, along with a full implementation in OCaml. The algorithm constructs a bisimulation between coalgebra built from NetKat expressions via the Brzozowski derivative. The matrix representation enables us to exploit sparseness to obtain a significant reduction in the size of the state space. The implementation is very efficient in practice—it can verify reachability in a real-world campus network in less than a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full notice on the first page. Copyrights for components of this work owned by others than ACM must be honored. No component may be reproduced without permission from ACM. This work is derived from NetKat, a domain-specific language and logic for specifying and verifying network packet-processing functions. Copyright © 2014 ACM 978-1-4503-2548-6/14/01...\$15.00. http://dx.doi.org/10.1145/2570818.2570822.

POPL’15

GoNeCo



KATch: A Fast Symbolic Verifier for NetKAT*

MARK MOELLER[†], Cornell University, USA
JULES JACOBS[†], Cornell University, USA
OLIVIER SAVARY BELANGER, Galois, USA
DAVID DARAI, Galois, USA
COLE SCHLESINGER, Galois, USA
STEFFEN SMOLKA, Google, USA
NATE FOSTER, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA



PLDI’24

NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson
Swarthmore College *

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts Amherst *

Jean-Baptiste Jeannin
Carnegie Mellon University *

Dexter Kozen
Cornell University

Cole Schlesinger
Princeton University

David 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, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation has left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKat, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKat, including primitives for filtering, modifying, and transmitting packets, as well as sequential composition operators, and a Kleene star operator that iterates programs. We show that NetKat is an instance of a canonical and well-studied mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking reachability, proving non-interference properties that ensure isolation between programs, and establishing the correctness of compilation algorithms.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

Keywords Software-defined networking, Formal, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKat.

1. Introduction

Traditional network devices have been called “the last bastion of mainframe computing” [9]. Unlike modern computers, which are

* This work performed at Cornell University.

implemented with commodity hardware and programmed using standard interfaces, networks have been built the same way since the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, a revolution has taken place with the recent rise of *software-defined networking* (SDN). In SDN, a general-purpose *controller* manages a collection of programmable *switches*. The controller responds to network events such as new connections from hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly. Because the controller has a global view of the network, it is easy to use SDN to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault tolerance.

A major appeal of SDN is that it deflates open standards that any vendor can implement. For example, the OpenFlow API [21] clearly specifies the capabilities and behavior of switch hardware and defines a low-level language for manipulating their configurations. However, programs written directly for SDN platforms such as OpenFlow are also to assembly: easy for hardware to implement, but difficult for humans to write.

Network programming languages. In recent years, several different research groups have proposed domain-specific languages for SDN [2–7, 23–25, 31, 32]. The goal of these network programming languages is to raise the level of abstraction of network programs above hardware-oriented APIs such as OpenFlow, thereby making it easier to build sophisticated and reliable network applications. Previous work developed the design of the language and its standard semantics, proved its soundness and completeness of the logic, defined a PSPACE algorithm for deciding equivalence, and presented several practical applications.

This paper develops the coalgebraic theory of NetKat, including a specialized version of the Brzozowski derivative, and presents a new efficient algorithm for deciding the equational theory using bisimulation. The coalgebraic structure admits an efficient sparse representation that results in a significant reduction in the size of the state space. We discuss the details of our implementation and optimizations that explain NetKat’s exceptional actions and coalgebraic structure to yield significantly improved performance. We present results from experiments demonstrating that our tool is competitive with state-of-the-art tools on several benchmarks including all-pairs connectivity, loop-broken, and translation validation.

Categories and Subject Descriptors F.4.3 [Formal Languages]: Classes defined by grammars or automata

Keywords Coalgebra, Kleene algebra with tests, Brzozowski derivative, automatic network verification, NetKat.

1. Introduction

Networks have received widespread attention in recent years as a target for domain-specific language design. The emergence of *software-defined networking* (SDN) as a popular paradigm for network programming has led to the appearance of a number of SDN programming languages including Frigate, NetCore, NetCore, Puma, Maple, and PANS, among others [10–12, 26, 27, 29, 40]. The details of these languages differ, but each seeks to provide high-level abstractions to simplify the task of specifying the packets



Concurrent NetKAT

Modeling and analyzing stateful, concurrent networks

Jana Wagemaker¹ ✉, Nate Foster² , Tobias Kappé³ , Dexter Kozen² , Jurriaan Rot¹, and Alexandra Silva²

¹ Radboud University, Nijmegen, The Netherlands
Jana.Wagemaker@ru.nl
² Cornell University, Ithaca, New York, USA
³ ILLC, University of Amsterdam, The Netherlands

Abstract. We introduce Concurrent NetKAT (CNetKAT), an extension of NetKAT with operators for specifying and reasoning about concurrency in scenarios where multiple packets interact through state. We provide a model of the language based on partially-ordered multisets (pomsets), which are a well-established mathematical structure for defining the denotational semantics of concurrent languages. We provide a sound and complete axiomatization of this model, and we illustrate the use of CNetKAT through examples. More generally, CNetKAT can be understood as an algebraic framework for reasoning about programs with both local state (in packets) and global state (in a global store).

Keywords: Concurrent Kleene algebra, NetKAT, completeness, concurrency

StackAT: Infinite State Network Verification

JULES JACOBS, Cornell University, USA
NATE FOSTER, Cornell University, USA
TOBIAS KAPPÉ, Leiden University, The Netherlands
DEXTER KOZEN, Cornell University, USA
LILY SAADA, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA
JANA WAGEMAKER, Radboud University, The Netherlands

POPL’14

POPL’17

PLDI’19

ESOP’22

PLDI’25

...

2014

2015

2017

2019

2022

2024

2025

2026



A Coalgebraic Decision Procedure for NetKat

Nate Foster
Cornell University

Dexter Kozen
Cornell University

Mae Milano
Cornell University

Alexandra Silva
Radboud University Nijmegen

Laure Thompson
Cornell University

Abstract

NetKat is a domain-specific language and logic for specifying and verifying network packet-processing functions. It consists of Kleene algebra with tests (KAT) augmented with operators for routing and modifying packet headers and encoding network topologies. Previous work developed the design of the language and its standard semantics, proved its soundness and completeness of the logic, defined a PSPACE algorithm for deciding equivalence, and presented several practical applications.

This paper develops the coalgebraic theory of NetKat, including a specialized version of the Brzozowski derivative, and presents a new efficient algorithm for deciding the equational theory using bisimulation. The coalgebraic structure admits an efficient sparse representation that results in a significant reduction in the size of the state space. We discuss the details of our implementation and optimizations that explain NetKat’s exceptional actions and coalgebraic structure to yield significantly improved performance. We present results from experiments demonstrating that our tool is competitive with state-of-the-art tools on several benchmarks including all-pairs connectivity, loop-broken, and translation validation.

Categories and Subject Descriptors F.4.3 [Formal Languages]: Classes defined by grammars or automata

Keywords Coalgebra, Kleene algebra with tests, Brzozowski derivative, automatic network verification, NetKat.

1. Introduction

Networks have received widespread attention in recent years as a target for domain-specific language design. The emergence of *software-defined networking* (SDN) as a popular paradigm for network programming has led to the appearance of a number of SDN programming languages including Frigate, NetCore, NetCore, Puma, Maple, and PANS, among others [10–12, 26, 27, 29, 40]. The details of these languages differ, but each seeks to provide high-level abstractions to simplify the task of specifying the packets

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with permission is permitted. For all other use, permission should be sought from ACM or the copyright owner. This paper is part of the Proceedings of the ACM on Programming Languages, Vol. 8, No. 1, January 2024, 1–25. Copyright © 2024, ACM 978-1-4503-2544-8/24/01...\$15.00. <http://doi.org/10.1145/3627088.3627092>.

GoNeCo

POPL’15

KATch: A Fast Symbolic Verifier for NetKAT*

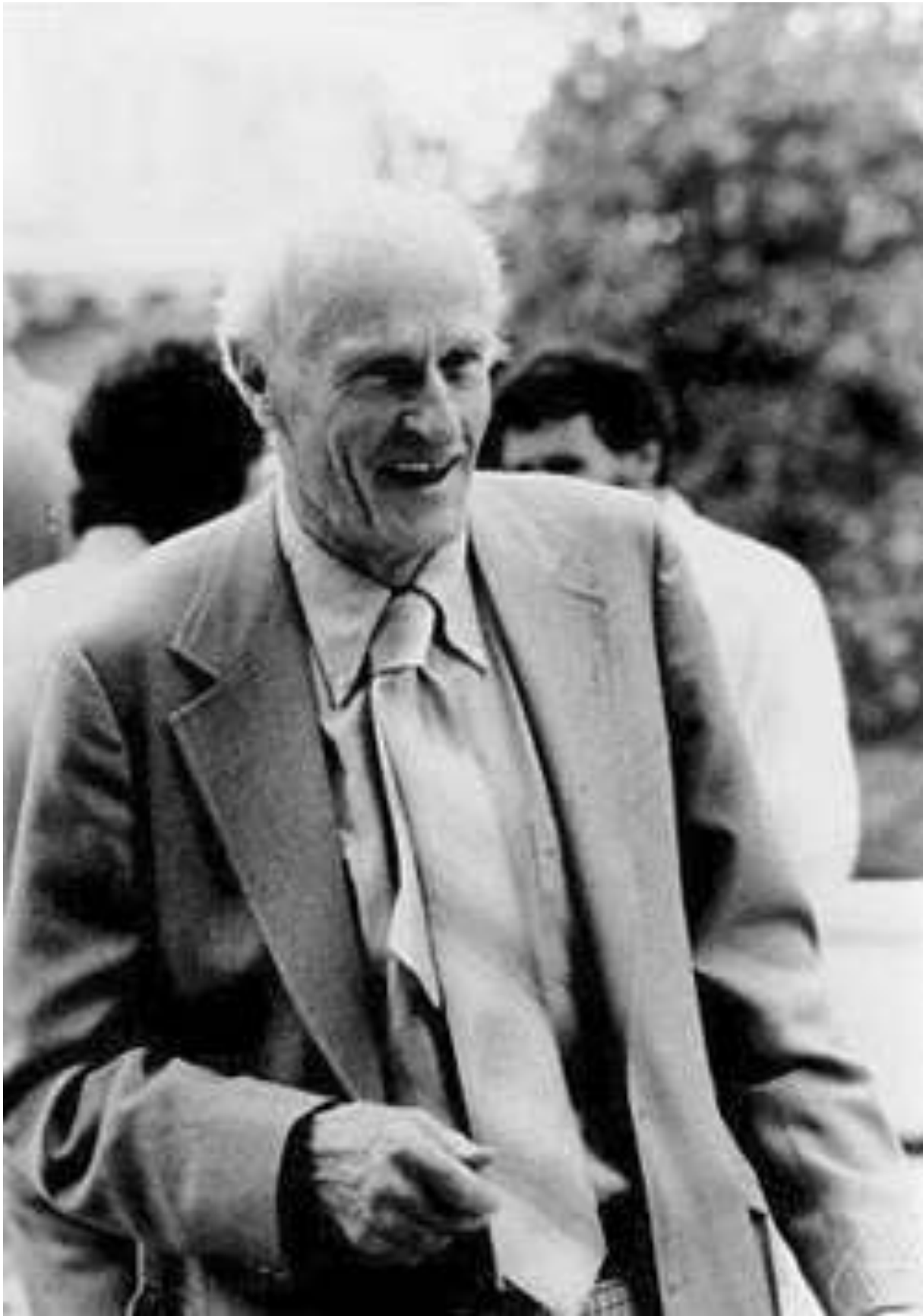
MARK MOELLER[†], Cornell University, USA
JULES JACOBS[†], Cornell University, USA
OLIVIER SAVARY BELANGER, Galois, USA
DAVID DARAI, Galois, USA
COLE SCHLESINGER, Galois, USA
STEFFEN SMOLKA, Google, USA
NATE FOSTER, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA



PLDI’24

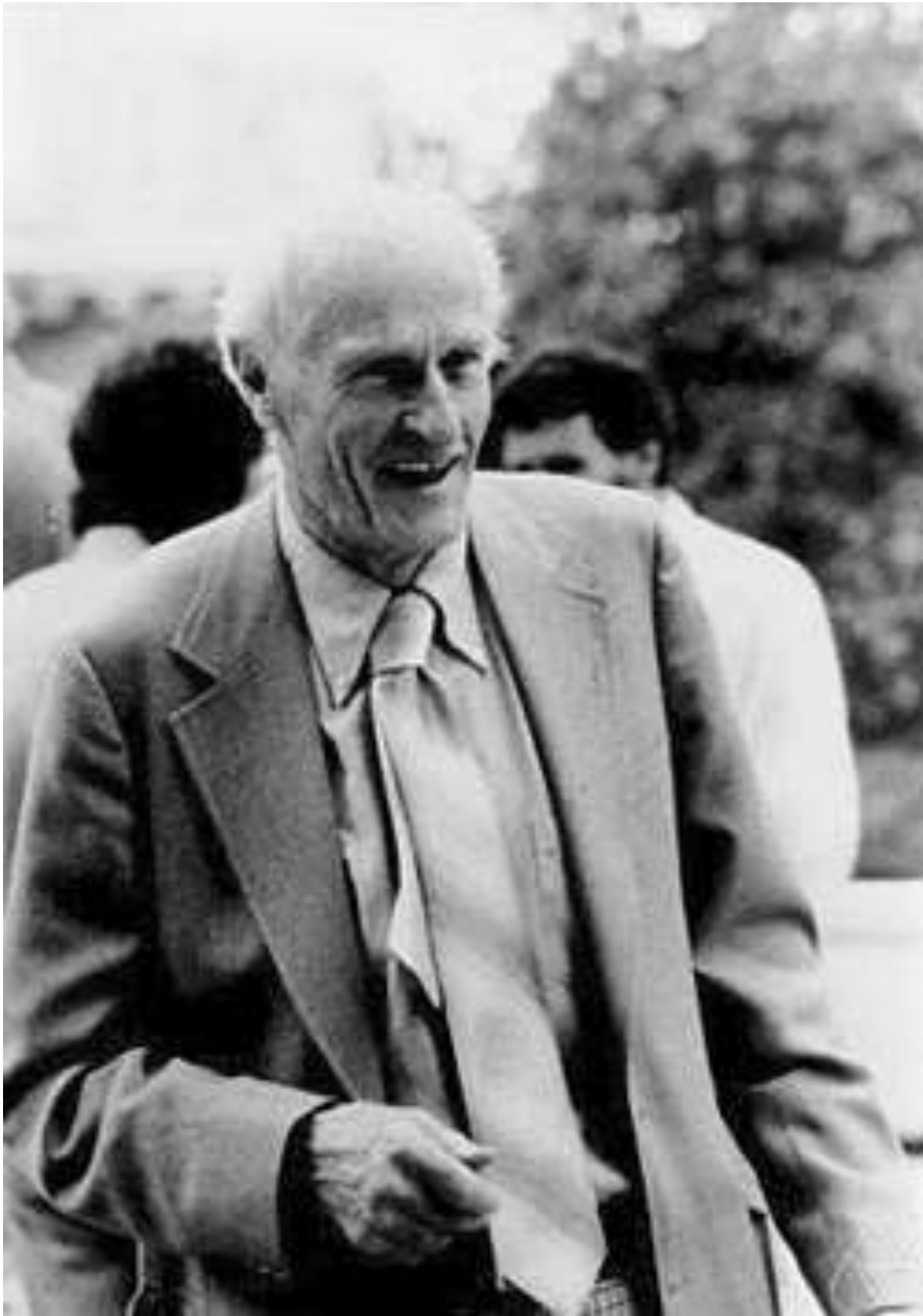
Let's go back to the beginning...

Kleene Algebra



Stephen Cole Kleene
(1909—1994)

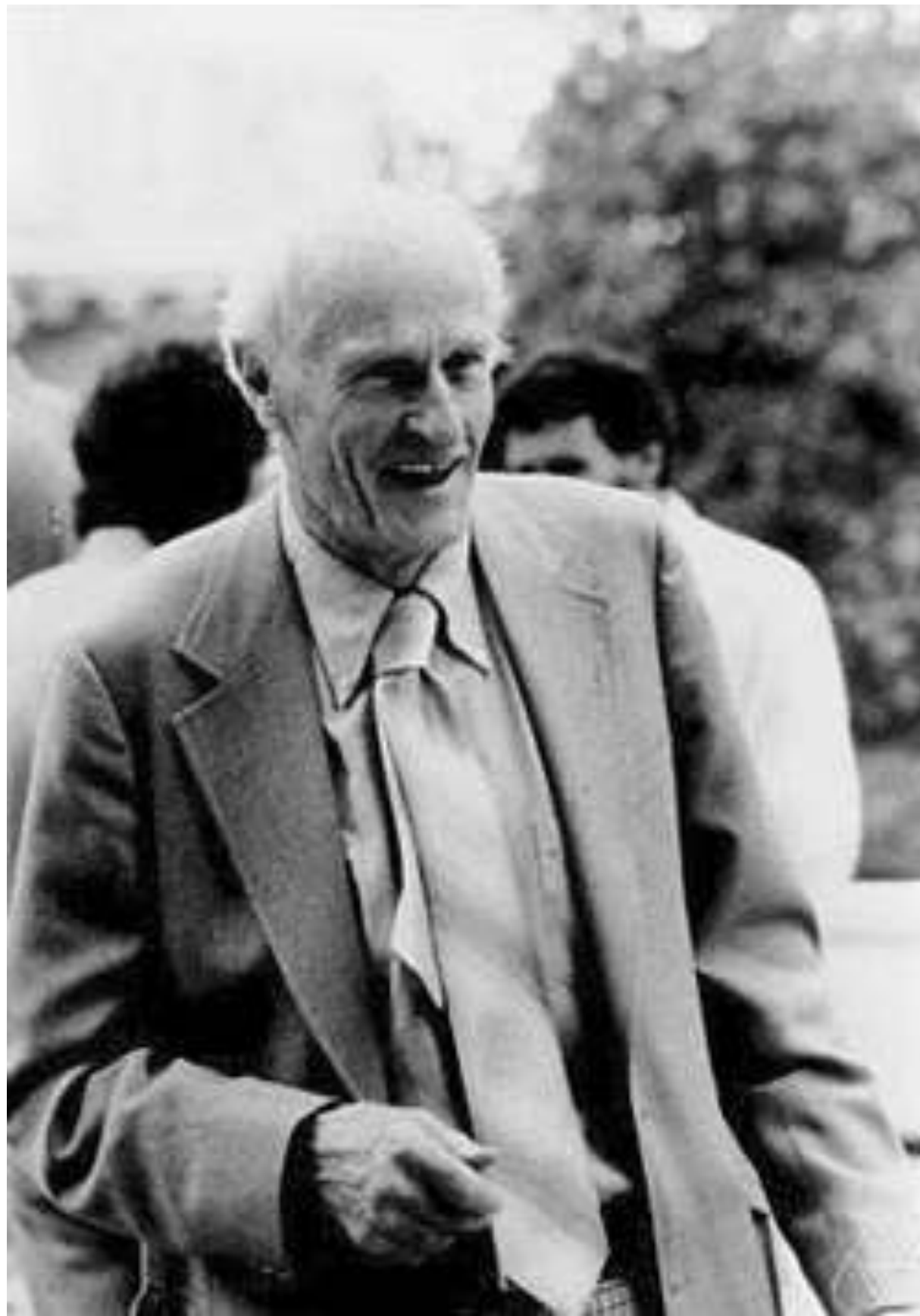
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

$p, q ::=$

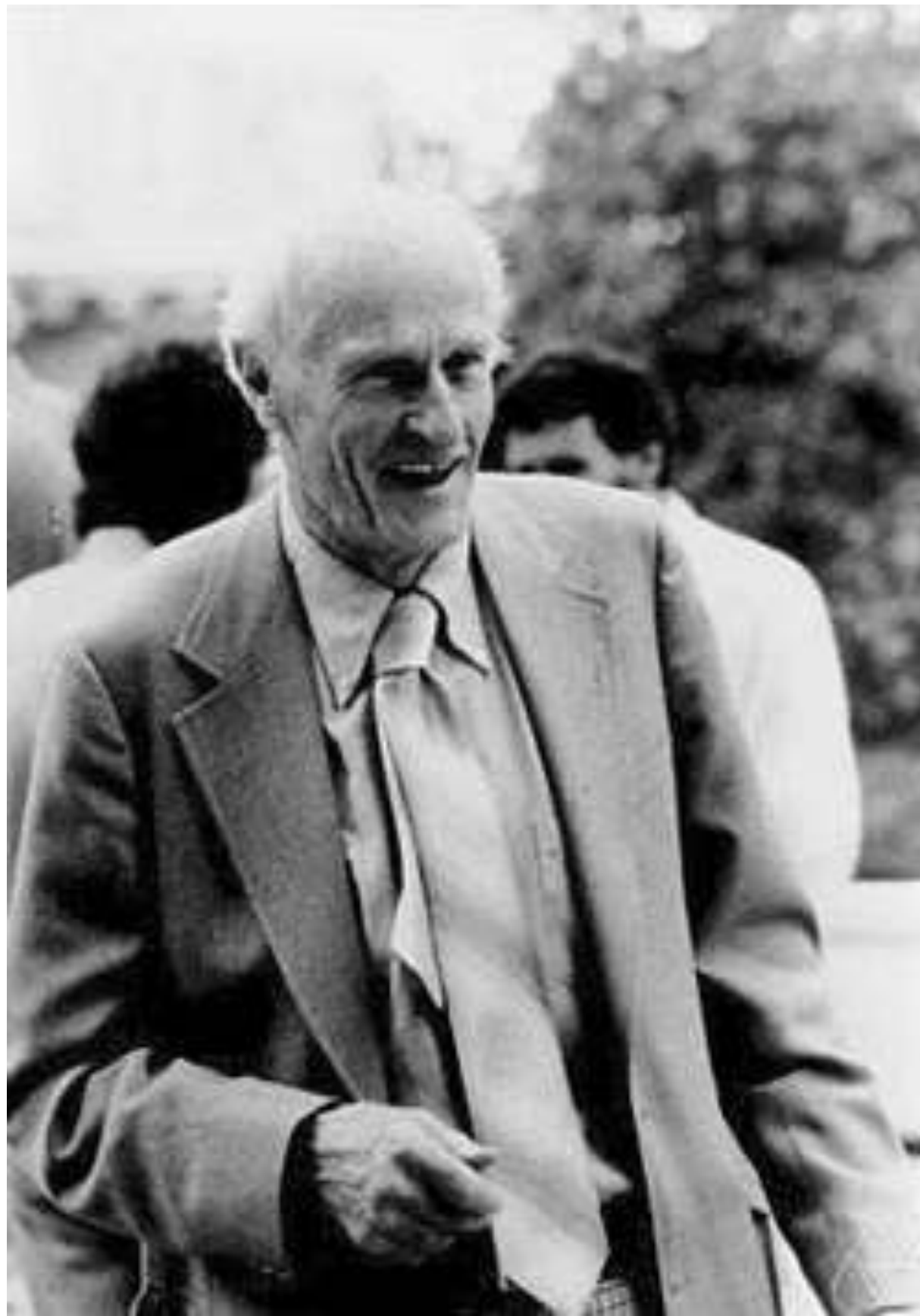
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=  
      0      // abort
```

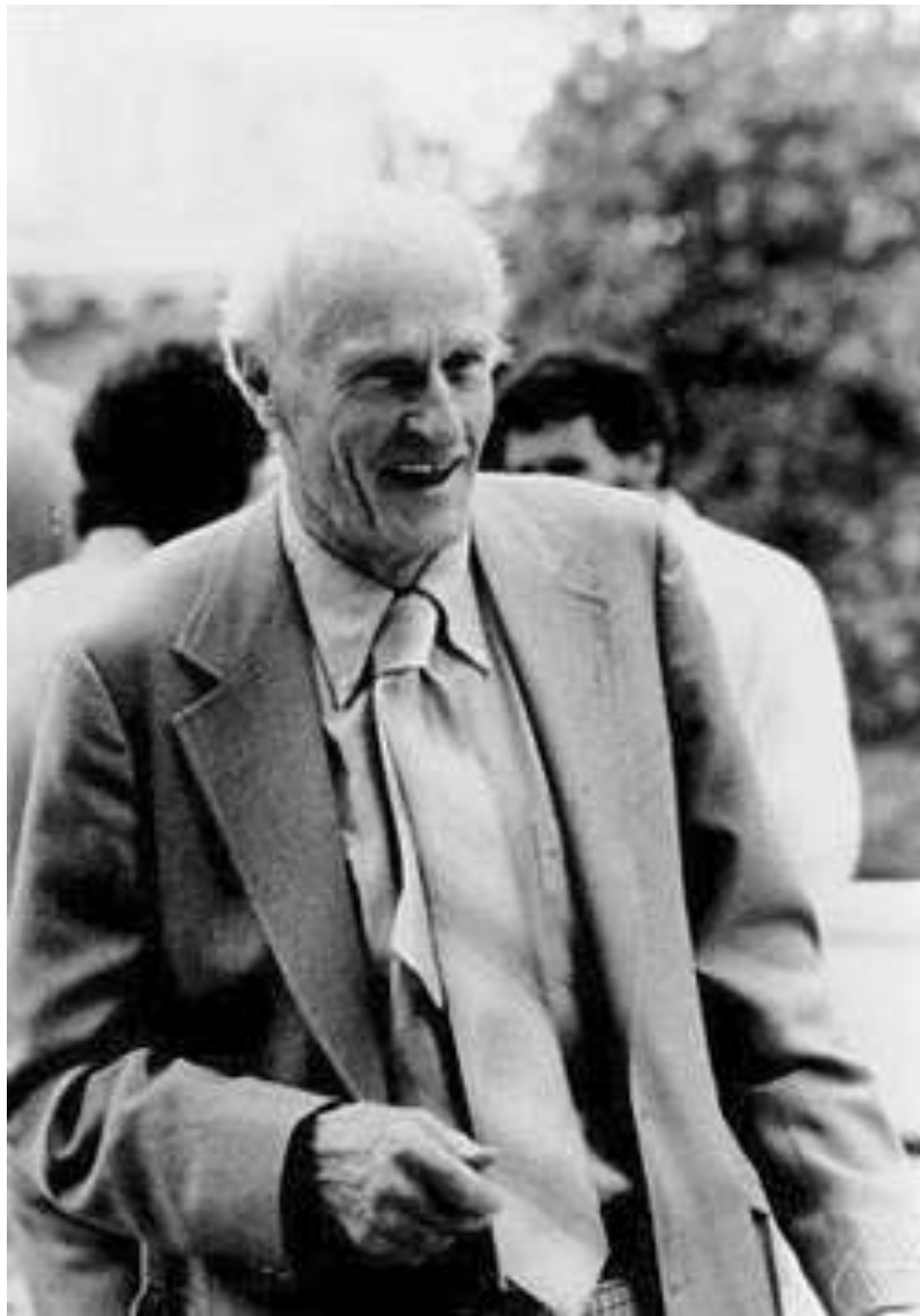
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=  
      0      // abort  
    | 1      // skip
```

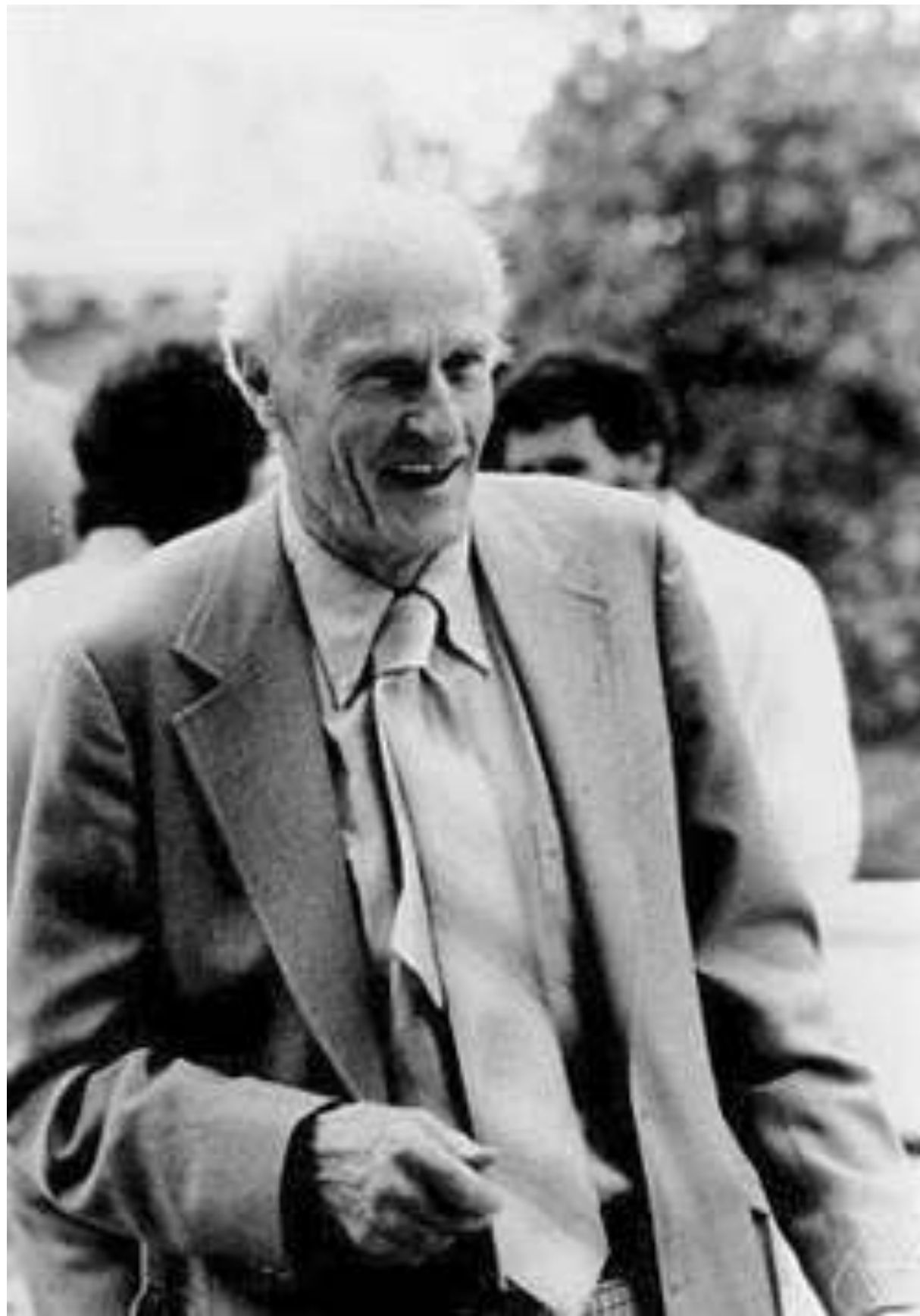
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=  
      0      // abort  
    | 1      // skip  
    | a      // action
```

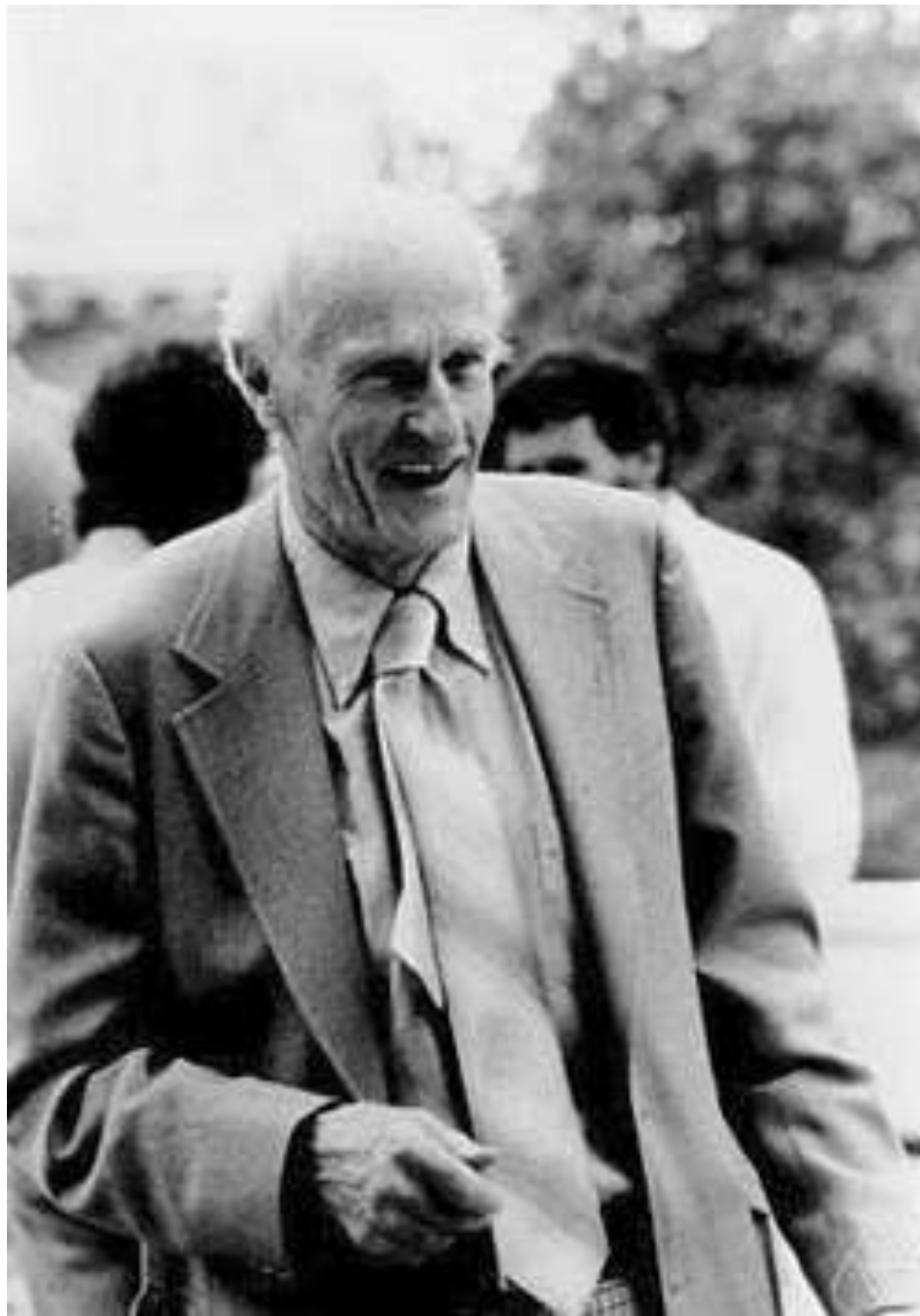
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=  
      0           // abort  
    | 1           // skip  
    | a           // action  
    | p + q       // choice
```

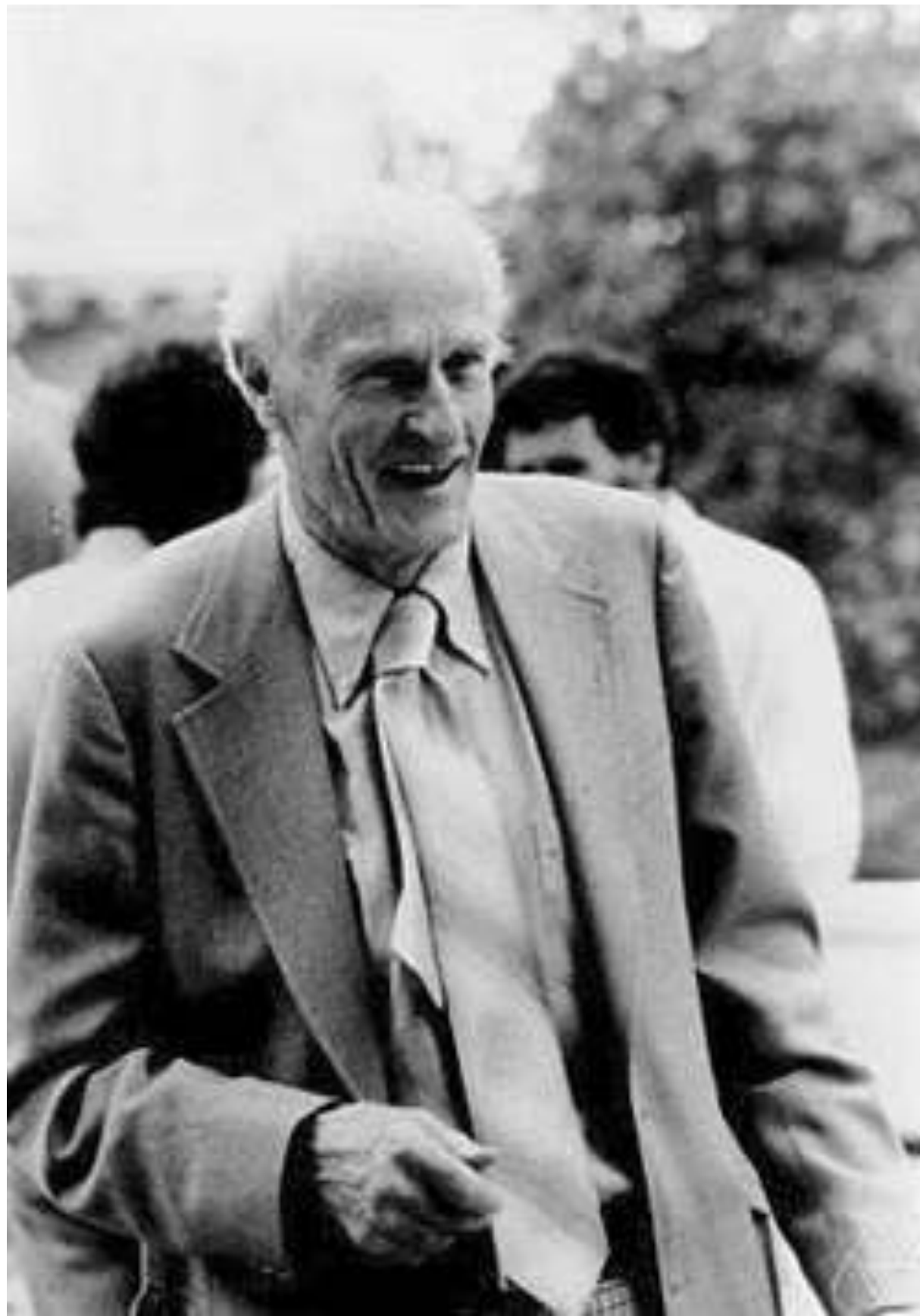
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=  
      0           // abort  
    | 1           // skip  
    | a           // action  
    | p + q       // choice  
    | p · q       // sequential composition
```

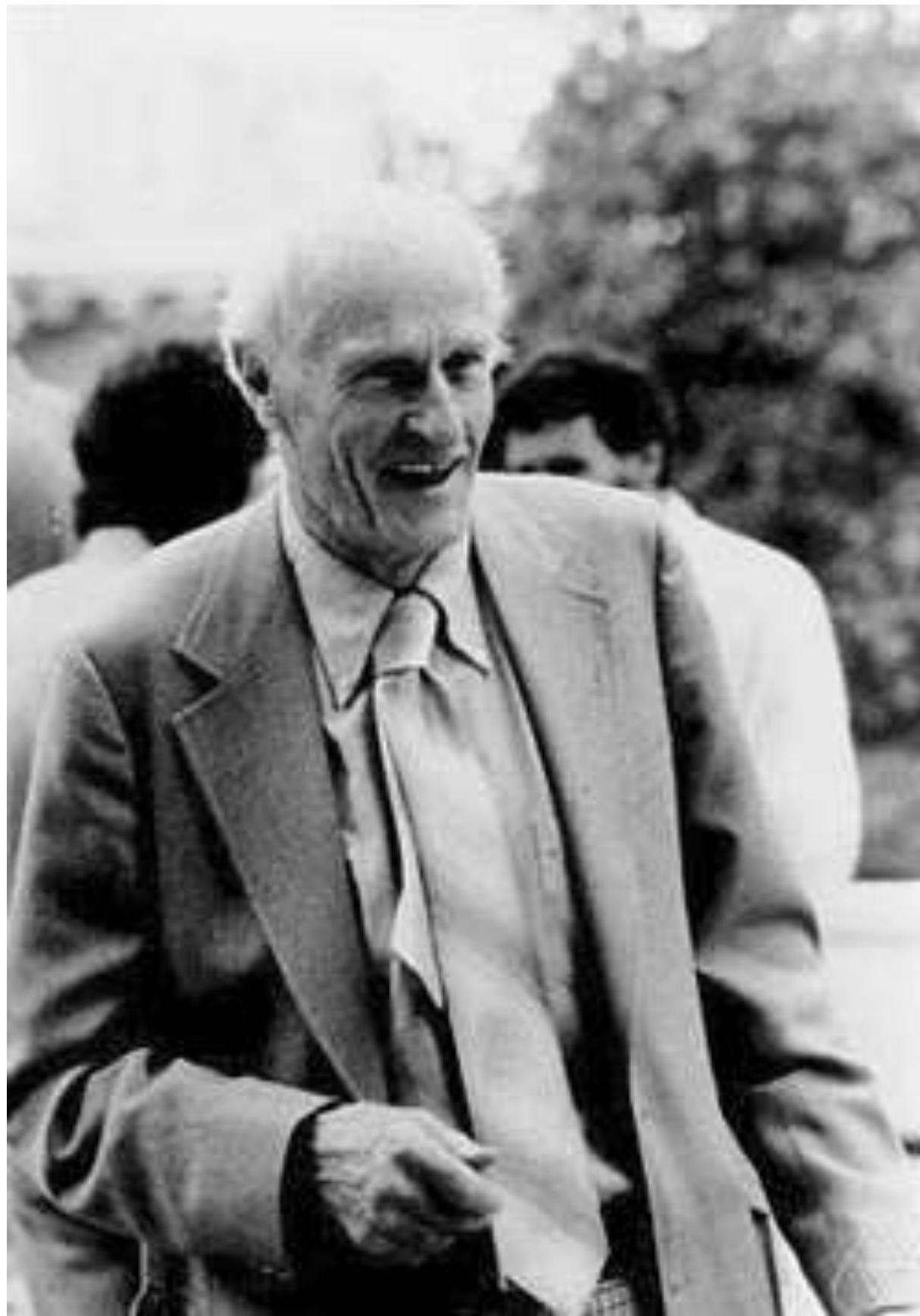
Kleene Algebra



Stephen Cole Kleene
(1909—1994)

```
p, q ::=
    0           // abort
  | 1           // skip
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

Kleene Algebra

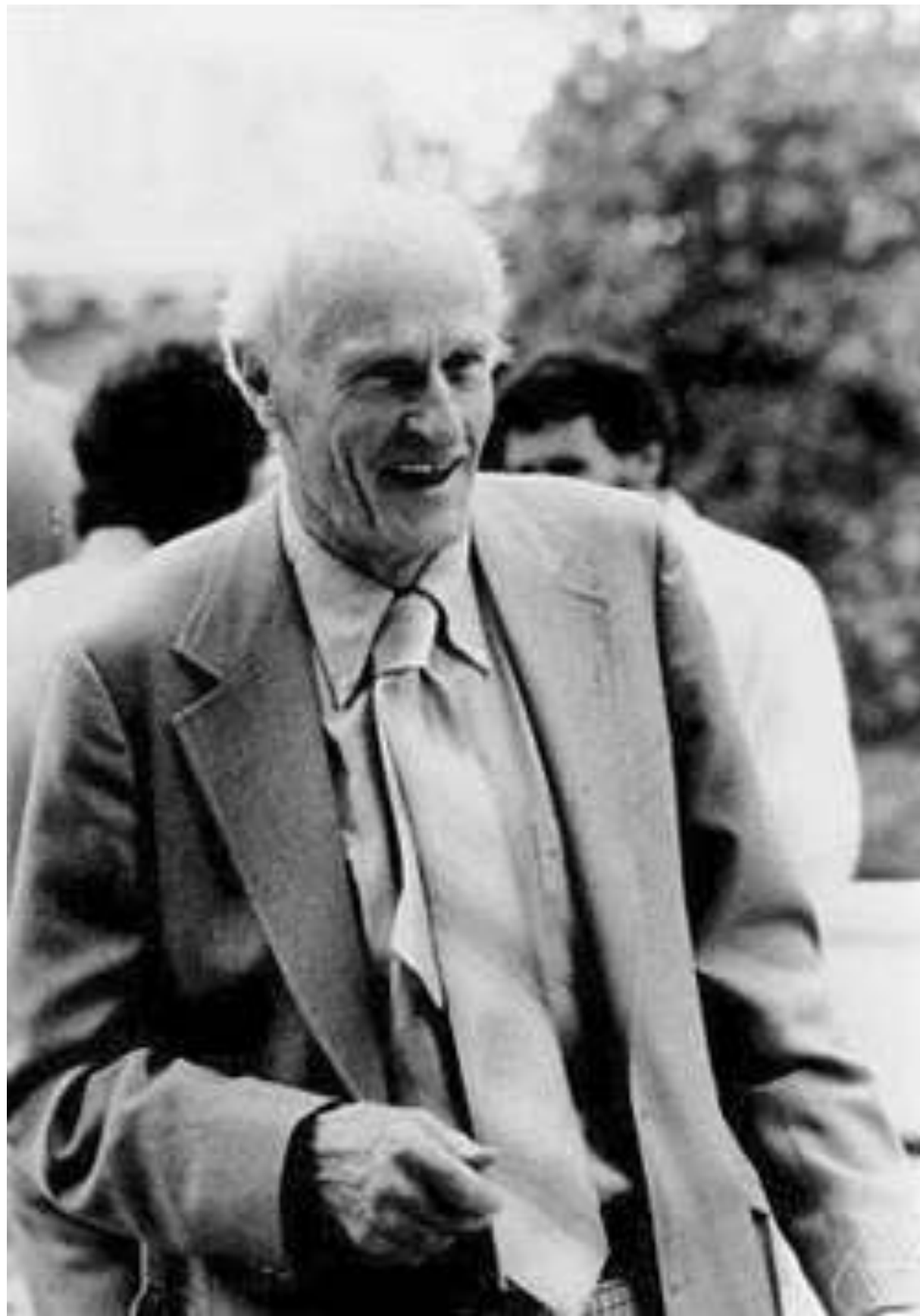


Stephen Cole Kleene
(1909—1994)

```
p, q ::=
    0           // abort
  | 1           // skip
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

```
(1+a) · (1+b) // ' ' or 'a' or 'b' or 'ab'
```

Kleene Algebra

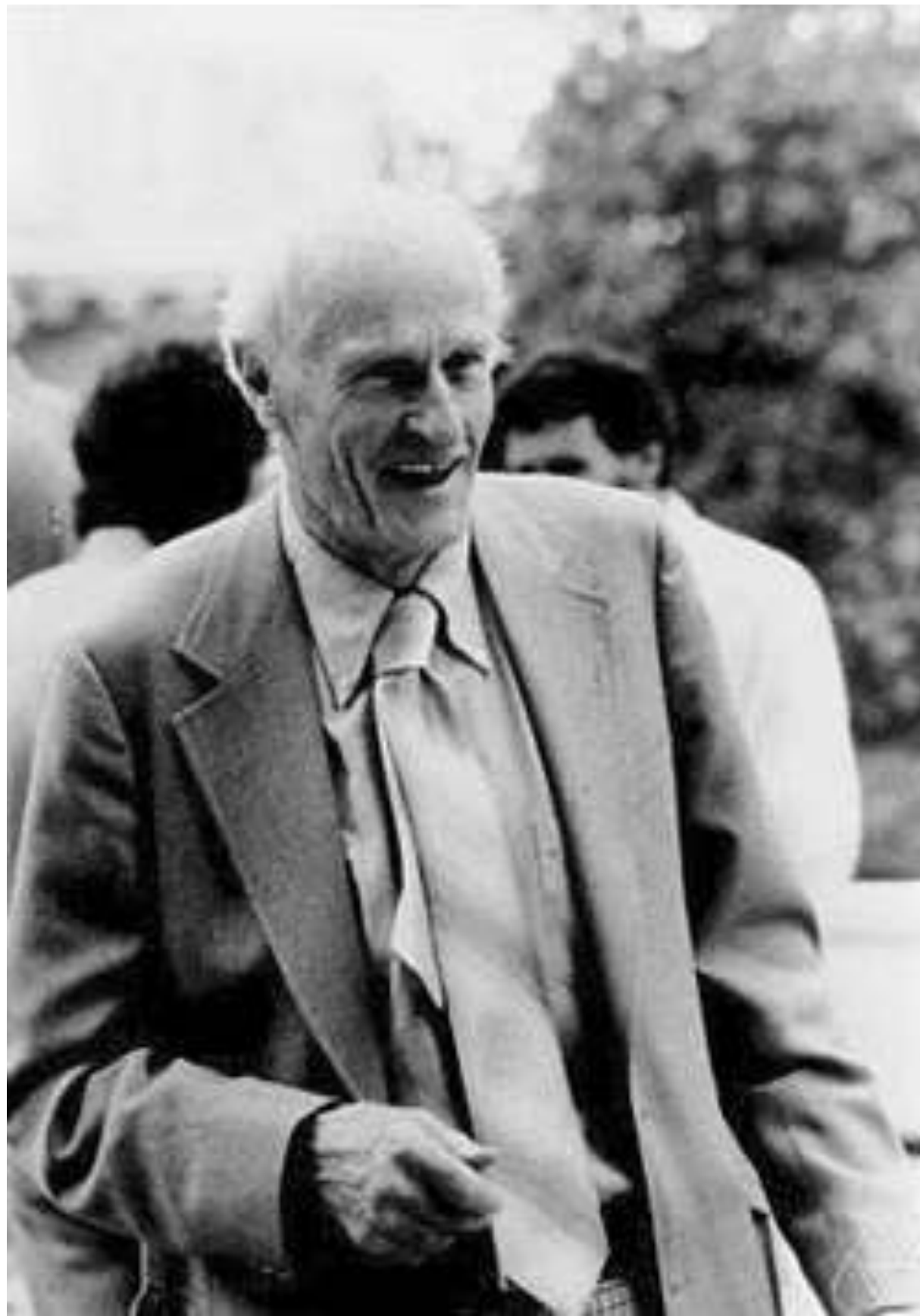


Stephen Cole Kleene
(1909—1994)

```
p, q ::=
    0           // abort
    | 1         // skip
    | a         // action
    | p + q     // choice
    | p · q     // sequential composition
    | p*        // iteration

(1+a) · (1+b) // ' ' or 'a' or 'b' or 'ab'
(a+b)*       // any sequence of 'a's and 'b's
```

Kleene Algebra

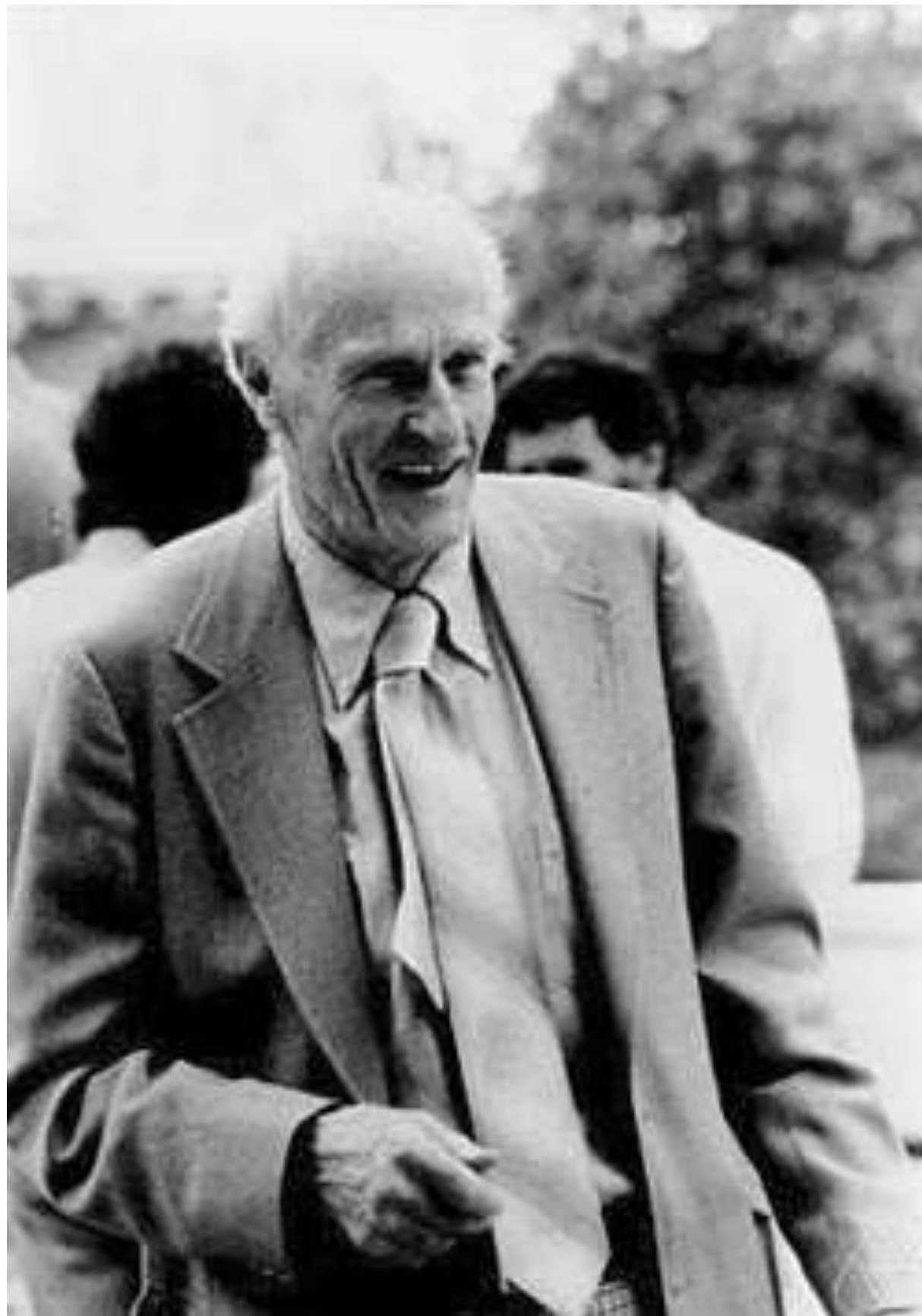


Stephen Cole Kleene
(1909—1994)

```
p, q ::=
    0           // abort
  | 1           // skip
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration

(1+a) · (1+b) // ' ' or 'a' or 'b' or 'ab'
(a+b)*        // any sequence of 'a's and 'b's
(a*b)* · a*   // ??
```

Kleene Algebra



Stephen Cole Kleene
(1909–1994)

```
p, q ::=
    0           // abort
  | 1           // skip
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

```
(1+a) · (1+b) // ' ' or 'a' or 'b' or 'ab'
(a+b)*        // any sequence of 'a's and 'b's
(a*b)* · a*   // ??
```

Theorem. $(a+b)^* \equiv (a^*b)^* \cdot a^*$

Kleene Algebra: Semantics

Kleene Algebra: Semantics

Language of traces
of actions

$\llbracket p \rrbracket \subseteq A^*$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

$$\text{where } \llbracket p \rrbracket^0 = \{ \epsilon \}$$

Kleene Algebra: Semantics

Language of traces
of actions

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

$$\text{where } \llbracket p \rrbracket^0 = \{ \epsilon \}$$

$$\text{and } \llbracket p \rrbracket^{i+1} = \llbracket p \rrbracket \cdot \llbracket p \rrbracket^i$$

Kleene Algebra: Semantics

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

$$\text{where } \llbracket p \rrbracket^0 = \{ \epsilon \}$$

$$\text{and } \llbracket p \rrbracket^{i+1} = \llbracket p \rrbracket \cdot \llbracket p \rrbracket^i$$

$$(a+b)^* \equiv (a^*b)^* \cdot a^*$$

def.



$$\llbracket (a+b)^* \rrbracket = \llbracket (a^*b)^* \cdot a^* \rrbracket$$

Kleene Algebra: Semantics

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

$$\text{where } \llbracket p \rrbracket^0 = \{ \epsilon \}$$

$$\text{and } \llbracket p \rrbracket^{i+1} = \llbracket p \rrbracket \cdot \llbracket p \rrbracket^i$$

$$(a+b)^* \equiv (a^*b)^* \cdot a^*$$

def.



$$\llbracket (a+b)^* \rrbracket = \llbracket (a^*b)^* \cdot a^* \rrbracket$$

How do we prove this
efficiently?



Kleene Algebra: Semantics

$$\llbracket p \rrbracket \subseteq A^*$$

$$\llbracket 0 \rrbracket = \{ \}$$

$$\llbracket 1 \rrbracket = \{ \epsilon \}$$

$$\llbracket a \rrbracket = \{ a \cdot \epsilon \}$$

$$\llbracket p \cdot q \rrbracket = \llbracket p \rrbracket \cdot \llbracket q \rrbracket$$

$$\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$$

$$\llbracket p^* \rrbracket = \bigcup_i \llbracket p \rrbracket^i$$

$$\text{where } \llbracket p \rrbracket^0 = \{ \epsilon \}$$

$$\text{and } \llbracket p \rrbracket^{i+1} = \llbracket p \rrbracket \cdot \llbracket p \rrbracket^i$$

$$(a+b)^* \equiv (a^*b)^* \cdot a^*$$

def.



$$\llbracket (a+b)^* \rrbracket = \llbracket (a^*b)^* \cdot a^* \rrbracket$$

How do we prove this efficiently?



Kleene's Theorem. TFAE:

1. $L = \llbracket e \rrbracket$, for some regular expression e
2. $L = L(M)$, for some deterministic finite automaton M

Kleene Algebra: Fundamental Results

Kleene Algebra: Fundamental Results

└ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

Kleene Algebra: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

✓ **Verification**

Can we automatically check whether two expressions are equivalent?

Kleene Algebra: Fundamental Results

└ Axiomatization

Can we characterize the set of valid program equalities/refinements?

Yes: Salomaa (1966); Kozen (1990)

✓ Verification

Can we automatically check whether two expressions are equivalent?

Kleene Algebra: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

Yes: Salomaa (1966); Kozen (1990)

✓ **Verification**

Can we automatically check whether two expressions are equivalent?

Yes: Stockmeyer (1974); Bonchi-Pous (2015)

Kleene Algebra with Tests



Dexter Kozen

```
p, q ::=
    0           // abort
  | 1           // skip
    a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

Kleene Algebra with Tests



Dexter Kozen

```
p, q ::=  
    0           // abort  
    | 1         // skip  
    | b, ~b     // Boolean tests  
    | a         // action  
    | p + q     // choice  
    | p · q     // sequential composition  
    | p*        // iteration
```

Kleene Algebra with Tests



Dexter Kozen

```
p, q ::=
    0           // abort
  | 1           // skip
  | b, ~b       // Boolean tests
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

Double meaning
+ = disjunction
· = conjunction

Kleene Algebra with Tests



Dexter Kozen

```
p, q ::=
    0           // abort
  | 1           // skip
  | b, ~b       // Boolean tests
  | a           // action
  | p + q       // choice
  | p · q       // sequential composition
  | p*          // iteration
```

```
if b then p else q = b · p + ~b · q
```

Kleene Algebra with Tests



Dexter Kozen

```
p, q ::=  
    0           // abort  
    | 1         // skip  
    | b, ~b     // Boolean tests  
    | a         // action  
    | p + q     // choice  
    | p · q     // sequential composition  
    | p*        // iteration
```

```
if b then p else q = b · p + ~b · q  
while b do p = (b · p)* · ~b
```

Kleene Algebra with Tests: Fundamental Results

Kleene Algebra with Tests: Fundamental Results

└ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

Kleene Algebra with Tests: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

✓ **Verification**

Can we automatically check whether two expressions are equivalent?

Kleene Algebra with Tests: Fundamental Results

└ Axiomatization

Can we characterize the set of valid program equalities/refinements?

Yes: Kozen (1997, 2001)

✓ Verification

Can we automatically check whether two expressions are equivalent?

Kleene Algebra with Tests: Fundamental Results

└ Axiomatization

Can we characterize the set of valid program equalities/refinements?

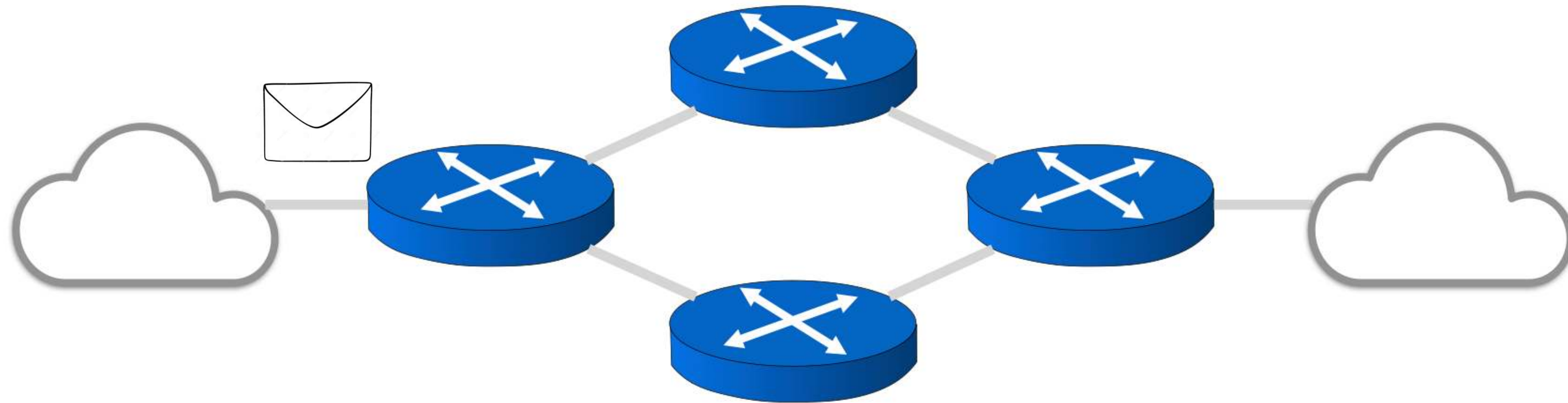
Yes: Kozen (1997, 2001)

✓ Verification

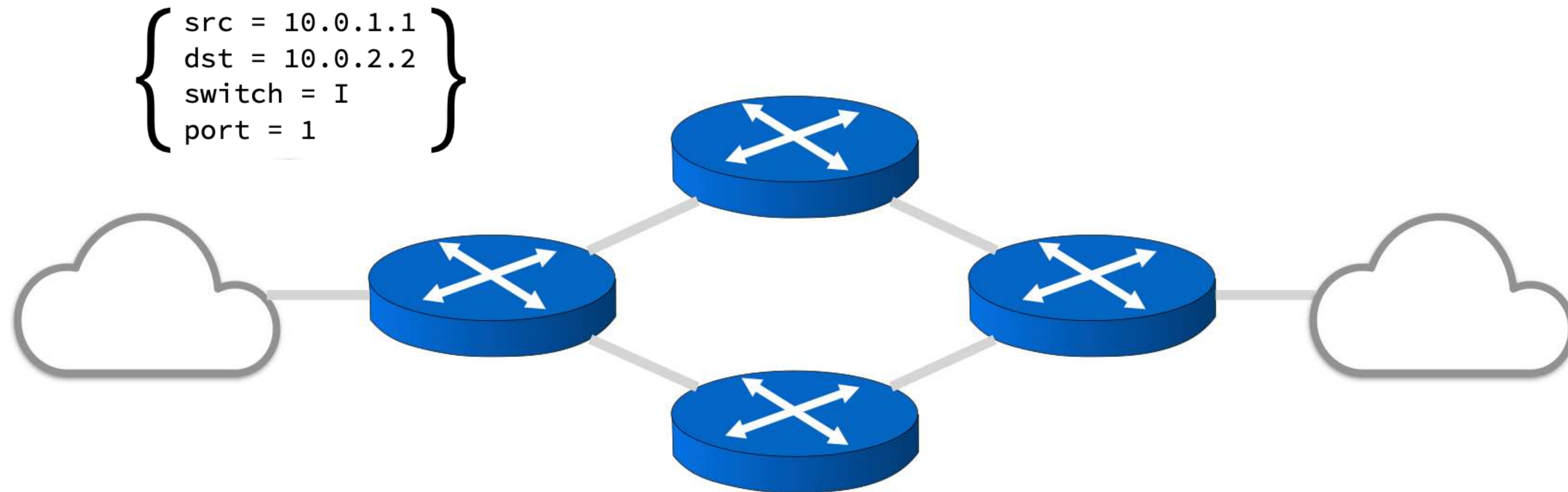
Can we automatically check whether two expressions are equivalent?

Yes: Chen and Pucella (2004); Kozen (2010); Pous (2016)

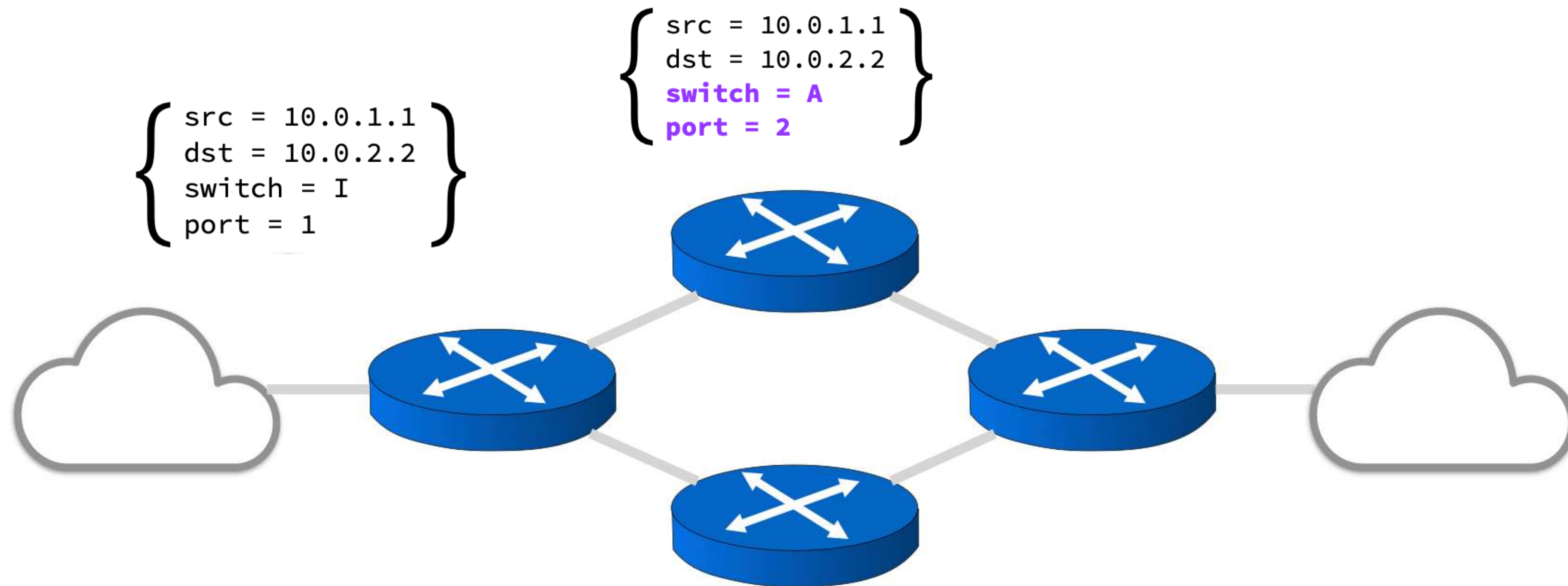
Packets: Records of fixed-width data



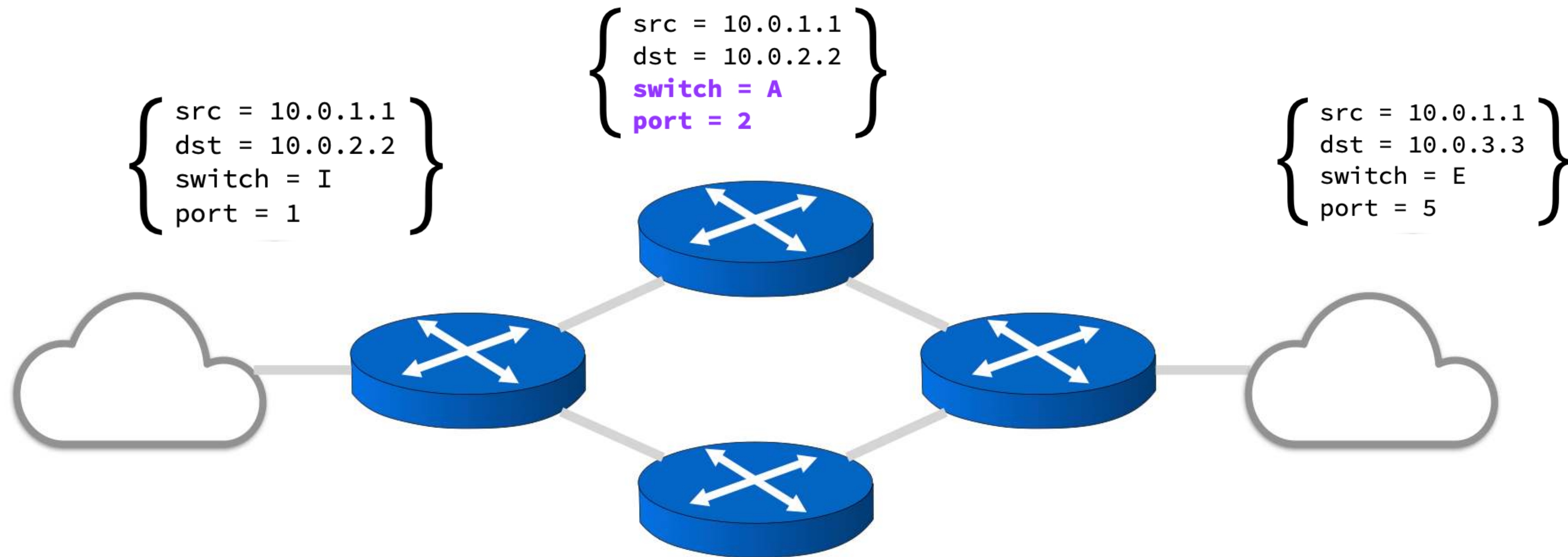
Packets: Records of fixed-width data



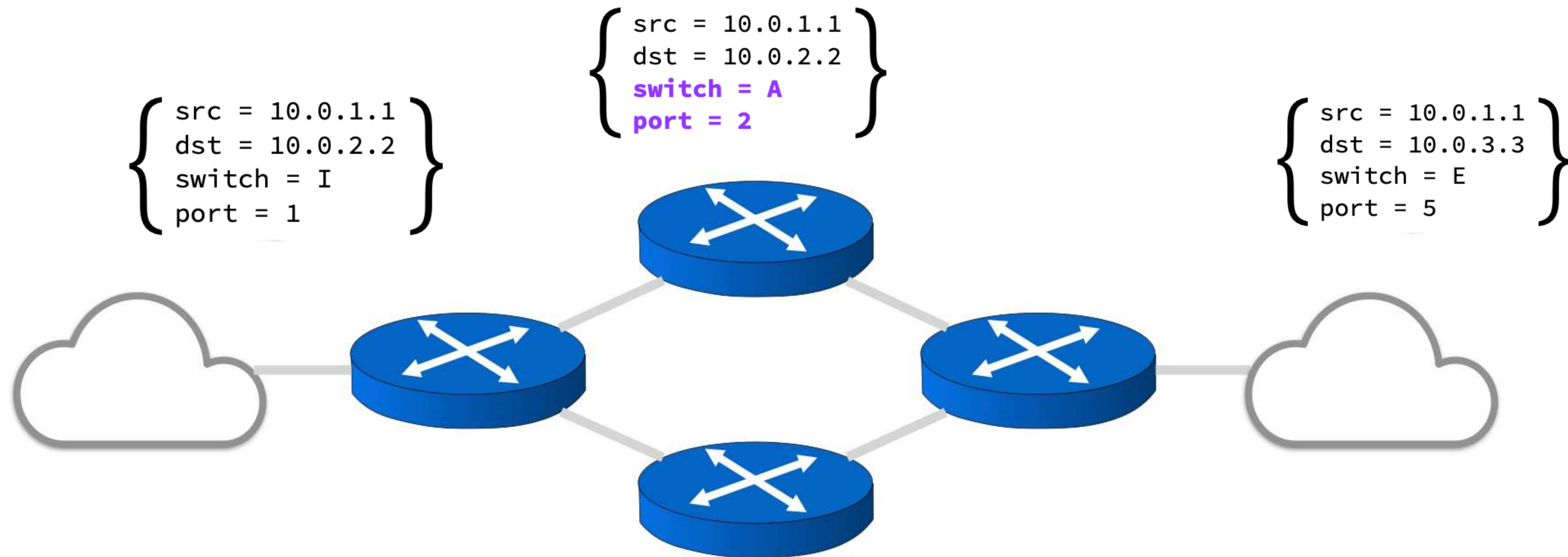
Packets: Records of fixed-width data



Packets: Records of fixed-width data



Packets: Records of fixed-width data



NetKAT = KAT where tests + actions are interpreted over packets!

NetKAT

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

NetKAT

`p, q ::=`

`| skip`

`| drop`

`| f = n`

`| !p`

`| f := n`

`| p ; q`

`| p + q`

`| p*`

`| dup`

Packet Classification

To model match-action tables, need predicates on packet headers (and other variables)

Match	Action
<code>tcp_dst=22; ip_dst=10.0.1.1</code>	<code>allow</code>
<code>tcp_dst=22</code>	<code>deny</code>
<code>ip_dst=10.0.2.2</code>	<code>allow</code>
<code>ip_dst=10.0.3.3</code>	<code>allow</code>
<code>*</code>	<code>deny</code>

NetKAT

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

NetKAT

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

Transformations

To model behavior of switches,
need imperative updates on
packets and a way to delimit
end of processing at a switch

$$A \xRightarrow{\text{def}} B$$

dup; sw = A; sw := B; dup

NetKAT

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

NetKAT

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

Compositionality

To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

Iteration

To model the potentially iterated processing performed via the topology, need general loops

```
if p then q else r  $\stackrel{\text{def}}{=}$  (p ; q) + (!p ; r)  
while p do q  $\stackrel{\text{def}}{=}$  (p ; q)* ; !p
```

NetKAT - semantics

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

Informal: NetKAT programs denote functions that take an input packet and produce a set of packet traces (i.e., non-empty lists).

Formal: $\llbracket p \rrbracket \in \text{Packet} \rightarrow \mathcal{P}(\text{Packet}^+)$

NetKAT - semantics

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

Informal: NetKAT programs denote functions that take an input packet and produce a set of packet traces (i.e., non-empty lists).

Formal: $\llbracket p \rrbracket \in \text{Packet} \rightarrow \mathcal{P}(\text{Packet}^+)$

Q: Why a set?

A: Enables dropping packets + multicast

Q: Why a trace?

A: Captures end-to-end forwarding path

Q: Why a function?

A: Simple model of “mostly stateless” forwarding

NetKAT - modeling the network

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

```
t = if sw=s1 then dup;l1  
    else if sw=s1 then dup;l2 ...  
s = match-action if-then-else  
  
net = (s;t)*  
// iterate packets going through  
switches and links  
  
sw=A ; net ; sw = B  $\equiv$  0  
// Non-Reachability
```

NetKAT - modeling the network

```
p, q ::=  
| skip  
| drop  
| f = n  
| !p  
| f := n  
| p ; q  
| p + q  
| p*  
| dup
```

```
t = if sw=s1 then dup;l1  
    else if sw=s1 then dup;l2 ...  
s = match-action if-then-else  
  
net = (s;t)*  
// iterate packets going through  
switches and links  
  
sw=A ; net ; sw = B  $\equiv$  0  
// Non-Reachability
```

Verification = Program Equivalence

NetKAT: Fundamental Results

NetKAT: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

NetKAT: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

✓ **Verification**

Can we automatically check whether two expressions are equivalent?

NetKAT: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

Yes: POPL'14

✓ **Verification**

Can we automatically check whether two expressions are equivalent?

NetKAT: Fundamental Results

⊢ **Axiomatization**

Can we characterize the set of valid program equalities/refinements?

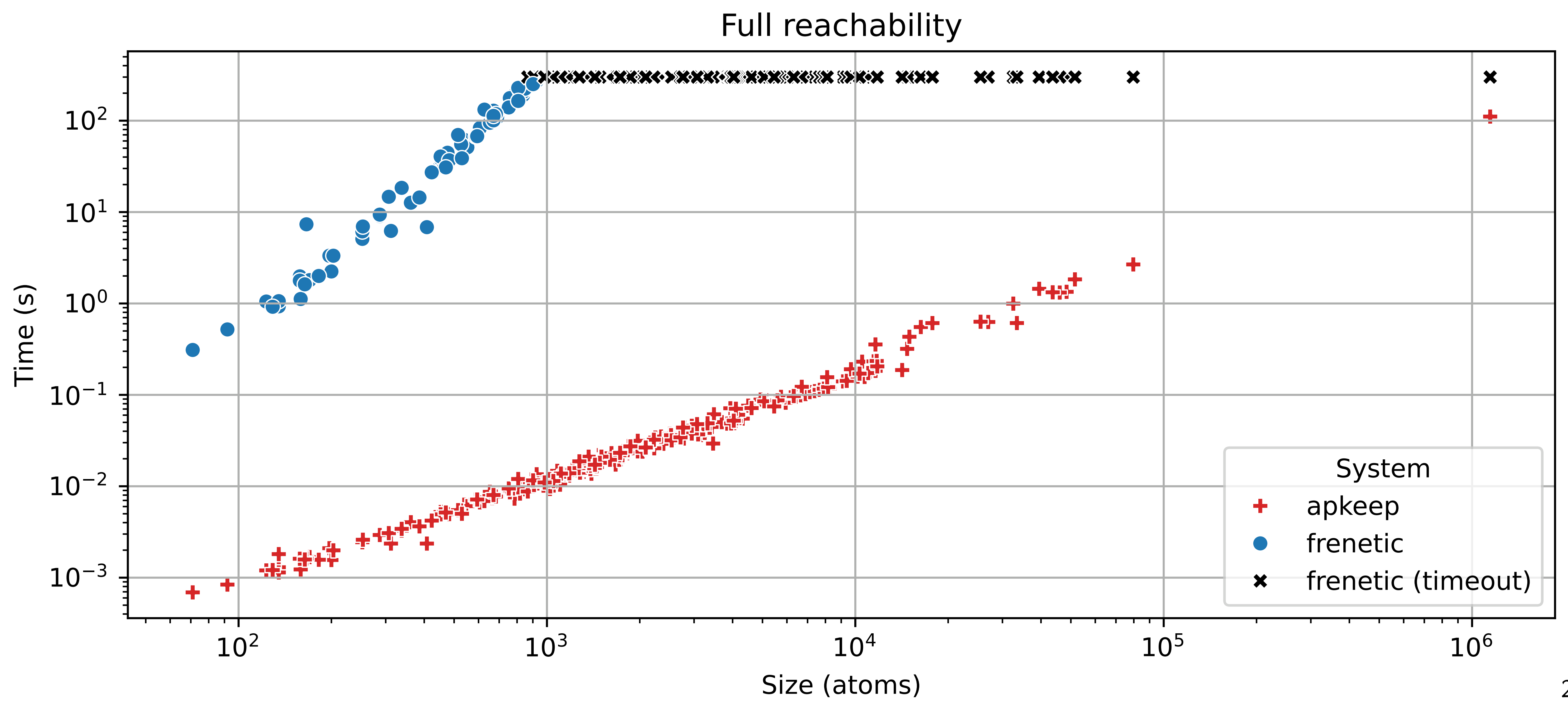
Yes: POPL'14

✓ **Verification**

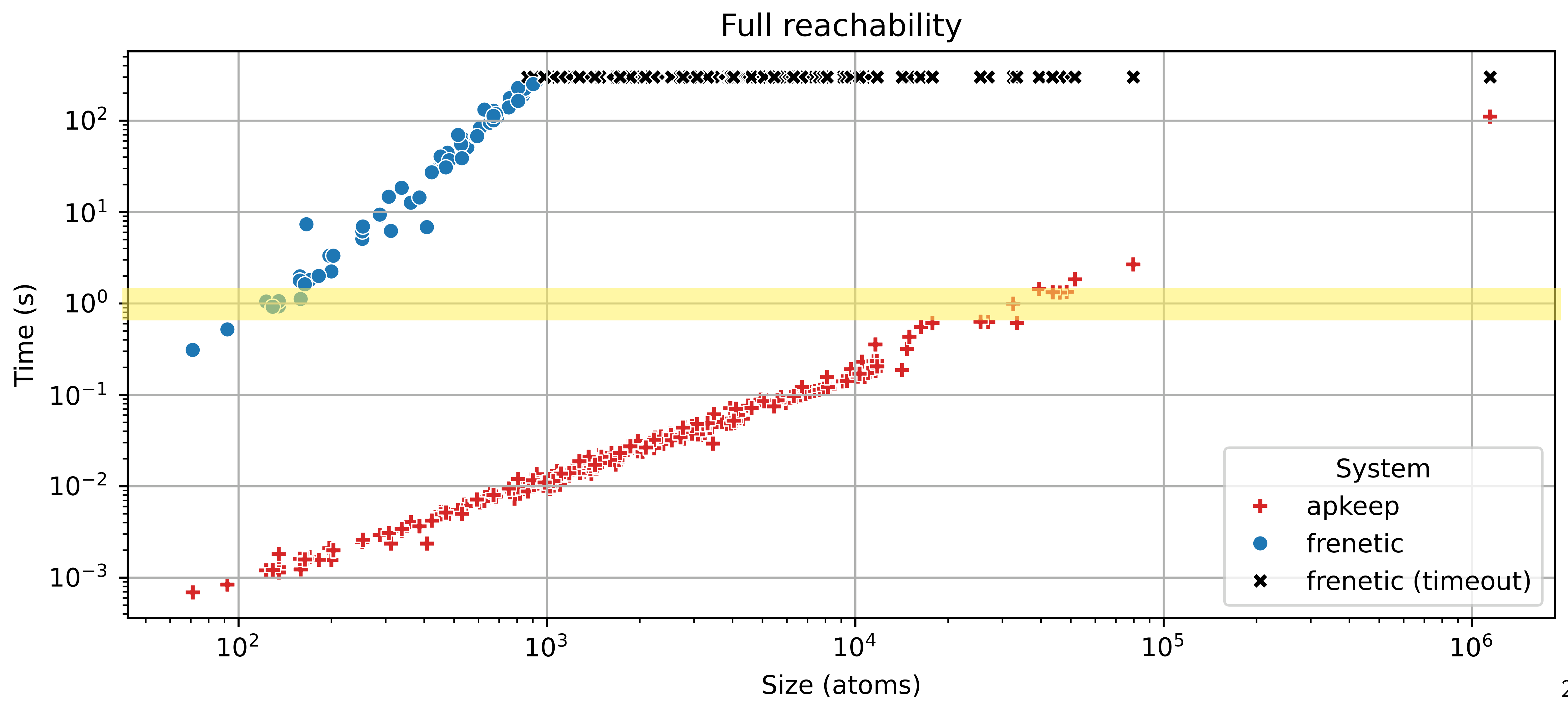
Can we automatically check whether two expressions are equivalent?

Yes: POPL'15, PLDI'24

NetKAT (POPL 2015) and **APKeep** (NSDI 2020)



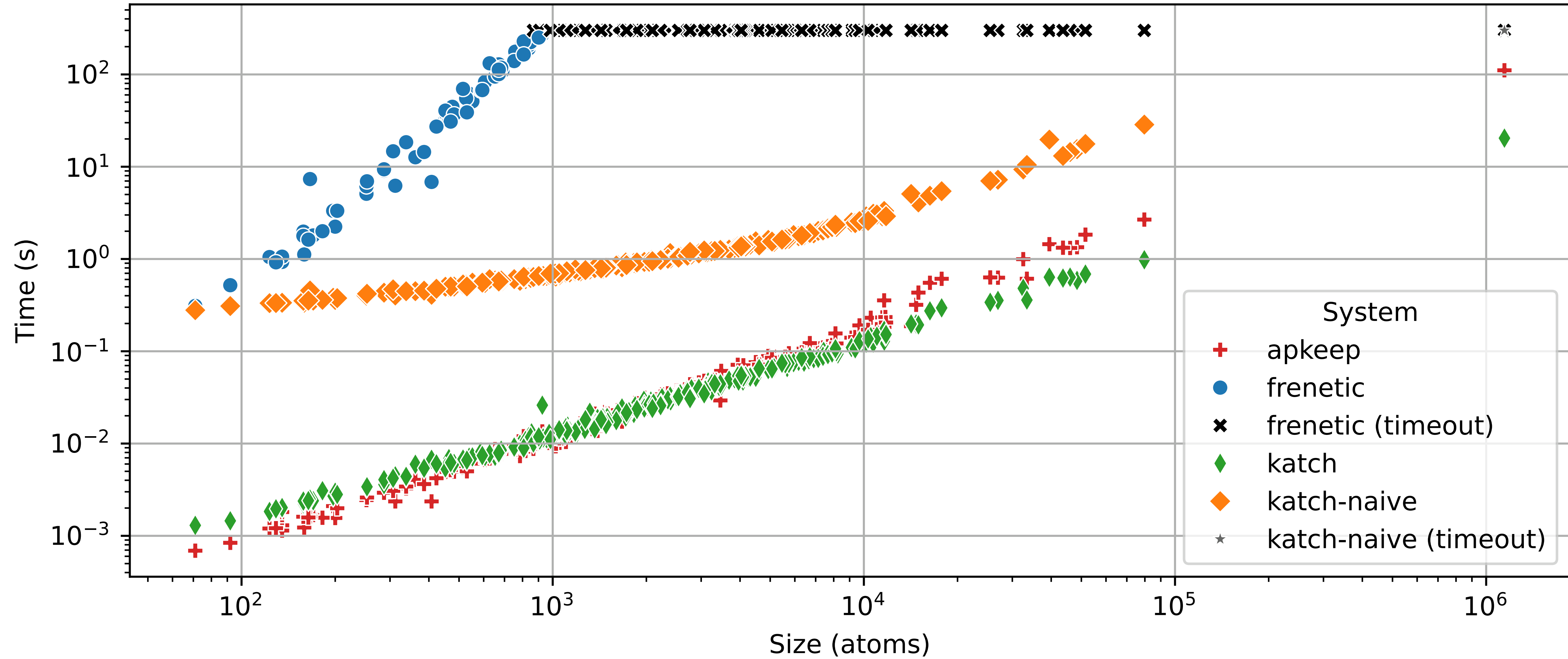
NetKAT (POPL 2015) and **APKeep** (NSDI 2020)



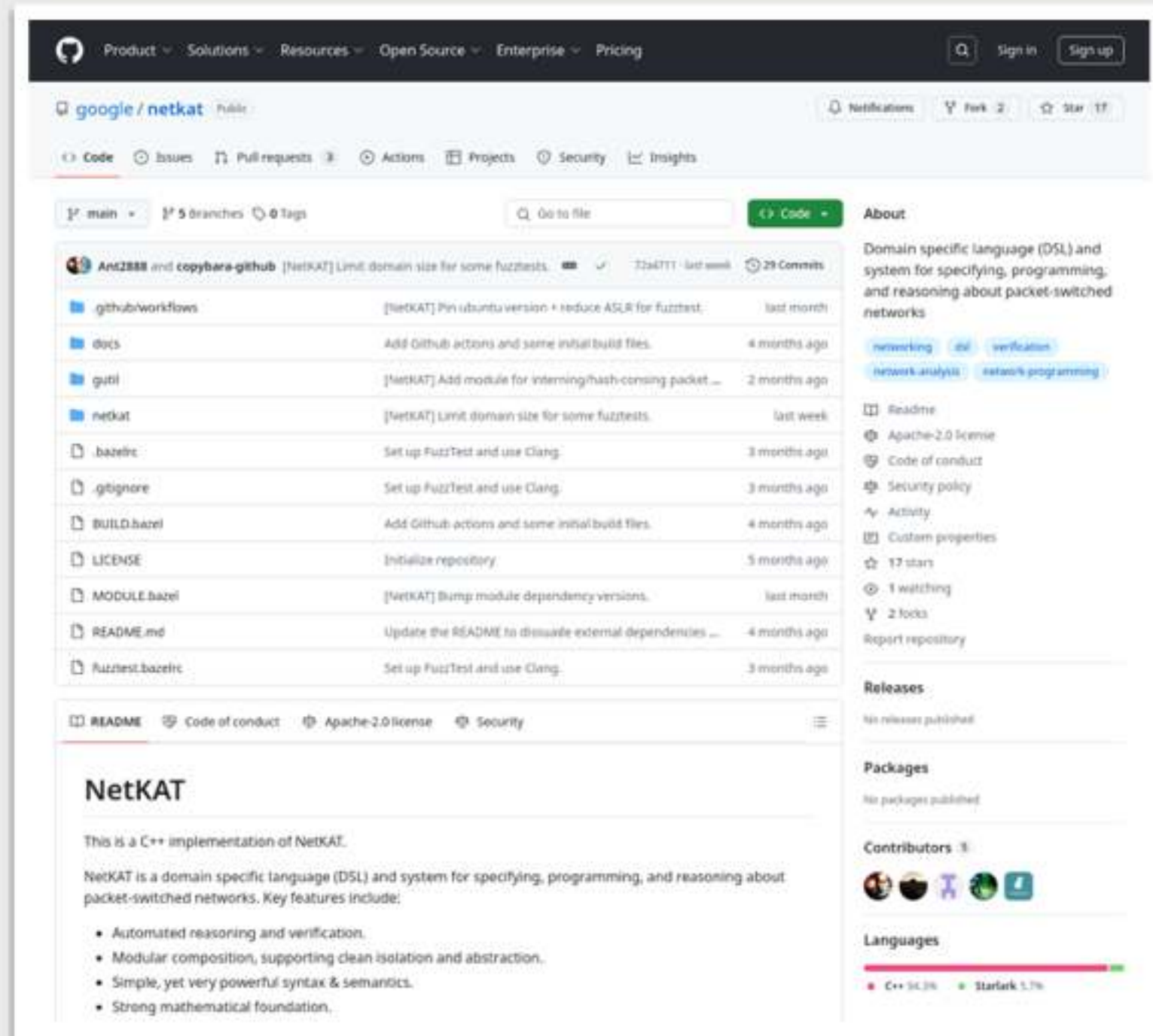
9 years later... NetKAT v. PLDI'24



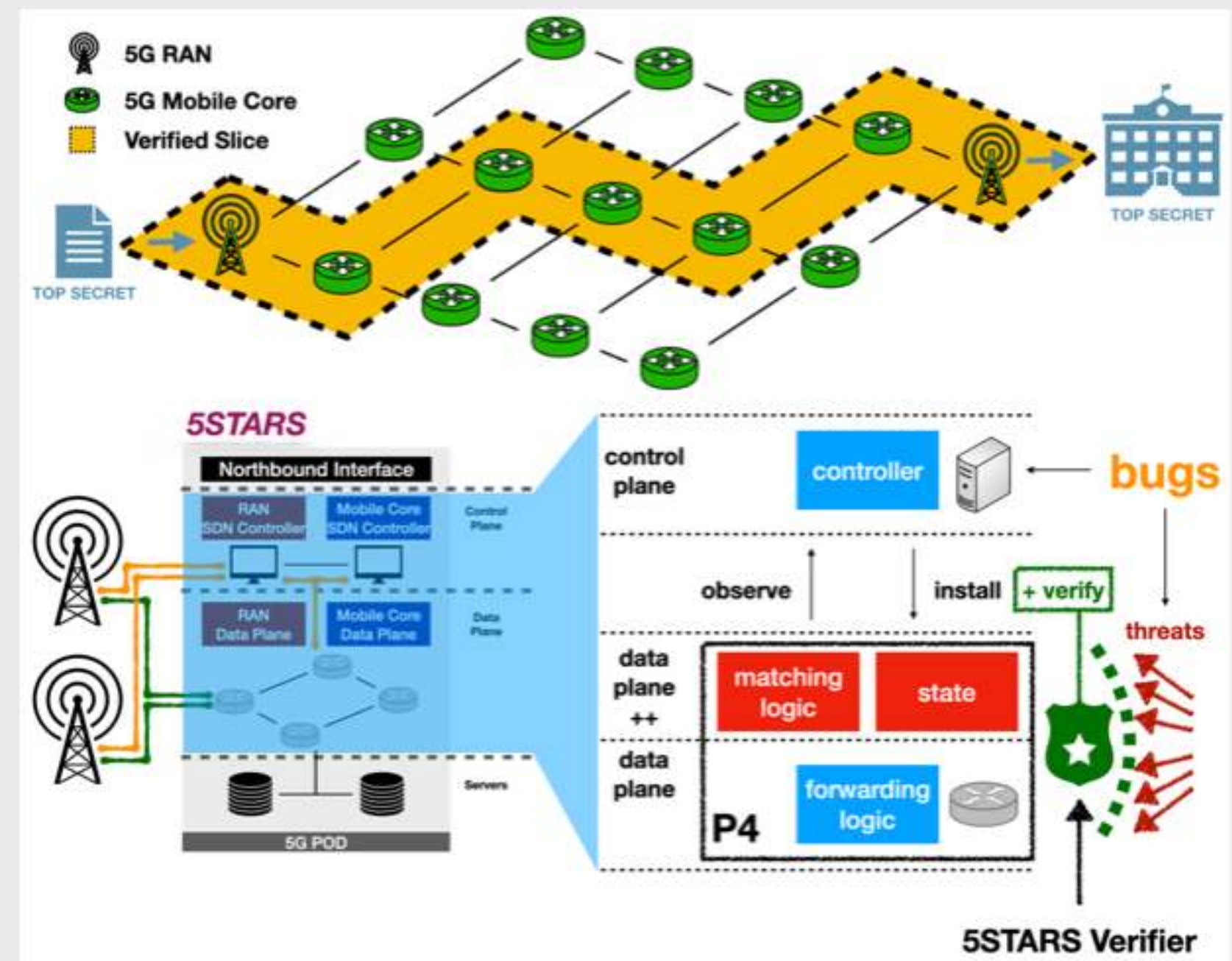
Full reachability



NetKAT in Industry



C++ implementation of NetKAT for verifying cloud isolation



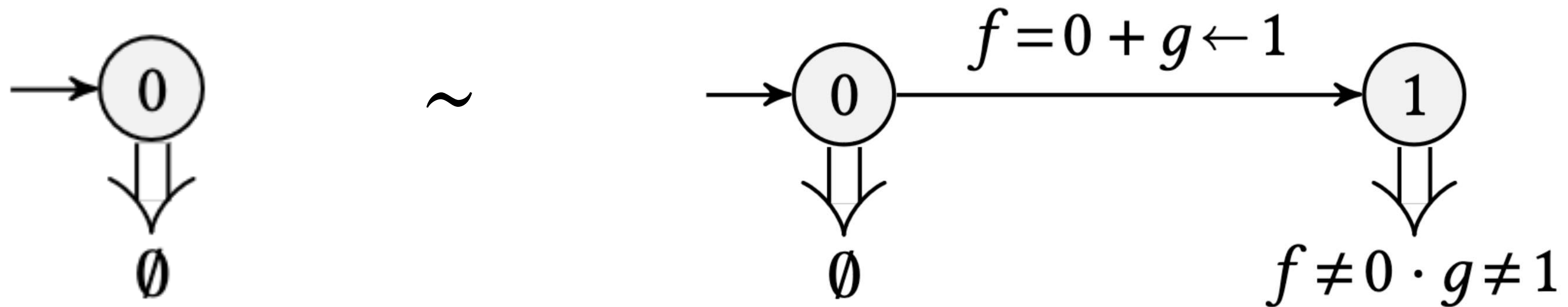
Haskell implementation of NetKAT for verifying secure 5G slicing

What's hard about NetKAT equivalence?

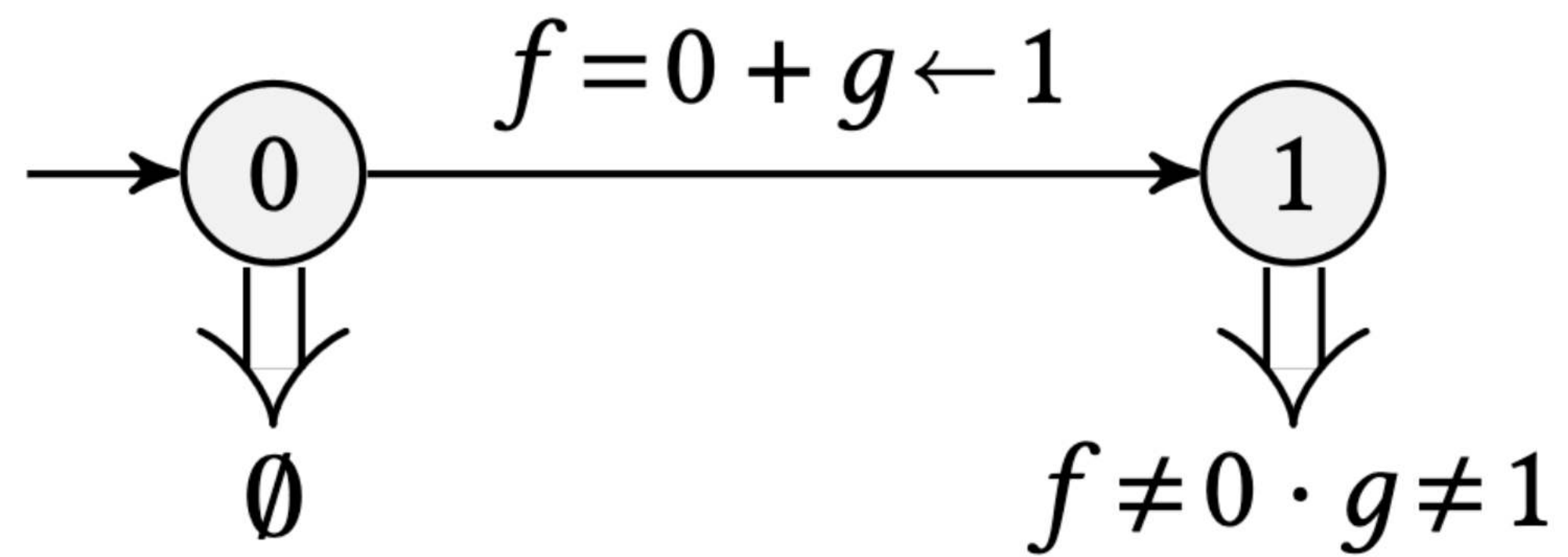
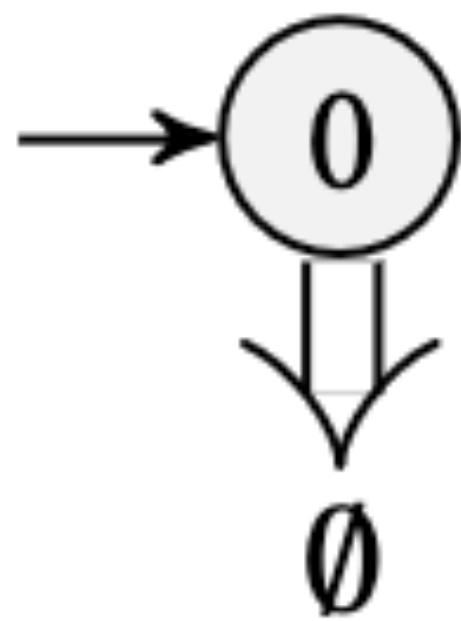
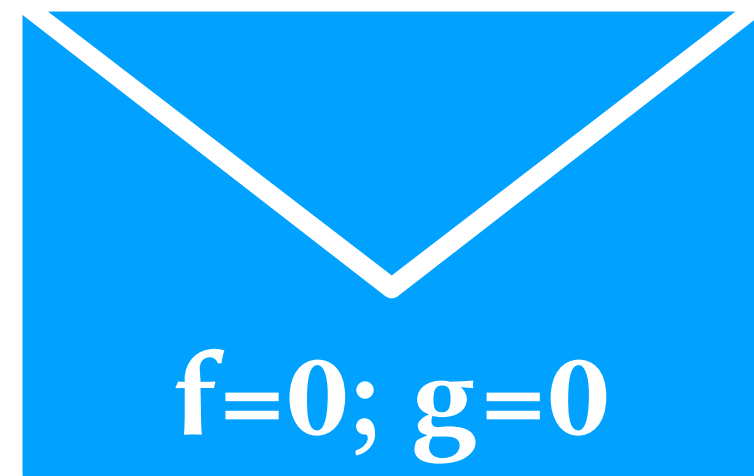
$$\delta : S \times Pk \rightarrow S^{Pk}$$

$$\epsilon : S \times Pk \rightarrow 2^{Pk}$$

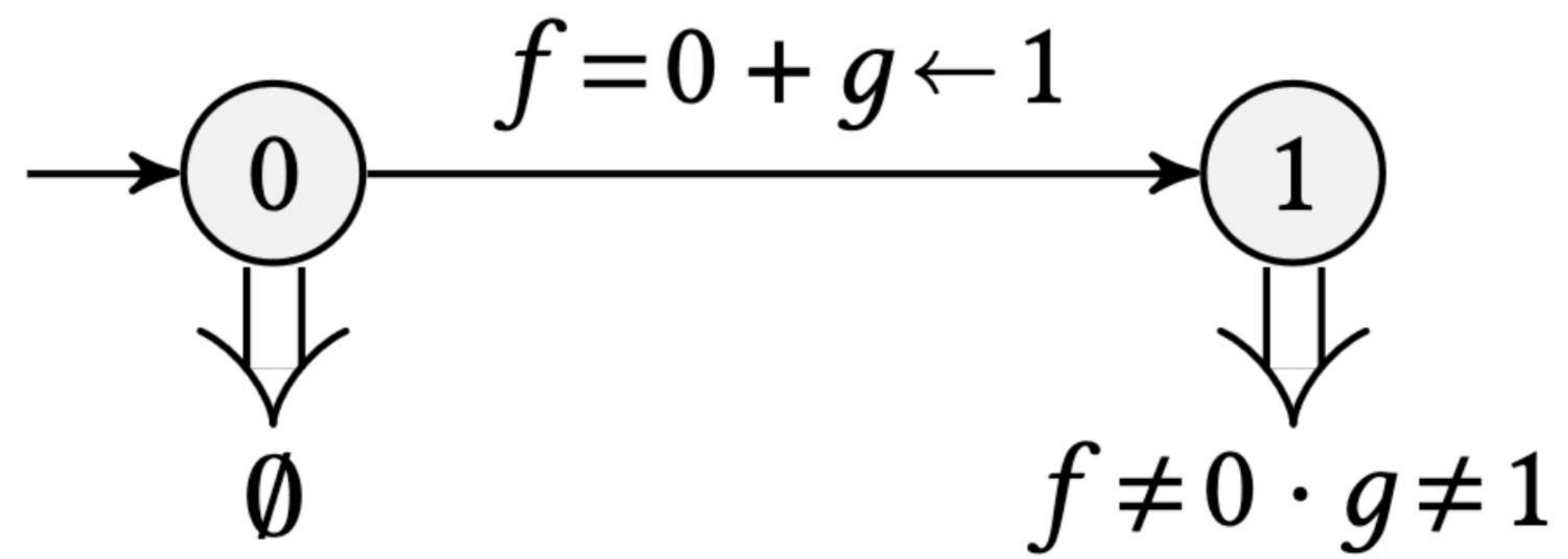
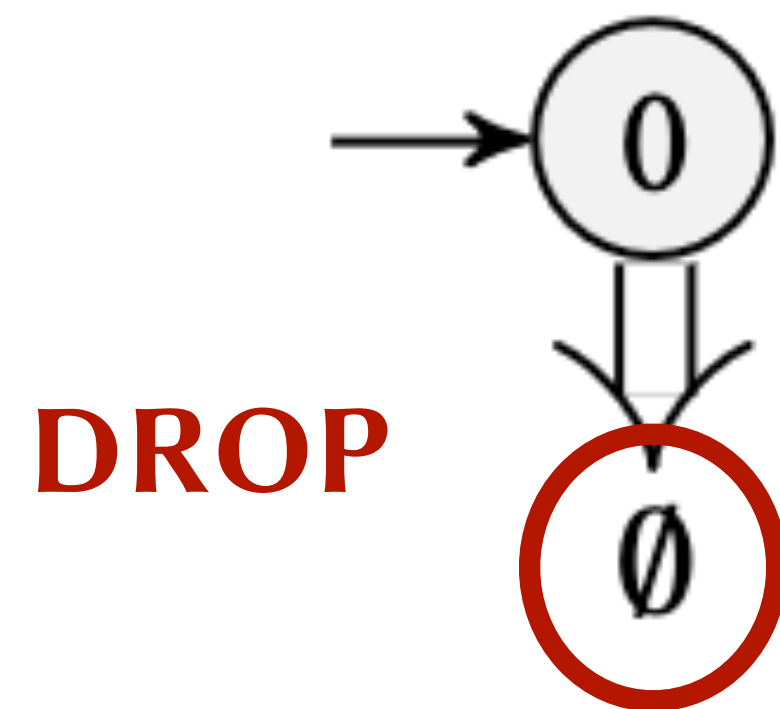
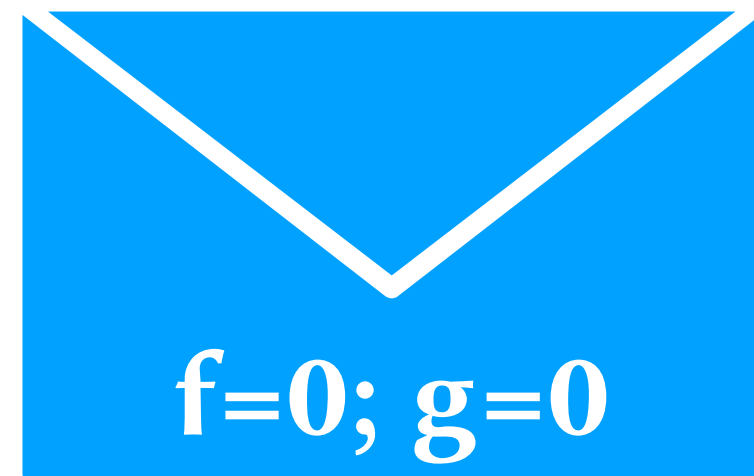
Pk is typically a VERY large set = Fields x Values



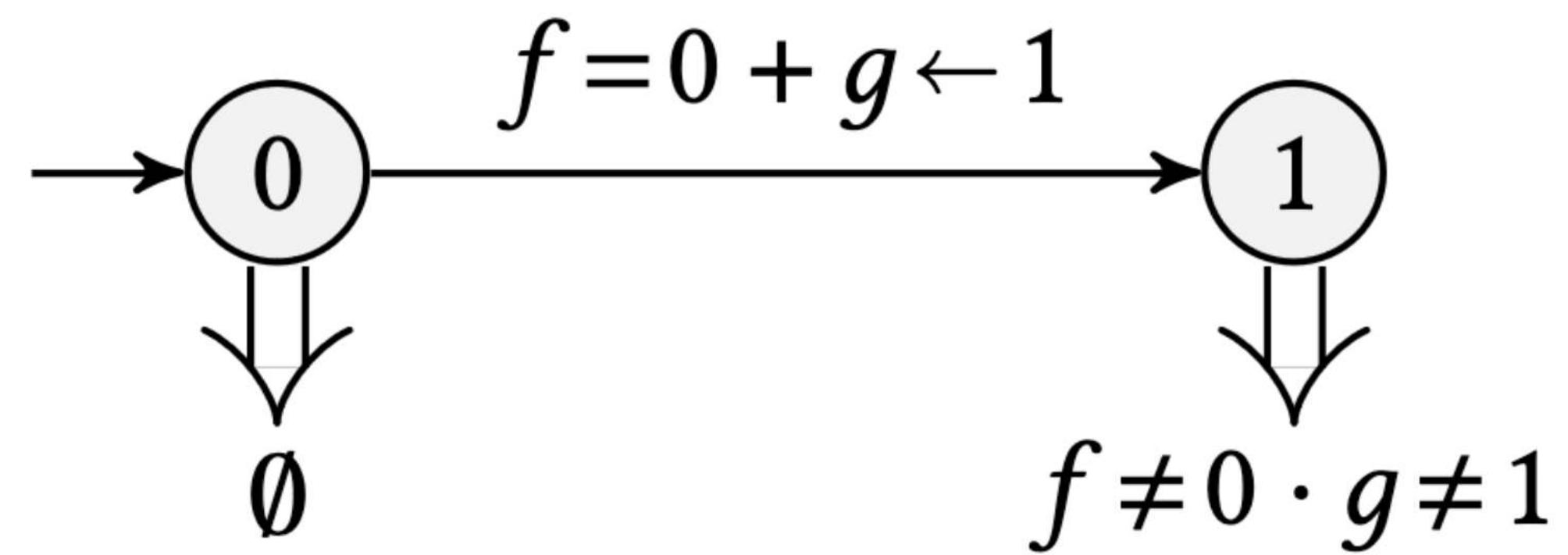
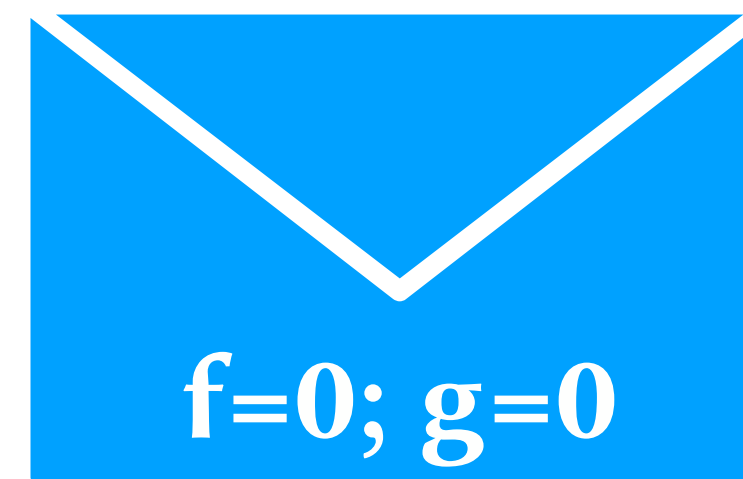
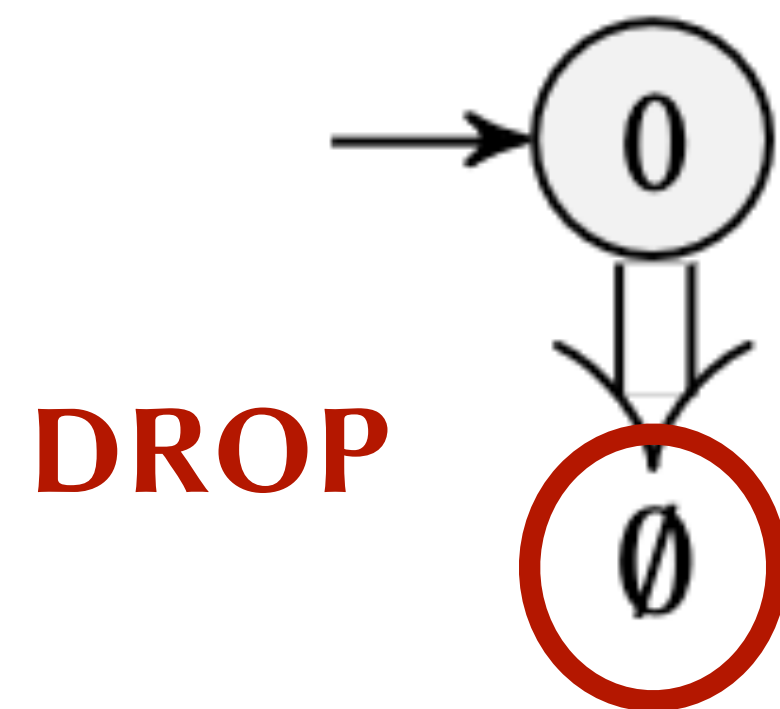
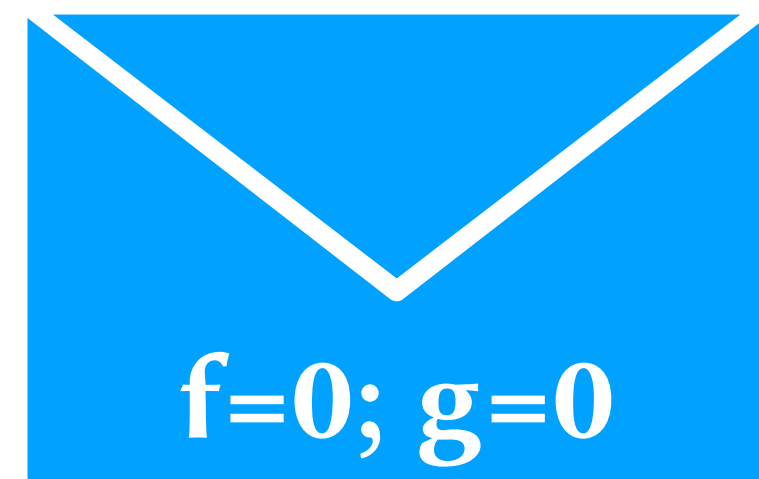
Checking Equivalence in NetKAT



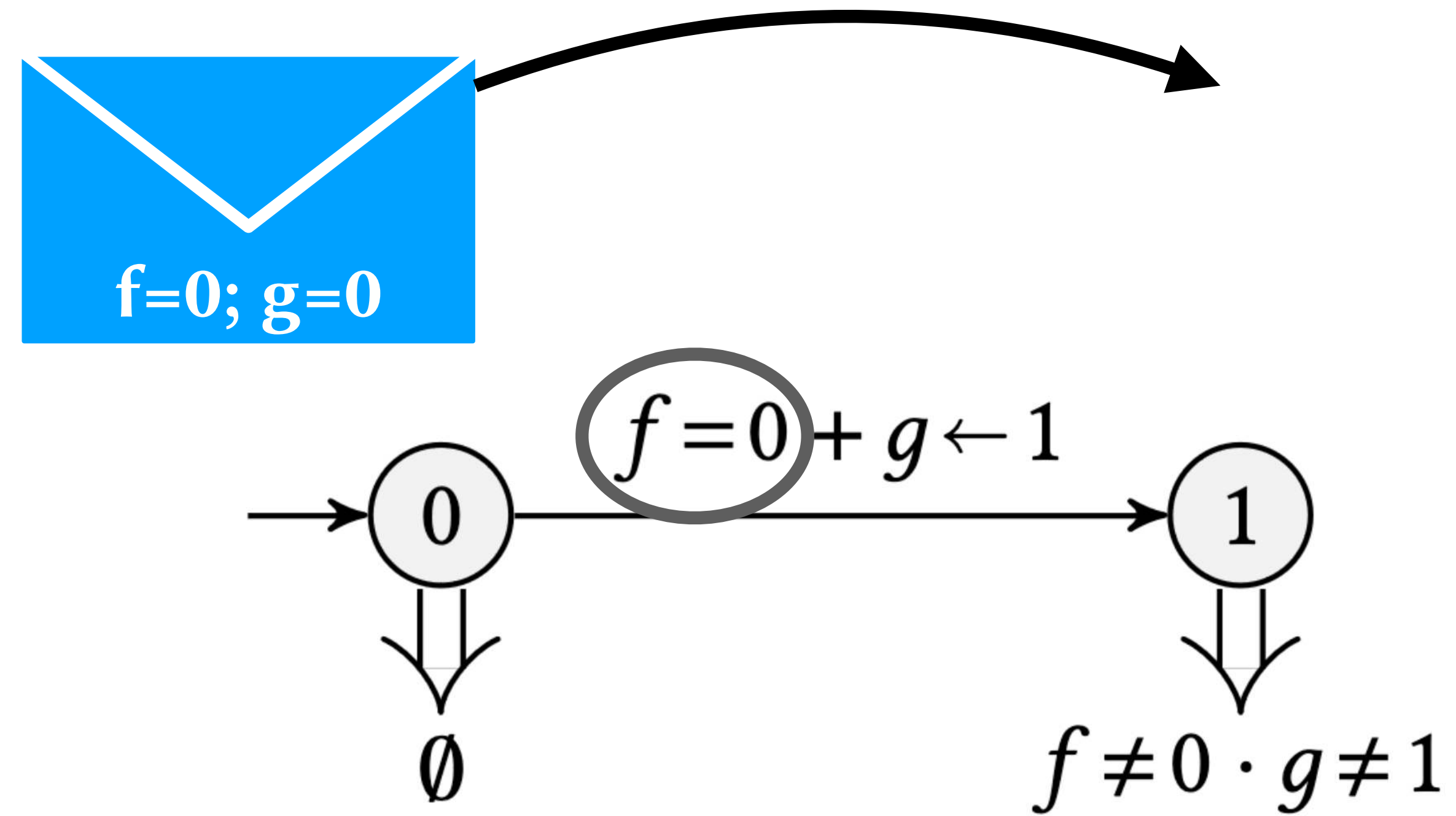
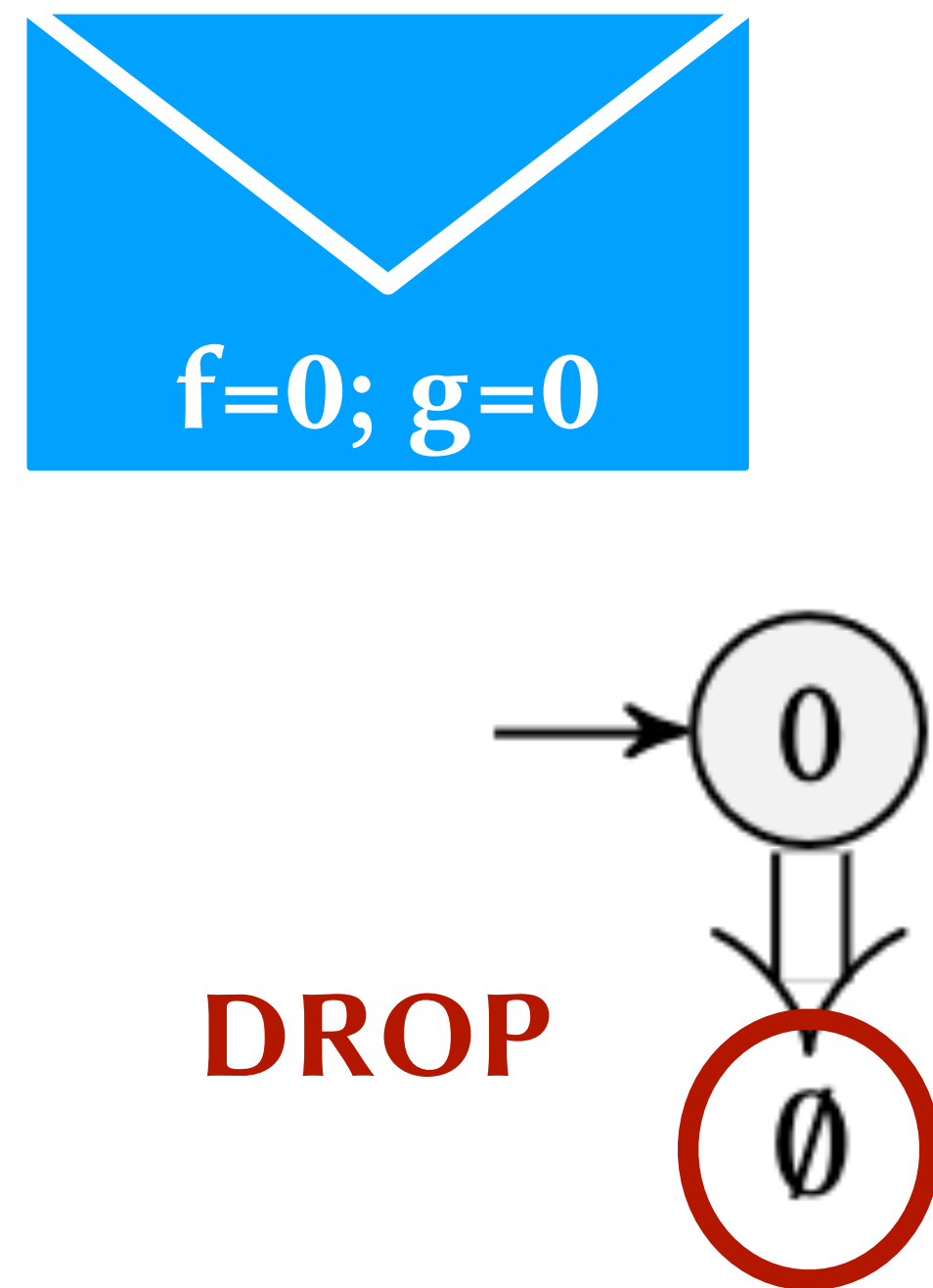
Checking Equivalence in NetKAT



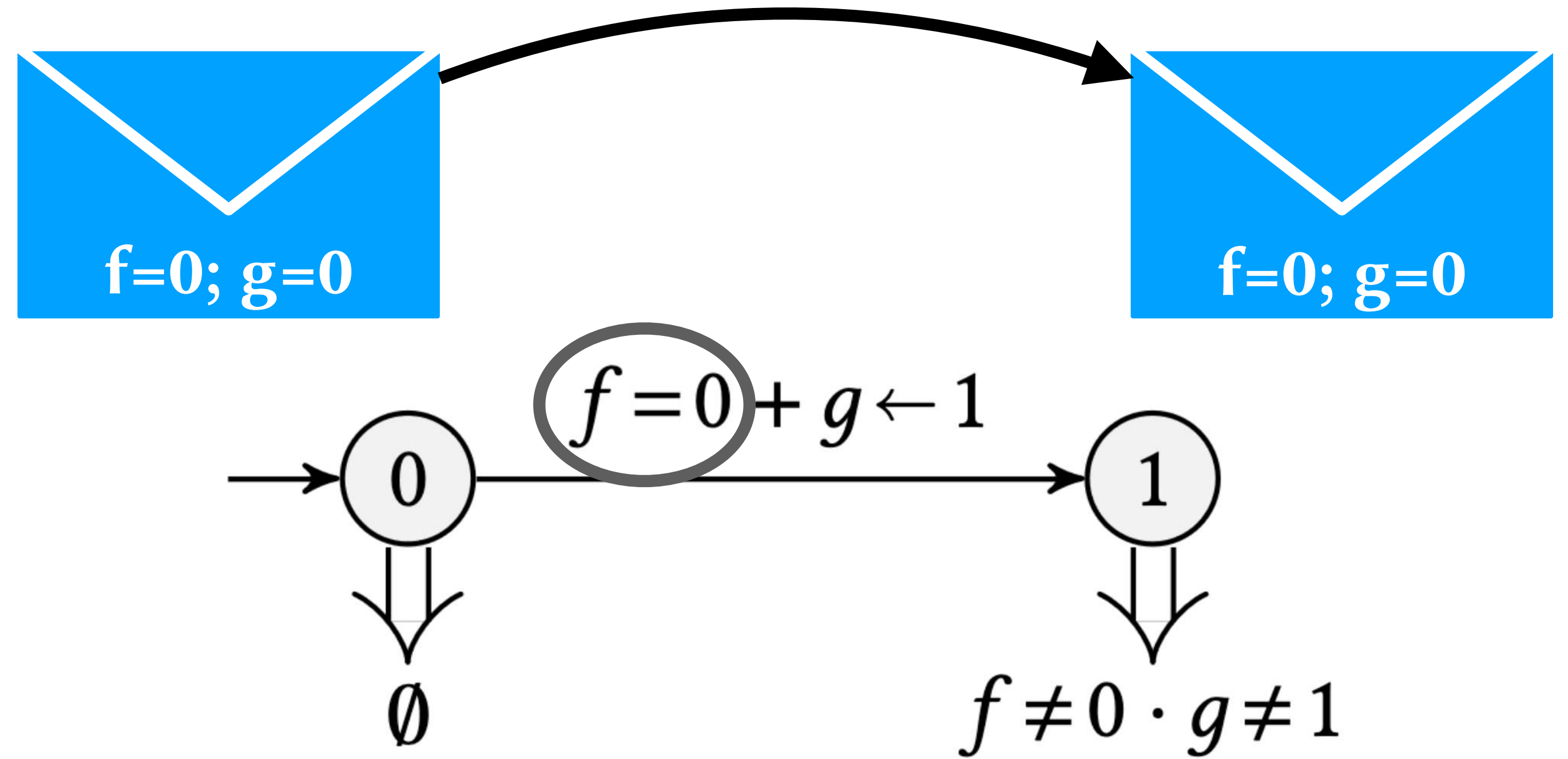
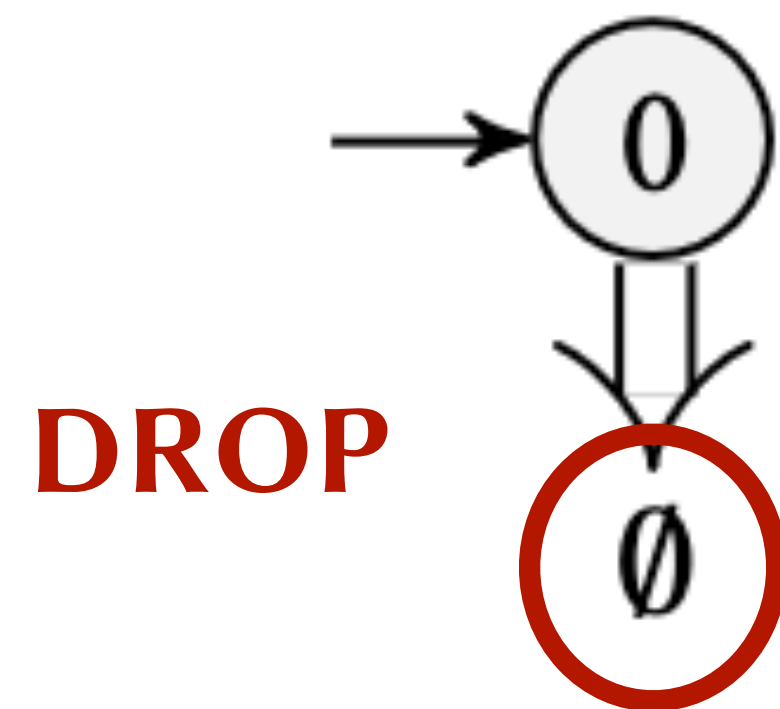
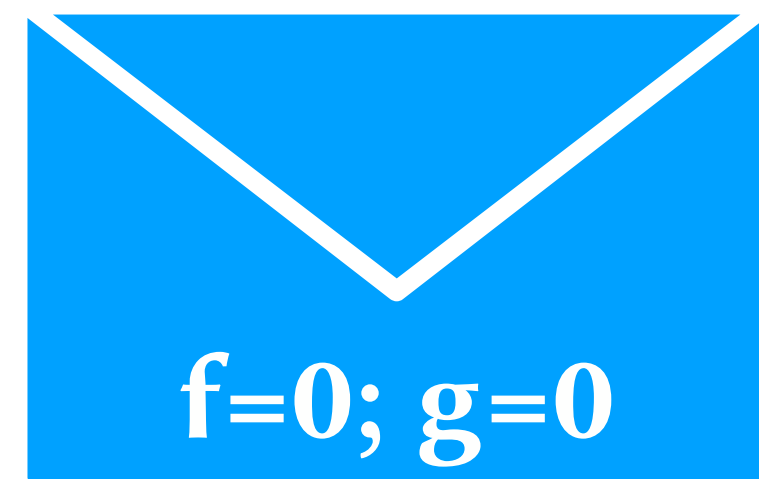
Checking Equivalence in NetKAT



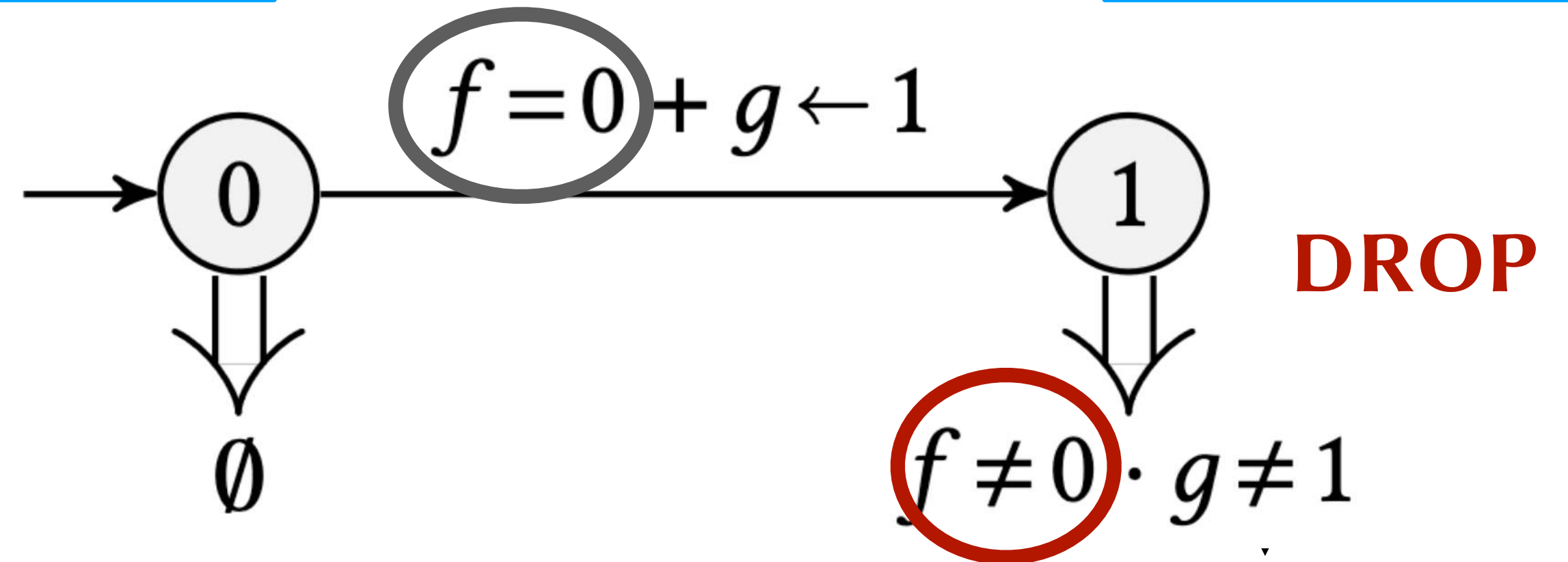
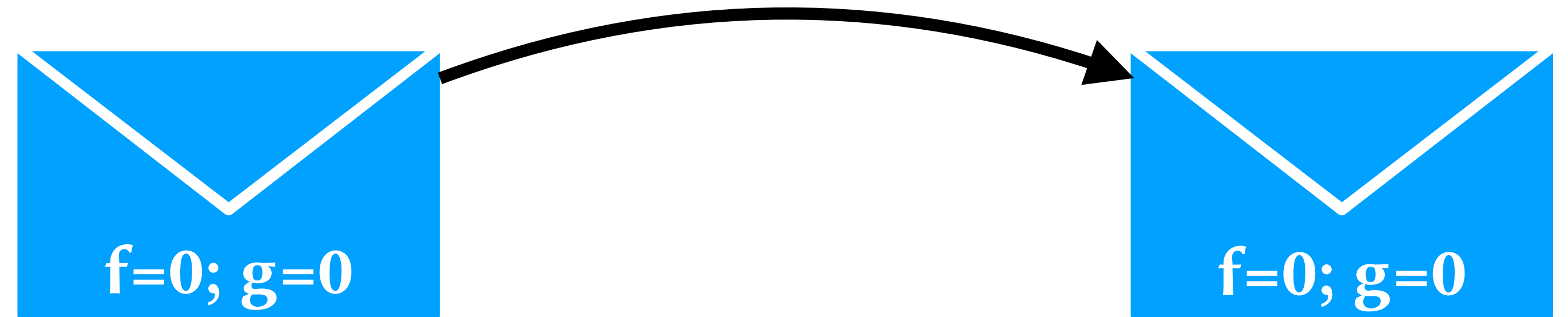
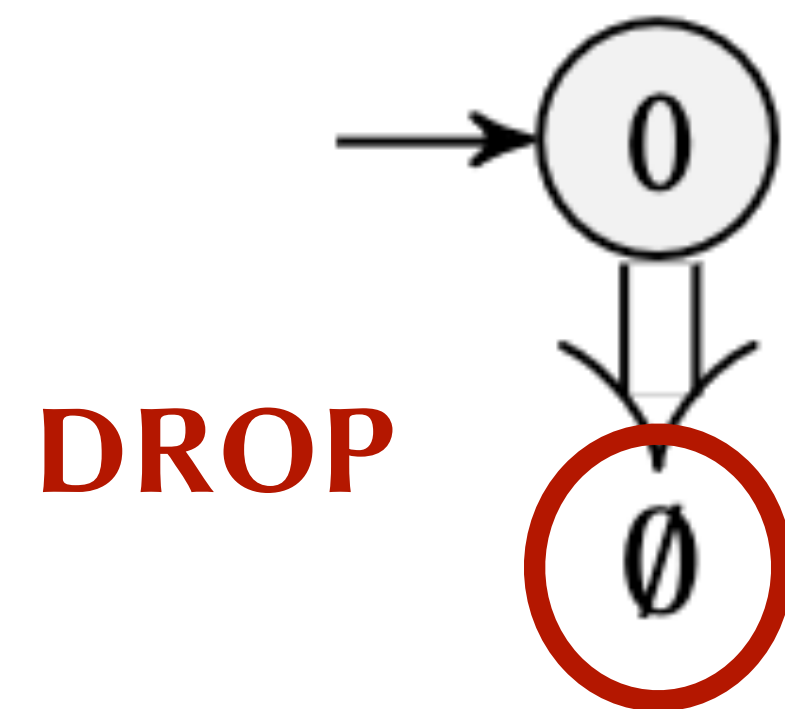
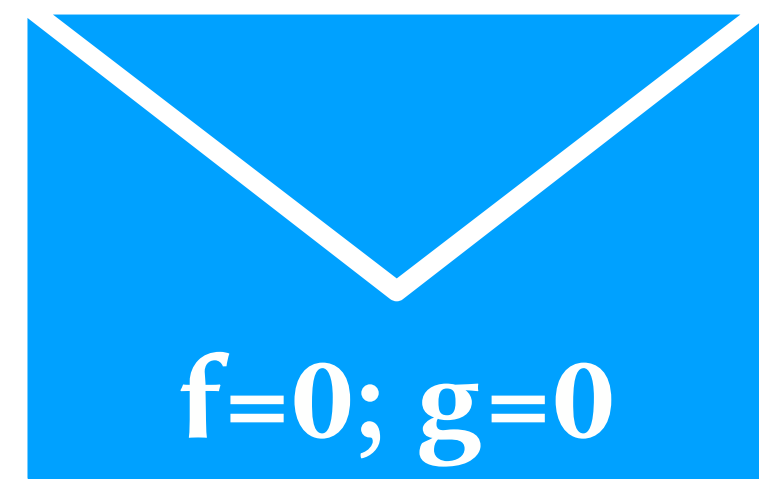
Checking Equivalence in NetKAT



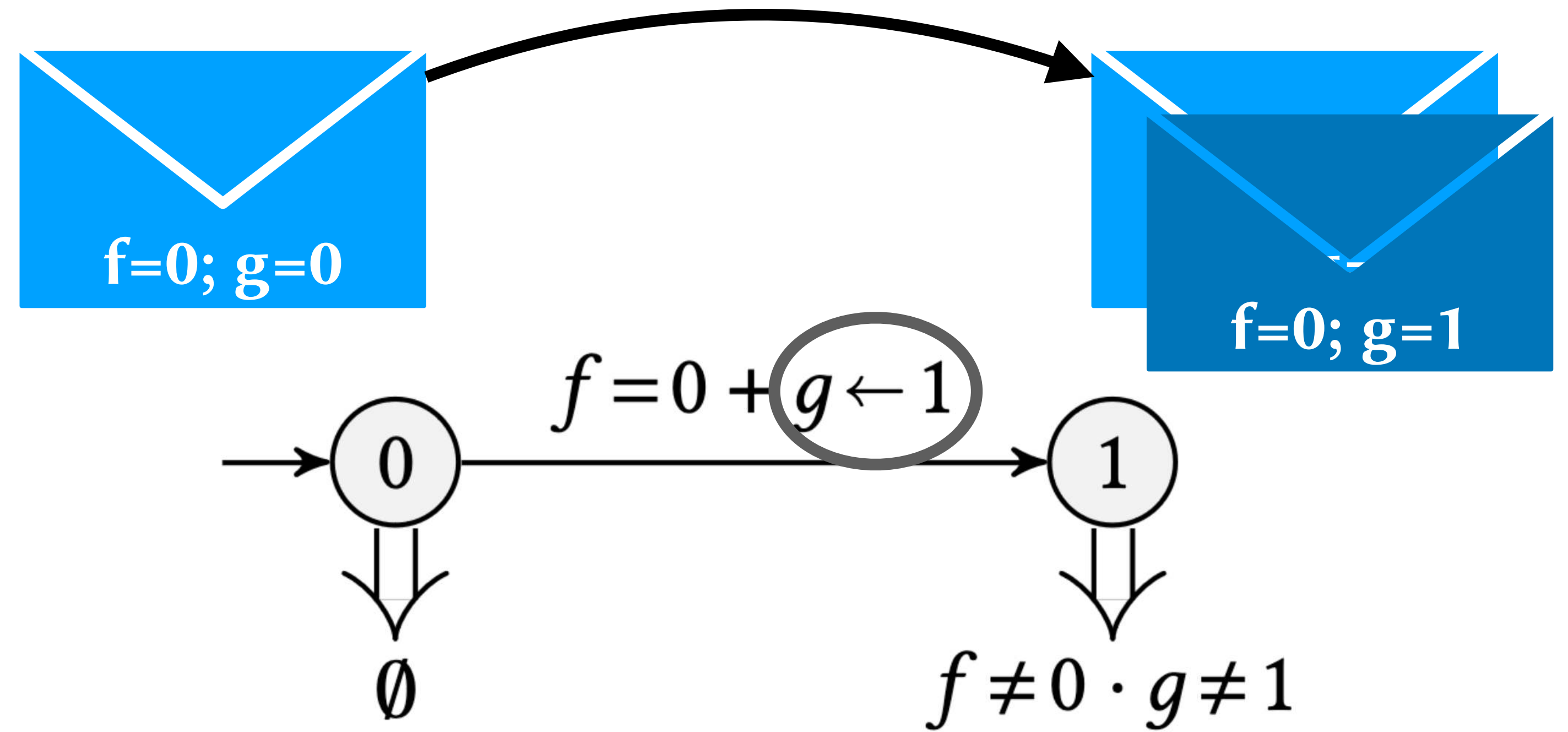
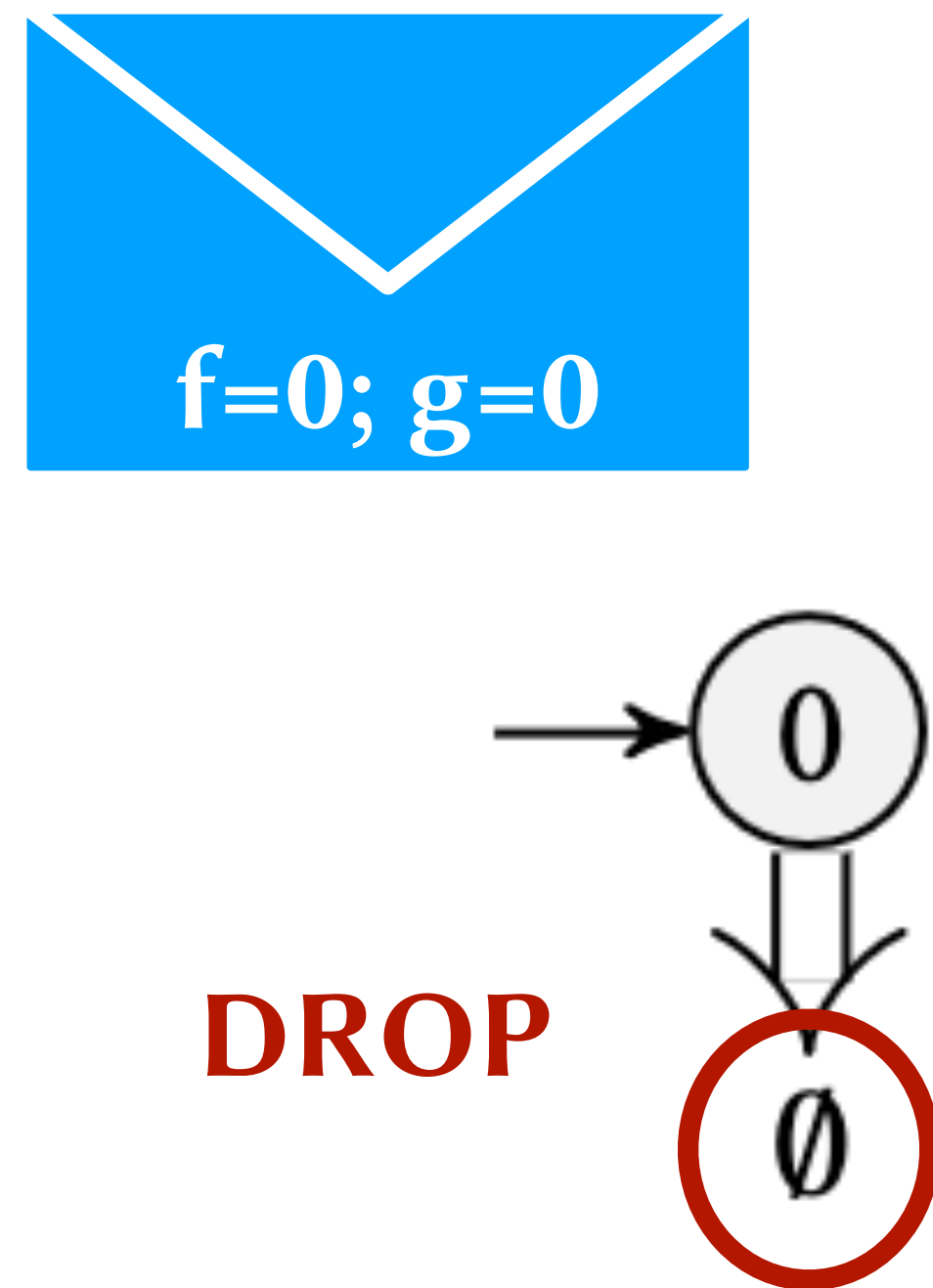
Checking Equivalence in NetKAT



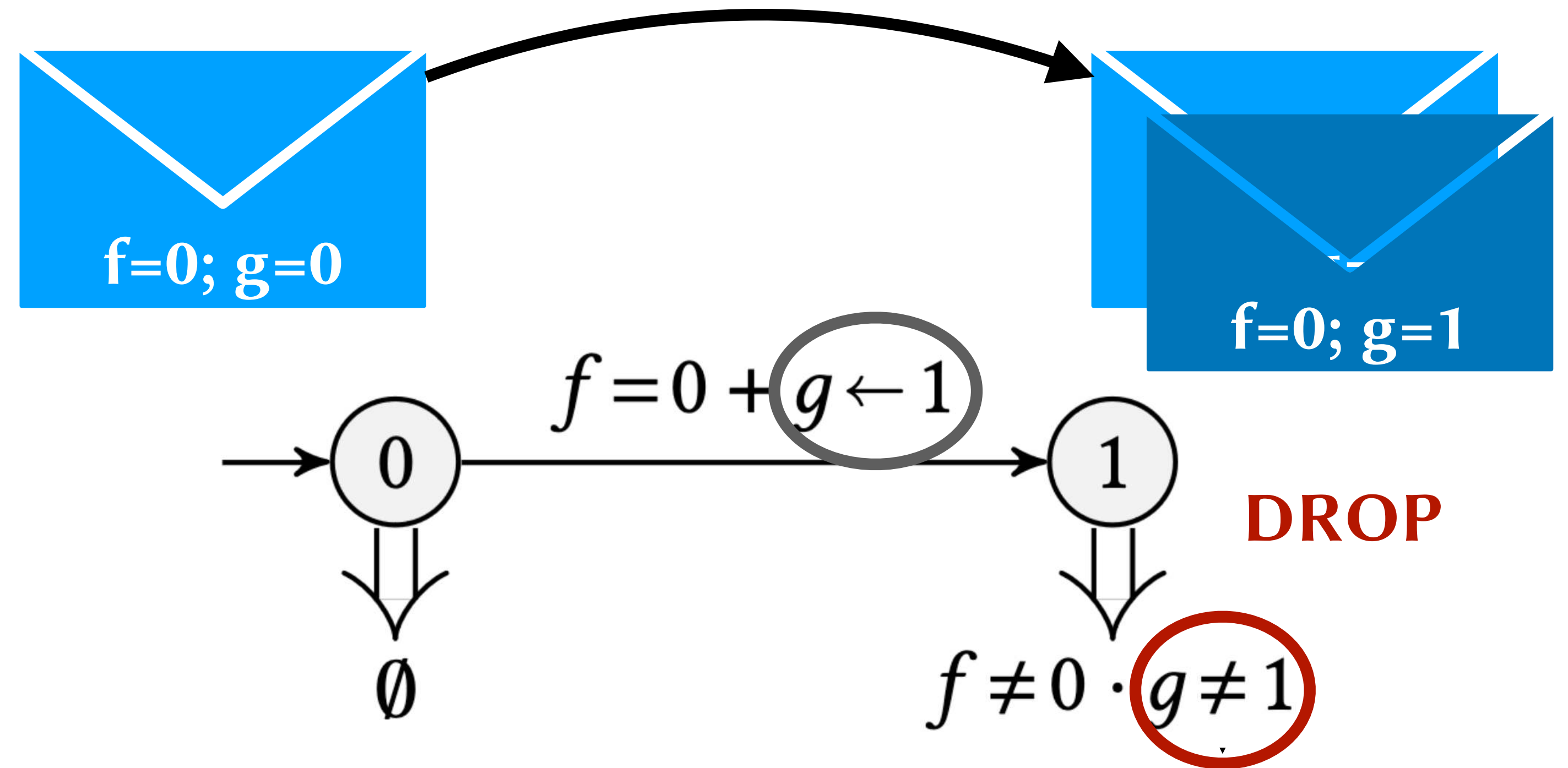
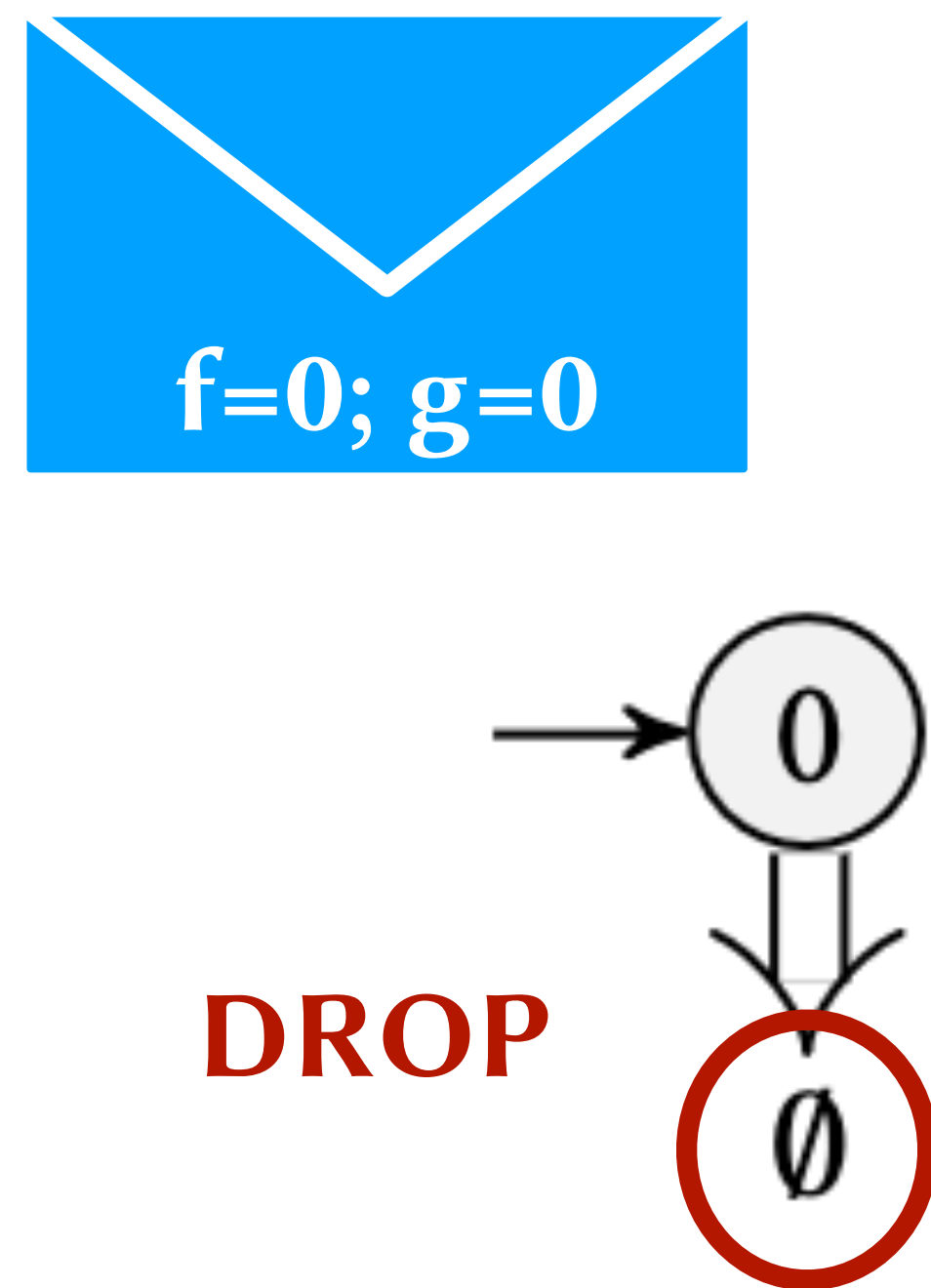
Checking Equivalence in NetKAT



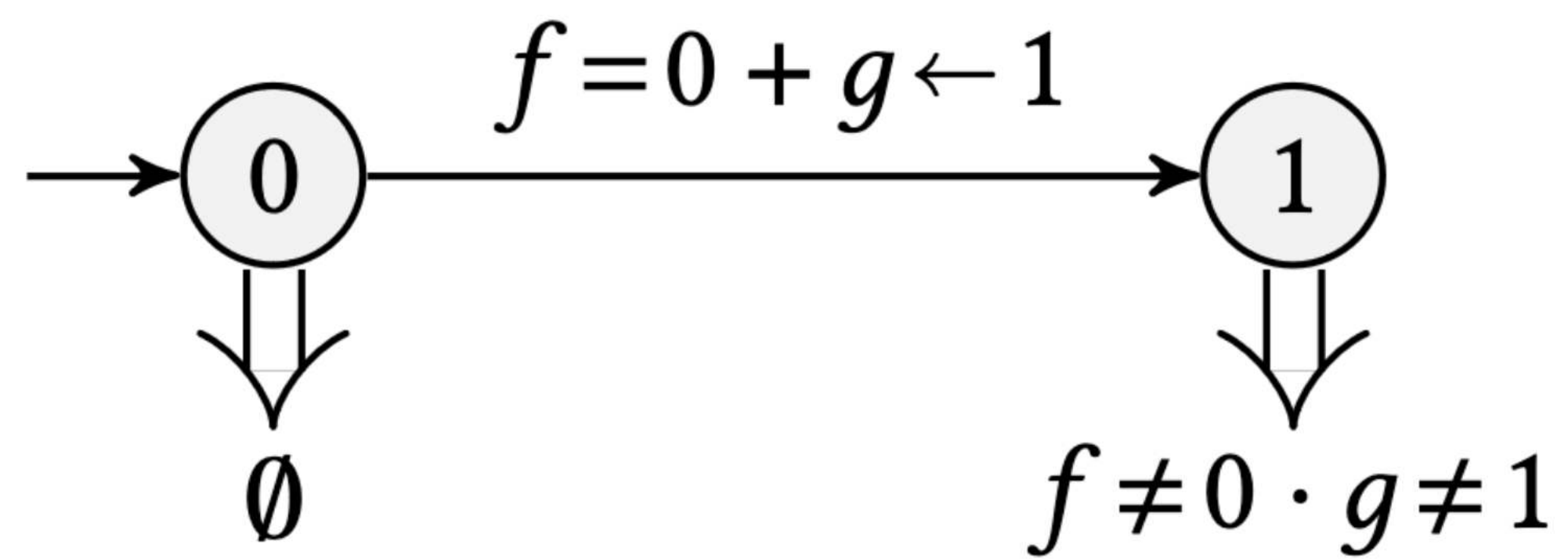
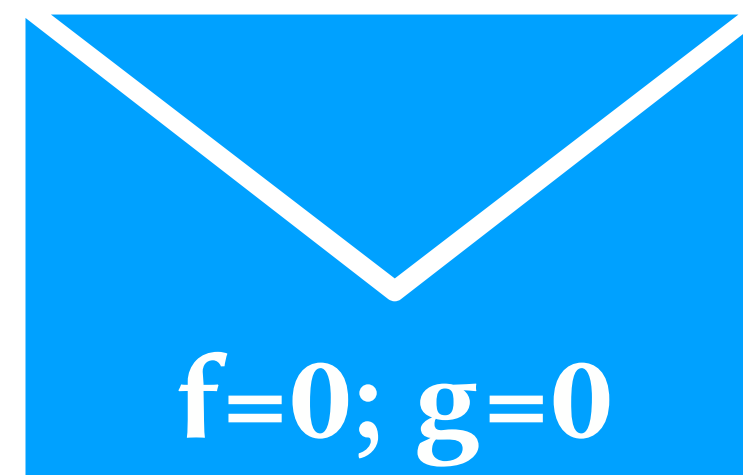
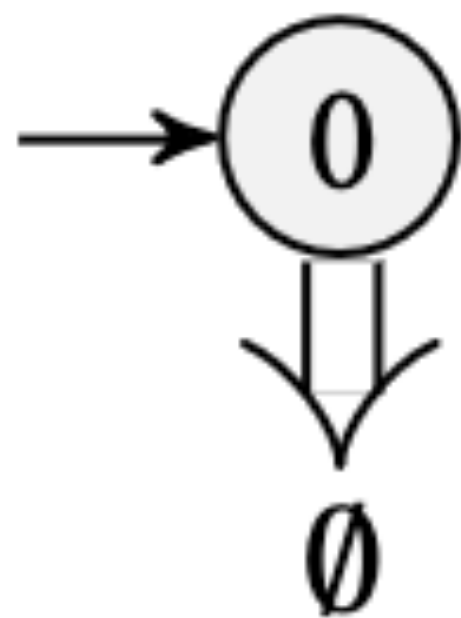
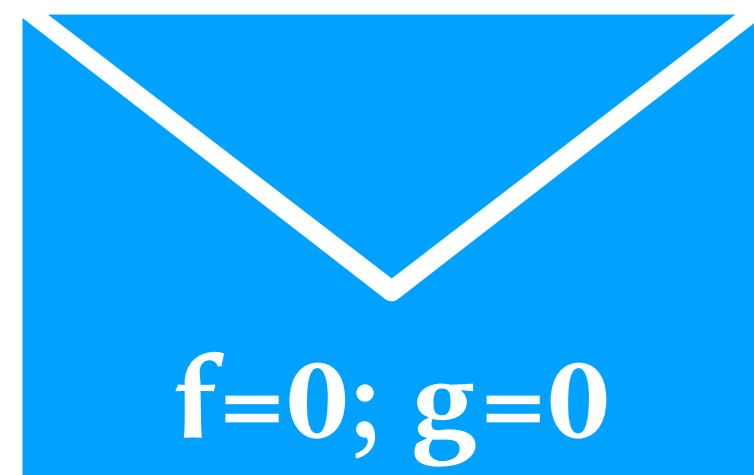
Checking Equivalence in NetKAT



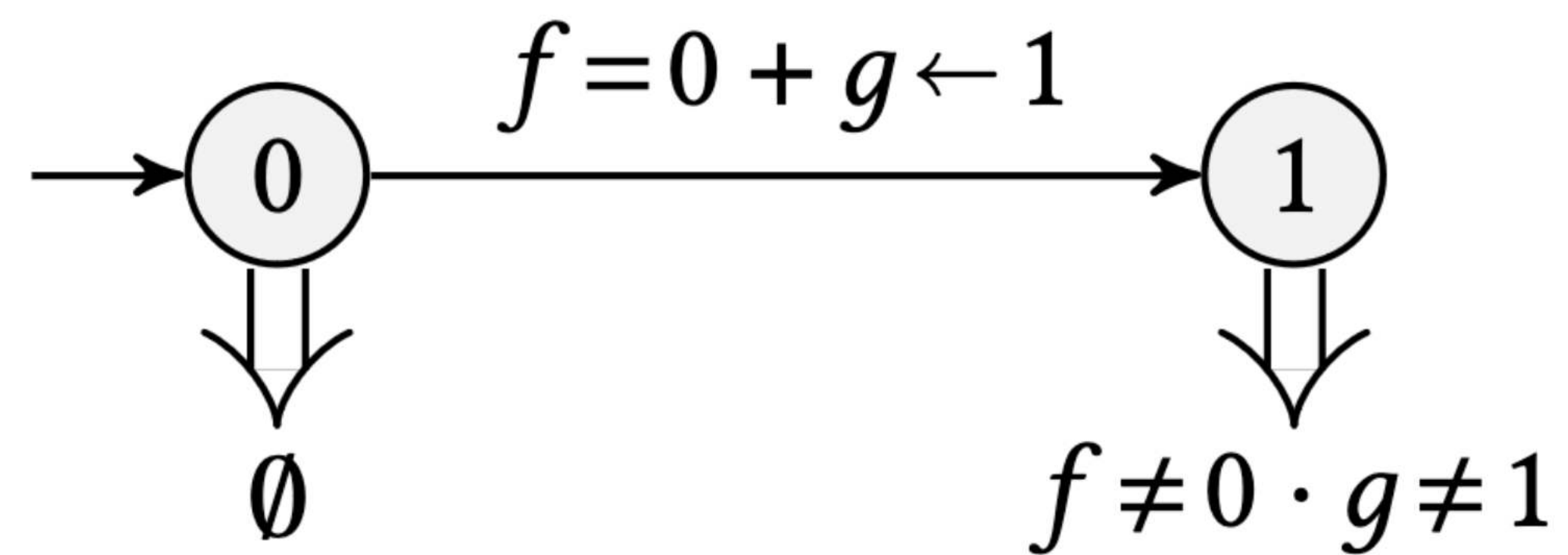
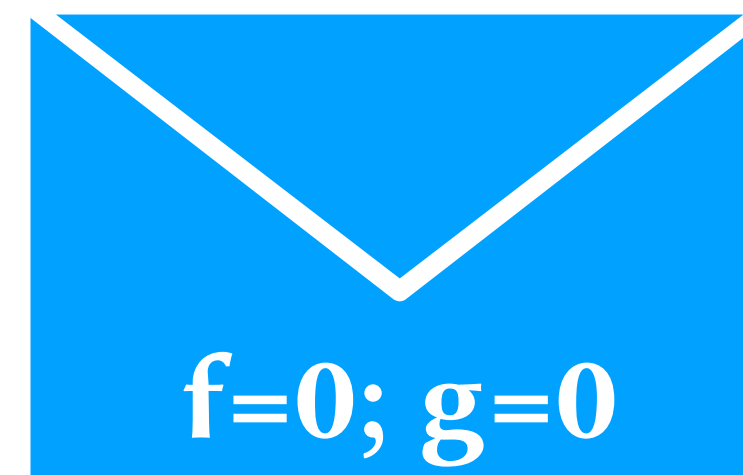
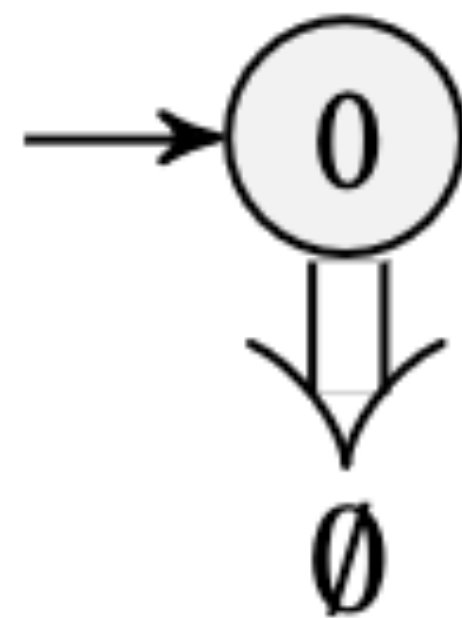
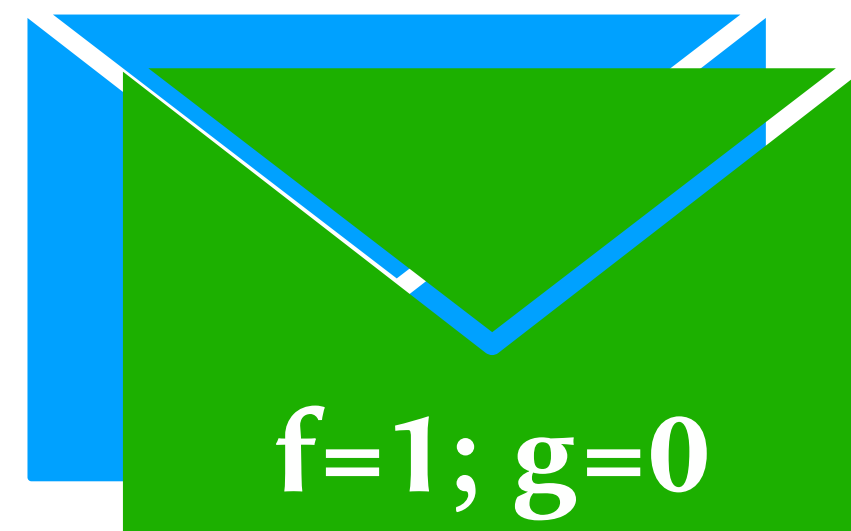
Checking Equivalence in NetKAT



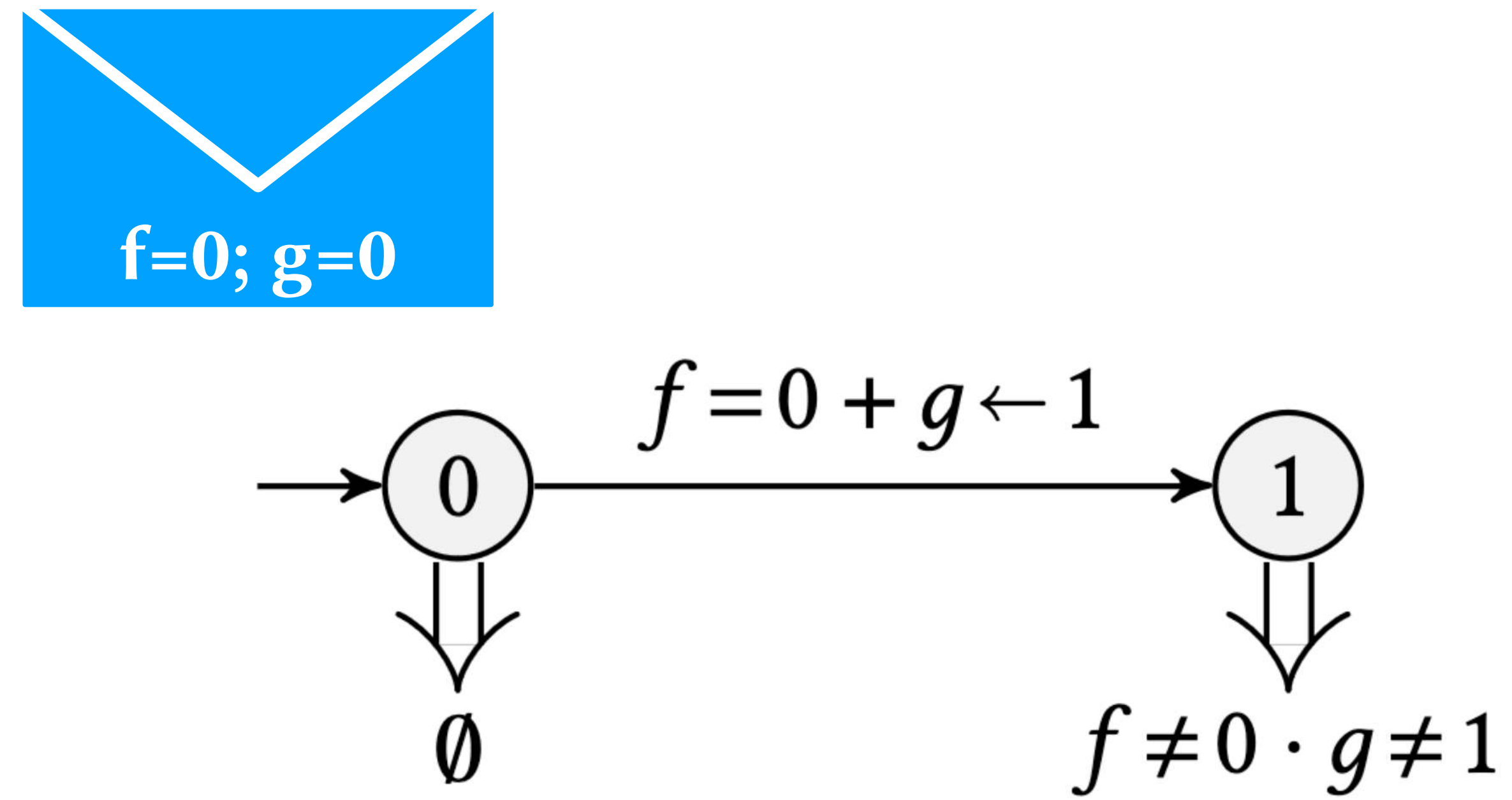
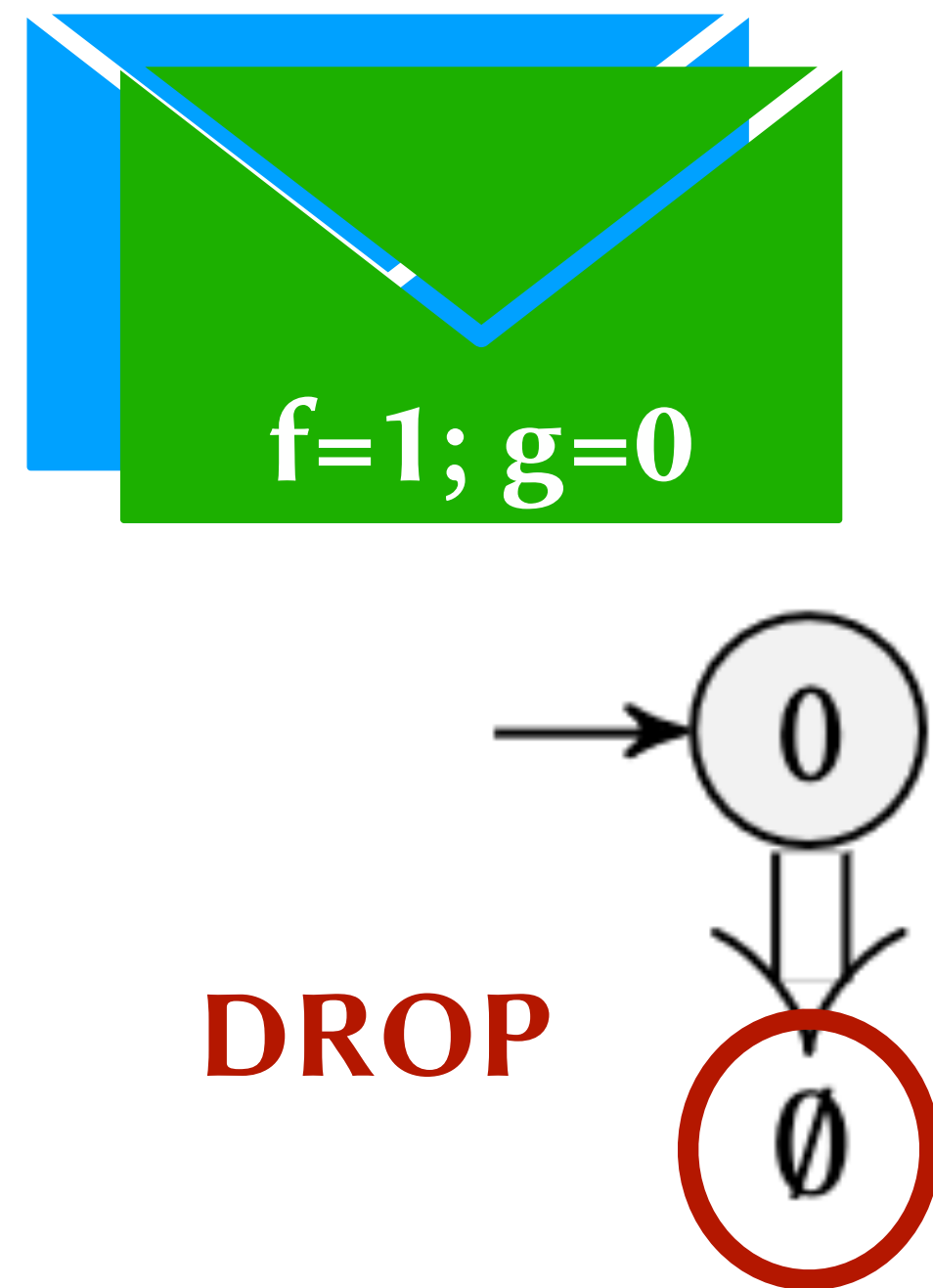
Checking Equivalence in NetKAT



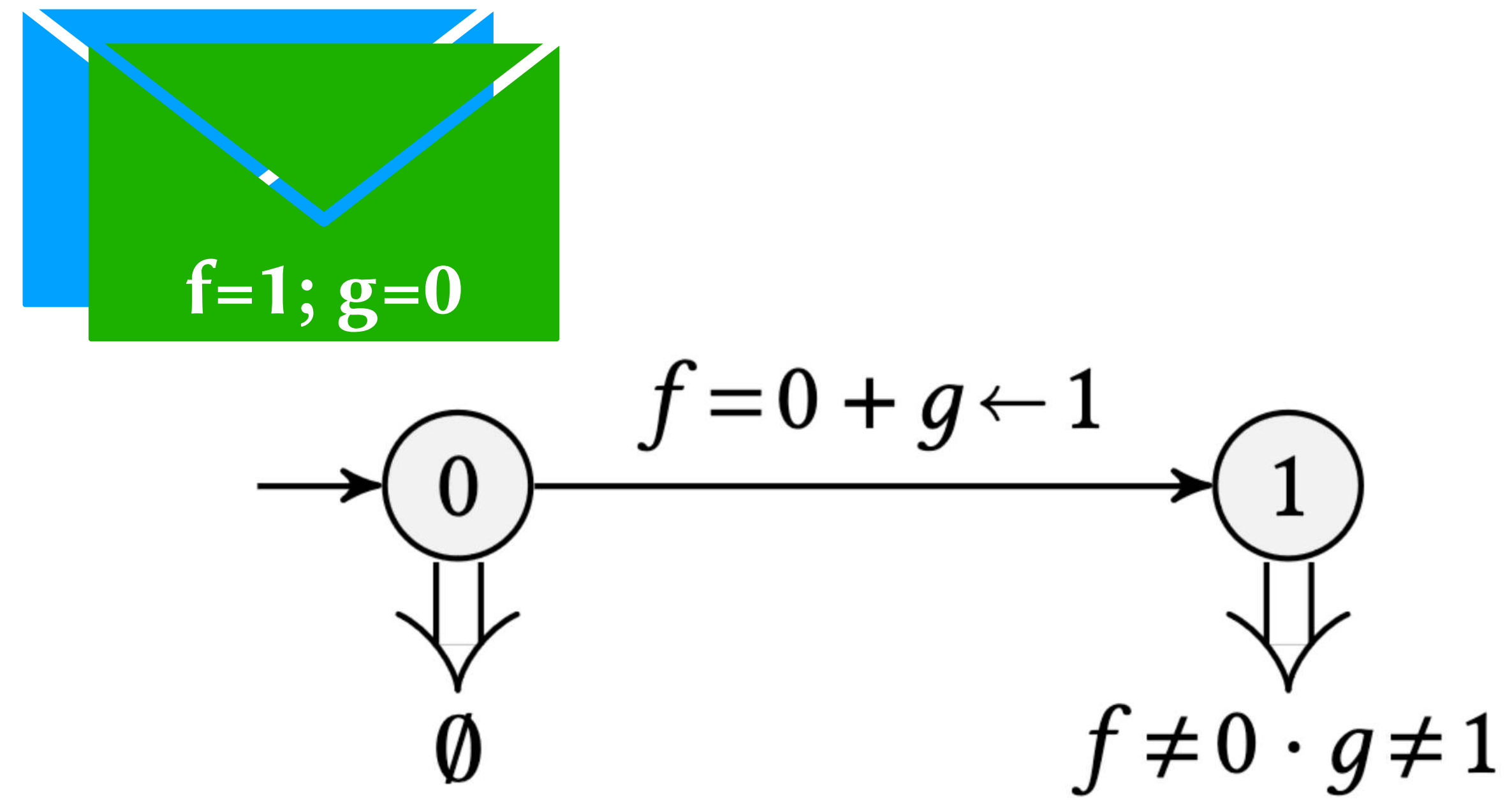
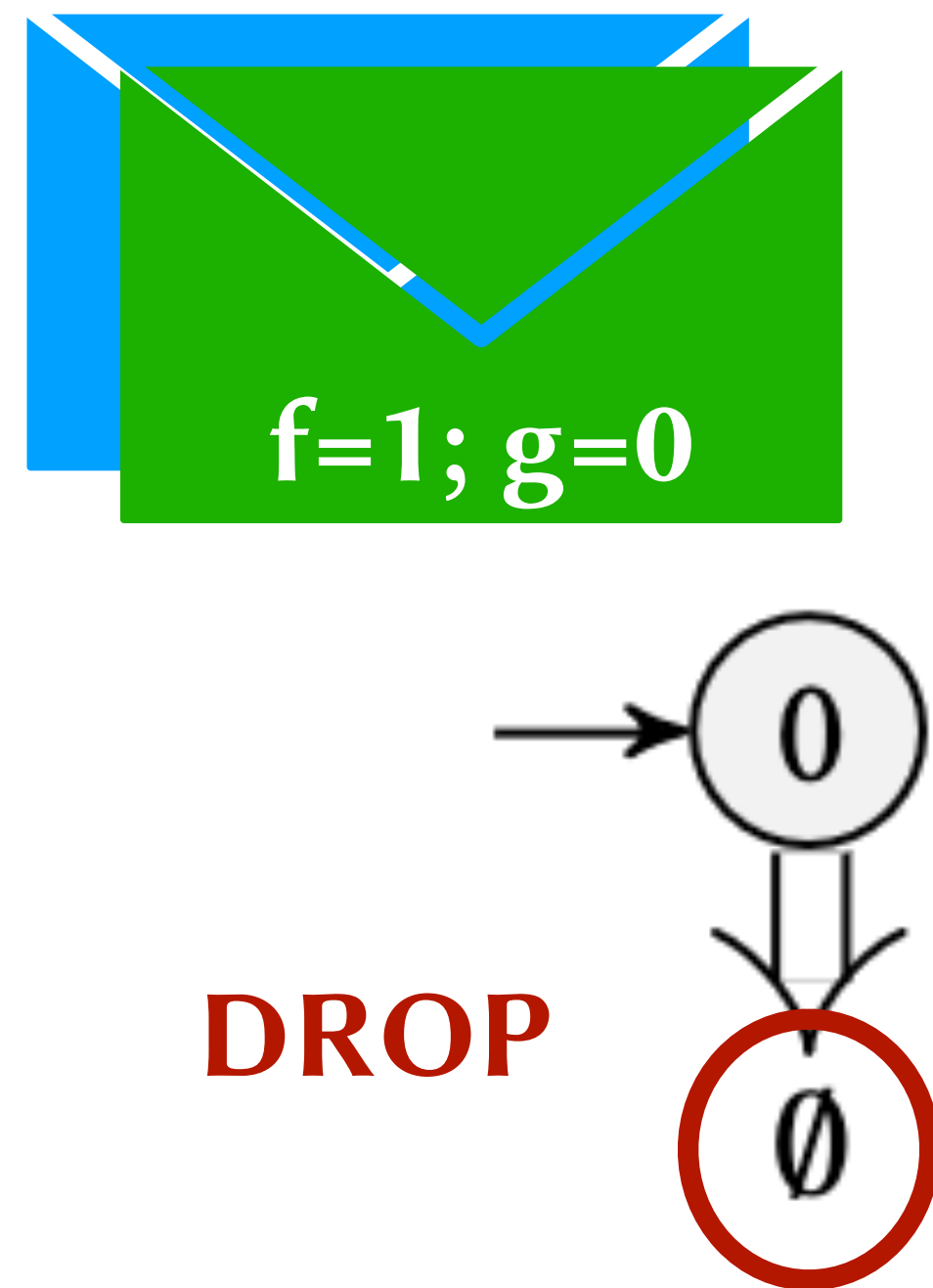
Checking Equivalence in NetKAT



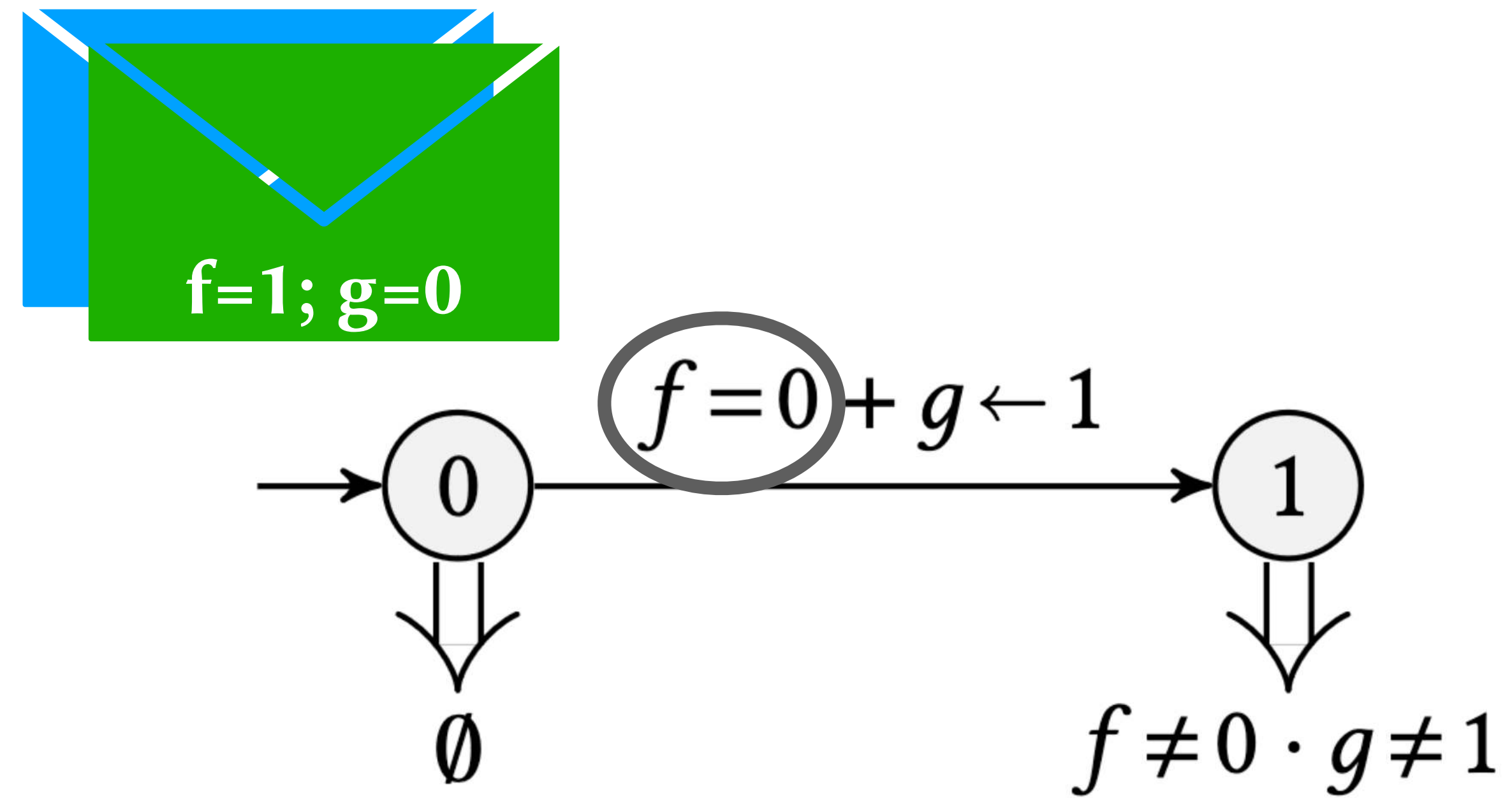
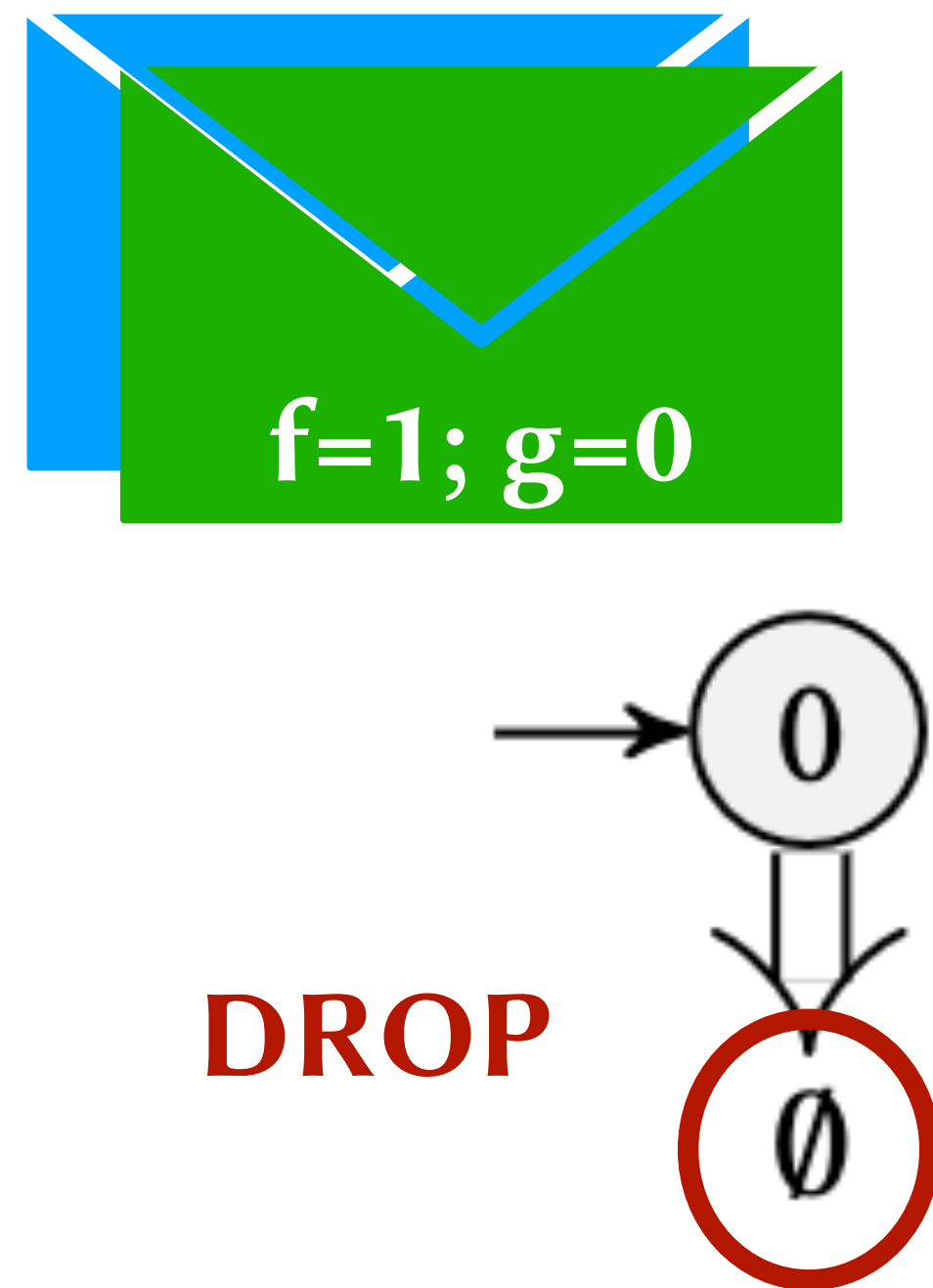
Checking Equivalence in NetKAT



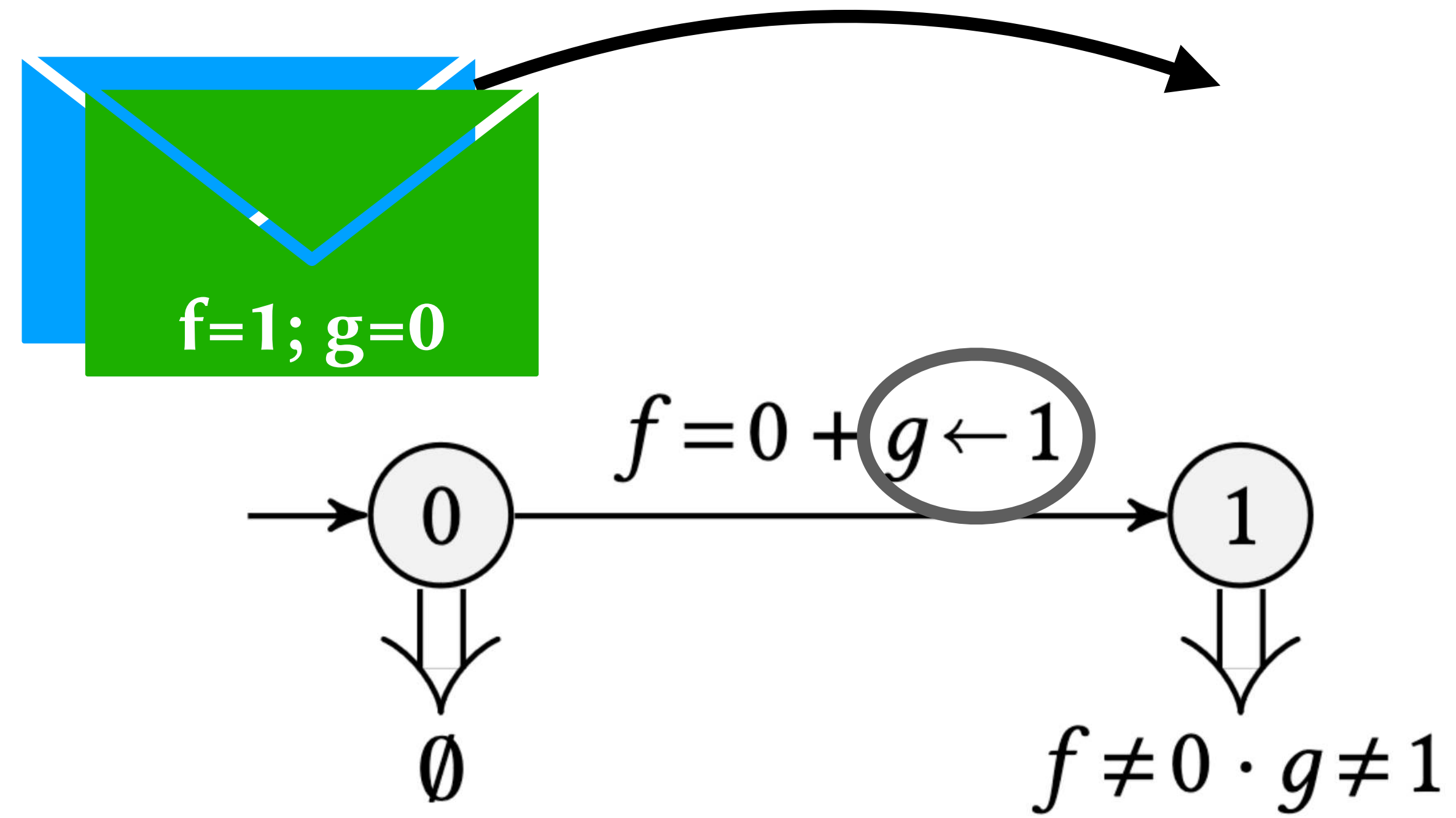
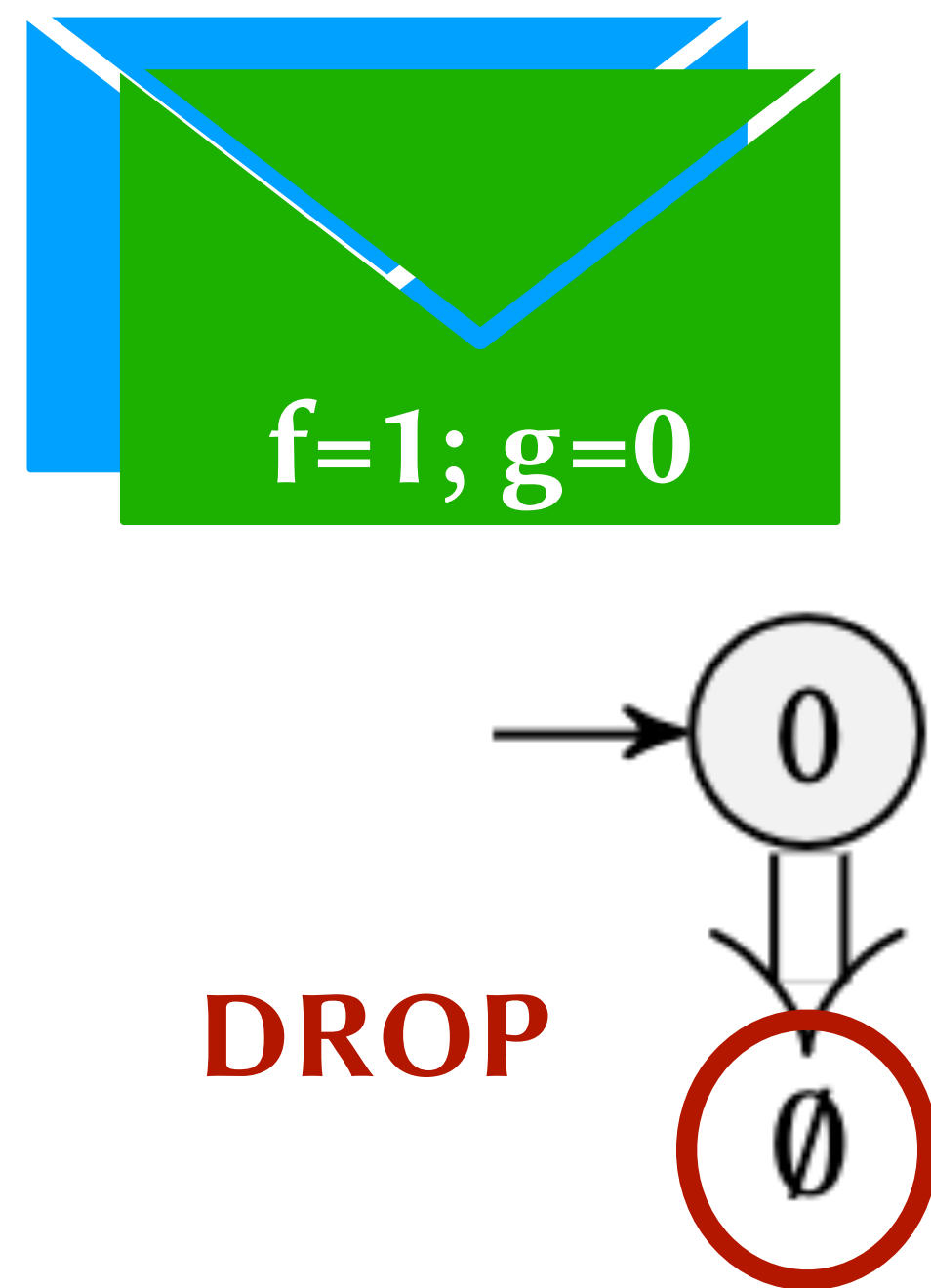
Checking Equivalence in NetKAT



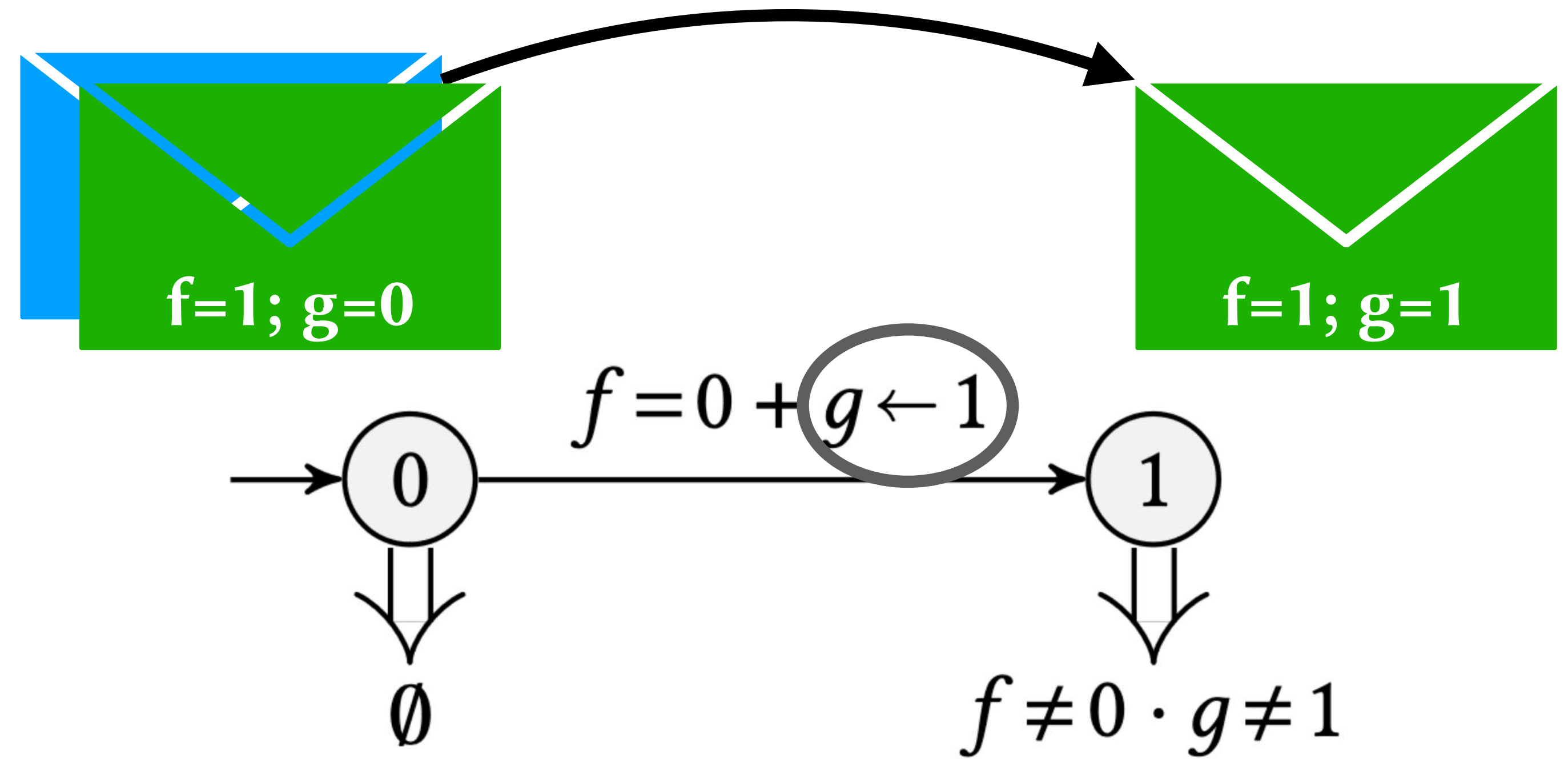
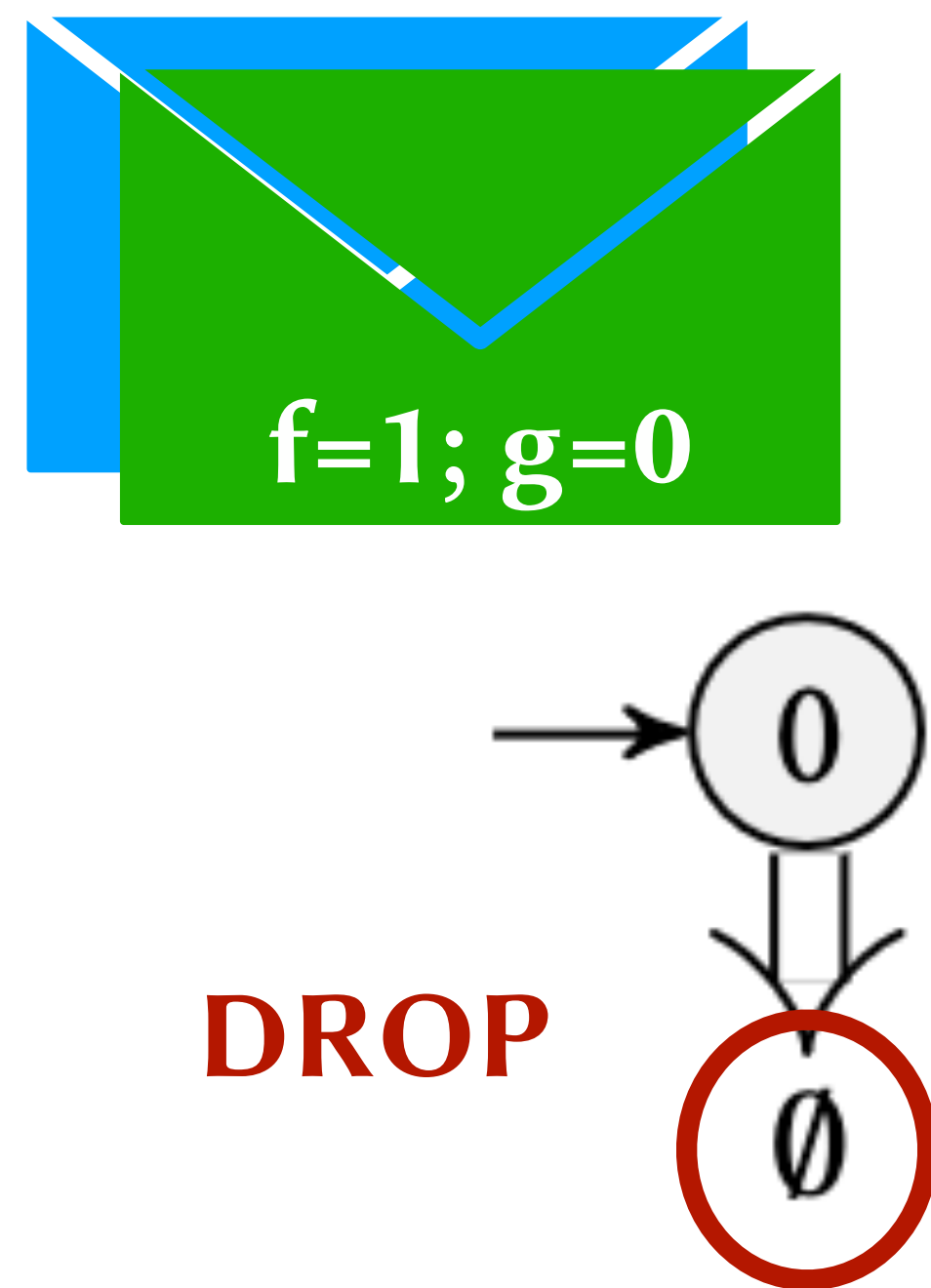
Checking Equivalence in NetKAT



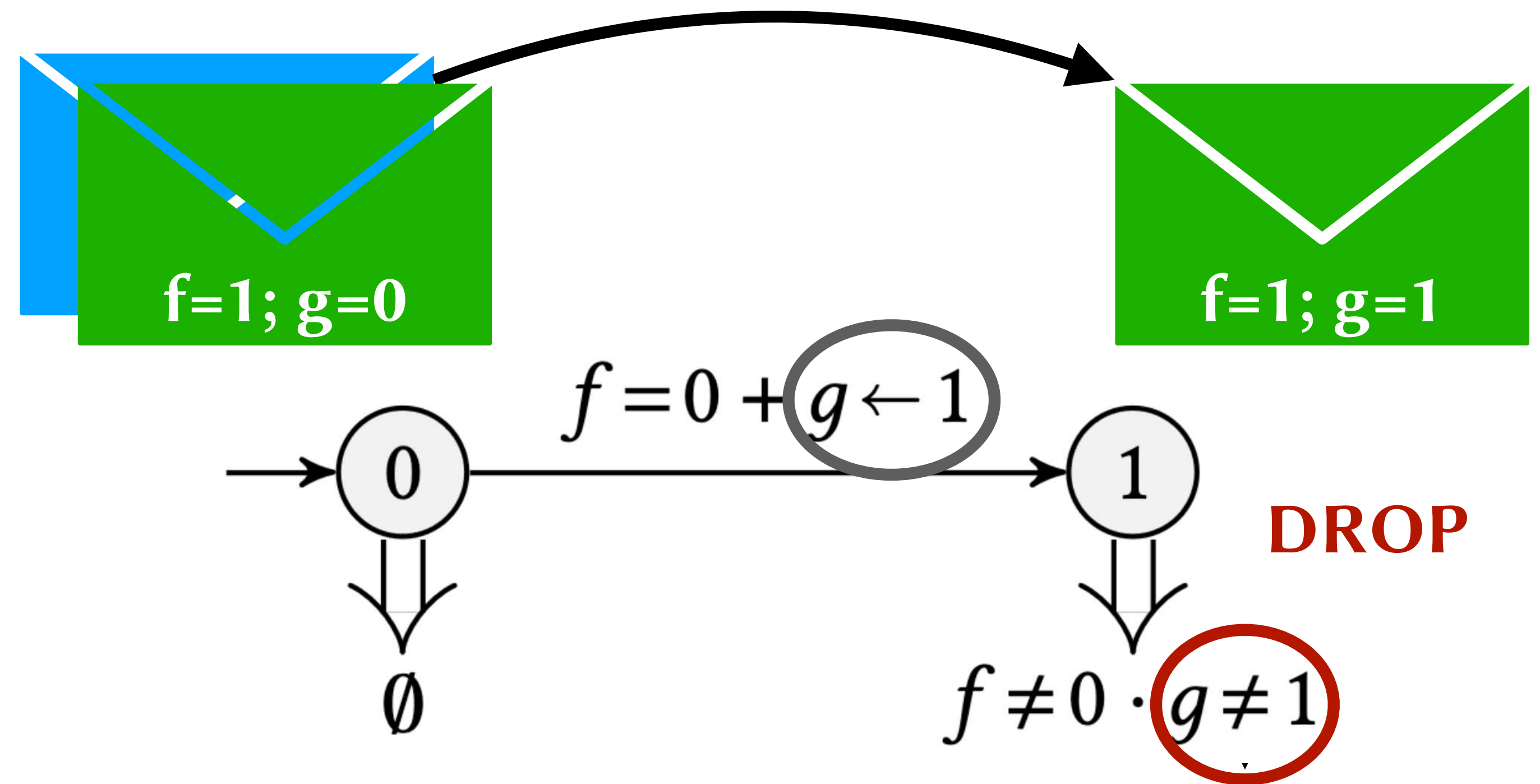
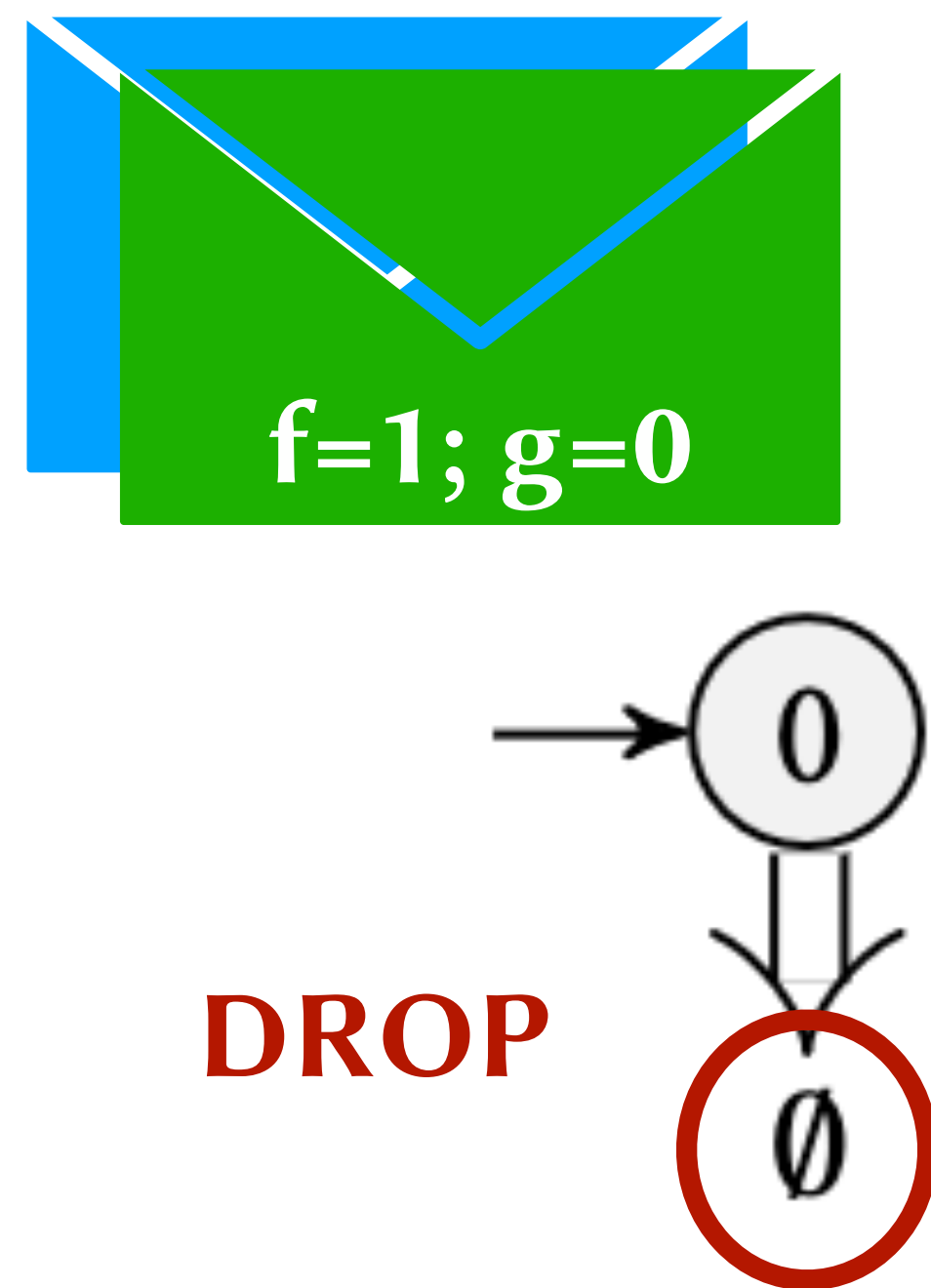
Checking Equivalence in NetKAT



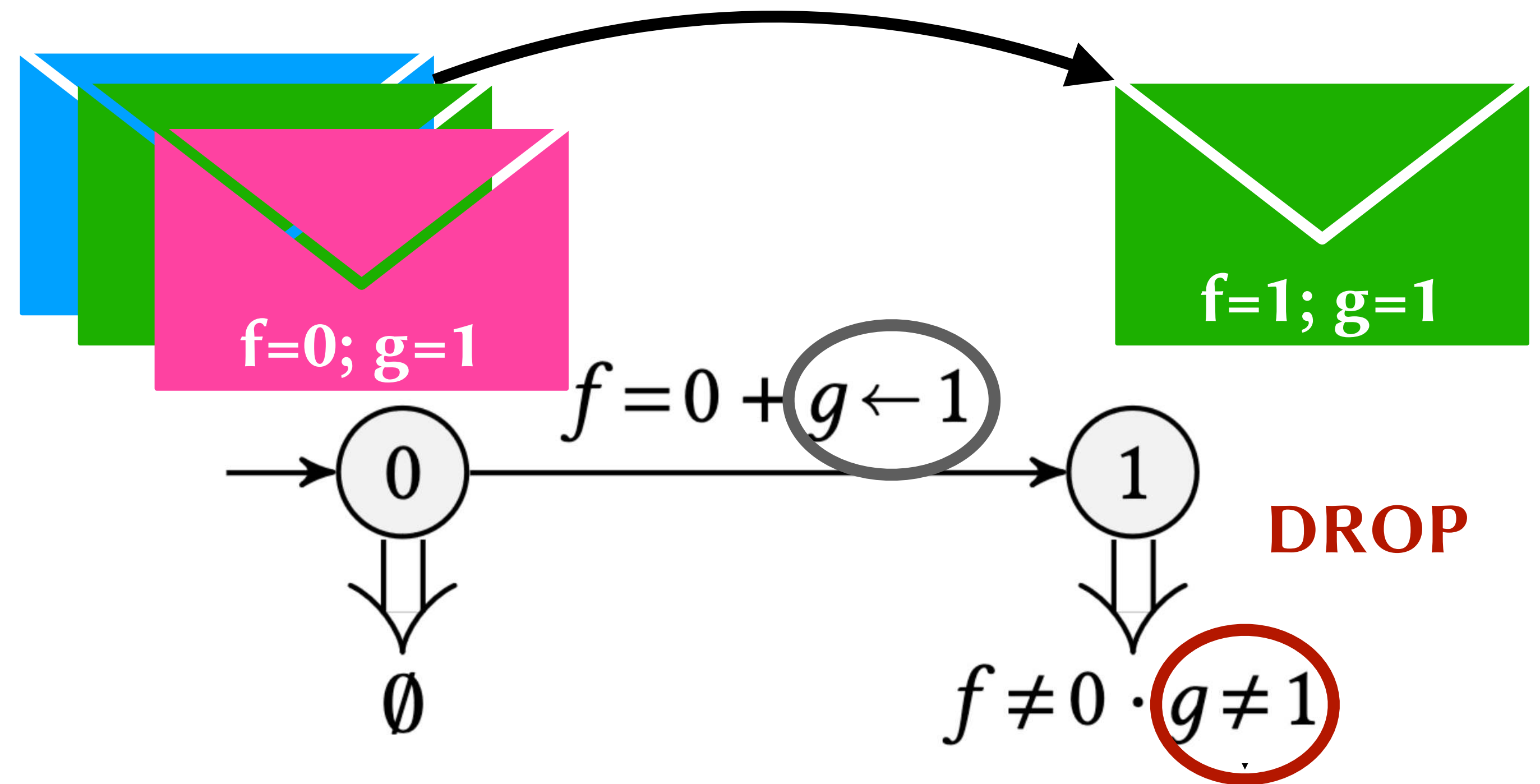
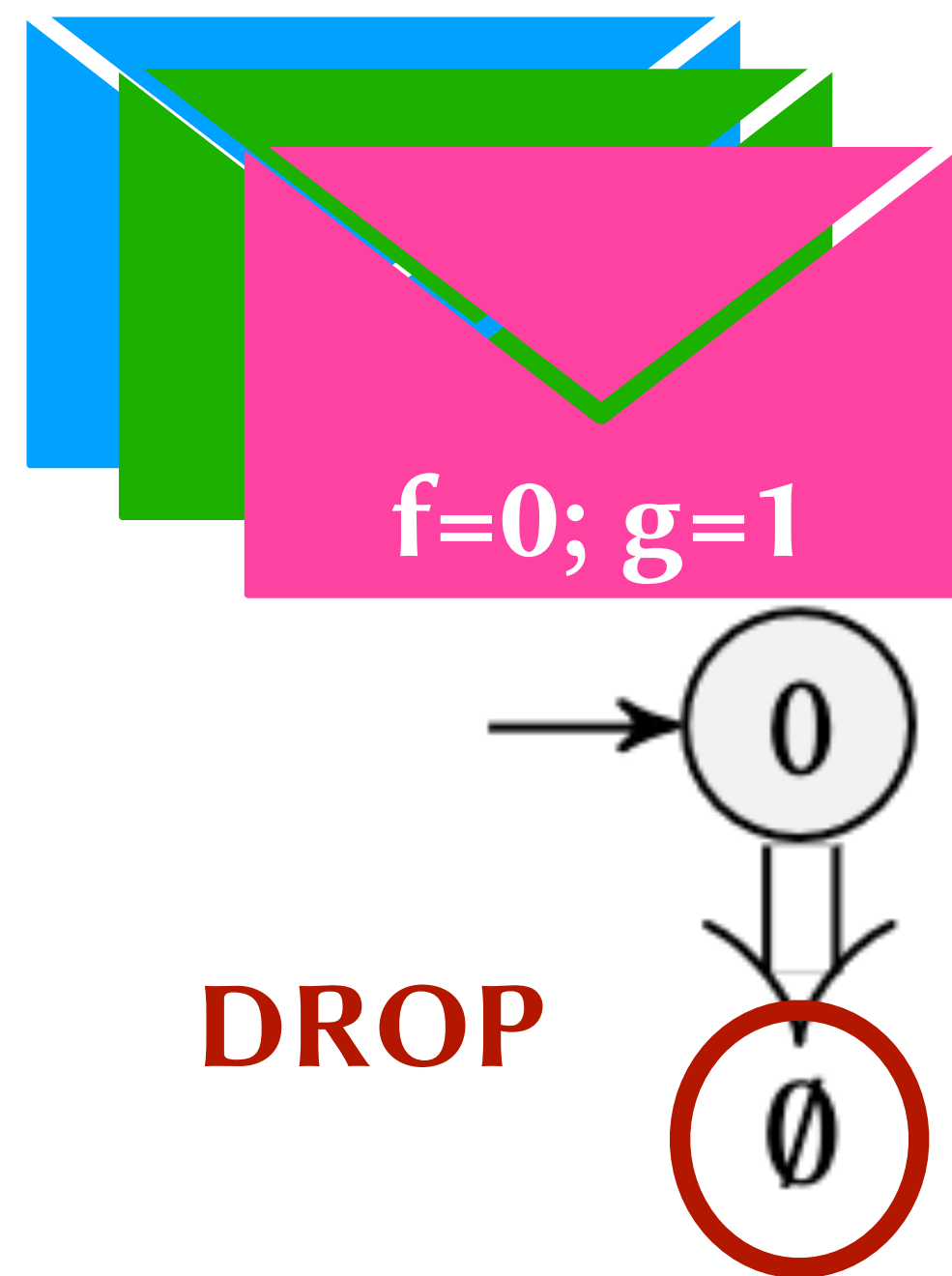
Checking Equivalence in NetKAT



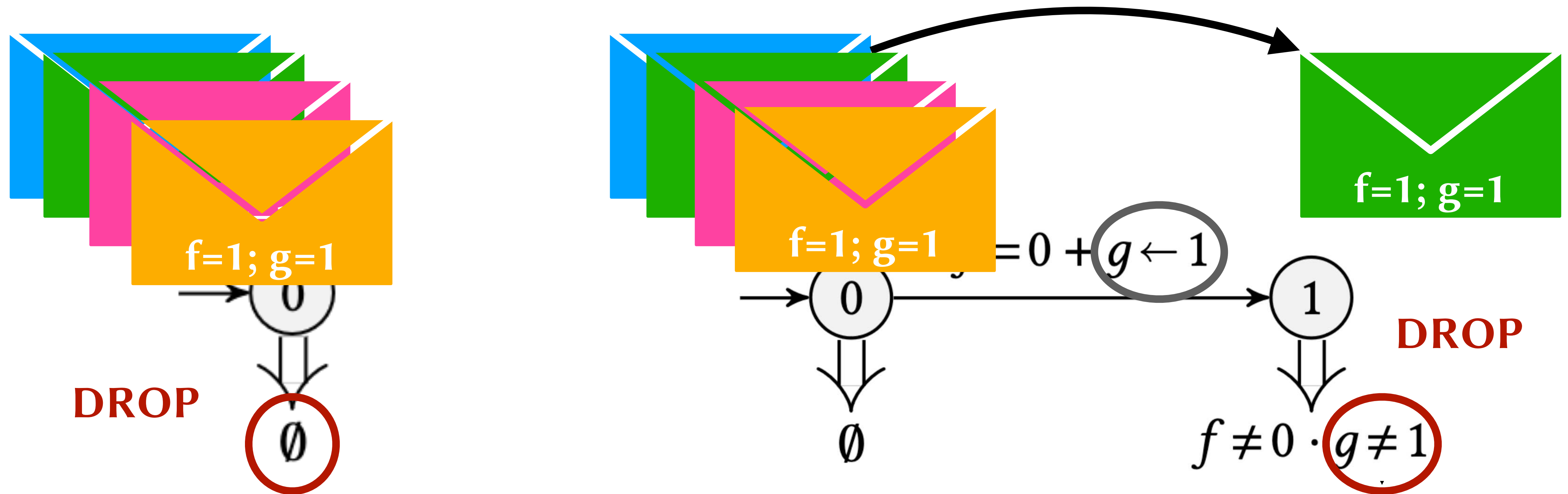
Checking Equivalence in NetKAT



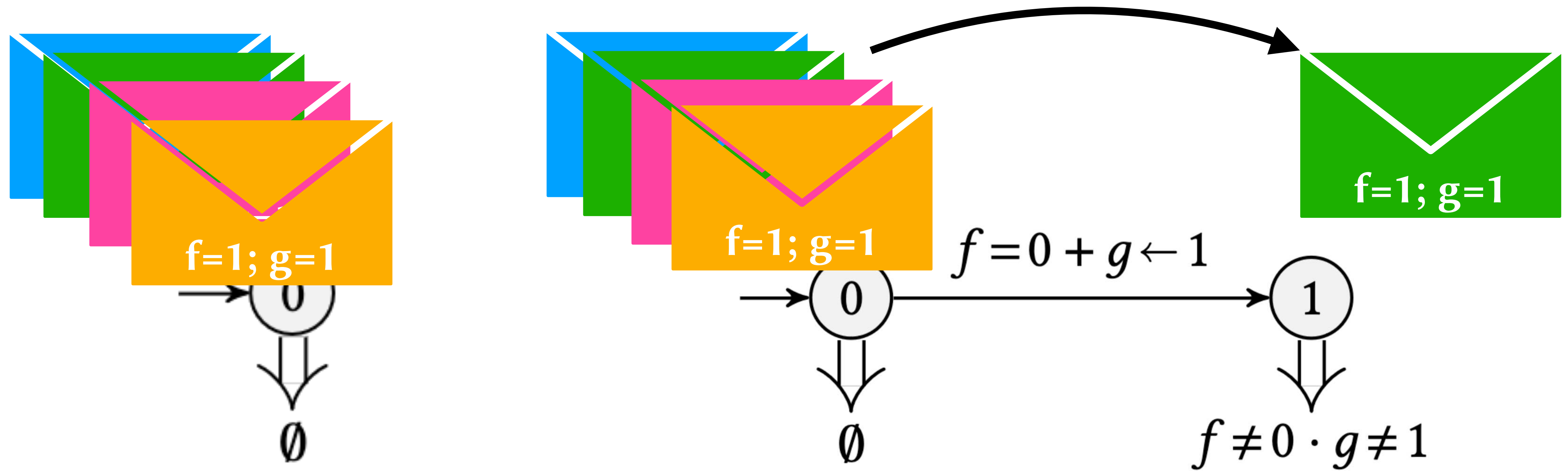
Checking Equivalence in NetKAT



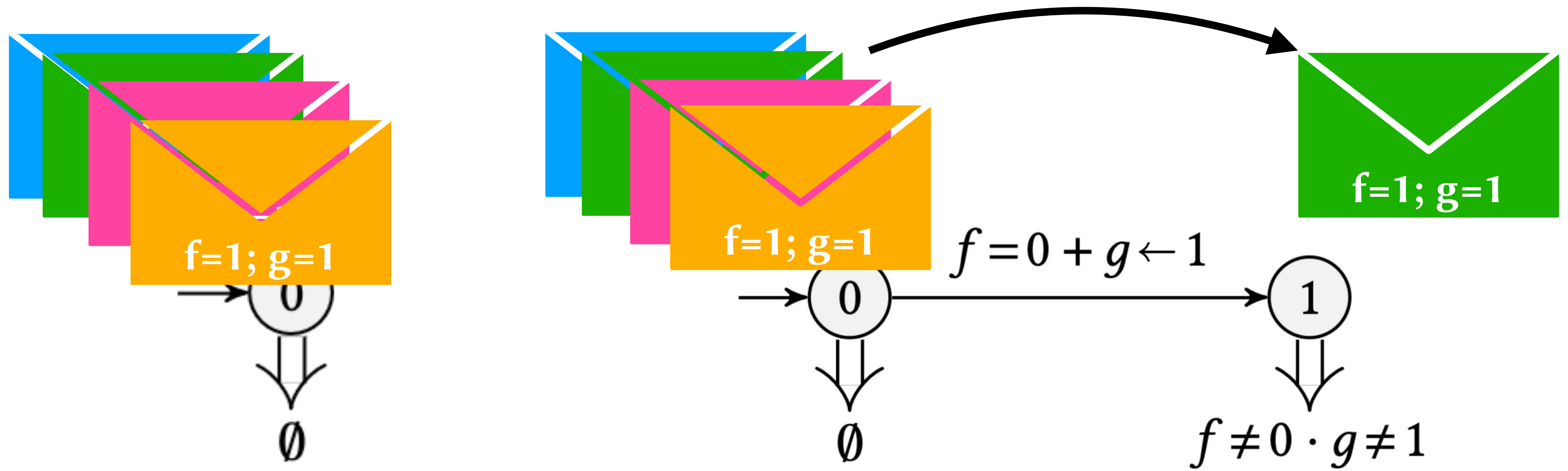
Checking Equivalence in NetKAT



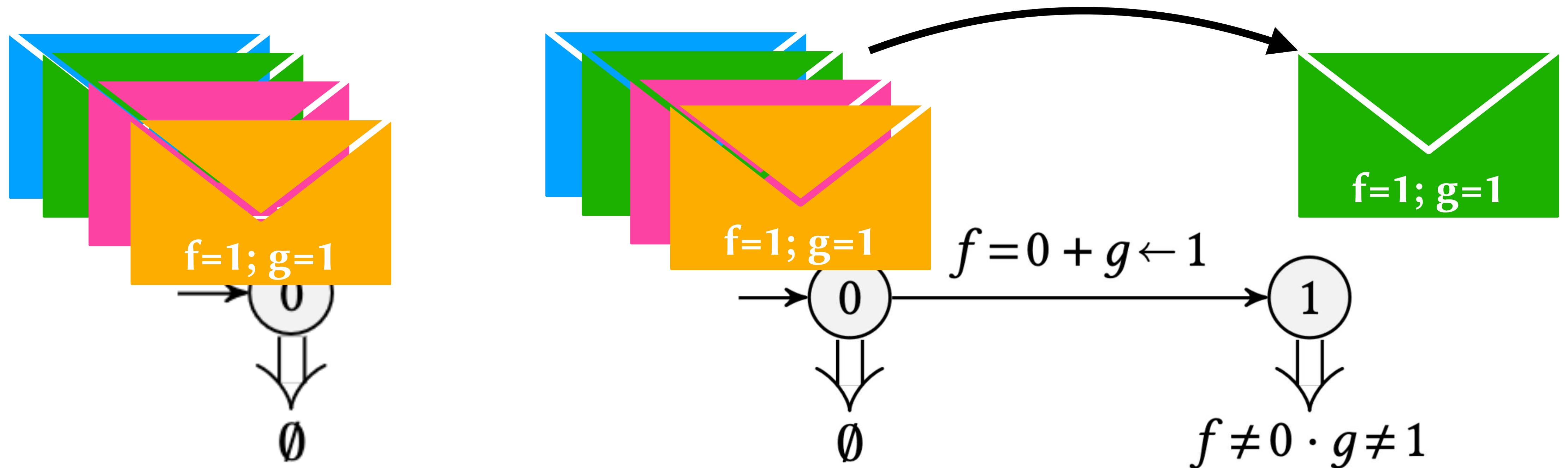
Checking Equivalence in NetKAT



Checking Equivalence in NetKAT

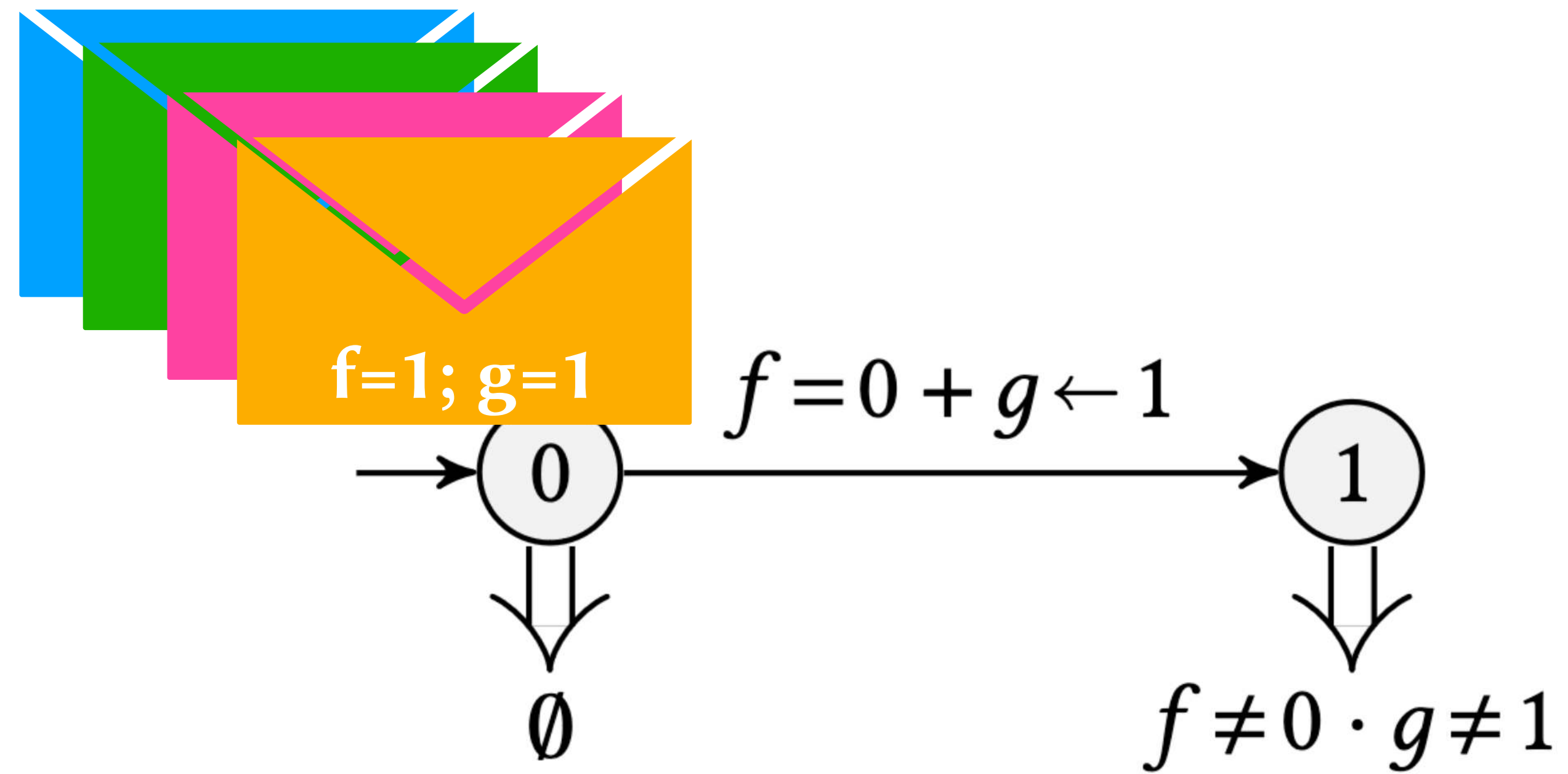


Checking Equivalence in NetKAT



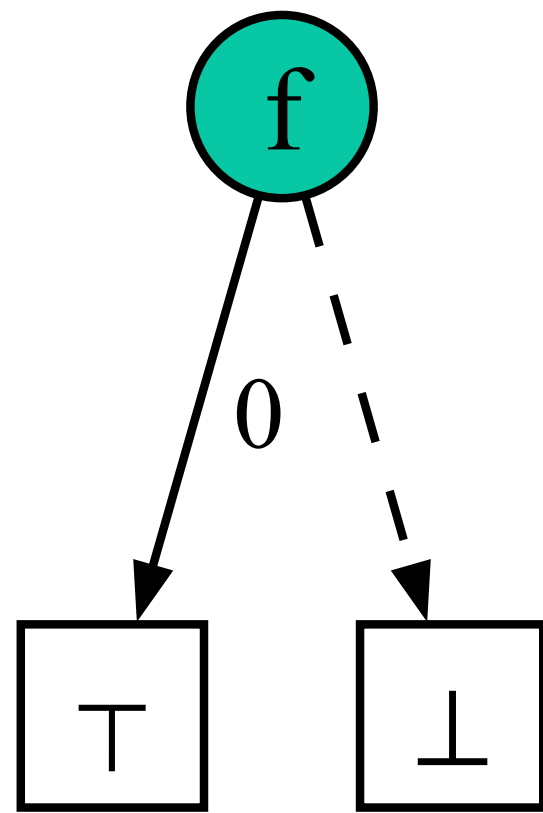
Core insight: packets are “clumped” in most policies and can be represented symbolically

Represent Sets of Packets Symbolically

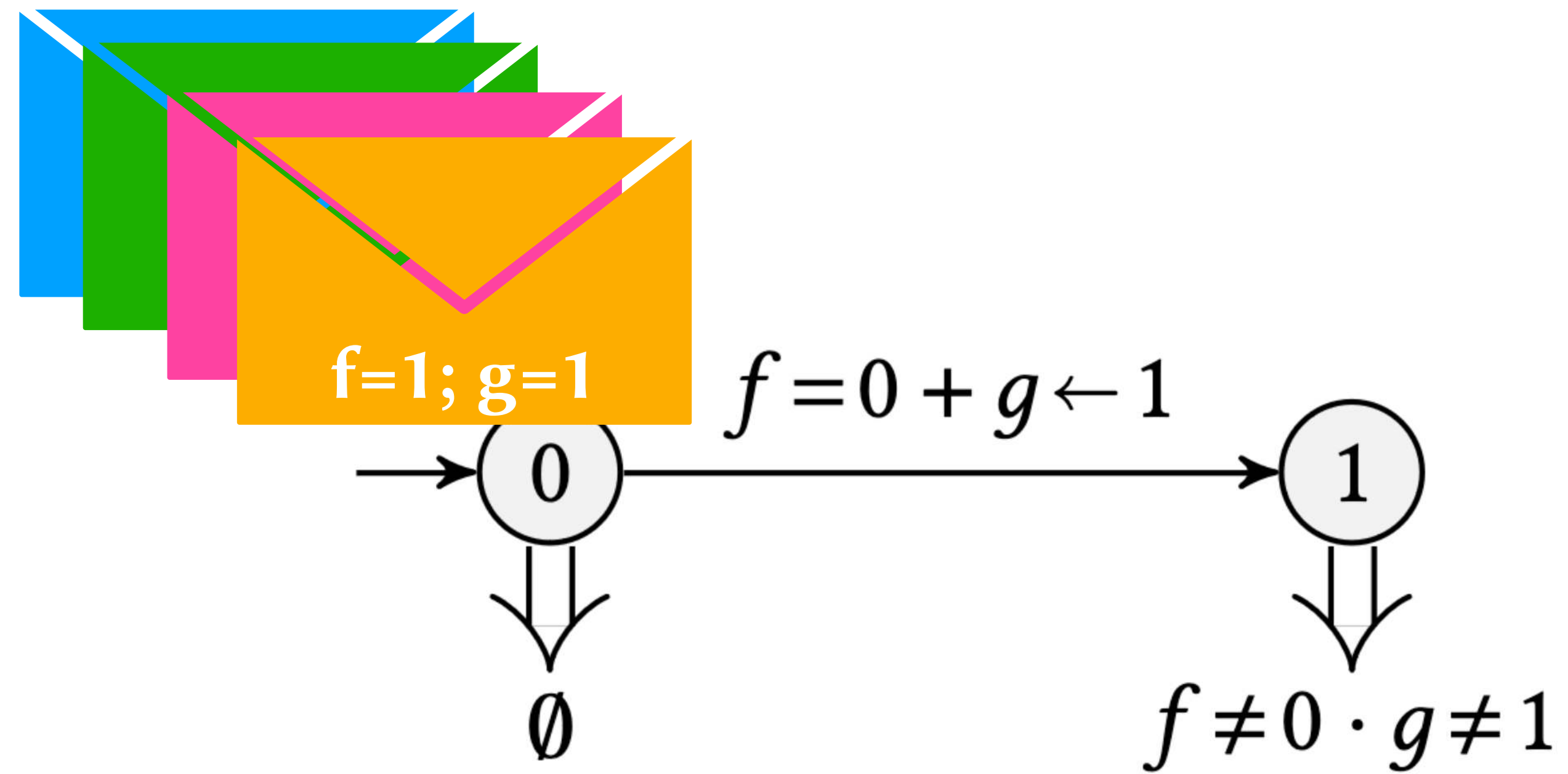


Represent Sets of Packets Symbolically

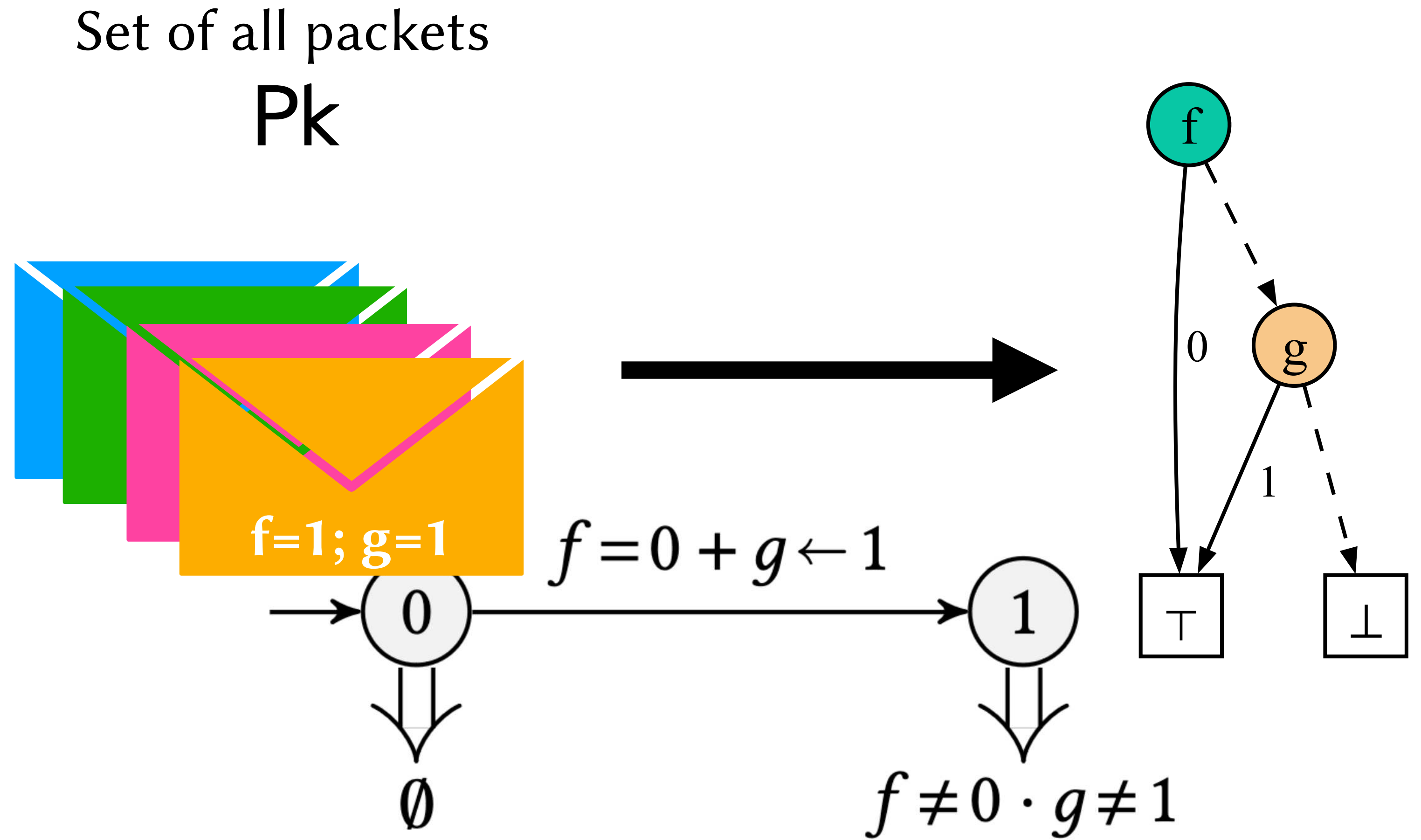
$$\{\alpha \in \text{Pk} \mid \alpha.f = 0\}$$



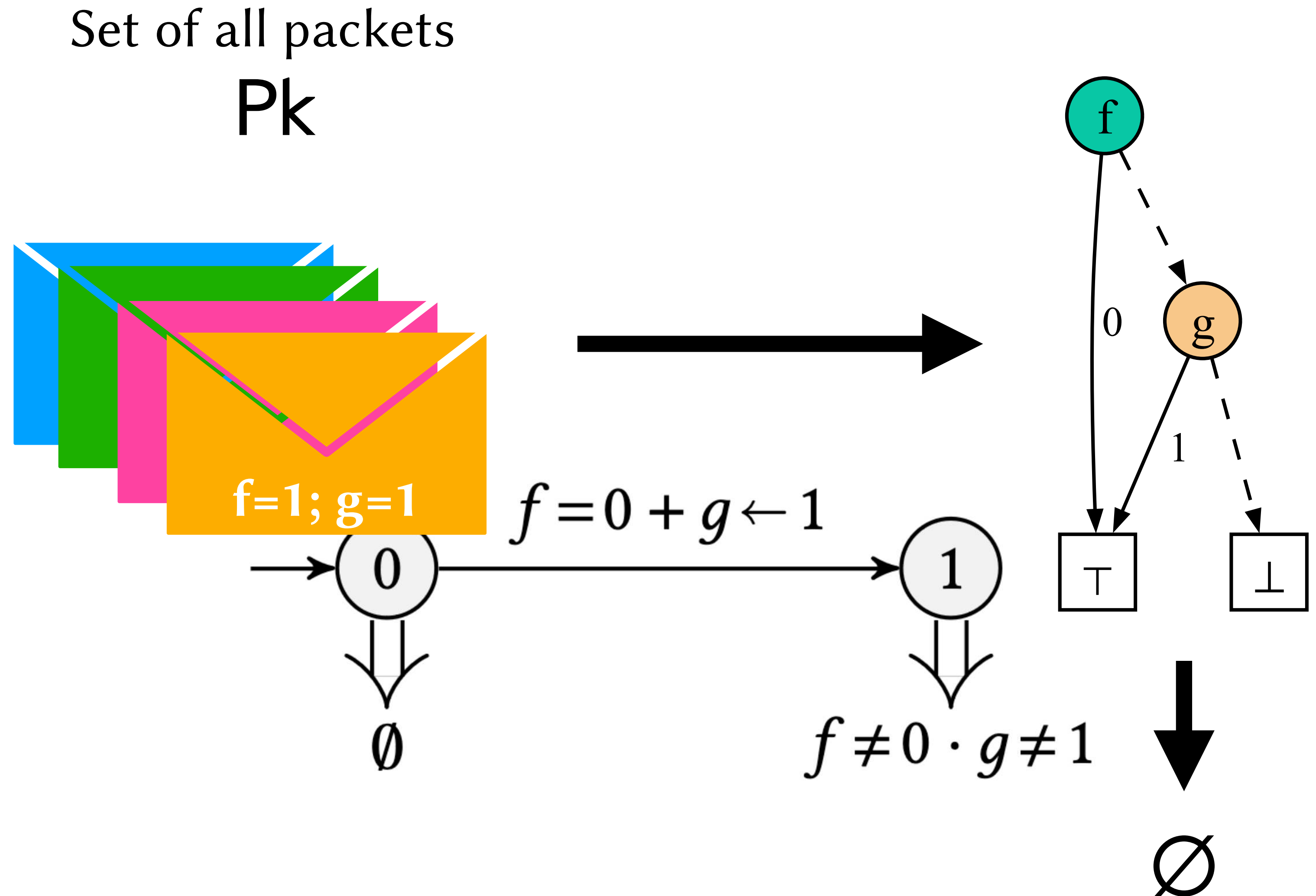
Binary Decision Diagram (BDD)



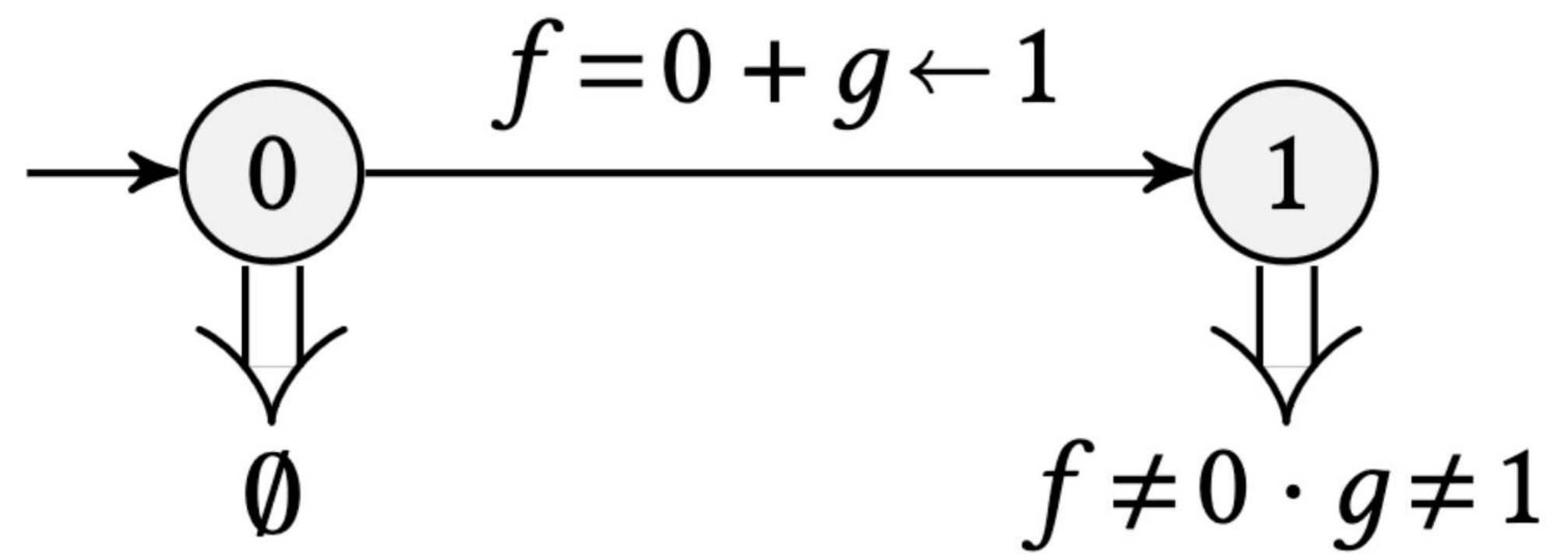
Represent Sets of Packets Symbolically



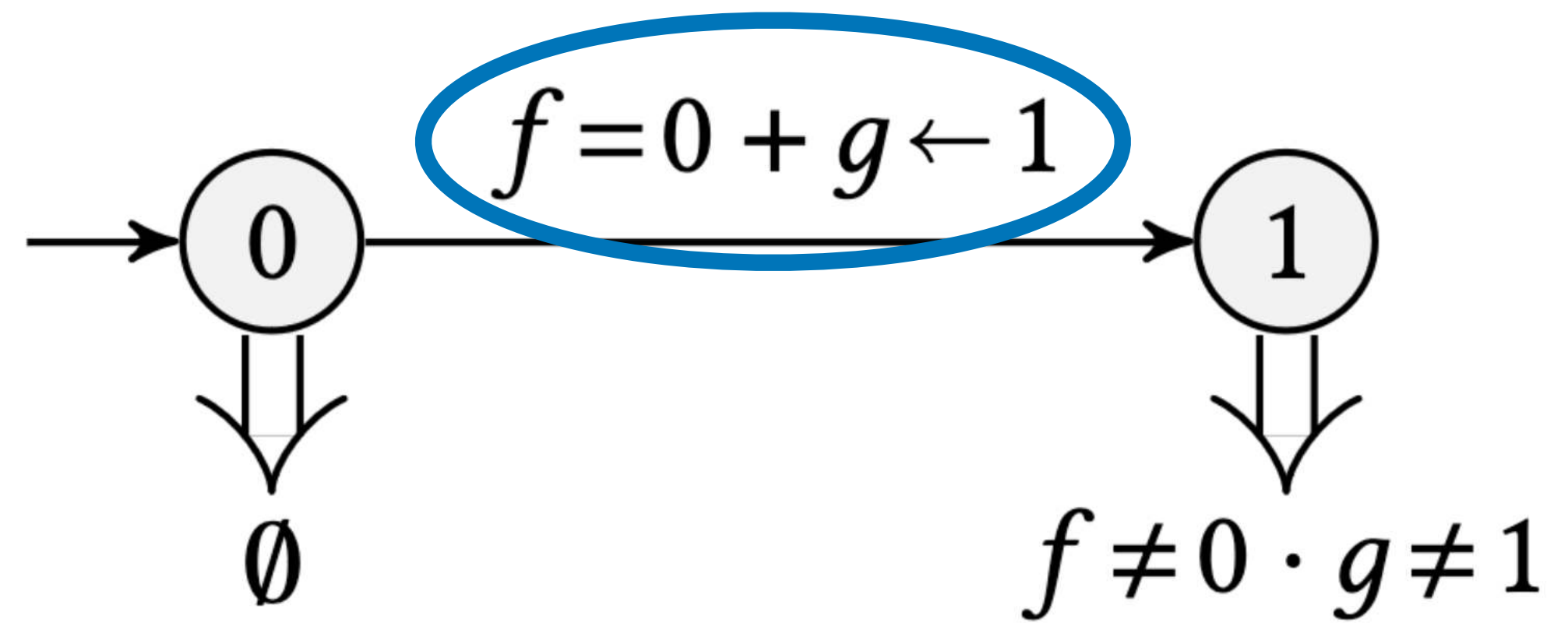
Represent Sets of Packets Symbolically



Represent Transitions Symbolically

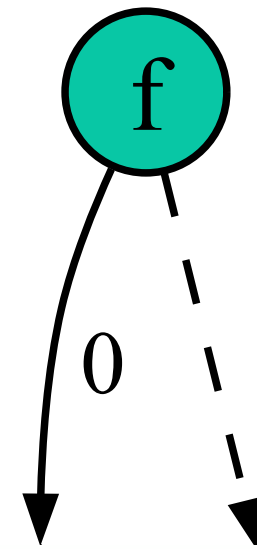


Represent Transitions Symbolically

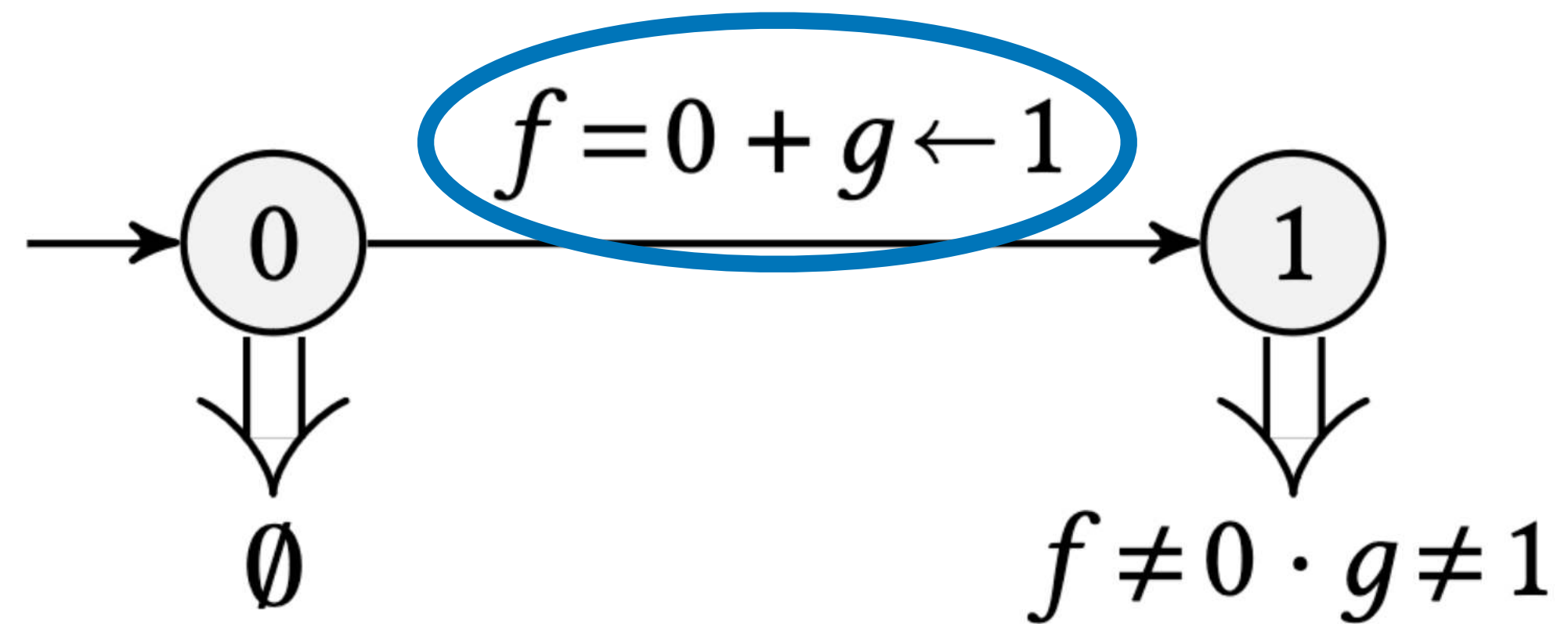


Represent Transitions Symbolically

Symbolic Packet
Program (SPP)

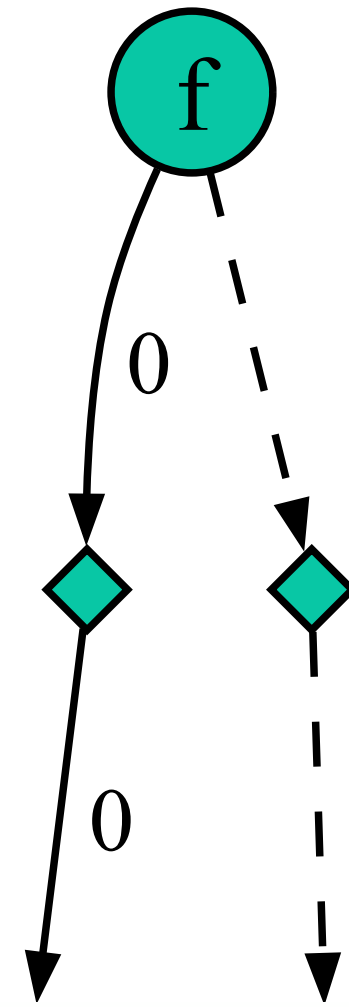


Circle-layers are filters



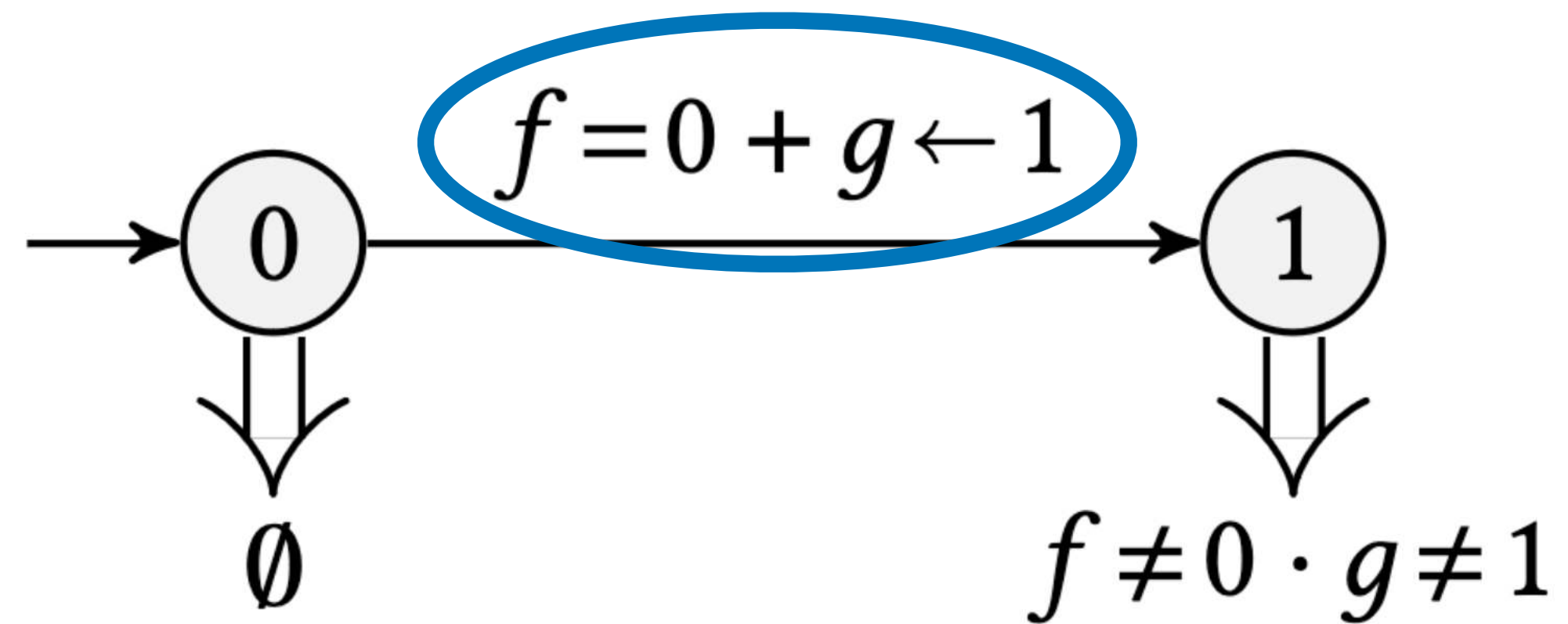
Represent Transitions Symbolically

Symbolic Packet
Program (SPP)



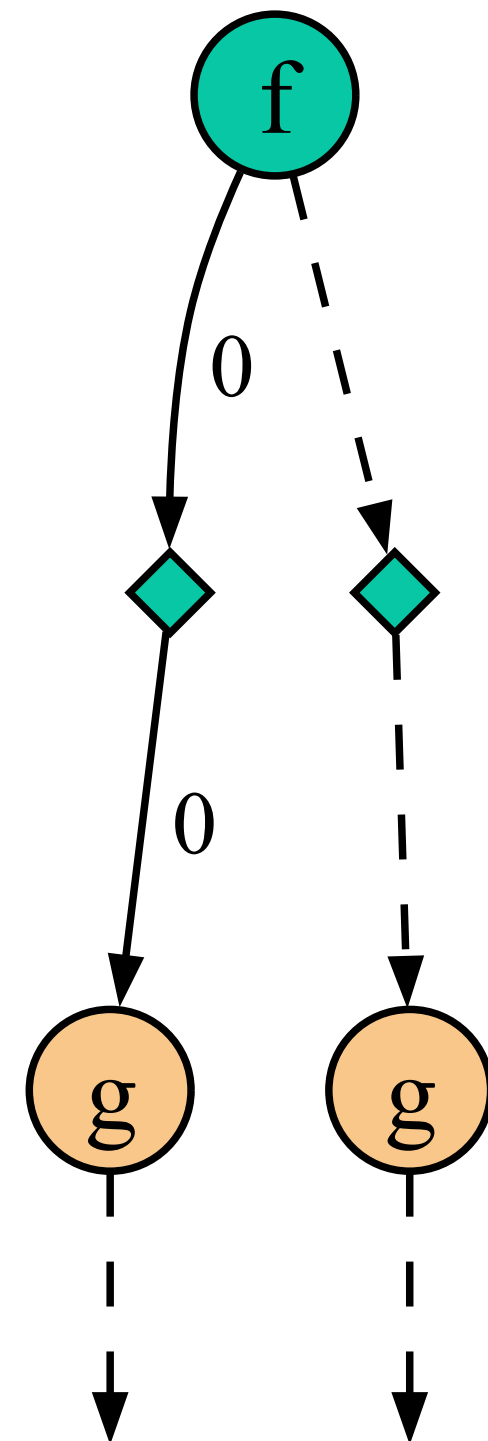
Circle-layers are filters

Diamond-layers are assignments



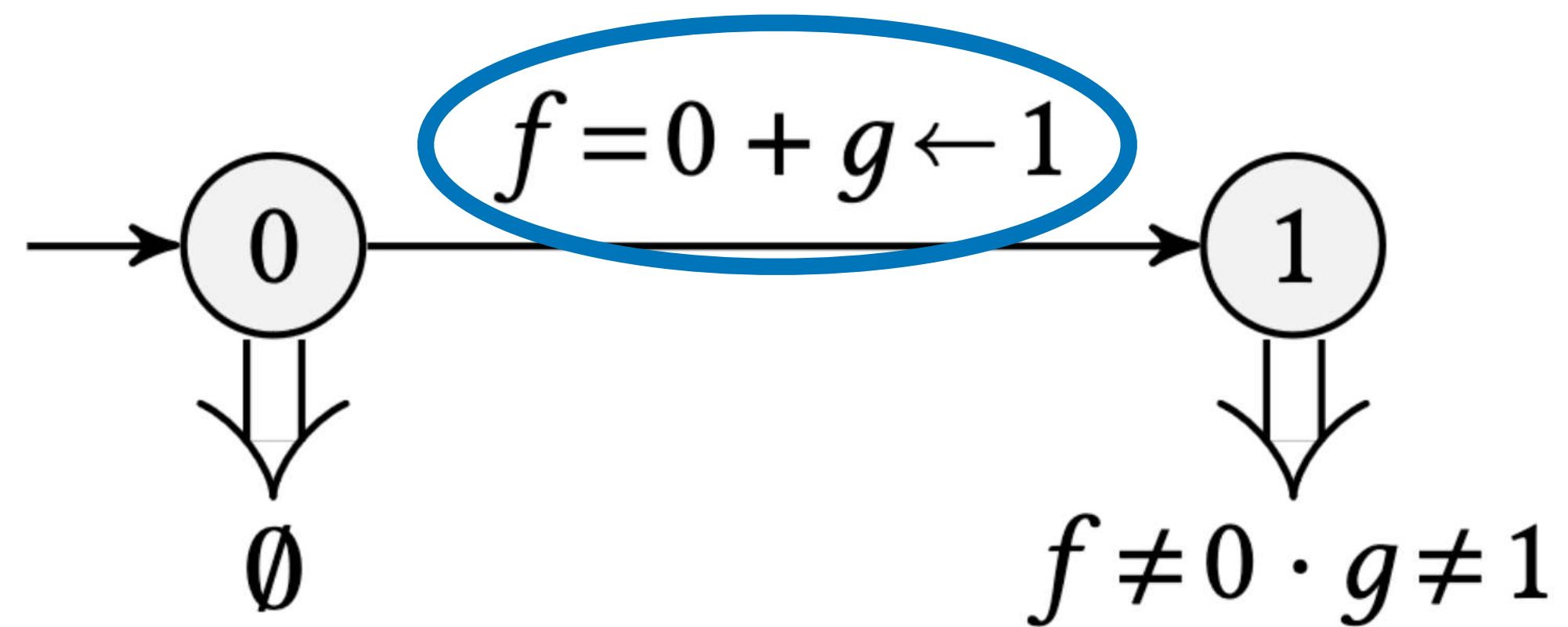
Represent Transitions Symbolically

Symbolic Packet
Program (SPP)



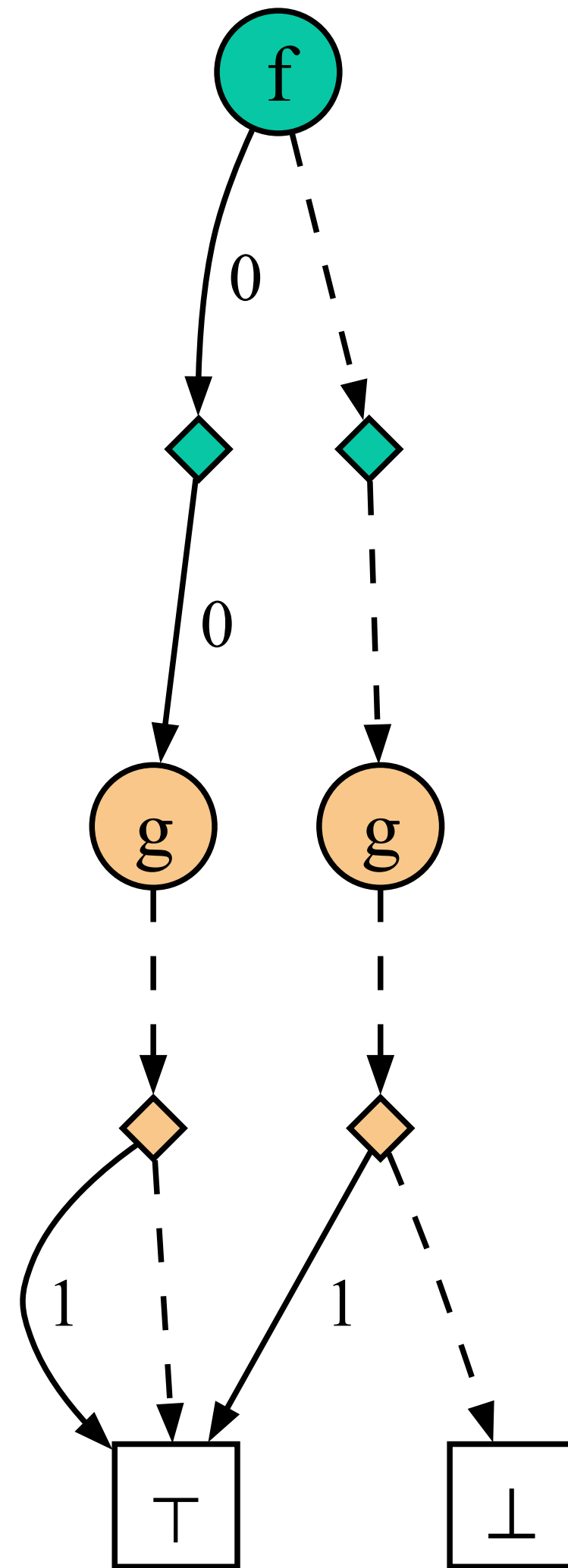
Circle-layers are filters

Diamond-layers are assignments



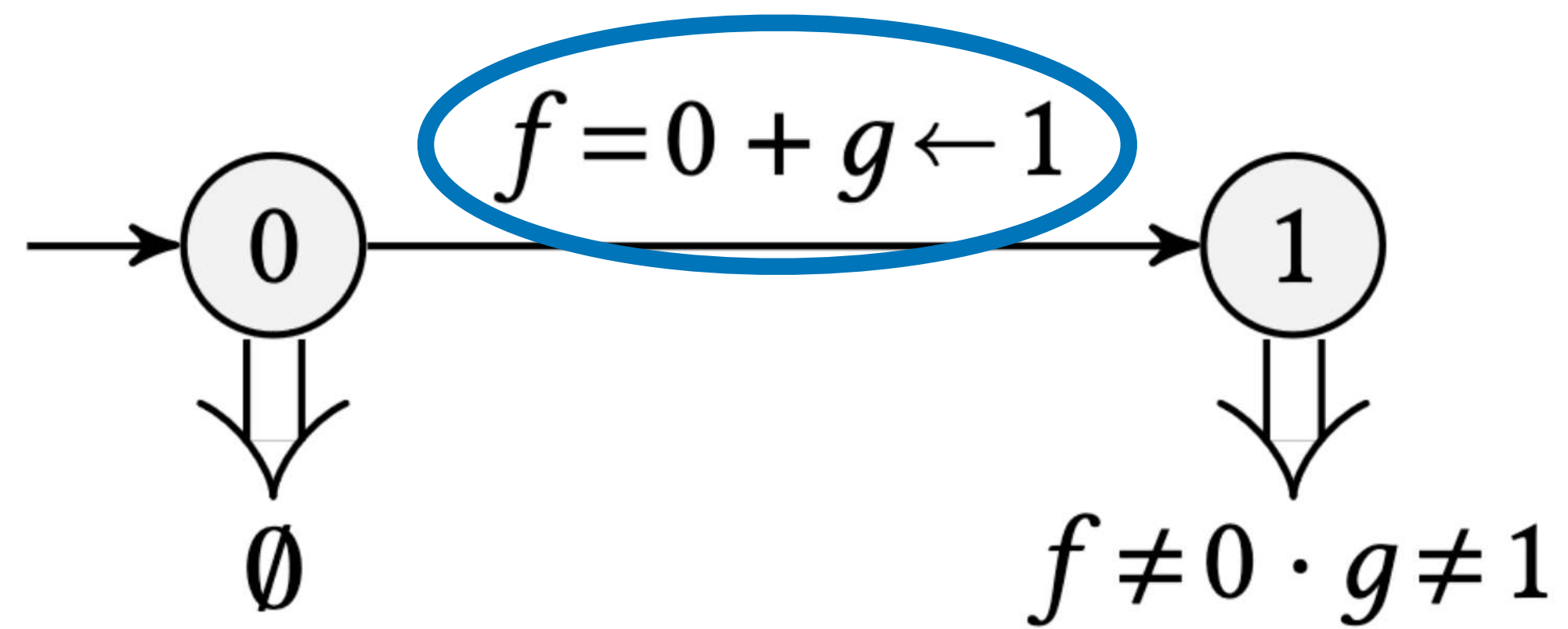
Represent Transitions Symbolically

Symbolic Packet
Program (SPP)



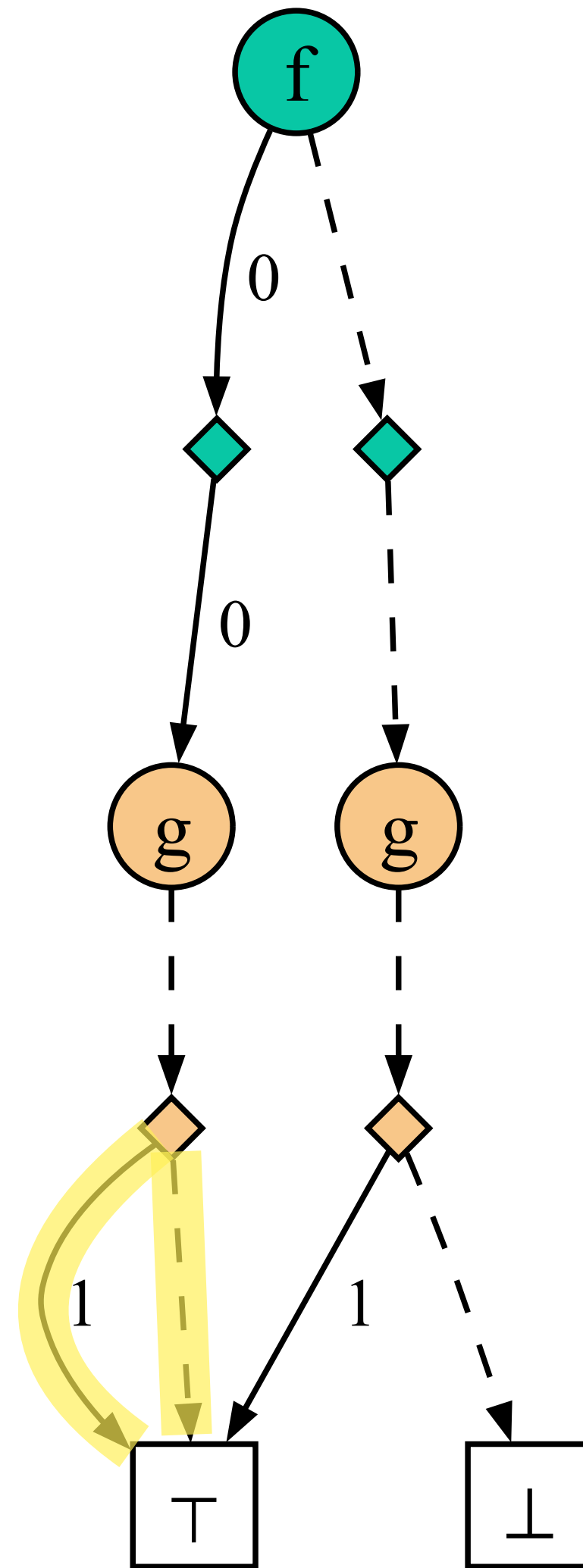
Circle-layers are filters

Diamond-layers are assignments



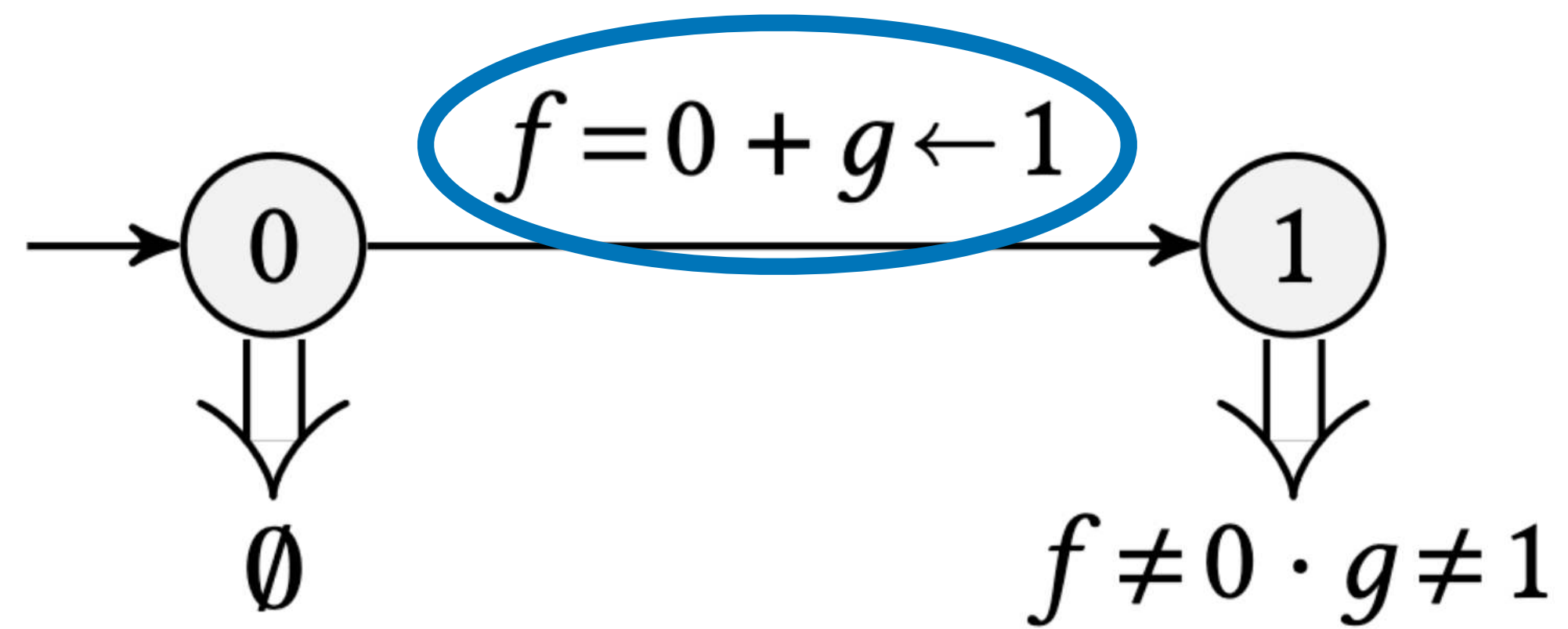
Represent Transitions Symbolically

Symbolic Packet
Program (SPP)



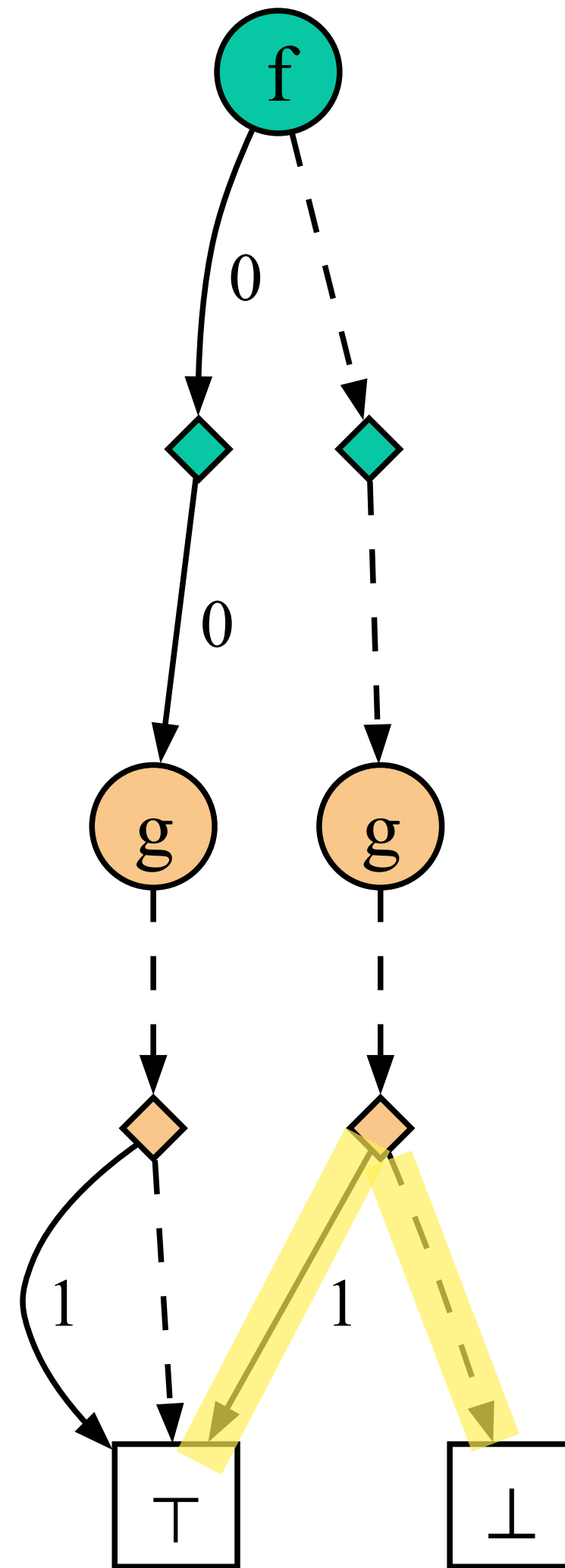
Circle-layers are filters

Diamond-layers are assignments



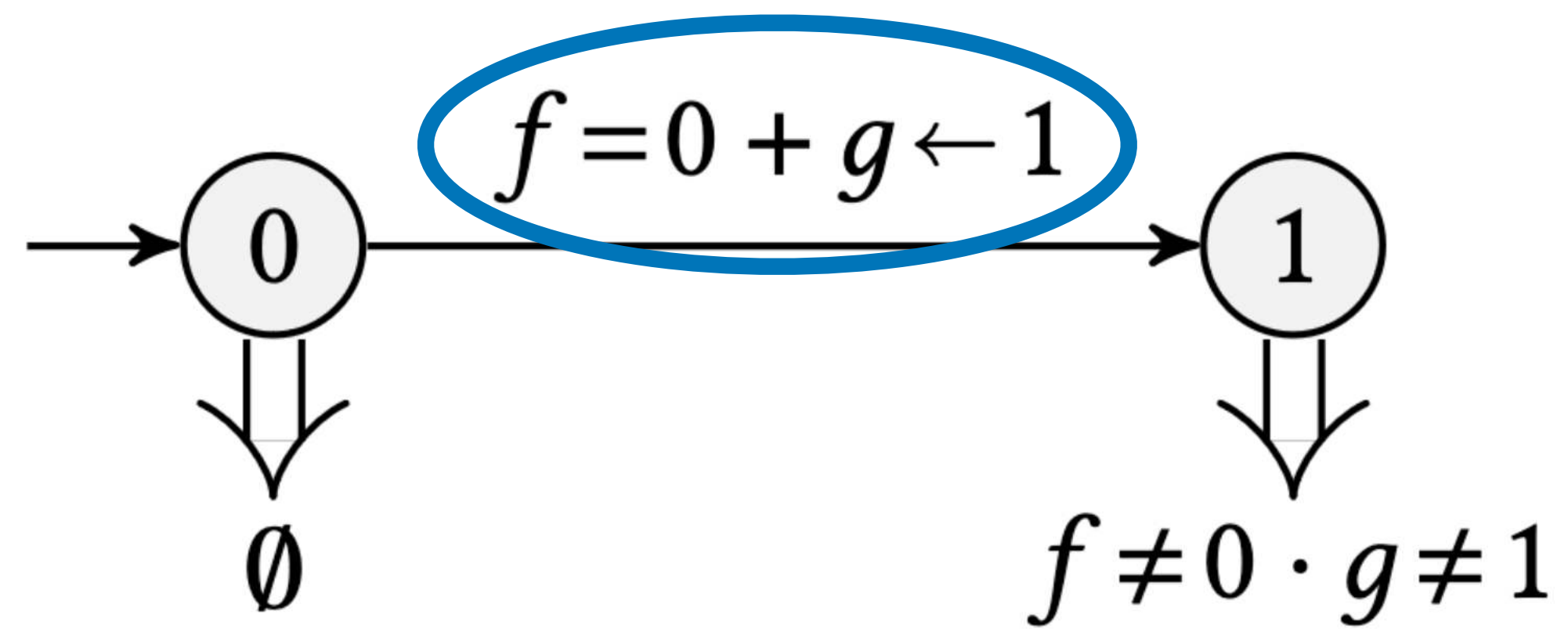
Represent Transitions Symbolically

Symbolic Packet
Program (SPP)



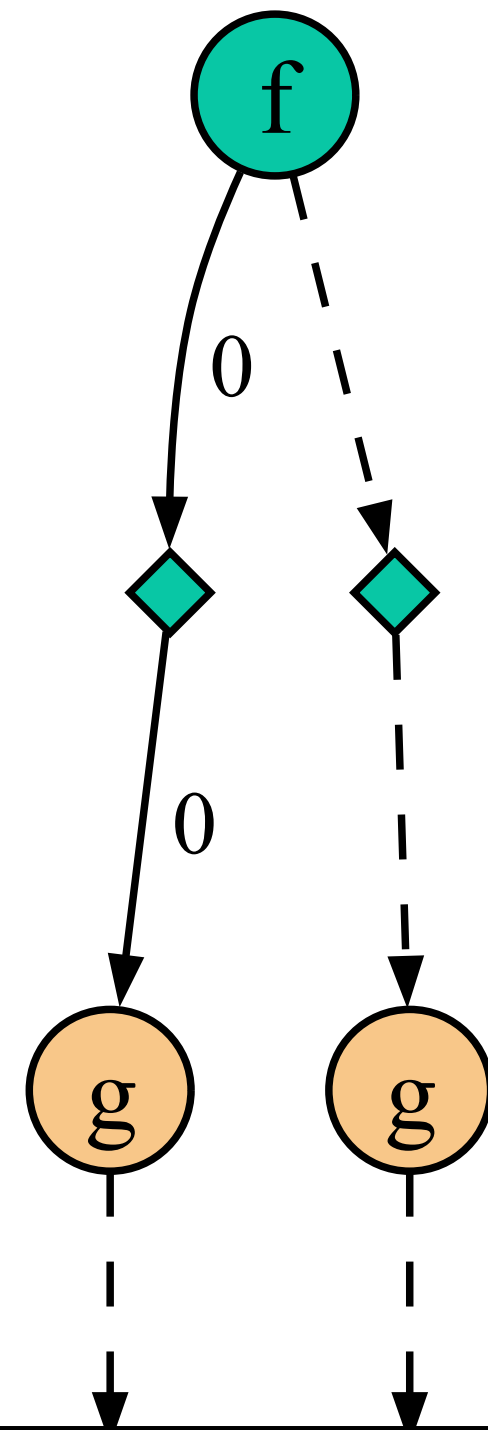
Circle-layers are filters

Diamond-layers are assignments



Represent Transitions Symbolically

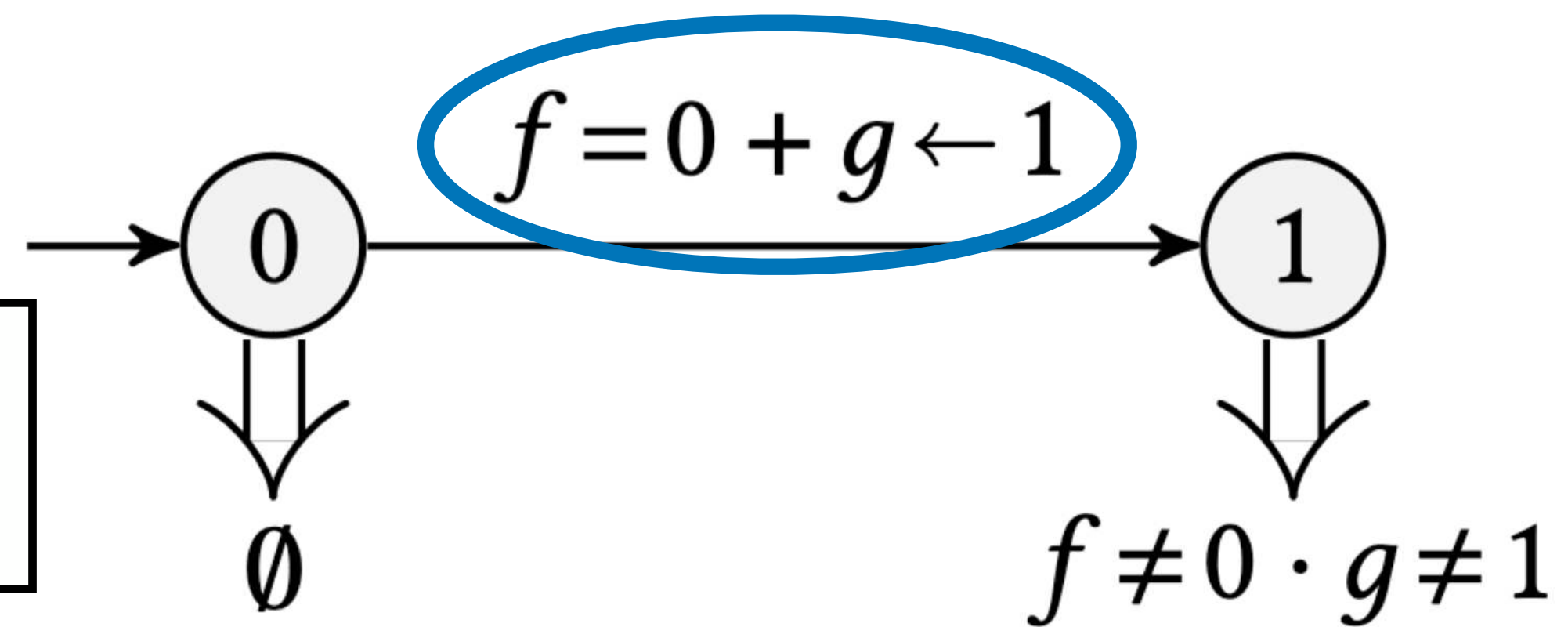
Symbolic Packet
Program (SPP)



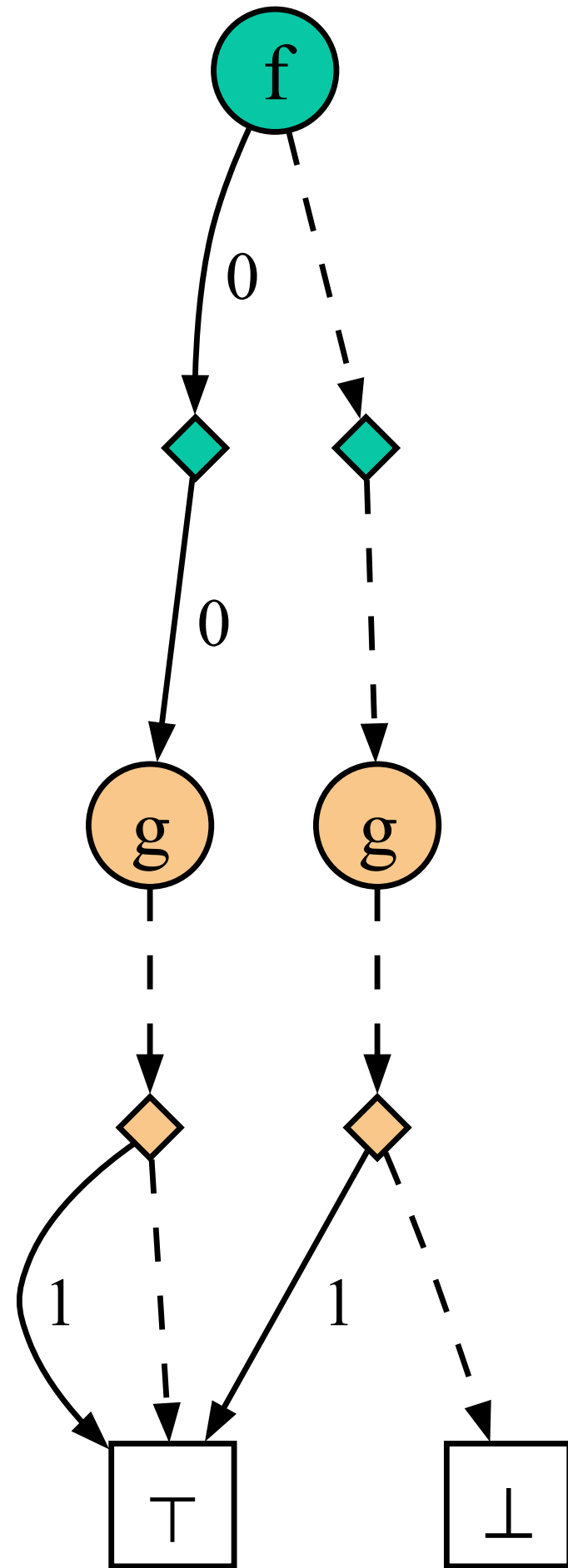
Circle-layers are filters

Diamond-layers are assignments

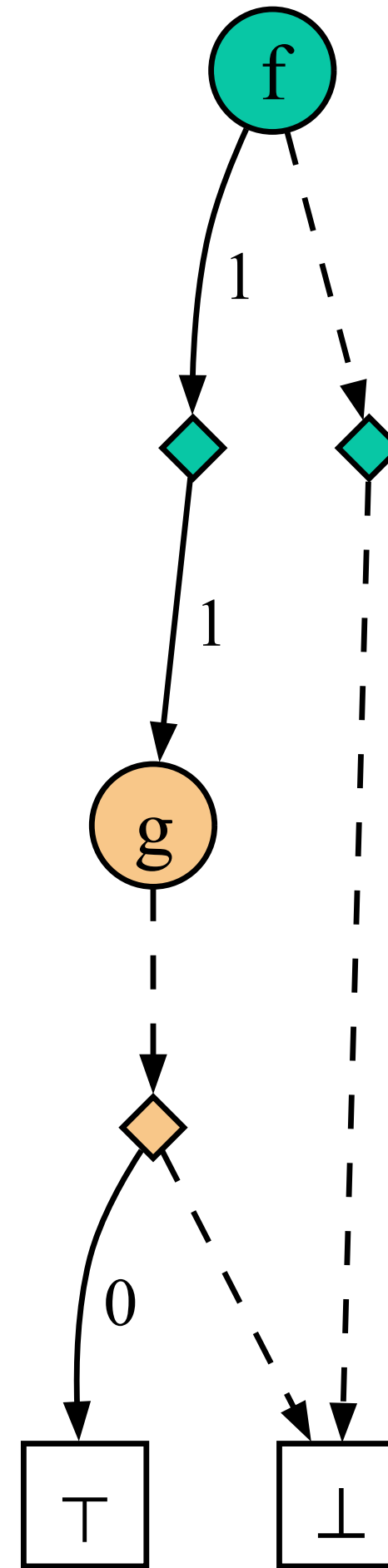
SPPs are canonical!



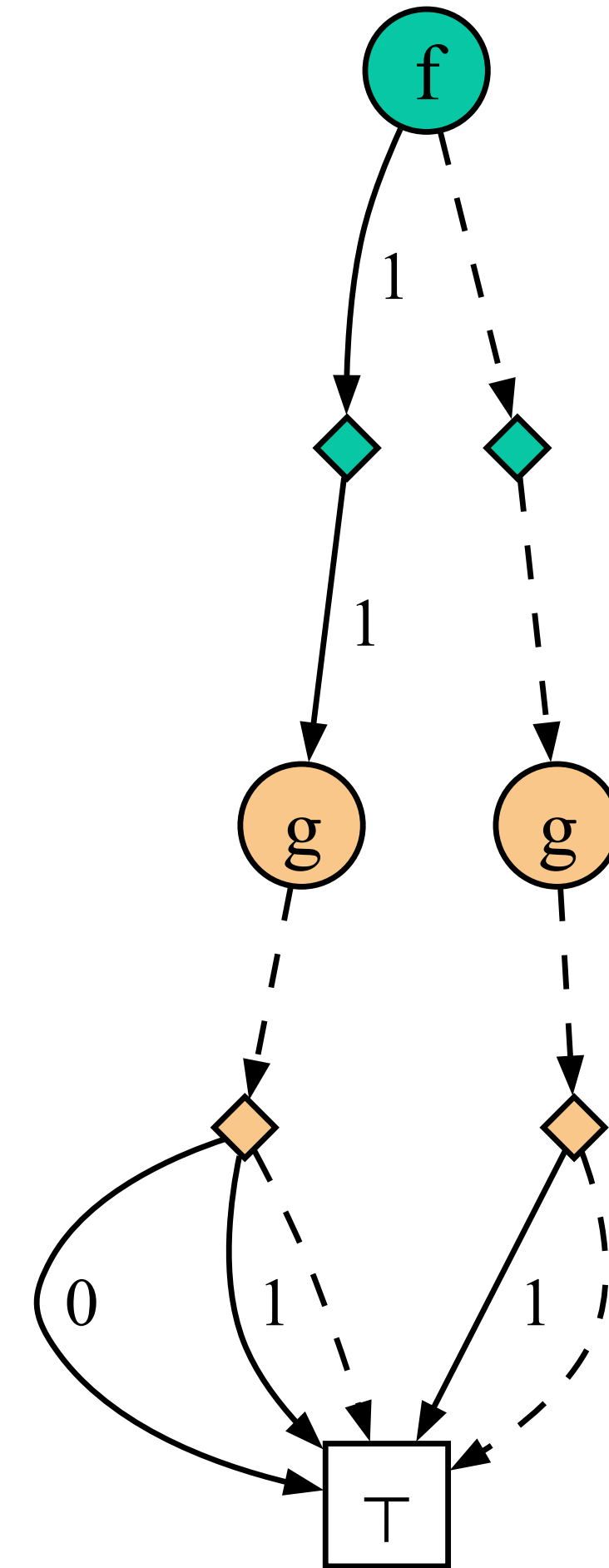
Operations on Symbolic Packet Programs (SPPs)



$$p \triangleq f=0 + g \leftarrow 1$$

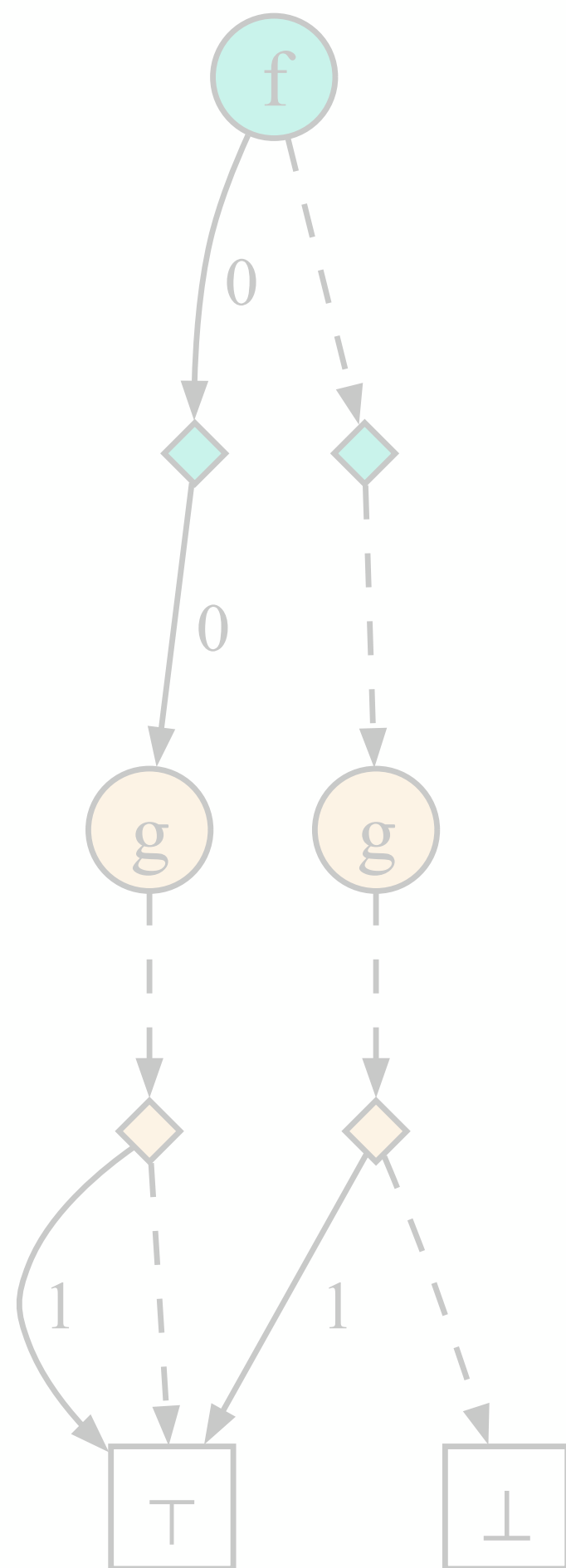


$$q \triangleq f=1 \cdot g \leftarrow 0$$

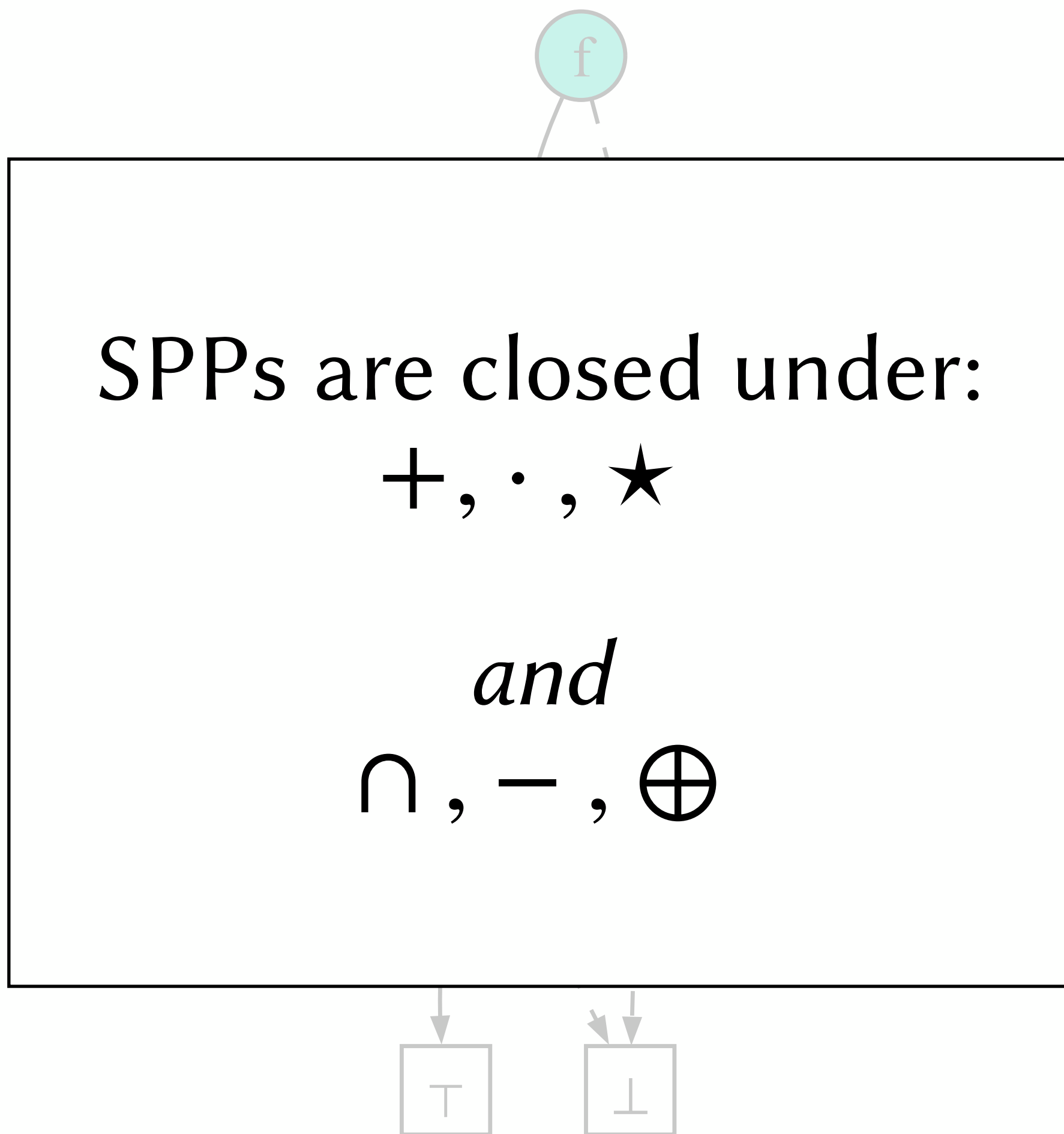


$$(p \oplus q)^\star$$

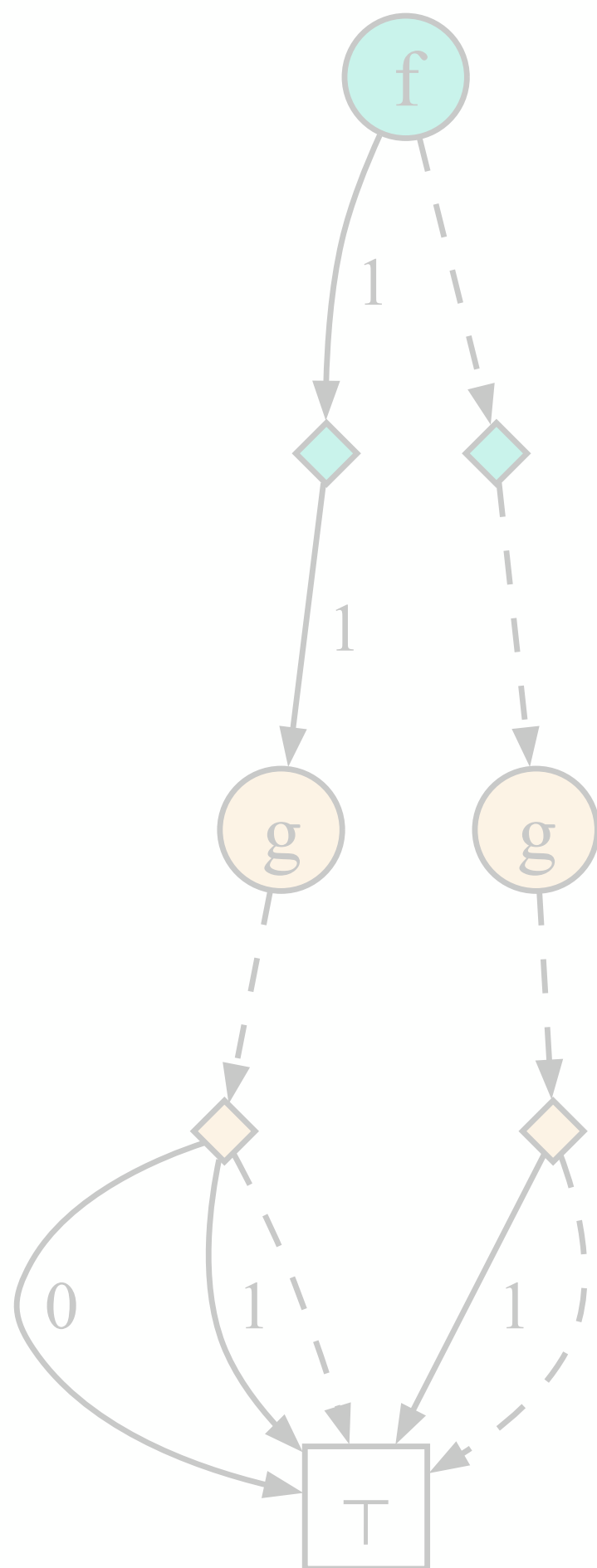
Operations on Symbolic Packet Programs (SPPs)



$$p \triangleq f=0 + g \leftarrow 1$$

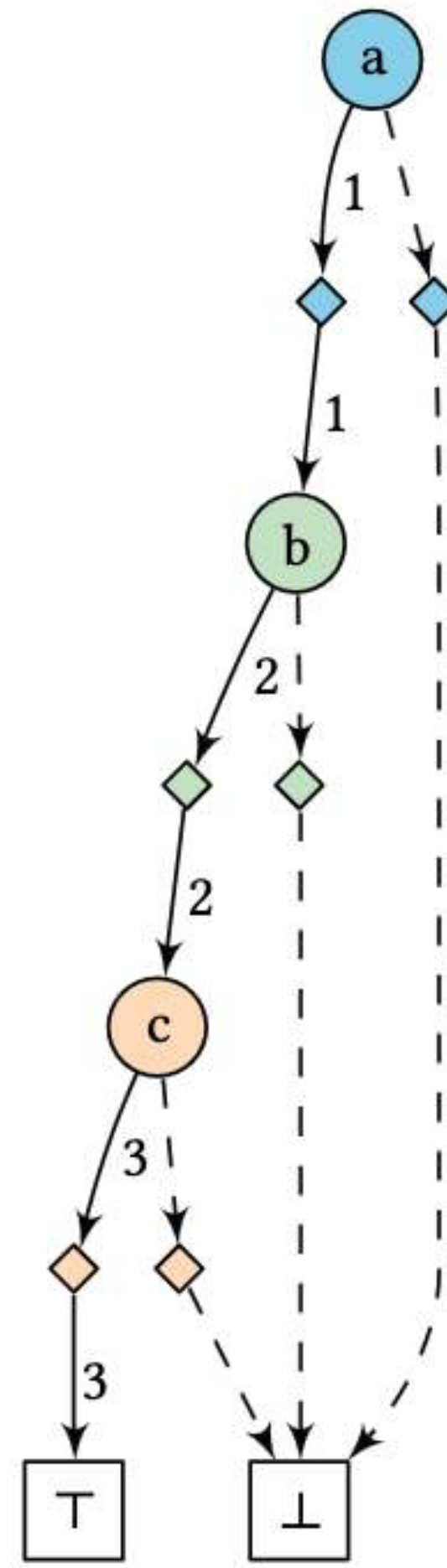
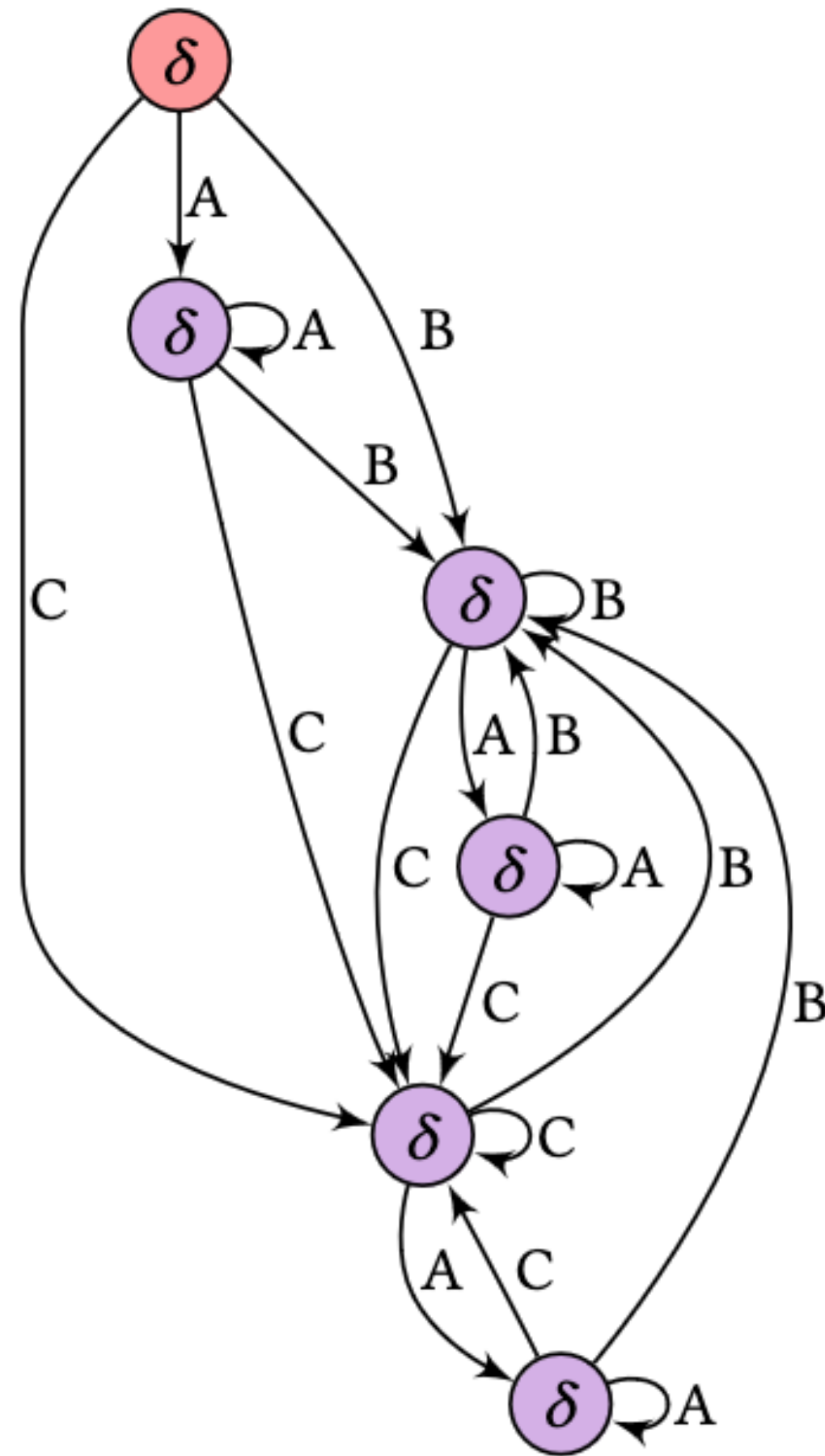


$$q \triangleq f=1 \cdot g \leftarrow 0$$

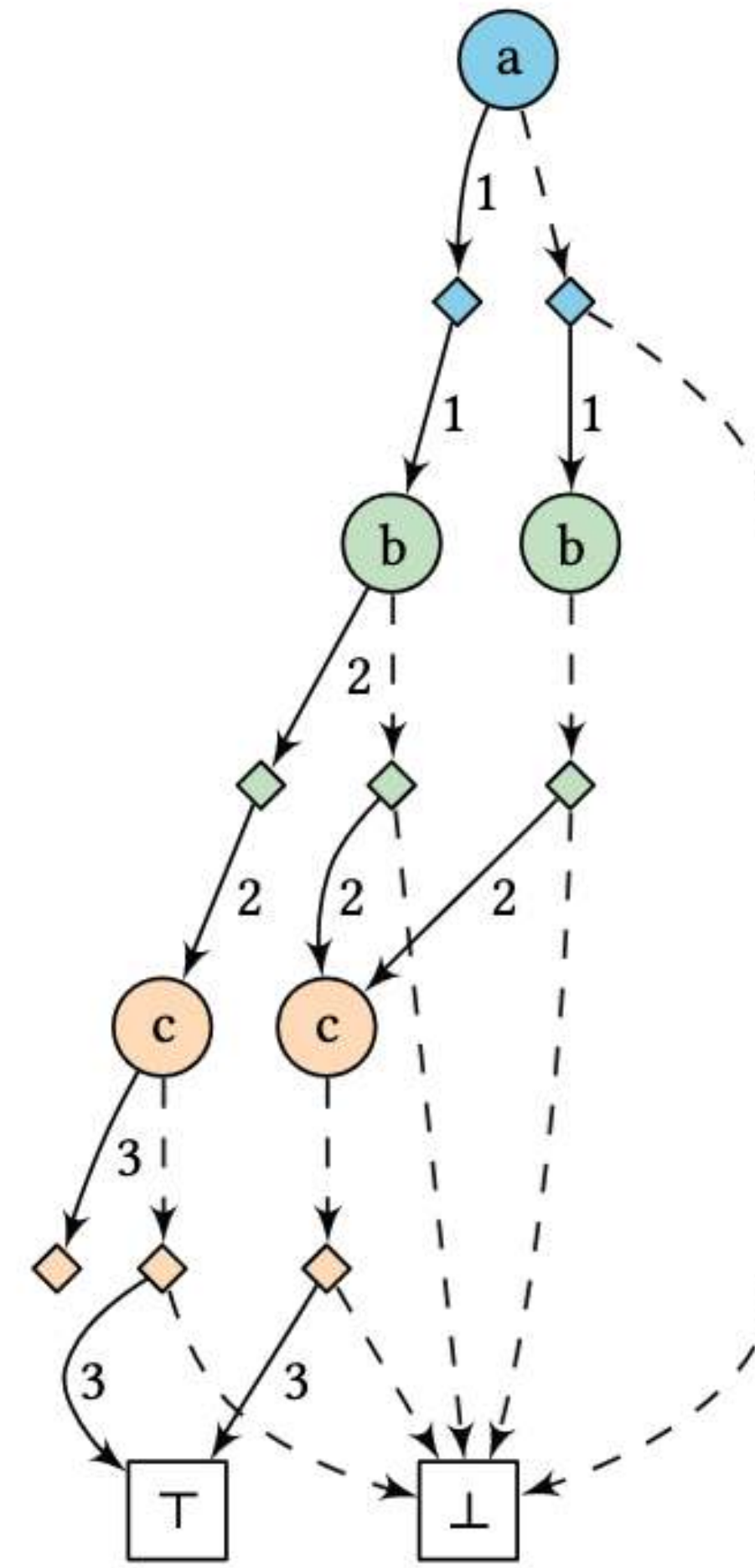


$$(p \oplus q)^{\star}$$

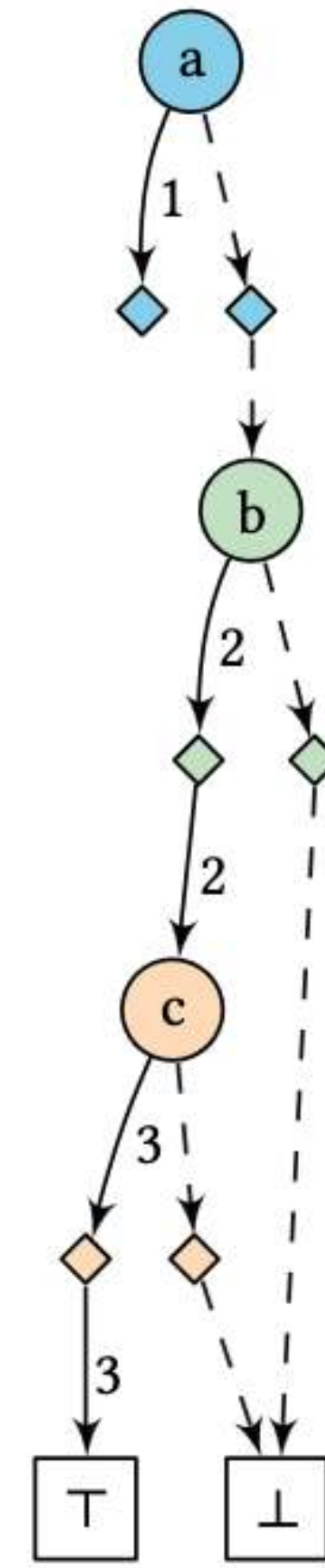
Symbolic Automaton Construction



A

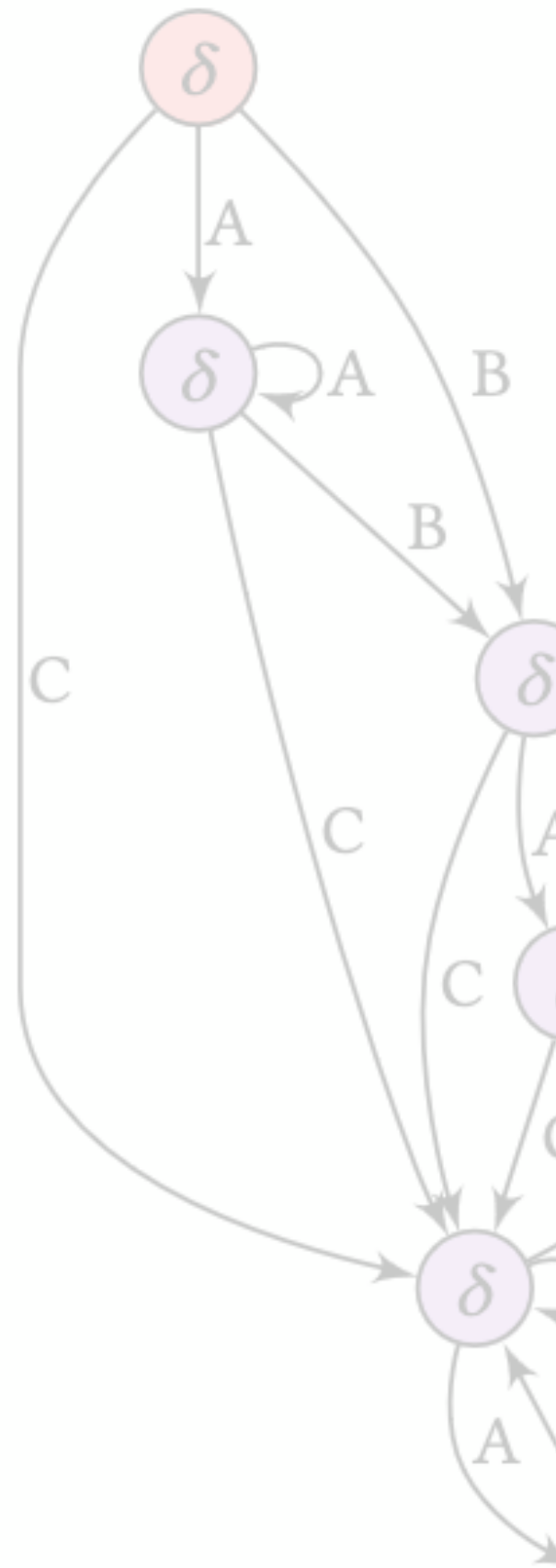


B



C

Symbolic Automaton Construction



Automata are also closed under
(using SPP operations!):

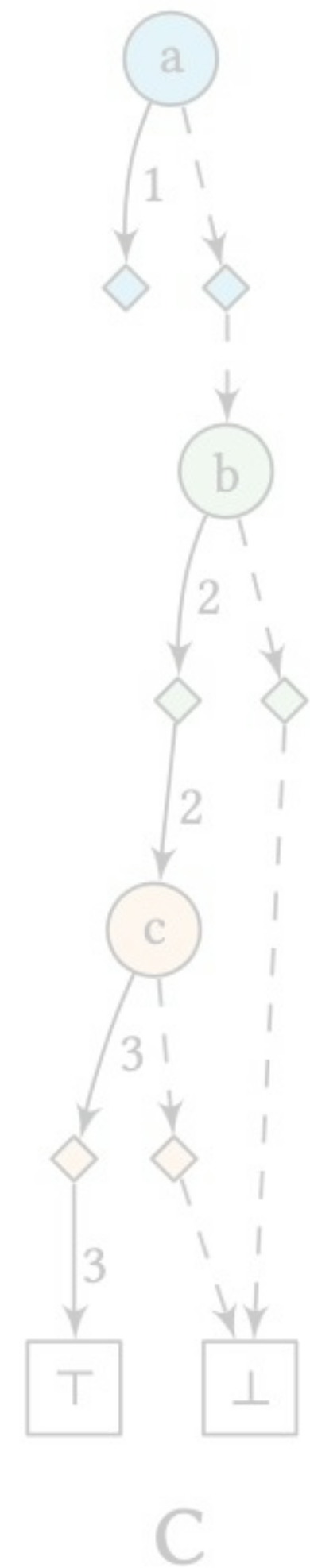
$+$, $\cdot$, $\star$

and

$\cap$, $-$, $\oplus$

Automata equivalence:

$$p \equiv q \iff p \oplus q \equiv \perp$$



NetKAT verification queries

All-pairs reachability queries, naively:

for $i, j \in 1..n$ do
 $sw = i \cdot \text{net}^\star \cdot sw = j \not\equiv \emptyset$

Number of queries
 $\propto n^2$

NetKAT verification queries

All-pairs reachability queries, naively:

for $i, j \in 1..n$ do
 $sw = i \cdot net^\star \cdot sw = j \not\equiv \emptyset$

Number of queries
 $\propto n^2$

... and using symbolic features:

for $i \in 1..n$ do
 check (forward ($sw = i \cdot net^\star$)) $\equiv (sw \in 1..n)$

Number of queries
 $\propto n$

NetKAT verification queries

All-pairs reachability queries, naively:

```
for  $i, j \in 1..n$  do  
   $sw = i \cdot net^\star \cdot sw = j \not\equiv \emptyset$ 
```

Number of queries
 $\propto n^2$

... and using symbolic features:

```
for  $i \in 1..n$  do  
  check (forward ( $sw = i \cdot net^\star$ ))  $\equiv (sw \in 1..n)$ 
```

Compute output symbolic packet

Number of queries
 $\propto n$

NetKAT verification queries

All-pairs reachability queries, naively:

for $i, j \in 1..n$ do
 $sw = i \cdot net^* \cdot sw = j \not\equiv \emptyset$

Number of queries
 $\propto n^2$

... and using symbolic features:

for $i \in 1..n$ do

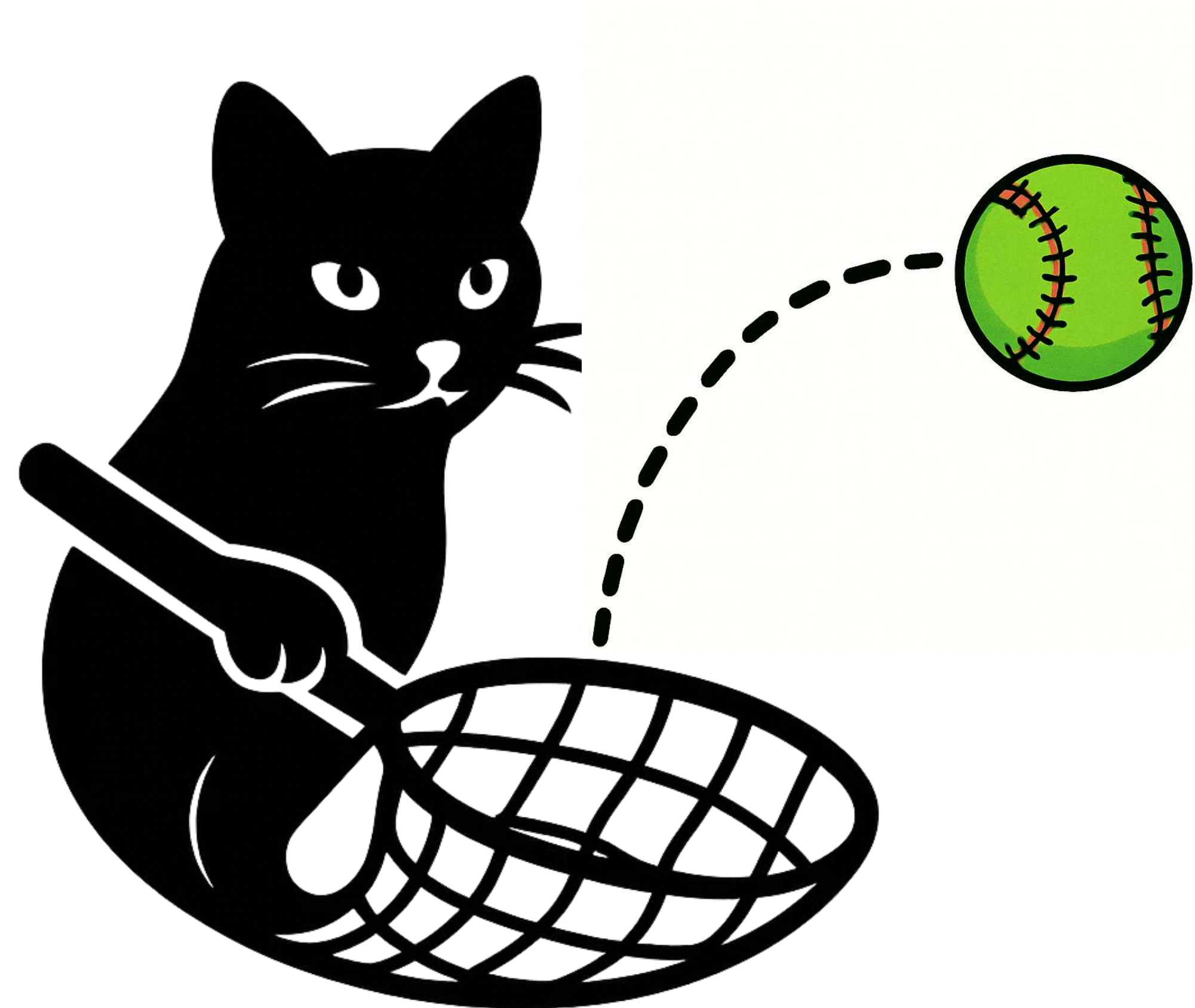
check $(\text{forward}(sw=i \cdot net^*)) \equiv (sw \in 1..n)$

“Any switch” symbolic packet

Compute output symbolic packet

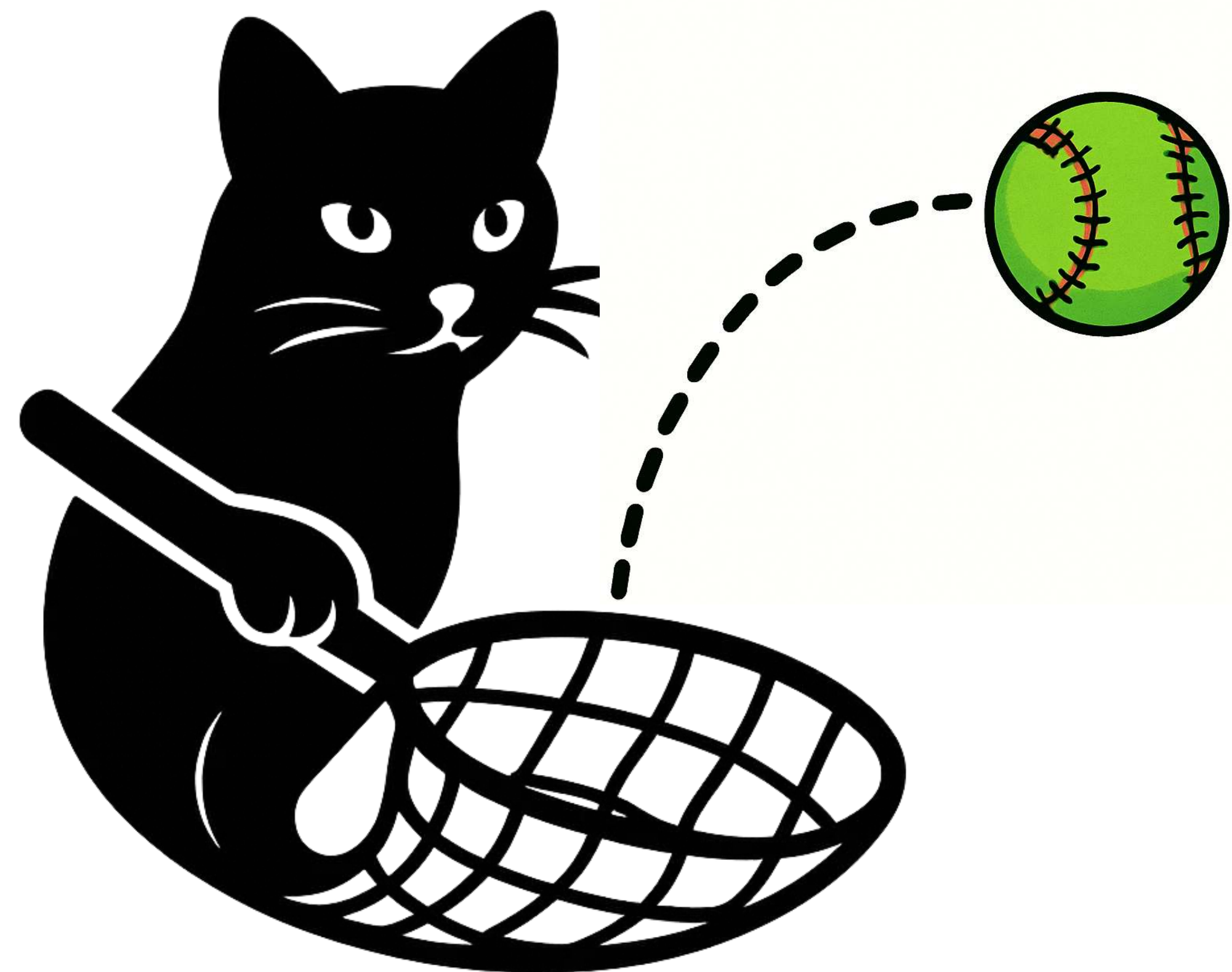
Number of queries
 $\propto n$

10 years of NetKAT — what did we learn?



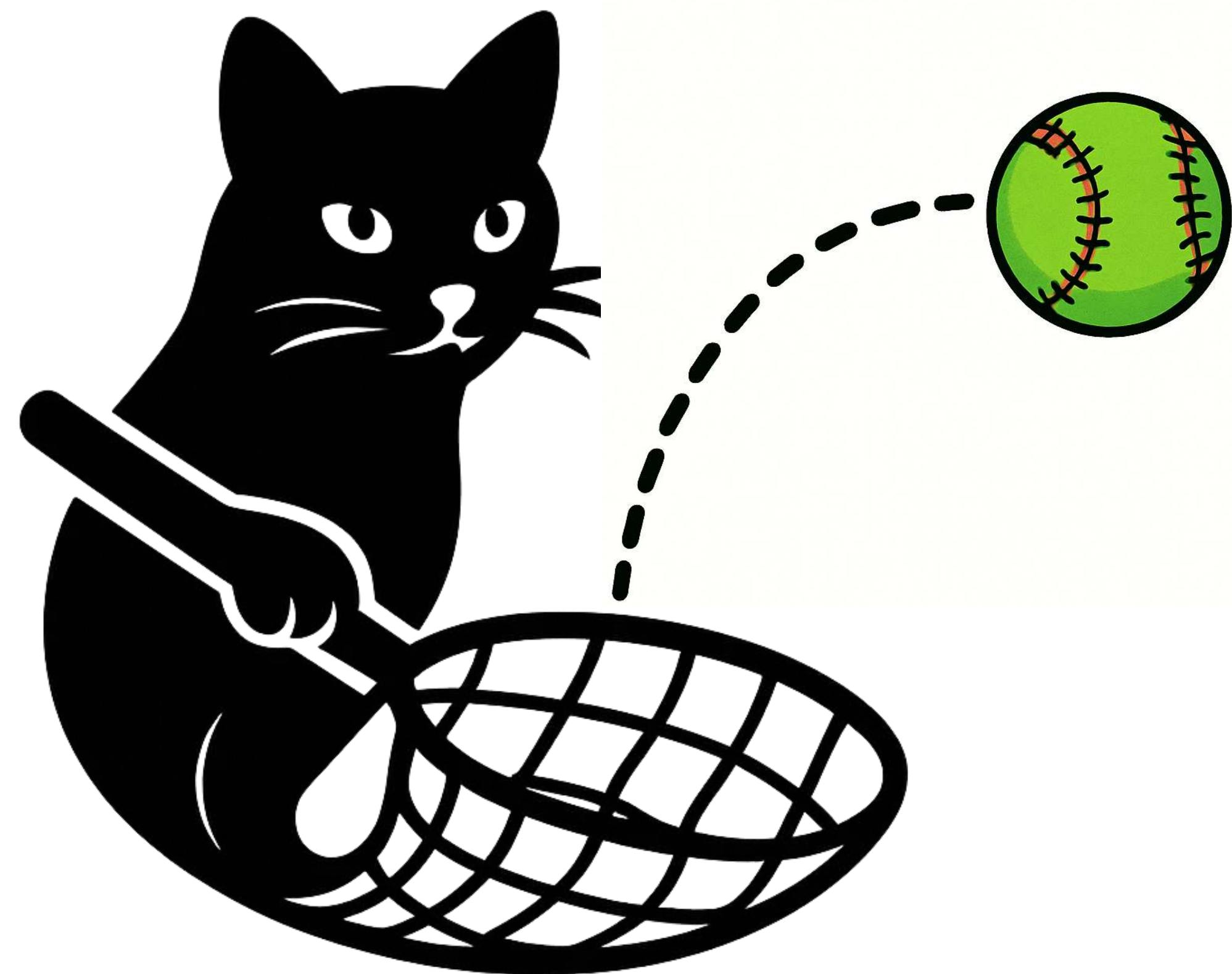
10 years of NetKAT — what did we learn?

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems



10 years of NetKAT — what did we learn?

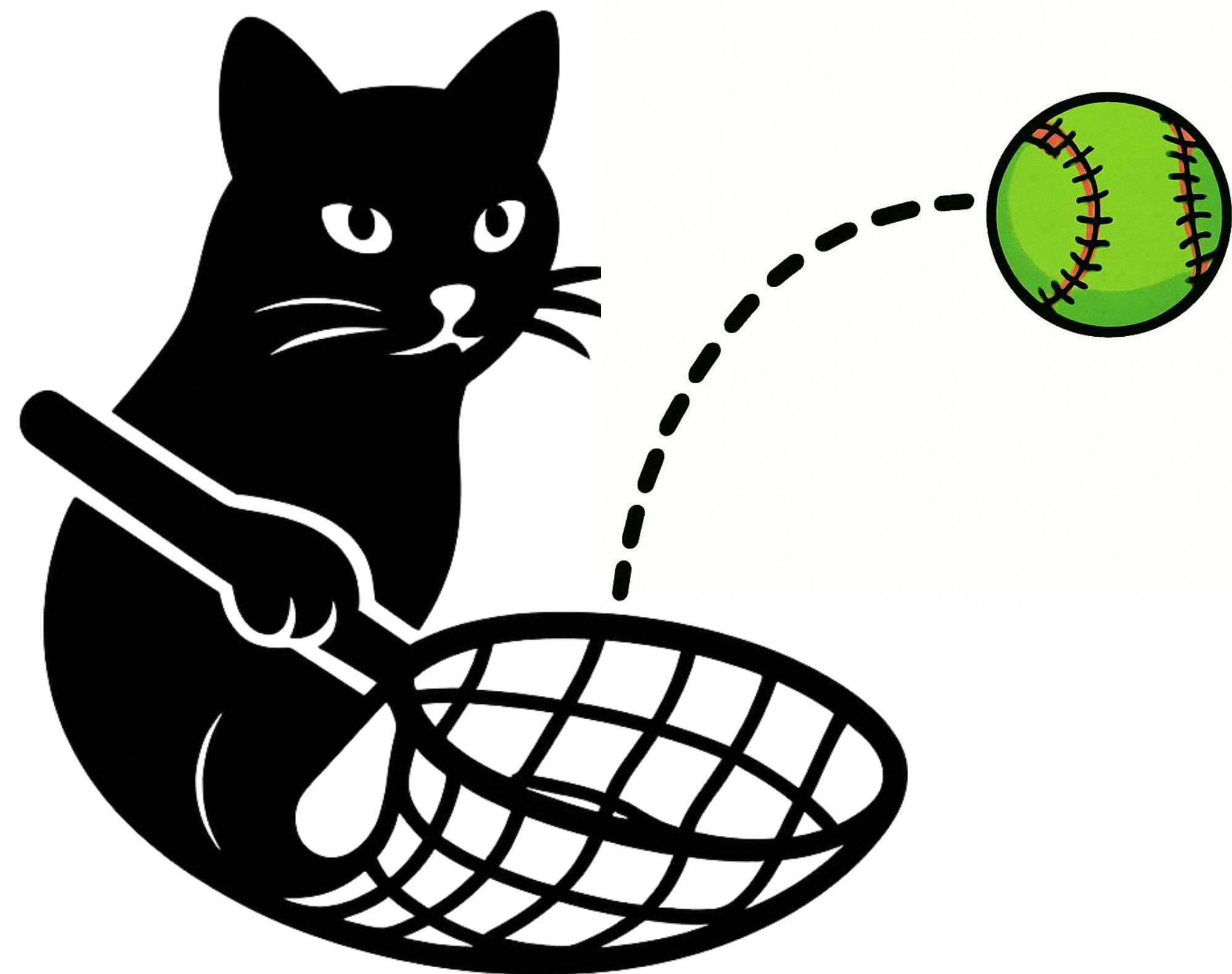
Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems



Compositional languages (syntax and semantics) offer a good foundation for expressive modelling and verification

10 years of NetKAT — what did we learn?

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems

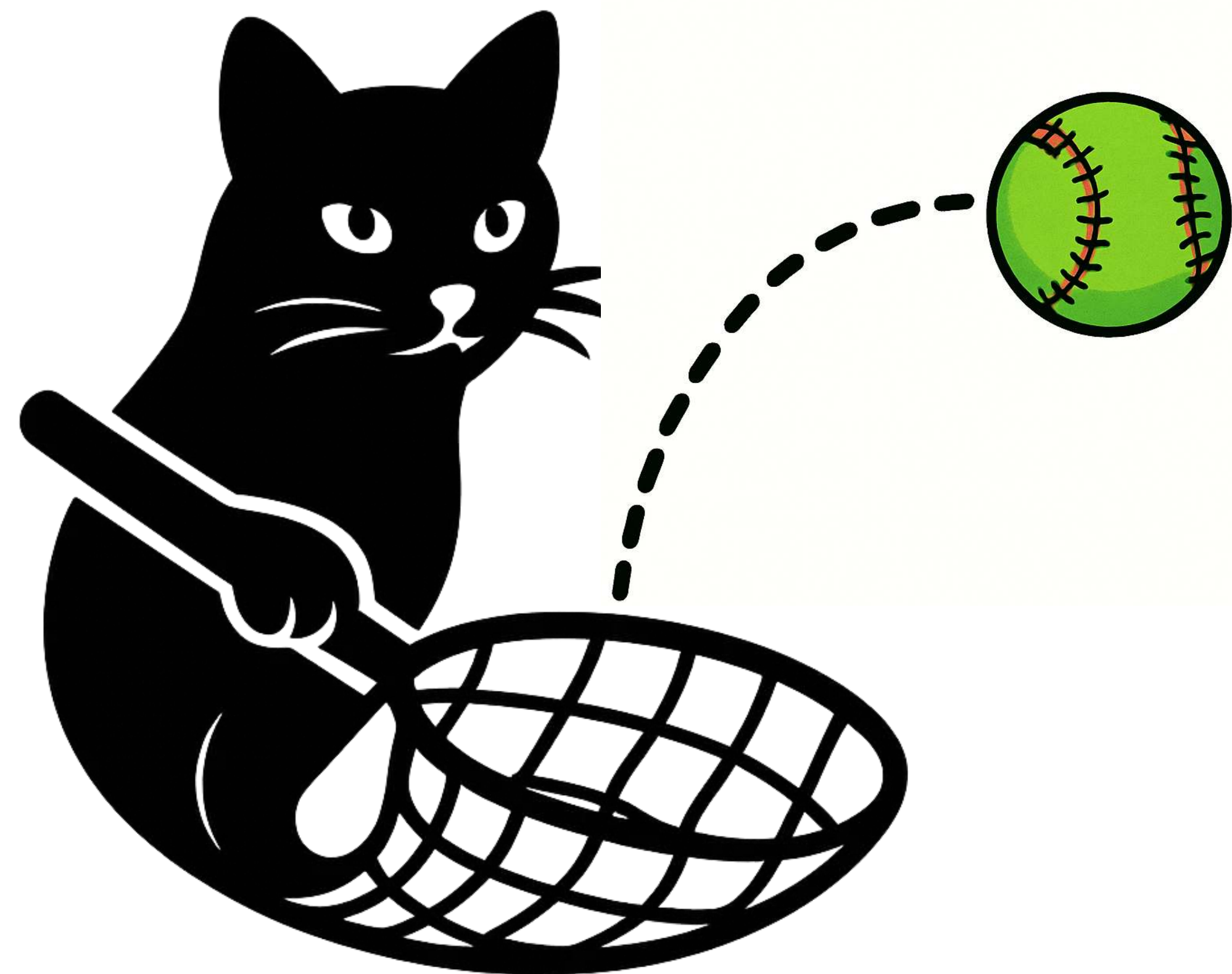


Compositional languages (syntax and semantics) offer a good foundation for expressive modelling and verification

Automata-based procedures offer a good foundation for scalable verification

10 years of NetKAT — what did we learn?

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems



Compositional languages (syntax and semantics) offer a good foundation for expressive modelling and verification

Automata-based procedures offer a good foundation for scalable verification

(It's a lot of fun!)

So much more to explore...

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with **fixed** number of bits

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with fixed number of bits

- ✗ No variable-length data
- ✗ Doesn't model packet parsing
- ✗ Cannot handle tunneling
- ✗ Cannot handle SR / MPLS
- ✗ Cannot model unbounded state
- ✗ Limited to finite-state verification

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with fixed number of bits

- ✗ No variable-length data
- ✗ Doesn't model packet parsing
- ✗ Cannot handle tunneling
- ✗ Cannot handle SR / MPLS
- ✗ Cannot model unbounded state
- ✗ Limited to finite-state verification

We need to model infinite state!

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with fixed number of bits

- ✗ No variable-length data
- ✗ Doesn't model packet parsing
- ✗ Cannot handle tunneling
- ✗ Cannot handle SR / MPLS
- ✗ Cannot model unbounded state
- ✗ Limited to finite-state verification

We need to model infinite state!

StackAT extends NetKAT with a stack to handle variable-length payload data

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with fixed number of bits

- ✗ No variable-length data
- ✗ Doesn't model packet parsing
- ✗ Cannot handle tunneling
- ✗ Cannot handle SR / MPLS
- ✗ Cannot model unbounded state
- ✗ Limited to finite-state verification

We need to model infinite state!

StackAT extends NetKAT with a stack to handle variable-length payload data

```
p, q ::=  
| skip  
...  
| push(n) | pop(n)
```

StackAT — Infinite-State Network Verification (PLDI'25)

Fixed number of fields with fixed number of bits

- ✗ No variable-length data
- ✗ Doesn't model packet parsing
- ✗ Cannot handle tunneling
- ✗ Cannot handle SR / MPLS
- ✗ Cannot model unbounded state
- ✗ Limited to finite-state verification

We need to model infinite state!

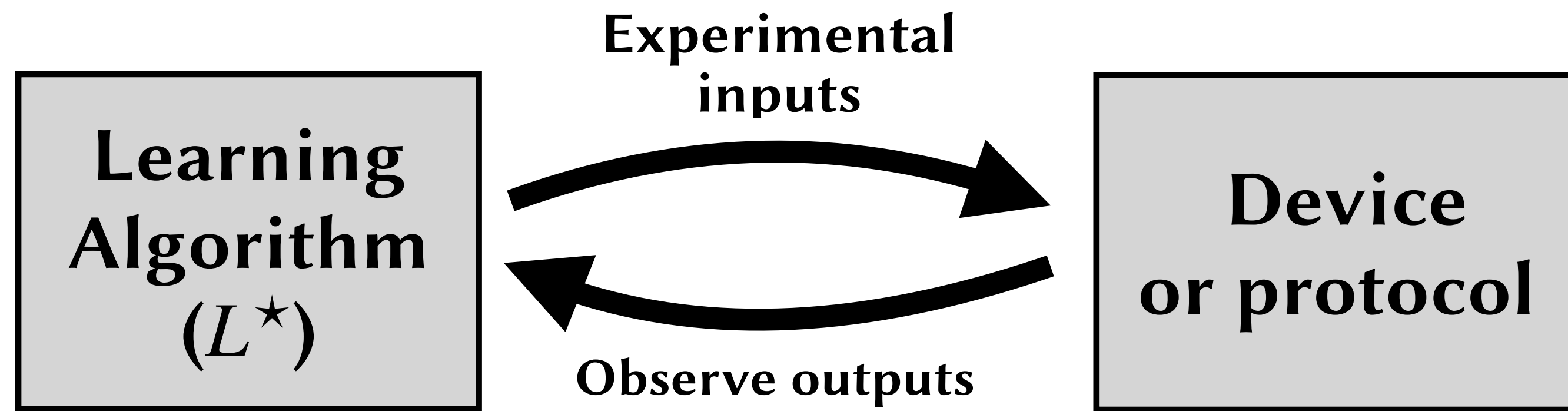
StackAT extends NetKAT with a stack to handle variable-length payload data

While retaining decidable program equivalence!

```
p, q ::=  
| skip  
...  
| push(n) | pop(n)
```

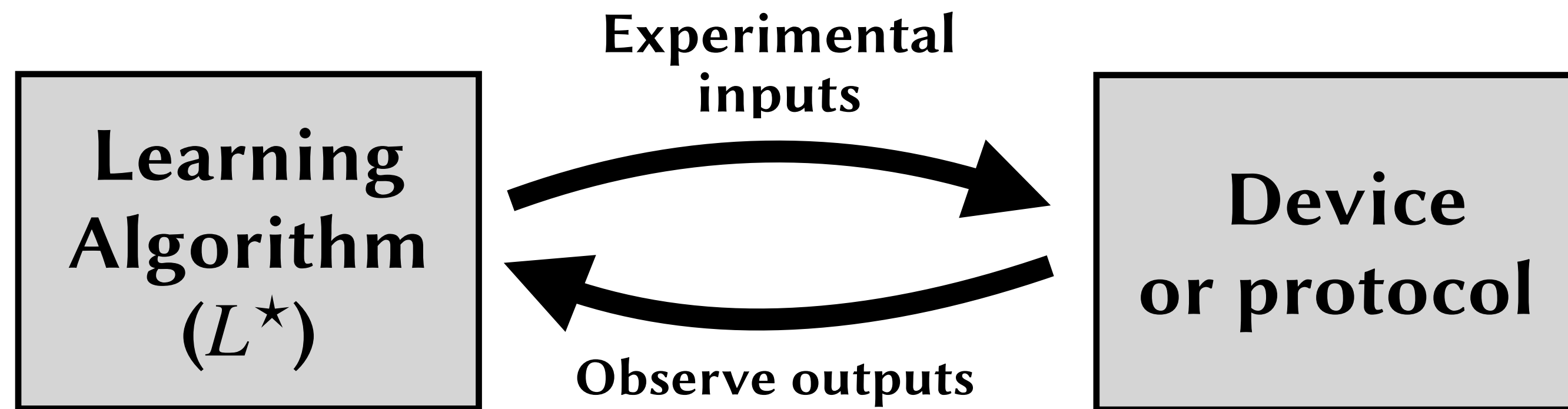
LearnKAT – Closed-Box Network Verification (PLDI'25)

What if we do not have access to configurations?



LearnKAT — Closed-Box Network Verification (PLDI'25)

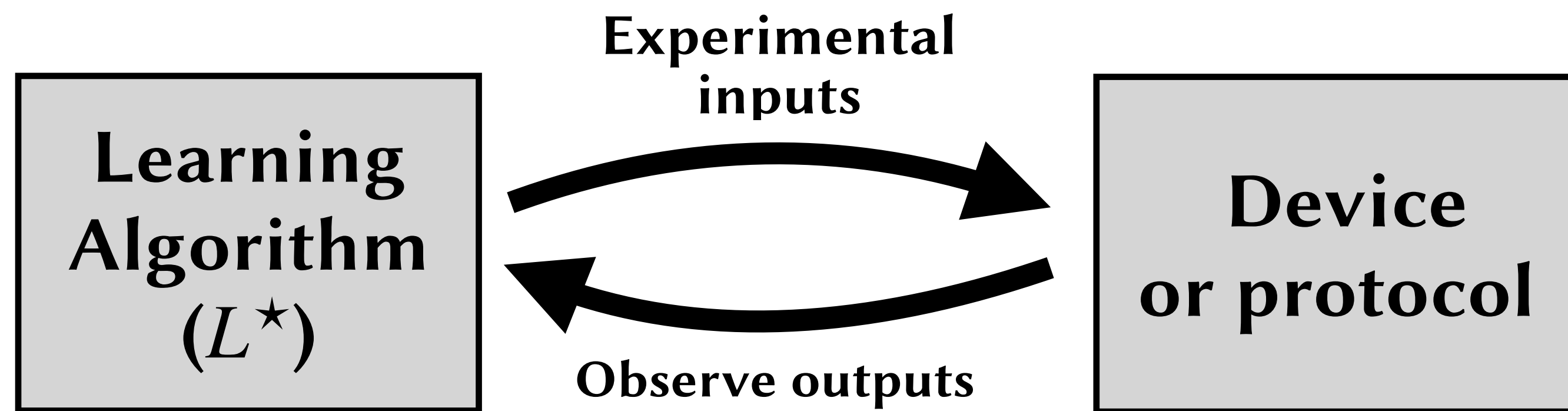
What if we do not have access to configurations?



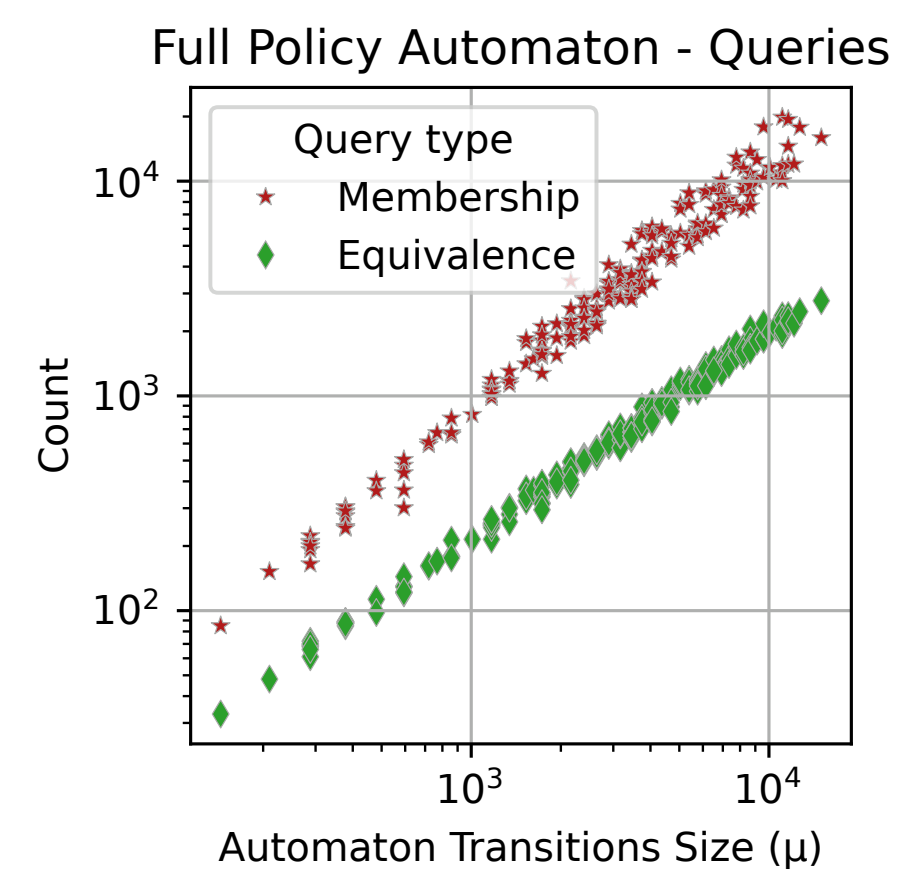
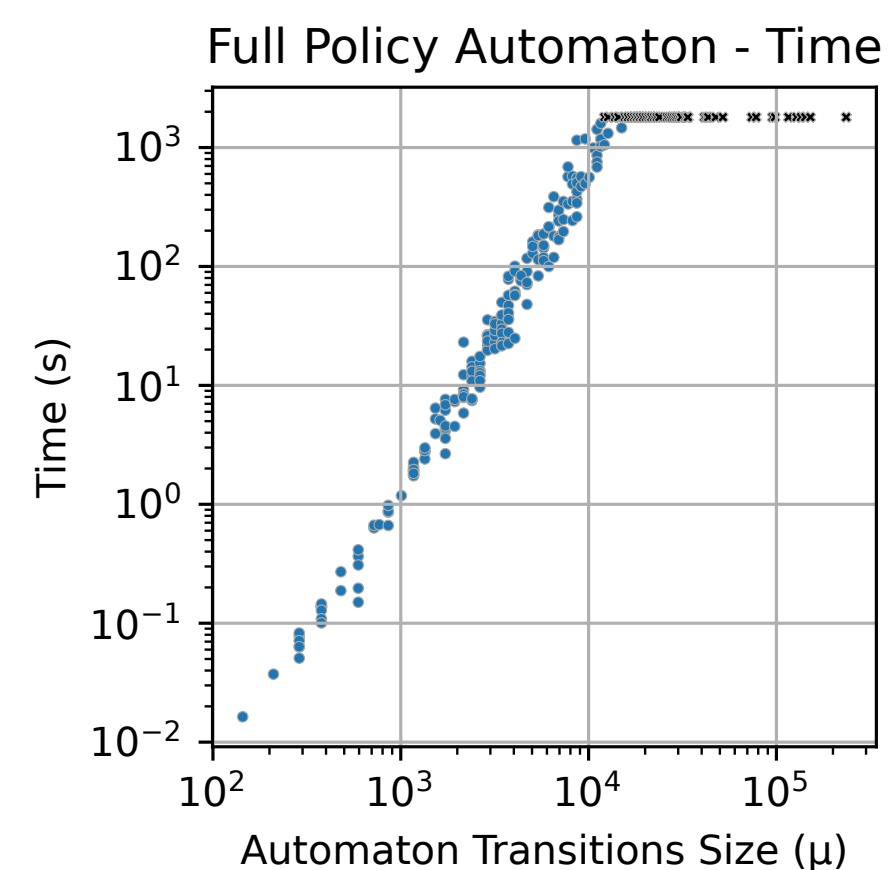
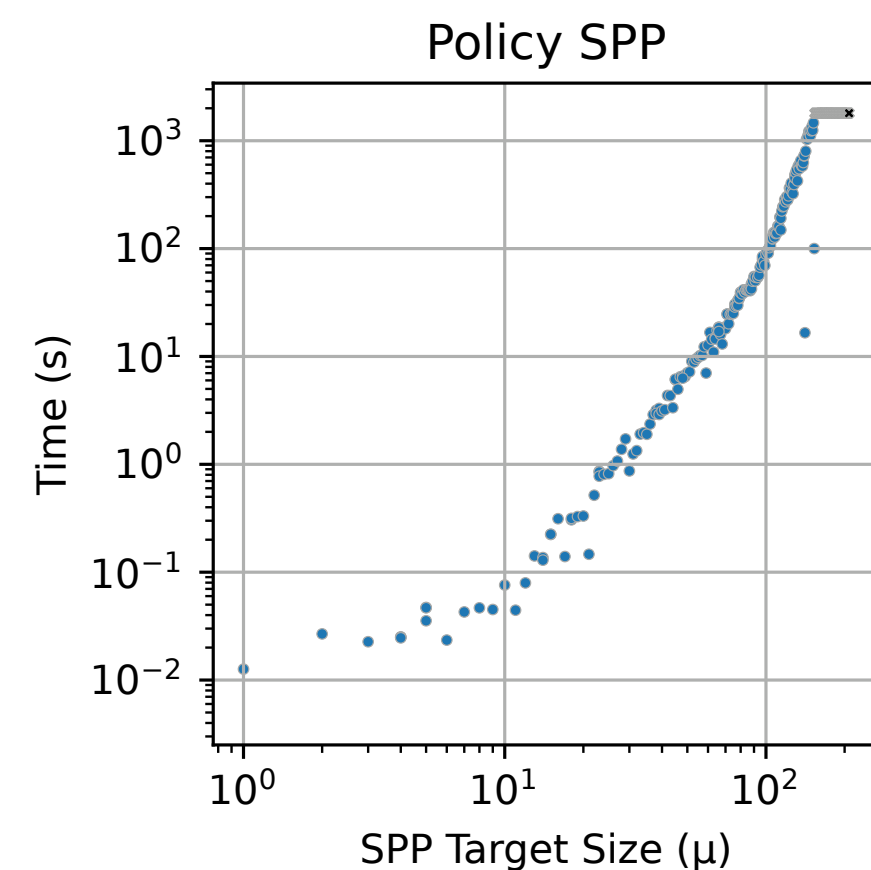
LearnKAT infers a symbolic NetKAT model of the network policy (under suitable abstractions)

LearnKAT – Closed-Box Network Verification (PLDI'25)

What if we do not have access to configurations?



LearnKAT infers a symbolic NetKAT model of the network policy (under suitable abstractions)



Quantitative verification (upcoming)

Do all incoming packets reach their destination with high security while not revealing their contents to link providers A, B, and C (e.g. out of providers A,B,C,D,E,F,...)?

Do all incoming packets reach their destination with at most 50ms latency?

Do all incoming packets reach their destination with at most 50ms (historical) latency while guaranteeing at least 10mbps bandwidth?

Quantitative verification (upcoming)

Do all incoming packets reach their destination with high security while not revealing their contents to link providers A, B, and C (e.g. out of providers A,B,C,D,E,F,...)?

Do all incoming packets reach their destination with at most 50ms latency?

Do all incoming packets reach their destination with at most 50ms (historical) latency while guaranteeing at least 10mbps bandwidth?

Weighted NetKAT

and

Symbolic Weighted NetKAT automata

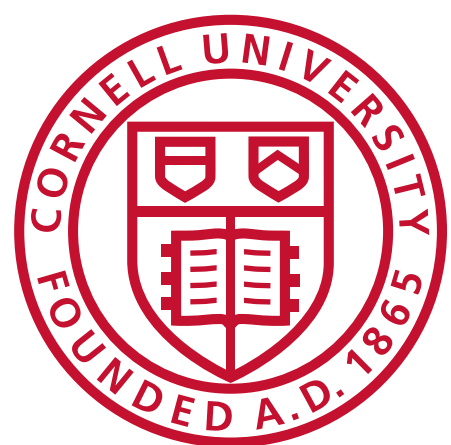
```
p, q ::=  
| skip  
  
...  
| p + q  
| w ⊙ p  
| p*
```

A team effort

<https://netkat.org/>

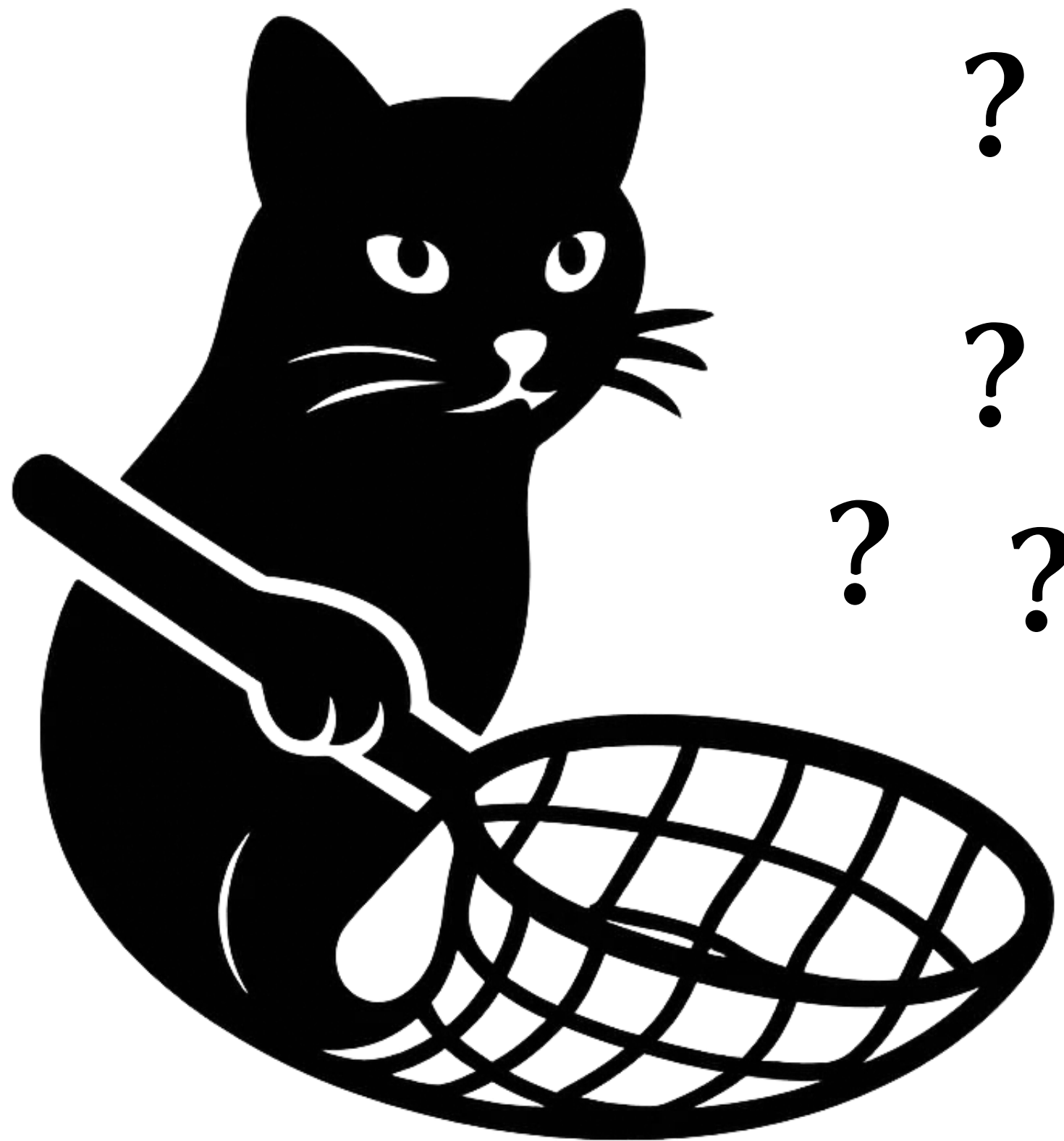


...



10 years of NetKAT — what did we learn?

Domain-specific languages *based on algebraic foundations* have a key role to play in formal verification of systems



Compositional languages (syntax and semantics) *offer a good foundation for expressive modelling and verification*

Automata-based procedures *offer a good foundation for scalable verification*

(So much more fun to come!)